Оглавление

Благодарности

Мы бы не смогли написать эту книгу без поддержки и технической помощи многих людей. Вот кого мы хотим поблагодарить особо.

Джеффри благодарит жену Кристин и сына Эйдана за безграничную любовь и поддержку.

Кристоф никогда бы не осилил пятое издание этой книги без любви и содействия своей жены Флоранс, а также неиссякаемого любопытства дочери Сели. Его кошки Канэль и Нуга также внесли свой вклад (уютным мурлыканьем). Теперь, когда книга закончена, Кристофу нечем будет оправдывать недостаток внимания к ним!

Чтобы написать такую книгу, мало личных усилий авторов. Джеффри и Кристоф выражают глубокую признательность сотрудникам Microsoft за неоценимую помощь. В частности, они благодарят Аруна Кишана (Arun Kishan) за оперативные ответы на заковыристые вопросы и поиск в команде создателей Windows людей, способных разъяснить тот или иной вопрос. Особой благодарности также заслуживают Киншуман Кинпгуман (Kinshu-man Kinshumann), Стефан Долл (Stephan Doll), Уэдсон Алмейда Фило (Wedson Almeida Filho), Эрик Ли (Eric Li), Жан-Ив Публан Qean-Yves Poub-lan), Сандип Ранад (Sandeep Ranade), Алан Чан (Alan Chan), Эл Контенти (Ale Contenti), Канн Су Гэтлин (Kang Su Gatlin), Кай Су (Kai Hsu), Мехмет Лайган (Mehmet Iyigun), Кен Юнь (Ken Jung), Павел Лебединский (Pavel Lebedynskiy), Пол Сливович (Paul Sliwowicz) и Лэнди Вонг (Landy Wang). Благодарим также всех, отвечавших на вопросы, опубликованные на внутренних форумах Майрософт, — Раймонда Чена (Raymond Chen), Сангука Чу (Sunggook Chue), Криса Корио (Chris Corio), Лари Остермана (Larry Osterman), Ричарда Рассела (Richard RusseU), Марка Руссиновича (Mark Russinovich), Майка Шелдона (Mike Sheldon), Дэмиена Уоткинса (Damien Watkins) и Юньфень Чжань (Junfeng Zhang). Также выражаем искреннюю благодарность Джону Роббинсу по прозвищу «Убийца багов» (John «Bugs-layer» Robbins) и Кенни Керру (Kenny Kerr), чьи отзывы были большим подспорьем в работе над этой книгой.

Авторы признательны рецензенту издательства Бену Райану (Ben Ryan), поверившему в такого ненормального француза как Кристоф, менеджерам Линн Финел (Lynn Finnel) и Кертису Филипсу (Curtis Philips) — за терпение, Скотту Сили (Scott Seely) — за стремление к точности в технических вопросах, Роже Леблану (Roger LeBlanc) — за виртуозный перевод «франко-английского» наречия Кристофа на понятный язык, Андреа Фокс (Andrea Fox) — за тщательную корректуру. Помимо команды из Редмонда, много личного времени отдала работе над книгой Джоанта Сен (Joyanta Sen).

Наконец, Кристоф искренне благодарит Джеффри Рихтера за то, что Джефф доверил ему работу над пятым изданием книги; Джеффри же благодарен Кристофу за неутомимость в поиске информации, а также за многократную перестройку и переработку материала в стремлении к идеалу (такому, каким его видел Джефф).

Введение

Microsoft Windows — сложная операционная система. Она включает в себя столько всего и делает так много, что одному человеку просто не под силу полностью разобраться в этой системе. Более того, из-за такой сложности и комплексности Windows трудно решить, с чего начать ее изучение. Лично я всегда начинаю с самого низкого уровня, стремясь получить четкое представление о базовых сервисах операционной системы. Разобравшись в основах, дальше двигаться проще. С этого момента я шаг за шагом, по мере необходимости, изучаю сервисы более высокого уровня, построенные именно на этом базисе. Поэтому основное внимание в этой книге уделяется фундаментальным концепциям, которые должны знать те, кому придется заниматься проектированием и реализацией программ для операционной системы Windows. Вкратце, читатели познакомятся с различными возможностями Windows и научатся использовать их в своих программах на языках C и C++.

Вопросы, относящиеся к компонентной модели объектов (Component Object Model, COM), в моей книге прямо не затрагиваются. Но COM — это архитектура, где используются процессы, потоки, механизмы управления памятью, DLL, локальная память потоков, Unicode и многое другое. Если вы знаете, как устроены и работают эти фундаментальные сервисы операционной системы, то для освоения COM достаточно понять, как они применяются в этой архитектуре. Мне очень жаль тех, кто пытается перепрыгнуть через все это и сразу же взяться за изучение архитектуры COM. Впереди у них долгий и тернистый путь; в их знаниях неизбежны пробелы, которые непременно будут мешать им в работе.

В этой книге также не рассматривается общеязыковая исполняющая среда (common language runtime, CLR) платформы .NET Framework от Майкрософт (ей посвящена отдельная книга: CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#, Русская редакция, Питер, 2006-2007). Однако CLR реализована в виде объекта COM внутри библиотеки DLL, которая загружается в процесс и использует потоки для манипулирования в памяти строками в формате Unicode. Так что полученные из этой книги знания о базовых «строительных блоках» помогут и при написании управляемого кода. Кроме того, поддерживаемый CLR механизм Platform Invocation (P/Invoke) позволяет вызывать различные API Windows, о которых много говорится в этой книге.

И вот тут мы подходим к тому, о чем же моя книга. А она — о строительных кирпичиках Windows, базовых сервисах, в которых (по крайней мере, на мой взгляд) должен досконально разбираться каждый разработчик Windows-приложений. Рассматривая тот или иной сервис, я буду рассказывать, как им пользуется система и как им должно пользоваться ваше приложение. Во многих главах я буду показывать, как на основе базовых сервисов Windows создавать собственные строительные кирпичики. Реализуя их в виде универсальных функций и классов C++ и комбинируя в них те или иные базовые сервисы Windows, вы получите нечто большее суммы отдельных частей.

64-разрядные версии Windows

В течение долгого времени Майкрософт поставляла 32-разрядные версии Windows для компьютеров с процессорами архитектуры xS6. Сейчас Майкрософт поставляет 64-разрядные версии операционные системы Windows, поддерживающие процессоры с архитектурой x6A и IA64. Компьютеры на основе 64-разрядных процессоров быстро завоевывают популярность. Ожидается, что в самом близком будущем такими процессорами будут оснащены все рабочие станции и серверы. Поэтому Майкрософт заявляет, что Windows Server 2008 станет последней 32-разрядной версией Windows! Стало быть, и разработчики должны писать свои программы так, чтобы они корректно работали с 64-разрядными версиями Windows. Многое из того, что нужно знать для создания программ, совместимых с 64-разрядной (и 32-разрядной) Windows вы найдете в этой книге.

Основное преимущество 64-разрядного адресного пространства — возможность легко манипулировать большими объемами данных, поскольку размер доступного адресного пространства для 64-разрядной системы не ограничен 2 Гб. Даже если вашему приложению столько не нужно, увеличение адресного пространства (до 8 Тб), несомненно, пойдет на пользу Windows, которая станет работать быстрее.

Вот что, вкратце, следует знать о 64-разрядных версиях Windows:

- Ядро 64-разрядной Windows — это ядро 32-разрядной Windows, портированное на 64-разрядную платформу. Это означает, что все механизмы и хитрости программирования, знакомые вам по 32-разрядной версии, будут работать и в 64-разрядном «мире». В действительности, Майкрософт компилирует 32- и 64-разрядные версии из одного и того же исходного кода. Иными словами, у этих версией единая база кода, и все новые возможности и исправления появляются в двух версиях одновременно.
- Поскольку обе платформы используют одного и то же ядро и одинаковые базовые концепции, Windows API также идентичен на обеих платформах. Следовательно, вам не придется заново проектировать и переписывать свои приложения, чтобы заставить их работать в 64-разрядной Windows — достаточного внести в код небольшие изменения и перекомпилировать его.

- Для сохранения преемственной совместимости 64-разрядная Windows поддерживает исполнение 32-разрядных приложений, однако их производительность на 64-разрядной платформе увеличится после перекомпиляции для этой платформы.
- Поскольку перенос 32-разрядного кода в 64-разрядную Windows оказался несложным делом, уже доступно немало драйверов устройств, утилит и приложений для 64-разрядной версии. Visual Studio — это «родное» 32-разрядное приложение, но Майкрософт, похоже, спешит сделать из него 64-разрядное приложение. К счастью, и 32-разрядная Visual Studio неплохо работает в 64-разрядной Windows, просто для нее действует то же ограничение на размер доступного адресного пространства, что и для всех 32-разрядных программ. Еще один плюс Visual Studio в том, что она позволяет отлаживать 64-разрядные приложения.
- В 64-разрядной Windows не так уж много нового. Вас должно обрадовать, что многие типы данных, включая int, DWORD, LONG, BOOL и ряд других, останутся 32-разрядным. Беспокоиться об указателях и описателях тоже не придется, поскольку теперь все они 64-разрядные.

Сведений о том, как подготовить исходный код к выполнению на 64-разрядной платформе, вполне хватает и на веб-сайте Майкрософт, так что я в эти детали вдаваться не буду. Но, что бы я ни писал в каждой главе, я все время помнил о 64-разрядной Windows и, где это было нужно, включал специфическую для нее информацию. Кроме того, все приведенные в этой книге программы-примеры я компилировал с использованием 64-разрядного компилятора. Если вы будете следовать тем же правилам, что и я, вам не составит труда создать единую базу исходного кода своих приложений для 32- и 64-разрядной Windows.

Что нового в пятом издании

Традиционно названия новых изданий этой книги отличались от предыдущих: *Advanced Windows NT*, *Advanced Windows* и *Programming Applications for Microsoft Windows*… И эта книга, получив название *Windows via C/C++*, не стала исключением. Из названия ясно, что она предназначена для программистов на языках C и C++, желающих разобраться в Windows. В новом издании описано более 170 новых функций, появившихся в Windows XP, Windows Vista и Windows Server 2008.

Некоторые главы полностью переписаны, например глава 11, посвященная работе с новым API пула потоков. Остальные главы были существенно доработаны и дополнены новыми сведениями. Так, в главу 4 добавлено описание механизма User Account Control, а в главу 8 — описание новых синхронизирующих механизмов (таких как Interlocked Singly-Linked List, Slim Reader-Writer Lock и условные переменные).

Намного детальнее я рассказываю и о том, как взаимодействует с системой библиотека C/C++ (C/C++ run-time library) — особенно при реализа-

ции защитных механизмов и обработке исключений. Наконец, я добавил две совершенно новые главы: о механизмах ввода-вывода и о работе новой системы Windows Error Reporting, изменившей подходы к созданию отчетов об ошибках и восстановлению приложений после сбоев.

Помимо этих изменений, в книге появилась целая тонна нового содержания. Упомяну лишь самое главное.

- **Новшества Windows Vista и Windows Server 2008.** Книгу было бы нельзя считать действительно переработанной, не будь в ней отражены новшества Windows XP, Windows Vista, Windows Server 2008 и библиотеки C/C++: безопасные строковые функции, изменения объектов ядра (дескрипторы границ и пространства имен), списки атрибутов потоков и процессов, планирование операций ввода-вывода, отмена синхронного ввода-вывода, векторная обработка исключений и многое другое.
- **Поддержка 64-разрядной Windows**. В книге приводится информация, специфическая для 64-разрядной Windows; все программы-примеры построены с учетом специфики этой версии Windows и протестированы в ней.
- **Применение C++**. По требованию читателей примеры написаны на C++. В итоге они стали компактнее и легче для понимания.
- **Повторно используемый код**. Я старался писать по возможности универсальный код, пригодный для многократного использования. Это позволит вам брать из него отдельные функции или целые C++-классы без изменений (незначительная модификация может понадобиться лишь в отдельных случаях). Код на C++ гораздо проще для повторного использования.
- **Утилита ProcessInfo** теперь отображает владельца процесса, командную строку и сведения об UAC.
- **Утилита LockCop** — новинка этого издания. Она отображает работающие в системе процессы, а при выборе процесса — список его потоков с указанием синхронизирующего механизма, используемого каждым потоком. При этом утилита определяет и показывает взаимные блокировки
- **Перехват API-вызовов**. Я представлю вам несколько C++-классов, которые сделают перехват API-вызовов в одном или всех модулях процесса тривиальной задачей. Вы сможете перехватывать даже вызовы Load-Library и GetProcAddress от библиотеки C/C++!
- **Более подробные сведения о структурной обработке исключений**. Эту часть я тоже переписал и во многом перестроил. Вы найдете здесь больше информации о необрабатываемых исключениях и познакомитесь с настройкой отчетов, генерируемых службой Windows Error Reporting, под свои потребности.

Примеры кода и требования к системе

Примеры кода для этой книги можно загрузить с веб-сайта поддержки по адресу:

http://www.wintellect.com/books.aspx

Для компиляции примеров потребуется Visual Studio 2005 или выше, Microsoft Platform SDK для Windows Vista и Windows Server 2008 (поставляется с некоторыми версиями Visual Studio). Для запуска приложений необходим компьютер (или виртуальная машина) под управлением Windows Vista.

Техническая поддержка

Я и мои редакторы сделали все возможное, чтобы в этой книге и примерах было как можно меньше ошибок. Исправления замеченных неточностей будут публиковаться на сайте:

http://www.wintellect.com/books.aspx

Microsoft Press публикует исправления на http://mspress.microsoft.com/support.

Если у вас есть какие-нибудь комментарии, вопросы или идеи, касающиеся моей книги, пожалуйста, направляйте их в Microsoft Press по обычной или электронной почте:

Microsoft Press
Attn: Windows via C/C++ Editor
One Microsoft Way
Redmond, WA 98052-6399
mspinput@microsoft.com

ЧАСТЬ I

МАТЕРИАЛЫ ДЛЯ ОБЯЗАТЕЛЬНОГО ЧТЕНИЯ

Оглавление

ГЛАВА 1

Обработка ошибок

Прежде чем изучать функции, предлагаемые Microsoft Windows, посмотрим, как в них устроена обработка ошибок.

Когда вы вызываете функцию Windows, она проверяет переданные ей параметры, а затем пытается выполнить свою работу. Если вы передали недопустимый параметр или если данную операцию нельзя выполнить по какой-то другой причине, она возвращает значение, свидетельствующее об ошибке. В таблице 1-1 показаны типы данных для возвращаемых значений большинства функций Windows.

Табл. 1-1. Стандартные типы значений, возвращаемых функциями Windows

Тип данных	Значение, свидетельствующее об ошибке
VOID	Функция всегда (или почти всегда) выполняется успешно. Таких функций в Windows очень мало
BOOL	Если вызов функции заканчивается неудачно, возвращается 0; в остальных случаях возвращаемое значение отлично от 0 (не пытайтесь проверять его на соответствие TRUE, лучше проверить его на соответствие FALSE)
HANDLE	Если вызов функции заканчивается неудачно, то обычно возвращается NULL; в остальных случаях HANDLE идентифицирует объект, которым Вы можете манипулировать. Будьте осторожны: некоторые функции возвращают HANDLE со значением INVALID_HANDLE_VALUE, равным -1. В документации Platform SDK для каждой функции четко указывается, что именно она возвращает при ошибке — NULL или INVALID_HANDLE_VALUE
PVOID	Если вызов функции заканчивается неудачно, возвращается NULL; в остальных случаях PVOID сообщает адрес блока данных в памяти

Табл. 1-1. *(окончание)*

Тип данных	Значение, свидетельствующее об ошибке
LONG	Это значение — «крепкий орешек». Функции, которые сообщают или DWORD значения каких-либо счетчиков, обычно возвращают LONG или DWORD. Если по какой-то причине функция не сумела сосчитать то, что вы хотели, она обычно возвращает 0 или -1 (все зависит от конкретной функции). Если вы используете одну из таких функций, проверьте по документации Platform SDK, каким именно значением она уведомляет об ошибке

При возникновении ошибки вы должны разобраться, почему вызов данной функции оказался неудачен. За каждой ошибкой закреплен свой код — 32-битное число.

Функция Windows, обнаружив ошибку, через механизм локальной памяти потока сопоставляет соответствующий код ошибки с вызывающим потоком. (Локальная память потока рассматривается в главе 21) Это позволяет потокам работать независимо друг от друга, не вмешиваясь в чужие ошибки. Когда функция вернет вам управление, ее возвращаемое значение будет указывать на то, что произошла какая-то ошибка. Какая именно — вы узнаете, вызвав функцию *GetLastError*.

```
DWORD GetLastError();
```

Она просто возвращает 32-битный код ошибки для данного потока.

Теперь, когда у вас есть код ошибки, вам нужно обменять его на что-нибудь более внятное. Список кодов ошибок, определенных Майкрософт, содержится в заголовочном файле WinError.h. Я приведу здесь его небольшую часть, чтобы вы представляли, на что он похож:

```
// MessageId: ERROR_SUCCESS
//
// MessageText:
//
// The operation completed successfully.
//
#define ERROR_SUCCESS                    0L

#define N0_ERR0R 0L                      // dderror
#define SEC_E_OK                         ((HRESULT)OxOOOOOOOOL)

//
// MessageId: ERROR_INVALID_FUNCTION
//
// MessageText:
//
// Incorrect function.
```

```
//
#define ERROR_INVALID_FUNCTION             1L      // dderror
//
// MessageId: ERROR_FILE_NOT_FOUND
//
// MessageText:
//
// The system cannot find the file specified.
//
#define ERROR_FILE_NOT_FOUND               2L
//
// MessageId: ERROR_PATH_NOT_FOUND
//
// MessageText:
//
// The system cannot find the path specified.
//
#define ERROR_PATH_NOT_FOUND               3L
//
// MessageId: ERROR_TOO_MANY_OPEN_FILES
//
// MessageText:
//
// The system cannot open the file.
//
#define ERROR_TOO_MANY_OPEN_FILES          4L
//
// MessageId: ERROR_ACCESS_DENIED
//
// MessageText:
//
// Access is denied.
//
#define ERROR_ACCESS_DENIED                5L
```

Как видите, с каждой ошибкой связаны идентификатор сообщения (его можно использовать в исходном коде для сравнения со значением, возвращаемым GetLastError), текст сообщения (описание ошибки на нормальном языке) и номер (вместо него лучше использовать идентификатор). Учтите, что я показал лишь крошечную часть файла WinError.h; на самом деле в нем более 39 000строк!

Функцию *GetLastError* нужно вызывать сразу же после неудачного вызова функции Windows, иначе код ошибки может быть потерян (перезаписан

кодом ошибки другой функции ИЛИ кодом ERROR_SUCCESS в случае успешного завершения функции).

Некоторые функции Windows всегда завершаются успешно, но по разным причинам. Например, попытка создать объект ядра «событие» с определенным именем может быть успешна либо потому, что вы действительно создали его, либо потому, что такой объект уже есть. Но иногда нужно знать причину успеха. Для возврата этой информации Майкрософт предпочла использовать механизм установки кода последней ошибки. Так что и при успешном выполнении некоторых функций вы можете вызывать *GetLastError* и получать дополнительную информацию. К числу таких функций относится, например, *CreateEvent*. Сведения о других функциях и примеры возврата ERROR_ALREADY_EXISTS в случае, если именованное событие существует, см. в Platform SDK.

На мой взгляд, особенно полезно отслеживать код последней ошибки в процессе отладки. Кстати, отладчик в Microsoft Visual Studio позволяет настраивать окно Watch так, чтобы оно всегда показывало код и описание последней ошибки в текущем потоке. Для этого надо выбрать какую-нибудь строку в окне Watch и ввести «@err,hr». Теперь посмотрите на рис. 1-1. Видите, я вызвал функцию *CreateFile*. Она вернула значение INVALID_HANDLE_VALUE (-1) типа HANDLE, свидетельствующее о том, что ей не удалось открыть заданный файл. Но окно Watch показывает нам код последней ошибки (который вернула бы функция *GetLastError*, если бы я ее вызвал), равный 0x00000002, и описание «The system cannot find the file specified» («Система не может найти указанный файл»). Именно эта строка и определена в заголовочном файле WinError.h для ошибки с кодом 2.

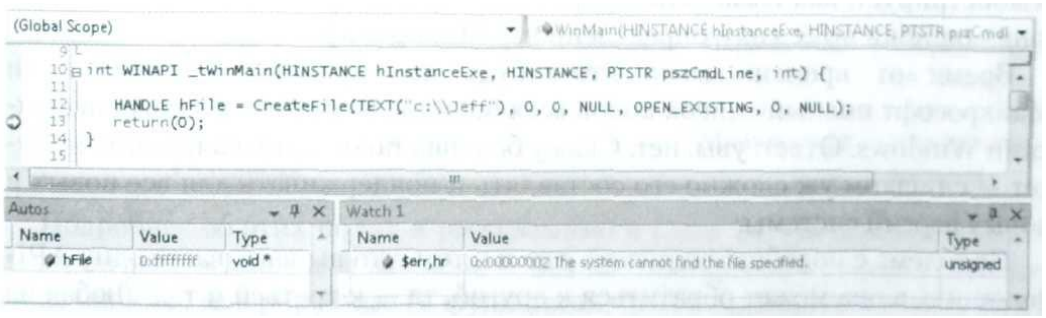
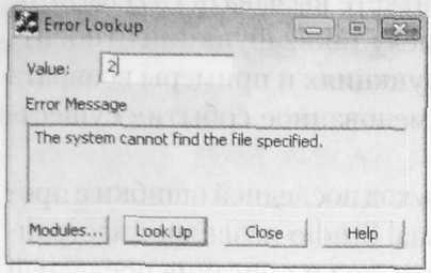


***Рис. 1-1**. Используя «@err,hr» в окне Watch среды Visual Studio, вы можете просматривать код последней ошибки в текущем потоке*

С Visual Studio поставляется небольшая утилита Error Lookup, которая позволяет получать описание ошибки по ее коду.

Если приложение обнаруживает какую-нибудь ошибку, то, как правило, сообщает о ней пользователю, выводя на экран ее описание. В Windows для этого есть специальная функция, которая «конвертирует» код ошибки в ее описание, — *FormatMessage*:

```
DWORD FormatMessage(
    DWORD dwFlags,
    LPCVOID pSource,
    DWORD dwMessageId,
    DWORD dwLanguageId,
    PTSTR pszBuffer,
    DWORD nSize,
    va_list *Arguments);
```



FormatMessage — весьма богатая по своим возможностям функция, и именно ее желательно применять при формировании всех строк, показываемых пользователю. Дело в том, что она позволяет легко работать с множеством языков. *FormatMessage* определяет, какой язык выбран в системе в качестве основного (этот параметр задается через апплет Regional Settings в Control Panel), и возвращает текст на соответствующем языке. Разумеется, сначала вы должны перевести строки на нужные языки и встроить этот ресурс в свой EXE- или DLL-модуль, зато потом функция будет автоматически выбирать требуемый язык. Программа-пример ErrorShow, приведенная в конце главы, демонстрирует, как вызывать эту функцию для получения текстового описания ошибки по ее коду, определенному Майкрософт.

Время от времени меня кто-нибудь да спрашивает, составит ли Майкрософт полный список кодов всех ошибок, возможных в каждой функции Windows. Ответ: увы, нет. Скажу больше, такого списка никогда не будет — слишком уж сложно его составлять и поддерживать для всё новых и новых версий системы.

Проблема с подобным списком еще и в том, что вы вызываете одну API-функцию, а она может обратиться к другой, та — к третьей и т. д. Любая из этих функций может завершиться неудачно (и по самым разным причинам). Иногда функция более высокого уровня сама справляется с ошибкой в одной из вызванных ею функций и в конечном счете выполняет то, что вы от нее хотели. В общем, для создания такого списка Майкрософт пришлось бы проследить цепочки вызовов в каждой функции, что очень трудно. А с появлением новой версии системы эти цепочки нужно было бы пересматривать заново.

Вы тоже можете это сделать

Итак, я показал, как функции Windows сообщают об ошибках. Майкрософт позволяет вам использовать этот механизм и в собственных функциях. Допустим, вы пишете функцию, к которой будут обращаться другие программы. Вызов этой функции может по какой-либо причине завершиться неудачно, и вам тоже нужно сообщать об ошибках. С этой целью вы просто устанавливаете код последней ошибки в потоке и возвращаете значение FALSE, INVALID_HANDLE_VALUE, NULL или что-то другое, более подходящее в Вашем случае. Чтобы установить код последней ошибки в потоке, вы вызываете *SetLastError*

```
VOID SetLastError(DWORD dwErrCode);
```

и передаете ей нужное 32-битное число. Я стараюсь использовать коды, уже определенные в WinError.h, — при условии, что они подходят под те ошибки, о которых могут сообщать мои функции. Если вы считаете, что ни один из кодов в WinError.h не годится для ошибки, возможной в вашей функции, определите свой код. Он представляет собой 32-битное значение, которое разбито на поля, показанные в следующей таблице.

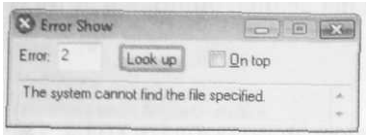
Табл. 1-2. Поля кода ошибки

Биты	31-30	29	28	27-16	15-0
Содержимое:	Код степени «тяжести» (severity)	Кем определен — Майкрософт или пользователем	Зарезервирован	Код подсистемы (facility code)	Код исключения
Значение:	0 = успех; 1 = информация; 2 = предупреждение; 3 = ошибка	0 = Майкрософт; 1 = пользователь	Должен быть 0	Первые 256 значений определяются Майкрософт	Определяется Майкрософт или пользователем

Подробнее об этих полях я рассказываю в главе 24. На данный момент единственное важное для вас поле — бит 29. Майкрософт обещает, что все коды ошибок, генерируемые ее функциями, будут содержать 0 в этом бите. Если вы определяете собственный код ошибки, запишите сюда 1. Тогда у вас будет гарантия, что ваш код ошибки не войдет в конфликт с кодом, определенным Майкрософт, — ни сейчас, ни в будущем. Заметьте, что код Facility может принимать 4096 возможных значений, из которых первые 256 Майкрософт зарезервировала для собственных нужд, а остальные доступны для использования в приложениях всем желающим.

Программа-пример ErrorShow

Эта программа, «01 ErrorShow.exe» (см. листинг на рис. 1-2), демонстрирует, как получить текстовое описание ошибки по ее коду. Файлы исходного кода и ресурсов программы находятся в каталоге 01-ErrorShow на компакт-диске, прилагаемом к книге, а также на сайте http://wintellect.com/Books.aspx. Программа ErrorShow в основном предназначена для того, чтобы вы увидели, как работают окно Watch отладчика и утилита Error Lookup. После запуска ErrorShow открывается следующее окно.



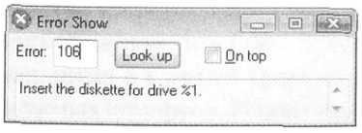
В поле Error можно ввести любой код ошибки. Когда вы щелкнете кнопку Look Up, внизу, в прокручиваемом окне появится текст с описанием данной ошибки. Единственная интересная особенность программы заключается в том, как она обращается к функции *FormatMessage*. Я использую эту функцию так:

```
// получаем код ошибки
DWORD dwError = GetDlgItemInt(hwnd, IDC_ERR0RC0DE, NULL, FALSE);

HL0CAL hlocal = NULL;                          // буфер для строки с описанием ошибки

// Мы ищем сообщения Windows, поэтому используем системные
// региональные параметры по умолчанию
// Примечание. Эта комбинация MAKELANGID имеет значение 0
DWORD systemLocale = MAKELANGID(LANG_NEUTRAL, SUBLANG_NEUTRAL);

// получаем описание ошибки по коду
BOOL fOk = FormatMessage(
    F0RMAT_MESSAGE_FR0M_8YSTEM | FORMAT_MESSAGE_IGNORE_INSERTS |
    FORMAT_MESSAGE_ALLOCATE_BUFFER,
    NULL, dwError, systemLocale,
    (PTSTR) &hlocal, 0, NULL);

if (!fOk) {
// Это ошибка, связанная с сетью?
HM0DULE hDll = LoadLibraryEx(TEXT("netmsg. dll"), NULL,
    DONT_RESOLVE_DLL_REFERENCES);

if (hDll != NULL) {
    fOk = FormatMessage(
        F0RMAT_MESSAGE_FR0M_HM0DULE | FORMAT_MESSAGE_IGNORE_INSERTS |
        FORMAT_MESSAGE_ALLOCATE_BUFFER,
```

```
        hDll, dwError, systemLocale,
        (PTSTR) &hlocal, 0, NULL);
     FreeLibrary(hDll);
   }
}

if (fOk && (hlocal 1= NULL)) {
   SetDlgItemText(hwnd, IDC_ERRORTEXT, (PCTSTR) LocalLock(hlocal));
   LocalFree(hlocal);
} else {
   SetDlgItemText(hwnd, IDC_ERRORTEXT,
       TEXT("No text found for this error number."));
```

Первая строка считывает код ошибки из текстового поля. Далее я создаю экземпляр описателя (handle) блока памяти и инициализирую его значением NULL. Функция FormatMessage сама выделяет нужный блок памяти и возвращает нам его описатель.

Вызывая *FormatMessage*, я передаю флаг FORMAT_MESSAGEJFROM_ SYSTEM. Он сообщает функции, что мне нужна строка, соответствующая коду ошибки, определенному в системе. Кроме того, я передаю флаг FORMAT_MESSAGE_ALLOCATE_BUFFER, чтобы функция выделила соответствующий блок памяти для хранения текста. Описатель этого блока будет возвращен в переменной hlocal. Флаг FORMAT_MESSAGE_IGNORE_INSERTS позволяет заменять в сообщениях параметры, которые используются Windows для передачи более детальной контекстной информации, подстановочными знаками, как показано на следующем рисунке:



Без этого флага необходимо передать вместо подстановочных знаков значения в параметре **Arguments**, но в случае Error Show это невозможно, поскольку содержимое сообщений заранее не известно.

Третий параметр указывает код интересующей нас ошибки, а четвертый — язык, на котором мы хотим увидеть ее описание. Поскольку мы хотим получить сообщения, переданные Windows, идентификатор языка создается из пары определенных констант, в результате получается 0 — значение, соответствующее языку по умолчанию, заданному в операционной системе. Это пример ситуации, в которой невозможно «зашить» в код идентификатор языка, поскольку нельзя узнать заранее, какой язык используется в той копии операционной системы, где будет запущена программа ErrorShow.

Если выполнение *FormatMessage* заканчивается успешно, описание ошибки помещается в блок памяти, и я копирую его в прокручиваемое окно, расположенное в нижней части окна программы. А если вызов *FormatMessage*

оказывается неудачным, я пытаюсь найти код сообщения в модуле NetMsg.dll, чтобы выяснить, не связана ли ошибка с сетью (о поиске DLL на диске см. в главе 20). Используя описатель NetMsg.dll, я вновь вызываю *FormatMessage*. Дело в том, что у каждого DLL или EXE-модуля может быть собственный набор кодов ошибок, который включается в модуль с помощью Message Compiler (MC.exe). Как раз это и позволяет делать утилита Error Lookup через свое диалоговое окно Modules.

ГЛАВА 2

Работа с символами и строками

Microsoft Windows становится все популярнее, и нам, разработчикам, надо больше ориентироваться на международные рынки. Раньше считалось нормальным, что локализованные версии программных продуктов выходят спустя полгода после их появления в США. Но расширение поддержки в операционной системе множества самых разных языков упрощает выпуск программ, рассчитанных на международные рынки, и тем самым сокращает задержки с началом их дистрибуции.

В Windows всегда были средства, помогающие разработчикам локализовать свои приложения. Программа получает специфичную для конкретной страны информацию (региональные стандарты), вызывая различные функции Windows, и узнает предпочтения пользователя, анализируя параметры, заданные в Control Panel. Кроме того, Windows поддерживает массу всевозможных шрифтов. И последний, но от этого не менее важный момент: Windows Vista теперь поддерживает Unicode 5.0 (подробнее о Unicode 5.0 см. в статье «Extend The Global Reach Of Your Applications With Unicode 5.0» по ссылке http://msdn.microsoft.com/msdnrnag/issues/07/01/Unicode/default.aspx).

Теперь приложения и даже компоненты часто атакуют через уязвимости, возникающие из-за ошибок в результате переполнения буфера (такие ошибки типичны при работе с текстовыми строками). В последнее время Майкрософт и ее партнеры приложили значительные усилия для укрепления безопасности в мире Windows. Во второй части этой главы рассказывается о новых функциях, добавленных Майкрософт в библиотеку времени выполнения языка C. Эти функции следует использовать для защиты кода от переполнения буфера при работе со строками.

Я решил переместить эту главу в начало книги, поскольку настоятельно рекомендую использовать в приложениях только Unicode-строки, а для манипулирования ими — только новые безопасные строковые функции. Вы

Оглавление

увидите, что почти во всех главах и примерах этой книги я касаюсь вопросов безопасности использования Unicode-строк. Разработчикам приложений, не поддерживающих Unicode, лучше позаботиться об их переводе на Unicode — это повысит производительность и подготовит приложения к локализации. Кроме того, это полезно для организации взаимодействия с COM и .NET Framework.

Наборы символов

Настоящей проблемой при локализации всегда были операции с различными наборами символов. Годами, кодируя текстовые строки как последовательности однобайтовых символов с нулем в конце, большинство программистов так к этому привыкло, что это стало чуть ли не второй их натурой. Вызываемая нами функция *strlen* возвращает количество символов в заканчивающемся нулем массиве однобайтовых символов.

Но существуют такие языки и системы письменности (классический пример — японские иероглифы), в которых столько знаков, что одного байта, позволяющего кодировать не более 256 символов, просто недостаточно. Для поддержки подобных языков были созданы двухбайтовые наборы символов (double-byte character sets, DBCS). В двухбайтовом наборе символ представляется либо одним, либо двумя байтами. Так, для японской каны (японской фонематической азбуки), если значение первого байта находится между 0x81 и 0x9F или между 0xE0 и 0xFC, надо проверить значение следующего байта в строке, чтобы определить полный символ. Работа с двухбайтовыми наборами символов — просто кошмар для программиста, так как часть их состоит из одного байта, а часть — из двух. К счастью, теперь можно забыть о DBCS и использовать поддержку Unicode-строк, реализованную в Windows-функциях и библиотечных функциях C.

Unicode — стандарт, первоначально разработанный Apple и Xerox в 1988 г. В 1991 г. был создан консорциум для совершенствования и внедрения Unicode. В него вошли компании Apple, Compaq, Hewlett-Packard, IBM, Microsoft, Oracle, Silicon Graphics, Sybase, Unisys и Xerox. (Полный список компаний — членов консорциума см. на www.unicode.org) Эта группа компаний наблюдает за соблюдением стандарта Unicode, описание которого Вы найдете в книге «The Unicode Standard» издательства Addison-Wesley (ее электронный вариант можно получить на том же www.unicode.org).

В Windows Vista для представления всех Unicode-символов используется кодировка UTF-16 (UTF — аббревиатура англ. Unicode Transformation Format). В UTF-16 символы представлены двумя байтами (16 битами). Если не сказано обратное, под Unicode в этой книге имеется в виду UTF-16. Эта кодировка используется в Windows, поскольку 16-битными значениями можно представить символы, составляющие алфавиты большинства языков мира, это позволяет программам быстрее обрабатывать строки и вычислять их длину. Однако для представления символов алфавита некоторых языков

16 бит недостаточно. Для таких случаев UTE-16 поддерживает «суррогатные» кодировки, позволяющие кодировать символы 32 битами (4 байтами). Впрочем, приложений, которым приходится иметь дело с символами таких языков, мало, поэтому UTF-16 — хороший компромисс между экономией памяти и простотой программирования. Заметьте, что в .NET Framework все символы кодируются с использованием UTF-16, поэтому применение UTF-16 в Windows-приложениях повышает производительность и снижает потребление памяти при передаче строк между «родным» и управляемым кодом. Существуют и другие стандарты UTF для представления символов, включая:

UTF-8. В кодировке UTF-8 разные символы могут быть представлены 1,2,3 или 4 байтами. Символы с значениями меньше 0x0080 сжимаются до 1 байта, что очень удобно для символов, применяемых в США. Символы, которым соответствуют значения из диапазона 0x0080-0x07FF, преобразуются в 2-байтовые значения, что хорошо работает с алфавитами европейских и ближневосточных языков. Символы с бóльшими значениями преобразуются в 3-байтовые значения, удобные при работе со среднеазиатскими языками. Наконец, «суррогатные» пары записываются в 4-байтовом формате. UTF-8— чрезвычайно популярная кодировка. Однако ее эффективность меньше по сравнению с UTF-16, если часто используются символы с значениями 0x0800 и выше.

UTF-32. В UTF-32 все символы представлены 4 байтами. Эта кодировка удобна для написания простых алгоритмов для перебора символов любого языка, не требующих обработки символов, представленных разным числом байтов. Например, при использовании UTF-32 можно забыть о «суррогатах», поскольку любой символ в этой кодировке представлен 4 байтами. Ясно, что с точки зрения использования памяти эффективность UTF-32 далека от идеала. Поэтому данную кодировку редко применяют для передачи строк по сети и сохранения их в файлы. Как правило, UTF-32 используется как внутренний формат представления данных в программе.

В настоящее время кодовые позиции[1] определены для арабского, китайского, греческого, еврейского, латинского (английского) алфавитов, а также для кириллицы (русского), японской каны, корейского хангыль и некоторых других алфавитов. В каждой версии Unicode добавляются новые символы и даже алфавиты, например финикийский (алфавит, использовавшийся в древней средиземноморской культуре). Кроме того, в набор символов включено большое количество знаков препинания, математических и технических символов, стрелок, диакритических и других знаков.

Эти 65 536 символов разбиты на отдельные группы. Некоторые группы, а также включенные в них символы показаны в таблице.

[1] Кодовая позиция — это положение символа в наборе символов.

Табл. 2-1. Наборы символов Unicode для различных алфавитов

16-битный код	Символы	16-битныЙ код	Символы
0000-007F	ASCII	0300-036F	Общие диакритические
0080-00FF	Символы Latin1	0400-04FF	Кириллица
0100-017F	Европейские латинские	0530-058F	Армянский
0180-01FF	Расширенные латинские	0590-05FF	Иврит
0250-02AF	Стандартные фонетические	0600-06FF	Арабский
02B0-02FF	Модифицированные литеры	0900-097F	Деванагари

Символьные и строковые типы данных для ANSI и Unicode

Уверен, вы знаете, что тип данных char в языке C представляет 8-битные ANSI-символы. По умолчанию при объявлении в коде строки-литерала компилятор C преобразует составляющие строку символы в массив 8-битных значений типа char.

```
// 8-битный символ
char c = 'A';

// массив 99 8-битных символов с нулем в конце (также 8-битным)
char szBuffer[100] = "A String";
```

Созданный Майкрософт компилятор C/C++ поддерживает встроенный тип данных *wchar_t*, представляющий 16-битные символы Unicode (в кодировке UTF-16). Прежние версии компилятора Майкрософт не поддерживали этот тип данных как встроенный и работали с ним только при указании параметра компилятора /Zc:wchar_t. Этот параметр компилятора установлен по умолчанию при создании C++-проекта в Microsoft Visual Studio. Рекомендуется всегда устанавливать этот параметр, поскольку с Unicode-символами лучше работать с использованием встроенных примитивных типов, понятных компилятору без «посредников».

> **Примечание**. Пока компилятор не поддерживал встроенный тип данных wchar_t, этот тип определялся в заголовочном файле C следующим образом:
>
> ```
> typedef unsigned short wchar_t;
> ```

Объявить Unicode-символ как строку можно следующим образом:

```
// 16-битный символ
wchar_t c = 'A';
```

```
// Массив 99 16-битных символов с нулем в конце (16-битным)
wchar_t szBuffer[100] = L"A String";
```

Заглавная буква L перед литералом сообщает компилятору, что указанный литерал следует компилировать как Unicode-строку. В этом простом примере при размещении данной строки в секции данных программы компилятор кодирует ее в UTF-16, вставляя нулевые байты между ASCII-символами.

Разработчики Windows хотят определить собственный тип данных, чтобы в некоторой степени изолировать себя от языка C. Поэтому в заголовочном файле Windows, WinNT.h, определены следующие типы данных:

```
typedef char     CHAR;     // 8-битный символ
typedef wchar_t  WCHAR;    // 16-битный символ
```

Кроме того, в заголовочном файле WinNT.h определен ряд общих типов данных для работы с указателями на символы и строки:

```
// указатель на 8-битные символы и строки
typedef CHAR *PCHAR;
typedef CHAR *PSTR;
typedef CONST CHAR *PCSTR

// указатель на 16-битные символы и строки
typedef WCHAR *PWCHAR;
typedef WCHAR *PWSTR;
typedef CONST WCHAR ♦PCWSTR;
```

> **Примечание.** В файле WinNT.h есть следующее определение:
>
> ```
> typedef __nullterminated WCHAR *NWPSTR, *LPWSTR, *PWSTR;
> ```
>
> Префикс **__nullterminated** представляет собой *заголовочную аннотацию (header annotation)*, описывающую применение типов в качестве параметров и возвращаемых значений функций. В редакции Enterprise среды Visual Studio можно включить параметр Code Analysis в свойствах проекта. В результате в командную строку компилятора будет добавлен параметр /analyze. Он заставит компилятор проверять, не вызываются ли функции с нарушением семантики, предписанной аннотациями. Заметьте: только в версии Enterprise поддерживает параметр /analyze. Для простоты в этой книге заголовочные аннотации убраны из примеров кода. Подробнее о заголовочных аннотациях см. в статье MSDN «Header Annotations» (http://msd2.microsoft.com/En-US/library/aa383701.aspx).

Независимо от того, какие именно типы данных вы используете в своем коде, рекомендую следить за их согласованностью ради удобства сопровождения вашего кода. Поскольку я программирую для Windows, я всегда использую типы данных Windows — они соответствуют документации MSDN, что существенно облегчает другим чтение моего кода.

Можно написать код так, чтобы он компилировался с ANSI- и Unicode-символами. В заголовочном файле WinNT.h определены следующие типы данных и макросы:

```
#ifdef UNICODE

typedef WCHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST WCHAR *PCTSTR;
#define __TEXT(quote) quote                    // r_winnt

#define __TEXT(quote) L##quote

#else

typedef CHAR TCHAR, *PTCHAR, PTSTR;
typedef CONST CHAR •PCTSTR;
#define __TEXT(quote) quote

#endif

#define TEXT(quote) __JEXT(quote)
```

Эти типы и макросы (и некоторые другие, менее востребованные и потому здесь не показанные) используются для создания код, который может компилироваться как с ANSI-, так и Unicode-символами, например:

```
// 16-битный символ, если определен UNICODE,
// или 8-битный символ в противном случае
TCHAR c = TEXT ('A');

// массив 16-битных символов, если определен UNICODE,
// либо 8-битных символов в противном случае
TCHAR szBuffer[100] = TEXT("A String");
```

Unicode- и ANSI-функции в Windows

Начиная с Windows NT, все версии Windows создаются на основе, включающей Unicode. Иными словами, все ключевые функции для создания окон, вывода текста, манипулирования строками и т.д. требуют Unicode-строки. Если любой Windows-функции передать при вызове ANSI-строку (строку 1-байтовых символов), эта функция сначала преобразует ANSI-строку в Unicode, и только после этого передаст ее операционной системе. Если некоторая функция должна возвращать ANSI-строку, операционная система преобразует Unicode-строку в ANSI и возвращает результат вашему приложению. Все эти преобразования выполняются незаметно для программиста и, естественно, вызывают дополнительный расход памяти и времени.

Я уже говорил, что существует две функции *CreateWindowEx*: одна принимает строки в Unicode, другая — в ANSI. Все так, но в действительности прототипы этих функций чуть-чуть отличаются:

```
HWND WINAPI CreateWindowExW (DWORD dwExStyle,
    PCWSTR pClassName,                      // Unicode-строка
    PCWSTR pWindowName,                     // Unicode-строка
    DWORD dwStyle,
    int X,
    int Y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hHenu,
    HINSTANCE hInstance,
    PVOID pParam);

HWND WINAPI CreateWindowExA (DWORD dwExStyle,
    PCSTR pClassName,                       // ANSI-строка
    PCSTR pWindowName,                      // ANSI-строка
    DWORD dwStyle,
    int X,
    int Y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HHENU hMenu,
    HINSTANCE hInstance,
    PVOID pParam);
```

CreateWindowExW — это Unicode-версия. Буква W конце имени функции — аббревиатура слова *wide* (широкий). Символы Unicode занимают по 16 битов каждый, поэтому их иногда называют широкими символами (wide characters). Буква A в конце имени *CreateWindowExA* указывает, что данная версия функции принимает ANSI-строки.

Но обычно *CreateWindowExW* или *CreateWindowExA* напрямую не вызывают, а обращаются к *CreateWindowEx* — макросу, определенному в файле WinUser.h:

```
#ifdef UNICODE
#define CreateWindowEx CreateWindowExW
#else
#define CreateWindowEx CreateWindowExA
#endif
```

Какая именно версия CreateWindowEx будет вызвана, зависит от того, определен ли UNICODE в период компиляции. Перенося 16-разрядное Windows-приложение на платформу Win32, вы, вероятно, не станете определять UNICODE. Тогда все вызовы *CreateWindowEx* будут преобразованы в вызовы CreateWindowExA — ANSI-версии функции. И перенос приложения упростится, ведь 16-разрядная Windows работает только с ANSI-версией *CreateWindowEx*.

В Windows Vista функция *CreateWindowExA* — просто шлюз (транслятор), который выделяет память для преобразования строк из ANSI в Unicode и вызывает *CreateWindowExW*, передавая ей преобразованные строки. Когда *CreateWindowExW* вернет управление, *CreateWindowExA* освободит буферы и передаст вам описатель окна.

Таким образом, при вызове функций, заполняющих буферы строками система должна преобразовать Unicode-строки в их эквиваленты, прежде чем ваше приложение сможет обработать строки. Из-за необходимости этих преобразований ваши приложения требуют больше памяти и медленнее работают. Чтобы повысить быстродействие приложений, следует с самого начала писать их в расчете на работу с Unicode-строками. Кроме того, в функциях, преобразующих строки, найдены ошибки, так что отказ от их использования уменьшает число потенциальных сбоев в приложениях.

Разрабатывая DLL, которую будут использовать и другие программисты, предусматривайте в ней по две версии каждой функции — для ANSI и для Unicode. В ANSI-версии просто выделяйте память, преобразуйте строки и вызывайте Unicode-версию той же функции. (Этот процесс я продемонстрирую позже.)

Некоторые функции Windows API (например, *WinExec* или *OpenFile*) существуют только для совместимости с 16-разрядными программами, и их надо избегать. Лучше заменить все вызовы *WinExec* и *OpenFile* вызовами *CreateProcess* и *CreateFile* соответственно. Тем более что старые функции просто обращаются к новым. Самая серьезная проблема с ними в том, что они не принимают строки в Unicode, — при их вызове вы должны передавать строки в ANSI. С другой стороны, в Windows 2000 у всех новых или пока не устаревших функций обязательно есть как ANSI-, так и Unicode-версия.

Некоторые функции Windows API, такие как *WinExec* и *OpenFile*, существуют исключительно для преемственной совместимости с 16-разрядными Windows-приложениями, поддерживающими исключительно ANSI-строки. Использовать их в современных программах не следует. Вызовы *WinExec* и *OpenFile* следует заменять вызовами *CreateProcess* и *CreateFile*, устаревшие функции все равно вызывают новые функции. Основная проблема с устаревшими функциями в том, что они не принимают Unicode-строки и предоставляют меньше возможностей, при вызове им необходимо передавать ANSI-строки. В Windows Vista для большинства функций, не успевших устареть, существуют как Unicode-, так и ANSI-версии. Однако Майкрософт проявляет тенденцию к созданию функций, поддерживающих только Unicode, примерами могут быть *ReadDirectoryChangesW* и *CreateProcessWithLogonW*.

При переносе COM из 16-разрядных версий Windows на Win32 руководство Майкрософт решило сделать так, чтобы все методы интерфейсов COM, требующие строки, принимали только Unicode-строки. Это было правильное решение, поскольку COM обычно применяют для организации «общения» различных компонентов, а Unicode обеспечивает максимум возможностей для передачи строковых данных. Таким образом, использование Unicode в приложении облегчает и взаимодействие с COM.

В итоге компилятор ресурсов обрабатывает все ваши ресурсы, записывая их в выходной файл в двоичном формате. Строковые ресурсы (таблицы, шаблоны диалоговых окон, меню и пр.) всегда записываются в файлы ресурсов в виде Unicode-строк. Если в приложении не определен макрос UNICODE, в Windows Vista соответствующие преобразования выполняет система. Например, если при компиляции модуля не определен UNICODE, в результате вызова *LoadString* в действительности будет вызвана функция *LoadStringA*. *LoadStringA* прочитает Unicode-строку из ресурса и преобразует ее в ANSI, а *LoadString* вернет приложению ANSI-представление строки.

Unicode- и ANSI-функции в библиотеке C

Подобно Windows, библиотека языка C поддерживает два набора функций: один для манипулирования ANSI-строками и символами, а другой — для работы с Unicode-строками и символами. Однако, в отличие от Windows, здесь ANSI-функции выполняют свою работу. Они не преобразуют внутренне полученные строки в Unicode и не вызывают затем Unicode-версии тех же функций. Ну, и, конечно, Unicode-версии сами делают то, что им положено, а не вызывают ANSI-версии.

Примером C-функции времени выполнения, возвращающей длину ANSI-строк, может служить *strlen*, а эквивалентной функции для Unicode-строк —*wcslen*. Прототипы обеих функций объявлены в String.h. В код, который может компилироваться как с ANSI-, так и с Unicode-строками, необходимо импортировать заголовочный файл TChar.h, в котором определен следующий макрос:

```
#ifdef _UNIC00E
#define _tcslen      wcslen
#else
#define _tcslen      strlen
#endif
```

Теперь вызовите в коде *_tcslen*. Если определен _UNICODE, вызов разрешается в вызов *wcslen*, в противном случае — в *strlen*. По умолчанию в новых C++-проектах Visual Studio определен _UNICODE (как и UNICODE). В библиотеке C времени выполнения все идентификаторы, которые не входят в стандарт C++, предваряются символами подчеркивания (_), тогда как разработчики Windows этого не делают. Таким образом, следует определять

приложениях либо оба параметра (UNICODE и _UNICODE), либо ни дин из них. Подробнее о заголовочном файле CmnHdr.h, использованном во всех примерах из этой книги, рассказывается в приложении А.

Безопасные строковые функции в библиотеке C

Любая функция, модифицирующая строку, несет потенциальную угрозу безопасности: если результирующая строка больше, чем предназначенный для нее буфер, содержимое памяти будет испорчено. Рассмотрим пример:

```
// Этот код записывает 4 символа в буфер
// длиной 3 символа, повреждая содержимое памяти
WCHAR szBuffer[3] = L"";
wcscpy(szBuffer, L"abc"); // Концевой 0 - тоже символ!
```

Проблема с функциями *strcpy* и *wcscpy* (как и с большинством других функций, манипулирующих строками) состоит в том, что у них нет параметра, задающего максимальный размер буфера. Следовательно, такая функция не сможет «узнать», не испортит ли она память. Если же функция не «узнает», повредит ли она содержимое памяти, то не сможет и сообщить об ошибке коду приложения, а вы не узнаете о том, что память испорчена. Конечно же, было бы лучше, если вызов функции просто завершался неудачей, оставляя память в целости и сохранности.

В прошлом эти сбои активно использовались вредоносными программами. Майкрософт заменила в хорошо знакомой и любимой программистами библиотеке C времени выполнения, небезопасные функции для манипулирования строками (такие, как вышеописанная *wcscat*) новыми функциями. Чтобы писать безопасный код, необходимо отказаться от традиционных C-функций, модифицирующих строки. (Впрочем, к таким функциям, как *strlen, wcslen* и *_tcslen*, это не относится: эти функции не пытаются изменить переданную им строку, хоть и предполагают, что эта строка заканчивается 0, что не всегда так.) Вместо них используйте новые безопасные функции для манипулирования строками, определенные Майкрософт в заголовочном файле StrSafe.h.

> **Примечание**. Майкрософт изменила внутреннее устройство своих библиотек классов ATL и MFC, добавив в них поддержку безопасных строковых функции. Поэтому для укрепления защиты программ, использующих эти библиотеки, достаточно просто перекомпилировать их.

Защита кода на C/C++ выходит за рамки этой книги, поэтому приведу лишь ссылки на источники дополнительных сведений по данному вопросу:

- статья Мартина Лоуэлла (Martyn Lovell) в MSDN Magazine «Repel Attacks on Your Code with the Visual Studio 2005 Safe C and C++ Libraries» (http://msdn.microsoft.com/msdninag/issucs/05/05/SafeCandC/default.aspx);

- презентация Мартина Лоуэлла на Channel9 (видео, см. http://channel9.msdn.com/Showpost.aspx?postid-186406);
- раздел по защите работы со строками в MSDN Online (см. http://msdn2.microsoft.com/en-us/library/ms647466.aspx);
- полный список безопасных С-функций времени выполнения см. в MSDN Online (http://msdn2.microsoft.com/en-us/library/wd3wzwts(VS.80).spx).

Однако некоторые вопросы все же стоит обсудить здесь. Я начну с описания общих особенностей новых функций. Затем я расскажу о «подводных камнях», подстерегающих программистов при переходе к использованию новых, безопасных версий строковых функций, например о применении *_tcscpy_s* вместо *_tcscpy*. В завершение я покажу, когда имеет смысл вызывать новые функции StringC*.

Введение в безопасные строковые функции

Вместе с StrSafe.h импортируется String.h, но прежние функции для манипулирования строками из библиотеки С времени выполнения, например из макроса *_tcscpy*, помечаются как устаревшие и генерируют предупреждения во время компиляции. Учтите, что импортировать StrSafe.h в исходном коде следует в последнюю очередь после импорта других заголовочных файлов. Рекомендую также использовать предупреждения, сгенерированные при компиляции, как ориентир для замены всех устаревших функций их безопасными версиями. В каждом случае следует продумать обработку переполнения буфера, а если восстановление после этой ошибки невозможно, то способ корректного завершения программы.

Для каждой из существующих строковых функций, таких как *_tcscpy* и *_tcscat*, предусмотрена новая версия, имя которой включает имя старой функции и суффикс *_s* (от англ. secure — безопасный). Новые функции имеют ряд общих особенностей, которые нужно пояснить. Проанализируем их прототипы на примере следующего листинга с парой определений обычных строковых функций:

```
PTSTR   _tcscpy   (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscpy_s (PTSTR strDestination, size_t numberOfCharacters,
    PCTSTR strSource);

PTSTR   _tcscat   (PTSTR strDestination, PCTSTR strSource);
errno_t _tcscat_s (PTSTR strDestination, size_t numberOfcharacters,
    PCTSTR strSource);
```

Если функции передается в качестве параметра буфер, доступный для записи, то одновременно необходимо передать и его размер. Размер буфера задают как число знаков, чтобы узнать его, достаточно вызвать макрос *_countof*, определенный в stdlib.h.

Все безопасные (имеющие в названии суффикс *_s*) функции первым делом проверяют переданные им аргументы. В частности, проверяются указатели (не равны ли они **NULL**), целочисленные значения (находятся ли они в допустимом диапазоне), перечислимые (допустимы ли их значения), а также размер буфера (достаточен ли он для хранения результирующих данных). Если хоть одна из проверок окончиться неудачей, функция устанавливает локальную C-переменную времени выполнения потока *errno* и возвращает значение *errno_t*, указывающее успешное или неудачное завершение функции. Однако при возникновении ошибки безопасные функции на самом деле не возвращают управление коду приложения, а вызывают окно с неудобоваримым содержимым (рис. 2-1), после чего приложение завершается (это происходит в отладочной сборке, а в случае окончательной сборки приложение завершается сразу).

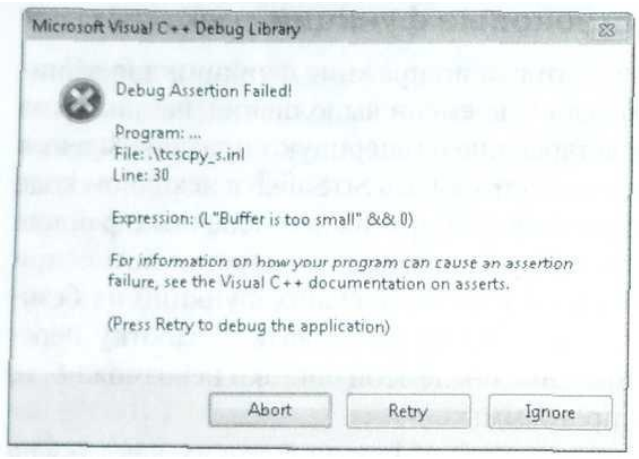


Рис. 2-1. *Окно, отображаемое в случае ошибки при вызове функции*

В действительности библиотека C позволяет определять собственные функции, которые будут вызываться при обнаружении недопустимого параметра. В этой функции можно занести ошибку в журнал или выполнить любые другие действия. Чтобы это работало, прежде всего, необходимо определить функцию, соответствующую следующему прототипу:

```
void InvalidParameterHandler(PCTSTR expression, PCTSTR function,
    PCTSTR file, unsigned int line, uintptr_t /*pReserved*/);
```

Параметр expression предоставляет описание ошибки, которое будет использоваться в реализации функции, например (L”Buffer is too small” && 0). Он не слишком понятен неспециалистам, поэтому не следует показывать его конечным пользователям. То же верно для следующих трех параметров, *function*, *file* и *line*, представляющих, соответственно, имя функции, файл с исходным кодом и номер строки, в которой возникла ошибка.

> **Примечание**. Если не определен параметр **DEBUG**, у всех этих аргументов будет значение **NULL.** Таким образом, использовать данный разработчик для регистрации ошибок можно лишь при тестировании отладочных версий. В окончательной сборке вывод окна, показанного на рис. 2-1, можно заменить более понятным пользователю сообщением, например о том, что работа приложения прервана из-за неожиданной ошибки, возможно, после имеет смысл занести ошибку в журнал и перезапустить приложение. Если данные состояния приложения в памяти были испорчены, его исполнение следует прервать. Впрочем, рекомендуется проверить ***errno_t***, чтобы определить, можно ли исправить эту ошибку.

Далее следует зарегистрировать этот обработчик, вызвав _set_invalid_parameter_handler. Однако это еще не все, поскольку при возникновении ошибки по-прежнему будет появляться системное окно. В начале исполнения приложения следует вызвать CrtSetReportMode(_CRT_ASSERT, 0); чтобы отключить все системные оповещения об ошибках, выводимые библиотекой C во время выполнения.

Теперь при сбое строковых функций, определенных в String.h, можно будет узнать причину ошибки, проверив *errno_t*. Только значение S_OK свидетельствует об успешном вызове функции. Остальные значения (см. errno.h) говорят о различных ошибках, например EINVAL — о недопустимых аргументах, таких как указатель, содержащие NULL.

Рассмотрим пример с копированием строки в буфер слишком малого размера:

```
TCHAR szBefore[5] = {
    TEXT('B'), TEXT('B'), TEXT('B'), TEXT('B'), '\0'
};

TCHAR szBuffer[10] = {
    TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'),
    TEXT('-'), TEXT('-'), TEXT('-'), TEXT('-'), '\0'
};

TCHAR szAfter[5] = {
TEXT('A*), TEXT('A'), TEXT('A'), TEXT('A'), '\0'
};

errno_t result = _tcscpy_8(8zBuffer, _countof(szBuffer),
    TEXT("0123456789"));
```

Содержимое переменных перед вызовом *_tcscpy_s* показано на рис. 2-2.

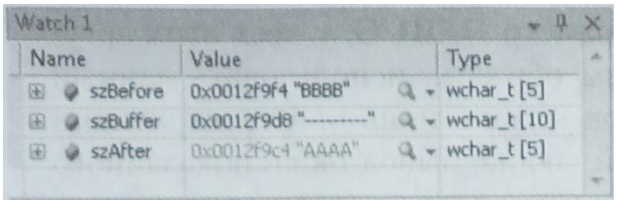


Рис. 2-2. *Состояние переменных перед вызовом _tcscpy_s*

Поскольку длина строки «1234567890», копируемой в буфер *szBuffer*, точно равна размеру буфера (10 символов), концевой символ строки, «\0», в буфер не уместится. Не думайте, что значение *result* будет усечено (вызовом STRUNCATE) и последний символ, «9», просто не будет скопирован в буфер. Вместо этого возвращается ERANGE и переменные приходят в состояние, показанное на рис. 2-3.

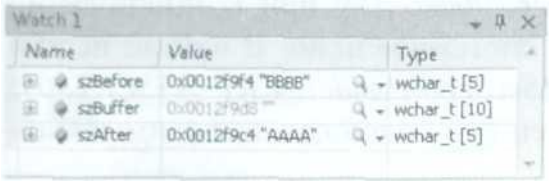


Рис. 2-3. *Состояние переменных после вызова _tcscpy_s*

Есть и еще один «побочный эффект», который нельзя увидеть, не проанализировав содержимое ячеек памяти, расположенных сразу за *szBuffer* (рис. 2-4).

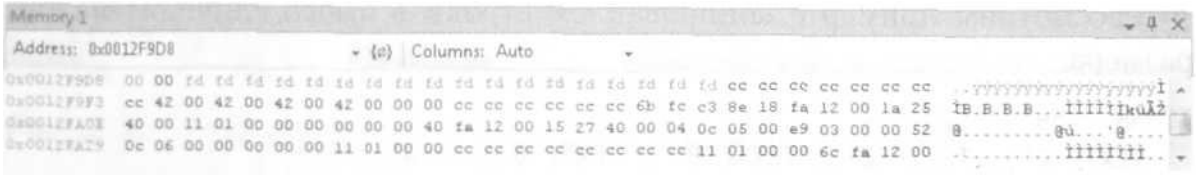


Рис. 2-4. *Содержимое szBuffer после неудачного вызова _tcscpy_s*

Первый символ установлен в «\0», а остальные байты теперь содержат значение 0xfd. В результате результирующая строка оказалась усеченной до пустой строки, а остальные байты буфера — занятыми значениями-заполнителями (0xfd).

> **Примечание.** Если вам интересно, почему ячейки памяти между переменными заполнены значениями 0xcc (см. рис. 2-4), скажу: из-за автоматического обнаружения компилятором переполнения буфера во время выполнения (оно включается флагами /RTCs, /RTCu и /RTC1). Если код компилируется без флагов /RTCx, переменные sz* будут располагаться в памяти «вплотную». Помните, что всегда следует устанавливать эти флаги при компиляции — это позволит обнаружить еще не выявленные ошибки из-за переполнения буфера на ранних стадиях цикла разработки.

Дополнительные возможности при работе со строками

Помимо безопасных строковых функций, библиотека C поддерживает ряд новых функций, предоставляющих дополнительные возможности при манипулировании строками. Например, с их помощью можно управлять символами-заполнителями и способом усечения строк. Естественно, поддерживаются как ANSI- (A), так и Unicode-версии (W) этих функций. Вот прототипы для некоторых их них:

```
HRESULT StringCchCat(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCatEx(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc,
    PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);
HRESULT StringCchCopy(PTSTR pszDest, size_t cchDest, PCTSTR pszSrc);
HRESULT StringCchCopyEx(PTSTR pszDest. size_t cchDest, PCTSTR pszSrc,
    PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags);

HRESULT StringCchPrintf(PTSTR pszDest, size_t cchDest,
    PCTSTR pszFormat, …);
HRESULT StringCchPrintfEx(PTSTR pszDest, size_t cchDest,
    PTSTR *ppszDestEnd, size_t *pcchRemaining, DWORD dwFlags,
    PCTSTR pszFormat, …);
```

Несложно заметить строку «Cch» в именах показанных здесь методов: это сокращение от *«Count of characters»*, обычно это значение получают при помощи макроса *_countof*. Также поддерживаются функции с именами, содержащими строку «Cb», такие как *StringCbCat(Ex)*, *StringCbCopy(Ex)* и *StringCbPrintf(Ex)*. Эти функции принимают в виде параметра размер, выраженный в байтах, а не символах. Как правило, его значение получают с помощью оператора *sizeof*. Все эти функции возвращают HRESULT с одним из значений, перечисленных в табл. 2-2.

Табл. 2-2. Значения HRESULT для безопасных строковых функций

Значение	Описание
S_OK	Успешный вызов. В целевом буфере находится исходная строка, оканчивающаяся знаками «\0»
STRSAFE_E_INVALID_PARAMETER	Неудачный вызов. В параметрах передан NULL
STRSAFE_E_INSUFFICIENT_BUFFER	Неудачный вызов. Исходная строка не уместилась в целевом буфере

В отличие от безопасных функций (функций с суффиксом _s в именах), эти функции усекают строку, если она не умещается в буфер. Об этом свидетельствует возврат STRSAFE_E_INSUFFICIENT_BUFFER. Из кода StrSafe.h видно, что этот код имеет значение 0x8007007a, и макрос SUCCEEDED/FAILED считает его неудачей вызова. Однако в этом слу-

чае копируется часть содержимого исходного буфера, способная уместиться в целевом буфере, при этом последним символом скопированной строки становится «\0». Таким образом, если в предыдущем примере использовать *StringCchCopy* вместо *_tcscpy_s*, то в *szBuffer* оказалась бы строка «012345678». В зависимости от поставленной задачи, усечение строки при копировании может быть допустимо, а, может, и нет. Именно поэтому данная ситуация по умолчанию считается неудачей вызова. Например, при сборке пути путем конкатенации строк усечение сделает полученный результат бесполезным. В случае же подготовки текстового сообщения для пользователя усечение может оказаться приемлемым вариантом. В любом случае, вам решать, что делать с усеченным при копировании результатом.

И последнее (по порядку, но не по важности): у многих показанных выше функций есть расширенная (*Ex*) версия. Расширенные версии принимают три дополнительных параметра, описанных в табл. 2-3.

Табл. 2-3. Параметры расширенных функций

Параметр	Описание
size_t* pcchRemaining	Указатель на переменную, представляющую число незанятых символов в целевом буфере (без учета копируемого концевого нуля, \0). Например, при копировании одного символа в буфер длиной 10 символов будет возвращено значение 9, из которых доступны лишь 8 (остальные символы будут усечены). Если pcchRemaining = NULL, число незанятых символов не возвращается
LPTSTR* ppszDestEnd	Значение ppszDestEnd, отличное от NULL, указывает на «\0», завершающий строку в целевом буфере
DWORD dwFlags	Одно или несколько значений, разделенных знаком «|»
STRSAFE_FILL_ BEHIND_NULL	При успешном вызове использует младший байт dwFlags для заполнения области буфера, оставшейся незанятой (после концевого «\0»). Подробнее см. в примечании о STRSAFE_FILL_BYTE после таблицы
STRSAFE_IGNORE_ NULLS	Заставляет обращаться с указателями на строки, содержащими NULL, как с пустыми строками (TEXT(“”))
STRSAFE_FILL_ ON_FAILURE	При неудачном вызове использует младший байт dwFlags для заполнения целевого буфера (за исключением концевого «\0»), подробнее см. в описании STRSAFE_FILL_BYTE после таблицы. В случае ошибки STRSAFE_E_INSUFFICIENT_BUFFER все символы возвращаемой строки замещаются байтами-заполнителями
STRSAFE NULL_ ON_FAILURE	При неудачном вызове первым символом целевого буфера становится «\0», то есть в буфер заносится пустая строка (TEXT(“”)). В случае ошибки STRSAFE_E_INS U FFI-CIENT_BUFFER любая усеченная строка будет перезаписана

Параметр	Описание
STRSAFE_NO_ TRUNCATION	Как и в случае STRSAFE_NULL_ON_FAILURE, при сбое функции в целевой буфер заносится пустая строка (TEXT (“”)). В случае ошибки STRSAFE_E_INSUFFICIENT_BUF-FER любая усеченная строка будет перезаписана

Примечание. Даже если установлен флаг STRSAFE_NO_TRUNCATION, в целевой буфер будут скопированы все символы исходной строки, которые в нем уместятся. Далее первый и последний символы в целевом буфере устанавливаются в «\0». Это важно только в ситуациях, когда по соображениям безопасности следует уничтожить ненужные данные.

В завершение вернемся к комментарию на стр. 21. Как видно из рис. 2-4, все ячейки целевого буфера, от «\0» и до конца, заменены значениями 0xfd. Расширенные (Ex) версии этих функций позволяют при желании отменить заполнение буфера, весьма затратное (особенно в случае большого буфера) в плане системных ресурсов. Если добавить STRSAFE_FILL_BEHIND_ NULL к *dwFlag*, остальные символы будут установлены в «\0». Если же заменить STRSAFE_FILL_BEHIND_NULL макросом STRSAFE_FILL_BYTE, для заполнения незанятой части целевого буфера будет использоваться заданный байт.

Строковые функции Windows

Windows предлагает внушительный набор функций, работающих со строками. Многие из них, такие как *lstrcat* и *lstrcpy*, считаются устаревшими и не рекомендуются к использованию, поскольку не позволяют обнаружить ошибок, возникающих из-за переполнения буфера. Кроме того, в ShlwApi.h определен ряд удобных строковых функций, таких как *StrFormatKBSize* и *StrFormatByteSize*, форматирующих числовые значения, связанные с работой операционной системы. Подробнее о строковых функциях оболочки см. по ссылке http://msdn2.microsoft.com/en-us/library/ms538658.aspx.

Операция сравнения строк используется при проверке равенства, а также при сортировке. Наилучшие функции для этой операции — *CompareString(Ex)* и *CompareStringOriginal*. *CompareString(Ex)* используется для сравнения строк, которые затем будут показаны пользователю, с учетом языковых особенностей. Вот прототип функции *CompareString*.

```
int CompareString(
    LCID locale,
    DWORD dwCmdFlags,
    PCTSTR pString1,
```

```
    int cch1,
    PCTSTR pString2,
    int cch2);
```

Она сравнивает две строки. Первый параметр задает так называемый идентификатор локализации (locale ID, LCID) — 32-битное значение, определяющее конкретный язык, С помощью этого идентификатора *CompareString* сравнивает строки с учетом значения конкретных символов в данном языке. Так что она действует куда осмысленнее, чем функции библиотеки С. Однако, эта операция существенно медленнее сравнения по порядку. Получить LCID можно вызовом Windows-функции GetThreadLocale:

```
LCID GetThreadLocale();
```

Второй параметр функции *CompareString* указывает флаги, модифицирующие метод сравнения строк. Допустимые флаги перечислены в следующей таблице.

Табл. 2-4. Флаги функции *CompareString*

Флаг	Действие
NORM_IGNORECASE	Различия в регистре букв игнорируются LINGUISTIC_IGNORECASE
NORM_IGNOREKANATYPE	Различия между знаками хираганы и катаканы игнорируются
NORM_IGNORENONSPACE	Знаки, отличные от пробелов, игнорируются LINGUISTIC_IGNOREDIACRITIC
NORM_ IGNORESYMBOLS	Символы, отличные от алфавитно-цифровых, игнорируются
NORM_IGNOREWIDTH	Разница между одно- и двухбайтовым представлением одного и того же символа игнорируется
NORM_STRDVGSORT	Знаки препинания обрабатываются так же, как и символы, отличные от алфавитно-цифровых

Остальные четыре параметра *CompareString* задают две строки и их длину в символах (а не в байтах!), соответственно. Если передать в параметре *cch1* отрицательное значение, функция рассчитывает длину строки *pString1*, предполагая, что эта строка оканчивается нулем. То же верно и для параметра *cch2* в отношении строки *pString2*. Если вам нужны дополнительные возможности, связанные с особенностями различных языков, взгляните на функции *CompareStringEx*.

Для сравнения строк при программной генерации строковых элементов (путей, разделов и параметров реестра, XML-атрибутов и элементов) применяют функцию CompareStringOrdinal:

```
int CompareStringOrdinal(
    PCWSTR p8tring1,
```

```
int cchCount1,
PCWSTR pString2,
int cchCount2,
BOOL bIgnoreCase);
```

Эта функция выполняет сравнение строк как строк программы, то есть без учета региональных параметров, и потому работает быстро. Это наиболее полезная функция, поскольку текст программы, как правило, не виден конечным пользователям. Учтите, что она работает только с Unicode-строками.

Функции *CompareString* и *CompareStringOrdinal* возвращают значения, которые не похожи на значения, возвращаемые функциями вида *cmp из библиотеки C.

CompareString(Ordinal) возвращает 0 в случае сбоя; CSTR_LESS_THAN (значение 1), если *pString1* меньше *pString2*; CSTR_EQUAL (значение 2), если *pString1 = pString2*; и CSTR_GREATER_THAN (значение 3), если *pString1* больше *pString2*. Можно немного упростить себе жизнь, если (после успешного вызова) вычесть 2 из возвращаемого значения — получится результат, соответствующий значению, возвращаемому функциями из библиотеки C (-1,0, и +1).

Почему Unicode?

Разрабатывая приложение, вы определенно должны использовать преимущества Unicode, поскольку они:

- упрощают локализацию приложений для распространения их на мировом рынке;
- позволяют распространять единственный двоичный EXE- или DLL-файл, поддерживающий все языки;
- увеличивают эффективность приложений, поскольку с Unicode-строками код работает быстрее, используя при этом меньше памяти. Внутренне Windows работает только с Unicode-символами и строками, поэтому получив ANSI-символ или строку, Windows должна преобразовать ее в Unicode-эквивалент, выделив для этого память;
- дают возможность вызывать все современные Windows-функции (некоторые Windows-функции поддерживают только Unicode-символы и строки);
- облегчают интеграцию с COM, требующей использования Unicode-символов и строк;
- гарантируют простую интеграцию вашего кода с .NET Framework (также требующей применения Unicode-символов и строк);
- упрощают манипулирование собственными ресурсами приложений (которые также всегда хранятся в Unicode).

Рекомендуемые приемы работы с символами и строками

В начале этого раздела я вкратце повторю правила, которых следует придерживаться при написании кода, а в конце дам несколько советов о том, как эффективнее манипулировать Unicode- и ANSI-строками. Даже если вы не планируете использовать Unicode сейчас, все же желательно создавать приложения в расчете на работу с Unicode. При разработке следуйте этим правилам:

- старайтесь думать о тестовых строках как о массивах символов, а не массивах значений типа *chars* или массивах байтов;
- используйте обобщенные типы данных (такие, как TCHAR/PTSTR) для текстовых строк и символов;
- используйте явные типы данных (такие как BYTE и PBYTE) для хранения байтов, указателей на байты и буферов;
- используйте для литералов макросы TEXT или _T но не применяйте их вперемешку — это вносит в код путаницу;
- пользуйтесь глобальной заменой (например, для замены PSTR на PTSTR);
- оптимизируйте выделение памяти для строк. Функции обычно принимают размер буфера, выраженный в символах, а не в байтах. Это означает, что передавать следует *_countof(szBuffer)*, а не *sizeof(szBuffer)*. Кроме того, помните, что размер блока памяти, выделяемого для строк, включающих известное число символов, указывают в байтах. Следовательно, необходимо вызывать *malloc(nCharacters* sizeof(TCHAR))*, а не *malloc(nCharacters)*. Это правило труднее всего запомнить, и компилятор никак не предупреждает о его нарушении, поэтому здесь придется весьма кстати макрос следующего вида:

```
#define chmalloc(nCharacters)   (TCHAR*)malloc(nCharacters * sizeof(TCHAR))
```

- избегайте функций из семейства *printf*, особенно при использовании типов полей %s и %S для преобразования между ANSI и Unicode. Вместо них используйте *MultiByteToWideChar* и *WideCharToMultiByte*, как показано ниже, в разделе о перекодировке строк;
- всегда определяйте символы UNICODE и _UNICODE одновременно либо не определяйте ни один из них.

Этих правил следует придерживаться при манипулировании строками:

- всегда используйте безопасные функции для манипулирования строками, в именах этих функций присутствует суффикс *_s* либо префикс *StringCch*. Последние применяют, когда нужно явно проконтролировать усечение строк, в остальных случаях используют безопасные функции с суффиксом *_s* в именах;
- не пользуйтесь небезопасными функциями библиотеки C для манипулирования строками (см. выше). Общее правило таково: не используй-

те функцию, манипулирующую буфером, если она не принимает размер целевого буфера как параметр. Для этой цели библиотека C поддерживает новые функции, такие как *memcpy_s*, *memmove_s*, *wmemcpy_s* и *wmemmove_s*. Они доступны, если определен символ __STDC_WANT_SECURE_LIB__ (в CrtDefs.h он определен по умолчанию, не отменяйте это умолчание);

- используйте преимущества флагов компилятора /GS (http://msdn2.microsoft.com/en-us/library/aa290051(VS.71).aspx) и /RTCs, позволяющих автоматически обнаруживать переполнение буфера;
- не используйте для манипулирования строками методы Kernel32, такие как *lstrcat* и *lstrcpy*;
- в коде приходится манипулировать двумя типами строк. Первый тип - это фрагменты текста программы, включая имена файлов, пути, XML-элементы и атрибуты, а также разделы и параметры реестра. Для работы с такими строками используют функцию *CompareStringOrdinal*, поскольку она не учитывает региональные параметры и потому работает очень быстро. Это — преимущество, поскольку ни регион, ни страна, в которой запущено приложение, не влияет на такие строки. Другой тип строк — это строки, отображаемые в пользовательском интерфейсе. С ними работают при помощи функции **CompareString(Ex)**, которая учитывает региональные параметры при сравнении строк.

Здесь у вас нет выбора: профессиональные разработчики просто не могут использовать в коде небезопасные функции, манипулирующие буфером. Именно по этой причине во всех примерах, приведенных в этой книге, используются безопасные функции из библиотеки C.

Перекодировка строк из Unicode в ANSI и обратно

Windows-функция *MultiByteToWideChar* преобразует мультибайтовые символы строки в «широкобайтовые»;

```
int MultiByteToWideChar(
    UINT uCodePage,
    DWORD dwFlags,
    PCSTR pMultiByteStr,
    int cbMultiByte,
    PWSTR pWideCharStr,
    int cchWideChar);
```

Параметр *uCodePage* задает номер кодовой страницы, связанной с мультибайтовой строкой. Параметр *dwFlags* влияет на преобразование букв с диакритическими знаками. Обычно эти флаги не используются, и *dwFlags* равен 0 (подробнее о допустимых значениях этого флага см. в документации MSDN по ссылке http://msdn2.microsoft.com/en-us/library/ms776413.aspx). Параметр *pMultiByteStr* указывает на преобразуемую строку, а *cchMultiByte*

определяет ее длину в байтах. Функция самостоятельно определяет длину строки, если *cchMultiByte* равен -1.

Unicode-версия строки, полученная в результате преобразования, записывается в буфер по адресу указанному в *pWideCharStr*. Максимальный размер этого буфера (в символах) задается в параметре *cchWideChar*. Если он равен 0, функция ничего не преобразует, а просто возвращает размер буфера, необходимого для сохранения результата преобразования (с учетом концевого символа «\0»), Обычно конверсия мультибайтовой строки в ее Unicode-эквивалент проходит так:

1. Вызывают *MultiByteToWideChar*, передавая NULL в параметре *pWideCharStr,* 0 в параметре *cchWideChar* и -1 — в параметре *cbMultiByte*.
2. Выделяют блок памяти, достаточный для сохранения преобразованной строки. Его размер получают путем умножения результата предыдущего вызова *MultiByteToWideChar* на *sizeof(wchar_t)*.
3. Снова вызывают *MultiByteToWideChar*, на этот раз передавая адрес выделенного буфера в параметре *pWideCharStr*, а размер буфера, полученный при первом обращении к этой функции, — в параметре *cchWideChar*.
4. Работают с полученной строкой.
5. Освобождают блок памяти, занятый Unicode-строкой.

Обратное преобразование выполняет функция *WideCharToMultiByte*.

```
int WideCharToMultiByte(
    UINT uCodePage,
    DWORD dwFlags,
    PCWSTR pWideCharStr,
    int cchWideChar,
    PSTR pMultiByteStr,
    int cbMultiByte,
    PCSTR pDefaultChar,
    PB00L pfUsedDefaultChar);
```

Она очень похожа на *MultiByteToWideChar*. И опять *uCodePage* определяет кодовую страницу для строки — результата преобразования. Дополнительный контроль над процессом преобразования дает параметр *dwFlags*. Его флаги влияют на символы с диакритическими знаками и на символы, которые система не может преобразовать. Такой уровень контроля обычно не нужен, и *dwFlags* приравнивается 0.

Параметр *pWideChar* указывает адрес преобразуемой строки, а *cchWideChar* задает ее длину в символах. Функция сама определяет длину исходной строки, если *cchWideChar* равен -1.

Мультибайтовый вариант строки, полученный в результате преобразования, записывается в буфер, на который указывает *pMultiByteStr*. Параметр *cchMultiByte* определяет максимальный размер этого буфера в байтах. Передав нулевое значение в *cchMultiByte*, вы заставите функцию сообщить размер буфера, требуемого для записи результата. Обычно

конверсия широкобайтовой строки в мультибайтовую проходит в той же последовательности, что и при обратном преобразовании, за одним исключением: сразу возвращается размер буфера (в байтах) для хранения результатов преобразования.

Очевидно, вы заметили, что *WideCharToMultiByte* принимает на два параметра больше, чем *MultiByteToWideChar*, это *pDefaultChar* и *pfUsedDefaultChar*. Функция *WideCharToMultiByte* использует их, только если встречает широкий символ, не представленный в кодовой странице, на которую ссылается *uCodePage*. Если его преобразование невозможно, функция берет символ, на который указывает *pDefaultChar*. Если этот параметр равен NULL (как обычно и бывает), функция использует системный символ по умолчанию. Таким символом обычно служит знак вопроса, что при операциях с именами файлов очень опасно, поскольку он является и символом подстановки.

Параметр *pfUsedDefaultChar* указывает на переменную типа BOOL, которую функция устанавливает как TRUE, если хоть один символ из широкосимвольной строки не преобразован в свой мультибайтовый эквивалент. Если же все символы преобразованы успешно, функция устанавливает переменную как FALSE. Обычно вы передаете NULL в этом параметре.

Подробнее эти функции и их применение описаны в документации Platform SDK.

Экспорт DLL-функций для работы с ANSI и Unicode

Эти две функции позволяют легко создавать ANSI- и Unicode-версии других функций, работающих со строками. Например, у вас есть DLL, содержащая функцию, которая переставляет все символы строки в обратном порядке. Unicode-версию этой функции можно было бы написать следующим образом.

```
BOOL StringReverseW(PWSTR pWideCharStr, DWORD cchLength) {

  // получаем указатель на последний символ в строке
  PWSTR pEndOfStr = pWideCharStr + wcsnlen_s(pWideCharStr , cchLength) - 1;
  wchar_t cCharT;
  // повторяем, пока не дойдем до середины строки
  while (pWideCharStr < pEndOfStr) {
    // записываем символ во временную переменную
    cCharT = *pWideCharStr;

    // помещаем последний символ на место первого
    *pWideCharStr = *pEndOfStr;

    // копируем символ из временной переменной на место,
    *pEndOfStr = cCharT;
```

```
    // продвигаеися на 1 символ вправо.
    pWideCharStr++;

    // продвигаемся на 1 символ влево.
    pEndOfStr--;
  }

  // символы в строке переставлены, возвращаем управление.
  return(TRUE);
}
```

ANSI-версию этой функции можно написать так, чтобы она вообще ничем не занималась, а просто преобразовывала ANSI-строку в Unicode, передавала ее в функцию *StringReverseW* и конвертировала обращенную строку снова в ANSI. Тогда функция должна выглядеть примерно так:

```
BOOL StringReverseA(PSTR pMultiByteStr, DWORD cchLength) {
    PV(STR pWideCharStr;
    int nLenOfWideCharStr;
    BOOL fOk = FALSE;

    // вычисляем количество символов, необходимых
    // для хранения широкосимвольной версии строки.
    nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,
        pMultiByteStr, cchLength, NULL, 0);

    // Выделяем память из стандартной кучи процесса,
    // достаточную для хранения широкосимвольной строки.
    // Не эабудьте, что MultiByteToWideChar возвращает
    // количество символов, а не байтов, поэтому мы должны
    // умножить это число на размер широкого символа.
    pWideCharStr = (PWSTR)HeapAlloc(GetProcessHeap(), 0,
        nLenOfWideCharStr * sizeof(wchar_t));

    if (pWideCharStr == NULL)
        return(fOk);

    // преобразуем мультибайтовую строку в широкосимвольную
    MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, cchLength,
        pWideCharStr, nLenOfWideCharStr);

    // вызываем широкосимвольную версию этой функции
    // для выполнения настоящей работы
    fOk = 8tringReverseW(pWideCharStr, cChLength);

    if (fOk) {
        // преобразуем широкосимвольную строку
        // обратно в мультибайтовую.
```

```
        WideCharToMultiByte(CP_ACP, 0, pWideCharStr, cchLength,
            pMultiByteStr, (int)strlen(pMultiByteStr), NULL, NULL);
    }

    // освобождаем память, выделенную под широкобайтовую строку.
    HeapFree(GetProcessHeap(), 0, pWideCharStr);

    return(fOk);
}
```

И, наконец, в заголовочном файле, поставляемом вместе с DLL, прототипы этих функций были бы такими:

```
BOOL StringReverseW(PWSTR pWideCharStr,  DWORD cchLength);
BOOL StringReverseA(PSTR  pMultiByteStr, DWORD cchLength);

#ifdef UNICODE
#define StringReverse StringReverseW #else
#define StringReverse StringReverseA #endif // !UNICODE
```

Определяем формат текста (ANSI или Unicode)

Блокнот Windows позволяет открывать и создавать как Unicode-, так и ANSI-файлы (см. диалог сохранения файла на рис. 2-5).

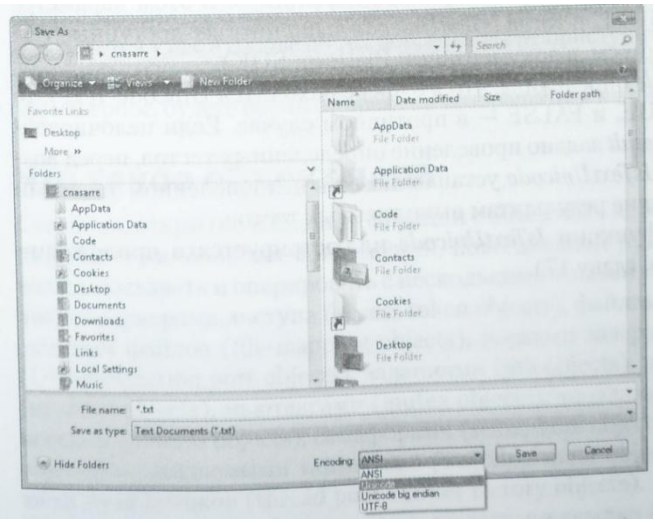


Рис. 2-5. *Диалог сохранения файла в Блокноте Windows Vista*

Для многих обрабатывающих текстовые файлы приложений, к числу которых относятся и компиляторы, была бы полезной возможность определения формата открытого текстового файла (ANSI или Unicode). В этом поможет функция *IsTextUnicode*, поддерживаемая AdvApi32.dll и объявленная в WmBase.h:

```
BOOL IsTextUnicode(CONST PVOID pvBuffer, int cb, PINT pResult);
```

Проблема с текстовыми файлами в том, что их содержимое весьма разнообразно и нет четких правил, регламентирующих его. Поэтому чрезвычайно сложно определить, содержит ли файл ANSI- или Unicode-символы. *IsTextUnicode* определяет тип содержимого буфера, используя ряд статистических и аналитических методов. Как сказано выше, точных критериев для этого не существует, следовательно, *IsTextUnicode* может ошибаться.

Первый параметр этой функции, *pvBuffer*, задает адрес анализируемого буфера. Используется void-указатель, поскольку при вызове функции не известно, находятся ли в массиве ANSI- или Unicode-символы.

Второй параметр, *cb*, задает число байтов буфера, на который указывает *pvBuffer*. И в этом случае тип содержимого буфера заранее не известно, поэтому значение *cb* выражается в байтах, а не символах. Заметьте, что указывать размер целого буфера не обязательно. Естественно, чем больше байтов проверит *IsTextUnicode*, тем точнее будет ее результат.

Третий параметр, *pResult*, представляет адрес целого числа, который необходимо инициализировать перед вызовом *IsTextUnicode*, чтобы указать этой функции, какую проверку она должна выполнить. В этом параметре можно передавать NULL, и тогда *IsTextUnicode* выполнит все доступные тесты (подробнее об этом см. в документации Platform SDK).

Если *IsTextUnicode* определила, что в буфере находится Unicode-текст, то она возвращает TRUE, и FALSE — в противном случае. Если целочисленным параметром *pRezult* задано проведение определенных тестов, перед возвратом управления *IsTextUnicode* устанавливает в целочисленном значении биты, соответствующие результатам выполненных тестов.

Использование функции *IsTextUnicode* иллюстрируется в приложении примере *FileRev* (см. главу 17).

Оглавление

ГЛАВА 3

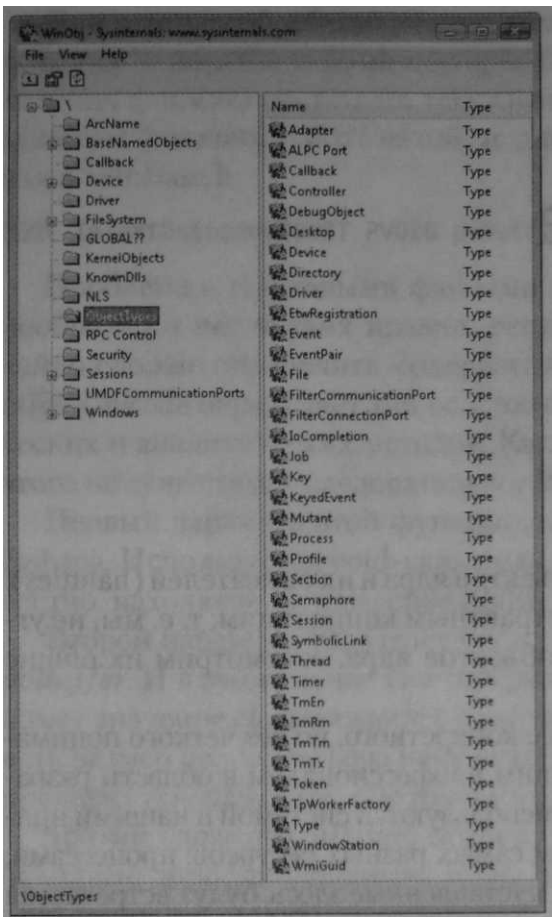
Объекты ядра

Изучение Windows API мы начнем с объектов ядра и их описателей (handles). Эта глава посвящена сравнительно абстрактным концепциям, т. е. мы, не углубляясь в специфику тех или иных объектов ядра, рассмотрим их общие свойства.

Я бы предпочел начать с чего-то более конкретного, но без четкого понимания объектов ядра вам не стать настоящим профессионалом в области разработки Windows-программ. Эти объекты используются системой и нашими приложениями для управления множеством самых разных ресурсов: процессами, потоками, файлами и т. д. Концепции, представленные здесь, будут встречаться на протяжении всей книги. Однако я прекрасно понимаю, что часть материалов не уляжется у вас в голове до тех пор, пока вы не приступите к работе с объектами ядра, используя реальные функции. И при чтении последующих глав книги вы, наверное, будете время от времени возвращаться к этой главе.

Что такое объект ядра

Создание, открытие и прочие операции с объектами ядра станут для вас, как разработчика Windows-приложений, повседневной рутиной. Система позволяет создавать и оперировать с несколькими типами таких объектов, в том числе: маркерами доступа (access token objects), файлами (file objects), проекциями файлов (file-mapping objects), портами завершения ввода-вывода (I/O completion port objects), заданиями (job objects), почтовыми ящиками (mailslot objects), мьютексами (mutex objects), каналами (pipe objects), процессами (process objects), семафорами (semaphore objects), потоками (thread objects) и ожидаемыми таймерами (waitable timer objects), а также фабриками пула потоков (thread pool worker factory objects). Бесплатная утилита WinObj от Sysinternals (ее можно скачать по ссылке http://www.microsoft.com/technet/sysintemak/utilities/winobj.mspx) позволяет просматривать списки типов объектов ядра (см. пример на следующей странице). Эту утилиту следует запускать из проводника из-под администраторской учетной записи.



Эти объекты создаются Windows-функциями. Например, *CreateFileMapping* заставляет систему сформировать объект «проекция файла», связанный с соответствующим объектом «секция» (это можно увидеть с помощью WinObj). Каждый объект ядра — на самом деле просто блок памяти, выделенный ядром и доступный только ему. Этот блок представляет собой структуру данных, в элементах которой содержится информация об объекте. Некоторые элементы (дескриптор защиты, счетчик числа пользователей и др .) присутствуют во всех объектах , но бóльшая их часть специфична для объектов конкретного типа. Например, у объекта «процесс» есть идентификатор, базовый приоритет и код завершения, а у объекта «файл» — смещение в байтах, режим разделения и режим открытия.

Поскольку структуры объектов ядра доступны только ядру, приложение не может самостоятельно найти эти структуры в памяти и напрямую модифицировать их содержимое. Такое ограничение Майкрософт ввела намеренно, чтобы ни одна программа не нарушила целостность структур объектов ядра. Это же ограничение позволяет Майкрософт вводить, убирать или изменять элементы структур, не нарушая работы каких-либо приложений.

Но вот вопрос: если мы не можем напрямую модифицировать эти структуры, то как же наши приложения оперируют объектами ядра? Ответ в том, что в Windows предусмотрен набор функций, обрабатывающих структуры

объектов ядра по строго определенным правилам. Мы получаем доступ к объектам ядра только через эти функции. Когда вы вызываете функцию, создающую объект ядра, она возвращает описатель, идентифицирующий созданный объект. Описатель следует рассматривать как «непрозрачное» значение, которое может быть использовано любым потоком вашего процесса. Описатель представляет собой 32- (в 32-разрядных Windows-процессах) или 64-разрядное (в 64-разрядных Windows-процессах) значение. Этот описатель вы передаете Windows-функциям, сообщая системе, какой объект ядра вас интересует. Но об описателях мы поговорим позже (в этой главе).

Для большей надежности, операционной системы Майкрософт сделала так, чтобы значения описателей зависели от конкретного процесса. Поэтому, если вы передадите такое значение (с помощью какого-либо механизма межпроцессной связи) потоку другого процесса, любой вызов из того процесса со значением описателя, полученного в вашем процессе, даст ошибку. Но не волнуйтесь, в конце главы мы рассмотрим три механизма корректного использования несколькими процессами одного объекта ядра.

Учет пользователей объектов ядра

Объекты ядра принадлежат ядру, а не процессу. Иначе говоря, если ваш процесс вызывает функцию, создающую объект ядра, а затем завершается, объект ядра может быть не разрушен. В большинстве случаев такой объект все же разрушается; но если созданный вами объект ядра используется другим процессом, ядро запретит разрушение объекта до тех пор, пока от него не откажется и тот процесс.

Ядру известно, сколько процессов использует конкретный объект ядра, поскольку в каждом объекте есть счетчик числа его пользователей. Этот счетчик — один из элементов данных, общих для всех типов объектов ядра. В момент создания объекта счетчику присваивается 1. Когда к существующему объекту ядра обращается другой процесс, счетчик увеличивается на 1. А когда какой-то процесс завершается, счетчики всех используемых им объектов ядра автоматически уменьшаются на 1. Как только счетчик какого-либо объекта обнуляется, ядро уничтожает этот объект.

Защита

Объекты ядра можно защитить дескриптором защиты (security descriptor), который описывает, кто создал объект и кто имеет права на доступ к нему. Дескрипторы защиты обычно используют при написании серверных приложений. Однако в Windows Vista это свойство объектов ядра доступно и клиентским приложениям, обладающим собственными пространствами имен (подробнее см. ниже в этой, главе).

Почти все функции, создающие объекты ядра, принимают указатель на структуру SECURITY_ATTRIBUTES как аргумент, например:

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwNaxinumSizeLow,
    PCTSTR pszName);
```

Большинство приложений вместо этого аргумента передает NULL и создает объект с защитой по умолчанию. Такая защита подразумевает, что создатель объекта и любой член группы администраторов получают к нему полный доступ, а все прочие к объекту не допускаются. Однако вы можете создать и инициализировать структуру SECURITY_ATTRIBUTES, а затем передать ее адрес. Она выглядит так:

```
typedef struct _SECURITY_ATTRIBUTES {
    DWORD nLength;
    LPVOID lpSecurityDescriptor;
    BOOL bInheritHandle;
} SECURITY_ATTRIBUTES;
```

Хотя структура называется SECURITY_ATTRIBUTES, лишь один ее элемент имеет отношение к защите — *lpSecurityDescriptor*. Если надо ограничить доступ к созданному вами объекту ядра, создайте дескриптор защиты и инициализируйте структуру SECURITY_ATTRIBUTES следующим образом:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);                    // используется для выяснения версий
sa.lpSecurityDescriptor = pSD;              // адрес инициализированной SD
sa.bInheritHandle = FALSE;                  // об этом позже
HANDLE hFileMapping = CreateFileMapping(INVALID_HANDLE_VALUE, &sa,
    PA6E_READWRITE, 0, 1024, TEXT("MyFileMapping"));
```

Рассмотрение элемента *bInheritHandle* я отложу до раздела о наследовании, так как этот элемент не имеет ничего общего с защитой.

Желая получить доступ к существующему объекту ядра (вместо того чтобы создавать новый), укажите, какие операции вы намерены проводить над объектом. Например, если бы я захотел считывать данные из существующей проекции файла, то вызвал бы функцию *OpenFileMapping* таким образом:

```
HANDLE hFileMapping = OpenFileMapping(FILE_MAP_READ, FALSE,
    TEXT("MyFileMapping"));
```

Передавая FILE_MAP_READ первым параметром в функцию *OpenFileMapping*, я сообщаю, что, как только мне предоставят доступ к проекции файла, я буду считывать из нее данные. Функция *OpenFileMapping*, прежде чем вернуть действительный описатель, проверяем тип защиты объекта: Если меня, как зарегистрировавшегося пользователя, допускают к существую-

щему объекту ядра «проекция файла», *OpenFileMapping* возвращает действительный описатель. Но если мне отказывают в доступе, *OpenFileMapping* возвращает NULL, а вызов *GetLastError* дает код ошибки 5 (или ERROR_ ACCESS_DENIED). Но опять же, в основной массе приложений защиту не используют, и поэтому я больше не буду задерживаться на этой теме.

Хотя в большинстве приложений нет нужды беспокоиться о защите, многие функции Windows требуют, чтобы вы передавали им информацию о нужном уровне защиты. Некоторые приложения, написанные для прежних версий Windows, в Windows Vista толком не работают из-за того, что при их реализации не было уделено должного внимания защите.

Представьте, что при запуске приложение считывает данные из какого-то раздела реестра. Чтобы делать это корректно, оно должно вызывать функцию *RegOpenKeyEx*, передавая значение KEY_QUERY_VALUE, которое разрешает операцию чтения в указанном разделе.

Однако многие приложения для версий Windows, предшествующих Windows 2000, создавались без учета вопросов, связанных с защитой. Поскольку эти версии Windows не защищают свой реестр, разработчики часто вызывали *RegOpenKeyEx* со значением KEY_ALL__ACCESS. Так проще и не надо ломать голову над тем, какой уровень доступа требуется па самом деле. Но проблема в том, что раздел реестра может быть доступен для чтения, и блокирован для записи. В Windows Vista вызов *RegOpenKeyEx* со значением KEY_ALL_ACCESS заканчивается неудачно, и без соответствующего контроля ошибок приложение может повести себя совершенно непредсказуемо.

Если бы разработчик хоть немного подумал о защите и поменял значение KEY_ALL_ACCESS на KEY_QUERY_VALUE (только-то и всего!), его продукт мог бы работать в обеих операционных системах.

Пренебрежение флагами, определяющими уровень доступа, — одна из самых крупных ошибок, совершаемых разработчиками. Правильное их использование позволило бы легко переносить многие приложения в новые версии Windows. Необходимо также учитывать, что в каждой новой версии Windows появляются ограничения, отсутствовавшие в прежней версии. Так, в Windows Vista добавлена функция контроля пользовательских учетных записей (User Account Control, UAC). По соображениям безопасности UAC заставляет систему исполнять приложения в ограниченном контексте защиты, даже если текущий пользователь обладает администраторскими правами. Подробнее о UAC — в главе 4.

Кроме объектов ядра ваша программа может использовать объекты других типов — меню, окна, указатели мыши, кисти и шрифты. Они относятся к объектам User или GDI. Новичок в программировании для Windows может запутаться, пытаясь отличить объекты User или GDI от объектов ядра. Как узнать, например, чьим объектом — User или ядра — является данный значок? Выяснить, не принадлежит ли объект ядру, проще всего так: проанализировать функцию, создающую объект. Практически у всех функций, создающих

объекты ядра, есть параметр, позволяющий указать атрибуты защиты, — как у *CreateFileMapping*.

В то же время у функций, создающих объекты User или GDI, нет параметра типа PSECURITY_ATTRIBUTES, и пример тому — функция *CreateIcon*:

```
HICON CreateIcon(
    HINSTANCE hinst,
    int nWidth,
    int nHeight,
    BYTE cPlanes,
    BYTE cBitsPixel,
    CONST BYTE *pbANDbits,
    CONST BYTE *pbXORbits);
```

Подробнее об объектах GDI и User, а также об их мониторинге см. в статье MSDN, доступной по ссылке http://msdn.microsoft.com/msdnmag/issues/03/01/GDILeaks.

Таблица описателей объектов ядра

При инициализации процесса система создает в нем таблицу описателей, используемую только для объектов ядра. Сведения о структуре этой таблицы и управлении ею незадокументированы. Вообще-то я воздерживаюсь от рассмотрения недокументированных частей операционных систем. Но в данном случае стоит сделать исключение — квалифицированный Windows-программист, на мой взгляд, должен понимать, как устроена таблица описателей в процессе. Поскольку информация о таблице описателей незадокументирована, я не ручаюсь за ее стопроцентную достоверность, к тому же эта таблица по-разному реализуется в разных версиях Windows. Таким образом, следующие разделы помогут понять, что представляет собой таблица описателей, но вот что система действительно делает с ней — этот вопрос я оставляю открытым.

В таблице 3-1 показано, как выглядит таблица описателей, принадлежащая процессу. Как видите, это просто массив структур данных. Каждая структура содержит указатель на какой-нибудь объект ядра, маску доступа и некоторые флаги.

Табл. 3-1. Структура таблицы описателей, принадлежащей процессу

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x????????	0x????????	0x????????
2	0x????????	0x????????	0x????????
…	…	…	…

Создание объекта ядра

Когда процесс инициализируется в первый раз, таблица описателей еще пуста. Но стоит одному из его потоков вызвать функцию, создающую объект ядра (например, *CreateFileMapping*), как ядро выделяет для этого объекта блок памяти и инициализирует его; далее ядро просматривает таблицу описателей, принадлежащую данному процессу, и отыскивает свободную запись. Поскольку таблица еще пуста (см. табл. 3-1), ядро обнаруживает структуру с индексом 1 и инициализирует ее. Указатель устанавливается на внутренний адрес структуры данных объекта, маска доступа — на доступ без ограничений и, наконец, определяется последний компонент — флаги. (О флагах мы поговорим позже, в разделе о наследовании.)

Вот некоторые функции, создающие объекты ядра (список ни в коей мере на полноту не претендует):

```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    size_t dwStackSize,
    LPTHREAD_START_ROUTINE pfnStartAddress,
    PVOID pvParam,
    DWORD dwCreationFlags,
    PDWORD pdwThreadId);

HANDLE CreateFile(
    PCTSTR pszFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreatlonDisposltlon,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);

HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);

HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTES psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);
```

Все функции, создающие объекты ядра, возвращают описатели, которые привязаны к конкретному процессу и могут быть использованы в любом потоке данного процесса. Значение описателя представляет собой индекс в таблице описателей, принадлежащей процессу, и таким образом идентифицирует место, где хранится информация, связанная с объектом ядра. Чтобы получить это значение, значение описателя следует разделить на 4 (или сдвинуть вправо на два разряда, игнорируя два последних бита, зарезервированные Windows для собственных нужд). Вот поэтому при отладке своего приложения и просмотре фактического значения описателя объекта ядра вы и видите такие малые величины: 4, 8 и т. д. Но помните, что физическое содержимое описателей не задокументировано и может быть изменено.

Всякий раз, когда вы вызываете функцию, принимающую описатель объекта ядра как аргумент, вы передаете ей значение, возвращенное одной из Create-функций. При этом функция смотрит в таблицу описателей, принадлежащую вашему процессу, и считывает адрес нужного объекта ядра.

Если вы передаете неверный индекс (описатель), функция завершается с ошибкой и *GetLastError* возвращает 6 (ERROR_INVALID_HANDLE). Это связано с тем, что на самом деле описатели представляют собой индексы в таблице, их значения привязаны к конкретному процессу и недействительны в других процессах.

Если вызов функции, создающей объект ядра, оказывается неудачен, то обычно возвращается 0 (NULL). Такая ситуация возможна только при острой нехватке памяти или при наличии проблем с защитой. К сожалению, отдельные функции возвращают в таких случаях не 0, а -1 (INVALID_HANDLE_VALUE, определено в WinBase.h). Например, если *CreateFile* не сможет открыть указанный файл, она вернет именно INVALID_HANDLE_VALUE, а не 0. Будьте очень осторожны при проверке значения, возвращаемого функцией, которая создает объект ядра. Так, для *CreateMutex* проверка на INVALID_HANDLE_VALUE бессмысленна:

```
HANDLE hHutex = CreateMutex(...);
if (hMutex == INVALID_HANDLE_VALUE) {
  // Этот код никогда не будет выполнен, так как
  // при ошибке CreateMutex возвращает NULL.
}
```

Точно так же бессмыслен и следующий код:

```
HANDLE hFile = CreateFile(…);
if (hFile == NULL) {
  // и этот код никогда не будет выполнен, так как
  // при ошибке CreateFile возвращает INVALID HANDLE VALUE (-1).
}
```

Закрытие объекта ядра

Независимо от того, как именно вы создали объект ядра, по окончании работы с ним его нужно закрыть вызовом *CloseHandle*:

```
BOOL CloseHandle(HANDLE hobject);
```

Эта функция сначала проверяет таблицу описателей, принадлежащую вызывающему процессу, чтобы убедиться, идентифицирует ли переданный ей индекс (описатель) объект, к которому этот процесс действительно имеет доступ. Если переданный индекс правилен, система получает адрес структуры данных объекта и уменьшает в этой структуре счетчик числа пользователей; как только счетчик обнулится, ядро удалит объект из памяти.

Если же описатель неверен, происходит одно из двух. В нормальном режиме выполнения процесса *CloseHandle* возвращает FALSE, а *GetLastError* — код ERROR_INVALID_HANDLE. Но при выполнении процесса в режиме отладки система просто уведомляет отладчик об ошибке генерацией исключения 0xC0000008 («An invalid handle was specified»).

Перед самым возвратом управления *CloseHandle* удаляет соответствующую запись из таблицы описателей: данный описатель теперь недействителен в вашем процессе, и использовать его нельзя. При этом запись удаляется независимо от того, разрушен объект ядра или нет! После вызова *CloseHandle* вы больше не получите доступ к этому объекту ядра; но, если его счетчик не обнулен, объект остается в памяти. Тут все нормально, это означает лишь то, что объект используется другим процессом (или процессами). Когда и остальные процессы завершат свою работу с этим объектом (тоже вызвав *CloseHandle*), он будет разрушен.

> **Примечание.** Обычно после создания объекта ядра его описатель сохраняют в переменной. После вызова функции *CloseHandle* с передачей ей этой переменной в качестве параметра, следует также сбросить переменную, записав в нее **NULL**. Если не сделать этого и по ошибке передать такую переменную Win32-функции, возможны две неожиданные ситуации. Поскольку запись, на которую ссылается эта переменная, уже удалена из таблицы описателей, Windows получит недействительный параметр, а вы — сообщение об ошибке. Другая ситуация еще хуже, поскольку ее трудно «отловить» при отладке. Windows создает новые объекты ядра, используя свободные записи таблицы описателей. Таким образом, в том месте таблицы, на которое указывает ссылка в переменной, не подвергнутой сбросу, уже может быть создан новый объект ядра. В этом случае вы получите ссылку на объект ядра другого или (что еще хуже) того же самого типа, что и закрытый до этого объект. Результат — невосстановимое повреждение состояние приложения.

А вдруг вы забыли вызвать *CloseHandle* — будет ли утечка памяти? И да, и нет. Утечка ресурсов (тех же объектов ядра) вполне вероятна, пока процесс еще исполняется. Однако по завершении процесса операционная система гарантированно освобождает все ресурсы, принадлежавшие этому процессу, и в случае объектов ядра действует так: в момент завершения процесса просматривает его таблицу описателей и закрывает любые открытые описатели. Если при этом счетчик пользователей какого-либо объекта обнуляется, система разрушает его.

Сказанное выше верно для всех объектов, включая объекты GDI и выделенные блоки памяти: система гарантирует, что после завершения процесса не остается ничего, ему принадлежавшего. Простой способ определения утечек памяти, занятой объектами ядра, — использование Диспетчера задач Windows (Task manager). Прежде всего, необходимо добавить на вкладку Processes столбец Handles (это делается в окне Select Process Page Columns, доступном через меню View/Select Columns).

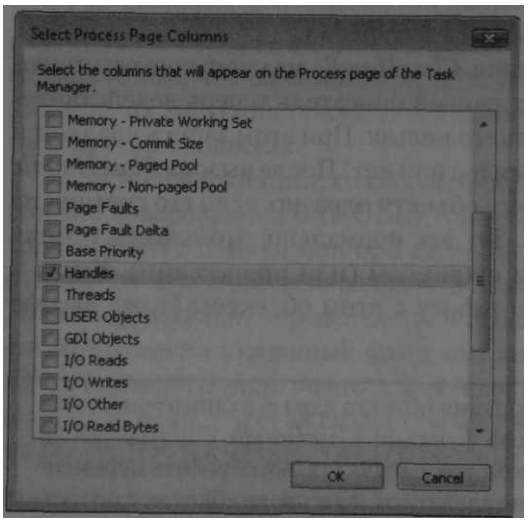


Рис. 3-1. *Добавление столбца Handles в окне Select Process Page Columns*

Это позволит отслеживать число объектов ядра, используемое любым приложением, как показано на рис. 3-2.

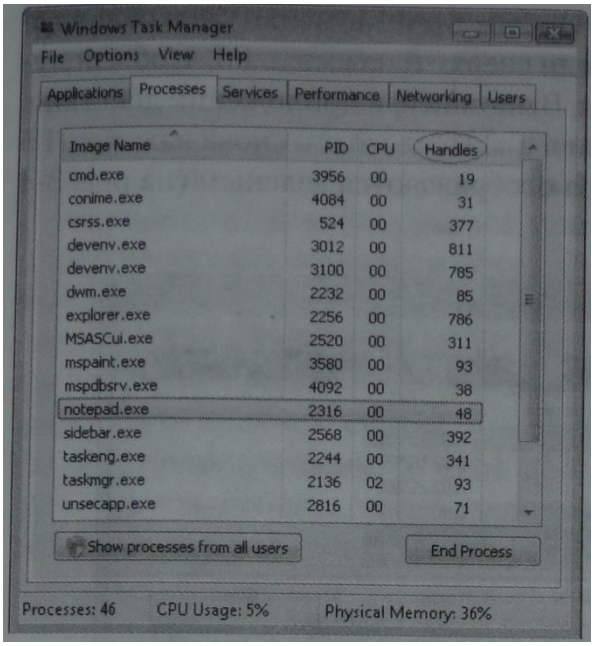


Рис. 3-2. *Подсчет описателей в окне Task Manager*

Число в столбце Handles непрерывно увеличивается. Далее нужно определить, какие объекты ядра не закрывается — эта информация нужна, чтобы воспользоваться бесплатной утилитой Process Explorer от Sysinternals (доступной по ссылке http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ProcessExplorer.mspx). Сначала щелкните правой кнопкой заголовок столбца Handles и выберите все столбцы в окне Select Columns (рис. 3-3).

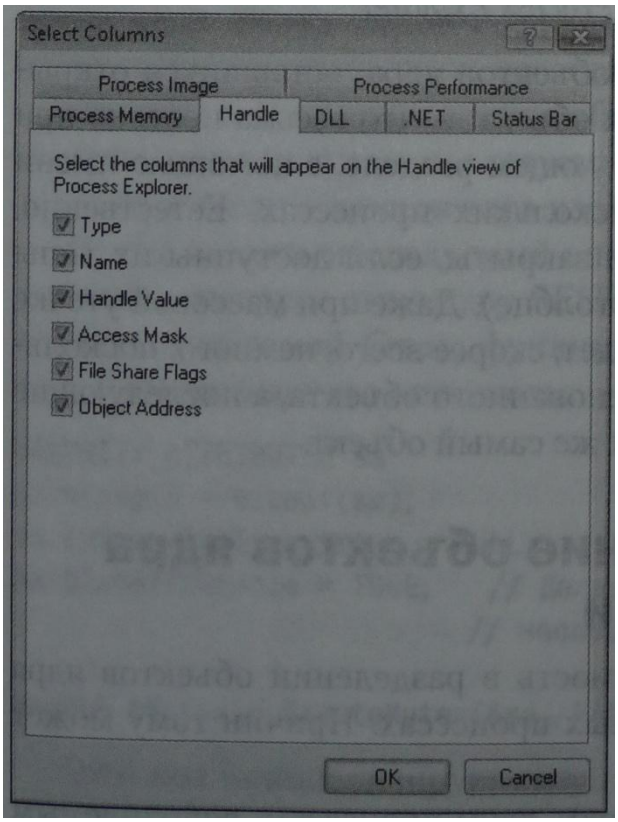


Рис. 3-3. *Настройка вида столбца Handle в Process Explorer*

После этого присвойте параметру Update Speed в меню View значение Paused. Выберите свой процесс на панели сверху и нажмите F5, чтобы получить актуальный список объектов ядра. Выполните в приложении действия, потенциально ведущие к утечке памяти ядра. После этого снова нажмите F5 в Process Explorer. Новые объекты ядра отображаются зеленым (на рис. 3-4 это темно-серый цвет).

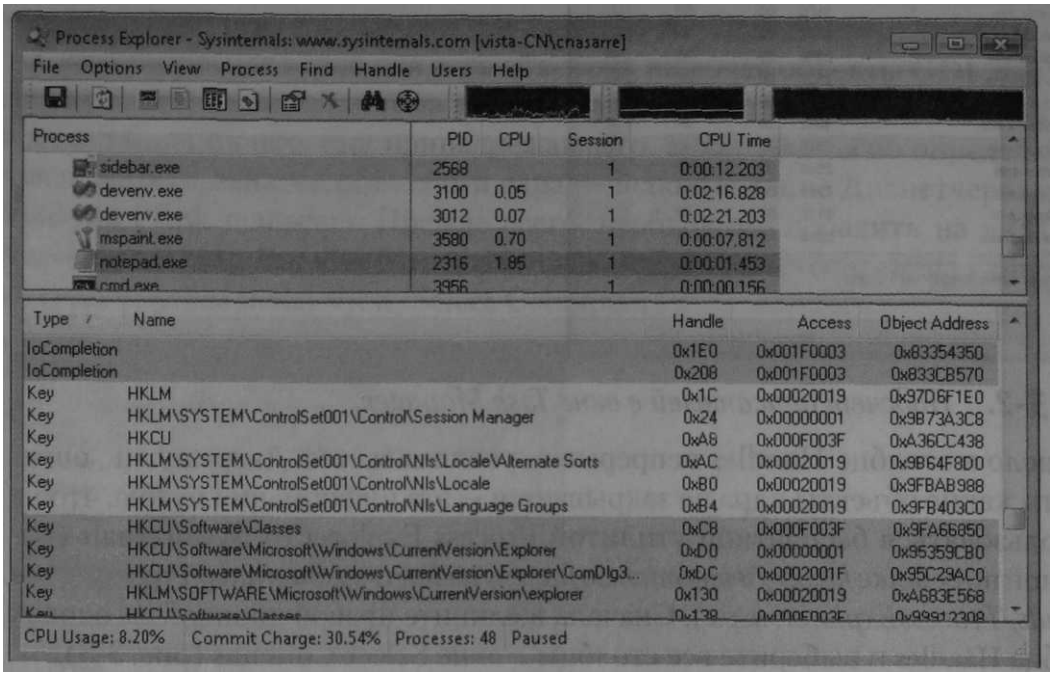


Рис. 3-4. *Поиск новых объектов ядра в Process Explorer*

В первом столбце отображается тип объектов ядра, оставшихся открытыми, а во втором столбце — имена этих объектов, что весьма полезно при поиске утечек. Как будет сказано в следующем разделе, с помощью имени объект ядра можно использовать в нескольких процессах. Естественно, проще найти объекты, которые не были закрыты, если доступны их типы (в первом столбце) и имена (во втором столбце). Даже при массовой утечке памяти отображаемых имен объектов будет, скорее всего, немного, поскольку создается только один экземпляр именованного объекта, а последующие обращения к нему просто открывают тот же самый объект.

Совместное использование объектов ядра несколькими процессами

Время от времени возникает необходимость в разделении объектов ядра между потоками, исполняемыми в разных процессах. Причин тому может быть несколько:

- объекты «проекции файлов» позволяют двум процессам, исполняемым на одной машине, совместно использовать одни и те же блоки данных;

- почтовые ящики и именованные каналы дают возможность программам обмениваться данными с процессами, исполняемыми на других машинах в сети;
- мьютексы, семафоры и события позволяют синхронизировать потоки, исполняемые в разных процессах, чтобы одно приложение могло уведомить другое об окончании той или иной операции.

Но поскольку описатели объектов ядра имеют смысл только в конкретном процессе, разделение объектов ядра между несколькими процессами — задача весьма непростая. У Майкрософт было несколько веских причин сделать описатели «процессно-зависимыми», и самая главная — устойчивость операционной системы к сбоям. Если бы описатели объектов ядра были общесистемными, то один процесс мог бы запросто получить описатель объекта, используемого другим процессом, и устроить в нем (этом процессе) настоящий хаос. Другая причина — защита. Объекты ядра защищены, и процесс, прежде чем оперировать ими, должен запрашивать разрешение на доступ к ним.

Три механизма, позволяющие процессам совместно использовать одни и те же объекты ядра, мы рассмотрим в следующем разделе.

Наследование описателя объекта

Наследование применимо, только когда процессы связаны «родственными» отношениями (родительский-дочерний). Например, родительскому процессу доступен один или несколько описателей объектов ядра, и он решает, породив дочерний процесс, передать ему по наследству доступ к своим объектам ядра. Чтобы такой сценарий наследования сработал, родительский процесс должен выполнить несколько операций.

Во-первых, еще при создании объекта ядра этот процесс должен сообщить системе, что ему нужен наследуемый описатель данного объекта. (Имейте в виду: описатели объектов ядра наследуются, но сами объекты ядра — нет.)

Чтобы создать наследуемый описатель, родительский процесс выделяет и инициализирует структуру SECURITY_ATTRIBUTES, а затем передает ее адрес требуемой Create-функции. Следующий код создает объект-мьютекс и возвращает его описатель:

```
SECURITY_ATTRIBUTES sa;
sa.nLength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;                    // Делаем возвращаеный описатель
                                             // наследуемым.

HANDLE hMutex=CreateMutex(&sa, FALSE, NULL);
```

Этот код инициализирует структуру SECURITY_ATTRIBUTES, указывая, что объект следует создать с защитой по умолчанию и что возвращаемый описатель должен быть наследуемым.

А теперь перейдем к флагам, которые хранятся в таблице описателей, принадлежащей процессу. В каждой ее записи присутствует битовый флаг, сообщающий, является данный описатель наследуемым или нет. Если вы, создавая объект ядра, передадите в параметре типа PSECURITY_ATTRIBUTES значение NULL, то получите ненаследуемый описатель, и этот флаг будет нулевым. А если элемент *bInheritHandle* равен TRUE, флагу присваивается 1.

Допустим, какому-то процессу принадлежит таблица описателей, как в таблице 3-2.

Табл. 3-2. Таблица описателей с двумя действительными записями

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0xF0000000	0x????????	0x00000000
2	0x00000000	(неприменим)	(неприменим)
3	0xF0000010	0x????????	0x00000001

Эта таблица свидетельствует, что данный процесс имеет доступ к двум объектам ядра: описатель 1 (ненаследуемый) и 3 (наследуемый).

Следующий этап — родительский процесс порождает дочерний. Это делается с помощью функции *CreateProcess*.

```
B00L CreateProcess(
    PCTSTR pszApplicationName,
    PTSTR pszCommandLine,
    PSECURITY_ATTRIBUTES psaProcess,
    PSECUWTY_ATTRIBUTES psaThread,
    B00L bInheritHandles,
    DWORD dwCreationFlags,
    PV0ID pvEnvironment,
    PCTSTR pezCurrentDirectory,
    LPSTARTUPINF0 pStartupInfo,
    PPR0CESS_INF0RMATI0N pProcessInformation);
```

Подробно мы рассмотрим эту функцию в следующей главе, а сейчас я хочу лишь обратить ваше внимание на параметр *bInheritHandles*. Создавая процесс, вы обычно передаете в этом параметре FALSE, тем самым сообщая системе, что дочерний процесс не должен наследовать наследуемые описатели, зафиксированные в таблице родительского процесса.

Если же вы передаете TRUE, дочерний процесс наследует описатели родительского. Тогда операционная система создает дочерний процесс, но не дает ему немедленно начать свою работу. Сформировав в нем, как обычно, новую (пустую) таблицу описателей, она считывает таблицу родительского процесса и копирует все ее действительные записи в таблицу дочернего — причем в те же позиции. Последний факт чрезвычайно важен, так как означает, что описатели будут идентичны в обоих процессах (родительском и дочернем).

Помимо копирования записей из таблицы описателей, система увеличивает значения счетчиков соответствующих объектов ядра, поскольку эти объекты теперь используются обоими процессами. Чтобы уничтожить какой-то объект ядра, его описатель должны закрыть (вызовом *CloseHandle*) оба процесса. Кстати, сразу после возврата управления функцией *CreateProcess* родительский процесс может закрыть свой описатель объекта, и это никак не отразится на способности дочернего процесса манипулировать этим объектом.

В таблице 3-3 показано состояние таблицы описателей в дочернем процессе — перед самым началом его исполнения. Как видите, записи 1 и 2 не инициализированы, и поэтому данные описатели неприменимы в дочернем процессе. Однако индекс 3 действительно идентифицирует объект ядра по тому же (что и в родительском) адресу 0xF0000010.

Табл. 3-3. Таблица описателей в дочернем процессе (после того как он унаследовал от родительского один наследуемый описатель)

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x00000000	(неприменим)	(неприменим)
2	0x00000000	(неприменим)	(неприменим)
3	0xF0000010	0x????????	0x00000001

Как будет сказано в главе 13, содержимое объектов ядра хранится в адресном пространстве ядра, общем для всех процессов. В 32-разрядных системах память ядра находится в диапазоне 0x80000000-0xFFFFFFFF, а в 64-разрядных — 0x00000400’00000000-0xFFFFFFFF’FFFFFFFF. При этом маска доступа и флаги в родительском и дочернем процессах тоже идентичны. Так что, если дочерний процесс в свою очередь породит новый («внука» по отношению к исходному родительскому) вызовом *CreateProcess* с параметром *bInheritHandles*=TRUE, «внук» унаследует данный описатель объекта ядра с теми же значением, правами доступа и флагами, а счетчик числа пользователей этого объекта ядра вновь увеличится на 1.

Наследуются только описатели объектов, существующие на момент создания дочернего процесса. Если родительский процесс создаст после этого новые объекты ядра с наследуемыми описателями, то эти описатели будут уже недоступны дочернему процессу.

Для наследования описателей объектов характерно одно очень странное свойство: дочерний процесс не имеет ни малейшего понятия, что он унаследовал какие-то описатели. Поэтому наследование описателей объектов ядра полезно, только когда дочерний процесс сообщает, что при его создании родительским процессом он ожидает доступа к какому-нибудь объекту ядра. Тут надо заметить, что обычно родительское и дочернее приложения пишутся одной фирмой, но в принципе дочернее приложение может написать и сторонняя фирма, если в этой программе задокументировано, чего именно она ждет от родительского процесса.

Для этого в дочерний процесс обычно передают значение ожидаемого им описателя объекта ядра как аргумент в командной строке. Инициализирующий код дочернего процесса анализирует командную строку (чаще всего вызовом *_stscanf_s*), извлекает из нее значение описателя, и дочерний процесс получает неограниченный доступ к объекту. При этом механизм наследования срабатывает только потому, что значение описателя общего объекта ядра в родительском и дочернем процессах одинаково, — и именно по этой причине родительский процесс может передать значение описателя как аргумент в командной строке;

Для наследственной передачи описателя объекта ядра от родительского процесса дочернему, конечно же, годятся и другие формы межпроцессной связи. Один из приемов заключается в том, что родительский процесс дожидается окончания инициализации дочернего (через функцию *WaitForInputdle*, рассматриваемую в главе 9), а затем посылает (синхронно или асинхронно) сообщение окну, созданному потоком дочернего процесса.

Еще один прием: родительский процесс добавляет в свой блок переменных окружения новую переменную. Она должна быть «узнаваема» дочерним процессом и содержать значение наследуемого описателя объекта ядра. Далее родительский процесс создает дочерний, тот наследует переменные окружения родительского процесса и, вызвав *GetEnvironmentVariable*, получает нужный описатель. Такой прием особенно хорош, когда дочерний процесс тоже порождает процессы, — ведь все переменные окружения вновь наследуются. Особый случай наследования дочерним процессом консоли родительского процесса рассматривается в статье «Базы знаний Майкрософт», доступной по ссылке http://support.microsoft.com/kb/190351.

Изменение флагов описателя

Иногда встречаются ситуации, в которых родительский процесс создает объект ядра с наследуемым описателем, а затем порождает два дочерних процесса. Но наследуемый описатель нужен только одному из них. Иначе говоря, время от времени возникает необходимость контролировать, какой из дочерних процессов наследует описатели объектов ядра. Для этого модифицируйте флаг наследования, связанный с описателем, вызовом SetHandleInformation:

```
BOOL SetHandleInformation(
    HANDLE hObject,
    DW0RD  dwMask,
    DW0R0  dwFlags);
```

Как видите, эта функция принимает три параметра. Первый (hObject) идентифицирует допустимый описатель. Второй (dwMask) сообщает функции, какой флаг (или флаги) вы хотите изменить. На сегодняшний день с каждым описателем связано два флага:

```
#define HANDLE_FLAG_INHERIT    0x00000001
```

```
#define HANDLE_FLAG_PROTECT_FROM_CLOSE 0x00000002
```

Чтобы изменить сразу все флаги объекта, нужно объединить их побитовой операцией OR.

И, наконец, третий параметр функции *SetHandleInformation* — *dwFlags* — указывает, в какое именно состояние следует перевести флаги. Например, чтобы установить флаг наследования для описателя объекта ядра:

```
SetHandleInformation(hObj, HANDLE_FLAG_INHERIT, HANDLE_FLAG_INHERIT);
```

а чтобы сбросить этот флаг:

```
SetHandleInformation(hObj, HANDLE_FLAG_INHERIT, 0);
```

Флаг HANDLE_FLAG_PROTECT_FROM_CLOSE сообщает системе, что данный описатель закрывать нельзя:

```
SetHandleInformation(hObj, HANDLE_FLAG_PROTECT_FROM_CLOSE,
    HANDLE_FLAG_PROTECT_FROM_CLOSE);
CloseHandle(hObj);                          // Генерируется исключение
```

Если какой-нибудь поток попытается закрыть защищенный описатель, *CloseHandle* приведет к исключению. Необходимость в такой защите возникает очень редко. Однако этот флаг весьма полезен, когда процесс порождает дочерний, а тот в свою очередь — еще один процесс. При этом родительский процесс может ожидать, что его «внук» унаследует определенный описатель объекта, переданный дочернему. Но тут вполне возможно, что дочерний процесс, прежде чем породить новый процесс, закрывает нужный описатель. Тогда родительский процесс теряет связь с «внуком», поскольку тот не унаследовал требуемый объект ядра. Защитив описатель от закрытия, вы исправите ситуацию, и «внук» унаследует предназначенный ему объект.

У этого подхода, впрочем, есть один недостаток. Дочерний процесс, вызвав:

```
SetHandleInformation(hobj, HANDLE_FLAG_PROTECT_FROM_CLOSE, 0);
CloseHandle(hObj);
```

может сбросить флаг HANDLE_FLAG_PROTECT_FROM_CLOSE и закрыть затем соответствующий описатель. Родительский процесс ставит на то, что дочерний не исполнит этот код. Но одновременно он ставит и на то, что дочерний процесс породит ему «внука», поэтому в целом ставки не слишком рискованны.

Для полноты картины стоит, пожалуй, упомянуть и функцию *GetHandleInformation*:

```
B00L GetHandleInformation(
    HANDLE h0bject,
    PDW0RD pdwFlags);
```

Эта функция возвращает текущие флаги для заданного описателя в переменной типа DWORD, на которую указывает *pdwFlags*. Чтобы проверить, является ли описатель наследуемым, сделайте так:

```
DWORD dwFlags;
GetHandleInformation(hObj, &dwFlags);
BOOL fHandleIsInheritable = (0 != (dwFlags & HANDLE_FLAG_INHERIT));
```

Именованные объекты

Второй способ, позволяющий нескольким процессам совместно использовать одни и те же объекты ядра, связан с именованием этих объектов. Именование допускают многие (но не все) объекты ядра. Например, следующие функции создают именованные объекты ядра:

```
HANDLE CreateMutex(
    PSECURITY_ATTRIBUTES psa,
    B00L bInitialOwner,
    PCTSTR pszName);

HANDLE CreateEvent(
    PSECURITY_ATTRIBUTES psa,
    B00L bManualReset,
    B00L bInitialState,
    PCTSTR pszName);

HANDLE CreateSemaphore(
    PS_CORITY_ATTRIBUTES psa,
    LONG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);

HANDLE CreateWaitableTimer(
    PSECURITY_ATTRIBUTES psa,
    B00L bManualReset,
    PCTSTR pszName);

HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD flProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);

HANDLE CreateJobObject(
    PSECURITY_ATTRIBUTES psa,
    PCT8TR pszName);
```

Последний параметр, *pszName*, у всех этих функций одинаков. Передавая в нем NULL, вы создаете безымянный (анонимный) объект ядра. В этом случае вы можете разделять объект между процессами либо через наследование

(см. предыдущий раздел), либо с помощью *DuplicateHandle* (см. следующий раздел). А чтобы разделять объект по имени, вы должны присвоить ему какое-нибудь имя.

Тогда вместо NULL в параметре *pszName* нужно передать адрес строки с именем, завершаемой нулевым символом. Имя может быть длиной до MAX_PATH знаков (это значение определено как 260). К сожалению, Майкрософт ничего не сообщает о правилах именования объектов ядра. Например, создавая объект с именем JerfObj, вы никак не застрахованы от того, что в системе еще нет объекта ядра с таким именем. И что хуже, все эти объекты делят единое пространство имен. Из-за этого следующий вызов *CreateSemaphore* будет всегда возвращать NULL:

```
HANDLE hMutex = CreateMutex(NULL, FALSE, TEXT("JeffObj"));
HANDLE hSem = CreateSemaphore(NULL, 1, 1, TEXT("JeffObj"));
DWORD dwErrorCode = GetLastError();
```

После выполнения этого фрагмента значение *dwErrorCode* будет равно 6 (ERROR_INVALID_HANDLE). Полученный код ошибки не слишком вразумителен, но другого не дано.

Итак, вы узнали, как давать объектам имена, а теперь посмотрим, как организовать совместное использование именованных объектов. Допустим, после запуска процесса A вызывается функция:

```
HANDLE hMutexProcessA = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

Этот вызов заставляет систему создать новенький, как с иголочки, объект ядра «мьютекс» и присвоить ему имя JeffMutex. Заметьте, что описатель *hMutexProcessA* в процессе A не является наследуемым, — он и не должен быть таковым при простом именовании объектов.

Спустя какое-то время некий процесс порождает процесс B. Необязательно, чтобы последний был дочерним от процесса A; он может быть порожден Explorer или любым другим приложением. (В этом, кстати, и состоит преимущество механизма именования объектов перед наследованием.) Когда процесс B приступает к работе, исполняется код:

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, TEXT("JeffMutex"));
```

При этом вызове система сначала проверяет, не существует ли уже объект ядра с таким именем. Если да, то ядро проверяет тип этого объекта. Поскольку мы пытаемся создать мьютекс и его имя тоже JeffMutex, система проверяет права доступа вызывающего процесса к этому объекту. Если у него есть все права, в таблице описателей, принадлежащей процессу B, создается новая запись, указывающая на существующий объект ядра. Если же вызывающий процесс не имеет полных прав на доступ к объекту или если типы двух объектов с одинаковыми именами не совпадают, вызов *CreateMutex* заканчивается неудачно и возвращается NULL.

> **Примечание**. Функции, создающие объекты ядра (например, **CreateSemaphore**), всегда возвращают описатели, обеспечивающие полный доступ. Чтобы ограничить доступ к созданному объекту, следует использовать расширенные версии этих функций (с суффиксом *Ex* в именах). Расширенные функции принимают дополнительный DWORD-параметр *dwDesiredAccess*. Например, можно разрешить либо запретить вызов *ReleaseSemaphore* на семафоре, соответственно указывая либо не указывая SEMAPHORE_MODIFY_STATE при вызове *CreateSemaphoreEx*. Подробнее о правах, поддерживаемых объектами ядра, см. в документации Windows SDK (http://msdn2.microsoft.com/en-us/library/ms686670.aspx)

Однако, хотя процесс B успешно вызвал *CreateMutex*, новый объект-мьютекс он не создал. Вместо этого он получил свой описатель существующего объекта-мьютекса. Счетчик объекта, конечно же, увеличился на 1, и теперь этот объект не разрушится, пока его описатели не закроют оба процесса — A и B. Заметьте, что значения описателей объекта в обоих процессах скорее всего разные, но так и должно быть: каждый процесс будет оперировать с данным объектом ядра, используя свой описатель.

> **Примечание**. Разделяя объекты ядра по именам, помните об одной крайне важной вещи, вызывая *CreateMutex*, процесс B передает ей атрибуты защиты и второй параметр. Так вот, эти параметры игнорируются, если объект с указанным именем уже существует! Приложение может определить, что оно делает: создает новый объект ядра или просто открывает уже существующий, — вызвав *GetLastError* сразу же после вызова одной из *Create*-функций:

```
HANDLE hMutex = CreateMutex(&sa, FALSE, TEXT("JeffObj"));
if (GetLastError() == ERROR_ALREADY_EXISTS) {
  // Открыт описатель существующего объекта;
  // sa.lpSecurityDescriptor и второй параметр
  // (FALSE) игнорируются.
} else {
  // Создан совершенно новый объект;
  // sa.lpSecurityDescriptor и второй параметр
  // (FALSE) используются при создании объекта.
}
```

Есть и другой способ разделения объектов по именам. Вместо вызова *Crease*-функции процесс может обратиться к одной из следующих *Open*-функций:

```
HANDLE OpenMutex(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

```
HANDLE OpenEvent(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenSemaphore(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenWaitableTimer(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenFileMapping(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);

HANDLE OpenJobObject(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

Заметьте: все эти функции имеют один прототип. Последний параметр, *pszName*, определяет имя объекта ядра. В нем нельзя передать NULL — только адрес строки с нулевым символом в конце. Эти функции просматривают единое пространство имен объектов ядра, пытаясь найти совпадение. Если объекта ядра с указанным именем нет, функции возвращают NULL, а *GetLastError* — код 2 (ERROR_FILE_NOT_FOUND). Но если объект ядра с заданным именем существует и если его тип идентичен тому, что вы указали, система проверяет, разрешен ли к данному объекту доступ запрошенного вида (через параметр *dwDesiredAccess*). Если такой вид доступа разрешен, таблица описателей в вызывающем процессе обновляется, и счетчик числа пользователей объекта возрастает на 1. Если вы присвоили параметру *bInheritHandle* значение TRUE, то получите наследуемый описатель.

Главное отличие между вызовом *Create*- и *Open*-функций в том, что при отсутствии указанного объекта *Create*-функция создает его, а *Open*-функция просто уведомляет об ошибке.

Как я уже говорил, Майкрософт ничего не сообщает о правилах именования объектов ядра. Но представьте себе, что пользователь запускает две программы от разных компаний и каждая программа пытается создать объект с именем «MyObject». Ничего хорошего из этого не выйдет. Чтобы избежать такой ситуации, я бы посоветовал создавать GUID и использовать его строковое представление как имя объекта.

Именованные объекты часто применяются для того, чтобы не допустить запуска нескольких экземпляров одного приложения. Для этого вы просто вызываете одну из *Create*-функций в своей функции *_tmain* или *_tWinMain* и создаете некий именованный объект. Какой именно — не имеет ни малейшего значения. Сразу после *Create*-функции вы должны вызвать *GetLastError*. Если она вернет ERROR_ALREADY_EXISTS, значит, один экземпляр Вашего приложения уже выполняется и новый его экземпляр можно закрыть. Вот фрагмент кода, иллюстрирующий этот прием:

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine,
    int nCmdShow) {
    HANDLE h = CreateMutex(NULL, FALSE,
        TEXT("{FA531CC1-0497-11d3-A180-00105A276C3E}"));
    if (GetLastError() == ERROR_ALREADY_EXISTS) {
        // экземпляр этого приложения уже выполняется
        // закрываем объект и возвращаем управление
        CloseHandle(h);
        return(0);
    }

    // запущен первый экземпляр этого приложения
    …
    // перед выходом закрываем объект
    CloseHandle(h);
    return(0);
}
```

Пространства имен Terminal Services

Службы терминалов (Terminal Services) несколько меняют описанный выше сценарий. Если на машине работают Terminal Services, существует множество пространств имен для объектов ядра. Объекты, которые должны быть доступны всем клиентам, используют одно глобальное пространство имен. (Такие объекты, как правило, связаны с сервисами, предоставляемыми клиентским программам.) В каждом клиентском сеансе формируется свое пространство имен, чтобы исключить конфликты между несколькими сеансами, в которых запускается одно и то же приложение. Ни из какого сеанса нельзя получить доступ к объектам другого сеанса, даже если у их объектов идентичные имена. Этот сценарий актуален не только для серверов, но и для клиентов, поскольку такие механизмы, как Remote Desktop и Fast User Switching также основаны на сеансах служб терминалов.

> **Примечание**. Первый (неинтерактивный) сеанс создается службами, когда в систему еще не вошел ни один пользователь. В Windows Vista, в отличие от предыдущих версий Windows, после входа пользователя приложения запускаются в новом сеансе, отличном от Session 0 (сеанс 0

принадлежит исключительно службам). Это позволит надежнее изолировать ключевые компоненты системы, которые обычно работают с высоким уровнем привилегий, от вредоносных программ, которые может запустить неосторожный пользователь.

Разработчикам служб следует учесть, что требование исполнения служб и их клиентских приложений в разных сеансах влияет на правила именования общих объектов ядра. Теперь объекты ядра, которые должны быть доступны пользовательским приложениям, необходимо создавать в глобальном пространстве имен. Сходные проблемы возникают при написании служб, взаимодействующих с приложениями, запущенными в разных сеансах, созданных посредством Fast User Switching. Подробнее об изоляции сеанса Session 0 и ее последствиях для разработчиков служб см. в статье «Impact of Session 0 Isolation on Services and Drivers in Windows Vista» (http://www.microsoft.com/whdc/system/vista/services.aspx)

Выяснить, в каком из сеансов Terminal Services работает ваш процесс, поможет функция *ProcessIdToSessionId* (объявленная в WinBase.h и доступная в библиотеке kernel32.dll):

```
DWORD processID = GetCurrentProcessId();
DWORD sessionID;
if (ProcessIdToSessionId(processID, &sessionID)) {
    tprintf(
        TEXT("Process '%u' runsi*ferroinal Services session '%u'"),
        processID, sessionID);
} else {
    // Вызов ProcessIdToSessionId закончится неудачей, если
    // у вас нет права доступа к процессу, ID которого вы передаете.
    // Здесь такой опасности нет, поскольку используется ID
    // процесса-владельца.
tprintf(
    TEXT("Unable to get Terminal Services session ID for process '%u'"),
    processID);
}
```

Именованные объекты ядра, относящиеся к какому-либо сервису, всегда находятся в глобальном пространстве имен, а аналогичный объект, связанный с приложением, Terminal Server по умолчанию помещает в пространство имен клиентского сеанса. Однако и его можно перевести в глобальное пространство имен, поставив перед именем объекта префикс <<Global\», как в примере ниже.

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, TEXT("Global\\MyName");
```

Если вы хотите явно указать, что объект ядра должен находиться в пространстве имен клиентского сеанса, используйте префикс <Local\»:

```
HANDLE h = CreateEvent(NULL, FALSE, FALSE, TEXT("Local\\MyName"));
```

Майкрософт рассматривает префиксы Global и Local как зарезервированные ключевые слова, которые не должны встречаться в самих именах объектов (если они не находятся в отдельном пространстве имен). К числу таких слов Майкрософт относит и Session, хотя на сегодняшний день оно не связано ни с какой функциОнальностью. Например, допустимо использовать Session\<*ID_текущего_сеанса*>\, но создавать объекты с именами, содержащими префикс *Session*, запрещено: соответствующий вызов заканчивается неудачей, а *GetLastError* возвращает ERROR_ACCESS_DENIED.

Примечание. Эти ключевые слова чувствительные к регистру.

Закрытые пространства имен

Доступ к объекту ядра можно защитить, передав при создании объекта указатель на структуру SECURITY_ATTRIBUTES. Однако в версиях Windows, предшествующих Vista, защитить общий именованный объект от несанкционированного доступа было невозможно. Любому процессу, даже обладающему минимальными привилегиями, было разрешено создавать именованные объекты. Так, в предыдущая программа-пример использовала именованный мьютекс, чтобы определить, работает ли она. При этом можно без труда написать приложение, которое создает объект ядра с тем же самым именем. Если такая программа стартует перед singleton-приложением, то запустить последнее уже не удастся: оно тут же завершится, так сочтет, что в системе уже работает экземпляр такого приложения. На этом приеме основаны некоторые DoS-атаки. Заметьте, что DoS-атаки по неименованным объектам ядра невозможны, поэтому в приложениях часто используются именно такие объекты, несмотря на то, что они не могут одновременно использоваться разными процессами.

Чтобы защитить объекты ядра, которые создает ваше приложение, от конфликтов имен и атак злоумышленников, следует, определив собственный префикс, создать закрытое пространство имен. Серверный процесс, ответственный за создание объектов ядра, определяет *дескриптор границ* (boundary descriptor), защищающий имя самого пространства имен.

Использование закрытых пространств имен для более безопасной реализации singleton-сценариев демонстрируется на примере приложения «Singleton» (файл 03-Singleton.exe, см.также листинг Singleton.cpp ниже). После запуска этого приложения открывается окно, показанное на рис. 3-5.

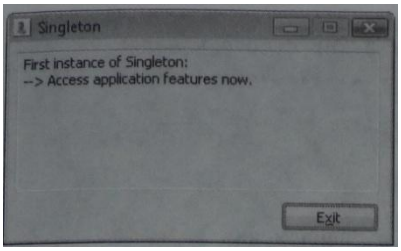


Рис. 3-5. *Первый экземпляр приложения Singleton*

Если, не завершая первого экземпляра Singleton, попытаться запустить второй, откроется окно с сообщением о том, что обнаружен работающий экземпляр программы (рис. 3-6).

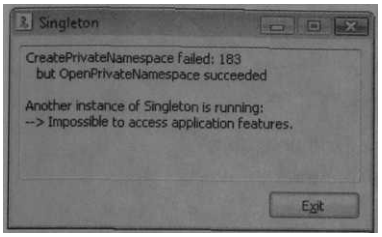


Рис. 3-6. *Результат попытки запуска второго экземпляра Singleton*

В листинге ниже на примере функции **CheckInstances** показано, как создать границу, связать с ней *идентификатор защиты* (security identifier, SID) группы Local Administrators и создать (или открыть) закрытое пространство имен, имя которого будет использовано в качествен префикса для имена объекта ядра (мьютекса). Дескриптор границы получает имя, и, что важнее, SID связанной с ним привилегированной группы. Таким образом Windows гарантирует, что только приложения, работающие в контексте пользователя из этой группы получат доступ к этому пространству имен и созданным в нем объектам ядра, имена которых содержат его имя в виде префикса.

Предположим, что вредоносная программа с низким уровнем привилегий, похитив имя и SID, создает идентичный дескриптор границы. Ее попытка создать или открыть закрытое пространство имен, защищенное привилегированной учетной записью, закончится неудачей, а *GetLastError* при этом вернет ERROR_ACCESS_DENIED. Если же у вредоносной программы окажется достаточно привилегий, чтобы проникнуть в закрытое пространство имен, это будет уже не важно, поскольку с такими привилегиями она сможет причинить куда больший вред, чем простой захват имени объекта ядра.

```
Singleton.cpp
/*****************************************************************
Module: Singleton.cpp
```

```
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************///

#include "stdafx.h"
#include "resource.h"

#include "..\CommonFiles\CmnHdr.h"          /* См. Приложение A. */
#include <windowsx.h>
#include <Sddl.h>                           // Необходимо для работы с SID
#include <tchar.h>
#include <strsafe.h>

//////////////////////////////////////////////////////////////////////

// Главный диалог
HWND       g_hDlg;

// Мьютекс, граница и пространство имен, необходимые для
// обнаружения существующего экземпляра программы.
HANDLE     g_hSingleton = NULL;
HANDLE     g_hBoundary = NULL;
HANOLE     g_hNamespace = NULL;

// переменная, отслеживающая состояние пространства имен
BOOL       g_bNamespaceOpened = FALSE;

// имена границы и закрытого пространства имен
PCTSTR     g_szBoundary = TEXT("3-Boundary");
PCTSTR     g_szNamespace = TEXT("3-Namespace");

#define DETAILS_CTRL GetDlgItem(g_hDlg, IDC_EDIT_DETAILS)

//////////////////////////////////////////////////////////////////////

// Adds a string to the "Details" edit control
void AddText(PCTSTR pszFormat, …) {

   va_list argList;
   va_start(argList, pszFormat);

   TCHAR 8z[20 * 1024];

   Edit_GetText(DETAILS_CTRL, sz, _countof(sz));
   _vstprintf_s(
       _tcschr(sz, TEXT('\0')), _countof(sz) - _tcslen(sz),
       pszFormat, argList);
Edit_SetText(DETAILS_CTRL, sz);
```

```
    va_end(argList);
}

//////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)
{
 switch (id) {
     case IDOK:
     case IDCANCEL:
        // Пользователь щелкнул кнопку Exit
        // либо нажал ESCAPE.
        EndDialog(hwnd, id);
        break;
 }
}

//////////////////////////////////////////////////////////////////////

void CheckInstances()
{
 // создаем дескриптор границы
 g_hBoundary = CreateBoundaryOescriptor(g_szBoundary, 0);

 // Создаем SID, соответствующий локальной группе Administrators
 BYTE localAdminSID[SECURITY_MAX_SID_SIZE];
 PSID pLocalAdminSID = &localAdminSIO;
 DWORD cbSID = sizeof(localAdminSID);
 if (!CreateWellKnownSid(
     WinBuiltinAdministratorsSid, NULL, pLocalAdminSID, &cbSID)) {
     AddText(TEXT("AddSIDToBoundaryDescriptor failed: Xu\r\n"),
        GetLastErrorO);
     return;
 }

 // Связываем полученный SID с дескриптором границы. В итоге
 // только приложения, запущенные администратором, смогут
 // получить доступ к объектам ядра из этого пространства имен,
 if (!AddSIDToBoundaryDescriptor(&g_hBoundary, pLocalAdminSID)) {
     AddText(TEXT("AddSIDToBoundaryDescriptor failed: Xu\r\n"), GetLastEr-
     ror());
     return;
 }

 // Создать пространство имен, доступное только локальному администратору
SECURITY_ATTRIBUTES sa;
```

```
 sa.nLength = sizeof(sa);
 sa.bInheritHandle = FALSE;
 if (!ConvertStringSecurityDescriptorToSecurityDescriptor(
     TEXT("D:(A;;GA;;;BA)"),
     SDDL_REVISION_1, &sa.lpSecurityDescriptor, NULL)) {
     AddText(TEXT("Security Descriptor creation failed: %u\r\n"),
     GetLastError());
     return;
 }

 g_hNamespace =
     CreatePrivateNamespace(&sa, g_hBoundary, g_szNamespace);

 // не забываем освободить память, занятую дескриптором
 LocalFree(sa.lpSecurityDescriptor);

 // проверим результат создания закрытого пространства имен
 DWORD dwLastError = GetLastError();
 if (g_hNamespace == NULL) {
     // Если в доступе отказано, делать нечего: этот код работает
     // только под учетной записью локального администратора,
     if (dwLastError == ERROR_ACCESS_DENIED) {
         AddText(TEXT("Access denied when creating the namespace.\r\n"));
         AddText(TEXT(" You must be running as Administrator.\r\n\r\n"));
         return;

     } else {
         if (dwLastError == ERROR_ALREADY_EXISTS) {
         // Если другой экземпляр уже создал пространство имен,
         // следует открыть его.
             AddText(TEXT("CreatePrivateNamespace failed: %u\r\n"), dwLastEr-
             ror);
             g_hNamespace = OpenPrivateNamespace(g_hBoundary, g_sz Namespace);
             if (g_hNamespace == NULL) {
                 AddText(TEXT("   and OpenPrivateNamespace failed: %u\r\n"),
                    dwLastError);
                 return;
             } else {
                 g_bNamespaceOpened = TRUE;
                 AddText(TEXT("    but OpenPrivateNamespace suc-
                 ceeded\r\n\r\n"));
             }
} else {
AddText(TEXT("Unexpected error occurred: %u\r\n\r\n"),dwLastError);
return;
```

```
         }
      }
   }

   // Попытаемся создать мьютекс
   // в закрытом пространстве имен.
   TCHAR szMutexName[64];
   StringCchPrintf(szMutexName, _countof(szMutexName), TEXT("%s\\Xs"),
      g_szNamespace, TEXT("Singleton"));

   g_hSingleton = CreateMutex(NULL, FALSE, szMutexName);
   if (GetLastError() == ERROR_ALREADY_EXISTS) {
      // Уже есть экземпляр этого singleton-объекта
      AddText(TEXT("Another instance of Singleton is running:\r\n"));
      AddText(TEXT("--> Impossible to access application features.\r\n"));
   } else {
      // singleton-еще не создан
      AddText(TEXT("First instance of Singleton:\r\n"));
      AddText(TEXT("--> Access application features now.\r\n"));
   }
}

//////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hwnd, HWN0 hwndFocus, LPARAM lParam) {

   chSETDLGICONS(hwnd, IDI_SINGLETON);

   // переменная для описателя главного диалога
   g_hDlg = hwnd;

   // проверить, нет ли других экземпляров программы
   CheckInstances();

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
      chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
   }
```

```
    return(FALSE);
}

//////////////////////////////////////////////////////////////////////

int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE   hPrevInstance,
                       LPTSTR      lpCmdLine,
                       int         nCmdShow)
{
  UNREFERENCED_PARAMETER(hPrevInstance);
  UNREFERENCED_PARAHETER(lpCmdLine);

  // показываем главное окно
  DialogBox(hInstance, MAKEINTRESOURCE(IDD_SIN6LETQN), NULL, DlgJ>roc);

  // Не забываем очистить и освободить ресурсы ядра
  if (g_hSingleton != NULL) {
      CloseHandle(g_hSingleton);
  }

  if (g_hNamespace ! = NULL) {
      if (g_bNamespaceOpened) {  // Пространство имен открыто
          ClosePrivateNamespace(g_hNamespace, 0);
      } else { // Пространство имен создано
          ClosePrivateNamespace(g_hNamespace, PRIVATE_NAMESPACE_FLAG_ DESTROY);
      }
  }

  if (g_hBoundary != NULL) {
      DeleteBoundaryDescriptor(g_hBoundary);
  }

  return(0);
}

//////////////////////////// End of File ////////////////////////////////
```

Рассмотрим, как работает функция *CkeckInstances*. Во-первых, для создания дескриптора границы необходима строка, идентифицирующая закрытое пространство имен. Она передается как первый параметры следующей функции:

```
HANDLE CreateBoundaryDescrlptor(
    PCTSTR pszName,
    0W0RD dwFlags);
```

В текущих версиях Windows второй параметр не используется, поэтому в нем следует передавать 0. заметьте, что сигнатура функции говорит о том, что она возвращает описатель объекта ядра, однако это не так. На самом деле возвращается указатель на структуру пользовательского режима, содержащую определение границы. Поэтому значение, которое вернула эта функция, ни в коем случае не следует передавать функции *CloseHandle*, оно должно передаваться *DeleteBoundaryDescriptor*.

Далее следует связать SID привилегированной пользовательской группы, от имени которой должно запускаться приложение, с дескриптором границы. Это делается так:

```
B00L AddSIDToBoundaryDescriptor(
    HANDLE* phBoundaryDescriptor,
    PSID pRequiredSid);
```

В нашем примере SID локальной группы администраторов создается вызовом AllocateAndInitializeSid с параметрами SECURITY_BUILTTN_DOMAIN_RID и DOMAIN_ALIAS_RID_ADMINS (эти параметры описывают группу). Список стандартных групп определен в заголовочном файле WinNT.h.

Описатель дескриптора границы передается как второй параметр при вызове следующей функции для создания закрытого пространства имен:

```
HANDLE CreatePrivateNamespace(
    PSECURITY_ATTRIBUTES psa,
    PV0ID pvBoundaryDescriptor,
    PCTSTR pszAliasPrefix);
```

Первый параметр SECUR_TY_ATTRIBUTES, указывает ли разрешить приложению, вызывающему *OpenPrivateNamespace*, доступ к пространству имен для создания или открытия в нем объектов. Параметры управления доступом здесь те же, что и для каталога файловой системы. Этот механизм представляет своего рода фильтр для доступа к пространству имен. Добавленный к дескриптору границы SID определяет, кому разрешено пересекать границу и работать с этим пространством имен. В нашем примере SECURITY_ATTRIBUTE создается вызовом функции *ConvertStringSecurityDescriptorToSecurityDescriptor*, принимающей в качестве первого параметра строку со сложным синтаксисом. Описание синтаксиса дескрипторов защиты доступно ПО ссылкам http://msdn2.microsoft.com/en-us/library/aa374928.aspx и http://msdn2.microsoft.com/en-us/library/aa379602.aspx.

Тип *pvBoundaryDescriptor* — PVOID, несмотря на то, что *CreateBoundaryDescriptor* возвращает HANDLE, даже в Майкрософт его называют псевдоописателем. Строка с префиксом для имен объектов ядра, которые будут создаваться в вашем пространстве имен, передается в третьем параметре. При попытке создания уже существующего пространства имен *CreatePrivateNamespace* возвращает NULL, а *GetLastError* — ERROR_ALREADY_EXISTS. В этом случае следует открыть существующее пространство имен вызовом следующей

функции:

```
HANDLE OpenPrivateNamespace(
    PVOID pvBoundaryDescriptor,
    PCTSTR pszAliasPrefix);
```

Заметьте, что значения HANDLE, возвращаемее функциями *CreatePrivateNamespace* и *OpenPrivateNamespace*, не являются описателями объектов ядра. Эти псевдоописатели закрывают вызовом *ClosePrivateNamespace*:

```
BOOLEAN ClosePrivateNamespace(
    HANDLE hNamespace,
    DWORD dwFlags);
```

Если после закрытия ваше пространство имен должно полностью исчезнуть, следует передать во втором параметре флаг PRIVATE_NAMESPACE_FLAG_DESTROY и 0 в противном случае. Граница уничтожается после завершения процесса либо при вызове функции *DeleteBoundaryDescriptor* принимающей единственный параметр — псевдоописатель дескриптора границы. Не следует закрывать пространство имен, пока используются расположенные в нем объекты ядра. В противном случае можно будет создавать объекты ядра с теми же именами, если воссоздать это пространство имен с идентичными границами, то есть снова откроется уязвимость для DoS-атак.

Итак, подведем итоги. Закрытое пространство имен — это просто каталог, в котором вы создаете объекты ядра. Как и у любого каталога, у пространства имен есть связанный с ним дескриптор защиты, который создается при вызове *CreatePrivateNamespace*. Однако, в отличие от каталогов файловой системы, у пространства имен нет ни родительского каталога, ни имени, имя ему заменяет дескриптор границы. Именно поэтому объекты ядра, созданные с префиксом из закрытого пространства имен, отображаются утилитой Process Explorer с префиксом «…\» вместо «пространство_имен\». Префикс «…\» скрывает информацию, усиливая защиту приложения от потенциальных атак взломщиков. Имя, то есть псевдоним, данный вами закрытому пространству имен, видим только внутри процесса. Другие процессы (и тот же самый процесс) могут, открыв это пространство имен, назначить ему другой псевдоним.

При создании вложенных каталогов файловой системы проверяется наличие соответствующих прав доступа к родительскому каталогу. При создании пространства имен проверяются права на доступ через границу: в маркере доступа текущего потока должны быть все SID, связанные с дескриптором границы.

Дублирование описателей объектов

Последний Механизм совместного использования объектов ядра несколькими процессами требует функции *DuplicateHandle*:

```
BOOL DuplicateHandle(
```

```
HANDLE hSourceProcessHandle,
HANDLE hSourceHandle,
HANDLE hTargetProcessHandle,
PHANDLE phTargetHandle,
DWORD dwDesiredAccess,
BOOL bInheritHandle,
DWORD dwOptions);
```

Говоря по-простому, эта функция берет запись в таблице описателей одного процесса и создает ее копию в таблице другого. *DuplicateHandle* принимает несколько параметров, но на самом деле весьма незамысловата. Обычно ее применение требует наличия в системе трех разных процессов.

Первый и третий параметры функции *DuplicateHandle*, *hSourceProcessHandle* и *hTargetProcessHandle*, представляют собой описатели объектов ядра, специфичные для вызывающего процесса. Кроме того, эти параметры должны идентифицировать именно процессы — функция завершится с ошибкой, если вы передадите описатели на объекты ядра любого другого типа. Подробнее объекты ядра «процессы» мы обсудим в главе 4, а сейчас вам достаточно знать только одно: объект ядра «процесс» создается при каждой инициации в системе нового процесса.

Второй параметр, *hSourceHandle,* — описатель объекта ядра любого типа. Однако его значение специфично не для процесса, вызывающего *DuplicateHandle*, а для того, на который указывает описатель *hSourceProcessHandle*. Параметр *phTargetHandle* — это адрес переменной типа HANDLE, в которой возвращается индекс записи с копией описателя из процесса-источника. Значение возвращаемого описателя специфично для процесса, определяемого параметром *hTargetProcessHandle*.

Предпоследние два параметра *DuplicateHandle* позволяют задать маску доступа и флаг наследования, устанавливаемые для данного описателя в процессе-приемнике. И, наконец, параметр *dwOptions* может быть 0 или любой комбинацией двух флагов: DUPLICATE_SAME__ACCESS и DUPLICATE_CLOSE_SOURCE.

Первый флаг подсказывает *DuplicateHandle*: у описателя, получаемого процессом-приемником, должна быть та же маска доступа, что и у описателя в процессе-источнике. Этот флаг заставляет *DuplicateHandle* игнорировать параметр *dwDesiredAccess*.

Второй флаг приводит к закрытию описателя в процессе-источнике. Он позволяет процессам обмениваться объектом ядра как эстафетной палочкой. При этом счетчик объекта не меняется.

Попробуем проиллюстрировать работу функции *DuplicateHandle* на примере. Здесь S — это процесс-источник, имеющий доступ к какому-то объекту ядра, T — это процесс-приемник, который получит доступ к тому же объекту ядра, а C — процесс-катализатор, вызывающий функцию *DuplicateHandle*. Очевидно, вы никогда не станете передавать в *DuplicateHandle* жестко зашитые значения, как это сделал я, просто демонстрируя работу функции. В ре-

альных программах значения описателей хранятся в переменных и, конечно же, именно эти переменные передаются функциям.

Таблица описателей в процессе C (см. таблицу 3-4) содержит два индекса — 1 и 2. Описатель с первым значением идентифицирует объект ядра «процесс S», описатель со вторым значением — объект ядра «процесс T».

Табл. 3-4. Таблица описателей в процессе C

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0xF0000000 (объект ядра процесса S)	0x????????	0x00000000
2	0xF0000010 (объект ядра процесса T)	0x????????	0x00000000

Таблица 3-5 иллюстрирует таблицу описателей в процессе S, содержащую единственную запись со значением описателя, равным 2. Этот описатель может идентифицировать объект ядра любого типа, а не только «процесс».

Табл. 3-5. Таблица описателей в процессе S

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x00000000	(неприменим)	(неприменим)
2	0xF0000020 (объект ядра любого типа)	0x????????	0x00000000

В таблице 3-6 показано, что именно содержит таблица описателей в процессе T перед вызовом процессом C функции *DuplicateHandle*. Как видите, в ней всего одна запись со значением описателя, равным 2, а запись с индексом 1 пока пуста.

Табл. 3-6. Таблица описателей в процессе T перед вызовом DuplicateHandle

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0x00000000	(неприменим)	(неприменим)
2	0xF0000030 (объект ядра любого типа)	0x????????	0x00000000

Если процесс C теперь вызовет *DuplicateHandle* так:

```
DuplicateHandle(1, 2, 2, &h0bj, 0, TRUE, DUPLICATE_SAME__ACCESS);
```

то после вызова изменится только таблица описателей в процессе T (см. таблицу 3-7).

Табл. 3-7. Таблица описателей в процессе T после вызова DuplicateHandle

Индекс	Указатель на блок памяти объекта ядра	Маска доступа (DWORD с набором битовых флагов)	Флаги (DWORD с набором битовых флагов)
1	0xF0000020	0x????????	0x00000001
2	0xF0000030 (объект ядра любого типа)	0x????????	0x00000000

Вторая строка таблицы описателей в процессе S скопирована в первую строку таблицы описателей в процессе T. Функция *DuplicateHandle* присвоила также переменной *hObj* процесса C значение 1 — индекс той строки таблицы в процессе T, в которую занесен новый описатель.

Поскольку функции *DuplicateHandle* передан флаг DUPLICATE_SAME_ACCESS, маска доступа для этого описателя в процессе T идентична маске доступа в процессе S. Кроме того, данный флаг заставляет *DuplicateHandle* проигнорировать параметр *dwDesiredAccess*. Заметьте также, что система установила битовый флаг наследования, так как в параметре *bInheritHandle* функции *DuplicateHandle* мы передали TRUE.

Как и механизм наследования, функция *DuplicateHandle* тоже обладает одной странностью: процесс-приемник никак не уведомляется о том, что он получил доступ к новому объекту ядра. Поэтому процесс C должен каким-то образом сообщить процессу T, что тот имеет теперь доступ к новому объекту; для этого нужно воспользоваться одной из форм межпроцессной связи и передать в процесс T значение описателя в переменной *hObj*. Ясное дело, в данном случае не годится ни командная строка, ни изменение переменных окружения процесса T, поскольку этот процесс уже выполняется. Здесь придется послать сообщение окну или задействовать какой-нибудь другой механизм межпроцессной связи.

Я рассказал вам о функции *DuplicateHandle* в самом общем виде. Надеюсь, вы увидели, насколько она гибка. Но эта функция редко используется в ситуациях, требующих участия трех разных процессов. Обычно ее вызывают применительно к двум процессам. Представьте, что один процесс имеет доступ к объекту, к которому хочет обратиться другой процесс, или что один процесс хочет предоставить другому доступ к «своему» объекту ядра. Например, если процесс S имеет доступ к объекту ядра и вам нужно, чтобы к этому объекту мог обращаться процесс T, используйте *DuplicateHandle* так:

```
// весь приведенный ниже код исполняется процессом S

// создаем объект-мьютекс, доступный процессу S
HANDLE hObjInProcessS = CreateMutex(NULL, FALSE, NULL);

// открываем описатель объекта ядра "процесс T"
HANDLE hProcessT = OpenProcess(PROCESS_ALL_ACCESS, FALSE, dwProcessIdT);
```

```
HANDLE hObjInProcessT;                  // неинициализированный описатель, связанный
                                        // с процессом T
// предоставляем процессу T доступ к объекту-мьютексу
DuplicateHandle(GetCurrentProcess(), hObjInProcessS, hProcessT,
    &hObjInProcessT, 0, FALSE, DUPLICATE_SAME_ACCESS);

// используем какую-нибудь форму межпроцессной связи, чтобы передать

// значение описателя из hObjProcessS в процесс T
…
// связь с процессом T больше не нужна
CloseHandle(hProcessT);
…
// если процессу S не нужен объект-мьютекс, он должен закрыть его
CloseHandle(hObjInProcessS);
```

Вызов *GetCurrentProcess* возвращает псевдоописатель, который всегда идентифицирует вызывающий процесс, в данном случае — процесс S. Как только функция *DuplicateHandle* возвращает управление, *hObjProcessT* становится описателем, связанным с процессом T и идентифицирующим тот же объект, что и описатель *hObjProcessS* (когда на него ссылается код процесса S). При этом процесс S ни в коем случае не должен исполнять следующий код:

```
// Процесс S никогда не должен пытаться исполнять код,
// закрывающий продублированный описатель.
CloseHandle(hObjInProcessT);
```

Если процесс S выполнит этот код, вызов может дать (а может и не дать) ошибку. Он будет успешен, если у процесса S случайно окажется описатель с тем же значением, что и в *hObjProcessT*. При этом неизвестно, какой объект будет закрыт процессом S, и что будет потом — остается только гадать.

Теперь о другом способе применения *DuplicateHandh*. Допустим, некий процесс имеет полный доступ (для чтения и записи) к объекту «проекция файла» и из этого процесса вызывается функция, которая должна обратиться к проекции файла и считать из нее какие-то данные. Так вот, если мы хотим повысить отказоустойчивость приложения, то могли бы с помощью *DuplicateHandle* создать новый описатель существующего объекта и разрешить доступ только для чтения. Потом мы передали бы этот описатель функции, и та уже не смогла бы случайно что-то записать в проекцию файла. Взгляните на код, который иллюстрирует этот пример:

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE,
    LPTSTR szCndLine, int nCmdShow) {
    // создаем объект "проекция файла";
```

```
    // его описатель разрешает доступ как для чтения, так и для записи
    HANDLE hFileMapRW = CreateFileMapping(INVALID_HANDLE_VALUE,
        NULL, PAGE_READWRITE, 0, 10240, NULL);

    // создаем другой описатель на тот же объект;
    // этот описатель разрешает доступ только для чтения
    HANDLE hFileMapR0;
    DuplicateHandle(GetCurrentProcess(), hFileMapRW, GetCurrentProcess(),
        &hFileMapR0, FILE_MAP_READ, FALSE, 0);

    // вызываем функцию, которая не должна ничего записывать в проекцию файла
    ReadFromTheFileMapping(hFileMapR0);

    // закрываем объект "проекция файла", доступный только для чтения
    CloseHandle(hFileMapR0);

    // проекция файла нам по-прежнему полностью доступна через hFileMapRW
    …
    // если проекция файла больше не нужна основному коду, закрываем ее
    CloseHandle(hFileMapRW);
}
```

ЧАСТЬ II

ПРИСТУПАЕМ К РАБОТЕ

ГЛАВА 4

Процессы

Эта глава о том, как система управляет выполняемыми приложениями. Сначала я определю понятие «процесс» и объясню, как система создает объект ядра «процесс». Затем я покажу, как управлять процессом, используя сопоставленный с ним объект ядра. Далее мы обсудим атрибуты (или свойства) процесса и поговорим о нескольких функциях, позволяющих обращаться к этим свойствам и изменять их. Я расскажу также о функциях, которые создают (порождают) в системе дополнительные процессы. Ну и, конечно, описание процессов было бы неполным, если бы я не рассмотрел механизм их завершения. Итак, приступим.

Процесс обычно определяют как экземпляр выполняемой программы, и он состоит из двух компонентов:

- объекта ядра, через который операционная система управляет процессом. Там же хранится статистическая информация о процессе;
- адресного пространства, в котором содержится код и данные всех EXE- и DLL-модулей. Именно в нем находятся области памяти, динамически распределяемой для стеков потоков и других нужд.

Процессы инертны. Чтобы процесс что-нибудь выполнил, в нем нужно создать поток. Именно потоки отвечают за исполнение кода, содержащегося в адресном пространстве процесса. В принципе, один процесс может владеть несколькими потоками, и тогда они «одновременно» исполняют код в адресном пространстве процесса. Для этого каждый поток должен располагать собственным набором регистров процессора и собственным стеком. В каждом процессе есть минимум один поток. При создании процесса система автоматически создает его первый поток, называемый *главным* потоком. Если бы у процесса не было ни одного потока, ему нечего было бы делать на этом свете, и система автоматически уничтожила бы его вместе с выделенным ему адресным пространством.

Чтобы все эти потоки работали, операционная система отводит каждому из них определенное процессорное время. Выделяя потокам отрезки време-

ни (называемые *квантами)* по принципу карусели, она создает тем самым иллюзию одновременного выполнения потоков. Рис. 4-1 иллюстрирует распределение процессорного времени между потоками на машине с одним процессором.

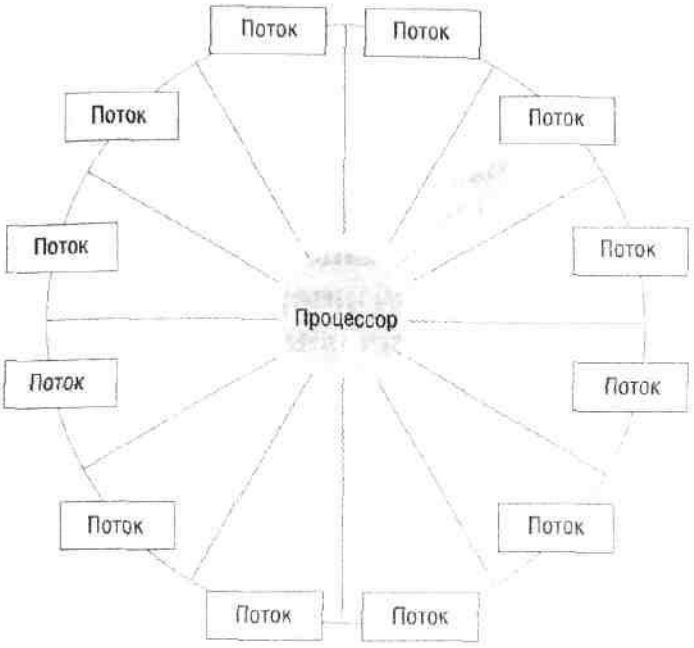


Рис. 4-1. *Операционная система выделяет потокам кванты времени по принципу карусели*

Если в машине установлено более одного процессора, алгоритм работы операционной системы значительно усложняется (в этом случае система стремится сбалансировать нагрузку между процессорами). Ядро Windows автоматически управляет планированием исполнения потоков. Для использования преимуществ многопроцессорности в своих программах разработчикам ничего не требуется делать — система автоматически делает все необходимое для этого. Однако в ваших силах оптимизировать алгоритм вашего приложения, чтобы оно лучше использовало ресурсы многопроцессорных компьютеров.

Ваше первое Windows-приложение

Windows поддерживает два типа приложений: основанные на графическом интерфейсе (graphical user interface, GUI) и консольные (console user interface, GUI). У приложений первого типа внешний интерфейс чисто графический. GUI-приложения создают окна, имеют меню, взаимодействуют с пользователем через диалоговые окна и вообще пользуются всей стандартной «Windows'овской» начинкой. Почти все стандартные программы Windows — Notepad, Calculator, WordPad и др. — являются GUI-приложениями. Приложения консольного типа работают в текстовом режиме: они не формируют окна, не обрабатывают сообщения и не требуют GUI. И хотя кон-

сольные приложения на экране тоже размещаются в окне в нем содержится только текст. Командные процессоры вроде Cmd.exe (в Windows 2000) или Command.com (в Windows 98)- типичные образцы подобных приложений.

Вместе с тем граница между двумя типами приложении весьма условна. Можно, например, создать консольное приложение, способное отображать диалоговые окна. Скажем, в командном процессоре вполне может быть специальная команда, открывающая графическое диалоговое окно со списком команд; вроде мелочь — а избавляет от запоминания лишней информации. В то же время можно создать и GUI-приложение, выводящее текстовые строки в консольное окно. Я сам часто писал такие программы: создав консольное окно, я пересылал в него отладочную информацию, связанную с исполняемым приложением. Но, конечно, графический интерфейс предпочтительнее, чем старомодный текстовый. Как показывает опыт, приложения на основе GUI «дружественнее» к пользователю, а значит и более популярны.

Когда вы создаете проект приложения, Microsoft Visual C++ устанавливает такие ключи для компоновщика, чтобы в исполняемом файле был указан соответствующий тип подсистемы. Для GUI-программ используется ключ /SUBSYSTEM:CONSOLE, а для GUI-приложений — /SUBSYSTEM:WINDOWS. Когда пользователь запускает приложение, загрузчик операционной системы проверяет номер подсистемы, хранящийся в заголовке образа исполняемого файла, и определяет, что это за программа — GUI или GUI. Если номер указывает на приложение последнего типа, загрузчик автоматически создает текстовое консольное окно, а если номер свидетельствует о противоположном — просто загружает программу в память. После того как приложение начинает работать, операционная система больше не интересуется, к какому типу оно относится.

Во всех Windows-приложениях должна быть входная функция, за реализацию которой отвечаете вы. Существует две такие функции:

```
int WINAPI _tWinMain(
    HINSTANCE hlnstanceExe,
    HINSTANCE,
    PTSTR pszCmdLine,
    int nCmdShow);

int _tmain(
    int argc,
    TCHAR *argv[],
    TCHAR *envp[]);
```

Конкретная функция зависит от того, используете ли вы Unicode или нет. На самом деле входная функция операционной системой не вызывается. Вместо этого происходит обращение к стартовой функции из библиотеки C/C++, заданной во время компоновки параметром *-entry:* командной строки. Она инициализирует библиотеку C/C++, чтобы можно было вызывать

такие функции, как *malloc и free,* а также обеспечивает корректное создание любых объявленных вами глобальных и статических C++-объектов до того, как начнется выполнение вашего кода. В следующей таблице показано, в каких случаях реализуются те или иные входные функции.

Табл. 4-1. Типы приложений и соответствующие им входные функции

Тип приложения	Входная функция	Стартовая функция, встраиваемая в ваш исполняемый файл
GUI-приложение, работающее с ANSI-символами и строками	*_tWinMain (WinMain)*	*WinMainCRTStartup*
GUI-приложение, работающее с Unicode-символами и строками	*_tWinMain (wWinMain)*	*wWinMainCRTStartup*
CUI-приложение, работающее с ANSI-символами и строками	*_tmain (main)*	*mainCRTStartup*
CUI-приложение, работающее с Unicode-символами и строками	*_tmain (wmain)*	*wmainCRTStartup*

Компоновщик отвечает за выбор подходящей стартовой функции из библиотеки C/C++ при компоновке исполняемого файла. Если зада ключ /SUBSYSTEM:WINDOWS, компоновщик ищет в коде функцию *WinMain* или *wWinMain.* Если этих функций нет, компоновщик сообщает об ошибке «unresolved external symbol». В противном случае компоновщик выбирает *WinMainCRTStartup* или *wWinMainCRTStartup,* соответственно.

Аналогичным образом, если задан ключ /SUBSYSTEM:CONSOLE, компоновщик ищет в коде функцию *main* или *wmain* и выбирает соответственно *mainCRTStartup* или *wmainCRTStartup;* если в коде нет ни *main,* ни *wmain,* сообщается о той же ошибке — «unresolved external symbol».

Но не многие знают, что в проекте можно вообще не указывать ключ / SUBSYSTEM компоновщика. Если вы так и сделаете, компоновщик будет сам определять подсистему для вашего приложения. При компоновке он проверит, какая из четырех функций *(WinMain, wWinMain, main* или *wmain)* присутствует в вашем коде, и на основании этого выберет подсистему и стартовую функцию из библиотеки C/C++.

Одна из частых ошибок, допускаемых теми, кто лишь начинает работать с Visual C++, — выбор неверного типа проекта. Например, разработчик хочет создать проект Win32 Application, а сам включает в код функцию *main.* При его сборке он получает сообщение об ошибке, так как для проекта Win32 Application в командной строке компоновщика автоматически указывается ключ /SUBSYSTEM:WINDOWS, который требует присутствия в коде функции *WinMain* или *wWinMain.* В этот момент разработчик может выбрать один из четырех вариантов дальнейших действий:

- заменить *main* на *WinMain.* Как правило, это не лучший вариант, поскольку разработчик скорее всего хотел создать консольное приложение;

- открыть новый проект, на этот раз — Win32 Console Application, и перенести в него все модули кода. Этот вариант весьма утомителен, и возникает ощущение, будто начинаешь все заново;
- открыть вкладку Link в диалоговом окне Project Settings и заменить ключ I /SUBSYSTEM:WINDOWS на /SUBSYSTEM:CONSOLE в Configuration Properties/Linker/System/SubSystem (рис. 4-2). Некоторые думают, что это единственный вариант;
- открыть вкладку Link в диалоговом окне Project Settings и вообще убрать ключ /SUBSYSTEM:WINDOWS. Я предпочитаю именно этот способ, потому что он самый гибкий. Компоновщик сам сделает все, что надо, в зависимости от входной функции, которую вы реализуете в своем коде. Никак не пойму, почему это не предлагается по умолчанию при создании нового проекта Win32 Application или Win32 Console Application.

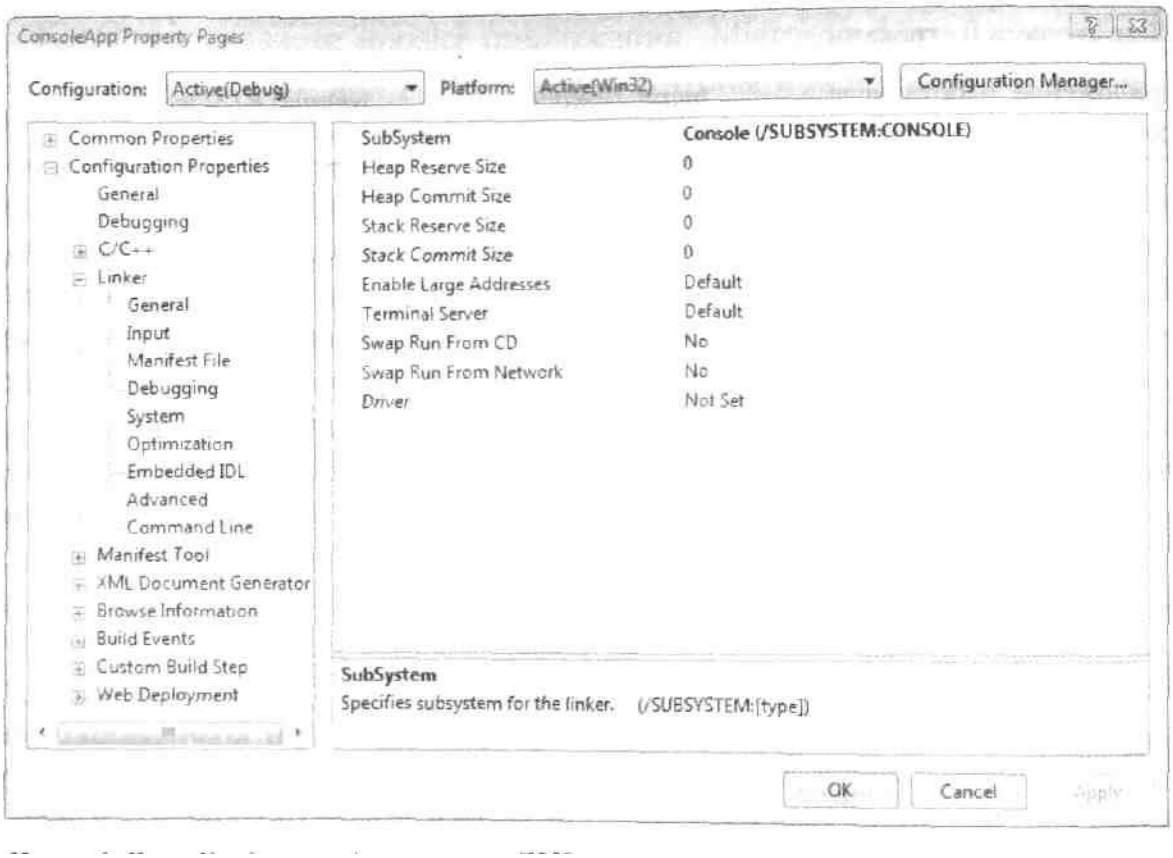


Рис. 4-2. *Выбор подсистемы CUI в окне свойств проекта*

Все стартовые функции из библиотеки C/C++ делают практически одно и то же. Разница лишь в том, какие строки они обрабатывают (в ANSI или Unicode) и какую входную функцию вызывают после инициализации библиотеки. Кстати, с Visual C++ поставляется исходный код этой библиотеки, и стартовые функции находятся в файле crtexe.c. А теперь рассмотрим, какие операции они выполняют:

- считывают указатель на полную командную строку нового процесса;
- считывают указатель на переменные окружения нового процесса;

- инициализируют глобальные переменные из библиотеки C/C++, доступ к которым из вашего кода обеспечивается включением файла StdLib.h. Список этих переменных приведен в таблице 4-2;
- инициализируют кучу (динамически распределяемую область памяти), используемую C-функциями выделения памяти (т. е. malloc и calloc) и другими процедурами низкоуровневого ввода-вывода;
- вызывают конструкторы всех глобальных и статических объектов C++-классов.

Закончив эти операции, стартовая функция обращается к входной функции в вашей программе. Если вы написали ее в виде *_tWinMain* и задали параметр _UNICODE, то она вызывается так:

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = wWinMain((HINSTANCE)&__ImageBase,  NULL,  pszCommandLineUnicode,
    (StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
       ? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

Если же параметр *_UNICODE* не задан, входная функция вызывается так:

```
GetStartupInfo(&StartupInfo);
int nMainRetVal = WinMain((HINSTANCE)&__ImageBase, NULL, pszCommandLineAnsi,
    (StartupInfo.dwFlags & STARTF_USESHOWWINDOW)
       ? StartupInfo.wShowWindow : SW_SHOWDEFAULT);
```

Заметьте, что *ImageBase* — это псевдопеременная, которая определяется компоновщиком и показывает размещение проекции исполняемого файла в памяти приложения, подробнее об этом см. в следующем разделе.

Функция *_tmain* вызывается так, если задан параметр _UNICODE:

```
int nMainRetVal = wmain(argc, argv, envp);
```

либо так, если этот параметр не задан:

```
int nMainRetVal = main(argc, argv, envp);
```

Обратите внимание, что при генерации приложений с использованием мастеров Visual Studio третий параметр (блок переменных окружения) во входной функции не определяется:

```
int _tmain(int argc, TCHAR* argv[]);
```

Если вам нужен доступ к переменным окружения процесса, просто замените предыдущую строку кода следующей:

```
int _tmain(int argc,  TCHAR* argv[],  TCHAR* env[])
```

Параметр *env* — это указатель на массив, содержащий переменные окружения с их значениями, разделенные знаком равенства (=), подробнее об этом см. ниже в разделе, посвященном переменным окружения.

Заметьте, что теперь эти переменные (табл. 4-2) не рекомендуется использовать по соображениям безопасности, поскольку использующий их код может быть запущен прежде, чем они будут инициализированы библиотекой C. В силу этих причин лучше напрямую вызывать соответствующие функции Windows API.

Когда ваша входная функция возвращает управление, стартовая обращается к функции *exit* библиотеки C/C++ и передает ей значение *nMainRetVal.* Функция *exit* выполняет следующие операции:

- вызывает все функции, зарегистрированные вызовами функции *_onexit;*
- вызывает деструкторы всех глобальных и статических объектов C++-классов;
- в отладочных сборках генерирует вызовом *_CrtDumpMemoryLeaks* список утечек памяти, управляемой средствами библиотеки C/C++ (если установлен флаг _CRTDBG_LEAK_CHECK_DF);
- вызывает Windows-функцию *ExitProcess,* передавая ей значение *nMain-RetVal* Это заставляет операционную систему уничтожить ваш процесс и установить код его завершения.

Табл. 4-2. Глобальные переменные библиотеки C/C++, доступные вашим программам

Имя переменной	Тип	Описание
_osver	*unsigned int*	Версия сборки операционной системы. Например, у Windows Vista RTM этот номер был 6000, соответственно *_osver* равна 6000. Заменяется функцией *GetVersionEx*
_winmajor	*unsigned int*	Основной номер версии Windows в шестнадцатеричной форме. Для Windows Vista это значение равно 6. Заменяется функцией *GetVersionEx*
_winminor	*unsigned int*	Дополнительный номер версии Windows в шестнадцатеричной форме. Для Windows Vista это значение равно 0. Заменяется функцией *GetVersionEx*
_winver	*unsigned int*	Вычисляется как (*_winmajor* << 8) + *_winminor*. Заменяется функцией *GetVersionEx*
__argc	*unsigned int*	Количество аргументов, переданных в командной строке. Заменяется функцией *GetCommandLine*
__argv *__wargv*	*char.* *wchar_t.*	Массив размером *__argc* с указателями на ANSI- или Unicode-строки. Каждый элемент массива указывает на один из аргументов командной строки. Заметьте, что *__argv* = NULL, если установлен флаг _UNICODE и *__wargv* = NULL в противном случае. Заменяется функцией *GetCommandLine*

Табл. 4-2. *(окончание)*

Имя переменной	Тип	Описание
_environ	*char*	Массив указателей на ANSI- или Unicode-строки.
_wenviron	*wchar_t*	Каждый элемент массива указывает на строку — переменную окружения. Заметьте, что *__wenviron* = NULL, если не установлен флаг _UNICODE, и *__environ* = NULL в противном случае. Заменяется функцией *GetEnvironmentVariable*
_pgmptr *_wpgmptr*	*char* *wchar_t*	Полный путь и имя (в ANSI или Unicode) запускаемой программы. Заметьте, что *__pgmptr* = NULL, если установлен флаг _UNICODE. Заменяется вызовом функции *GetModuleFileName* с передачей NULL в первом параметре

Описатель экземпляра процесса

Любому EXE- или DLL-модулю, загружаемому в адресное пространство процесса, присваивается уникальный описатель экземпляра. Описатель экземпляра вашего EXE-файла передается как первый параметр функции *(w)WinMain* — *MnstanceExe*. Это значение обычно требуется при вызовах функций, загружающих те или иные ресурсы. Например, чтобы загрузить из образа EXE-файла такой ресурс, как значок, надо вызвать:

```
HICON LoadIcon(
    HINSTANCE hInstance,
    PCTSTR pszIcon);
```

Первый параметр в *LoadIcon* указывает, в каком файле (EXE или DLL) содержится интересующий вас ресурс. Многие приложения сохраняют параметр *hInstanceExe* функции *(w)WinMain* в глобальной переменной, благодаря чему он доступен из любой части кода EXE-файла.

В документации Platform SDK утверждается, что некоторые Windows-функции требуют параметр типа HMODULE. Пример — функция *GetModuleFileName:*

```
DWORD GetModuleFileName(
    HMODULE hlnstModule,
    PTSTR pszPath,
    DWORD cchPath);
```

> **Примечание.** Как оказалось, HMODULE и HINSTANCE — это одно и то же. Встретив в документации указание передать какой-то функции HMODULE, смело передавайте HINSTANCE, и наоборот. Они существуют в таком виде лишь потому, что в 16-разрядной Windows идентифицировали совершенно разные вещи.

Истинное значение параметра *InstanceExe* функции *(w)WinMain* — базовый адрес в памяти, определяющий ту область в адресном пространстве процесса, куда был загружен образ данного EXE-файла. Например, если система открывает исполняемый файл и загружает его содержимое по адресу 0x00400000, то *hInstanceExe* функции *(w)WinMain* получает значение 0x00400000.

Базовый адрес, по которому загружается приложение, определяется компоновщиком. Разные компоновщики выбирают и разные (по умолчанию) базовые адреса. Компоновщик Visual C++ использует по умолчанию базовый адрес 0x00400000 - самый нижний в Windows 98, начиная с которого в ней допускается загрузка образа исполняемого файла. Указав параметр /BASE: *адрес* (в случае компоновщика от Майкрософт), можно изменить базовый адрес, по которому будет загружаться приложение.

Функция *GetModuleHandle:*

```
HM0DULE GetModuleHandle(PCTSTR pszModule);
```

возвращает описатель/базовый адрес, указывающий, куда именно (в адресном пространстве процесса) загружается EXE- или DLL-файл. При вызове этой функции имя нужного EXE- или DLL-файла передается как строка с нулевым символом в конце. Если система находит указанный файл, *GetModuleHandle* возвращает базовый адрес, по которому располагается образ данного файла. Если же файл системой не найден, функция возвращает NULL. Кроме того, можно вызвать эту функцию, передав ей NULL вместо параметра *pszModule,* — тогда вы узнаете базовый адрес EXE-файла. Именно это и делает стартовый код из библиотеки C/C++ при вызове *(w)WinMain* из вашей программы. Узнать, в каком модуле работает код из DLL, можно двумя способами: использовать псевдопеременную компоновщика *ImageBase,* указывающую на базовый адрес текущего исполняемого модуля.

Другой способ — вызов *GetModuleHandleEx* с передачей GET_MODULE_ HANDLE_EX_FLAG_FROM_ADDRESS в качестве первого параметра и адреса текущего метода - в качестве второго параметра. В указатель на HMODULE, переданный как последний параметр, будет записан базовый адрес DLL, содержащей переданную функцию. Вот примеры, иллюстрирующие оба способа:

```
extern "C" const IMAGE_D0S_HEADER __ImageBase;
void DumpModule() {
    // Получаем базовый адрес исполняемого приложения.
    // Он может отличаться от такового у запущенного модуля,
    // если этот код содержится в DLL.
    HMODULE hModule = GetModuleHandle(NULL);
    _tprintf(TEXT("with GetModuleHandle(NULL) = 0x%x\r\n"), hModule);
```

```
    // Используем псевдопеременную __ImageBase для получения
    // адреса текущего модуля hModule/hlnstance.
    _tprintf(TEXT("with __ImageBase = 0x%x\r\n"), (HINSTANCE)&__ImageBase);

    // Передаем GetModuleHandleEx адрес текущего метода
    // DumpModule как параметр для получения адреса
    // текущего модуля hModule/hInstance.
    hModule = NULL;
    GetModuleHandleEx(
        GET_MODULE_HANDLE_EX_FLAG_FROM_ADDRESS,
        (PCTSTR)DumpModule,
        &hModule);
    _tprintf(TEXT("with GetModuleHandleEx = 0x%x\r\n"), hModule);
}

int _tmain(int argc, TCHAR* argv[]) {
    DumpModule();
    return(0);
}
```

Есть еще две важные вещи, касающиеся *GetModuleHandle*. Во-первых, она проверяет адресное пространство только того процесса, который ее вызвал. Если этот процесс не использует никаких функций, связанных со стандартными диалоговыми окнами, то, вызвав *GetModuleHandle* и передав ей аргумент «ComDlg32», вы получите NULL — пусть даже модуль ComDlg32.dll и загружен в адресное пространство какого-нибудь другого процесса. Во-вторых, вызов этой функции и передача ей NULL дает в результате базовый адрес EXE-файла в адресном пространстве процесса. Так что, вызывая функцию в виде *GetModuleHandle (NULL)* — даже из кода в DLL, — вы получаете базовый адрес EXE-, а не DLL-файла.

Описатель предыдущего экземпляра процесса

Я уже говорил, что стартовый код из библиотеки C/C++ всегда передает в функцию *(w)WinMain* параметр *hPrevInstance* как NULL. Этот параметр предусмотрен исключительно для совместимости с 16-разрядными версиями Windows и не имеет никакого смысла для Windows-приложений. Поэтому я всегда пишу заголовок *(w)WinMain* так:

```
int WINAPI _tWinMain(
    HINSTANCE hInstanceExe,
    HINSTANCE,
    PSTR pszCmdLine,
    int nCmdShow);
```

Поскольку у второго параметра нет имени, компилятор не выдает предупреждение «parameter not referenced» («нет ссылки на параметр»). В Visual

Studio выбрано иное решение: сгенерированный мастером проекта-приложения на C++ использует макрос UNREFERENCED PARAMETER для удаления этих предупреждений следующим образом:

```
int APIENTRY _tWinMain(HINSTANCE hInstance,
                       HINSTANCE hPrevInstance,
                       LPTSTR lpCmdLine,
                       int nCmdShow) {
    UNREFERENCED_PARAMETER(hPrevInstance);
    UNREFERENCED_PARAMETER(lpCmdLine);
    …
}
```

Командная строка процесса

При создании новому процессу передается командная строка, которая почти никогда не бывает пустой — как минимум, она содержит имя исполняемого файла, использованного при создании этого процесса. Однако, как вы увидите ниже (при обсуждении функции *CreateProcess),* возможны случаи, когда процесс получает командную строку, состоящую из единственного символа — нуля, завершающего строку. В момент запуска приложения стартовый код из библиотеки C считывает командную строку процесса, пропускает имя исполняемого файла и заносит в параметр *pszCmdLine* функции *WinMain* указатель на оставшуюся часть командной строки.

Программа может анализировать и интерпретировать командную строку как угодно. Поскольку *pszCmdLine* относится к типу PSTR, а не PCSTR, не стесняйтесь и записывайте строку прямо в буфер, на который указывает этот параметр, но ни при каких условиях не переступайте границу буфера. Лично я всегда рассматриваю этот буфер как «только для чтения». Если в командную строку нужно внести изменения, я сначала копирую буфер, содержащий командную строку, в локальный буфер (в своей программе), который затем и модифицирую.

Указатель на полную командную строку процесса можно получить и вызовом функции *GetCommandLine:*

```
PTSTR GetCommandLine();
```

Она возвращает указатель на буфер, содержащий полную командную строку, включая полное имя (вместе с путем) исполняемого файла. Учтите, что *GetCommandLine* всегда возвращает адрес одного и того же буфера. Это еще одна причина, по которой следует воздерживаться от записи данных в *pszCmdLine:* при этом будет модифицирован исходный буфер, после чего восстановить исходную командную строку не удастся.

Во многих приложениях, безусловно, удобнее использовать командную строку, предварительно разбитую на отдельные компоненты, доступ к которым приложение может получить через глобальные переменные *__argc* и *__argv* (или *__wargv),* использовать которые, теперь не рекомендуется.

Функция *CommandLineToArgoW*из *ShellAPI* расщепляет Unicode-строку на отдельные компоненты:

```
PWSTR* CommandLineToArgvW(
    PWSTR pszCmdLine,
    int* pNumArgs);
```

Буква *W* в конце имени этой функции намекает на «широкие» (wide) символы и подсказывает, что функция существует только в Unicode-версии. Параметр *pszCmdLine* указывает на командную строку. Его обычно получают предварительным вызовом *GetCommandLineW*. Параметр *pNumArgs* — это адрес целочисленной переменной, в которой задается количество аргументов в командной строке. Функция *CommandLineToArgvW* возвращает адрес массива указателей на Unicode-строки.

CommandLineToArgvW выделяет нужную память автоматически. Большинство приложений не освобождает эту память, полагаясь на операционную систему, которая проводит очистку ресурсов по завершении процесса. И такой подход вполне приемлем. Но если вы хотите сами освободить эту память, сделайте так:

```
int nNumArgs;
PWSTR *ppArgv = CommandLineToArgvW(GetCommandLineW(), AnNumArgs);

// используйте эти аргументы…
if (*ppArgv[1] == L'x') {
…
}
// освободите блок памяти
HeapFree(GetProcessHeap(), 0, ppArgv);
```

Переменные окружения

С любым процессом связан блок переменных окружения — область памяти, выделенная в адресном пространстве процесса. Каждый блок содержит группу строк такого вида:

```
=::=::\ …
VarName1=VarValue1\0
VarName2=VarValue2\0
VarName3=VarValue3\0 …
VarNameX=VarValueX\0
\0
```

Первая часть каждой строки — имя переменной окружения. За ним следует знак равенства и значение, присваиваемое переменной. Обратите внимание, что не только первая строка (=:: = :: \), но и другие строки в этом блоке могут начинаться символом «=». Такие строки не считаются переменными окружения (подробнее об этом — ниже).

Показанные выше способы доступа к блоку переменных окружения дают разные результаты в зависимости от метода разбора. Первый способ позволяет получить блок переменных окружения целиком вызовом функции *GetEnvironmentStrings* (см. выше); извлечение отдельных переменных окружения иллюстрирует следующий пример:

```
void DumpEnvStrings() {
    PTSTR pEnvBlock = GetEnvironmentStrings();
    // разбор блока,  имеющего следующий формат:
    //   =:: = ::\
    //   =
    //   var=value\0
    //   …
    //   var=value\0\0
    // Заметьте, что некоторые строки могут начинаться знаком '='.
    // В этом примере приложение запущено с сетевого диска.
    //   [0] =::=::\
    //   [1] =C:=C:\Windows\System32
    //   [2] =ExitCode=00000000
    //
    TCHAR szName[MAX_PATH];
    TCHAR szValue[MAX_PATH];
    PTSTR pszCurrent = pEnvBlock;
    HRESULT hr = S_OK;
    PCTSTR pszPos = NULL;
    int current = 0;

    while (pszCurrent != NULL) {
        // пропуск незначащих строк, таких как эта:
        // "=::=::\"
        if (*pszCurrent != TEXT('=')) {
            // поиск разделителя '='
            pszPos = _tcschr(pszCurrent, TEXT('='));

            // указатель на первый символ значения
            pszPos++;

            // копирование имени переменной…
            size_t cbNameLength =  //        …без знака ' ='
                (size_t)pszPos - (size_t)pszCurrent - sizeof(TCHAR);
            hr = StringCbCopyN(szName, MAX_PATH, pszCurrent, cbNameLength);
            if (FAILED(hr)) {
                break;
            }
```

```
            // Копируем значение переменной с NULL-символом в конце
            // и разрешаем усечение, поскольку эта строка
            // предназначена только для отображения в UI.
            hr = StringCchCopyN(szValue, MAX_PATH, pszPos, _tcslen(pszPos)+1);
            if (SUCCEEDED(hr)) {
                _tprintf(TEXT("[Xu] %s=%s\r\n"), current, szName, szValue);
            } else // что-то не так, проверить усечение
            if (hr == STRSAFE_E_INSUFFICIENT_BUFFER) {
                _tprintf(TEXT("[%u] %s=%s…\r\n"), current, szName, szValue);
            } else { // такого быть не должно
                _tprintf(
                    TEXT("[%u] %s=???\r\n"), current, szName
                    )
                break;
            }
        } else {
            _tprintf(TEXT("[%u] %s\r\n"), current, pszCurrent);
        }

        // переходим к следующей переменной
        current++;

        // Move to the end of the string.
        while (*pszCurrent != TEXT('\0'))
            pszCurrent++;
        pszCurrent++;

        // проверим, не последняя ли это строка
        if (*pszCurrent == TEXT('\0'))
            break;
    };

    // не забудьте освободить память!
    FreeEnvironmentStrings(pEnvBlock);
}
```

Недопустимые строки (строки, начинающиеся с «=», знака-разделителя) игнорируются, остальные строки поочередно подвергаются разбору. При этом знак «=» интерпретируется как разделитель имени переменной и ее значения. Закончив работу с блоком памяти, возвращенным *GetEnvironmentStrings*, освободите его вызовом *FreeEnvironmentStrings:*

```
BOOL FreeEnvironmentStrings(PTSTR pszEnvironmentBlock);
```

Заметьте, что в этом примере для работы со строками используются безопасные С-функции, например, для расчета размера строки в байтах при использовании *StringCbCopyN* и усечении строки, не умещающейся в буфере, при использовании *StringCchCopyN*.

Второй способ доступа к переменным окружения поддерживается только для GUI-приложений и основан на параметре TCHAR* env[] входной функции. В отличие от параметра, возвращаемого *GetEnvironmentStnngs,* env -это массив указателей на строки, представляющие различные переменные окружения в обычном формате «имя=значение». За указателем на последнюю строку идет NULL-указатель, как показано ниже:

```
void DumpEnvVariables(PTSTR pEnvBlock[]) {
   int current = 0;
   PTSTR* pElement = (PTSTR*)pEnvBlock;
   PTSTR pCurrent = NULL;
   while (pElement != NULL) {
       pCurrent = (PTSTR)(*pElement);
       if (pCurrent == NULL) {
           // переменные среды закончились
           pElement = NULL;
       } else {
           _tprintf(TEXT("[%u] %s\r\n"), current, pCurrent);
           current++;
           pElement++;
       }
   }
}
```

Обратите внимание, что негодные строки, начинающиеся символом «=», удаляются до возврата *env,* поэтому вам не придется самостоятельно делать это.

Поскольку знак равенства используется в качестве разделителя имен и значений переменных, этот символ не может быть частью имени переменной. Пробелы также являются значащими символами. Например, объявив следующие переменные

```
XYZ= Windows (обратите внимание на пробел за знаком равенства)
ABC-Windows
```

и сравнив значения переменных *XYZnABC,* вы увидите, что система их различает, - она учитывает любой пробел, поставленный перед знаком равенства или после него. Вот что будет, если записать, скажем, так:

```
XYZ *Home   (обратите внимание на пробел за знаком равенства)
XYZ*Work
```

Вы получите первую переменную с именем «XYZ», содержащую строку «Home», и вторую переменную «XYZ», содержащую строку «Work».

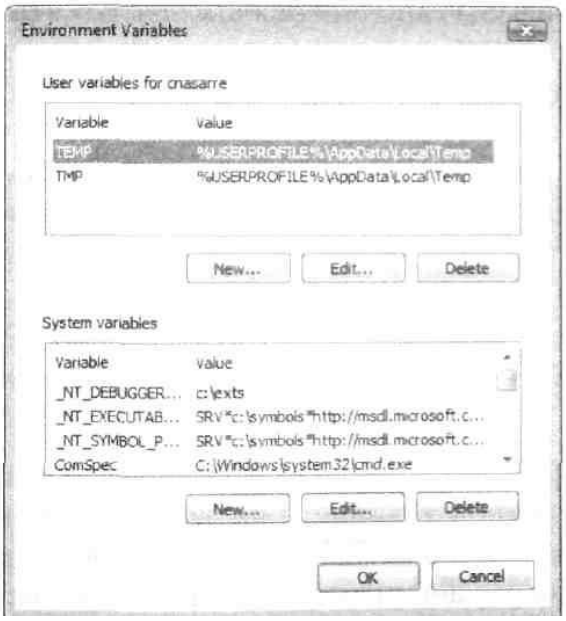
Примечание. При регистрации пользователя на входе в Windows система создает процесс-оболочку, связывая с ним группу строк — переменных окружения. Система получает начальные значения этих строк, анализируя два раздела в реестре. В первом:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
    Session Manager\Environment
```

содержится список переменных окружения, относящихся к системе, а во втором:

```
HKEY_CURRENT_USER\Environment
```

находится список переменных окружения, относящихся к пользователю, который в настоящее время зарегистрирован в системе. Пользователь может добавлять, удалять или изменять любые переменные через аплет System из Control Panel. В этом аплете надо открыть вкладку Advanced и щелкнуть кнопку Environment Variables — тогда на экране появится следующее диалоговое окно:



Модифицировать переменные из списка System Variables разрешается только пользователю с правами администратора. Кроме того, для модификации записей в реестре ваша программа может обращаться к Windows-функциям, позволяющим манипулировать реестром. Однако, чтобы изменения вступили в силу, пользователь должен выйти из системы и вновь войти в нее. Некоторые приложения типа Explorer, Task Manager или Control Panel могут обновлять свои блоки переменных окружения на базе новых значений в реестре, когда их главные окна получают сообщение WM_SETTINGCHANGE. Например, если вы, изменив реестр, хотите, чтобы какие-то приложения соответственно обновили свои блоки переменных окружения, вызовите:

```
SendMessage(HWND_BROADCAST, WM_SETTINGCHANGE, 0, (LPARAM) TEXTEnviron-
    ment"));
```

Обычно дочерний процесс наследует набор переменных окружения от родительского. Однако последний способен управлять тем, какие переменные окружения наследуются дочерним процессом, а какие — нет. Но об этом я расскажу, когда мы займемся функцией *CreateProcess*. Под наследованием я имею в виду, что дочерний процесс получает свою копию блока переменных окружения от родительского, а не то, что дочерний и родительский процессы совместно используют один и тот же блок. Так что дочерний процесс может добавлять, удалять или модифицировать переменные в своем блоке, и эти изменения не затронут блок, принадлежащий родительскому процессу.

Переменные окружения обычно применяются для тонкой настройки приложения. Пользователь создает и инициализирует переменную окружения, затем запускает приложение, и оно, обнаружив эту переменную, проверяет ее значение и соответствующим образом настраивается.

Увы, многим пользователям не под силу разобраться в переменных окружения, а значит, трудно указать правильные значения. Ведь для этого надо не только хорошо знать синтаксис переменных, но и, конечно, понимать, что стоит за теми или иными их значениями. С другой стороны, почти все (а может, и все) приложения, основанные на GUI, дают возможность тонкой настройки через диалоговые окна. Такой подход, естественно, нагляднее и проще.

А теперь, если у вас еще не пропало желание манипулировать переменными окружения, поговорим о предназначенных для этой цели функциях. *GetEnvironmentVariable* позволяет выявлять присутствие той или иной переменной окружения и определять ее значение:

```
DWORD GetEnvironmentVariable(
    PCTSTR pszName,
    PTSTR pszValue,
    DWORD cchValue);
```

При вызове *GetEnvironmentVariable* параметр *pszName* должен указывать на имя интересующей вас переменной, *pszValue* — на буфер, в который будет помещено значение переменной, а в *cchValue* следует сообщить размер буфера в символах. Функция возвращает либо количество символов, скопированных в буфер, либо 0, если ей не удалось обнаружить переменную окружения с таким именем. Однако размер значения переменной окружения заранее не известен, поэтому при передаче в параметре *cchValue* значения *0* функция *GetEnvironmentVariable* возвращает размер (число символов плюс завершающий NULL-символ). Вот пример безопасного использования этой функции:

```
void PrintEnvironmentvariable(PCTSTR pszVariableName) {
   PTSTR pszValue = NULL;
   // получаем размер буфера,  необходимого для хранения значения
   DWORD dwResult = GetEnvironmentVariable(pszVariableName, pszValue, 0);
   if (dwResult != 0) {
```

```
      // выделяем буфер для значения переменной окружения
      DWORD size = dwResult * sizeof(TCHAR);
      pszValue = (PTSTR)malloc(size);
      GetEnvironmentVariable(pszVariableName,  pszValue,  size);
      _tprintf(TEXT("%s=%s\n"), pszVariableName,  pszValue);
      free(pszValue);
   } else {
      _tprintf(TEXT(",%s,=<unknown value>\n"), pszVariableName);
   }
}
```

Кстати, в реестре многие строки содержат подставляемые части, например:

```
%USERPROFILE%\Documents
```

Часть, заключенная в знаки процента, является подставляемой. В данном случае в строку должно быть подставлено значение переменной окружения USERPROFILE. На моей машине эта переменная выглядит так:

```
C:\Users\jrichter
```

После подстановки переменной в строку реестра получим:

```
C:\Users\jrichter\Documents
```

Поскольку такие подстановки делаются очень часто, в Windows есть функция *ExpandEnvironmentStrings*:

```
DWORD ExpandEnvironmentStrings(
    PTCSTR pszSrc,
    PTSTR pszDst,
    DWORD chSize);
```

Параметр *pszSrc* принимает адрес строки, содержащей подставляемые части, а параметр *pszDst* — адрес буфера, в который записывается развернутая строка. Параметр *chSize* определяет максимальный размер буфера в символах. Возвращает функция размер буфера (в символах), необходимого для хранения результата подстановки. Если значение параметра *chSize* меньше этого значения, вместо подстановки переменные *%%* заменяются пустыми строками. Поэтому *ExpandEnvironmentStrings* обычно вызывают дважды, как показано в примере ниже:

```
DWORD chValue =
    ExpandEnvironmentStrings(TEXT("PATH=,'%PATH%'"), NULL, 0);
PTSTR pszBuffer = new TCHAR[chValue];
chValue = ExpandEnvironmentStrings(TEXT("PATH='%PATH%'"), pszBuffer, chValue);
_tprintf(TEXT("%s\r\n"), pszBuffer);
deleted pszBuffer;
```

Наконец, функция *SetEnvironmentVanable* позволяет добавлять, удалять и модифицировать значение переменной:

```
BOOL SetEnvironmentVariable(
    PCTSTR pszName,
    PCTSTR pszValue);
```

Она устанавливает ту переменную, на чье имя указывает параметр *pszName,* и присваивает ей значение, заданное параметром *pszValue.* Если такая переменная уже существует, функция модифицирует ее значение. Если же *в pszValue* содержится NULL, переменная удаляется из блока.

Для манипуляций с блоком переменных окружения всегда используйте именно эти функции.

Привязка к процессорам

Обычно потоки внутри процесса могут выполняться на любом процессоре компьютера. Однако их можно закрепить за определенным подмножеством процессоров из числа имеющихся на компьютере. Это свойство называется *привязкой к процессорам* (processor affinity) и подробно обсуждается в главе 7. Дочерние процессы наследуют привязку к процессорам от родительских.

Режим обработки ошибок

С каждым процессом связан набор флагов, сообщающих системе, каким образом процесс должен реагировать на серьезные ошибки: повреждения дисковых носителей, необрабатываемые исключения, ошибки операций поиска файлов и неверное выравнивание данных. Процесс может указать системе, как обрабатывать каждую из этих ошибок, через функцию *SetErrorMode:*

```
UINT SetErrorMode(UINT fuErrorMode);
```

Параметр *fuErrorMode* — это набор флагов (таблица 4-3), комбинируемых побитовой операцией OR.

Табл. 4-3. Флаги SetErrorMode

Флаг	Описание
SEM_FAILCRITICALERRORS	Система не выводит окно с сообщением от обработчика критических ошибок и возвращает ошибку в вызывающий процесс
SEM_NOGPFAULTERRORBOX	Система не выводит окно с сообщением о нарушении общей защиты; этим флагом манипулируют только отладчики, самостоятельно обрабатывающие нарушения общей защиты с помощью обработчика исключений
SEM_NOOPENFILEERRORBOX	Система не выводит окно с сообщением об отсутствии искомого файла

Табл. 4-3. *(окончание)*

Флаг	Описание
SEM_NOALIGNMENTFAULTEXCEPT	Система автоматически исправляет нарушения в выравнивании данных, и они становятся невидимы приложению; этот флаг не действует на процессорах *x86*

По умолчанию дочерний процесс наследует от родительского флаги, указывающие на режим обработки ошибок. Иначе говоря, если у процесса в данный момент установлен флаг SEM_NOGPFAULTERRORBOX и он порождает другой процесс, этот флаг будет установлен и у дочернего процесса. Однако «наследник» об этом не уведомляется, и он вообще может быть не рассчитан на обработку ошибок такого типа (в данном случае — нарушений общей защиты). В результате, если в одном из потоков дочернего процесса все-таки произойдет подобная ошибка, этот процесс может завершиться, ничего не сообщив пользователю. Но родительский процесс способен предотвратить наследование дочерним процессом своего режима обработки ошибок, указав при вызове функции *CreateProcess* флаг CREATE_DEFAULT_ERROR_MODE (о *CreateProcess* чуть позже).

Текущие диск и каталог для процесса

Текущий каталог текущего диска — то место, где Windows-функции ищут файлы и подкаталоги, если полные пути в соответствующих параметрах не указаны. Например, если поток в процессе вызывает функцию *CreateFile*, чтобы открыть какой-нибудь файл, а полный путь не задан, система просматривает список файлов в текущем каталоге текущего диска. Этот каталог отслеживается самой системой, и, поскольку такая информация относится ко всему процессу, смена текущего диска или каталога одним из потоков распространяется и на остальные потоки в данном процессе.

Поток может получать и устанавливать текущие каталог и диск для процесса с помощью двух функций:

```
DWORD GetCurrentDirectory(
    DWORD cchCurDir,
    PTSTR pszCurDir);
BOOL SetCurrentDirectory(PCTSTR pszCurDir);
```

Если предоставленный буфер недостаточно велик, *GetCurrentDirectory* возвращает размер буфера (в символах), необходимого для хранения пути к этой папке (включая концевую строку «/0»), ничего не копируя в предоставленный буфер; в этом случае его можно установить в NULL. Если вызов успешен, функция возвращает длину строки (в символах, без учета концевой строки «/0»).

> **Примечание.** В WinDef.h значение константы MAX_PATH установлено в 260, это максимальная длина имени файла или каталога. Таким образом, при вызове *GetCurrentDirectory* можно смело передавать буфер, содержащий MAX_PATH элементов типа TCHAR.

Текущие каталоги для процесса

Система отслеживает текущие диск и каталог для процесса, но не текущие каталоги на каждом диске. Однако в операционной системе предусмотрен кое-какой сервис для манипуляций с текущими каталогами на разных дисках. Он реализуется через переменные окружения конкретного процесса. Например:

```
=C:=C:\Utility\Bin
=D:=D:\Program Files
```

Эти переменные указывают, что текущим каталогом на диске C является \Utility\Bin, а на диске D — Program Files.

Если вы вызываете функцию, передавая ей путь с именем диска, отличного от текущего, система сначала просматривает блок переменных окружения и пытается найти переменную, связанную с именем указанного диска. Если таковая есть, система выбирает текущий каталог на заданном диске в соответствии с ее значением, нет — текущим каталогом считается корневой.

Скажем, если текущий каталог для процесса — C:\Utility\Bin и вы вызываете функцию *CreateFile,* чтобы открыть файл D:\ReadMe.txt, система ищет переменную **=D:**. Поскольку переменная **=D:** существует, система пытается открыть файл ReadMe.txt в каталоге D:\Program Files. А если бы таковой переменной не было, система искала бы файл ReadMe.txt в корневом каталоге диска D. Кстати, файловые Windows-функции никогда не добавляют и не изменяют переменные окружения, связанные с именами дисков, а лишь считывают их значения.

> **Примечание.** Для смены текущего каталога вместо Windows-функции *SetCurrentDirectory* можно использовать функцию *_chdir* из библиотеки C. Внутренне она тоже обращается к *SetCurrentDirectory,* но, кроме того, способна добавлять или модифицировать переменные окружения, что позволяет запоминать в программе текущие каталоги на различных дисках.

Если родительский процесс создает блок переменных окружения и хочет передать его дочернему процессу, тот не наследует текущие каталоги родительского процесса автоматически. Вместо этого у дочернего процесса текущими на всех дисках становятся корневые каталоги. Чтобы дочерний процесс унаследовал текущие каталоги родительского, последний должен создать соответствующие переменные окружения (и сделать это до порождения другого процесса). Родительский процесс может узнать, какие каталоги являются текущими, вызвав *GetFullPathName*:

```
DWORD GetFullPathName(
    PCTSTR pszFile,
    DWORD cchPath,
    PTSTR pszPath,
    PTSTR *ppszFilePart);
```

Например, чтобы получить текущий каталог на диске C, функцию вызывают так:

```
TCHAR szCurDir[MAX_PATH];
DWORD cchLength = GetFullPathName(TEXT("C:"), MAX_PATH, szCurDir, NULL);
```

Не забывайте, что переменные окружения процесса должны всегда храниться в алфавитном порядке. Поэтому переменные, связанные с дисками, обычно приходится размещать в самом начале блока.

Определение версии системы

Весьма часто приложению требуется определять, в какой версии Windows оно выполняется. Причин тому несколько. Например, программа может использовать функцию *CreateFileTransacted* из Windows API, но в полной мере эта функция реализованы лишь в Windows Vista.

Насколько я помню, функция *GetVersion* есть в API всех версий Windows:

```
DWORD GetVersion();
```

С этой простой функцией связана целая история. Сначала ее разработали для 16-разрядной Windows, и она должна была в старшем слове возвращать номер версии MS-DOS, а в младшем — номер версии Windows. Соответственно в каждом слове старший байт сообщал основной номер версии, младший — дополнительный номер версии.

Увы, программист, писавший ее код, слегка ошибся, и получилось так, что номера версии Windows поменялись местами: в старший байт попадал дополнительный номер, а в младший — основной. Поскольку многие программисты уже начали пользоваться этой функцией, Майкрософт пришлось оставить все, как есть, и изменить документацию с учетом ошибки.

Из-за всей этой неразберихи вокруг *GetVersion* в Windows API включили новую функцию — *GetVersionEx:*

```
BOOL GetVersionEx(POSVERSIONINFOEX pVersionInformation);
```

Перед обращением к *GetVersionEx* программа должна создать структуру OSVERSIONINFOEX, показанную ниже, и передать ее адрес этой функции. В

```
typedef struct {
    DWORD dwOSVersionlnfoSize;
    DWORD dwMajorVersion;
    DWORD dwMinorVersion;
```

```
    DWORD dwBuildNumber;
    DWORD dwPlatformId;
    TCHAR szCSDVersion[128];
    WORD  wServicePackMajor;
    WORD  wServicePackMinor;
    WORD  wSuiteMask;
    BYTE  wProductType;
    BYTE  wReserved;
} OSVERSIONINFOEX,   *POSVERSIONINFOEX;
```

Эта структура появилась в Windows 2000. В остальных версиях Windows используется структура OSVERSIONINFO, в которой нет последних пяти элементов, присутствующих в структуре OSVERSIONINFOEX.

Обратите внимание, что каждому компоненту номера версии операционной системы соответствует свой элемент структуры. Это сделано специально — чтобы программисты не возились с выборкой данных из всяких там старших-младших байтов-слов (и не путались в них!); теперь программе гораздо проще сравнивать ожидаемый номер версии операционной системы с действительным. Назначение каждого элемента структуры OSVERSIONINFOEX описано в таблице 4-4. Анализ полей этой структуры подробно разбирается в статье «Getting the System Version» на веб-сайте MSDN (http://msdn2.microsoft.com/en-gb/library/ms724429.aspx).

В Windows 2000 появилась новая функция, *VerifyVersionInfo*, которая сравнивает версию установленной операционной системы с тем, что требует ваше приложение:

```
BOOL VerifyVersionInfo(
    POSVERSIONINFOEX pVersionInformation,
    DWORD dwTypeMask,
    DWORDLONG dwlConditionMask);
```

Табл. 4-4. Члены структуры OSVERSIONINFOEX

Элемент	Описание
dwOSVersionInfoSize	Размер структуры; перед обращением к функции GetVersionEx должен быть заполнен вызовом sizeof (OSVERSIONINFO) или sizeof (OSVERSIONINFOEX)
dwMajorVersion	Основной номер версии операционной системы
dwMinorVersion	Дополнительный номер версии операционной системы
dwBuildNumber	Версия сборки данной системы
dwPlatformId	Идентификатор платформы, поддерживаемой данной системой; его возможные значения: VER_PLATFORM_WIN32s (Win32-платформы), VER_PLATFORM_WIN32_WINDOWS (Windows 95/98), VER_PLATFORM_WIN32_NT (Windows NT/2000/XP, Windows Server 2003, Windows Vista)

Табл. 4-4. *(окончание)*

Элемент	Описание
szCSDVersion	Этот элемент содержит текст — дополнительную информацию об установленной операционной системе
wServicePackMajor	Основной номер версии последнего установленного пакета исправлений (service pack)
wServicePackMinor	Дополнительный номер версии последнего установленного пакета исправлений
wSuiteMask	Сообщает, какие программные пакеты (suites) доступны в системе; его возможные значения: VER_SUITE_SMALLBUSINESS, VER_SUITE ENTERPRISE, VER_SUITE_BACKOFFICE, VER_SUITE_ COMMUNICATIONS, VER_SUITE_TERMINAL, VER_SUITE_ SMALLBUSINESS_RESTRICTED, VER_SUITE_EMBEDDEDNT, VER_SUITE_DATACENTER, VER_SUITE_SINGLEUSERTS (один сеанс служб терминалов в расчете на пользователя), VER_SUITE_PERSONAL (позволяет различить редакции Home и Professional), VER_SUITE_BLADE, VER_SUITE_ EMBEDDED_ESTRICTED, VER_SUITE_SECURITY_ APPLIANCE, VER_SUITE_STORAGE_SERVER, VER_SUITE_ COMPUTE_SERVER)
wProductType	Сообщает, какой именно вариант операционной системы установлен; его возможные значения: VER_NT_WORKSTATION, VER_NT_SERVER, VER_NT_DOMAIN_CONTROLLER
wReserved	Зарезервирован на будущее

Чтобы использовать эту функцию, создайте структуру OSVERSIONINFOEX, запишите в ее элемент dwOSVersionInfoSize размер структуры, а потом инициализируйте любые другие элементы, важные для вашей программы. При вызове VerifyVersionInfo параметр dwTypeMask указывает, какие элементы структуры вы инициализировали. Этот параметр принимает любые комбинации следующих флагов: VER_MINORVERSION, VER_MAJORVERSION, VER_BUILDNUMBER, VER_PLATFORMID, VER_SERVICEPACKMINOR, VER_SERVICEPACKMAJOR, VER_SUITENAME и VER_PRODUCT_TYPE. Последний параметр, dwlConditionMask, является 64-разрядным значением, которое управляет тем, как именно функция сравнивает информацию о версии системы с нужными вам данными.

Параметр dwlConditionMask устанавливает правила сравнения через сложный набор битовых комбинаций. Для создания требуемой комбинации используйте макрос VER_SET_CONDITION:

```
VER_SET_C0NDITI0N(
    DWORDLONG dwlConditionMask,
    ULONG dwTypeBitMask,
    ULONG dwConditionMask)
```

Первый параметр, *dwlConditionMask,* идентифицирует переменную, битами которой вы манипулируете. Вы не передаете адрес этой переменной, потому что VER_SET_CONDITION — макрос, а не функция. Параметр *dwTypeBitMask* указывает один элемент в структуре OSVERSIONINFOEX, который вы хотите сравнить со своими данными. (Для сравнения нескольких элементов придется обращаться к VER_SET_CONDITION несколько раз подряд.) Флаги, передаваемые в этом параметре (VER_MINORVERSION, VER_BUILDNUMBER и пр.), идентичны передаваемым в параметре *dwTypeMask* функции *Verify VersionInfo.*

Последний параметр макроса VER_SET_CONDITION, *dwConditionMask,* сообщает, как вы хотите проводить сравнение. Он принимает одно из следующих значений: VER_EQUAL, VER_GREATER, VER_GREATER_EQUAL, VER_LESS или VER_LESS_EQUAL Вы можете использовать эти значения в сравнениях по VER_PRODUCT_TYPE. Например, значение VER_NT_WORKSTATION меньше, чем VER_NT_SERVER. Но в сравнениях по VER_SUITENAME вместо этих значений применяется VER_AND (должны быть установлены все программные пакеты) или VER_OR (должен быть установлен хотя бы один из программных пакетов).

Подготовив набор условий, вы вызываете *VerifyVersionInfo* и получаете ненулевое значение, если система отвечает требованиям вашего приложения, или 0, если она не удовлетворяет этим требованиям или если вы неправильно вызвали функцию. Чтобы определить, почему *VerifyVersionInfo* вернула 0, вызовите *GetLastError.* Если та вернет ERROR_OLD_WIN_VERSION, значит, вы правильно вызвали функцию *VerifyVersionInfo,* но система не соответствует предъявленным требованиям.

Вот как проверить, установлена ли Windows Vista:

```
// Готовим структуру OSVERSIONINFOEX, сообщая,
// что нам нужна Windows Vista.
OSVERSIONINFOEX osver = { 0 };
osver.dwOSVersionlnfoSize = sizeof(osver);
osver.dwMajorVersion = 6;
osver.dwMinorVersion = 0;
osver.dwPlatformId = VER_PLATF0RM_WIN32_NT;

// формируем маску условий
DWORDLONG dwlConditionMask = 0;  // You MUST initialize this to 0.
VER_SET_C0NDITI0N(dwlConditionMask,  VER_MAJ0RVERSI0N,  VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask,  VER_MIN0RVERSI0N,  VER_EQUAL);
VER_SET_C0NDITI0N(dwlConditionMask,  VER_PLATFORMID,    VER_EQUAL);

// проверяем версию
if (VerifyVersionInfo(&osver,  VER_MAJORVERSION | VER_MIN0RVERSI0N  |
    VER_PLATF0RMID, dwlConditionMask)) {
```

```
    // хост-система точно соответствует Windows Vista
} else {
    // хост-система не является Windows Vista
}
```

Функция CreateProcess

Процесс создается при вызове вашим приложением функции *CreateProcess:*

```
BOOL CreateProcess(
    PCTSTR pszApplicationName,
    PTSTR pszCommandLine,
    PSECURITY_ATTRIBUTES psaProcess,
    PSECURITY.ATTRIBUTES psaThread,
    BOOL bInheritHandles,
    DWORD fdwCreate,
    PVOID pvEnvironment,
    PCTSTR pszCurDir,
    PSTARTUPINFO psiStartInfo,
    PPROCESS_INFORMATION ppiProcInfo);
```

Когда поток в приложении вызывает *CreateProcess,* система создает объект ядра «процесс» с начальным значением счетчика числа его пользователей, равным 1. Этот объект — не сам процесс, а компактная структура данных, через которую операционная система управляет процессом. (Объект ядра «процесс» следует рассматривать как структуру данных со статистической информацией о процессе.) Затем система создает для нового процесса виртуальное адресное пространство и загружает в него код и данные как для исполняемого файла, так и для любых DLL (если таковые требуются).

Далее система формирует объект ядра «поток» (со счетчиком, равным 1) для первичного потока нового процесса. Как и в первом случае, объект ядра «поток» — это компактная структура данных, через которую система управляет потоком. Первичный поток начинает с исполнения стартового кода из библиотеки C/C++, который, в конечном счете, вызывает функцию *WinMain, wWinMain, main* или *wmain* в вашей программе. Если системе удастся создать новый процесс и его первичный поток, *CreateProcess* вернет TRUE.

> **Примечание.** *CreateProcess* возвращает TRUE до окончательной инициализации процесса. Это означает, что на данном этапе загрузчик операционной системы еще не искал все необходимые DLL. Если он не сможет найти хотя бы одну из DLL или корректно провести инициализацию, процесс завершится. Но, поскольку *CreateProcess* уже вернула TRUE, родительский процесс ничего не узнает об этих проблемах.

На этом мы закончим общее описание и перейдем к подробному рассмотрению параметров функции *CreateProcess.*

Параметры pszApplicationName и pszCommandLine

Эти параметры определяют имя исполняемого файла, которым будет пользоваться новый процесс, и командную строку, передаваемую этому процессу. Начнем *с pszCommandLine*.

> **Примечание.** Обратите внимание на тип параметра *pszCommandLine:* PTSTR. Он означает, что *CreateProcess* ожидает передачи адреса строки, которая не является константой. Дело в том, что *CreateProcess* в процессе своего выполнения модифицирует переданную командную строку, но перед возвратом управления восстанавливает ее.
> Это очень важно: если командная строка содержится в той части образа вашего файла, которая предназначена только для чтения, возникнет ошибка доступа. Например, следующий код приведет к такой ошибке, потому что Visual C++ поместит строку «NOTEPAD» в память только для чтения:

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
CreateProcess(NULL, TEXT("NOTEPAD"), NULL, NULL,
   FALSE, 0, NULL, NULL, &si, &pi);
```

> Когда *CreateProcess* попытается модифицировать строку, произойдет ошибка доступа. (В прежних версиях Visual C++ эта строка была бы размещена в памяти для чтения и записи, и вызовы *CreateProcess* не приводили бы к ошибкам доступа.)
> Лучший способ решения этой проблемы — перед вызовом *CreateProcess* копировать константную строку во временный буфер:

```
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR szCommandLine[] = TEXT("NOTEPAD");
CreateProcess(NULL, szCommandLine, NULL, NULL,
   FALSE, 0, NULL, NULL, &si, &pi);
```

> Возможно, вас заинтересуют ключи /Gf и /GF компилятора Visual C++, которые исключают дублирование строк и запрещают их размещение в области только для чтения. (Также обратите внимание на ключ /ZI, который позволяет задействовать отладочную функцию Edit & Continue, поддерживаемую Visual Studio, и подразумевает активизацию ключа /GE) В общем, лучшее, что можете сделать вы, — использовать ключ /GF или создать временный буфер. А еще лучше, если Майкрософт исправит функцию *CreateProcess*, чтобы та не морочила нам голову. Надеюсь, в следующей версии Windows так и будет.
> Кстати, при вызове ANSI-версии *CreateProcess* в Windows 2000 таких проблем нет, поскольку в этой версии функции командная строка копируется во временный буфер (см. главу 2).

Параметр *pszCommandLine* позволяет указать полную командную строку, используемую функцией *CreateProcess* при создании нового процесса. Разбирая эту строку, функция полагает, что первый компонент в ней представляет собой имя исполняемого файла, который вы хотите запустить. Если в имени этого файла не указано расширение, она считает его EXE. Далее функция приступает к поиску заданного файла и делает это в следующем порядке:

1. Каталог, содержащий EXE-файл вызывающего процесса.
2. Текущий каталог вызывающего процесса.
3. Системный каталог Windows (System32, согласно *GetSystemDirectory*).
4. Основной каталог Windows.
5. Каталоги, перечисленные в переменной окружения PATH.

Конечно, если в имени файла указан полный путь доступа, система сразу обращается туда и не просматривает эти каталоги. Найдя нужный исполняемый файл, она создает новый процесс и проецирует код и данные исполняемого файла на адресное пространство этого процесса. Затем обращается к процедурам стартового кода из библиотеки C/C++. Тот в свою очередь, как уже говорилось, анализирует командную строку процесса и передает *(w)WinMain* адрес первого (за именем исполняемого файла) аргумента как *pszCmdLine*.

Все, о чем я сказал, произойдет, только если параметр *pszApplicationName* равен NULL (что и бывает в 99% случаев). Вместо NULL можно передать адрес строки с именем исполняемого файла, который надо запустить. Однако тогда придется указать не только его имя, но и расширение, поскольку в этом случае имя не дополняется расширением EXE автоматически. *CreateProcess* предполагает, что файл находится в текущем каталоге (если полный путь не задан). Если в текущем каталоге файла нет, функция не станет искать его в других каталогах, и на этом все закончится.

Но даже при указанном в *pszApplicationName* имени файла *CreateProcess* все равно передает новому процессу содержимое параметра *pszCommandLine* как командную строку. Допустим, вы вызвали *CreateProcess* так:

```
// размещаем строку пути в области памяти для чтения и записи
TCHAR szPath[] = TEXT("WORDPAD README.TXT");

// порождаем новый процесс
CreateProcess (TEXT("C:\\WINDOWS\\SYSTEM32\\NOTEPAD.EXE"), SZpATH, …);
```

Система запускает Notepad, а в его командной строке мы видим «WORDPAD README.TXT». Странно, да? Но так уж она работает, эта функция *CreateProcess*. Упомянутая возможность, которую обеспечивает параметр *pszApplicationName*, на самом деле введена в *CreateProcess* для поддержки подсистемы POSIX в Windows.

Параметры psaProcess, psaThread и blnheritHandles

Чтобы создать новый процесс, система должна сначала создать объекты ядра «процесс» и «поток» (для первичного потока процесса). Поскольку это объекты ядра, родительский процесс получает возможность связать с ними атрибуты защиты. Параметры *psaProcess* и *psaThread* позволяют определить нужные атрибуты защиты для объектов «процесс» и «поток» соответственно. В эти параметры можно занести NULL, и система закрепит за данными объектами дескрипторы защиты по умолчанию. В качестве альтернативы можно объявить и инициализировать две структуры SECURITY_ ATTRIBUTES; тем самым вы создадите и присвоите объектам «процесс» и «поток» свои атрибуты защиты. Структуры SECURITY_ATTRIBUTES для параметров *psaProcess* и *psaThread* используются и для того, чтобы какой-либо из этих двух объектов получил статус наследуемого любым дочерним процессом. (О теории, на которой построено наследование описателей объектов ядра, я рассказывал в главе 3.)

Короткая программа Inherit.cpp демонстрирует, как наследуются описатели объектов ядра. Будем считать, что процесс А порождает процесс В и заносит в параметр *psaProcess* адрес структуры SECURITYATTRIBUTES, в которой элемент *blnheritHandle* установлен как TRUE. Одновременно параметр *psaThread* указывает на другую структуру SECURITY_ATTRIBUTES, в которой значение элемента *blnheritHandle* — FALSE.

Создавая процесс В, система формирует объекты ядра «процесс» и «поток», а затем — в структуре, на которую указывает параметр *ppiProcInfo* (о нем поговорим позже), — возвращает их описатели процессу А, и с этого момента тот может манипулировать только что созданными объектами «процесс» и «поток».

Теперь предположим, что процесс А собирается вторично вызвать функцию *CreateProcess,* чтобы породить процесс С. Сначала ему нужно определить, стоит ли предоставлять процессу С доступ к своим объектам ядра. Для этого используется параметр *bInheritHandles.* Если он приравнен TRUE, система передает процессу С все наследуемые описатели. В этом случае наследуется и описатель объекта ядра «процесс» процесса В. А вот описатель объекта «первичный поток» процесса В не наследуется ни при каком значении *bInheritHandles.* Кроме того, если процесс А вызывает *CreateProcess,* передавая через параметр *bInheritHandles* значение FALSE, процесс С не наследует никаких описателей, используемых в данный момент процессом А.

```
Inherit.cpp

/******************************************************************
Module name: Inherit.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************/
```

```
#include <Windows.h>

int WINAPI _tWinMain (HINSTANCE hInstanceExe, HINSTANCE,
    PTSTR pszCmdLine, int nCmdShow) {
    // готовим структуру STARTUPINFO для создания процессов
    STARTUPINFO si = { sizeof(si) };
    SECURITY_ATTRIBUTES saProcess, saThread;
    PROCESS_INFORMATION piProcessB, piProcessC;
    TCHAR szPath[MAX_PATH];

    //Готовимся к созданию процесса B из процесса A;
    // описатель, идентифицирующий новый объект "процесс",
    // должен быть наследуемым.
    saProcess.nLength = sizeof(saProcess);
    saProcess.lpSecurityDescriptor = NULL;
    saProcess.bInheritHandle = TRUE;

    // Описатель, идентифицирующий новый объект "поток",
    // НЕ должен быть наследуемым.
    saThread.nLength = sizeof(saThread);
    saThread.lpSecurityDescriptor = NULL;
    saThread.bInheritHandle = FALSE;

    // порождаем процесс B
    _tcscpy_s(szPath, _countof(szPath), TEXT("ProcessB"));
    CreateProcess(NULL, szPath, &saProcess, &saThread,
        FALSE, 0, NULL, NULL, &si, &piProcessB);

    // структура pi содержит два описателя, относящиеся к процессу A:
    // hProcess, который идентифицирует объект "процесс" процесса B
    // и является наследуемым, и hThread, который идентифицирует объект
    // "первичный поток"процесса B и НЕ является наследуемым

    // готовимся создать процесс C из процесса A;
    // так как в psaProcess и psaThread передаются NULL, описатели
    // объектов "процесс"и "первичный поток" процесса C считаются
    // ненаследуемыми поумолчанию

    // если процесс A создаст еще один процесс, тот НЕ унаследует
    // описатели объектов "процесс" и "первичный поток" процесса C

    // поскольку в параметре bInheritHandles передается TRUE,
    // процесс C унаследует описатель, идентифицирующий объект
    //"процесс"процесса B, но НЕ описатель, идентифицирующий
    // объект "первичный поток" того же процесса
    _tcscpy_s(szPath, _countof(szPath), TEXT("ProcessC"));
```

```
    CreateProcess(NULL, szPath, NULL, NULL,
        TRUE, 0, NULL, NULL, &si, &piProcessC);

    return(0);
}
```

Параметр fdwCreate

Параметр *fdwCreate* определяет флаги, влияющие на то, как именно создается новый процесс. Следующие флаги комбинируются булевым оператором OR:

- Флаг DEBUG_PROCESS дает возможность родительскому процессу проводить отладку дочернего, а также всех процессов, которые последним могут быть порождены. Если этот флаг установлен, система уведомляет родительский процесс (он теперь получает статус отладчика) о возникновении определенных событий в любом из дочерних процессов (а они получают статус отлаживаемых).
- Флаг DEBUG_ONLY_THIS_PROCESS аналогичен флагу DEBUG_ PROCESS с тем исключением, что заставляет систему уведомлять родительский процесс о возникновении специфических событий только в одном дочернем процессе — его прямом потомке. Тогда, если дочерний процесс создаст ряд дополнительных, отладчик уже не уведомляется о событиях, «происходящих» в них; см. также статью «Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities, Part 2» на сайте MSDN (*http://msdn.microsoft.com/en-us/magazine/cc301686.aspx*) — в ней рассказывается об использовании этих флагов для написания отладчика и получения информации в реальном времени о DLL и потоках отлаживаемого приложения.
- Флаг CREATE_SUSPENDED позволяет создать процесс и в то же время приостановить его первичный поток. Это позволяет родительскому процессу модифицировать содержимое памяти в адресном пространстве дочернего, изменять приоритет его первичного потока или включать этот процесс в задание (job) до того, как он получит шанс на выполнение. Внеся нужные изменения в дочерний процесс, родительский разрешает выполнение его кода вызовом функции *ResumeThread* (см. главу 7).
- Флаг DETACHEDPROCESS блокирует доступ процессу, инициированному консольной программой, к созданному родительским процессом консольному окну и сообщает системе, что вывод следует перенаправить в новое окно. CUI-процесс, создаваемый другим CUI-процессом, по умолчанию использует консольное окно родительского процесса. (Вы, очевидно, заметили, что при запуске компилятора C из командного процессора новое консольное окно не создается; весь его вывод «подписывается» в нижнюю часть существующего консольного окна.) Таким образом, этот флаг заставляет новый процесс перенаправлять свой вывод в новое консольное окно вызовом функции *AllocConsole*.

- Флаг CREATE_NEW_CONSOLE приводит к созданию нового консольного окна для нового процесса. Имейте в виду, что одновременная установка флагов CREATE_NEW_CONSOLE и DETACHED_PROCESS недопустима.
- Флаг CREATE_NO_WINDOW не дает создавать никаких консольных окон для данного приложения и тем самым позволяет исполнять его без пользовательского интерфейса.
- Флаг CREATE_NEW_PROCESS_GROUP служит для модификации списка процессов, уведомляемых о нажатии клавиш Ctrl+C и Ctrl+Break. Если в системе одновременно исполняется несколько CUI-процессов, то при нажатии одной из упомянутых комбинаций клавиш система уведомляет об этом только процессы, включенные в группу. Указав этот флаг при создании нового CUI-процесса, вы создаете и новую группу.
- Флаг CREATE_DEFAULT_ERROR_MODE сообщает системе, что новый процесс не должен наследовать режимы обработки ошибок, установленные в родительском (см. раздел, где я рассказывал о функции *SetErrorMode).*
- Флаг CREATE_SEPARATE_WOW_VDM полезен только при запуске 16-разрядного Windows-приложения в 32-разрядной Windows. Если он установлен, система создает отдельную виртуальную DOS-машину (Virtual DOS-machine, VDM) и запускает 16-разрядное Windows-приложение именно в ней. (По умолчанию все 16-разрядные Windows-приложения выполняются в одной, общей VDM.) Выполнение приложения в отдельной VDM дает большое преимущество: «рухнув», приложение уничтожит лишь эту VDM, а программы, выполняемые в других VDM, продолжат нормальную работу. Кроме того, 16-разрядные Windows-приложения, выполняемые в раздельных VDM, имеют и раздельные очереди ввода. Это значит, что, если одно приложение вдруг «зависнет», приложения в других VDM продолжат прием ввода. Единственный недостаток работы с несколькими VDM в том, что каждая из них требует значительных объемов физической памяти. Windows 98 выполняет все 16-разрядные Windows-приложения только в одной VDM, и изменить тут ничего нельзя.
- Флаг CREATE_SHARED_WOW_VDM полезен только при запуске 16-разрядного Windows-приложения в Windows. По умолчанию все 16-разрядные Windows-приложения выполняются в одной VDM, если только не указан флаг CREATE_SEPARATE_WOW_VDM. Однако стандартное поведение Windows 2000 можно изменить, присвоив значение «yes» параметру DefaultSeparateVDM в разделе HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\WOW (После модификации этого параметра систему надо перезагрузить.) Установив значение «yes», но указав флаг CREATE_SHARED_WOW_VDM, вы вновь заставите Windows 2000 выполнять все 16-разрядные Windows-приложения в од-

ной VDM. Заметьте, что приложения могут определять ситуации, когда 32-разрядные процессы работают в 64-разряднои ОС, вызовом функции *IsWow64Process*. Этой функции передается описатель процесса как первый параметр, а также указатель на булево значение которое будет установлено в TRUE, если проверка даст положительный результат.

- Флаг CREATE_UNICODE_ENVIRONMENT сообщает системе, что блок переменных окружения дочернего процесса должен содержать Unicode-символы. По умолчанию блок формируется на основе ANSI-символов.
- Флаг CREATE_FORCEDOS заставляет систему выполнять программу MS-DOS, встроенную в 16-разрядное приложение OS/2.
- Флаг CREATE_BREAKAWAY_FROM_JOB позволяет процессу, включенному в задание, создать новый процесс, отделенный от этого задания (см. главу 5).
- Флаг EXTENDED_STARTUPINFO_PRESENT уведомляет ОС о том, что в параметре *psiStartInfo* передана структура STARTUPINFOEX. Параметр *fdwCreate* разрешает задать и класс приоритета процесса.

Однако это необязательно и даже, как правило, не рекомендуется; система присваивает новому процессу класс приоритета по умолчанию. Возможные классы приоритета перечислены в следующей таблице.

Табл. 4-5. Классы приоритета, которые задаются параметром fdwCreate

Класс приоритета	Флаговый идентификатор
Idle (простаивающий)	IDLE_PRIORITY_CLASS
Below normal (ниже обычного)	BELOW_NORMAL_PRIORITY_CLASS
Normal (обычный)	NORMAL_PRIORITY_CLASS
Above normal (выше обычного)	ABOVE_NORMAL_PRIORITY_CLASS
High (высокий)	HIGH_PRIORITY_CLASS
Realtime (реального времени)	REALTIME_PRIORITY_CLASS

Классы приоритета влияют на распределение процессорного времени между процессами и их потоками.

Параметр pvEnvironment

Параметр *pvEnvironment* указывает на блок памяти, хранящий строки переменных окружения, которыми будет пользоваться новый процесс Обычно вместо этого параметра передается NULL, в результате чего дочерний процесс наследует строки переменных окружения от родительского процесса. В качестве альтернативы можно вызвать функцию *GetEnviwnmentStnngs:*

```
PVOID GetEnvironmentStrings();
```

Она позволяет узнать адрес блока памяти со строками переменных окружения, используемых вызывающим процессом. Полученный адрес можно

занести в параметр *pvEnvironment* функции *CreateProcess*. (Именно это и делает *CreateProcess*, если вы передаете ей NULL *вместо pvEnvironment*.) Если этот блок памяти вам больше не нужен, освободите его, вызвав функцию *FreeEnvironmentStrings*:

```
BOOL FreeEnvironmentStrings(PTSTR pszEnvironmentBlock);
```

Параметр pszCurDir

Он позволяет родительскому процессу установить текущие диск и каталог для дочернего процесса. Если его значение — NULL, рабочий каталог нового процесса будет тем же, что и у приложения, его породившего. А если он отличен от NULL, то должен указывать на строку (с нулевым символом в конце), содержащую нужный диск и каталог. Заметьте, что в путь надо включать и букву диска.

Параметр psiStartInfo

Этот параметр указывает на структуру STARTUPINFO или STARTUPINFOEX:

```
typedef struct _STARTUPINFO {
    DWORD cb;
    PSTR lpReserved;
    PSTR lpDesktop;
    PSTR lpTitle;
    DWORD dwX;
    DWORD dwY;
    DWORD dwXSize;
    DWORD dwYSize;
    DWORD dwXCountChars;
    DWORD dwYCountChars;
    DWORD dwFillAttribute;
    DWORD dwFiags;
    WORD wShowWindow;
    WORD cbReserved2;
    PBYTE lpReserved2;
    HANDLE hStdlnput;
    HANDLE hStdOutput;
    HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;

typedef struct _STARTUPINFOEX {
    STARTUPINFO StartupInfo;
    struct _PROC_THREAD_ATTRIBUTE_LIST *lpAttributeList;
} STARTUPINFOEX, *LPSTARTUPINFOEX;
```

Элементы структуры STARTUPINFO используются Windows-функциями при создании нового процесса. Надо сказать, что большинство приложений порождает процессы с атрибутами по умолчанию. Но и в этом случае вы должны инициализировать все элементы структуры STARTUPINFO хотя бы нулевыми значениями, а в элемент *cb* — заносить размер этой структуры:

```
STARTUPINFO si = { sizeof(si)  };
CreateProcess(…, &si, …);
```

К сожалению, разработчики приложений часто забывают о необходимости инициализации этой структуры. Если вы не обнулите ее элементы, в них будет содержаться мусор, оставшийся в стеке вызывающего потока. Функция *CreateProcess,* получив такую структуру данных, либо создаст новый процесс, либо нет — все зависит от того, что именно окажется в этом мусоре.

Когда вам понадобится изменить какие-то элементы структуры, делайте это перед вызовом *CreateProcess*. Все элементы этой структуры подробно рассматриваются в таблице 4-6. Но заметьте, что некоторые элементы имеют смысл, только если дочернее приложение создает перекрываемое (overlapped) окно, а другие — если это приложение осуществляет ввод-вывод на консоль.

Табл. 4-6. Элементы структур STARTUPINFO и STARTUPINFOEX

Элемент	Окно или консоль	Описание
cb	То и другое	Содержит количество байтов, занимаемых структурой STARTUPINFO. Служит для контроля версий — на тот случай, если Майкрософт расширит эту структуру в будущем. Программа должна инициализировать *cb* как *sizeof (STARTUPINFO) sizeof (STARTUPINFOEX)*
lpReserved	То и другое	Зарезервирован. Инициализируйте как NULL
lpDesktop	То и другое	Идентифицирует имя рабочего стола, на котором запускается приложение. Если указанный рабочий стол существует, новый процесс сразу же связывается с ним. В ином случае система сначала создает рабочий стол с атрибутами по умолчанию, присваивает ему имя, указанное в данном элементе структуры, и связывает его с новым процессом. Если *lpDesktop* равен NULL (что чаще всего и бывает), процесс связывается с текущим рабочим столом
lpTitle	Консоль	Определяет заголовок консольного окна. Если *lpTitle* — NULL, в заголовок выводится имя исполняемого файла

Табл. 4-6. *(окончание)*

Элемент	Окно или консоль	Описание
dwX *dwY*	То и другое	Указывают *x*- и *y*-координаты (в пикселах) окна приложения. Эти координаты используются, только если дочерний процесс создает свое первое перекрываемое окно с идентификатором CW_USE-DEFAULT в параметре *x* функции *CreateWindow.* В приложениях, создающих консольные окна, данные элементы определяют верхний левый угол кон сольного окна
dwXSize *dwYSize*	То и другое	Определяют ширину и высоту (в пикселах) окна приложения. Эти значения используются, только если дочерний процесс создает свое первое перекрываемое окно с идентификатором CWUSEDE-FAULT в параметре *nWidth* функции *CreateWindow.* В приложениях, создающих консольные окна, данные элементы определяют ширину и высоту кон сольного окна.
dwXCountChars *dwYCountChars*	Консоль	Определяют ширину и высоту (в символах) консольных окон дочернего процесса.
dwFillAttribute	Консоль	Задает цвет текста и фона в консольных окнах дочернего процесса.
dwFlags	То и другое	См. ниже и следующую таблицу.
wShowWindow	Окно	Определяет, как именно должно выглядеть первое перекрываемое окно дочернего процесса. При первом вызове функции *ShowWindow* в качестве параметра вместо *nCmdShow* передается значение *wShowWindow*. При последующих вызовах *ShowWindow* значение *wShowWindow* передается только при передаче идентификатора SWSHOWDEFAULT. Чтобы параметр *wShowWindow* возымел действие, в *dwFlags* должен быть установлен флаг STARTF USESHOWWINDOW
cbReserved2	То и другое	Зарезервирован. Инициализируйте как 0.
lpReserved2	То и другое	Зарезервирован. Инициализируйте как NULL. Значения *cbReserved2* и *lpReserved2* используются библиотекой C для передачи информации при использовании *_dospawn* для запуска приложения. Реализацию можно увидеть в файлах dospawn.c and ioinitx, расположенных в папке VC\crt\src\ внутри каталога Visual Studio
hStdInput *hStdOutput* *hStdError*	Консоль	Определяют описатели буферов для консольного ввода-вывода. По умолчанию *hStdInput* идентифицирует буфер клавиатуры, а *hStdOutput* и *hStdError*— буфер консольного окна

Теперь, как я и обещал, обсудим элемент *dwFlags*. Он содержит набор флагов, позволяющих управлять созданием дочернего процесса. Большая часть флагов просто сообщает функции *CreateProcess*, содержат ли прочие элементы структуры STARTUPINFO полезную информацию или некоторые из них можно игнорировать. Список допустимых флагов приведен в следующей таблице.

Табл. 4-7. Флаги элемента dwFlags

Флаг	Описание
STARTF_USESIZE	Заставляет использовать элементы *dwXSize* и *dwYSize*
STARTF_USESHOWWINDOW	Заставляет использовать элемент *wShowWindow*
STARTF_USEPOSITION	Заставляет использовать элементы *dwX* и *dwY*
STARTF_USECOUNTCHARS	Заставляет использовать элементы *dwXCountChars* и *dwYCountChars*
STARTF_USEFILLATTRIBUTE	Заставляет использовать элемент *dwFillAttribute*
STARTF_USESTDHANDLES	Заставляет использовать элементы *hStdInput*, *hStdOutput* и *hStdError*
STARTF_RUN_FULLSCREEN	Приводит к тому, что консольное приложение на компьютере с процессором типа *x*86 запускается в полноэкранном режиме

Два дополнительных флага — STARTF_FORCEONFEEDBACK и STARTF_FORCEOFFFEEDBACK — позволяют контролировать форму указателя мыши в момент запуска нового процесса. Поскольку Windows поддерживает истинную вытесняющую многозадачность, можно запустить одно приложение и, пока оно инициализируется, поработать с другой программой. Для визуальной обратной связи с пользователем функция *CreateProcess* временно изменяет форму системного указателя мыши:

Указатель такой формы подсказывает: можно либо подождать чего-нибудь, что вот-вот случится, либо продолжить работу в системе. Если же вы укажете флаг STARTF_FORCEOFFFEEDBACK, *CreateProcess* не станет добавлять «песочные часы» к стандартной стрелке.

Флаг STARTF_FORCEONFEEDBACK заставляет *CreateProcess* отслеживать инициализацию нового процесса и в зависимости от результата проверки изменять форму указателя. Когда функция *CreateProcess* вызывается с этим флагом, указатель преобразуется в «песочные часы». Если спустя две секунды от нового процесса не поступает GUI-вызов, она восстанавливает исходную форму указателя.

Если же в течение двух секунд процесс все же делает GUI-вызов, *CreateProcess* ждет, когда приложение откроет свое окно. Это должно произойти в течение пяти секунд после GUI-вызова. Если окно не появилось, *CreateProcess* восстанавливает указатель, а появилось — сохраняет его в виде «песочных часов» еще на пять секунд. Как только приложение вызовет функцию *GetMessage,* сообщая тем самым, что оно закончило инициализацию, *CreateProcess* немедленно сменит указатель на стандартный и прекратит мониторинг нового процесса.

Теперь несколько слов об элементе *wShowWindow* структуры STARTUP-INFO. Этот элемент инициализируется значением, которое вы передаете в *(w)WinMain* через ее последний параметр, *nCmdShow.* Он позволяет указать, в каком виде должно появиться главное окно вашего приложения. В качестве значения используется один из идентификаторов, обычно передаваемых в *ShowWindow* (чаще всего SW_SHOWNORMAL или SW_SHOWMINNOACTIVE, но иногда и SW_SHOWDEFAULT).

После запуска программы из Explorer ее функция *(w)WinMain* вызывается с SW_SHOWNORMAL в параметре *nCmdShow.* Если же вы создаете для нее ярлык, то можете указать в его свойствах, в каком виде должно появляться ее главное окно. На рис. 4-3 показано окно свойств для ярлыка Notepad. Обратите внимание на список Run, в котором выбирается начальное состояние окна Notepad.

Когда вы активизируете этот ярлык из Explorer, последний создает и инициализирует структуру STARTUPINFO, а затем вызывает *CreateProcess.* Это приводит к запуску Notepad, а его функция *(w)WinMain* получает SW_SHOWMINNOACTIVE в параметре *nCmdShow.*

Таким образом, пользователь может легко выбирать, в каком окне запускать программу — нормальном, свернутом или развернутом.

Чтобы получить копию структуры STARTUPINFO, инициализированной родительским процессом, приложение может вызвать:

```
VOID GetStartupInfo(LPSTARTUPINFO pStartupInfo);
```

Анализируя эту структуру, дочерний процесс может изменять свое поведение в зависимости от значений ее элементов.

Эта функция всегда заполняет структуру STARTUPINFO, даже в дочерних процессах, созданных вызовом *CreateProcess* с передачей STARTUPINFOEX. Дополнительные атрибуты (см. ниже) имеют смысл лишь в адресном пространстве родительского процесса, где для них выделена память. Следовательно, нужен другой способ передачи наследуемых описателей, например, через командную строку.

В завершение этого раздела я расскажу о структуре STARTUPINFOEX. Сигнатура функции *CreateProcess* не менялась со времени появления Win32. В Майкрософт решили расширить ее возможности, но не стали ни изменять сигнатуру функции, ни создавать ее новые версии *(CreateProcessEx, CreateProcess2* и т. д.). В результате структура STARTUPINFOEX в дополне-

ние к полю *StartupInfo* получила поле *lpAttributeList,* в котором передаются дополнительные параметры, называемые атрибутами:

```
typedef struct _STARTUPINFOEXA {
    STARTUPINFOA StartupInfo;
    struct _PROC_THREAD_ATTRIBUTE_LIST *lpAttributeList;
} STARTUPINFOEXA, *LPSTARTUPINFOEXA;
typedef struct _STARTUPINFOEXW {
    STARTUPINFOW StartupInfo;
    struct _PROC_THREAD_ATTRIBUTE_LIST *iPAttributeList;
} STARTUPINFOEXW,   *LPSTARTUPINFOEXW;
```

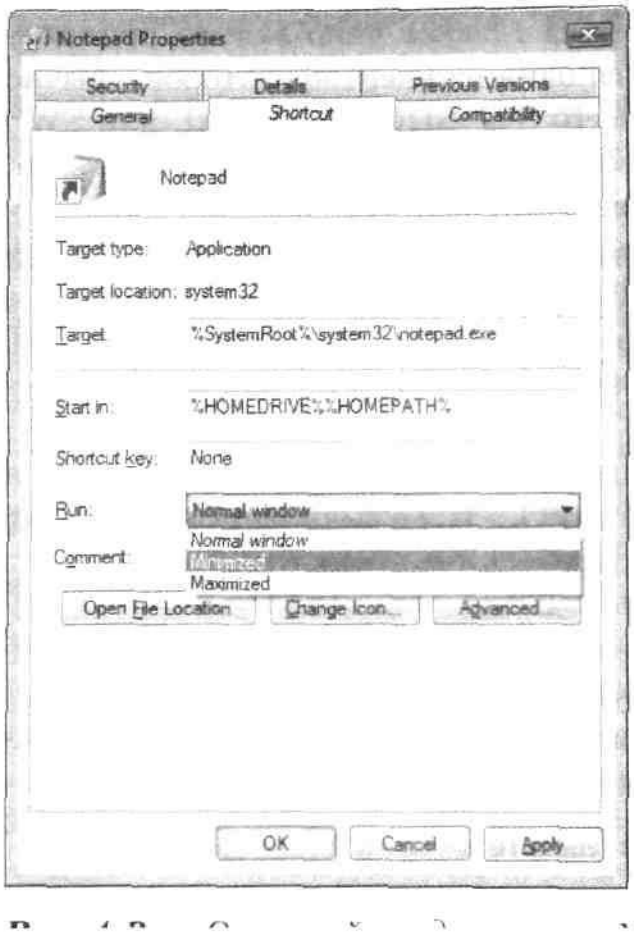


Рис. 4-3. *Окно свойств для ярлыка Notepad*

Список атрибутов состоит из пар «ключ - значение», представляющих атрибуты. В настоящее время задокументированы два ключа-атрибута:

- атрибут PROC_THREAD_ATTRIBUTE_HANDLE_LIST сообщает функции *CreateProcess,* какие именно описатели объектов ядра должен унаследовать дочерний процесс. Эти описатели должны быть созданы как наследуемые, при этом в их структуре SECURITY_ATTRIBUTES поле *bInheritHandle* должно быть установлено в TRUE (см. выше). Однако передавать TRUE в параметре *bInheritHandles* функции *CreateProcess* не обязательно. Использование этого атрибута позволяет выборочно передавать дочернему процессу наследуемые описатели. Это особенно важно для процессов, порождающих множество дочерних процессов в разных-

контекстах защиты: в этих случаях нельзя передавать каждому дочернему процессу все описатели, принадлежащие родительскому процессу, так как это создает угрозу безопасности;

- в атрибуте PROC_THREAD_ATTRIBUTE_PARENT_PROCESS передается значение описателя процесса. Заданный процесс (вместе с его наследуемыми описателями, настройками привязки к процессорам, классом приоритета, квотами, маркером пользователя и связанным заданием) будет использован при вызове *CreateProcess* в качестве родительского процесса вместо текущего процесса. Обратите внимание, что такая «смена» родительского процесса не касается процесса отладчика, который по-прежнему будет получать контролировать созданный им отлаживаемый процесс и получать от него все отладочные уведомления. Однако показанная ранее программа ToolHelp API будет отображать процесс, указанный этим атрибутом, как родительский для процесса, созданного с передачей данного атрибута.

Список атрибутов устроен так, что для создания пустого списка атрибутов необходимо дважды вызвать следующую функцию:

```
BOOL InitializeProcThreadAttributeList(
    PPROC_THREAD_ATTRIBUTE_LIST pAttributeList,
    DWORD dwAttributeCount,
    DWORD dwFlags,
    PSIZE_T pSize);
```

Заметьте, что параметр *dwFlags* зарезервирован, в нем всегда следует передавать 0. Первый вызов необходим, чтобы узнать размер блока памяти, необходимого Windows для хранения атрибутов:

```
SIZE_T cbAttributeListSize = 0;
BOOL bReturn = InitializeProcThreadAttributeList(
    NULL,  1,  0,  &cbAttributeListSize);
// bReturn = FALSE, но GetLastError() возвращает ERROR_INSUFFICIENT_BUFFER
```

Размер блока будет рассчитан на основе числа передаваемых атрибутов, заданного *dwAttributeCount*, и записан в переменную SIZE_T, на которую указывает *pSize:*

```
pAttributeList = (PPROC_THREAD_ATTRIBUTE_LIST)
HeapAlloc(GetProcessHeap(),  0,  cbAttributeListSize);
```

После выделения памяти для списка атрибутов функция *InitializeProcThradAttributeList* вызывается снова для инициализации его содержимого:

```
bReturn = InitializeProcThreadAttributeList(
    pAttributeList,  1,  0,  AcbAttributeListSize);
```

После выделения памяти и инициализации атрибутов следует добавить необходимые пары «ключ — значение» с помощью этой функции:

```
BOOL UpdateProcThreadAttribute(
    PPROC_THREAD_ATTRIBUTE_LIST pAttributeList,
    DWORD dwFlags,
    DWORD_PTR Attribute,
    PVOID pValue,
    SIZE_T cbSize,
    PVOID pPreviousValue,
    PSIZE_T pReturnSize);
```

Параметр *pAttributeList* содержит инициализированный список атрибутов, для которого ранее была выделена память; к этому списку функция будет добавлять заданные пары «ключ - значение». Параметр *Attribute* представляет ключ в составе пары и принимает значение PROC_THREAD_ATTRIBUTE_PARENT_ PROCESS либо PROC_THREAD_ATTRIBUTE_HANDLE_LIST. В первом случае параметр *pValue* должен указывать на переменную, содержащую описатель нового родительского процесса, а параметр *cbSize* должен быть равен *sizeof (HANDLE).* Во втором случае *pValue* должен указывать на начало массива описателей наследуемых объектов ядра, которые необходимо передать дочернему процессу, а *cbSize* должен быть равен произведению *sizeof (HANDLE)* и числа описателей. Параметры *dwFlags, pPreviousValue* и *pReturnSize* зарезервированы и должны быть установлены в 0, NULL и NULL, соответственно.

> **Внимание!** Если требуется одновременно передать оба атрибута, помните, что описатели, связанные с *PROC_THREAD_ATTRIBUTE_HANDLELIST*, должны быть действительными в новом родительском процессе, представленном *PROC_THREAD_ATTRIBUTE_PARENT_PROCESS* (поскольку они наследуются от него), а не в текущем процессе, вызывающем *CreateProcess*.

Перед вызовом *CreateProcess* с установленным в *dwCreationFlags* флагом EXTENDED_STARTUPINFO_PRESENT необходимо определить переменную STARTUPINFOEX (с только что инициализированным списком атрибутов в поле *pAttributeList),* которая будет передана как параметр pStartupInfo:

```
STARTUPINFOEX esi = { sizeof(STARTUPINFOEX) };
esi.lpAttributeList = pAttributeList;
bReturn = CreateProcess(
    …, EXTENDED_STARTUPINFO_PRESENT, …
    &esi.StartupInfo, …);
```

Если эти параметры вам больше не нужны, следует освободить занятую ими память, предварительно очистить список атрибутов вызовом следующего метода:

```
VOID DeleteProcTh readAttributeList(
    PPROC_THREAD_ATTRIBUTE_LIST pAttributeList);
```

Параметр ppiProcInfo

Параметр *ppiProcInfo* указывает на структуру PROCESS_INFORMATION, которую вы должны предварительно создать; ее элементы инициализируются самой функцией *CreateProcess*. Структура представляет собой следующее:

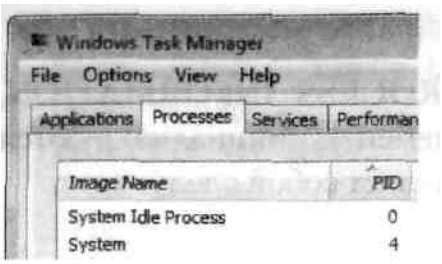
```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD  dwProcessId;
    DWORD  dwThreadId;
} PROCESS_INFORMATION;
```

Как я уже говорил, создание нового процесса влечет за собой создание объектов ядра «процесс» и «поток». В момент создания система присваивает счетчику каждого объекта начальное значение — единицу. Далее функция *CreateProcess* (перед самым возвратом управления) открывает объекты «процесс» и «поток» и заносит их описатели, специфичные для данного процесса, в элементы *hProcess* и *hThread* структуры PROCESS_INFORMATION. Когда *CreateProcess* открывает эти объекты, счетчики каждого из них увеличиваются до 2.

Это означает, что, перед тем как система сможет высвободить из памяти объект «процесс», процесс должен быть завершен (счетчик уменьшен до 1), а родительский процесс обязан вызвать функцию *CloseHandle* (и тем самым обнулить счетчик). То же самое относится и к объекту «поток»: поток должен быть завершен, а родительский процесс должен закрыть описатель объекта «поток». Подробнее об освобождении объектов «поток» см. раздел «Дочерние процессы» в этой главе.

> **Примечание.** Не забываете закрывать описатели дочернего процесса и его первичного потока, иначе, пока вы не закроете свое приложение, будет происходить утечка ресурсов. Конечно, система высвободит все эти ресурсы после завершения вашего процесса, но хорошо написанная программа должна сама закрывать описатели дочернего процесса и его первичного потока (вызовом *CloseHandle),* как только необходимость в них отпадает. Пропуск этой операции — одна из самых частых ошибок. Почему-то многие разработчики считают, будто закрытие описателя процесса или потока заставляет систему уничтожить этот процесс или поток. Это абсолютно неправильно. Закрывая описатель, вы просто сообщаете системе, что статистические данные для этого процесса или потока вас больше не интересуют, но процесс или поток продолжает исполняться системой до тех пор, пока он сам не завершит себя.

Созданному объекту ядра «процесс» присваивается уникальный идентификатор; ни у каких других объектов этого типа в системе не может быть одинаковых идентификаторов. Это же касается и объектов ядра «поток». Причем идентификаторы процесса и потока тоже разные, и их значения никогда не бывают нулевыми.



Завершая свою работу, *CreateProcess* заносит значения идентификаторов в элементы *dwProcessId* и *dwThreadId* структуры PROCESS_INFORMATION. Эти идентификаторы просто облегчают определение процессов и потоков в системе; их используют, как правило, лишь специализированные утилиты вроде Task Manager. Обратите внимание, что Task Manager присваивает идентификатор «О» процессу «System Idle» («Бездействие системы», см. ниже). Однако на самом деле такого процесса нет, просто в Task Manager так обозначается поток, исполняемый в то время, когда не исполняется никакой другой код. Число потоков «процесса» System Idle всегда равно числу установленных в системе процессоров. Этот «процесс» отображает процентную долю времени процессора, не использованную настоящими процессами.

Подчеркну еще один чрезвычайно важный момент (особенно если вы пишете программы, учитывающие процессы и потоки и по их идентификаторам): система способна повторно использовать идентификаторы процессов и потоков. Например, при создании процесса система формирует объект «процесс», присваивая ему идентификатор со значением, допустим, 124. Создавая новый объект «процесс», система уже не присвоит ему данный идентификатор. Но после выгрузки из памяти первого объекта следующему создаваемому объекту «процесс» может быть присвоен тот же идентификатор — 124.

Эту особенность нужно учитывать при написании кода, избегая ссылок на неверный объект «процесс» (или «поток»). Действительно, затребовать и сохранить идентификатор процесса несложно, но задумайтесь, что получится, если в следующий момент этот процесс будет завершен, а новый получит тот же идентификатор: сохраненный ранее идентификатор уже связан совсем с другим процессом.

Узнать идентификатор текущего процесса можно вызовом *GetCurrentProcessId,* а идентификатор текущего потока — вызовом функции *GetCurrentThreadId.* Кроме того, идентификатор процесса можно узнать по его описателю, вызвав GetProcessId (а в случае потока — *GetThreadId).* И последнее (по порядку, но не по важности): обладая описателем потока, можно узнать идентификатор процесса, которому этот поток принадлежит, для этого достаточно вызвать *GetProcessIdOfThread.*

Программе иногда приходится определять свой родительский процесс. Однако родственные связи между процессами существуют лишь на стадии создания дочернего процесса. Непосредственно перед началом исполнения кода в дочернем процессе Windows перестает учитывать его родственные

связи. В предыдущих версиях Windows не было функций, которые позволяли бы программе обращаться с запросом к ее родительскому процессу. Но ToolHelp-функции, появившиеся в современных версиях Windows, сделали это возможным. С этой целью вы должны использовать структуру PROCESSENTRY32: ее элемент *th32ParentProcessID* возвращает идентификатор «родителя» данного процесса. Тем не менее, если вашей программе нужно взаимодействовать с родительским процессом, от идентификаторов лучше отказаться. Почему — я уже говорил. Для определения родительского процесса существуют более надежные механизмы: объекты ядра, описатели окон и т. д.

Единственный способ добиться того, чтобы идентификатор процесса или потока не использовался повторно, — не допускать разрушения объекта ядра «процесс» или «поток». Если вы только что создали новый процесс или поток, то можете просто не закрывать описатели на эти объекты — вот и все. А по окончании операций с идентификатором, вызовите функцию *Close-Handle* и освободите соответствующие объекты ядра. Однако для дочернего процесса этот способ не годится, если только он не унаследовал описатели объектов ядра от родительского процесса.

Завершение процесса

Процесс можно завершить четырьмя способами:

- входная функция первичного потока возвращает управление (рекомендуемый способ);
- один из потоков процесса вызывает функцию *ExitProcess* (нежелательный способ);
- поток другого процесса вызывает функцию *TerminateProcess* (тоже нежелательно);
- все потоки процесса умирают по своей воле (большая редкость).

В этом разделе мы обсудим только что перечисленные способы завершения процесса, а также рассмотрим, что на самом деле происходит в момент его окончания.

Возврат управления входной функцией первичного потока

Приложение следует проектировать так, чтобы его процесс завершался только после возврата управления входной функцией первичного потока. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших первичному потоку. При этом:

- любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система освобождает память, которую занимал стек потока;

- L система устанавливает код завершения процесса (поддерживаемый объектом ядра «процесс») — его и возвращает ваша входная функция;
- счетчик пользователей данного объекта ядра «процесс» уменьшается на 1.

Функция ExitProcess

Процесс завершается, когда один из его потоков вызывает *ExitProcess*:

```
VOID ExitProcess(UINT fuExitCode);
```

Эта функция завершает процесс и заносит в параметр *fuExitCode* код завершения процесса. Возвращаемого значения у *ExitProcess* нет, так как результат ее действия - завершение процесса. Если за вызовом этой функции в программе присутствует какой-нибудь код, он никогда не исполняется.

Когда входная функция *(WinMain, wWinMain, main* или *wmain)* в вашей программе возвращает управление, оно передается стартовому коду из библиотеки C/C++, и тот проводит очистку всех ресурсов, выделенных им процессу, а затем обращается к *ExitProcess,* передавая ей значение, возвращенное входной функцией. Вот почему возврат управления входной функцией первичного потока приводит к завершению всего процесса. Обратите внимание, что при завершении процесса прекращается выполнение и всех других его потоков.

Кстати, в документации из Platform SDK утверждается, что процесс не завершается до тех пор, пока не завершится выполнение всех его потоков. Это, конечно, верно, но тут есть одна тонкость. Стартовый код из библиотеки C/C++ обеспечивает завершение процесса, вызывая *ExitProcess* после того, как первичный поток вашего приложения возвращается из входной функции. Однако, вызвав из нее функцию *ExitThread* (вместо того чтобы вызвать *ExitProcess* или просто вернуть управление), вы завершите первичный поток, но не сам процесс — если в нем еще выполняется какой-то другой поток (или потоки).

Заметьте, что такой вызов *ExitProcess* или *ExitThread* приводит к уничтожению процесса или потока, когда выполнение функции еще не завершилось. Что касается операционной системы, то здесь все в порядке: она корректно очистит все ресурсы, выделенные процессу или потоку. Но в приложении, написанном на C/C++, следует избегать вызова этих функций, так как библиотеке C/C++ скорее всего не удастся провести должную очистку. Взгляните на этот код:

```
«include <windows.h>
«include <stdio.h>

class CSomeObj {
public:
    CSomeObj()  { printf("Constructor\r\n"); }
    ~CSomeObJ() { printf("CDestructor\r\n"); }
};
```

```
CSomeObj g.GlobalObj;

void main () {
    CSomeObj LocalObj;
    ExitProcess(0);                          // этого здесь не должно быть

    // в конце этой функции компилятор автоматически вставил код
    // для вызова деструктора LocalObj, но ExitProcess не дает его выполнить
}
```

При исполнении эта программа выводит следующие строки;

```
Constructor
Constructor
```

Код конструирует два объекта: глобальный и локальный. Но вы никогда не увидите строку *Destructor*. C++-объекты не разрушаются должным образом из-за того, что *ExitProcess* форсирует уничтожение процесса и библиотека C/C++ не получает шанса на очистку.

Как я уже говорил, никогда не вызывайте *ExitProcess* в явном виде. Если я уберу из предыдущего примера вызов *ExitProcess*, программа выведет такие строки:

```
Constructor
Constructor
Destructor
Destructor
```

Простой возврат управления от входной функции первичного потока позволил библиотеке C/C++ провести нужную очистку и корректно разрушить C++-объекты. Кстати, все, о чем я рассказал, относится не только к объектам, но и ко многим другим вещам, которые библиотека C/C++ делает для вашего процесса.

> **Примечание.** Явные вызовы *ExitProcess* и *ExitThread* — распространенная ошибка, которая мешает правильной очистке ресурсов. В случае *ExitThread* процесс продолжает работать, но при этом весьма вероятна утечка памяти или других ресурсов.

Функция TerminateProcess

Вызов функции *TerminateProcess* тоже завершает процесс:

```
BOOL TerminateProcess(
    HANDLE hProcess,
    UINT fuExitCode);
```

Главное отличие этой функции от *ExitProcess* в том, что ее может вызвать любой поток и завершить любой процесс. Параметр *hProcess* идентифициру-

ет описатель завершаемого процесса, а в параметре *fuExitCode* возвращается код завершения процесса.

Пользуйтесь *TerminateProcess* лишь в том случае, когда иным способом завершить процесс не удается. Процесс не получает абсолютно никаких уведомления о том, что он завершается, и приложение не может ни выполнить очистку, ни предотвратить свое неожиданное завершение (если оно, конечно, не использует механизмы защиты). При этом теряются все данные, которые процесс не успел переписать из памяти на диск.

Процесс действительно не имеет ни малейшего шанса самому провести очистку, но операционная система высвобождает все принадлежавшие ему ресурсы: возвращает себе выделенную им память, закрывает любые открытые файлы, уменьшает счетчики соответствующих объектов ядра и разрушает все его User- и GDI-объекты.

По завершении процесса (не важно каким способом) система гарантирует: после него ничего не останется - даже намеков на то, что он когда-то выполнялся. *Завершенный процесс не оставляет за собой никаких следов.* Надеюсь, я сказал ясно.

> **Примечание.** *TerminateProcess* — функция асинхронная, т. е. она сообщает системе, что вы хотите завершить процесс, но к тому времени, когда она вернет управление, процесс может быть еще не уничтожен. Так что, если вам нужно точно знать момент завершения процесса, используйте *WaitForSingleObject* (см. главу 9) или аналогичную функцию, передав ей описатель этого процесса.

Когда все потоки процесса уходят

В такой ситуации (а она может возникнуть, если все потоки вызвали *ExitThread* или их закрыли вызовом *TerminateThread)* операционная система больше не считает нужным «содержать» адресное пространство данного процесса. Обнаружив, что в процессе не исполняется ни один поток, она немедленно завершает его. При этом код завершения процесса приравнивается коду завершения последнего потока.

Что происходит при завершении процесса

А происходит вот что:

1. Выполнение всех потоков в процессе прекращается.
2. Все User- и GDI-объекты, созданные процессом, уничтожаются, а объекты ядра закрываются (если их не использует другой процесс).
3. Код завершения процесса меняется со значения STILL_ACTIVE на код, переданный в *ExitProcess* или *TerminateProcess.*
4. Объект ядра «процесс» переходит в свободное, или незанятое (signaled), состояние. (Подробнее на эту тему см. главу 9.) Прочие потоки в системе могут приостановить свое выполнение вплоть до завершения данного процесса.

5. Счетчик объекта ядра «процесс» уменьшается на 1.

Связанный с завершаемым процессом объект ядра не высвобождается, пока не будут закрыты ссылки на него и из других процессов. В момент завершения процесса система автоматически уменьшает счетчик пользователей этого объекта на 1, и объект разрушается, как только его счетчик обнуляется. Кроме того, закрытие процесса не приводит к автоматическому завершению порожденных им процессов.

По завершении процесса его код и выделенные ему ресурсы удаляются из памяти. Однако область памяти, выделенная системой для объекта ядра «процесс», не освобождается, пока счетчик числа его пользователей не достигнет нуля. А это произойдет, когда все прочие процессы, создавшие или открывшие описатели для ныне покойного процесса, уведомят систему (вызовом *CloseHandle)* о том, что ссылки на этот процесс им больше не нужны.

Описатели завершенного процесса уже мало на что пригодны. Разве что родительский процесс, вызвав функцию *GetExitCodeProcess,* может проверить, завершен ли процесс, идентифицируемый параметром *hProcess,* и, если да, определить код завершения:

```
BOOL GetExitCodeProcess(
    HANDLE hProcess,
    PDWORD pdwExitCode);
```

Эта функция анализирует объекта ядра «процесс» (заданный параметром *hProcess)* и извлекает элемент, содержащий структуру данных с кодом завершения процесса. Код завершения возвращается в значении DWORD, на которое указывает параметр *pdwExitCode.*

Вызывать эту функцию можно в любое время. Если на момент вызова *GetExitCodeProcess* процесс еще не завершился, в DWORD заносится идентификатор STILL_ACTIVE (определенный как 0x103). А если он уничтожен, функция возвращает реальный код его завершения.

Вероятно, вы подумали, что можно написать код, который, периодически вызывая функцию *GetExitCodeProcess* и проверяя возвращаемое ею значение, определял бы момент завершения процесса. В принципе такой код мог бы сработать во многих ситуациях, но он был бы неэффективен. Как правильно решить эту задачу, я расскажу в следующем разделе.

В завершение раздела позвольте мне напомнить о том, что если статистические данные процесса вам больше не нужны, сообщите об этом системе вызовом функции *CloseHandle.* Если процесс уже завершен, *CloseHandle* уменьшит до нуля счетчик объекта ядра, представляющего этот процесс, что приведет к его разрушению.

Дочерние процессы

При разработке приложения часто бывает нужно, чтобы какую-то операцию выполнял другой блок кода. Поэтому — хочешь, не хочешь — прихо-

дится постоянно вызывать функции или подпрограммы. Но вызов функции приводит к приостановке выполнения основного кода вашей программы до возврата из вызванной функции. Альтернативный способ — передать выполнение какой-то операции другому потоку в пределах данного процесса (поток, разумеется, нужно сначала создать). Это позволит основному коду программы продолжить работу в то время, как дополнительный поток будет выполнять другую операцию. Прием весьма удобный, но, когда основному потоку потребуется узнать результаты работы другого потока, вам не избежать проблем, связанных с синхронизацией.

Есть еще один прием: ваш процесс порождает дочерний и возлагает на него выполнение части операций. Будем считать, что эти операции очень сложны. Допустим, для их реализации вы просто создаете новый поток внутри того же процесса. Вы пишете тот или иной код, тестируете его и получаете некорректный результат — может, ошиблись в алгоритме или запутались в ссылках и случайно перезаписали какие-нибудь важные данные в адресном пространстве своего процесса. Так вот, один из способов защитить адресное пространство основного процесса от подобных ошибок как раз и состоит в том, чтобы передать часть работы отдельному процессу. Далее можно или подождать, пока он завершится, или продолжить работу параллельно с ним.

К сожалению, дочернему процессу, по-видимому, придется оперировать с данными, содержащимися в адресном пространстве родительского процесса. Было бы неплохо, чтобы он работал исключительно в своем адресном пространстве, а в «вашем» — просто считывал нужные ему данные; тогда он не сможет что-то испортить в адресном пространстве родительского процесса. В Windows предусмотрено несколько способов обмена данными между процессами: DDE (Dynamic Data Exchange), OLE, каналы (pipes), почтовые ящики (mailslots) и т. д. А один из самых удобных способов, обеспечивающих совместный доступ к данным, — использование файлов, проецируемых в память (memory-mapped files). (Подробнее на эту тему см. главу 17.)

Если вы хотите создать новый процесс, заставить его выполнить какие-либо операции и дождаться их результатов, напишите примерно такой код:

```
PROCESS_INF0RMATI0N pi;
DWORD dwExitCode;

// порождаем дочерний процесс
BOOL fSuccess = CreateProcess(…, &pi);
if (fSuccess) {
   // закрывайте описатель потока, как только потребность в нем отпадает!
   CloseHandle(pi.hThread);
   // приостанавливаем выполнение родительского процесса,
   // пока не завершится дочерний процесс
   WaitForSingleObject(pi.hProcess, INFINITE);
```

```
    // дочерний процесс завершился; получаем код его завершения
    GetExitCodeProcess(pi.hProcess, idwExitCode);

    // закрывайте описатель потока, как только потребность в нем отпадает!
    CloseHandle(pi.hProcess);
}
```

В этом фрагменте кода мы создали новый процесс и, если это прошло успешно, вызвали функцию *WaitForSingleObject*

```
DWORD WaitForSingleObject(HANDLE hObject,  DWORD dwTimeout);
```

Подробное рассмотрение данной функции мы отложим до главы 9, а сейчас ограничимся одним соображением. Функция задерживает выполнение кода до тех пор, пока объект, определяемый параметром *hObject*, не перейдет в свободное (незанятое) состояние. Объект «процесс» переходит в такое состояние при его завершении. Поэтому вызов *WaitForSingleObject* приостанавливает выполнение потока родительского процесса до завершения порожденного им процесса. Когда *WaitForSingleObject* вернет управление, вы узнаете код завершения дочернего процесса через функцию *GetExitCodeProcess*.

Обращение к *CloseHandle* в приведенном выше фрагменте кода заставляет систему уменьшить значения счетчиков объектов «поток» и «процесс» до нуля и тем самым освободить память, занимаемую этими объектами.

Вы, наверное, заметили, что в этом фрагменте я закрыл описатель объекта ядра «первичный поток» (принадлежащий дочернему процессу) сразу после возврата из *CreateProcess*. Это не приводит к завершению первичного потока дочернего процесса — просто уменьшает счетчик, связанный с упомянутым объектом. А вот почему это делается — и, кстати, даже рекомендуется делать — именно так, станет ясно из простого примера. Допустим, первичный поток дочернего процесса порождает еще один поток, а сам после этого завершается. В этот момент система может высвободить объект «первичный поток» дочернего процесса из памяти, если у родительского процесса нет описателя данного объекта. Но если родительский процесс располагает таким описателем, система не сможет удалить этот объект из памяти до тех пор, пока и родительский процесс не закроет его описатель.

Запуск обособленных дочерних процессов

Что ни говори, но чаще приложение все-таки создает другие процессы как *обособленные* (detached processes). Это значит, что после создания и запуска нового процесса родительскому процессу нет нужды с ним взаимодействовать или ждать, пока тот закончит работу. Именно так и действует Explorer: запускает для пользователя новые процессы, а дальше его уже не волнует, что там с ними происходит.

Чтобы обрубить все пуповины, связывающие Explorer с дочерним процессом, ему нужно (вызовом *CloseHandle*) закрыть свои описатели, связан-

ные с новым процессом и его первичным потоком. Приведенный ниже фрагмент кода демонстрирует, как, создав процесс, сделать его обособленным:

```
PROCESS_INFORMATION pi;

// порождаем дочерний процесс
BOOL fSuccess = CreateProcess(…, &pi):
if (fSuccess) {

    // Разрешаем системе уничтожить объекты ядра "процесс" и "поток"
    // сразу после завершения дочернего процесса.
    CloseHandle(pi.hTh read);
    CloseHandle(pi.hP rocess);
}
```

Работа администратора с пользовательскими полномочиями

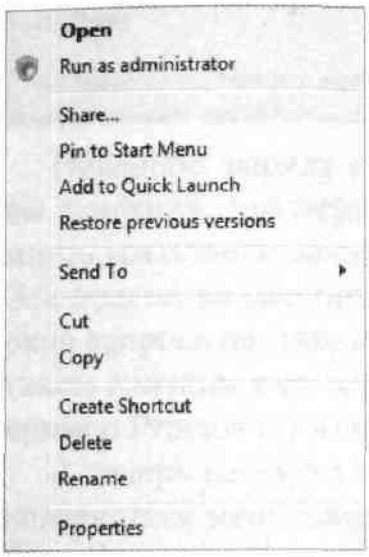
Создатели Windows Vista существенно повысили безопасность пользователей новой ОС благодаря использованию новых технологий. Для разработчиков приложений особенно важна, несомненно, технология под названием User Account Control (UAC).

По наблюдениям Майкрософт, большинство пользователей работает в Windows под администраторской учетной записью, привилегии которой предоставляют им почти ничем не ограниченный доступ к жизненно важным ресурсам системы. При входе пользователя в прежние версии Windows под привилегированной учетной записью создавался маркер защиты, который операционная система затем проверяла при обращении любых программ к защищенным ресурсам. Этот маркер связывался со всеми новыми процессами, начиная с Explorer, передавался их дочерним процессам и т. д. При этом вредоносный код, загруженный из Интернета либо принятый вместе с сообщением электронной почты, наследовал высокие привилегии администратора от исполняющего этот код «добропорядочного» приложения. Так он получал возможность творить на зараженной машине все, что угодно, вплоть до запуска других процессов, передавая им свои высокие привилегии.

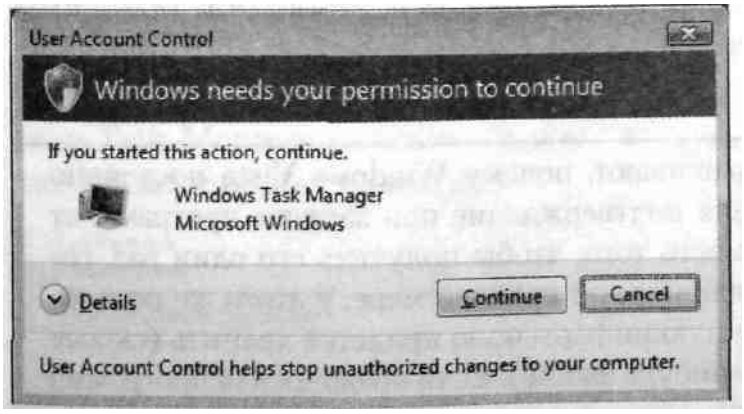
В Windows Vista при входе под привилегированной учетной записью, такой как учетная запись администратора, в дополнение к стандартному маркеру защиты этой учетной записи, создается т.н. отфильтрованный маркер защиты (filtered token) с привилегиями обычного пользователя. Далее отфильтрованный маркер связывается со всем процессами, запущенными от имени этого пользователя, начиная с Explorer. Но как, спросите вы, работая с таким маркером защиты, получать доступ к защищенным ресурсам? Если говорить кратко, никак: процесс с ограниченными привилегиями не может обращаться к защищенным ресурсам, для доступа к которым нужен высо-

кий уровень привилегий. Но я не ограничусь этим, а все же расскажу, как использовать при разработке преимущества технологии UAC.

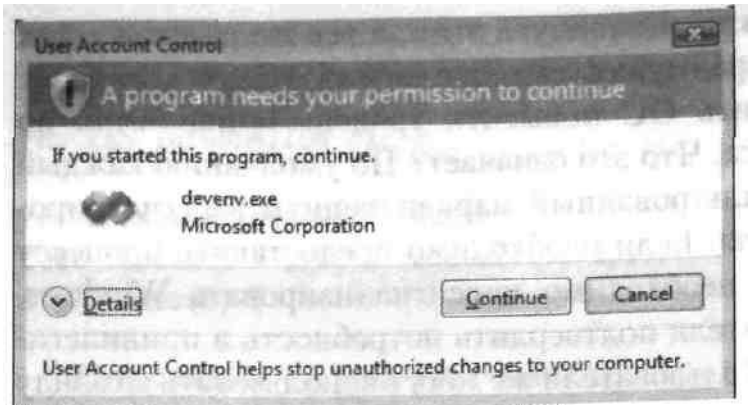
Во-первых, можно попросить ОС повысить уровень привилегий, но только для отдельного процесса. Что это означает? По умолчанию каждый новый процесс получает отфильтрованный маркер защиты зарегистрированного в системе пользователя. Если необходимо предоставить процессу дополнительные привилегии, необходимо просигнализировать Windows, чтобы она попросила пользователя подтвердить потребность в привилегиях *до* запуска этого процесса. Пользователи же могут использовать команду Run As Administrator, доступную в контекстном меню исполняемых файлов, вызываемом щелчком правой кнопки в Explorer.



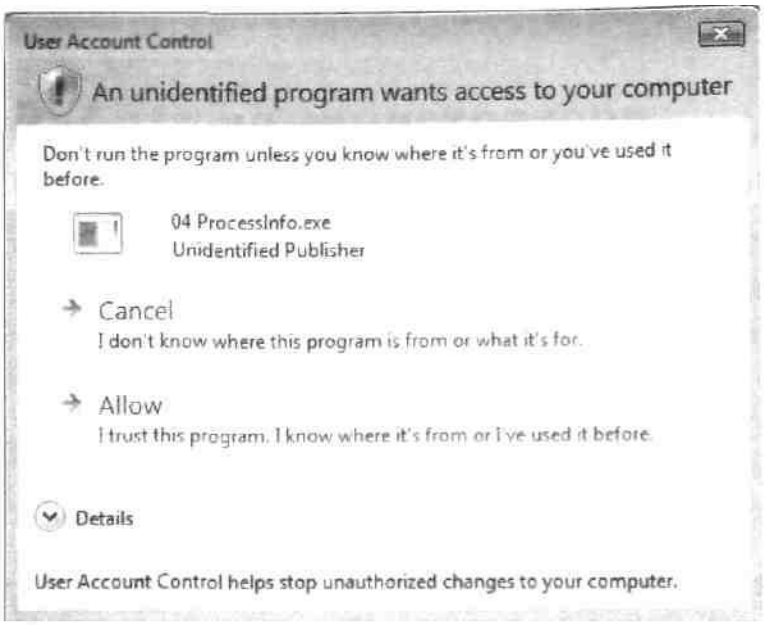
Если вы работаете под учетной записью администратора, откроется показанное ниже окно с запросом подтверждения повышения привилегий до уровня, заданного неотфильтрованным маркером защиты. Такие окна бывают трех типов. Если запускаемая программа входит в состав системы, отображается окно запроса подтверждения с синей полосой наверху:



Если файл программы имеет цифровую подпись, отображается окно с серой полосой, вселяющей меньшую уверенность:

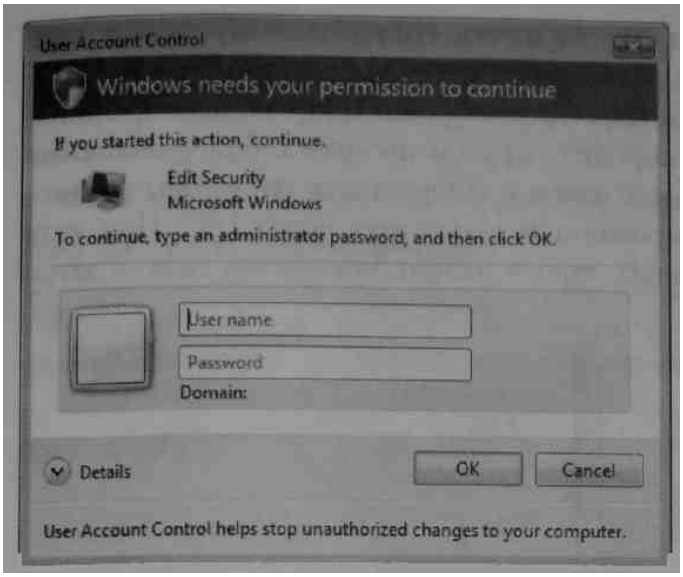


Наконец, при попытке запуска неподписанного приложения выводится окно с оранжевой полосой и соответствующими предостережениями:



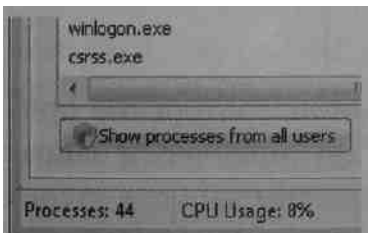
Если пользователь вошел в систему под учетной записью обычного пользователя, откроется окно для ввода удостоверений привилегированной учетной записи — так пользователи с обычным уровнем привилегий могут выполнять действия, требующие администраторских полномочий. Такой вход в систему называется входом «через плечо» (over-the-shoulder, OTS logon) другого пользователя.

> **Примечание.** Многие спрашивают, почему Windows Vista постоянно запрашивает у пользователя подтверждение при запуске программ от имени администратора, вместо того, чтобы получить его один раз, сохранить и больше не спрашивать, по крайней мере, у этого же пользователя. Дело в том, что данную информацию придется хранить (скорее всего, в реестре или каком-нибудь файле). Если вредоносная программа проникнет в это хранилище и модифицирует его, то сможет выполнять любые операции, требующие повышенных привилегий, без разрешения пользователя.

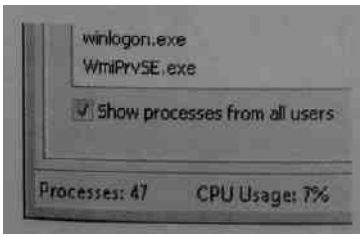


Наверное, наряду с командой Run As Administrator контекстного меню, вы заметили, что некоторые кнопки и ссылки, открывающие доступ к административным задачам в Windows Vista, помечены значком в виде щита. Эта подсказка говорит о том, что щелчок этой кнопки или ссылки откроет окно запроса подтверждения повышения привилегий. Узнать, как снабдить таким значком кнопки в своей программе, вы сможете, изучив программу-пример Process Information (о ней — в конце главы).

А теперь рассмотрим простой пример. При запуске Task Manager через контекстное меню панели задач значок в виде щита отображается на кнопке Show Processes From All Users.

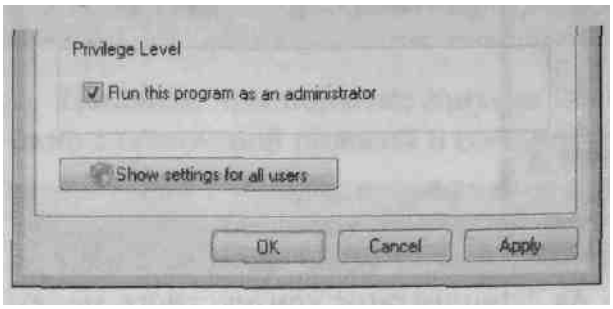


Взгляните на идентификаторы процессов в окне Task Manager, прежде чем щелкнуть эту кнопку. Получив подтверждение повышения привилегий, окно Task Manager исчезает на пару секунд, а потом снова появляется, но уже с флажком вместо щита.



Также можно заметить, что идентификаторы процессов в окне Task Manager изменились после подтверждения повышения привилегий. Можно догадываться, что Task Manager породил новый экземпляр самого себя, уже

с повышенными привилегиями. Да, так и есть. Причина этому только одна: Windows допускает повышение привилегий только при запуске нового процесса, после запуска просить систему об этом уже поздно. Однако непривилегированный процесс может породить другой процесс с более высокими привилегиями и COM-сервером, а затем поддерживать его, чтобы взаимодействовать с более привилегированным процессом через IPC. При этом процессу не обязательно создавать новый экземпляр самого себя, а затем завершаться.



> **Примечание.** По ссылке http://www.microsoft.com/emea/itsshowtime/sessionh.aspx?videoid=360 доступны видеоролик и презентация PowerPoint, в которых Марк Руссинович (Mark Russinovich) рассказывает о внутреннем устройстве UAC, попутно освещая виртуализацию системных ресурсов, обеспечивающую лучшую совместимость Windows с приложениями, не поддерживающими новые ограничения при использовании администраторских полномочий.

Автоматическое повышение привилегий процесса

Если вашему приложению постоянно требуются администраторские привилегии, например, во время установки, ОС может автоматически запрашивать у пользователя повышение привилегий при каждом запуске такого приложения. Но как механизмы UAC решают, что делать при создании нового процесса?

Если в исполняемый файл приложения внедрен ресурс особого типа (RT_MANIFEST), система ищет секцию <trustInfo> (см. пример ниже) и анализирует ее содержимое.

```
…
<trustInfo xmlns="urn:schemas-microsoft-com:asm.v2">
    <security>
        <requestedPrivileges>
            <requestedExecutionLevel
                level="requireAdministrator"
            />
```

```
      </requestedPrivileges>
    </security>
</trustInfo>
```

Атрибут *level* принимает одно из трех значений (см. таблицу 4-8).

Табл. 4-8. Значения атрибута level

Значение	Описание
requireAdministrator	Приложение запустится только с администраторскими привилегиями
highestAvailable	Приложение запускается с максимальными доступными привилегиями. Если пользователь вошел под учетной записью администратора, открывается окно запроса подтверждения повышения привилегий. Если пользователь вошел под обычной учетной записью, приложение запускается без запроса подтверждения со стандартным уровнем привилегий
asInvoker	Приложение запускается с теми же привилегиями, что имеются у вызывающего приложения

Вместо внедрения в ресурсы исполняемого файла манифест можно сохранить в одной с ним папке. При этом у манифеста должны быть то же имя, что и у исполняемого файла, но с дополнительным расширением *.manifest.*

Параметры манифеста, хранимого в отдельном файле, могут подействовать не сразу, особенно если исполняемый файл был запущен до создания файла манифеста. Манифест вступит в силу только после повторного входа пользователя в Windows. Если в исполняемом файле имеется внедренный манифест, то хранимый в файле манифест игнорируется.

Помимо явных требований, определенных в XML-манифесте, ОС также следует особым правилам обеспечения совместимости, в частности, для распознавания программ-установщиков, требующих автоматического запроса повышения привилегий.

В случае приложений, не имеющих встроенного манифеста и не похожих на установщики, пользователь сам выбирает, запустить ли приложение с администраторскими привилегиями, помечая соответствующий флажок на вкладке Compatibility в окне свойств исполняемого файла (см. рисунок ниже).

> **Примечание.** Механизмы обеспечения совместимости выходят за рамки этой книги, подробнее о разработке UAC-совместимых приложений для Windows Vista можно узнать по ссылке http://www.microsoft.com/downloads/details.aspx?FamilyID=ba73b169-a648-49af-bc5e-a2eebb74c16b&DisplayLang=en.

Повышение привилегий процесса вручную

Прочитав подробное описание функции *CreateProcess* (см. выше), вы не могли не заметить отсутствия специальных флагов и параметров для повышения привилегий. Вместо них используется функция *ShellExecuteEx:*

```
BOOL ShellExecuteEx(LPSHELLEXECUTEINFO pExednfo);

typedef struct _SHELLEXECUTEINFO {
    DWORD cbSize;
    ULONG fMask;
    HWND hwnd;
    PCTSTR lpVerb;
    PCTSTR lpFile;
    PCTSTR lpParameters;
    PCTSTR lpDirectory;
    int nShow;
    HINSTANCE hInstApp;
    PVOID lpIDList;
    PCTSTR lpClass;
    HKEY hkeyClass;
    DWORD dwHotKey;
    union {
        HANDLE hIcon;
        HANDLE hMonitor;
    } DUMMYUNIONNAME;
    HANDLE hProcess;
} SHELLEXECUTEINFO, *LPSHELLEXECUTEINFO;
```

У структуры SHELLEXECUTEINFO интересны лишь поля *lpVerb*, которое необходимо установить в *«runas»*, и *lpFile*, куда необходимо занести путь к исполняемому файлу, который необходимо запустить с повышенными привилегиями (см. следующий пример кода).

```
// инициализируем структуру
SHELLEXECUTEINFO sei = { sizeof(SHELLEXECUTEINFO) };

// запрашиваем повышение привилегий
sei.lpVerb = TEXT("runes");

// Создаем окно командной строки,  в котором можно будет
// запускать и другие приложения с повышенными привилегиями.
sei.lpFile | TEXTCcmd.exe");

// не забудьте этот параметр, иначе окно останется скрытым
sei.nShow | SW_SH0WN0RMAL;

if (!ShellExecuteEx(&sei)) {
   DWORD dwStatus I GetLastError();
```

```
    if (dwStatus Щ ERROR_CANCELLED) {
        // пользователь отказал в повышении привилегий }
    else
        if (dwStatus == ERR0R_FILE_N0T_F0UND) {
            // Не найден файл,  заданный lpFile,
            // выводим сообщение об ошибке. }
        }
}
```

Если пользователь отказал в повышении привилегий, *ShellExecuteEx* возвращает FALSE, такую ситуацию идентифицирует значение ERROR_CANCELLED в GetLastError.

Обратите внимание, что каждый процесс, порожденный процессом с повышенными привилегиями, также получает повышенные привилегии процесса-родителя, вызывать *ShellExecuteEx* в этом случае не требуется. Если же попытаться создать вызовом *CreateProcess* новый, требующий повышенных привилегий экземпляр приложения, работающего с отфильтрованным маркером, попытка закончится неудачей, а *GetLastError* вернет ERROR_ELEVATION_REQUIRED.

Подводя итог, можно сказать что «добропорядочные» приложения для Windows Vista должны «уметь» большую часть времени работать под учетной записью обычного пользователя, а на элементах интерфейса (кнопках, ссылках и т.п.), использование которых требует повышенных привилегий, должен отображаться соответствующий значок (см. программу-пример ниже). Поскольку административные задачи должны исполняться в отдельном процессе либо COM-сервером в другом процессе, следует сосредоточить все операции, требующие администраторских привилегий, во втором приложении и повышать его привилегии вызовом *ShellExecuteEx* с передачей «runas» в *IpVerb* и привилегированной операции — в параметре командной строки нового процесса (в поле структуры SHELLEXECUTEINFO).

> **Совет** Отладка процессов, использующих повышение привилегий, может доставить немало неприятностей. Облегчить вам жизнь сможет «золотое правило»: запускайте Visual Studio с тем же уровнем привилегий, которой должен унаследовать отлаживаемый процесс.
>
> Для отладки процесса, работающего с отфильтрованным маркером защиты и привилегиями обычного пользователя, следует запустить Visual Studio с теми же привилегиями (именно так Visual Studio запускается ярлыком по умолчанию или командой меню Start). В противном случае отлаживаемый процесс унаследует повышенные привилегии Visual Studio, запущенной от имени администратора, что нежелательно.
>
> Если отлаживаемый процесс требует администраторских привилегий (например, если это записано в манифесте), экземпляр Visual Studio для

его отладки следует запускать от имени администратора. Иначе вы получите сообщение об ошибке «the requested operation requires elevation» (для запрошенной операции необходимо повышение привилегий), а отлаживаемый процесс вовсе не запустится.

О текущем контексте привилегий

Вернемся к примеру с Task Manager, в окне которого отображался значок с щитом либо флажок, в зависимости от текущего уровня привилегий. В этом примере есть еще два вопроса, заслуживающих отдельного рассмотрения. Первый вопрос - как определить, работает ли приложение под учетной записью с администраторскими привилегиями? Второй, более важный вопрос -как узнать, повышены ли привилегии приложения до уровня администраторских либо оно продолжает работать с отфильтрованным маркером защиты?

Показанная ниже вспомогательная функция *GetProcessElevation* возвращает тип повышения привилегий и булево значение, которое показывает, работает ли приложение с привилегиями администратора.

```
BOOL GetProcessElevation(TOKEN_ELEVATION_TYPE* pElevationType, BOOL*
    pIsAdmin) {
   HANDLE hToken = NULL;
   DWORD dwSize;

   // получаем текущий маркер защиты процесса
   if (!OpenProcessToken(GetCurrentProcess(), TOKEN_QUERY, &hToken))
      return(FALSE);

   BOOL bResult = FALSE;

   // определяем тип повышения привилегий
   if (GetTokenInformation(hToken,  TokenElevationType,
      pElevationType,  sizeof(TOKEN_ELEVATION_TYPE),  &dwSize)) {
      // Создаем SID, соответствующий группе Administrators group
      BYTE adminSID[SECURITY_MAX_SID_SIZE];
      dwSize = sizeof(adminSID);
      CreateWellKnownSid(WinBuiltinAdministratorsSid,  NULL,  &adminSID,
         &dwSize);
      if (*pElevationType == TokenElevationTypeLimited) {
         // Получаем описатель связанного маркера
         // (если приложение работает под локальной учетной
         // записью, маркер уже есть).
         HANDLE hUnfilteredToken = NULL;
         GetTokenlnformation(hToken, TokenLinkedToken,  (VOID*)
            &hUnfilteredToken, sizeof(HANDLE), AdwSize);
```

```
            // проверяем SID администратора в исходном маркере
            if (CheckTokenMembership(hUnfilteredToken, iadminSID, pIsAdmin)) {
                bResult = TRUE;
            }

            // не забывайте закрывать неотфильтрованный маркер!
            CloseHandle(hUnfilteredToken);
        } else {
            *pIsAdmin = IsUserAnAdmin();
            bResult = TRUE;
        }

        // не забывайте закрывать описатель маркера процесса!
        CloseHandle(hToken);

        return(bResult);
    }
```

Заметьте, что *GetTokenInformation* вызывается с передачей маркера, связанного с процессом, и параметра *TokenElevationType*, представляющего тип повышения привилегий. Этот параметр принимает значения перечислимого TOKEN_ELEVATION_TYPE (см. таблицу 4-9).

Табл. 4-9. Значения TOKEN_ELEVATION_TYPE

Значение	Описание
TokenElevationTypeDefault	Процесс работает с привилегиями пользователя по умолчанию либо отключен UAC
TokenElevationTypeFull	Привилегии успешно повышены, процессу назначен неотфильтрованный маркер защиты
TokenElevationTypeLimited	Процесс работает с ограниченными привилегиями, соответствующими неотфильтрованному маркеру защиты

Эти значения позволят выяснить, назначен ли процессу отфильтрованный маркер защиты или нет. Теперь можно определить, обладает ли текущий пользователь полномочиями администратора. Если текущий маркер защиты не отфильтрован, лучше всего вызвать для этого функцию *IsUserAnAdmin*. Если же используется отфильтрованный маркер, следует прежде получить исходный (неотфильтрованный) маркер, передав функции *GetTokenInformation* параметр *TokenLinkedToken*, а затем проверить наличие SID администратора в полученном маркере (с помощью *CreateWellKnownSid* и *CheckTokenMembership*).

Так, приложение-пример Process Information (см. следующий раздел) использует показанную выше вспомогательную функцию в коде, обрабатывающем сообщение WM_INITDIALOG, чтобы вывести в заголовке окна сведения о повышении привилегий, а также показать либо убрать значок в виде щита.

> **Совет.** Для того чтобы отобразить на кнопке или скрыть значок в виде щита, используется макрос *Button_SetElevationRequiredState* (определенный в CommCtrl.h). Напрямую получить этот значок можно, вызвав *SHGetStockIconInfo* с параметром SIID_SHIELD (и функция, и ее параметр объявлены в shellapi.h). Об остальных элементах интерфейса, поддерживающих вывод данного значка, см. в документации MSDN по ссылке http://msdn2.microsoft.com/en-us/library/aa905330.aspx.

Перечисление процессов, выполняемых в системе

Многие разработчики программного обеспечения пытаются создавать инструментальные средства или утилиты для Windows, требующие перечисления процессов, выполняемых в системе. Изначально в Windows API не было функций, которые позволяли бы это делать. Однако в Windows NT ведется постоянно обновляемая база данных Performance Data. В ней содержится чуть ли не тонна информации, доступной через функции реестра вроде *RegQueryValueEx,* для которой надо указать корневой раздел HKEY_PERFORMANCE_DATA. Мало кто из программистов знает об этой базе данных, и причины тому кроются, на мой взгляд, в следующем.

- Для нее не предусмотрено никаких специфических функций; нужно использовать обычные функции реестра.
- Ее нет в Windows 95 и Windows 98.
- Структура информации в этой базе данных очень сложна; многие просто избегают ею пользоваться (и другим не советуют).

Чтобы упростить работу с этой базой данных, Майкрософт создала набор функций под общим названием Performance Data Helper (содержащийся в PDH.dll). Если вас интересует более подробная информация о библиотеке PDH.dll, ищите раздел по функциям Performance Data Helper в документации Platform SDK.

Как я уже упоминал, в Windows 95 и Windows 98 такой базы данных нет. Вместо них предусмотрен набор функций, позволяющих перечислять процессы. Они включены в ToolHelp API. За информацией о них я вновь отсылаю вас к документации Platform SDK — ищите разделы по функциям *Process32First* и *Process32Next.*

Но самое смешное, что разработчики Windows NT, которым ToolHelp-функции явно не нравятся, не включили их в Windows NT. Для перечисления процессов они создали свой набор функций под общим названием Process Status (содержащийся в PSAPI.dll). Так что ищите в документации Platform SDK раздел по функции *EnumProcesses.*

Майкрософт, которая до сих пор, похоже, старалась усложнить жизнь разработчикам инструментальных средств и утилит, все же включила ToolHelp-функции в Windows 2000. Наконец-то и эти разработчики смогут унифицировать свой код хотя бы для Windows 95, Windows 98 и т. д. — до Windows Vista!

Программа-пример ProcessInfo

Эта программа, «04 ProcessInfo.exe», демонстрирует, как создать очень полезную утилиту на основе ToolHelp-функций. Файлы исходного кода и ресурсов программы находятся в каталоге 04-ProcessInfo внутри архива, доступного на сайте поддержки этой книги. После запуска ProcessInfo открывается окно, показанное на рис:4-4.

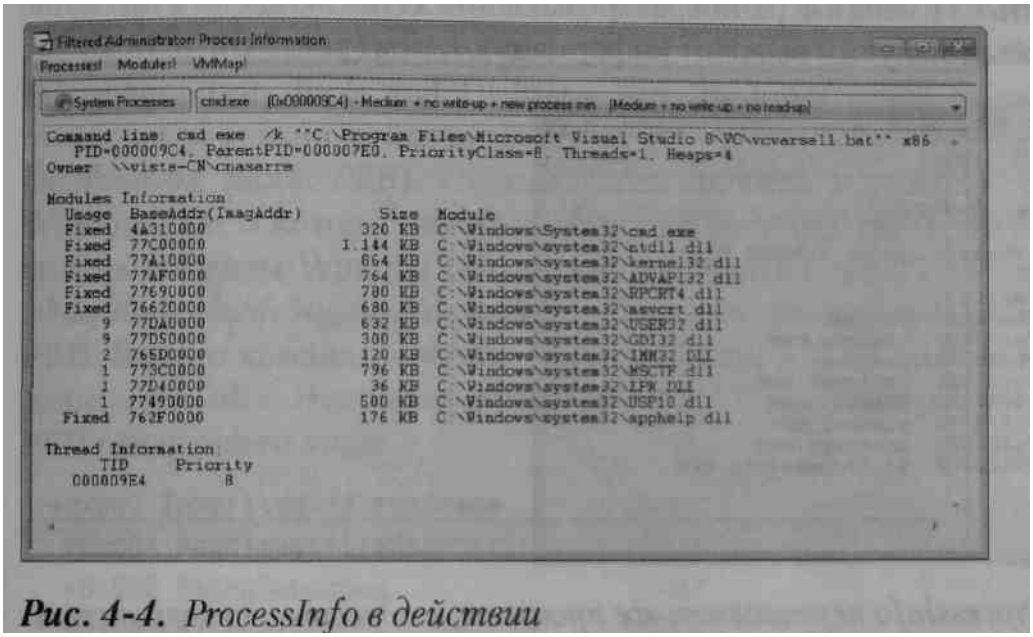


Рис. 4-4. *ProcessInfo в действии*

ProcessInfo сначала перечисляет все процессы, выполняемые в системе, а затем выводит в верхний раскрывающийся список имена и идентификаторы каждого процесса. Далее выбирается первый процесс и информация о нем показывается в большом текстовом поле, доступном только для чтения. Как видите, для текущего процесса сообщается его идентификатор (вместе с идентификатором родительского процесса), класс приоритета и количество потоков, выполняемых в настоящий момент в контексте процесса. Объяснение большей части этой информации выходит за рамки данной главы, но будет рассматриваться в последующих главах.

При просмотре списка процессов становится доступен элемент меню VMMap. (Он отключается, когда вы переключаетесь на просмотр информации о модулях.) Выбрав элемент меню VMMap, вы запускаете программу-пример VMMap (см. главу 14). Эта программа «проходит» по адресному пространству выбранного процесса.

В информацию о модулях входит список всех модулей (EXE- и DLL-файлов), спроецированных на адресное пространство текущего процесса.

Фиксированным модулем (fixed module) считается тот, который был неявно загружен при инициализации процесса. Для явно загруженных DLL показываются счетчики числа пользователей этих DLL. Во втором столбце выводится базовый адрес памяти, на который спроецирован модуль. Если модуль размещен не по заданному для него базовому адресу, в скобках появляется и этот адрес. В третьем столбце сообщается размер модуля в байтах, а в последнем - полное (вместе с путем) имя файла этого модуля. И, наконец, внизу показывается информация о потоках, выполняемых в данный момент в контексте текущего процесса. При этом отображается идентификатор потока (thread ID, TID) и его приоритет.

В дополнение к информации о процессах вы можете выбрать элемент меню Modules. Это заставит ProcessInfo перечислить все модули, загруженные в системе, и поместить их имена в верхний раскрывающийся список. Далее ProcessInfo выбирает первый модуль и выводит информацию о нем (рис. 4-5).

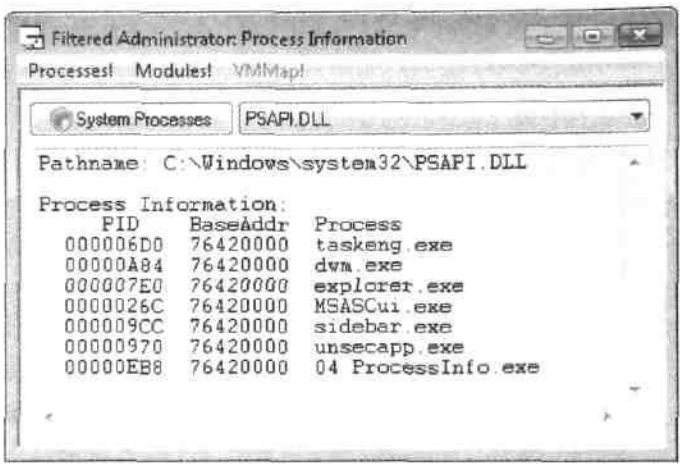


Рис. 4-5. *ProcessInfo перечисляет все процессы, в адресные пространства которых загружен модуль Psapi.dll*

В этом режиме утилита ProcessInfo позволяет легко определить, в каких процессах задействован данный модуль. Как видите, полное имя модуля появляется в верхней части текстового поля, а в разделе Process Information перечисляются все процессы, содержащие этот модуль. Там же показываются идентификаторы и имена процессов, в которые загружен модуль, и его базовые адреса в этих процессах.

Всю эту информацию утилита ProcessInfo получает в основном от различных ToolHelp-функций. Чтобы чуточку упростить работу с ToolHelp-функциями, я создал C++-класс CToolhelp (содержащийся в файле Toolhelp.h). Он инкапсулирует все, что связано с получением «моментального снимка» состояния системы, и немного облегчает вызов других ToolHelp-функций.

Особый интерес представляет функция *GetModulePreferredBaseAddr* в файле Processlnfo.cpp:

```
PVOID GetModulePreferredBaseAddr(
    DWORD dwProcessId,
    PVOID pvModuleRemote);
```

Принимая идентификатор процесса и адрес модуля в этом процессе, она просматривает его адресное пространство, находит модуль и считывает информацию из заголовка модуля, чтобы определить, какой базовый адрес для него предпочтителен. Модуль должен всегда загружаться именно по этому адресу, а иначе приложения, использующие данный модуль, потребуют больше памяти и будут инициализироваться медленнее. Поскольку такая ситуация крайне нежелательна, моя утилита сообщает о случаях, когда модуль загружен не по предпочтительному базовому адресу. Впрочем, на эти темы мы поговорим в разделе «Модификация базовых адресов модулей».

Получить командную строку процесса невозможно. Как сказано в статье из MSDN Magazine «Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities, Part 2», командную строку удаленного процесса можно извлечь из блока его переменных окружения *(Process Environment Block, PEB)*. Однако здесь появились некоторые отличия от Windows XP, о которых необходимо рассказать отдельно. Во-первых, изменились команды WinDbg (его можно загрузить с http://www.microsoft.com/whdc/devtools/debugging/default.mspx) для получения сведения о структуре PEB. Вместо команды «strct», реализованной в расширении kdex2x86, используется «dt». Например, команда «dt nt! PEB» сгенерирует определение PEB следующего вида:

```
+0x000 InheritedAddressSpace          : UChar
 +0x001 ReadImageFileExecOptions      : UChar
 +0x002 BeingDebugged                 : UChar
 +0x003 BitField                      : UChar
 +0x003 ImageUsesLargePages           : Pos 0, 1 Bit
 +0x003 IsProtectedProcess            : Pos 1, 1 Bit
 +0x003 IsLegacyProcess               : Pos 2, 1 Bit
 +0x003 IsImageDynamicallyRelocated   : Pos 3, 1 Bit
 +0x003 SpareBits                     : Pos 4, 4 Bits
 +0x004 Mutant                        : Ptr32 Void
 +0x008 ImageBaseAddress              : Ptr32 Void
 +0x00c Ldr                           : Ptr32 _PEB_LDR_DATA
 +0x010 ProcessParameters             : Ptr32 _RTL_USER_PROCESS_PARAMETERS
 +0x014 SubSystemData                 : Ptr32 Void
 +0x018 ProcessHeap                   : Ptr32 Void
 …
```

Определение структуры RTL_USER_PROCESS_PARAMETERS можно получить, отдав WinDbg команду «dt nt!_RTL_USER_PROCESS_PARAMETERS»:

```
+0x000 NaximumLength                : Uint4B
 +0x004 Length                      : Uint4B
 +0x008 Flags                       : Uint4B
 +0x00c DebugFlags                  : Uint4B
 +0x010 ConsoleHandle               : Ptr32 Void
 +0x014 ConsoleFlags                : Uint4B
 +0x018 StandardInput               : Ptr32 Void
 +0x01c StandardOutput              : Ptr32 Void
 +0x020 StandardError               : Ptr32 Void
 +0x024 CurrentDirectory            : _CURDIR
 +0x030 DllPath                     : _UNICODE_STRING
 +0x038 ImagePathName               : _UNICODE_STRING
 +0x040 CotnmandLine                : _UNICODE_STRING
 +0x048 Environment                 : Ptr32 Void
```

А так определяются внутренние структуры для доступа к командной строке:

```
typedef struct
{
    DWORD Filler[4];
    DWORD InfoBlockAddress;
} __PEB;

typedef struct
{
    DWORD Filler[17];
    DWORD wszCmdLineAddress;
} __INF0BL0CK;
```

Во-вторых, в Windows Vista системные DLL загружаются в адресное пространство процесса по случайным адресам (см. главу 14). Поэтому вместо жесткого программирования адреса PEB как 0x7ffdf000 (именно так делалось в Windows XP) следует вызвать функцию *NtQueryInformationProcess* с параметром *ProcessBasicInformation*. Помните, что в полученной таким образом информации встречаются незадокументированные элементы, которые могут изменяться от версии к версии Windows.

И последний важный момент. Можно заметить, что в программе Process Information список процессов не отражает дополнительную информацию, такую как сведения о загруженных DLL. Например, процесс audiodg.exe (Windows Audio Device Graph Isolation) является *защищенным* (protected process). В Windows Vista появился новый тип процессов, обеспечивающих более полную изоляцию приложений, поддерживающих DRM. В этой связи не удивительно, что у удаленных процессов нет прав на доступ к виртуальной памяти защищенных процессов. Поскольку этот механизм необходим для получения списка загруженных DLL, ToolHelp API не удается получить

данную информацию. Подробнее о защищенных процессах см. по ссылке http://www.microsoft.com/whdc/system/vista/process_Vista.mspx.

Есть и другая причина, по которой Process Information не удается получить сведения о работающих процессах. Если запустить эту программу с обычным уровнем привилегий, ей может быть отказано в доступе для чтения (и уж точно будет отказано в доступе для записи) к процессам с повышенным уровнем привилегий. Это упрощенная картина, в действительности же все может оказаться сложнее. Так, Windows Vista поддерживает новый механизм защиты под названием *Windows Integrity Mechanism* (ранее известный как Mandatory Integrity Control).

В дополнение к традиционным дескрипторам защиты и спискам управления доступом теперь защищенным ресурсам назначаются т.н. *уровни целостности* (integrity levels). Для них в *системных списках управления доступом* (system access control list, SACL) предусмотрен новый тип записи в ACL (ACL entry, ACE) — *mandatory label* Система считает, что защищенным объектам с такой ACE неявно назначен средний (medium) уровень целостности. У каждого процесса также есть уровень доверия (level of trust), основанный на связанном маркере защиты (см. таблицу 4-10).

Табл. 4-10. Уровни доверия

Значение	Описание
Low	На этом уровне работает Internet Explorer в режиме высокой безопасности (Protected Mode) для предотвращения модификации загруженным кодом других приложений и среды Windows
Medium	По умолчанию приложения запускаются на этом уровне и работают с отфильтрованными маркерами защиты
High	На этом уровне работают приложения с повышенными привилегиями
System	На этом уровне работают только процессы, запущенные под учетными записями Local System и Local Service

Когда программа пытается обратиться к объекту ядра, система сравнивает уровень доверия вызывающего процесса и уровень целостности, назначенный объекту ядра. Если последний выше первого, процессу будут запрещены операции модификации и удаления. Заметьте, что это сравнение выполняется до проверки ACL. Таким образом, даже при наличии у процесса прав на доступ к ресурсу, ему может быть отказано в доступе, если уровень доверия процесса ниже, чем уровень целостности ресурса. Это особенно важно в случае приложений, исполняющих код, включая сценарии, загруженные из Интернета. В частности, Internet Explorer 7 в Windows Vista использует этот механизм и работает на уровне Low, запрещая тем самым загруженному из веба коду модифицировать состояние других приложений, по умолчанию работающих на уровне Medium.

Совет. Утилита Process Explorer от компании Sysinternals (http://www.microsoft.com/technet/sysintemals/ProcessesAndThreads/ProcessExplorer.mspx) отображает уровень целостности процессов в одноименном столбце, который следует добавить на вкладке Process Image в Select Columns. В ее исходном коде имеется функция *GetProcessIntegntyLevel*, иллюстрирующая программное получение этих и многих других сведений. Консольная утилита AccessChk того же разработчика (http://www.micmsoft.com/techmt/sysintemah/utilities/accesschk.mspx) выводит уровень целостности, необходимый для доступа к таким ресурсам, как файлы, папки и разделы реестра при вызове с ключами *-i* и *-e*. Консольная утилита icacls.exe для Vista поддерживает ключ */setintegntylevel* для назначения уровня защиты целостности ресурсам файловой системы. После определения уровней целостности маркера защиты процесса и объекта ядра, к которому он пытается обратиться, система выясняет разрешенные операции, проверяя *политики кода* (code *policy),* которые хранятся и в маркере защиты, и в ресурсе. Сначала вызывается функция *GetTokenInformation* с передачей параметра *TokenMandatoryPolicy* и маркера защиты процесса. Эта функция возвращает значение DWORD с битовой маской, описывающей действующую политику (см. таблицу 4-11).

Табл. 4-11. Политики кода

Значение	Описание
POLICY_NO_WRITE_UP	Запрещает коду модификацию ресурсов с более высоким уровнем целостности
POUCY_NEW_PROCESS_MIN	Процессы, порожденные кодом с данной политикой, наследуют минимальный уровень (из уровней, заданных для родительского процесса и в манифесте; если манифест отсутствует, считается что им задан уровень Medium)

Также в коде определены две константы для распознавания отсутствия политики (TOKEN_MANDATORY_POLICY_OFF как 0) и наличия битовой маски (TOKEN_MANDATORY_POLICY_VALID_MASK), служащей для проверки политики (см. ProcessInfo.cpp).

Далее политика ресурса устанавливается по битовой маске Label ACE объекта ядра. (Подробнее о том, как это делается, см. в коде функции *GetProcessIntegntyLevel* в файле ProcessInfo.cpp.) Так получаются две политики доступа к ресурсу (resource policies), которые позволяют определить операции, разрешенные и запрещенные для данного ресурса. По умолчанию действует политика SYSTEM_MANDATORY_LABEL_NO_WRITE_UP, разрешающая процессам с более низким уровнем целостности чтение, но запрещающая запись и удаление ресурса с более высоким уровнем целостности. Политика SYSTEM_MANDATORY_LABEL_NO_READ_UP накладывает еще более строгие ограничения, запрещая процессам с более низким уровнем целостности даже чтение ресурсов с более высоким уровнем целостности.

> **Примечание.** Процессам с низким уровнем целостности разрешено читать содержимое объектов ядра с более высоким уровнем целостности, даже если установлена политика «No-Read-Up», но только при наличии у процесса разрешения Debug. Поэтому утилита Process Information способна читать командную строку процессов с уровнем целостности System (при работе под администраторской учетной записью, под которой можно выдать себе разрешение Debug).

Помимо защиты доступа к объектам ядра, принадлежащим разным процессам, уровни целостности также используются оконной системой, которая запрещает процессам с уровнем целостности Low модифицировать пользовательские интерфейсы процессов с более высоким уровнем целостности. Этот механизм называется *User Interface Privilege Isolation*, UIPI (изоляция пользовательского интерфейса по привилегиям). Операционная система блокирует отправленные (асинхронно и синхронно, соответственно функциями *PostMessage* и *SendMessage),* а также перехваченные (ловушками Windows) оконные сообщения от процессов с более низким уровнем целостности, чтобы предотвратить несанкционированное чтение и запись данных в окна, принадлежащие процессам с более высоким уровнем целостности. Этот механизм особенно наглядно иллюстрирует утилита WindowDump (http://download.microsoft.com/download/8/3/f/83f69587-47f1-48e2-86a6-aab14f01f1fe/EscapeFromDLLHell.exe). Для получения содержимого списка WindowDump вызывает *сначала SendMessage* с параметром LB_GETCOUNT, чтобы узнать число элементов списка, а затем ту же функцию с параметром LB_GETTEXT для извлечения текста из этих элементов. Если запустить эту утилиту с уровнем целостности, более низким, чем у процесса, которому принадлежит окно со списком, первый вызов SendMessage закончится успешно, но вернет 0 вместо числа элементов списка. Аналогично, то есть неудачно, заканчивается попытка получения с помощью утилиты Spy++, запущенной с уровнем целостности Medium, сообщений от окна программы, работающей с более высоким уровнем целостности.

Оглавление

ГЛАВА 5

Задания

Группу процессов зачастую нужно рассматривать как единую сущность. Например, когда вы командуете Visual Studio собрать проект, она порождает процесс Cl.exe, а тот в свою очередь может создать другие процессы (скажем, для дополнительных проходов компилятора). Но, если вы пожелаете прервать сборку, Visual Studio должен каким-то образом завершить Cl.exe и все его дочерние процессы. Решение этой простой (и распространенной) проблемы в Windows было весьма затруднительно, поскольку она не отслеживает родственные связи между процессами. В частности, выполнение дочерних процессов продолжается даже после завершения родительского.

При разработке сервера тоже бывает полезно группировать процессы. Допустим, клиентская программа просит сервер выполнить приложение (которое создает ряд дочерних процессов) и сообщить результаты. Поскольку к серверу может обратиться сразу несколько клиентов, было бы неплохо, если бы он умел как-то ограничивать ресурсы, выделяемые каждому клиенту, и тем самым не давал бы одному клиенту монопольно использовать все серверные ресурсы. Под ограничения могли бы подпадать такие ресурсы, как процессорное время, выделяемое на обработку клиентского запроса, и размеры рабочего набора (working set). Кроме того, у клиентской программы не должно быть возможности завершить работу сервера и т. д.

Windows поддерживает объект ядра под названием «задание» (job). Он позволяет группировать процессы и помещать их в нечто вроде песочницы, которая определенным образом ограничивает их действия. Относитесь к этому объекту как к контейнеру процессов. Кстати, очень полезно создавать задание и с одним процессом — это позволяет налагать на процесс ограничения, которые иначе указать нельзя.

Взгляните на мою функцию *StartRestrictedProcess*. Она включает процесс в задание, которое ограничивает возможность выполнения определенных операций.

```
void StartRestrictedProcess() {
   // Проверить, не связан ли этот процесс с заданием.
   // Если да, переключиться на другое задание невозможно.
   BOOL bInJob = FALSE;
   IsProcessInJob(GetCurrentProcess(), NULL, &bInJob);
   if (bInJob) {
      MessageBox(NULL, TEXT("Process already in a job"),
         TEXT(""), MB_ICONINFORMATION | MB_OK);
      return;
   }
   // создаем объект ядра "задание"
   HANDLE hjob = CreateJobObject(NULL,
      TEXT("Wintellect_RestrictedProcessJob"));

   // вводим ограничения для процессов в задании

   // сначала определяем некоторые базовые ограничения
   JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };

   // процесс всегда выполняется с классом приоритета idle
   jobli.PriorityClass = IDLE_PRIORITY_CLASS;

   // задание не может использовать более одной секунды процессорного времени
   jobli.PerJobUserTimeLimit.QuadPart = 10000;  // 1 секунда, выраженная в
                                                // 100-наносекундных интервалах
   // два ограничения, которые я налагаю на задание (процесс)
   jobli.LimitFlags = J0B_0BJECT_LIMIT_PRI0RITY_CLASS
      J0B_0BJECT_LIMIT_J0B_TIME;
   SetInformationJobObject(hjob, JobObjectBasicLimitInformation, &jobli,
      sizeof(jobli));

   // теперь вводим некоторые ограничения по пользовательскому интерфейсу
   J0B0BJECT_BASIC_UI_RESTRICTI0NS jobuir;
   jobuir.UIRestrictionsClass = J0B_0BJECT_UILIMIT_N0NE;
   // "замысловатый" нуль

   // процесс не имеет права останавливать систему
   jobuir.UIRestrictionsClass |= J0B_0BJECT_UILIMIT_EXITWIND0WS;

   // Процесс не имеет права обращаться к USER-объектам в системе
   // (например, к другим окнам).
   jobuir.UIRestrictionsClass |= J0B_0BJECT_UILIMIT_HANDLES;

   SetInformationJobObject(hjob, JobObjectBasicUIRestrictions, &jobuir,
      sizeof(jobuir));
```

```
// Порождаем процесс, который будет размещен в задании.
// ПРИМЕЧАНИЕ: процесс нужно сначала создать и только потом поместить
// в задание. А это значит, что поток процесса должен быть создан
// и тут же приостановлен, чтобы он не смог выполнить какой-нибудь код
// еще до введения ограничений.
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi;
TCHAR szCmdLine[8];
_tcscpy_s(szCmdLine, _countof(szCmdLine), TEXT("CMD"));
BOOL bResult =
    CreateProcess(
        NULL, szCmdLine, NULL, NULL, FALSE,
        CREATE_SUSPENDED | CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
// Включаем процесс в задание.
// ПРИМЕЧАНИЕ: дочерние процессы, порождаемые этим процессом,
// автоматически становятся частью того же задания.
AssignProcessToJobObject(hjob, pi.hProcess);

// теперь потоки дочерних процессов могут выполнять код
ResumeThread(pi.hThread);
CloseHandle(pi.hThread);

// Ждем, когда процесс завершится или будет исчерпан.
// Лимит процессорного времени, указанный для задания.
HANDLE h[2];
h[0] = pi.hProcess;
h[1] = hjob;
DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
switch (dw - WAIT_OBJECT_0) {
    case 0:
        // процесс завершился…
        break;
    case 1:
        // лимит процессорного времени исчерпан…
        break;
}

FILETIME CreationTime;
FILETIME ExitTime;
FILETIME KernelTime;
FILETIME UserTime;
TCHAR szInfo[MAX_PATH];
GetProcessTimes(pi.hProcess, ACreationTime, &ExitTime,
    &KernelTime, &UserTime);
StringCchPrintf(szInfo, _countof(szInfo), TEXT("Kernel = %u | User = %u\n"),
    KernelTime.dwLowDateTime / 10000, UserTime.dwLowDateTime / 10000);
MessageBox(GetActiveWindow(), szInfo, TEXT("Restricted Process times"),
```

```
        MB_ICONINFORMATION  | MB_OK);
    // проводим очистку
    CloseHandle(pi.hProcess);
    CloseHandle(hjob);
}
```

А теперь я объясню, как работает *StartRestrictedProcess*. Сначала я создаю новый объект ядра «задание», вызывая:

```
BOOL IsProcessInJob( HANDLE hProcess, HANDLE hJob, PBOOL pbInJob);
```

Как и любая функция, создающая объекты ядра, *CreateJobObject* принимает в первом параметре информацию о защите и сообщает системе, должна ли она вернуть наследуемый описатель. Параметр *pszName* позволяет присвоить заданию имя, чтобы к нему могли обращаться другие процессы через функцию *OpenJobObject*

Если процесс уже включен в задание, то ни его, ни порожденные им процессы не удастся вывести из задания. Это защитный механизм, который не дает процессам обойти наложенные на них ограничения.

> **Внимание!** По умолчанию процессы, которые создаются при запуске приложений через Windows Explorer, автоматически включаются в особое задание, имя которого предваряет префикс «PCA». Как сказано в разделе «Уведомления заданий», возможно получение уведомлений о завершении процесса в составе задания. Этот механизм позволяет запустить Program Compatibility Assistant при сбое унаследованного приложения, запущенного через Windows Explorer.
>
> Если вашему приложению потребуется создать задание, как это делает программа-пример Job Lab, показанная в конце этой главы, у вас ничего не выйдет, поскольку объект-задание, имя которого начинается с «PCA», уже связан с вашими процессами.
>
> Эта функция поддерживается Windows Vista исключительно в целях обеспечения совместимости. Таким образом, если снабдить приложение манифестом (см. главу 4), Проводник Windows не включит его процесс в задание, имя которого начинается префиксом «PCA», предполагая, что разработчики этого приложения уже решили все проблемы с совместимостью.
>
> Однако при отладке приложений возможна следующая ситуация. Если отладчик запущен из-под Windows Explorer, то ваше приложение, даже если у него есть манифест, как и отладчик, будет включено в задание с именем «PCA...». Простое решение этой проблемы - запускать отладчик из командной оболочки, а не из-под Проводника, тогда процесс отладчика не будет включен в это задание.

Затем я создаю новый объект ядра «задание», вызывая

```
HANDLE CreateJobObject(
    PSECURITY_ATTRIBUTES psa,
    PCTSTR pszName);
```

Как и любая функция, создающая объекты ядра, *CreatejobObject* принимает в первом параметре информацию о защите и сообщает системе, должна ли она вернуть наследуемый описатель. Параметр *pszName* позволяет присвоить заданию имя, чтобы к нему могли обращаться другие процессы через функцию *OpenJobObject.*

```
HANDLE OpenJobObject(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

Закончив работу с объектом-заданием, закройте его описатель, вызвав, как всегда, *CloseHandle.* Именно так я и делаю в конце своей функции *StartRestrictedProcess.* Имейте в виду, что закрытие объекта-задания не приводит к автоматическому завершению всех его процессов. На самом деле этот I объект просто помечается как подлежащий разрушению, и система уничтожает его только после завершения всех включенных в него процессов. Заметьте, что после закрытия описателя объект-задание становится недоступным для процессов, даже несмотря на то что объект все еще существует. Этот факт иллюстрирует следующий код:

```
// создаем именованный объект-задание
HANDLE hJob = CreateJobObject(NULL, TEXT("Jeff"));

// включаем в него наш процесс
AssignProcessToJobObject(hJob, GetCurrentProces());

// закрытие объекта-задания не убивает ни наш процесс, ни само задание,
// но присвоенное ему имя ("Jeff") моментально удаляется
CloseHandle(hJob);

// пробуем открыть существующее задание
hJob = OpenJobObject(JOB_OBJECT_ALL_ACCESS,  FALSE,  TEXT("Jeff));
// OpenJobObject терпит неудачу и возвращает NULL,  поскольку имя ("Jeff")

// Уже не указывает на объект-задание после вызова CloseHandle;
// получить описатель этого объекта больше нельзя
```

Определение ограничений, налагаемых на процессы в задании

Создав задание, вы обычно строите «песочницу» (набор ограничений) для включаемых в него процессов. Ограничения бывают нескольких видов:

- базовые и расширенные базовые ограничения — не дают процессам в задании монопольно захватывать системные ресурсы;
- базовые ограничения по пользовательскому интерфейсу (UI) — блокируют возможность его изменения;
- ограничения, связанные с защитой, — перекрывают процессам в задании доступ к защищенным ресурсам (файлам, подразделам реестра и т. д.).

Ограничения на задание вводятся вызовом:

```
BOOL SetInformationJobObject(
    HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInformationClass,
    PVOID pJobObjectInformation,
    DWORD cbJobObjectlnformationSize);
```

Первый параметр определяет нужное вам задание, второй параметр (перечислимого типа) — вид ограничений, третий — адрес структуры данных, содержащей подробную информацию о задаваемых ограничениях, а четвертый — размер этой структуры (используется для указания версии). Следующая таблица показывает, как устанавливаются ограничения.

Табл. 5-1. Типы ограничений

Вид ограничений	Значение второго параметра	Структура, указываемая в третьем параметре
Базовые ограничения	*JobObjectBaskLimitInformation*	JOBOBJECT_BASIC_ LIMIT_INFORMATION
Расширенные базовые ограничения	*JobObjectExtendedLimttInformation*	JOBOBJECT_ EXTENDED_ LIMIT_INFORMATION
Базовые ограничения по пользовательскому интерфейсу	*JobObjectBasicUIRestrictions*	JOBOBJECT_BASIC_ UI_RESTRICTIONS
Ограничения, связанные с защитой	*JobObjectSecurityLimitInformation*	JOBOBJECT_ SECURITY_LIMIT_ INFORMATION

В функции *StartRestrictedProcess* я устанавливаю для задания лишь несколько базовых ограничений. Для этого я создаю структуру JOB_OBJECT_ BASIC LIMIT_INFORMATION, инициализирую ее и вызываю функцию *Setlnformation-JobObject* Данная структура выглядит так:

```
typedef struct _JOBOBJECT_BASIC_LIMIT_INFORMATION {
    LARGE_INTEGER   PerProcessUserTimeLimit;
    LARGE_INTEGER   PerJobUse rTimeLimit;
    DWORD           LimitFlags;
    DWORD           MinimumWorkingSetSize;
    DWORD           MaximumWorkingSetSize;
    DWORD           ActiveProcessLimit;
    DWORD_PTR       Affinity;
    DWORD           PriorityClass;
    DWORD           SchedulingClass;
} JOBOBJECT_BASIC_LIMIT_INFORMATION, *PJOBOBJECT_BASIC_LIMIT_INFORMATION;
```

Все элементы этой структуры кратко описаны в таблице 5-2. Хочу пояснить некоторые вещи, связанные с этой структурой, которые, по-моему довольно туманно изложены в документации Platform SDK. Указывая ограничения для задания, вы устанавливаете те или иные биты в элементе *LimitFlags*. Например, в *StartRestrictedProcess* я использовал флаги JOB_OBJECT_LIMIT_PRIORITY_CLASS и JOB_ OBJECT_LIMIT_JOB_TIME, т. е. определил всего два ограничения.

При выполнении задание ведет учет по нескольким показателям — например, сколько процессорного времени уже использовали его процессы. Всякий раз, когда вы устанавливаете базовые ограничения с помощью флага JOB_OBJECT_LIMIT_JOB_TIME, из общего процессорного времени, израсходованного всеми процессами, вычитается то, которое использовали завершившиеся процессы. Этот показатель сообщает, сколько процессорного времени израсходовали активные на данный момент процессы. А что если вам понадобится изменить ограничения на доступ к подмножеству процессоров, не сбрасывая при этом учетную информацию по процессорному времени? Для этого вы должны ввести новое базовое ограничение флагом JOB_OBJECT_LIMIT_AFFINITY и отказаться от флага JOB_OBJECT_LIMIT_JOBTIME. Но тогда получится, что вы снимаете ограничения на процессорное время.

Вы хотели другого: ограничить доступ к подмножеству процессоров, сохранив существующее ограничение на процессорное время, и не вычитать время, израсходованное завершенными процессами, из общего времени. Чтобы решить эту проблему, используйте специальный флаг JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME. Этот флаг и JOB_OBJECT_LIMIT_JOB_TIME являются взаимоисключающими. Флаг JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME указывает системе«изменить ограничения, не вычитая процессорное время, использованное уже завершенными процессами.

Обсудим также элемент *SchedulingClass* структуры JOBOBJECT_BASIC_LIMIT_INFORMATION. Представьте, что для двух заданий определен класс приоритета NORMAL_PRIORITY_CLASS, а вы хотите, чтобы процессы одного задания получали больше процессорного времени, чем

процессы другого. Так вот, элемент *SchedulingClass* позволяет изменять распределение процессорного времени между заданиями с одинаковым классом приоритета, вы можете присвоить ему любое значение в пределах 0-9 (по умолчанию он равен 5). Увеличивая его значение, вы заставляете Windows 2000 выделять потокам в процессах конкретного задания более длительный квант времени, а снижая — напротив, уменьшаете этот квант.

Табл. 5-2. Элементы структуры JOBOBJECT_BASIC_LIMIT_INFORMATION

Элементы	Описание	Примечание
PerProcessUser-TimeLimit	Максимальное время в пользовательском режиме, выделяемое каждому процессу (в порциях)	Система автоматически завершает любой процесс, который пытается использовать больше отведенного по 100 нс времени. Это ограничение вводится флагом JOB_OBJECT_LIMIT_PROCESS_TIME в *LimitFlags*
PerJobUserTime-Limit	Максимальное время в пользовательском режиме для всех процессов в данном задании (в порциях по 100 нс)	По умолчанию система автоматически завершает все процессы, когда заканчивается это время. Данное значение можно изменять в процессе выполнения задания. Это ограничение вводится флагом JOB_OBJECT_LIMIT_JOB_TIME в *LimitFlags*
LimitFlags	Виды ограничений для задания	См. раздел после таблицы
Minimum Working SetSize и *Maximum Working SetSize*	Верхний и нижний предел рабочего набора для каждого процесса (а не для всех процессов в задании)	Обычно рабочий набор процесса может расширяться за стандартный предел; указав *MaximumWorking-SetSize,* вы введете жесткое ограничение. Когда размер рабочего набора какого-либо процесса достигнет заданного предела, процесс начнет сбрасывать свои страницы на диск. Вызовы функции *SetProcessWorkingSetSize* этим процессом будут игнорироваться, если только он не обращается к ней для того, чтобы очистить свой рабочий набор. Это ограничение вводится флагом JOB_OBJECT_LIMIT_WORKINGSET в *LimitFlags*

Табл. 5-2. *(окончание)*

Элементы	Описание	Примечание
ActiveProcessLimit	Максимальное количество процессов, одновременно выполняемых	Любая попытка обойти такое ограничение приведет к завершению в задании нового процесса с ошибкой «not enough quota» («превышение квоты»). Это ограничение вводится флагом JOB_OBJECT_LIMIT_ACTIVE_PROCESS в *LimitFlags*
Affinity	Подмножество процессоров, на которых можно выполнять процессы этого задания	Для индивидуальных процессов это ограничение можно еще больше детализировать. Вводится флагом JOB_OBJECT_LIMIT_AFFINITY в *LimitFlags*
PriorityClass	Класс приоритета для всех процессов в задании	Вызванная процессом функция *SetPriorityClass* сообщает об успехе даже в том случае, если на самом деле она не выполнила свою задачу, а *GetPriorityClass* возвращает класс приоритета, каковой и пытался установить процесс, хотя в реальности его класс может быть совсем другим. Кроме того, *SetThreadPriority* не может поднять приоритет потоков выше normal, но позволяет понижать его. Это ограничение вводится флагом JOB_OBJECT_LIMIT_PRIORITY_CLASS в *LimitFlags*
SchedulingClass	Относительная продолжительность кванта времени, выделяемого всем потокам в задании	Этот элемент может принимать значения от 0 до 9; по умолчанию устанавливается 5. Подробнее о его назначении см. ниже. Это ограничение вводится флагом JOB_OBJECT_LIMIT_SCHEDULING_CLASS в *LimitFlags*

Допустим, у меня есть два задания с обычным (normal) классом приоритета: в каждом задании — по одному процессу, а в каждом процессе — по одному потоку (тоже с обычным приоритетом). В нормальной ситуации эти два потока обрабатывались бы процессором по принципу карусели и получали бы равные кванты процессорного времени. Но если я запишу в элемент *SchedulingClass* для первого задания значение 3, система будет выделять его потокам более короткий квант процессорного времени, чем потокам второго задания.

Используя *SchedulingClass*, избегайте слишком больших его значений, иначе вы замедлите общую реакцию других заданий, процессов и потоков на какие-либо события в системе.

И последнее ограничение, которое заслуживает отдельного упоминания, связано с флагом JOB_OBJECT_LIMIT_DIE_ONJUNHANDLED_ EXCEPTION. Он отключает для всех процессов в задании вывод диалогового окна с сообщением о необработанном исключении. Система реагирует на этот флаг вызовом *SetErrorMode* с флагом SEM_NOGPFAULTERRORBOX для каждого из процессов в задании. Процесс, в котором возникнет необрабатываемое им исключение, немедленно завершается без уведомления пользователя. Этот флаг полезен в сервисных и других пакетных заданиях. В его отсутствие один из процессов в задании мог бы вызвать исключение и не завершиться, впустую расходуя системные ресурсы.

Помимо базовых ограничений, вы можете устанавливать расширенные, для чего применяется структура JOBOBJECT_EXTENDED_LIMIT_ INFORMATION:

```
typedef struct _JOBOBJECT_EXTENDED_LIMIT_INFORMATION {
    JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitlnformation;
    IO_COUNTERS IoInfo;
    SIZE_T ProcessMemoryLimit;
    SIZE_T JobMemoryLimit;
    SIZE_T PeakProcessMemoryUsed;
    SIZE_T PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION, *PJOBOBJECT_EXTENDED_LIMIT_INFORMATION;
```

Как видите, она включает структуру JOBOBJECT_BASIC_LIMIT_ INFORMATION, являясь фактически ее надстройкой. Это несколько странная структура, потому что в ней есть элементы, не имеющие никакого отношения к определению ограничений для задания. Во-первых, элемент *IoInfo* зарезервирован, и вы ни в коем случае не должны обращаться к нему. О том, как узнать значение счетчика ввода-вывода, я расскажу позже. Кроме того, элементы *PeakProcessMemoryUsed* и *PeakJobMemoryUsed* предназначены только для чтения и сообщают о максимальном объеме памяти, переданной соответственно одному из процессов или всем процессам в задании.

Остальные два элемента, *ProcessMemoryLimit* и *JobMemoryLimit*, ограничивают соответственно объем переданной памяти, который может быть использован одним из процессов или всеми процессами в задании. Чтобы задать любое из этих ограничений, укажите в элементе *LimitFlags* флаг JOB_OBJECT_LIMIT_JOB_ MEMORY или JOB_OBJECT_LIMIT_PROCESS_MEMORY.

А теперь вернемся к прочим ограничениям, которые можно налагать на задания. Структура JOBOBJECT_BASIC_UI_RESTRICTIONS выглядит так:

```
typedef struct _JOBOBJECT_BASIC_UI_RESTRICTIONS {
    DWORD UIRestrictionsClass;
} JOBOBJECT_BASIC_UI_RESTRICTIONS, *PJOBOBJECT_BASIC_UI_RESTRICTIONS;
```

В этой структуре всего один элемент, *UIRestrictionsClass*, который содержит набор битовых флагов, кратко описанных в таблице 5-3.

Табл. 5-3. Битовые флаги базовых ограничений по пользовательскому интерфейсу для объекта-задания

Флаг	Описание
JOB_OBJECT_UILIMIT_ EXITWINDOWS	Запрещает выдачу команд из процессов на выход из системы, завершение ее работы, перезагрузку или выключение компьютера через функцию *ExitWindowsEx*
JOB_OBJECT_UILIMIT_ READCLIPBOARD	Запрещает процессам чтение из буфера обмена
JOB_OBJECT_UILIMIT_ WRITECLIPBOARD	Запрещает процессам стирание буфера обмена
JOB_OBJECT_UILIMIT_ SYSTEMPARAMETERS	Запрещает процессам изменение системных параметров через *SystemParametersInfo*
JOB_OBJECT_UILIMIT_ DISPLAYSETTINGS	Запрещает процессам изменение параметров экрана через *ChangeDisplaySettings*
JOB_OBJECT_UILIMIT_ GLOBALATOMS	Предоставляет заданию отдельную глобальную таблицу атомарного доступа (global atom table) и разрешает его процессам пользоваться только этой таблицей
JOB_OBJECT_UILIMIT_ DESKTOP	Запрещает процессам создание новых рабочих столов или переключение между ними через функции *CreateDesktop* или *SwitchDesktop*
JOB_OBJECT_UILIMIT_ HANDLES	Запрещает процессам в задании использовать USER-объекты (например, HWND), созданные внешними по отношению к этому заданию процессами

Последний флаг, JOB_OBJECT_UILIMIT_HANDLES, представляет особый интерес: он запрещает процессам в задании обращаться к USER-объектам, созданным внешними по отношению к этому заданию процессами. Так, запустив утилиту Microsoft Spy++ из задания, вы не обнаружите никаких окон, кроме тех, которые создаст сама Spy++. На рис. 5-1 показано окно Microsoft Spy++ с двумя открытыми дочерними MDI-окнами.

Заметьте, что в левой секции (Threads 1) содержится список потоков в системе. Кажется, что лишь у одного из них, 000006AC SPYXX, есть дочерние окна. А все дело в том, что я запустил Microsoft Spy++ из задания и ограничил ему права на использование описателей USER-объектов. В том же окне сообщается о потоках MSDEV и EXPLORER, но никаких упоминаний о созданных ими окнах нет. Уверяю вас, эти потоки наверняка создали какие-нибудь окна - просто Spy++ лишена возможности их видеть. В правой секции

(Windows 1) утилита Spy++ должна показывать иерархию окон на рабочем столе, но там нет ничего, кроме одного элемента — 00000000. (Это не настоящий элемент, но Spy++ была обязана поместить сюда хоть что-нибудь)

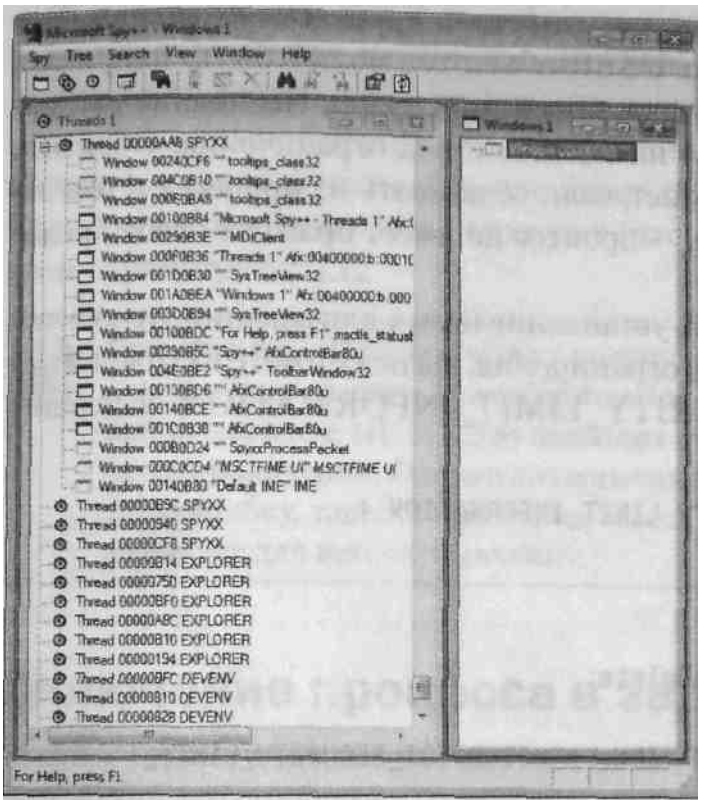


Рис. 5-1. *Microsoft Spy++ работает в задании, которому ограничен доступ к описателям USER-объектов*

Обратите внимание, что такие ограничения односторонни, т. е. внешние процессы все равно видят USER-объекты, которые созданы процессами, включенными в задание. Например, если запустить Notepad в задании, а Spy++ — вне его, последняя увидит окно Notepad, даже если для задания указан флаг JOB_OBJECT_UILIMIT_HANDLES. Кроме того, Spy++, запущенная в отдельном задании, все равно увидит это окно Notepad, если только для ее задания не установлен флаг JOB_OBJECT_UILIMIT_HANDLES.

Ограничение доступа к описателям USER-объектов — вещь изумительная, если вы хотите создать по-настоящему безопасную песочницу, в которой будут «копаться» процессы вашего задания. Однако часто бывает нужно, чтобы процесс в задании взаимодействовал с внешними процессами.

Одно из самых простых решений здесь — использовать оконные сообщения, но, если процессам в задании доступ к описателям пользовательского интерфейса запрещен, ни один из них не сможет послать сообщение (синхронно или асинхронно) окну, созданному внешним процессом. К счастью, теперь есть функция, которая поможет решить эту проблему:

```
BOOL UserHandleGrantAccess( HANDLE hUserObj,
```

```
    HANDLE hJob,
    BOOL bGrant);
```

Параметр *hUserObj* идентифицирует конкретный USER-объект, доступ к I которому вы хотите предоставить или запретить процессам в задании. Это почти всегда описатель окна, но USER-объектом может быть, например, рабочий стол, программная ловушка, ярлык или меню. Последние два параметра, *hjob* и *fGrant*, указывают на задание и вид ограничения. Обратите внимание, что функция не сработает, если ее вызвать из процесса в том задании, на которое указывает *hjob*, - процесс не имеет права сам себе предоставлять доступ к объекту.

И последний вид ограничений, устанавливаемых для задания, относится к защите. (Введя в действие такие ограничения, вы не сможете их отменить.) Структура JOBOBJECT_SECURITY_LIMIT_INFORMATION выглядит так:

```
typedef struct _JOBOBJECT_SECURITY_LIMIT_INFORMATION {
    DWORD SecurityLimitFlags;
    HANDLE JobToken;
    PTOKEN_GROUPS SidsToDisable;
    PTOKEN_PRIVILEGES PrivilegesToDelete;
    PTOKEN_GROUPS RestrictedSids;
} JOBOBJECT_SECURITY_LIMIT_INFORMATION, *PJOBOBJECT_SECURITY_LIMIT_INFORMATION;
```

Ее элементы описаны в следующей таблице.

Табл. 5-4. Элементы структуры JOBOBJECT_SECURITY_LIMIT_INFORMATION

Элемент	Описание
SecurityLimitFlags	Набор флагов, которые закрывают доступ администратору, запрещают маркер неограниченного доступа, принудительно назначают заданный маркер доступа, блокируют доступ по каким-либо идентификаторам защиты (security ID, SID) и отменяют указанные привилегии
JobToken	Маркер доступа, связываемый со всеми процессами в задании
SidsToDisable	Указывает, по каким SID не разрешается доступ
PrivilegesToDelete	Определяет привилегии, которые снимаются с маркера доступа
RestrictedSids	Задает набор SID, по которым запрещается доступ к любому защищенному объекту (deny-only SIDs); этот набор добавляется к маркеру доступа

Естественно, если вы налагаете ограничения, то потом вам, наверное, понадобится информация о них. Для этого вызовите:

```
BOOL QueryInformationJobObject(
    HANDLE hJob,
    JOBOBJECTINFOCLASS JobObjectInformationClass,
```

```
PVOID pvJobObjectInformation,
DWORD cbJobObjectIlnformationSize,
PDWORD pdwReturnSize);
```

В эту функцию, как и в *SetInformationJobObject,* передается описатель задания, переменная перечислимого типа JOBOBJECTINFOCLASS. Она сообщает информацию об ограничениях, адрес и размер структуры данных, инициализируемой функцией. Последний параметр, *pdwReturnLength,* заполняется самой функцией и указывает, сколько байтов помещено в буфер. Если эти сведения вас не интересуют (что обычно и бывает), передавайте в этом параметре NULL.

> **Примечание.** Процесс может получить информацию о своем задании, передав при вызове QueryInformationJobObject вместо описателя задания значение NULL. Это позволит ему выяснить установленные для него ограничения. Однако аналогичный вызов SetInformationJobObject даст ошибку, так как процесс не имеет права самостоятельно изменять заданные для него ограничения.

Включение процесса в задание

О'кей, с ограничениями на этом закончим. Вернемся к *StartRestrictedProcess.* Установив ограничения для задания, я вызываю *CreateProcess* и создаю процесс, который помещаю в это задание. Я использую здесь флаг CREATE_SUSPENDED, и он приводит к тому, что процесс порождается, но код пока не выполняет. Поскольку *StartRestrictedProcess* вызывается из процесса, внешнего по отношению к заданию, его дочерний процесс тоже не входит в это задание. Если бы я разрешил дочернему процессу немедленно начать выполнение кода, он проигнорировал бы мою песочницу со всеми ее ограничениями. Поэтому сразу после создания дочернего процесса и перед началом его работы я должен явно включить этот процесс в только что сформированное задание, вызвав:

```
BOOL AssignProcessToJobObject( HANDLE hJob, HANDLE hProcess);
```

Эта функция заставляет систему рассматривать процесс, идентифицируемый параметром *hProcess,* как часть существующего задания, на которое указывает *hJob.* Обратите внимание, что *AssignProcessToJobObject* позволяет включить в задание только тот процесс, который еще не относится ни к одному заданию. Как только процесс стал частью какого-нибудь задания, его нельзя переместить в другое задание или отпустить на волю. Кроме того, когда процесс, включенный в задание, порождает новый процесс, последний автоматически помещается в то же задание. Однако этот порядок можно изменить.

- Включая в *LimitFlags* структуры JOBOBJECT_BASIC_LIMIT_INFORMATION флаг JOB_OBJECT_BREAKAWAY_OK, вы сообщаете системе, что новый процесс может выполняться вне задания. Потом вы должны вызвать *CreateProcess* с новым флагом CREATE_BREAKAWAY_FROM_ JOB. (Если вы сделаете это без флага JOB_OBJECT_BREAKAWAY_OK в *LimitFlags*, функция *CreateProcess* завершится с ошибкой.) Такой механизм пригодится на случай, если новый процесс тоже управляет заданиями.
- Включая в *LimMags* структуры JOBOBJECT_BASIC_LIMIT_INFORMATION флаг JOB_OBJECT_SILENT_BREAKAWAY_OK, вы тоже сообщаете системе, что новый процесс не является частью задания. Но указывать в *CreateProcess* какие-либо флаги на этот раз не потребуется. Данный механизм полезен для процессов, которым ничего не известно об объектах-заданиях.

Что касается *StartRestrictedProcess*, то после вызова *AssignProcessToJobObject* новый процесс становится частью задания. Далее я вызываю *ResumeThread*, чтобы поток нового процесса начал выполняться в рамках ограничений, установленных для задания. В этот момент я также закрываю описатель потока, поскольку он мне больше не нужен.

Завершение всех процессов в задании

Уверен, именно это вы и будете делать чаще всего. В начале главы я упомянул о том, как непросто остановить сборку в Developer Studio, потому что для этого ему должны быть известны все процессы, которые успел создать его самый первый процесс. (Это очень каверзная задача. Как Developer Studio справляется с ней, я объяснял в своей колонке «Вопросы и ответы по Win32» в июньском выпуске Microsoft Systems Journal за 1998 год, см. http://www.microsoft.com/msj/0698/win320698.aspx.) Подозреваю, что следующие версии Developer Studio будут использовать механизм заданий, и решать задачу, о которой мы с вами говорили, станет гораздо легче.

Чтобы уничтожить все процессы в задании, вы просто вызываете:

```
BOOL TerminateJobObject(
    HANDLE hJob,
    UINT uExitCode);
```

Вызов этой функции похож на вызов *TerminateProcess* для каждого процесса в задании и присвоение всем кодам завершения одного значения — *uExitCode*.

Получение статистической информации о задании

Мы уже обсудили, как с помощью *QueryInformationjobObject* получить информацию о текущих ограничениях, установленных для задания. Этой функцией можно пользоваться и для получения статистической информации.

Например, чтобы выяснить базовые учетные сведения, вызовите ее, передав *JobObjectBasicAccountingInformation* во втором параметре и адрес структуры JOBOBJECT_BASIC_ACCOUNTING_INFORMATION:

```
typedef struct _JOBOBJECT_BASIC_ACCOUNTING_INFORMATION {
    LARGE_INTEGER TotalUserTime;
    LARGE_INTEGER TotalKernelTime;
    LARGE_INTEGER ThisPeriodTotalUserTime;
    LARGE_INTEGER ThisPeriodTotalKernelTime;
    DWORD TotalPageFaultCount;
    DWORD TotalProcesses;
    DWORD ActiveProcesses;
    DWORD TotalTerminatedProcesses;
} JOBOBJECT_BASIC_ACCOUNTING_INFORMATION,
    *PJOBOBJECT_BASIC_ACCOUNTING_INFORMATION;
```

Элементы этой структуры кратко описаны в таблице 5-5.

Табл. 5-5. Элементы структуры JOBOBJECT_BASIC_ACCOUNTING_ INFORMATION

Элемент	Описание
TotalUserTime	Процессорное время, израсходованное процессами задания в пользовательском режиме
TotalKernelTime	Процессорное время, израсходованное процессами задания в режиме ядра
ThisPeriodTotalUserTime	То же, что *TotalUserTime,* но обнуляется, когда базовые ограничения изменяются вызовом *SetInformationJobObject*, *JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME* не используется
ThisPeriodTotalKernelTime	То же, что *ThisPeriodTotalUserTime,* но относится к процессорному времени, израсходованному в режиме ядра
TotalPageFaultCount	Общее количество ошибок страниц, вызванных процессами задания
TotalProcesses	Общее число процессов, когда-либо выполнявшихся в этом задании
ActiveProcesses	Текущее количество процессов в задании
TotalTerminatedProcesses	Количество процессов, завершенных из-за превышения ими отведенного лимита процессорного времени

Как видите, код, расположенный в конце функции StartRestrictedProcess, позволяет получить сведения об утилизации процессорного времени для любого процесса, даже не включенного в состав задания. Это делается с помощью функции GetProcessTimes, о которой будет рассказано в главе 7.

Вы можете извлечь те же сведения вместе с учетной информацией по вводу-выводу, передав *JobObjectBasicAndloAccountingInformation* во втором пара-

метре и адрес структуры JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION:

```
typedef struct JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
    JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
    IO_COUNTERS IoInfo;
} JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION,
    *PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION;
```

Как видите, она просто возвращает JOBOBJECT_BASIC_ACCOUNTING_INFORMATION и IO_COUNTERS. Последняя структура показана на следующей странице.

```
typedef struct _I0_C0UNTERS {
    ULONGLONG ReadOperationCount;
    ULONGLONG WriteOperationCount;
    ULONGLONG OtherOperationCount;
    ULONGLONG ReadTransferCount;
    ULONGLONG WriteTransferCount;
    ULONGLONG OtherTransferCount;
} IO_COUNTERS, *PI0_C0UNTERS;
```

Она сообщает о числе операций чтения, записи и перемещения (а также о количестве байтов, переданных при выполнении этих операций). Данные относятся ко всем процессам в задании. Кстати, новая функция *GetProcessIoCounters* позволяет получить ту же информацию о процессах, не входящих ни в какие задания.

```
BOOL GetProcessIoCounters(
    HANDLE hProcess,
    PIO_COUNTERS ploCounters);
```

QueryInformationjobObject также возвращает набор идентификаторов текущих процессов в задании. Но перед этим вы должны прикинуть, сколько их там может быть, и выделить соответствующий блок памяти, где поместятся массив идентификаторов и структура JOBOBJECT_BASIC_PROCESS_ID_LIST:

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
    DWORD NumberOfAssignedProcesses;
    DWORD NumberOfProcessIdsInList;
    DWORD ProcessIdList[1];
} JOBOBJECT_BASIC_PROCESS_ID_LIST, *PJOBOBJECT_BASIC_PROCESS_ID_LIST;
```

В итоге, чтобы получить набор идентификаторов текущих процессов в задании, нужно написать примерно такой код:

```
void EnumProcessIdsInJob(HANDLE hjob) {
```

```
    // Я исхожу из того, что количество процессов
    // в этом задании никогда не превысит 10.
    #define MAX_PROCESS_IDS  10

    // определяем размер блока памяти (в байтах)
    DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
        (MAX_PROCESS_IDS - 1) * sizeof(DWORD);

    // для хранения идентификаторов и структуры
    PJ0B0BJECT_BASIC_PR0CESS_ID_LIST pjobpil =
        (PJ0B0BJECT_BASIC_PR0CESS_ID_LIST)_alloca(cb);

    // Сообщаем функции, на какое максимальное число процессов
    // рассчитана выделенная нами память.
    pjobpil->NumberOfAssignedProcesses = MAX_PROCESS_IDS;

    // запрашиваем текущий список идентификаторов процессов
    QueryInformationJobObject(hjob, JobObjectBasicProcessIdList,
        pjobpil, cb, &cb);

    // перечисляем идентификаторы процессов
    for (DWORD x = 0; x < pjobpil->NumberOfProcessIdsInList;  x++) {
        // используем pjobpil->ProcessIdList[x]…
    }

    // Так как для выделения памяти мы вызывали _alloca,
    // освобождать память нам не потребуется.
}
```

Вот и все, что вам удастся получить через эти функции, хотя на самом деле операционная система знает о заданиях гораздо больше. Эту информацию, которая хранится в специальных счетчиках, можно извлечь с помощью функций из библиотеки Performance Data Helper (PDH.dll) или через модуль Performance Monitor, подключаемый к Microsoft Management Console (MMC). Для просмотра сведений о заданиях также используют инструмент Reliability and Performance Monitor (из категории Administrative Tools), но он отображает только объекты заданий с глобальными именами. Для работы с заданиями весьма удобна утилита Process Explorer от Sysinternals (http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ProcessExplorer.mspx). По умолчанию процессы, на которые наложены ограничения посредством заданий, выделяются в окне Process Explorer (см. рис. 5-2).

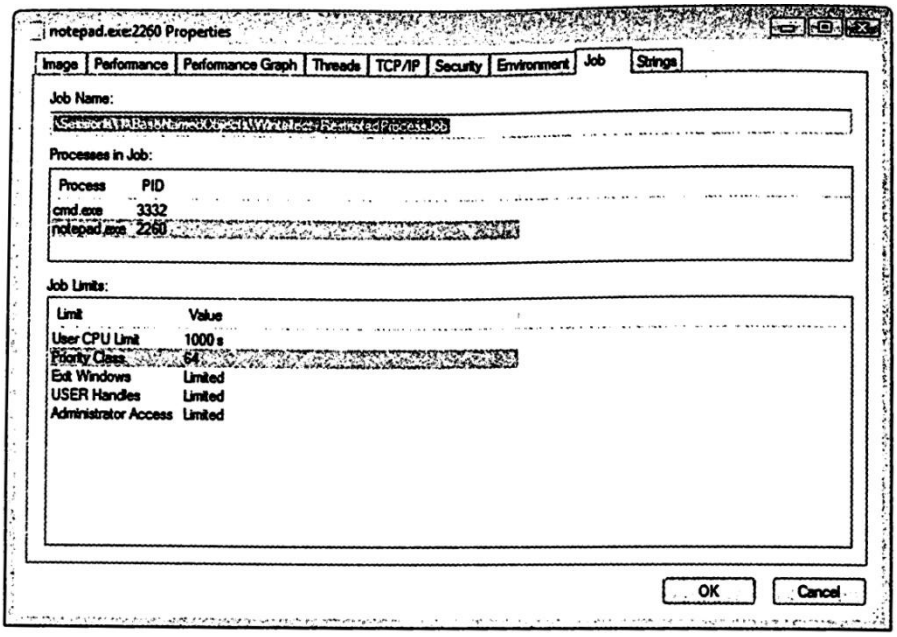


Рис. 5-2. *Сведения об ограничения процессов на вкладке Job в окне Process Explorer*

> **Внимание!** Значение параметра «User CPU Limit» по ошибке отображается в секундах, тогда как он измеряется в миллисекундах. Эта ошибка будет исправлена в следующей версии программы.

Уведомления заданий

Итак, базовые сведения об объектах-заданиях я изложил. Единственное, что осталось рассмотреть, — уведомления. Допустим, вам нужно знать, когда завершаются все процессы в задании или заканчивается все отпущенное им процессорное время. Либо выяснить, когда в задании порождается или уничтожается очередной процесс. Если такие уведомления вас не интересуют (а во многих приложениях они и не нужны), работать с заданиями будет очень легко — не сложнее, чем я уже рассказывал. Но если они все же понадобятся, вам придется копнуть чуть поглубже.

Информацию о том, все ли выделенное процессорное время исчерпано, получить нетрудно. Объекты-задания не переходят в свободное состояние до тех пор, пока их процессы не израсходуют отведенное процессорное время. Как только оно заканчивается, система уничтожает все процессы в задании и переводит его объект в свободное состояние (signaled state). Это событие легко перехватить с помощью *WaitForSingleObject* (или похожей функции). Кстати, потом вы можете вернуть объект-задание в состояние

«занято» (nonsignaled state), вызвав *SetlnformationJobObject* и выделив ему дополнительное процессорное время.

Когда я только начинал разбираться с заданиями, мне казалось, что объект-задание должен переходить в свободное состояние после завершения всех его процессов. В конце концов, прекращая свою работу, объекты процессов и потоков освобождаются; то же самое вроде бы должно происходить и с заданиями. Но Майкрософт предпочла сделать по-другому: объект-задание переходит в свободное состояние после того, как исчерпает выделенное ему время. Поскольку большинство заданий начинает свою работу с одним процессом, который существует, пока не завершатся все его дочерние процессы, вам нужно просто следить за описателем родительского процесса — он освободится, как только завершится все задание. Моя функция *StartRestrictedProcess* как раз и демонстрирует данный прием.

Но это были лишь простейшие уведомления — более «продвинутые», например о создании или разрушении процесса, получать гораздо сложнее. В частности, вам придется создать объект ядра «порт завершения ввода-вывода» и связать с ним объект или объекты «задание». После этого нужно будет перевести один или больше потоков в режим ожидания порта завершения.

Создав порт завершения ввода-вывода, вы сопоставляете с ним задание, вызывая *SetInformationJobObject* следующим образом:

```
J0B0BJECT_ASS0CIATE_C0MPLETI0N_P0RT joacp;
joacp.CompletionKey =1;                    // Любое значение,  уникально
                                           // идентифицирующее это задание,
joacp.CompletionPort = hIOCP;              // Описатель порта завершения,
                                           // принимающего уведомления.
SetInformationJobObject(hJob, JobObjectAssociateCompletionPortlnformation,
    &joacp, sizeof(jaocp));
```

После выполнения этого кода система начнет отслеживать задание и при возникновении событий передавать их порту завершения. (Кстати, вы можете вызывать *QueryInformationJobObject* и получать ключ завершения и описатель порта, но вряд ли это вам когда-нибудь понадобится.) Потоки следят за портом завершения ввода-вывода, вызывая *GetQueuedCompletionStatus:*

```
BOOL GetQueuedCompletionStatus(
    HANDLE hlOCP,
    PDWORD pNumBytesTransferred,
    PULONG_PTR pCompletionKey,
    POVERLAPPED *pOverlapped,
    DWORD dwMilliseconds);
```

Когда эта функция возвращает уведомление о событии задания, *pCompletionKey* содержит значение ключа завершения, заданное при вызове *SetInformationJobObject* для связывания задания с портом завершения. По нему вы узнаете, в каком из заданий возникло событие. Значение в *pNumBytesTransferred* указывает, какое именно событие произошло (табли-

ца 5-6). В зависимости от конкретного события в *pOverlapped* может возвращаться идентификатор процесса.

Табл. 5-6. Уведомления о событиях задания, посылаемые системой связанному с этим заданием порту завершения

Событие	Описание
JOB_OBJECT_MSG_ACTIVE_ PROCESS_ZERO	В задании нет работающих процессов
JOB_OBJECT_MSG_END_OF_ PROCESS_TIME	Процессорное время, выделенное процессу, исчерпано; процесс завершается, и сообщается его идентификатор
JOB_OBJECT_MSG_ACTIVE_ PROCESS_LIMIT	Была попытка превысить ограничение на число активных процессов в задании
JOB_OBJECT_MSG_PROCESS_ MEMORY_LIMIT	Была попытка превысить ограничение на объем памяти, которая может быть передана процессу; сообщается идентификатор процесса
JOB_OBJECT_MSG_ JOB_MEMORY_LIMIT	Была попытка превысить ограничение на объем памяти, которая может быть передана заданию; сообщается идентификатор процесса
JOB_OBJECT_MSG_NEW_ PROCESS	В задание добавлен процесс; сообщается идентификатор процесса
JOB_OBJECT_MSG_EXIT_ PROCESS	Процесс завершен; сообщается идентификатор процесса
JOB_OBJECT_MSG_ ABNORMAL_EXIT_PROCESS	Процесс завершен из-за необработанного им исключения; сообщается идентификатор процесса
JOB_OBJECT_MSG_END_ OF_OB_TIME	Процессорное время, выделенное заданию, исчерпано; процессы не завершаются, и вы можете либо возобновить их работу, задав новый лимит по времени, либо самостоятельно завершить процессы, вызвав *TerminateJobObject*

И последнее замечание: по умолчанию объект-задание настраивается системой на автоматическое завершение всех его процессов по истечении выделенного ему процессорного времени, а уведомление JOB_OBJECT_MSG_END_OF_ JOB_TIME не посылается. Если вы хотите, чтобы объект-задание не уничтожал свои процессы, а просто сообщал о превышении лимита на процессорное время, вам придется написать примерно такой код:

```
// Создаем структуру J0B0BJECT_END_0F_J0B_TIME_INF0RMATI0N
// и инициализируем ее единственный элемент.
J0B0BJECT_END_0F_J0B_TIME_INF0RMATI0N joeojti;
joeojti.EndOfJobTimeAction = J0B_0BJECT_P0ST_AT_END_0F_J0B;

// сообщаем заданию, что ему нужно делать по истечении его времени
SetInformationJobObJect(hJob, JobObjectEndOfJobTimelnformation,
    &joeojti, &sizeof(joeojti));
```

Вы можете указать и другое значение, JOB_OBJECTTERMINATE_AT_END_OF_JOB, но оно задается по умолчанию, еще при создании задания.

Программа-пример JobLab

Эта программа, «05JobLab.exe» позволяет легко экспериментировать с заданиями. Ее файлы исходного кода и ресурсов находятся в каталоге 05-JobLab на компакт-диске, прилагаемом к книге. После запуска JobLab открывается окно, показанное на рис. 5-3.

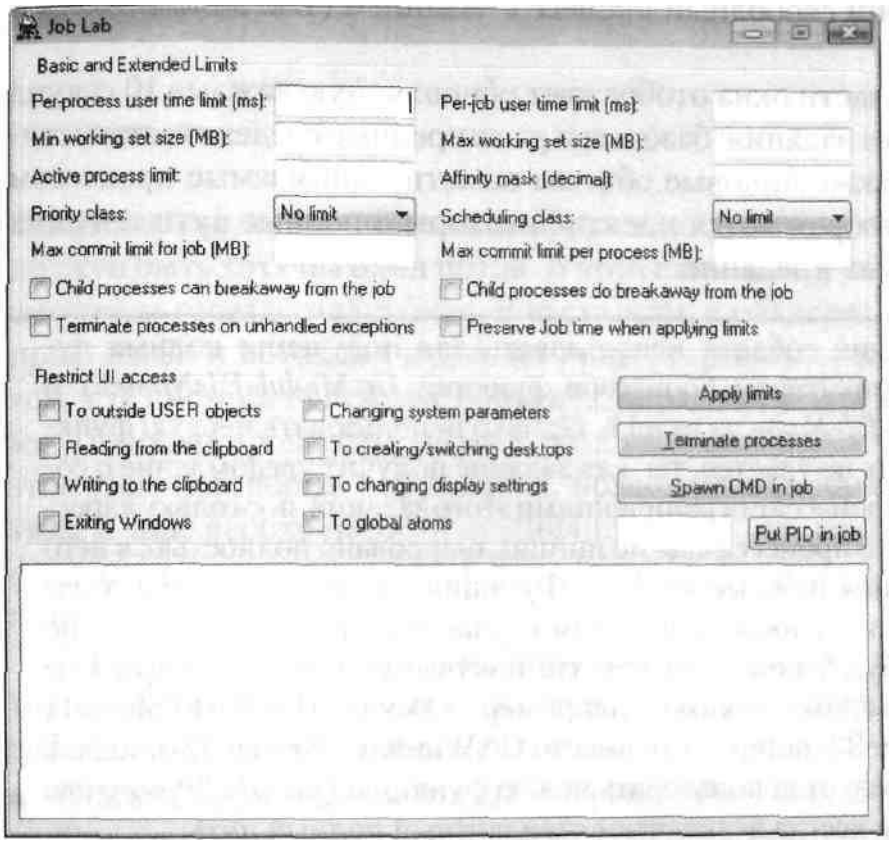


Рис. 5-3. *Программа-пример JobLab*

Когда процесс инициализируется, он создает объект «задание». Я присваиваю ему имя JobLab, чтобы вы могли наблюдать за ним с помощью MMC Performance Monitor. Моя программа также создает порт завершения ввода-вывода и связывает с ним объект-задание. Это позволяет отслеживать уведомления от задания и отображать их в списке в нижней части окна.

Изначально в задании нет процессов, и никаких ограничений для него не установлено. Поля в верхней части окна позволяют задавать базовые и расширенные ограничения. Все, что от вас требуется, — ввести в них допустимые значения и щелкнуть кнопку Apply Limits. Если вы оставляете поле пустым, соответствующие ограничения не вводятся. Кроме базовых и расширенных, вы можете задавать ограничения по пользовательскому интерфейсу. Обратите внимание: помечая флажок Preserve Job Time When Applying Limits, вы не устанавливаете ограничение, а просто полу-

чаете возможность изменять ограничения, не сбрасывая значения элементов *ThisPeriodTotalUserTime* и *ThisPeriodTotalKernelTime* при запросе базовой учетной информации. Этот флажок становится недоступен при наложении ограничений на процессорное время для отдельных задании.

Остальные кнопки позволяют управлять заданием по-другому. Кнопка Terminate Processes уничтожает все процессы в задании. Кнопка Spawn CMD In Job запускает командный процессор, сопоставляемый с заданием. Из этого процесса можно запускать дочерние процессы и наблюдать, как они ведут себя, став частью задания. И последняя кнопка, Put PID In Job, позволяет связать существующий свободный процесс с заданием (т. е. включить его в задание).

Список в нижней части окна отображает обновляемую каждые 10 секунд информацию о статусе задания: базовые и расширенные сведения, статистику ввода-вывода, а также пиковые объемы памяти, занимаемые процессом и заданием. Также отображаются идентификаторы и полные пути текущих процессов, включенных в задание.

> **Внимание!** Велик соблазн использовать для получения полных путей по идентификаторам процессов функции *GetModuleFileNameEx* и *GetProcessImageFileName* из psapi.h. Однако использовать первую функцию в этих целях не удастся, так как задание получит уведомление о создании нового процесса с ограничениями этого задания, поскольку адресное пространство процесса еще не инициализировано полностью: в него не спроецированы нужные модули. Функция *GetProcessImageFileName* интересна тем, что способна и в этом случае получит полный путь, но его синтаксис будет похож на тот, что получается в режиме ядра, а не в пользовательском режиме, например \Device\Harddisk Volume 1\ Windows\System32\notepad.exe вместо C:\Windows\System32\notepad. exe. Поэтому следует использовать новую функцию *QueryFullProcessImageName,* которая всегда возвращает стандартный полный путь.

Кроме этой информации, в списке показываются уведомления, поступающие от задания в порт завершения ввода-вывода. (Кстати, вся информация обновляется и при приеме уведомления.)

И еще одно: если вы измените исходный код и будете создавать безымянный объект ядра «задание», то сможете запускать несколько копий этой программы, создавая тем самым два и более объектов-заданий на одной машине. Это расширит ваши возможности в экспериментах с заданиями.

Что касается исходного кода, то специально обсуждать его нет смысла -в нем и так достаточно комментариев. Замечу лишь, что в файле Job.h я определил C++-класс CJob, инкапсулирующий объект «задание» операционной системы. Это избавило меня от необходимости передавать туда-сюда описатель задания и позволило уменьшить число операций приведения типов, которые обычно приходится выполнять при вызове функций *QueryInformationJobObject* и *SetInformationJobObject.*

Оглавление

ГЛАВА 6

Базовые сведения о потоках

Тематика, связанная потоками, очень важна, потому что в любом процессе должен быть хотя бы один поток. В этой главе концепции потоков будут рассмотрены гораздо подробнее. В частности, я объясню, в чем разница между процессами и потоками и для чего они предназначены. Также я расскажу о том, как система использует объекты ядра «поток» для управления потоками. Подобно процессам, потоки обладают определенными свойствами, поэтому мы поговорим о функциях, позволяющих обращаться к этим свойствам и при необходимости модифицировать их. Кроме того, вы узнаете о функциях, предназначенных для создания (порождения) дополнительных потоков в системе.

В главе 4 я говорил, что процесс фактически состоит из двух компонентов: объекта ядра «процесс» и адресного пространства. Так вот, любой поток тоже состоит из двух компонентов:

- объекта ядра, через который операционная система управляет потоком. Там же хранится статистическая информация о потоке;
- стека потока, который содержит параметры всех функций и локальные переменные, необходимые потоку для выполнения кода. (О том, как система управляет стеком потока, я расскажу в главе 16.)

В той же главе 4 я упомянул, что процессы инертны. Процесс ничего не исполняет, он просто служит контейнером потоков. Потоки всегда создаются в контексте какого-либо процесса, и вся их жизнь проходит только в его границах. На практике это означает, что потоки исполняют код и манипулируют данными в адресном пространстве процесса. Поэтому, если два и более потока выполняются в контексте одного процесса, все они делят одно адресное пространство. Потоки могут исполнять один и тот же код и манипулировать одними и теми же данными, а также совместно использовать описатели объектов ядра, поскольку таблица описателей создается не в отдельных потоках, а в процессах.

Как видите, процессы используют куда больше системных ресурсов, чем потоки. Причина кроется в адресном пространстве. Создание виртуального адресного пространства для процесса требует значительных системных ресурсов. При этом ведется масса всяческой статистики, на что уходит немало памяти. В адресное пространство загружаются EXE- и DLL-файлы, а значит, нужны файловые ресурсы. С другой стороны, потоку требуются лишь соответствующий объект ядра и стек; объем статистических сведений о потоке невелик и много памяти не занимает.

Так как потоки расходуют существенно меньше ресурсов, чем процессы, старайтесь решать свои задачи за счет использования дополнительных потоков и избегайте создания новых процессов. Только не принимайте этот совет за жесткое правило — многие проекты как раз лучше реализовать на основе множества процессов. Нужно просто помнить об издержках и соразмерять цель и средства.

Прежде чем мы углубимся в скучные, но крайне важные концепции, давайте обсудим, как правильно пользоваться потоками, разрабатывая архитектуру приложения.

В каких случаях потоки создаются

Поток (thread) определяет последовательность исполнения кода в процессе. При инициализации процесса система всегда создает первичный поток. Начинаясь со стартового кода из библиотеки C/C++, который в свою очередь вызывает входную функцию (*__tmain*, *_tWinMain*) из вашей программы, он живет до того момента, когда входная функция возвращает управление стартовому коду и тот вызывает функцию *ExitProcess*. Большинство приложений обходится единственным, первичным потоком. Однако процессы могут создавать дополнительные потоки, что позволяет им эффективнее выполнять свою работу.

У каждого компьютера есть чрезвычайно мощный ресурс — центральный процессор. И нет абсолютно никаких причин тому, чтобы этот процессор простаивал (не считая экономии электроэнергии). Чтобы процессор всегда был при деле, вы нагружаете его самыми разнообразными задачами. Вот несколько примеров.

- Вы активизируете службы индексации Windows Indexing Services. Она создает поток с низким приоритетом, который, периодически пробуждаясь, индексирует содержимое файлов на дисковых устройствах вашего компьютера. Чтобы найти какой-либо файл, вы открываете окно Search Results (щелкнув кнопку Start и выбрав из меню Search команду For Files Or Folders) и вводите в поле Containing Text нужные критерии поиска. После этого начинается поиск по индексу, и на экране появляется список файлов, удовлетворяющих этим критериям. Служба индексации данных значительно увеличивает скорость поиска, так как при ее использовании больше не требуется открывать, сканировать и закрывать каждый файл на диске.

- Вы запускаете программу для дефрагментации дисков, поставляемую с Windows. Обычно утилиты такого рода предлагают массу настроек для администрирования, в которых средний пользователь совершенно не разбирается, — например, когда и как часто проводить дефрагментацию. Благодаря потокам с более низким приоритетом вы можете пользоваться этой программой в фоновом режиме и дефрагментировать диски в те моменты, когда других дел у системы нет.
- Visual Studio IDE автоматически компилирует исходный код на C# и Visual Basic .NET, пока вы продолжаете ввод ее текста. При этом в окне редактора ошибочные фрагменты кода выделяются подчеркиванием, а IDE выводит соответствующие предупреждения и сообщения об ошибках, если навести указатель мыши на подчеркнутый фрагмент.
- Электронные таблицы пересчитывают данные в фоновом режиме.
- Текстовые процессоры разбивают текст на страницы, проверяют его на орфографические и грамматические ошибки, а также печатают в фоновом режиме.
- Файлы можно копировать на другие носители тоже в фоновом режиме.
- Веб-браузеры способны взаимодействовать с серверами в фоновом режиме. Благодаря этому пользователь может перейти на другой веб-узел, не дожидаясь, когда будут получены результаты с текущего веб-узла.

Одна важная вещь, на которую вы должны были обратить внимание во всех этих примерах, заключается в том, что поддержка многопоточности позволяет упростить пользовательский интерфейс приложения. Если компилятор ведет сборку вашей программы в те моменты, когда вы делаете паузы в наборе ее текста, отпадает необходимость в командах меню Build. То же самое относится к командам Check Spelling и Check Grammar в текстовых процессорах.

В примере с веб-браузером выделение ввода-вывода (сетевого, файлового или какого-то другого) в отдельный поток обеспечивает «отзывчивость» пользовательского интерфейса приложения даже при интенсивной передаче данных. Вообразите приложение, которое сортирует записи в базе данных, печатает документ или копирует файлы. Возложив любую из этих задач, так или иначе связанных с вводом-выводом, на отдельный поток, пользователь может по-прежнему работать с интерфейсом приложения и при необходимости отменить операцию, выполняемую в фоновом режиме.

Многопоточное приложение легче масштабируется. Как вы увидите в следующей главе, каждый поток можно закрепить за определенным процессором. Так что, если в вашем компьютере имеется два процессора, а в приложении — два потока, оба процессора будут при деле. И фактически вы сможете выполнять две задачи одновременно.

В каждом процессе есть хотя бы один поток. Даже не делая ничего особенного в приложении, вы уже выигрываете только потому, что оно выполняется в многопоточной операционной системе. Например, вы можете собирать

программу и одновременно пользоваться текстовым процессором (довольно часто я так и работаю). Если в компьютере установлено два процессора, то сборка выполняется на одном из них, а документ обрабатывается на другом. Иначе говоря, какого-либо падения производительности не наблюдается. И кроме того, если компилятор из-за той или иной ошибки входит в бесконечный цикл, на остальных процессах это никак не отражается. (Конечно, о программах для MS-DOS и 16-разрядной Windows речь не идет.)

И в каких случаях потоки не создаются

До сих пор я пел одни дифирамбы многопоточным приложениям. Но, несмотря на все преимущества, у них есть и свои недостатки. Некоторые разработчики почему-то считают, будто *любую* проблему можно решить, разбив программу на отдельные потоки. Трудно совершить большую ошибку!

Потоки — вещь невероятно полезная, когда ими пользуются с умом. Увы, решая старые проблемы, можно создать себе новые. Допустим, вы разрабатываете текстовый процессор и хотите выделить функциональный блок, отвечающий за распечатку, в отдельный поток. Идея вроде неплоха: пользователь, отправив документ на распечатку, может сразу вернуться к редактированию. Но задумайтесь вот над чем: значит, информация в документе может быть изменена *при* распечатке документа? Как видите, теперь перед вами совершенно новая проблема, с которой прежде сталкиваться не приходилось. Тут-то и подумаешь, а стоит ли выделять печать в отдельный поток, зачем искать лишних приключений? Но давайте разрешим при распечатке редактирование любых документов, кроме того, который печатается в данный момент. Или так: скопируем документ во временный файл и отправим на печать именно его, а пользователь пусть редактирует оригинал в свое удовольствие. Когда распечатка временного файла закончится, мы его удалим — вот и все.

Еще одно узкое место, где неправильное применение потоков может привести к появлению проблем, — разработка пользовательского интерфейса в приложении. В подавляющем большинстве программ все компоненты пользовательского интерфейса (окна) обрабатываются одним и тем же потоком. И дочерние окна любого окна определенно должен создавать только один поток. Создание разных окон в разных потоках иногда имеет смысл, но такие случаи действительно редки.

Обычно в приложении существует один поток, отвечающий за поддержку пользовательского интерфейса, — он создает все окна и содержит цикл *GetMessage*. Любые другие потоки в процессе являются рабочими (т. е. отвечают за вычисления, ввод-вывод и другие операции) и не создают никаких окон. Поток пользовательского интерфейса, как правило, имеет более высокий приоритет, чем рабочие потоки, — это нужно для того, чтобы он всегда быстро реагировал на действия пользователя.

Несколько потоков пользовательского интерфейса в одном процессе можно обнаружить в таких приложениях, как Windows Explorer. Он создает

отдельный поток для каждого окна папки. Это позволяет копировать файлы из одной папки в другую и попутно просматривать содержимое еще какой-то папки. Кроме того, если какая-то ошибка в Explorer приводит к краху одного из его потоков, прочие потоки остаются работоспособны, и вы можете пользоваться соответствующими окнами, пока не сделаете что-нибудь такое, из-за чего рухнут и они.

В общем, мораль этого вступления такова: многопоточность следует использовать разумно. Не создавайте несколько потоков только потому, что это возможно. Многие полезные и мощные программы по-прежнему строятся на основе одного первичного потока, принадлежащего процессу.

Ваша первая функция потока

Каждый поток начинает выполнение с некоей входной функции. В первичном потоке таковой является *Jmain,* или *_tWinMain*. Если вы хотите создать вторичный поток, в нем тоже должна быть входная функция, которая выглядит примерно так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam){
    DW0RD dwResult = 0;
    …
    return(dwResult);
}
```

Функция потока может выполнять любые задачи. Рано или поздно она закончит свою работу и вернет управление. В этот момент ваш поток остановится, память, отведенная под его стек, будет освобождена, а счетчик пользователей его объекта ядра «поток» уменьшится на 1. Когда счетчик обнулится, этот объект ядра будет разрушен. Но, как и объект ядра «процесс», он может жить гораздо дольше, чем сопоставленный с ним поток.

А теперь поговорим о самых важных вещах, касающихся функций потоков

- В отличие от входной функции первичного потока, у которой должно быть одно из четырех имен: *main, vmain, WinMain* или *wWinMain* (за исключением случаев, когда ключом /ENTRY: компоновщика другая задана как входная), — функцию потока можно назвать как угодно. Однако, если в программе несколько функций потоков, вы должны присвоить им разные имена, иначе компилятор или компоновщик решит, что вы создаете несколько реализаций единственной функции.
- Поскольку входным функциям первичного потока передаются строковые параметры, они существуют в ANSI- и Unicode-версиях: *main — wmain* и *WinMain — wWinMain*. Но функциям потоков передается единственный параметр, смысл которого определяется вами, а не операционной системой. Поэтому здесь нет проблем с ANSI/Unicode.
- Функция потока должна возвращать значение, которое будет использоваться как код завершения потока. Здесь полная аналогия с библиотекой

C/C++: код завершения первичного потока становится кодом завершения процесса.

- Функции потоков (да и все ваши функции) должны по мере возможности обходиться своими параметрами и локальными переменными. Так как к статической или глобальной переменной могут одновременно обратиться несколько потоков, есть риск повредить ее содержимое. Однако параметры и локальные переменные создаются в стеке потока, поэтому они в гораздо меньшей степени подвержены влиянию другого потока.

Вот вы и узнали, как должна быть реализована функция потока. Теперь рассмотрим, как заставить операционную систему создать поток, который выполнит эту функцию.

Функция *CreateThread*

Мы уже говорили, как при вызове функции *CreateProcess* появляется на свет первичный поток процесса. Если вы хотите создать дополнительные потоки, нужно вызвать из первичного потока функцию *CreateThread*:

```
HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    DW0RD cbStackSize,
    PTHREAD_START_ROUTIME pfnStartAddr,
    PVOID pvParam,
    DWORD dwCreateFlags,
    PDWORD pdwThreadID);
```

При каждом вызове этой функции система создает объект ядра «поток». Это не сам поток, а компактная структура данных, которая используется операционной системой для управления потоком и хранит статистическую информацию о потоке. Так что объект ядра «поток» — полный аналог объекта ядра «процесс».

Система выделяет память под стек потока из адресного пространства процесса. Новый поток выполняется в контексте того же процесса, что и родительский поток. Поэтому он получает доступ ко всем описателям объектов ядра, всей памяти и стекам всех потоков в процессе. За счет этого потоки в рамках одного процесса могут легко взаимодействовать друг с другом.

> **Примечание.** *CreateThread* — это Windows-функция, создающая поток. Но никогда не вызывайте ее, если вы пишете код на C/C++. Вместо нее вы должны использовать функцию *_beginthreadex* из библиотеки Visual C++. (Если вы работаете с другим компилятором, он должен поддерживать свой эквивалент функции *CreateThread*.) Что именно делает _beginthreadex и почему это так важно, я объясню потом.

Итак, общее представление о функции *CreateThread* вы получили. Давайте рассмотрим все ее параметры.

Параметр *psa*

Параметр *psa* является указателем на структуру SECURITY_ATTRIBUTES. Если вы хотите, чтобы объекту ядра «поток» были присвоены атрибуты защиты по умолчанию (что чаще всего и бывает), передайте в этом параметре NULL. А чтобы дочерние процессы смогли наследовать описатель этого объекта, определите структуру SECURITY_ATTRIBUTES и инициализируйте ее элемент *bInheritHandle* значением TRUE (см. главу *3*).

Параметр *cbStackSize*

Этот параметр определяет, какую часть адресного пространства поток сможет использовать под свой стек. Каждому потоку выделяется отдельный стек. Функция *CreateProcess*, запуская приложение, вызывает ***CreateThread***, и та инициализирует первичный поток процесса. При этом *CreateProcess* заносит в параметр *cbStackSize* значение, хранящееся в самом исполняемом файле. Управлять этим значением позволяет ключ /STACK компоновщика:

```
/STACK:[reserve] [,commit]
```

Аргумент *reserve* определяет объем адресного пространства, который система должна зарезервировать под стек потока (по умолчанию — 1 Мб). Аргумент *commit* задает объем физической памяти, который изначально передается области, зарезервированной под стек (по умолчанию — 1 страница). По мере исполнения кода в потоке вам, весьма вероятно, понадобится отвести под стек больше одной страницы памяти. При переполнении стека возникнет исключение. (О стеке потока и исключениях, связанных с его переполнением, см. главу 16, а об общих принципах обработки исключений — главу *23.)* Перехватив это исключение, система передаст зарезервированному пространству еще одну страницу (или столько, сколько указано в аргументе *commit*). Такой механизм позволяет динамически увеличивать размер стека лишь по необходимости.

Если вы, обращаясь к *CreateThread*, передаете в параметре *cbStackSize* ненулевое значение, функция резервирует всю указанную вами память. Ее объем определяется либо значением параметра *cbStackSize*, либо значением, заданным в ключе /STACK компоновщика (выбирается большее из них). Но передается стеку лишь тот объем памяти, который соответствует значению в *cbStackSize*. Если же вы передаете в параметре *cbStack* нулевое значение, *CreateThread* создает стек для нового потока, используя информацию, встроенную компоновщиком в EXE-файл.

Значение аргумента *reserve* устанавливает верхний предел для стека, и это ограничение позволяет прекращать деятельность функций с бесконечной рекурсией. Допустим, вы пишете функцию, которая рекурсивно вызывает сама себя. Предположим также, что в функции есть «жучок», приводящий к бесконечной рекурсии. Всякий раз, когда функция вызывает сама себя, в стеке создается новый стековый фрейм. Если бы система не позволяла ог-

раничивать максимальный размер стека, рекурсивная функция так и вызывала бы сама себя до бесконечности, а стек поглотил бы все адресное пространство процесса. Задавая же определенный предел, вы, во-первых, предотвращаете разрастание стека до гигантских объемов и, во-вторых, гораздо быстрее узнаете о наличии ошибки в своей программе. (Программа-пример Summation в главе 16 продемонстрирует, как перехватывать и обрабатывать переполнение стека в приложениях.)

Параметры *pfhStartAddr* и *pvParam*

Параметр *pfnStartAddr* определяет адрес функции потока, с которой должен будет начать работу создаваемый поток, а параметр *pvParam* идентичен параметру *pvParam* функции потока. *CreateThread* лишь передает этот параметр по эстафете той функции, с которой начинается выполнение создаваемого потока. Таким образом, данный параметр позволяет передавать функции потока какое-либо инициализирующее значение. Оно может быть или просто числовым значением, или указателем на структуру данных с дополнительной информацией.

Вполне допустимо и даже полезно создавать несколько потоков, у которых в качестве входной точки используется адрес одной и той же функции. Например, можно реализовать веб-сервер, который обрабатывает каждый клиентский запрос в отдельном потоке. При создании каждому потоку передается свое значение *pvParam*.

Учтите, что Windows — операционная система с вытесняющей многозадачностью, а следовательно, новый поток и поток, вызвавший *CreateThread,* могут выполняться одновременно. В связи с этим возможны проблемы. Остерегайтесь, например, такого кода:

```
DWORD WINAPI FirstThread(PVOID pvParam) {
    // инициализируем переменную, которая содержится в стеке
    int x = 0;
    DWORD dwThreadID;

    // создаем новый поток
    HANDLE hThread = CreateThread(NULL, 0, 8econdThread, (PV0ID) &x,
        0, &dwThreadID);

    // Мы больше не ссылаемся на новый поток,
    // поэтому закрываем свой описатель этого потока.
    CloseHandle(hThread);

    // Наш поток закончил работу.
    // ОШИБКА: его стек будет разрушен, но SecondThread
    // может попытаться обратиться к нему.
    return(0);
}
```

```
DWORD WINAPI SecondThread(PVOID pvParam) {
    // здесь выполняется какая-то длительная обработка...
    // Пытаемся обратиться к переменной в стеке FirstThread.
    // ПРИМЕЧАНИЕ: это может привести к ошибке общей защиты –
    // нарушению доступа!
    * ((int *) pvParam) * 5; ...       return(0);
}
```

Не исключено, что в приведенном коде *FirstThread* закончит свою работу до того, как *SecondThread* присвоит значение 5 переменной *x* из *FirstThread*. Если так и будет, *SecondThread* не узнает, что *FirstThread* больше не существует, и попытается изменить содержимое какого-то участка памяти с недействительным теперь адресом. Это неизбежно вызовет нарушение доступа: стек первого потока уничтожен по завершении *FirstThread*. Что же делать? Можно объявить *x* статической переменной, и компилятор отведет память для хранения переменной *x* не в стеке, а в разделе данных приложения (application's data section).

Но тогда функция станет нереентерабельной. Иначе говоря, в этом случае вы не смогли бы создать два потока, выполняющих одну и ту же функцию, так как оба потока совместно использовали бы статическую переменную. Другое решение этой проблемы (и его более сложные варианты) базируется на методах синхронизации потоков, речь о которых пойдет в главах 8, 9 и 10.

Параметр *dwCreateFlags*

Этот параметр определяет дополнительные флаги, управляющие созданием потока. Он принимает одно из двух значений: 0 (исполнение потока начинается немедленно) или CREATE_SUSPENDED. В последнем случае система создает поток, инициализирует его и приостанавливает до последующих указаний.

Флаг CREATE_SUSPENDED позволяет программе изменить какие-либо свойства потока перед тем, как он начнет выполнять код. Правда, необходимость в этом возникает довольно редко. Одно из применений этого флага демонстрирует программа-пример JobLab из главы 5.

Параметр *pdwThreadID*

Последний параметр функции *CreateThread* — это адрес переменной типа DWORD, в которой функция возвращает идентификатор, приписанный системой новому потоку. (Идентификаторы процессов и потоков рассматривались в главе 4.) Вы можете (чаще всего так и поступают) передавать NULL в этом параметре. Так можно сообщить функции о том, что идентификатор потока вас не интересует.

Завершение потока

Поток можно завершить четырьмя способами:

- функция потока возвращает управление (рекомендуемый способ);
- поток самоуничтожается вызовом функции *ExitThread* (нежелательный способ);
- один из потоков данного или стороннего процесса вызывает функцию *TerminateThread* (нежелательный способ);
- завершается процесс, содержащий данный поток (тоже нежелательно).

В этом разделе мы обсудим перечисленные способы завершения потока, а также рассмотрим, что на самом деле происходит в момент его окончания.

Возврат управления функцией потока

Функцию потока следует проектировать так, чтобы поток завершался только после того, как она возвращает управление. Это единственный способ, гарантирующий корректную очистку всех ресурсов, принадлежавших вашему потоку. При этом:

- любые C++-объекты, созданные данным потоком, уничтожаются соответствующими деструкторами;
- система корректно освобождает память, которую занимал стек потока;
- система устанавливает код завершения данного потока (поддерживаемый объектом ядра «поток») — его и возвращает ваша функция потока;
- счетчик пользователей данного объекта ядра «поток» уменьшается на 1.

Функция *ExitThread*

Поток можно завершить принудительно, вызвав:

```
VOID ExitThread(DWORD dwExitCode);
```

При этом освобождаются все ресурсы операционной системы, выделенные данному потоку, но C/C++-ресурсы (например, объекты, созданные из C++-классов) не очищаются. Именно поэтому лучше возвращать управление из функции потока, чем самому вызывать функцию *ExitThread*. (Подробнее на эту тему см. раздел «Функция ExitProcess» в главе 4.)

В параметр *dwExitCode* вы помещаете значение, которое система рассматривает как код завершения потока. Возвращаемого значения у этой функции нет, потому что после ее вызова поток перестает существовать.

> **Примечание.** *ExitThread* — это Windows-функция, которая уничтожает поток. Но никогда не вызывайте ее, если вы пишете код на C/C++. Вместо нее вы должны использовать функцию *_endthreadex* из библиотеки Visual C++. (Если вы работаете с другим компилятором, он должен поддерживать свой эквивалент функции *ExitThread*.) Что именно делает *_endthreadex* и почему это так важно, я объясню потом.

Функция *TermInateThread*

Вызов этой функции также завершает поток:

```
BOOL TerminateThread(
    HANDLE hThread,
    DWORD dwExitCode):
```

В отличие от *ExitThread*, которая уничтожает только вызывающий поток, эта функция завершает поток, указанный в параметре *hThread*. В параметр *dwExitCode* вы помещаете значение, которое система рассматривает как код завершения потока. После того как поток будет уничтожен, счетчик пользователей его объекта ядра «поток» уменьшится на 1.

> **Примечание.** *TerminateThread* — функция асинхронная, т. е. она сообщает системе, что вы хотите завершить поток, но к тому времени, когда она вернет управление, поток может быть еще не уничтожен. Так что, если вам нужно точно знать момент завершения потока, используйте *WaitForSingleObject* (см. главу 9) или аналогичную функцию, передав ей описатель этого потока.

Корректно написанное приложение не должно вызывать эту функцию, поскольку поток не получает никакого уведомления о завершении; из-за этого он не может выполнить должную очистку ресурсов.

> **Примечание.** Уничтожение потока при вызове *ExitThread* или возврате управления из функции потока приводит к разрушению его стека. Но если он завершен функцией *TerminateThread*, система не уничтожает стек, пока не завершится и процесс, которому принадлежал этот поток. Так сделано потому, что другие потоки могут использовать указатели, ссылающиеся на данные в стеке завершенного потока. Если бы они обратились к несуществующему стеку, произошло бы нарушение доступа. Кроме того, при завершении потока система уведомляет об этом все DLL, подключенные к процессу — владельцу завершенного потока. Но при вызове *TerminateThread* такого не происходит, и процесс может быть завершен некорректно. (Подробнее на эту тему см. главу 20.)

Если завершается процесс

Функции *ExitProcess* и *TerminateProcess*, рассмотренные в главе 4, тоже завершают потоки. Единственное отличие в том, что они прекращают выполнение всех потоков, принадлежавших завершенному процессу. При этом гарантируется высвобождение любых выделенных процессу ресурсов, в том числе стеков потоков. Однако эти две функции уничтожают потоки принудительно — так, будто для каждого из них вызывается функция *TerminateThread*. А это означает, что очистка проводится некорректно: деструкторы C++-объектов не вызываются, данные на диск не сбрасываются и т. д.

Как я уже говорил в начале главы, когда входная функция приложения возвращает управления, стартовый код библиотеки C/C++ вызывает ***ExitProcess***. Поэтому в многопоточном приложении необходимо явно позаботиться о корректной остановке каждого потока до завершения главного потока. В противном случае все потоки «погибнут» внезапно и молча.

Что происходит при завершении потока

А происходит вот что:

- Освобождаются все описатели User-объектов, принадлежавших потоку. В Windows большинство объектов принадлежит процессу, содержащему поток, из которого они были созданы. Сам поток владеет только двумя User-объектами: окнами и ловушками (hooks). Когда поток, создавший такие объекты, завершается, система уничтожает их автоматически. Прочие объекты разрушаются, только когда завершается владевший ими процесс.
- Код завершения потока меняется со STILL_ACTIVE на код, переданный в функцию *ExitThread* или *TerminateThread*.
- Объект ядра «поток» переводится в свободное состояние.
- Если данный поток является последним активным потоком в процессе, завершается и сам процесс.
- Счетчик пользователей объекта ядра «поток» уменьшается на 1.

При завершении потока сопоставленный с ним объект ядра «поток» не освобождается до тех пор, пока не будут закрыты все внешние ссылки на этот объект.

Когда поток завершился, толку от его описателя другим потокам в системе в общем немного. Единственное, что они могут сделать, — вызвать функцию *GetExitCodeThread*, проверить, завершен ли поток, идентифицируемый описателем *hThread*, и, если да, определить его код завершения.

```
BOOL GetExitCodeThread(
    HANDLE hThread,
    PDWORD pdwExitCode);
```

Код завершения возвращается в переменной типа DWORD, на которую указывает *pdwExitCode*. Если поток не завершен на момент вызова *GetExitCodeThread*, функция записывает в эту переменную идентификатор STILL_ACTIVE (0x103). При успешном вызове функция возвращает TRUE. К использованию описателя для определения факта завершения потока мы еще вернемся в главе 9.

Кое-что о внутреннем устройстве потока

Я уже объяснил вам, как реализовать функцию потока и как заставить систему создать поток, который выполнит эту функцию. Теперь мы попробуем разобраться, как система справляется с данной задачей.

На рис. 6-1 показано, что именно должна сделать система, чтобы создать и инициализировать поток.

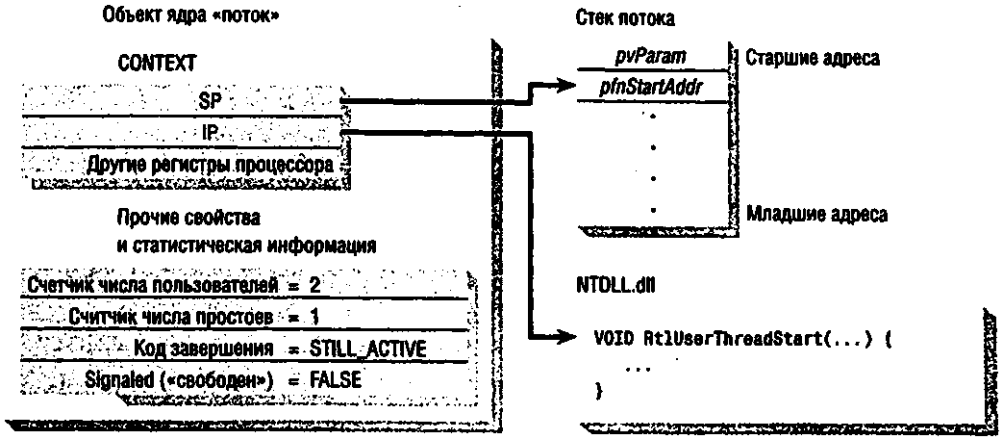


Рис. 6-1. *Так создается и инициализируется поток*

Давайте приглядимся к этой схеме внимательнее. Вызов *CreateThread* заставляет систему создать объект ядра «поток». При этом счетчику числа его пользователей присваивается начальное значение, равное 2. (Объект ядра «поток» уничтожается только после того, как прекращается выполнение потока и закрывается описатель, возвращенный функцией *CreateThread*.) Также инициализируются другие свойства этого объекта: счетчик числа простоев (suspension count) получает значение 1, а код завершения — значение STILL_ACTIVE (0x103). И, наконец, объект переводится в состояние «занято».

Создав объект ядра «поток», система выделяет стеку потока память из адресного пространства процесса и записывает в его самую верхнюю часть два значения. (Стеки потоков всегда строятся от старших адресов памяти к младшим.) Первое из них является значением параметра *pvParam*, переданного вами функции *CreateThread*, а второе — это содержимое параметра *pfnStartAddr*, который вы тоже передаете в *CreateThread*.

У каждого потока собственный набор регистров процессора, называемый *контекстом* потока. Контекст отражает состояние регистров процессора на момент последнего исполнения потока и записывается в структуру CONTEXT (она определена в заголовочном файле WinNT.h). Эта структура содержится в объекте ядра «поток».

Указатель команд (IP) и указатель стека (SP) — два самых важных регистра в контексте потока. Вспомните: потоки выполняются в контексте процесса.

Соответственно эти регистры всегда указывают на адреса памяти в адресном пространстве процесса. Когда система инициализирует объект ядра «поток», указателю стека в структуре CONTEXT присваивается тот адрес, по которому в стек потока было записано значение pfhStartAddr, а указателю команд — адрес недокументированной (и неэкспортируемой) функции *RtlUserThreadStart*. Эта функция содержится в модуле NTDLLdll (см. рис. 6-1).

Вот главное, что делает *RtlUserThreadStart*:

```
VOID RtlUserThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam)
{
    _try {
        ExitThread((pfnStartAddr)(pvParam));
    }
    __except(UnhandledExceptlonFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // ПРИИЕЧАНИЕ: мы никогда не попадем сюда
}
```

После инициализации потока система проверяет, был ли передан функции *CreateThread* флаг CREATE_SUSPENDED. Если нет, система обнуляет его счетчик числа простоев, и потоку может быть выделено процессорное время. Далее система загружает в регистры процессора значения, сохраненные в контексте потока. С этого момента поток может выполнять код и манипулировать данными в адресном пространстве своего процесса.

Поскольку указатель команд нового потока установлен на *RtlUserThreadStart*, именно с этой функции и начнется выполнение потока. Глядя на ее прототип, можно подумать, будто *RtlUserThreadStart* передаются два параметра, а значит, она вызывается из какой-то другой функции, но это не так. Новый поток просто начинает с нее свою работу. *RtlUserThreadStart* получает доступ к двум параметрам, которые появляются у нее потому, что операционная система записывает соответствующие значения в стек потока (а через него параметры как раз и передаются функциям). Правда, на некоторых аппаратных платформах параметры передаются не через стек, а с использованием определенных регистров процессора. Поэтому на таких аппаратных платформах система — прежде чем разрешить потоку выполнение функции *RtlUserThreadStart* — инициализирует нужные регистры процессора.

Когда новый поток выполняет *RtiUserThreadStort;nponcxojwr* следующее.

- Ваша функция потока включается во фрейм структурной обработки исключений (далее для краткости — SEH-фрейм), благодаря чему любое исключение, если оно происходит в момент выполнения вашего потока, получает хоть какую-то обработку, предлагаемую системой по умолчанию. Подробнее о структурной обработке исключений см. главы 23,24 и 25.
- Система обращается к вашей функции потока, передавая ей параметр *pvParam*, который вы ранее передали функции *CreateThread*.

- Когда ваша функция потока возвращает управление, *RtlUserThreadStart* вызывает *ExitThread*, передавая ей значение, возвращенное вашей функцией. Счетчик числа пользователей объекта ядра «поток» уменьшается на 1, и выполнение потока прекращается.
- Если ваш поток вызывает необрабатываемое им исключение, его обрабатывает SEH-фрейм, построенный функцией *RtlUserThreadStart*. Обычно в результате этого появляется окно с каким-нибудь сообщением, и, когда пользователь закрывает его, *RtlUserThreadStart* вызывает *ExitProcess* и завершает весь процесс, а не только тот поток, в котором произошло исключение.

Обратите внимание, что из *RtlUserThreadStart* поток вызывает либо *ExitThread*, либо *ExitProcess*. А это означает, что поток никогда не выходит из данной функции; он всегда уничтожается внутри нее. Вот почему у *RtlUserThreadStart* нет возвращаемого значения — она просто ничего не возвращает.

Кстати, именно благодаря *RtlUserThreadStart* ваша функция потока получает возможность вернуть управление по окончании своей работы. *RtlUserThreadStart*, вызывая функцию потока, заталкивает в стек свой адрес возврата и тем самым сообщает ей, куда надо вернуться. Но сама *RtlUserThreadStart* не возвращает управление. Иначе возникло бы нарушение доступа, так как в стеке потока нет ее адреса возврата.

При инициализации первичного потока его указатель команд устанавливается на ту же недокументированную функцию — *RtlUserThreadStart*.

Функция *RtlUserThreadStart* обращается к стартовому коду библиотеки C/C++, который выполняет необходимую инициализацию, а затем вызывает вашу входную функцию *_tmain* или *_tWinMain*. Когда входная функция возвращает управление, стартовый код библиотеки C/C++ вызывает ***ExitProcess***. Поэтому первичный поток приложения, написанного на C/C++, никогда не возвращается в *RtlUserThreadStart*.

Некоторые соображения по библиотеке C/C++

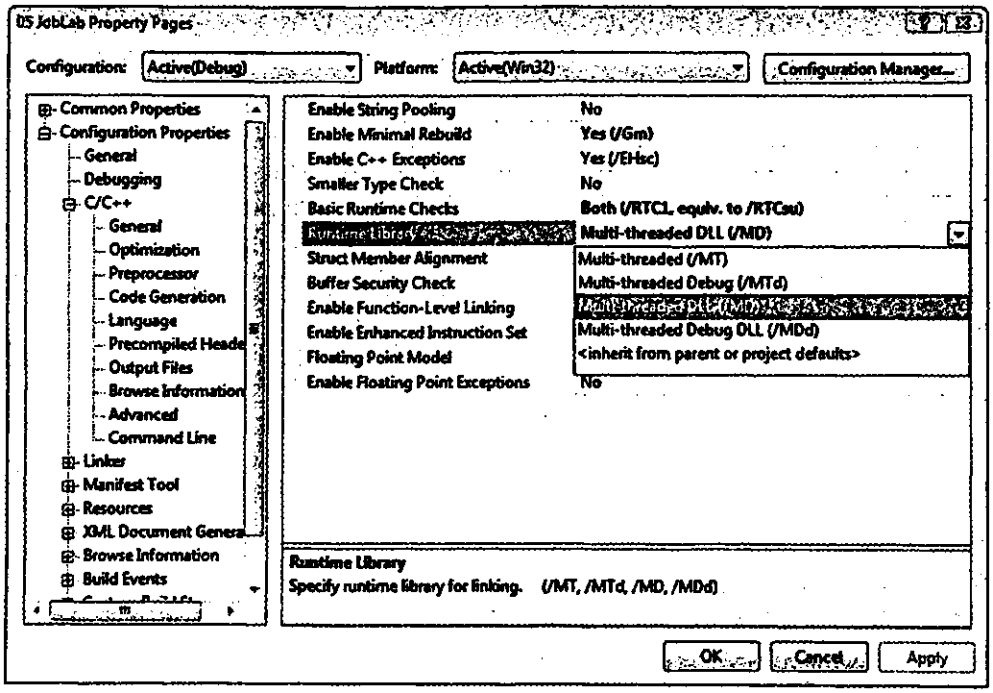
Майкрософт поставляет с Visual Studio шесть библиотек C/C++ (четыре «родные» и две — для управляемого мира .NET). Их краткое описание представлено в следующей таблице.

Табл. 6-1. Библиотеки C/C++, поставляемые с Visual Studio

Имя библиотеки	Описание
LibCMt.lib	Статически подключаемая библиотека для многопоточных приложений
LibCMtD.lib	Отладочная версия статически подключаемой библиотеки для многопоточных приложений
MSVCRt.lib	Библиотека импорта для динамического подключения рабочей версии MSVCR80.dll; поддерживает как одно-, так и многопоточные приложения (используется с новыми проектами)

Табл. 6-1. *(окончание)*

Имя библиотеки	Описание
MSVCRtD.lib	Библиотека импорта для динамического подключения отладочной версии MSVCR80D.dll; поддерживает как одно-, так и много поточные приложения
MSVCMRt.lib	Библиотека импорта для смешанного (управляемого и «родного») кода
MSVCURt.lib	Библиотека импорта для «чистого» MSIL-кода



При реализации любого проекта нужно знать, с какой библиотекой его следует связать. Конкретную библиотеку можно выбрать в диалоговом окне Project Settings: на вкладке C/C++ в списке Category укажите Code Generation, а в списке Use Run-Time Library — одну из шести библиотек.

Наверное, вам уже хочется спросить: «А зачем мне отдельные библиотеки для однопоточных и многопоточных программ?» Отвечаю. Стандартная библиотека C была разработана где-то в 1970 году — задолго до появления самого понятия многопоточности. Авторы этой библиотеки, само собой, не задумывались о проблемах, связанных с многопоточными приложениями.

Возьмем, к примеру, глобальную переменную *errno* из стандартной библиотеки C. Некоторые функции, если происходит какая-нибудь ошибка, записывают в эту переменную соответствующий код. Допустим, у вас есть такой фрагмент кода:

```
BOOL fFailure = (system("NOTEPAD.EXE README.TXT") == -1);

if (fFailure) {
    switch (errno) {
    case E2BIG:     // список аргументов или размер
                    // окружения слишком велик
       break;
    case ENOENT:    // командный интерпретатор не найден
       break;
    case ENOEXEC:   // неверный формат командного интерпретатора
       break;
    case ENONEN:    // недостаточно памяти для выполнения команды
       break;
    }
}
```

Теперь представим, что поток, выполняющий показанный выше код, прерван после вызова функции *system* и до оператора *if*. Допустим также, поток прерван для выполнения другого потока (в том же процессе), который обращается к одной из функций библиотеки C, и та тоже заносит какое-то значение в глобальную переменную *errno*. Смотрите, что получается: когда процессор вернется к выполнению первого потока, в переменной *errno* окажется вовсе не то значение, которое было записано функцией *system*. Поэтому для решения этой проблемы нужно закрепить за каждым потоком свою переменную *errno*. Кроме того, понадобится какой-то механизм, который позволит каждому потоку ссылаться на свою переменную *errno* и не трогать чужую.

Это лишь один пример того, что стандартная библиотека C/C++ не рассчитана на многопоточные приложения. Кроме *errno*, в ней есть еще целый ряд переменных и функций, с которыми возможны проблемы в многопоточной среде: *_doserrno*, *strtok*, *_wcstok*, *strerror*, *_strerror*, *tmpnam*, *tmpfile*, *asctime*, *_wasctime*, *gmtime*, *_ecvt*, *_fcvt* — список можно продолжить.

Чтобы многопоточные программы, использующие библиотеку C/C++, работали корректно, требуется создать специальную структуру данных и связать ее с каждым потоком, из которого вызываются библиотечные функции. Более того, они должны знать, что, когда вы к ним обращаетесь, нужно просматривать этот блок данных в вызывающем потоке, чтобы не повредить данные в каком-нибудь другом потоке.

Так откуда же система знает, что при создании нового потока надо создать и этот блок данных? Ответ очень прост: не знает и знать не хочет. Вся ответственность — исключительно на вас. Если вы пользуетесь небезопасными в многопоточной среде функциями, то должны создавать потоки библиотечной функцией *_beginthreadex*, а не Windows-функцией *CreateThread*.

```
unsigned long _beginthreadex(
```

```
    void *security,
    unsigned stack_size,
    unsigned (*start_address)(void *),
    void *arglist,
    unsigned initflag,
    unsigned *thrdaddr);
```

У функции *_beginthreadex* тот же список параметров, что и у *CreateThread*, но их имена и типы несколько отличаются. (Группа, которая отвечает в Майкрософт за разработку и поддержку библиотеки C/C++, считает, что библиотечные функции не должны зависеть от типов данных Windows.) Как и *CreateThread*, функция *_beginthreadex* возвращает описатель только что созданного потока. Поэтому, если вы раньше пользовались функцией *CreateThread*, ее вызовы в исходном коде несложно заменить на вызовы *_beginthreadex*. Однако из-за некоторого расхождения в типах данных вам придется позаботиться об их приведении к тем, которые нужны функции *_beginthreadex*, и тогда компилятор будет счастлив. Лично я создал небольшой макрос, *chBEGINTHREADEX*, который и делает всю эту работу в исходном коде.

```
typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr,  \
    pvParam, fdwCreate, pdwThreadID)                 \
        ((HANDLE) _beginthreadex(                    \
            (void *) (psa),                          \
            (unsigned) (cbStackSize),                \
            (PTHREAD_START) (pfnStartAddr).          \
            (void *) (pvParam),                      \
            (unsigned) (dwCreateFlags),              \
            (unsigned *) (pdwThreadID)))
```

Поскольку Майкрософт поставляет исходный код библиотеки C/C++, несложно разобраться в том, что такого делает *_begtnthreadex*, чего не делает *CreateThread*. На жестком диске Visual Studio ее исходный код содержится в файле <Program Files>\Microsoft Visual Studio 8\VC\crt\src\Threadex.c. Чтобы не перепечатывать весь код, я решил дать вам ее версию в псевдокоде, выделив самые интересные места.

```
uintptr_t __cdecl _beginthreadex (
    void *psa,
    unsigned cbStackSize,
    unsigned (__stdcall * pfnStartAddr) (void *),
    void * pvParam,
    unsigned dwCreateFlags,
    unsigned *pdwThreadID) {
```

```
    _ptiddata ptd;                          // указатель на блок данных потока
    uintptr_t thdl;                         // описатель потока

    // выделяется блок данных для нового потока
    if ((ptd = (_ptiddata)_calloc_crt(1, sizeof(struct _tiddata))) == NULL)
        goto error_return;

    // инициализация блока данных
    initptd(ptd);

    // Здесь запоминается нужная функция потока и параметр,
    // который мы хотим поместить в блок данных.
    ptd->_initaddr = (void *) pfnStartAddr;
    ptd->_initarg = pvParam;
    ptd->_thandle = (uintptr_t)(-1);

    // создание нового потока.
    thdl * (uintptr_t) CreateThread((LPSECURITY_ATTRIBUTES)psa, cbStackSize,
        _threadstartex, (PVOID) ptd,  dwCreateFlags, pdwThreadID);
    if (thdl == 0) {
        // создать поток не удалось; проводится очистка и сообщается об ошибке
        goto error_return;
    }

    // поток успешно создан; возвращается его описатель
    return(thdl);

error_return:
    // Ошибка: не удалось создать блок данных или сам поток.
    // GetLastError() сопоставлена соответствующим значениям errno,
    // которые возвращаются при ошибке CreateThread.
    _free_crt(ptd);
    return((uintptr_t)0L);
}
```

Несколько важных моментов, связанных с *_beginthreadex*.

- Каждый поток получает свой блок памяти tiddata, выделяемый из кучи, которая принадлежит библиотеке C/C++.
- Адрес функции потока, переданный _beginthreadex, запоминается в блоке памяти _tiddata (определен в заголовочном файле Mtdll.h). Там же сохраняется и параметр, который должен быть передан этой функции.
- Функция _beginthreadex вызывает CreateThread, так как лишь с ее помощью операционная система может создать новый поток.
- При вызове CreateThread сообщается, что она должна начать выполнение нового потока с функции _threadstartex, а не с того адреса, на кото-

рый указывает pfhStartAddr. Кроме того, функции потока передается не параметр pvParam, а адрес структуры _tiddata.

- Если все проходит успешно, _beginthreadex, как и CreateThread, возвращает описатель потока. В ином случае возвращается 0.

```
struct _tiddata {
    unsigned long  _tid;                    /* идентификатор потока */
    unsigned long  _thandle;                /* описатель потока */
    int      _terrno;                       /* значение errno */
    unsigned long  _tdoserrno;              /* значение _doserrno */
    unsigned int   _fpds;                   /* сегмент данных Floating Point */
    unsigned long  _holdrand;               /* зародышевое значение для rand() */
    char*          _token;                  /* указатель (ptr) на метку strtok()*/
    wchar_t*       _wtoken;                 /* ptr на метку wcstok() */
    unsigned char* _mtoken;                 /* ptr на метку jnbstok()  */

    /* следующие указатели обрабатываются функцией roalloc в период выполнения */
    char*          _errrosg;                /* ptr to strerror()/_strerror() buff*/
    wchar_t*       _werrrosg;               /* ptr на буфер strerror()/_strerror()*/
    char*          _namebuf0;               /* ptr на буфер tmpnam() */
    wchar_t*       _wnamebuf0;              /* ptr на буфер _wtmpnam() */
    char*          _namebuf1;               /* ptr на буфер tropfile() */
    wchar_t*       _wnamebuf1;              /* ptr на буфер _wtmpfile() */
    char*          _asctimebuf;             /* ptr на буфер asctiroe() */
    wchar_t*       _wasctiroebuf;           /* ptr на буфер _wasctiroe() */
    void*          _gmtiroebuf;             /* ptr на структуру gmtime() */
    char*          _cvtbuf;                 /* ptr на буфер ecvt()/fcvt */

    unsigned char _con_ch_buf[MB_LEM_MAX];
                                            /* ptr на буфер putch() */
    unsigned short _ch_buf_used;/* используется ли _con_ch_buf */

    /* следующие поля используются кодом _beginthread  */
    void*      _initaddr;          /* начальный адрес пользовательского потока*/
    void*      _initarg;           /* начальный аргумент пользовательского потока */

    /* следующие три поля нужны для поддержки функции signal и обработки ошибок,
    * возникающих в период выполнения */
    void*      _pxcptacttab;       /* ptr на таблицу "исключение-действие" */
    void*      _tpxcptinfoptrs;    /* ptr на указатели к информации об исключении */
    int        _tfpecode;          /* код исключения для операций над числами
                                    * с плавающей точкой */

    /* указатель на копию мультибайтовых ресурсов потока */
    pthreadrobcinfo ptnbcinfo;
```

```
    /* указатель на копию локализованных ресурсов потока */
    pthreadlocinfo  ptlocinfo;
    int        _ownlocale;        /* если равно 1, у потока собственные
                                  региональные параметры */

    /* следующее поле нужно подпрограммам NLG */
    Unsigned   long               _NLG_dwCode;

    /*
    * данные для отдельного потока, используемые при обработке исключений в C++
    */
    void*      _terminate;        /* подпрограмма terminate() */
    void*      _unexpected;       /* подпрограмма unexpected() */
    void*      _translator;       /* транслятор S.E. */
    void*      _purecall;         /* для чисто виртуальных функций */
    void*      _curexception;     /* текущее исключение */
    void*      _curcontext;       /* контекст текущего исключения */
    int        _ProcessingThrow;  /* для неперехваченных исключений */
    void*      _curexcspec;       /* для обработки исключений, сгенерированных
                                  std::unexpected */

#if defined (_M_IA64) || defined (_M_AMD64)
    void *    _pExitContext;
    void*     _pUnwindContext;
    void*     _pFrameInfoChain;
    unsigned _int64 _ImageBase;
#if defined(_M_IA64)
    unsigned __int64 _TargetGp;
#endif /* defined (JLIA64) */
    unsigned __int64     _ThrowImageBase;
    void*     _pForeignException;
#elif defined (_M_IX86)
    void*     _pFrameInfoChain;
#endif /* defined (_M_IX86) */
    _setloc_struct _setloc_data;

    void*     _encode_ptr;     /* процедура EncodePointer()*/
    void*     _decode_ptr;     /* процедура DecodePointer()*/

    void*     _reserved1;      /* пусто */
    void*     _reserved2;      /* пусто */
    void*     _reserved3;      /* пусто */

    int _     cxxReThrow;      /* для повторно сгенерированных исключений
                               устанавливается в True */
```

```
    unsigned long __initDomain;     /* домен, который _beginthread[ex]
                                       изначально использует для управляемого кода */
};

typedef struct _tiddata * _ptiddata;
```

Выяснив, как создается и инициализируется структура *_tiddata* для нового потока, посмотрим, как она сопоставляется с этим потоком. Взгляните на исходный код функции *_threadstartex* (который тоже содержится в файле Threadex.c библиотеки C/C++). Вот моя версия этой функции в псевдокоде (со вспомогательной функцией *__callthreadstartex*):

```
static unsigned long WINAPI _threadstartex (void* ptd) {
    // Примечание: ptd - это адрес блока tiddata данного потока

    // блок tiddata сопоставляется с данным потоком
    // _getptd() найдет его в _callthreadstartex
    TlsSetValue(__tlsindex, ptd);

    // идентификатор этого потока записывается в tiddata
    ((_ptiddata) ptd)->_tid = GetCurrentThreadId();

    // здесь инициализируется поддержка операций над числами с плавающей точкой

    // вызов вспомогательной функции
    _callth read sta rtex();

    // Сюда мы никогда не попадем - по завершении _callthreadstartex поток умирает
    return(0L);
}

static void _callthreadstartex(void) {
    _ptiddata ptd; /* указатель на структуру _tiddata потока */

    // получаем указатель на данные потока из TLS
    ptd = _getptd();

    // Пользовательская функция потока включается в SEH-фрейм для обработки
    // ошибок периода выполнения и поддержки signal.
    __try {
        // Здесь вызывается функция потока, которой передается нужный параметр;
        // код завершения потока передается _endthreadex.
        _endthreadex(
            ( (unsigned (WINAPI *)(void *))(((_ptiddata)ptd)->_initaddr) )
                ( ((_ptiddata)ptd)->_initarg ) ) ;
    }
    __except(_XcptFilter(GetExceptionCode(), GetExceptionInformation())){
        // Обработчик исключений из библиотеки C не даст нам попасть сюда
```

```
        // здесь мы тоже никогда не будем, так как в этой функции поток умирает.
        _exit(GetExceptionCode());
    }
}
```

Несколько важных моментов, связанных со *_threadstartex*.

- Новый поток начинает выполнение с *RtlUserThreadStart* (in NTDLLdll), а затем переходит в *Jhreadstartex*.
- В качестве единственного параметра функции *_threadstartex* передается адрес блока *_tiddata* нового потока.
- Windows-функция *TkSetValue* сопоставляет с вызывающим потоком значение, которое называется локальной памятью потока (Thread Local Storage, TLS) (о ней я расскажу в главе 21), а *_threadstartex* сопоставляет блок *_tiddata* с новым потоком.
- Функция потока заключается в SEH-фрейм. Он предназначен для обработки ошибок периода выполнения (например, не перехваченных исключений C++), поддержки библиотечной функции *signal* и др. Этот момент, кстати, очень важен. Если бы вы создали поток с помощью CreateThread, а потом вызвали библиотечную функцию *signal*, она работала бы некорректно.
- Далее вызывается функция потока, которой передается нужный параметр. Адрес этой функции и ее параметр были сохранены в блоке *_tiddata* в TLS функцией *_beginthreadex*, извлекают их из TLS функцией *_callthreadstartex*.
- Значение, возвращаемое функцией потока, считается кодом завершения этого потока. Обратите внимание: *_callthreadstartex* не возвращается в *_threadstartex* и затем в *_RtlUserThreadStart*. Иначе после уничтожения потока его блок *_tiddata* так и остался бы в памяти. А это привело бы к утечке памяти в вашем приложении. Чтобы избежать этого, *_threadstartex* вызывает другую библиотечную функцию, *_endthreadex*, и передает ей код завершения.

Последняя функция, которую нам нужно рассмотреть, — это *_endthreadex* (ее исходный код тоже содержится в файле Threadex.c). Вот как она выглядит в моей версии (в псевдокоде):

```
void __cdecl _endthreadex (unsigned retcode) {
    _ptiddata ptd;                          // указатель на блок данных потока

    // здесь проводится очистка ресурсов, выделенных для поддержки операций
    // над числами с плавающей точкой (код не показан)

    // определение адреса блока tiddata данного потока
    ptd = _getptd_noexit ();

    // высвобождение блока tiddata
    if (ptd !* NULL)
        _freeptd(ptd);
```

```
    // завершение потока
    ExitThread(retcode);
}
```

Несколько важных моментов, связанных с *_endthreadex*.

- Библиотечная функция *_getptd_noexit* обращается к Windows-функции *TlsGetValue*, которая сообщает адрес блока памяти *_tiddata* вызывающего потока.
- Этот блок освобождается, и вызовом *ExitThread* поток разрушается. При этом, конечно, передается корректный код завершения.

Где-то в начале главы я уже говорил, что прямого обращения к функции *ExitThread* следует избегать. Это правда, и я не отказываюсь от своих слов. Тогда же я сказал, что это приводит к уничтожению вызывающего потока и не позволяет ему вернуться из выполняемой в данный момент функции. А поскольку она не возвращает управление, любые созданные вами C++-объекты не разрушаются. Так вот, теперь у вас есть еще одна причина не вызывать *ExitThread*: она не дает освободить блок памяти *tiddata* потока, из-за чего в вашем приложении может наблюдаться утечка памяти (до его завершения).

Разработчики Microsoft Visual C++, конечно, прекрасно понимают, что многие все равно будут пользоваться функцией *ExitThread*, поэтому они кое-что сделали, чтобы свести к минимуму вероятность утечки памяти. Если вы действительно так хотите самостоятельно уничтожить свой поток, можете вызвать из него *_endthreadex* (вместо *ExitThread*) и тем самым освободить его блок *tiddata*. И все же я не рекомендую этого.

Сейчас вы уже должны понимать, зачем библиотечным функциям нужен отдельный блок данных для каждого порождаемого потока и каким образом после вызова *_beginthreadex* происходит создание и инициализация этого блока данных, а также его связывание с только что созданным потоком. Кроме того, вы уже должны разбираться в том, как функция *_endthreadex* освобождает этот блок по завершении потока.

Как только блок данных инициализирован и сопоставлен с конкретным потоком, любая библиотечная функция, к которой обращается поток, может легко узнать адрес его блока и таким образом получить доступ к данным, принадлежащим этому потоку (через *TlsGetValue*). Ладно, с функциями все ясно, теперь попробуем проследить, что происходит с глобальными переменными вроде *errno*. В заголовочных файлах C эта переменная определена так:

```
_CRTIMP extern int * __cdecl _errno(void);
#define errno   (*_errno())

int* __cdecl _errno(void) {
    _ptiddata ptd = _getptd_noexit();
    if (!ptd) {
        return &ErrnoNoMem;
    } else {
```

```
        return (Aptd->_terrno);
    }
}
```

Ссылаясь на *errno,* вы будете на самом деле вызывать внутреннюю функцию *_errno* из библиотеки C/C++. Она возвращает адрес элемента данных *errno* в блоке, сопоставленном с вызывающим потоком. Кстати, макрос *errno* составлен так, что позволяет получать содержимое памяти по этому адресу. А сделано это для того, чтобы можно было писать, например, такой код:

```
int *p = &errno;
if (*p == ENOMEM) {
    …
}
```

Если бы внутренняя функция *_errno* просто возвращала значение *errno,* этот код не удалось бы скомпилировать.

Многопоточная версия библиотеки C/C++, кроме того, «обертывает» некоторые функции синхронизирующими примитивами. Ведь если бы два потока одновременно вызывали функцию *malloc,* куча могла бы быть повреждена. Поэтому в многопоточной версии библиотеки потоки не могут одновременно выделять память из кучи. Второй поток она заставляет ждать до тех пор, пока первый не выйдет из функции *malloc,* и лишь тогда второй поток получает доступ к *malloc*. (Подробнее о синхронизации потоков мы поговорим в главах 8 и 9).

Конечно, все эти дополнительные операции не могли не отразиться на быстродействии многопоточной версии библиотеки. Поэтому Майкрософт, кроме многопоточной, поставляет и однопоточную версию статически подключаемой библиотеки C/C++.

Динамически подключаемая версия библиотеки C/C++ вполне универсальна: ее могут использовать любые выполняемые приложения и DLL, которые обращаются к библиотечным функциям. По этой причине данная библиотека существует лишь в многопоточной версии. Поскольку она поставляется в виде DLL, ее код не нужно включать непосредственно в EXE-и DLL-модули, что существенно уменьшает их размер. Кроме того, если Майкрософт исправляет какую-то ошибку в такой библиотеке, то и программы, построенные на ее основе, автоматически избавляются от этой ошибки.

Как вы, наверное, и предполагали, стартовый код из библиотеки C/C++ создает и инициализирует блок данных для первичного потока приложения. Это позволяет без всяких опасений вызывать из первичного потока любые библиотечные функции. А когда первичный поток заканчивает выполнение своей входной функции, блок данных завершаемого потока освобождается самой библиотекой. Более того, стартовый код делает все необходимое для структурной обработки исключений, благодаря чему из первичного потока можно спокойно обращаться и к библиотечной функции *signal*.

Ой, вместо *_beginthreadex* я по ошибке вызвал *CreateThread*

Вас, наверное, интересует, что случится, если создать поток не библиотечной функцией *_beginthreadex*, а Windows-функцией *CreateThread*. Когда этот поток вызовет какую-нибудь библиотечную функцию, которая манипулирует со структурой *_tiddata*, произойдет следующее. (Большинство библиотечных функций реентерабельно и не требует этой структуры.) Сначала эта функция попытается выяснить адрес блока данных потока (вызовом *TlsGetValue*). Получив NULL вместо адреса *_tiddata*, она узнает, что вызывающий поток не сопоставлен с таким блоком. Тогда библиотечная функция тут же создаст и инициализирует блок *tiddata* для вызывающего потока. Далее этот блок будет сопоставлен с потоком (через *TlsSetValue*) и останется при нем до тех пор, пока выполнение потока не прекратится. С этого момента данная функция (как, впрочем, и любая другая из библиотеки C/C++) сможет пользоваться блоком *_tiddata* потока.

Как это ни фантастично, но ваш поток будет работать почти без глюков. Хотя некоторые проблемы все же появятся. Во-первых, если этот поток воспользуется библиотечной функцией *signal*, весь процесс завершится, так как SEH-фрейм не подготовлен. Во-вторых, если поток завершится, не вызвав *_endthreadex*, его блок данных не высвободится и произойдет утечка памяти. (Да и кто, интересно, вызовет ***_endthreadex*** из потока, созданного с помощью *CreateThread*?)

> **Примечание.** Если вы связываете свой модуль с многопоточной DLL-версией библиотеки C/C++, то при завершении потока и высвобождении блока *_tiddata* (если он был создан), библиотека получает уведомление DLL_THREAD_DETACH. Даже несмотря на то что это предотвращает утечку памяти, связанную с блоком *_tiddata*, я настоятельно советую создавать потоки через *_beginthreadex*, а не с помощью *CreateThread*.

Библиотечные функции, которые лучше не вызывать

В библиотеке C/C++ содержится две функции:

```
unsigned long _beginthread(
    void (__cdecl *start_address)(void *),
    unsigned stack_size,
    void *arglist);
```

и

```
void _endthread(void);
```

Первоначально они были созданы для того, чем теперь занимаются новые функции *_beginthreadex* и *_endthreadex*. Но, как видите, у *_beginthread* параметров меньше, и, следовательно, ее возможности ограничены в сравнении

с полнофункциональной *_beginthreadex*. Например, работая с *_beginthread*, нельзя создать поток с атрибутами защиты, отличными от присваиваемых по умолчанию, нельзя создать поток и тут же его задержать — нельзя даже получить идентификатор потока. С функцией *_endthread* та же история: она не принимает никаких параметров, а это значит, что по окончании работы потока его код завершения всегда равен 0.

Однако с функцией *_endthread* дело обстоит куда хуже, чем кажется: перед вызовом *ExitThread* она обращается к *CloseHandle* и передает ей описатель нового потока. Чтобы разобраться, в чем тут проблема, взгляните на следующий код:

```
DW0RD dwExitCode;
HANDLE hThread * _beginthread(...);
GetExitCodeThread(hThread, &dwExitCode);
CloseHandle(hThread);
```

Весьма вероятно, что созданный поток отработает и завершится еще до того, как первый поток успеет вызвать функцию *GetExitCodeThread*. Если так и случится, значение в *hThread* окажется недействительным, потому что *_endthreadymt* закрыла описатель нового потока. И, естественно, вызов *CloseHandle* даст ошибку.

Новая функция *_endthreadex* не закрывает описатель потока, поэтому фрагмент кода, приведенный выше, будет нормально работать (если мы, конечно, заменим вызов *_beginthread* на вызов *_beginthreadex*). И в заключение, напомню еще раз: как только функция потока возвращает управление, *_beginthreadex* самостоятельно вызывает *_endthreadex*, а *_beginthread* обращается к *_endthread*.

Как узнать о себе

Потоки часто обращаются к Windows-функциям, которые меняют среду выполнения. Например, потоку может понадобиться изменить свой приоритет или приоритет процесса. (Приоритеты рассматриваются в главе 7.) И поскольку это не редкость, когда поток модифицирует среду (собственную или процесса), в Windows предусмотрены функции, позволяющие легко ссылаться на объекты ядра текущего процесса и потока:

```
HANDLE GetCurrentProcess();
HANDLE GetCurrentThread();
```

Обе эти функции возвращают псевдоописатель объекта ядра «процесс» или «поток». Они не создают новые описатели в таблице описателей, которая принадлежит вызывающему процессу, и не влияют на счетчики числа пользователей объектов ядра «процесс» и «поток». Поэтому, если вызвать *CloseHandle* и передать ей псевдоописатель, она проигнорирует вызов и просто вернет FALSE, а *GetLastError* — ERROR_INVALID_HANDLE.

Псевдоописатели можно использовать при вызове функций, которым нужен описатель процесса. Так, поток может запросить все временные показатели своего процесса, вызвав *GetProcessTimes*:

```
FILETIME ftCreationTime, ftExitTiroe, ftKernelTiroe, ftUserTime;
GetProcessTimes(GetCurrentProcess(),
    &ftCreationTime, &ftExitTiroe, &ftKernelTiroe, &ftUserTime);
```

Аналогичным образом поток может выяснить собственные временные показатели, вызвав *GetThreadTimes*:

```
FILETIME ftCreationTime, ftExitTime, ftKernelTiroe, ftUserTiroe;
GetThreadTimes(GetCurrentThread(),
    &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

Некоторые Windows-функции позволяют указывать конкретный процесс или поток по его уникальному в рамках всей системы идентификатору. Вот функции, с помощью которых поток может выяснить такой идентификатор — собственный или своего процесса:

```
DWORD GetCurrentProcessId();
DWORD GetCurrentThreadId();
```

По сравнению с функциями, которые возвращают псевдоописатели, эти функции, как правило, не столь полезны, но когда-то и они могут пригодиться.

Преобразование псевдоописателя в настоящий описатель

Иногда бывает нужно выяснить настоящий, а не псевдоописатель потока. Под «настоящим» я подразумеваю описатель, который однозначно идентифицирует уникальный поток. Вдумайтесь в такой фрагмент кода:

```
DW0R0 WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent = GetCurrentThread();
    CreateThread(NULL, 0, ChildThread, (PV0ID) hThreadParent, 0, NULL);
    // далее следует какой-то код...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE) pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTiroe, ftUserTime;
    GetThreadTimes(hThreadParent,
        &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
    // далее следует какой-то код...
}
```

Вы заметили, что здесь не все ладно? Идея была в том, чтобы родительский поток передавал дочернему свой описатель. Но он передает псевдо-, а не настоящий описатель. Начиная выполнение, дочерний поток передает

этот псевдоописатель функции *GetThreadJimes*, и она вследствие этого возвращает временные показатели своего — а вовсе не родительского! — потока. Происходит так потому, что псевдоописатель является описателем текущего потока, т.е. того, который вызывает эту функцию.

Чтобы исправить приведенный выше фрагмент кода, превратим псевдоописатель в настоящий через функцию *DuplicateHandle* (о ней я рассказывал в главе 3):

```
BOOL DuplicateHandle(
    HANDLE hSourceProcess,
    HANDLE hSource,
    HANDLE hTargetProcess,
    PHANDLE phTarget,
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DW0RD dwOptions);
```

Обычно она используется для создания нового «процессо-зависимого» описателя из описателя объекта ядра, значение которого увязано с другим процессом. А мы воспользуемся *DuplicateHandle* не совсем по назначению и скорректируем с ее помощью наш фрагмент кода так:

```
DW0RD WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent;

    DuplicateHandle(
        GetCurrentProcess(),        // описатель процесса.к которому
                                    // относится псевдоописатель потока;
        GetCurrentThread(),         // псевдоописатель родительского потока;
        GetCurrentProcess(),        // описатель процесса, к которому
                                    // относится новый, настоящий описатель потока;
        &hThreadParent,             // даст новый, настоящий описатель,
                                    // идентифицирующий родительский поток;
        0,                          // игнорируется из-за DUPLICATE_SAME_ACCESS;
        FALSE,                      // новый описатель потока ненаследуемый;
        DUPLICATE_SAME_ACCESS);     // новому описателю потока присваиваются
                                    // те же атрибуты защиты, что и псевдоописателю

    CreateThread(NULL, 0, ChildThread, (PV0ID) hThreadParent, 0, NULL);
    // далее следует какой-то код...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE) pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTlme, ftUserTime;
    GetThreadTimes(hThreadParent,
        &ftCreationTlme, &ftExltTlme, &ftKernelTlme, &ftUserTime);
    CloseHandle(hTh readParent);
```

```
    // далее следует какой-то код...
}
```

Теперь родительский поток преобразует свой «двусмысленный» псевдоописатель в настоящий описатель, однозначно определяющий родительский поток, и передает его в *CreateThread*. Когда дочерний поток начинает выполнение, его параметр *pvParam* содержит настоящий описатель потока. В итоге вызов какой-либо функции с этим описателем влияет не на дочерний, а на родительский поток.

Поскольку *DuplicateHandle* увеличивает счетчик пользователей указанного объекта ядра, то, закончив работу с продублированным описателем объекта, очень важно не забыть уменьшить счетчик. Сразу после обращения к *GetThreadTmes* дочерний поток вызывает *CloseHanadle*, уменьшая тем самым счетчик пользователей объекта «родительский поток» на 1. В этом фрагменте кода я исходил из того, что дочерний поток не вызывает других функций с передачей этого описателя. Если же ему надо вызвать какие-то функции с передачей описателя родительского потока, то, естественно, к *CloseHandle* следует обращаться только после того, как необходимость в этом описателе у дочернего потока отпадет.

Надо заметить, что *DuplicateHandle* позволяет преобразовать и псевдоописатель процесса. Вот как это сделать:

```
HANDLE hProcess;
DuplicateHandle(
    GetCurrentProcess(),          // описатель процесса, к которому
                                  // относится псевдоописатель;
    GetCurrentProcess(),          // псевдоописатель процесса;
    GetCurrentProcess(),          // описатель процесса, к которому
                                  // относится новый, настоящий описатель;
    &hProcess,                    // даст новый, настоящий описатель,
                                  // идентифицирующий процесс;
    0,                            // игнорируется из-за DUPLICATE_SAME_ACCESS;
    FALSE,                        // новый описатель процесса ненаследуемый;
    DUPLICATE_SAME_ACCESS);       // новому описателю процесса присваиваются
                                  // те же атрибуты защиты, что и псевдоописателю
```

Оглавление

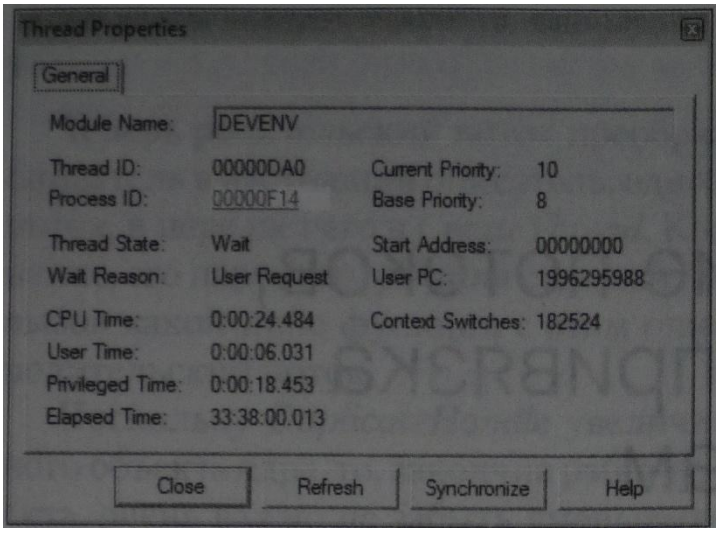
ГЛАВА 7

Планирование потоков, приоритет и привязка к процессорам

Операционная система с вытесняющей многозадачностью должна использовать тот или иной алгоритм, позволяющий ей распределять процессорное время между потоками. Здесь мы рассмотрим алгоритмы, применяемые в Windows Vista.

В главе 6 мы уже обсудили структуру CONTEXT, поддерживаемую в объекте ядра «поток», и выяснили, что она отражает состояние регистров процессора на момент последнего выполнения потока процессором. Каждые 20 мс (или около того, как задано вторым параметром *GetSystemTimeAdjustment*) Windows просматривает все существующие объекты ядра «поток» и отмечает те из них, которые могут получать процессорное время. Далее она выбирает один из таких объектов и загружает в регистры процессора значения из его контекста. Эта операция называется переключением контекста (context switching). По каждому потоку Windows ведет учет того, сколько раз он подключался к процессору. Этот показатель сообщают специальные утилиты вроде Microsoft Spy++. Например, на иллюстрации ниже показан список свойств одного из потоков. Обратите внимание, что этот поток подключался к процессору 182524 раза.

Поток выполняет код и манипулирует данными в адресном пространстве своего процесса. Примерно через 20 мс Windows сохранит значения регистров процессора в контексте потока и приостановит его выполнение. Далее система просмотрит остальные объекты ядра «поток», подлежащие выполнению, выберет один из них, загрузит его контекст в регистры процессора, и все повторится. Этот цикл операций — выбор потока, загрузка его контекста, выполнение и сохранение контекста — начинается с момента запуска системы и продолжается до ее выключения.



Таков вкратце механизм планирования работы множества потоков. Детали мы обсудим позже, но главное я уже показал. Все очень просто, да? Windows потому и называется системой с вытесняющей многозадачностью, что в любой момент может приостановить любой поток и вместо него запустить другой. Как вы еще увидите, этим механизмом можно управлять, правда, крайне ограниченно. Всегда помните: вы не в состоянии гарантировать, что ваш поток будет выполняться непрерывно, что никакой другой поток не получит доступ к процессору и т. д.

> **Примечание.** Меня часто спрашивают, как сделать так, чтобы поток гарантированно запускался в течение определенного времени после какого-нибудь события — например, не позднее чем через миллисекунду после приема данных с последовательного порта? Ответ прост: никак. Такие требования можно предъявлять к операционным системам реального времени, но Windows к ним не относится. Лишь операционная система реального времени имеет полное представление о характеристиках аппаратных Средств, на которых она работает (об интервалах запаздывания контроллеров жестких дисков, клавиатуры и т. д.). А создавая Windows, Майкрософт ставила другую цель: обеспечить поддержку максимально широкого спектра оборудования — различных процессоров, дисковых устройств, сетей и др. Короче говоря, Windows не является операционной системой реального времени. Впрочем, Windows Vista поддерживает некоторые расширения, в некоторых отношениях приближающие ее к ОС реального времени. К таким расширениями относятся службы Thread Ordering (подробнее о ней см. по ссылке http://msdn2.microsoft.com/en-us/library/ms686752.aspx) и Multimedia Class Scheduler для мультимедийных приложений, таких как Windows Media Player 11.

Хочу особо подчеркнуть, что система планирует выполнение только тех потоков, которые могут получать процессорное время, но большинство потоков в системе к таковым не относится. Так, у некоторых объектов-потоков значение счетчика простоев (suspend count) больше 0, а значит, соответствующие потоки приостановлены и не получают процессорное время. Вы може-

те создать приостановленный поток вызовом *CreateProcess* или *CreateThread* с флагом CREATE_SUSPENDED. (В следующем разделе я расскажу и о таких функциях, как *SuspendThread* и *ResumeThread*.)

Кроме приостановленных, существуют и другие потоки, не участвующие в распределении процессорного времени, — они ожидают каких-либо событий. Например, если вы запускаете Notepad и не работаете в нем с текстом, его поток бездействует, а система не выделяет процессорное время тем, кому нечего делать. Но стоит лишь сместить его окно, прокрутить в нем текст или что-то ввести, как система автоматически включит поток Notepad в число планируемых. Это вовсе не означает, что поток Notepad тут же начнет выполняться. Просто система учтет его при планировании потоков и когда-нибудь выделит ему время — по возможности в ближайшем будущем.

Приостановка и возобновление потоков

В объекте ядра «поток» имеется переменная — счетчик числа простоев данного потока. При вызове *CreateProcess* или *CreateThread* он инициализируется значением, равным 1, которое запрещает системе выделять новому потоку процессорное время. Такая схема весьма разумна: сразу после создания поток не готов к выполнению, ему нужно время для инициализации.

После того как поток полностью инициализирован, *CreateProcess* или *CreateThread* проверяет, не передан ли ей флаг CREATE_SUSPENDED, и, если да, возвращает управление, оставив поток в приостановленном состоянии. В ином случае счетчик простоев обнуляется, и поток включается в число планируемых — если только он не ждет какого-то события (например, ввода с клавиатуры).

Создав поток в приостановленном состоянии, Вы можете настроить некоторые его свойства (например, приоритет, о котором мы поговорим позже). Закончив настройку, вы должны разрешить выполнение потока. Для этого вызовите *ResumeThread* и передайте описатель потока, возвращенный функцией *CreateThread* (описатель можно взять и из структуры, на которую указывает параметр *ppiProcInfo,* передаваемый в *CreateProcess*).

```
DWORD ResumeThread(HANDLE hThread);
```

Если вызов ResumeThread прошел успешно, она возвращает предыдущее значение счетчика простоев данного потока; в ином случае — 0xFFFFFFFE

Выполнение отдельного потока можно приостанавливать несколько раз. Если поток приостановлен 3 раза, то и возобновлен он должен быть тоже 3 раза — лишь тогда система выделит ему процессорное время. Выполнение потока можно приостановить не только при его создании с флагом CREATE_ SUSPENDED, но и вызовом *SuspendThread*:

```
DWORD SuspendThread(HANDLE hThread);
```

Любой поток может вызвать эту функцию и приостановить выполнение другого потока (конечно, если его описатель известен). Хоть об этом нигде и не говорится (но я все равно скажу!), приостановить свое выполнение поток способен сам, а возобновить себя без посторонней помощи — нет. Как и *ResumeThread*, функция *SuspendThread* возвращает предыдущее значение счетчика простоев данного потока. Поток можно приостанавливать не более чем MAXIMUM_SUSPEND_COUNT раз (в файле WinNT.h это значение определено как 127). Обратите внимание, что *SuspendThread* в режиме ядра работает асинхронно, но в пользовательском режиме не выполняется, пока поток остается в приостановленном состоянии.

Создавая реальное приложение, будьте осторожны с вызовами *SuspendThread*, так как нельзя заранее сказать, чем будет заниматься его поток в момент приостановки. Например, он пытается выделить память из кучи и поэтому заблокировал к ней доступ. Тогда другим потокам, которым тоже нужна динамическая память, придется ждать его возобновления. *SuspendThread* безопасна только в том случае, когда вы точно знаете, что делает (или может делать) поток, и предусматриваете все меры для исключения вероятных проблем и взаимной блокировки потоков. (О взаимной блокировке и других проблемах синхронизации потоков я расскажу в главах 8, 9 и 10.)

Приостановка и возобновление процессов

В Windows понятия «приостановка» и «возобновление» неприменимы к процессам, так как они не участвуют в распределении процессорного времени. Однако меня не раз спрашивали, как одним махом приостановить все потоки определенного процесса. Это можно сделать из другого процесса, причем он должен быть отладчиком и, в частности, вызывать функции вроде *WaitForDebugEvent* и *ContinueDebugEvent*. Того же можно добиться с помощью команды Suspend Process утилиты Process Explorer от Sysinternals (см. http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/ProcessExplorer.mspx): она приостанавливает все потоки процесса.

Других способов приостановки всех потоков процесса в Windows нет: программа, выполняющая такую операцию, может «потерять» новые потоки. Система должна как-то приостанавливать в этот период не только все существующие, но и вновь создаваемые потоки. Майкрософт предпочла встроить эту функциональность в системный механизм отладки.

Вам, конечно, не удастся написать идеальную функцию *SuspendProcess*, но вполне по силам добиться ее удовлетворительной работы во многих ситуациях. Вот мой вариант функции *SuspendProcess*.

```
VOID SuspendProcess(DWORD dwProcessID, BOOL fSuspend) {

   // получаем список потоков в системе
   HANDLE hSnapsnot = CreateToolhelp32Snapshot(
      TH32CS_SNAPTHREAD, dwProcessID);
```

```
    if (hSnapshot != INVALID_HANDLE_VALUE) {

        // просматриваем список потоков
        THREADENTRY32 te = { sizeof(te) };
        BOOL f0k = Thread32First(hSnapshot, &te);
        for (; f0k; f0k = Thread32Next(hSnapshot, &te)) {

            // относится ли данный поток к нужному процессу?
            if (te.th32OwnerProcessID == dwProcessID) {

                // пытаемся получить описатель потока по его идентификатору
                HANDLE hThread = OpenThread(THREAD_SUSPEND_RESUME,
                    FALSE, te.th32ThreadID);

                if (hThread != NULL) {

                    // приостанавливаем или возобновляем поток
                    if (fSuspend)
                        SuspendThread(hThread);
                    else
                        ResumeThread(hThread);
                }
                CloseHandle(hThread);
            }
        }
        CloseHandle(hSnapshot);
    }
}
```

Для перечисления списка потоков я использую ToolHelp-функции (они рассматривались в главе 4). Определив потоки нужного процесса, я вызываю OpenThread:

```
HANDLE OpenThread(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    DWORD dwThreadID);
```

Это новая функция, которая появилась в Windows 2000. Она находит объект ядра «поток» по идентификатору, указанному в *dwThreadlD*, увеличивает его счетчик пользователей на 1 и возвращает описатель объекта. Получив описатель, я могу передать его в *SuspendThread* (или *ResumeThread*).

Вероятно, вы уже догадались, почему *SuspendProcess* будет срабатывать не во всех случаях: при перечислении могут создаваться новые и уничтожаться существующие потоки. После вызова *CreateToolhelp32Snapshot* в процессе может появиться новый поток, который моя функция уже не увидит, а значит, и не приостановит. Впоследствии, когда я попытаюсь возобновить потоки, вновь вызвав *SuspendProcess*, она возобновит поток, который собст-

венно и не приостанавливался. Но может быть еще хуже: при перечислении текущий поток уничтожается и создается новый с тем же идентификатором. Тогда моя функция приостановит неизвестно какой поток (и даже непонятно в каком процессе).

Конечно, все эти ситуации крайне маловероятны, и, если вы точно представляете, что делает интересующий вас процесс, никаких проблем не будет. В общем, используйте мою функцию на свой страх и риск.

Функция Sleep

Поток может попросить систему не выделять ему процессорное время на определенный период, вызвав:

```
VOID Sleep(DWORD dwMilliseconds);
```

Эта функция приостанавливает поток на *dwMilliseconds* миллисекунд. Отметим несколько важных моментов, связанных с функцией *Sleep.*

- Вызывая *Sleep*, поток добровольно отказывается от остатка выделенного ему кванта времени.
- Система прекращает выделять потоку процессорное время на период, *примерно* равный заданному. Все верно: если вы укажете остановить поток на 100 мс, приблизительно на столько он и «заснет», хотя не исключено, что его сон продлится на несколько секунд или даже минут больше. Вспомните, Windows не является системой реального времени. Ваш поток может возобновиться в заданный момент, но это зависит от того, какая ситуация сложится в системе к тому времени.
- Вы можете вызвать *Sleep* и передать в *dwMilliseconds* значение INFINITE, вообще запретив планировать *поток*. Но это не очень практично — куда лучше корректно завершить поток, освободив его стек и объект ядра.
- Вы можете вызвать *Sleep* и передать в *dwMilliseconds* нулевое значение. Тогда вы откажетесь от остатка своего кванта времени и заставите систему подключить к процессору другой поток. Однако система может снова запустить ваш поток, если других планируемых потоков с тем же приоритетом нет.

Переключение потоков

Функция *SwitchToThread* позволяет подключить к процессору другой поток (если он есть):

```
BOOL SwitchToThread();
```

Когда вы вызываете эту функцию, система проверяет, есть ли поток, которому не хватает процессорного времени. Если нет, *SwitchToThread* немедленно возвращает управление, а если да, планировщик отдает ему дополнительный квант времени (приоритет этого потока может быть ниже, чем

у вызывающего). По истечении этого кванта планировщик возвращается в обычный режим работы.

SwitchToThread позволяет потоку, которому не хватает процессорного времени, отнять этот ресурс у потока с более низким приоритетом. Она возвращает FALSE, если на момент ее вызова в системе нет ни одного потока, готового к исполнению; в ином случае — ненулевое значение.

Вызов *SwitchToThread* аналогичен вызову *Sleep* с передачей в *dwMilliseconds* нулевого значения. Разница лишь в том, что *SwitchToThread* дает возможность выполнять потоки с более низким приоритетом, которым не хватает процессорного времени, а *Sleep* действует без оглядки на «голодающие» потоки.

Переключение потоков на компьютерах с процессором, поддерживающим HyperThreading

Чипы процессоров Xeon, некоторых Pentium 4 и более поздних моделей содержат несколько «логических процессоров», каждый из которых способен исполнять потоки (такие процессоры поддерживают технологию HyperThreading, HT). У каждого потока имеется отдельное состояние (набор регистров процессора), но основные ресурсы, такие как кэш процессора, все потоки используют сообща. Когда исполнение одного из потоков приостанавливается, процессор автоматически переходит к исполнению следующего потока без участия операционной системы. Такие паузы возникают при промахах кэша, ошибках в предсказании ветвления, ожидании результатов предыдущей команды и в некоторых других обстоятельствах. По утверждению Intel, HT-процессоры работают на 10-30% быстрее, в зависимости от приложения и алгоритма работы с памятью. Подробнее о HT-процессорах см. по ссылке http://www.micmsoft.com/whdc/system/CEC/HT-Windows.mspx.

При исполнении циклов на HT-процессорах необходимо принудительно приостанавливать исполнение текущего потока, чтобы дать доступ к процессору и другим потокам. Процессор типа *x*86 поддерживает инструкцию ассемблера PAUSE. Эта инструкция предотвращает нарушение порядка доступа к памяти, повышает производительность системы и снижает энергопотребление. В *x*86 эквивалентом PAUSE являются инструкция REP NOP, обеспечивающая совместимость с прежними моделями процессоров с архитектурой IA-32, не поддерживающими HyperThreading. PAUSE заставляет процессор сделать паузу конечной длительности (на некоторых моделях это пауза нулевой длины). В Win32 API инструкцию PAUSE генерирует вызов макроса YieldProcessor, объявленного в WinNT.h. Этот макрос позволяет разработчикам писать код, независимый от архитектуры процессора, а также позволяет избежать лишних вызовов функций.

Определение периодов выполнения потока

Иногда нужно знать, сколько времени затрачивает поток на выполнение той или иной операции. Многие в таких случаях, используя новую функцию GetTtckCount64, пишут что-то вроде этого:

```
// получаем стартовое время
ULONGLONG qwStartTlne = GetTickCount64();

// здесь выполняем какой-нибудь сложный алгоритм

// вычитаем стартовое время из текущего
ULONGLONG qwElapsedTime = GetTickCount64() - qwStartTime;
```

Этот код основан на простом допущении, что он не будет прерван. Но в операционной системе с вытесняющей многозадачностью никто не знает, когда поток получит процессорное время, и результат будет сильно искажен. Что нам здесь нужно, так это функция, которая сообщает время, затраченное процессором на обработку данного потока. К счастью, в Windows есть такая функция:

```
BOOL GetThreadTimes(
    HANDLE hThread,
    PFILETIME pftCreationTime,
    PFILETIME pftExitTime,
    PFILETIME pftKernelTime,
    PFILETIME pftUserTime);
```

GetThreadTimes возвращает четыре временных параметра (см. табл. 7-1).

Табл. 7-1. Значения, возвращаемые GeetThreadTimes

Показатель времени	Описание
Время создания (creation time)	Абсолютная величина, выраженная в интервалах по 100 не. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента создания потока
Время завершения (exit time)	Абсолютная величина, выраженная в интервалах по 100 не. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента завершения потока. Если поток все еще выполняется, этот показатель имеет неопределенное значение
Время выполнения ядра (kernel time)	Относительная величина, выраженная в интервалах по 100 не. Сообщает время, затраченное этим потоком на выполнение кода операционной системы
Время выполнения User (User time)	Относительная величина, выраженная в интервалах по 100 не. Сообщает время, затраченное потоком на выполнение кода приложения

С помощью этой функции можно определить время, необходимое для выполнения сложного алгоритма:

```
__int64 FileTimeToQuadWord (PFILETIME pft) {
    return(Int64ShllMod32(pft->dwHighDateTime, 32) | pft->dwLowDateTime);
}

void PerformLongOperation () {
    FILETIME ftKernelTimeStart, ftKernelTimeEnd;
    FILETIME ftUserTimeStart,   ftUserTimeEnd;
    FILETIME ftDummy;
    __int64 qwKernelTimeElapsed, qwUserTimeElapsed,
        qwTotalTimeElapsed;

    // получаем начальные показатели времени
    GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
        &ftKernelTimeStart, &ftUserTimeStart);

    // здесь выполняем сложный алгоритм

    // получаем конечные показатели времени
    GetThreadTimes(GetCurrentThread(), &ftDummy, &ftDummy,
        &ftKernelTimeEnd, &ftUserTimeEnd);

    // Получаем значения времени, затраченного на выполнение ядра и
    // User, преобразуя начальные и конечные показатели времени из
    // FILETIME в учетверенные слова, вычитая затем начальные
    // показатели из конечных.
    qwKernelTimeElapsed = FileTimeToQuadWord(&ftKernelTimeEnd) -
        FileTimeToQuadWord(&ftKernelTimeStart);

    qwUserTimeElapsed = FileTimeToQuadWord(&ftUserTimeEnd) -
        FileTimeToQuadWord(&ftUserTimeStart);

    // получаем общее время, складывая время выполнения ядра и User
    qwTotalTimeElapsed = qwKernelTimeElapsed + qwUserTimeElapsed;

    // общее время хранится в qwTotalTimeElapsed
}
```

Заметим, что существует еще одна функция, аналогичная *GetThreadTimes* и применимая ко всем потокам в процессе:

```
BOOL GetProcessTimes(
    HANDLE hProcess,
    PFILETIME pftCreationTime,
    PFILETIME pftExitTime,
    PFILETIME pftKernelTime,
    PFILETIME pftUserTime);
```

GetProcessTimes возвращает временные параметры, суммированные по всем потокам (даже уже завершенным) в указанном процессе. Так, время выполнения ядра будет суммой периодов времени, затраченного всеми потоками процесса на выполнение кода операционной системы.

В Windows Vista изменился способ подсчета времени процессора, использованного потоком. Теперь вместо таймера, отсчитывающего время интервалами по 10-15 мс (о нем и утилите ClockRes см. http://www.microsoft.com/technet/sysinternals/information/higresolutiontimers.mspx операционная система используется 64-разрядный аппаратный механизм *Time Stamp Counter* (TSC). TSC поддерживается процессором и отсчитывает число тактов с момента включения компьютера. Несложно представить, насколько этот механизм, работающий в системе с процессором, таковая частота которого измеряется в МГц, точнее таймера, «цена деления» которого — несколько миллисекунд.

Когда планировщик останавливает исполнение потока, вычисляется разность между текущим значением TSC и его значением на момент начала кванта, результат прибавляется к общему времени исполнения потока. При этом, в отличие от предыдущих версий Windows, пауза не учитывается. Функции *QueryThreadCycleTime* и *QueryProcessCycleTime* возвращают число тактов, использованных заданным потоком или всеми потоками данного процесса, соответственно. Если вы хотите заменить *GetTickCount64* более точным «хронометром», можно получать текущие значения TSC вызовом *ReadTimeStampCounter* макроса, объявленного в WinNT.h, использующего внутреннюю функцию *__rdtsc* компилятора C++.

GetThreadTimes не годится для высокоточного измерения временных интервалов — для этого в Windows предусмотрено две специальные функции:

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER* pliFrequency);

BOOL OueryPerfomanceCounter(LARGE_INTEGER* pliCount);
```

Они построены на том допущении, что поток не вытесняется, поскольку высокоточные измерения проводятся, как правило, в очень быстро выполняемых блоках кода. Чтобы слегка упростить работу с этими функциями, я создал следующий C++-класс:

```
class CStopwatch {
public:
    CStopwatch() { QueryPerfor»anc6Frequency(&m_liPerfFreq); Start(); }

    void Start() { QueryPerformanceCounter(&m_liPerfStart); }

    __int64 Now() const {    // возвращает число миллисекунд после вызова Start
        LARGE_INTEQER liPerfNow;
        QueryPerformanceCounter (&liPerfNow);
        return(((liPerfNow.QuadPart - m_liPerfStart.QuadPart) * 1000)
```

```
        / m_liParfFreq.QuadPart);
    }

    __int64 NowInMicro() const {           // возвращает число миллисекунд
                                           // после вызова Start
        LARGE_INTEGER liPerfNow;
        QueryPerfornanceCounter(&liPerfNow);
        return(((liPerfNow.QuadPart - m_liPerfStart.QuadPart) * 1000000)
            / m_liPerfFreq.QuadPart);
        }

private:
    LARGE_INTEGER m_liPerfFreq;            // количество отсчетов в секунду
    LARGE_INTEGER m_liPerfStart;   // начальный отсчет
};
```

Я применяю этот класс так:

```
// создаю секундомер (начинающий отсчет с текущего момента времени)
CStopwatch stopwatch;

// здесь я помещаю код, время выполнения которого нужно измерить

// определяю, сколько времени прошло
__int64 qwElapsedTime = stopwatch.Now();

// qwElapsedTime сообщает длительность выполнения в миллисекундах
```

Функции для работы с высокоточным таймером удобны для преобразования значений, возвращаемых новыми *Get*CycleTime*-функциями. Поскольку длина такта (а значит, и результаты измерения) зависят от тактовой частоты процессора, для преобразования числа тактов в более информативную величину необходимо узнать тактовую частоту. Так, 2-ГГц процессор совершает 2 млрд. тактов в секунду, следовательно, 800 000 тактов этого процессора будут соответствовать 0,4 мс, тогда как на более медленном 1-ГГц процессоре то же число тактов соответствует 0,8 мс. Вот реализация функции *GetCPUFrequencylnMHz*:

```
DWORD GetCPUFrequencylnMHz() {
    // Изменяем приоритет потока, чтобы у него было больше шансов на
    // на подключение к процессору по окончании Sleep().
    int currentPriority = GetThreadPriority(GetCurrentThread());
    SetThreadPriority(GetCurrentThread(), THREAD_PRIORITY_HIGHEST);

    // отсчет времени с помощью другого таймера
    __int64 elapsedTime = 0;

    // создаем секундомер (по умолчанию он отсчитывает время, начиная
        с текущего момента).
```

```
    CStopwatch stopwatch;
    __int64 perfCountStart = stopwatch.NowInMicro();

    // получаем текущее число тактов
    unsigned __int64 cyclesOnStart = ReadTimeStampCounter();

    // ожидаем в течение 1 с
    Sleep(1000);

    // получаем число тактов по истечении 1 с
    unsigned __int64 numberOfCycles = ReadTimeStampCounter() - cyclesOnStart;

    // измеряем прошедшее время более точно
    elapsedTime = stopwatch.NowInMicro() - perfCountStart;

    // восстанавливаем приоритет потоков
    SetThreadPriority(GetCurrentThread(), currentPriority);

    // вычисляем частоту процессора в МГц
    DWORD dwCPUFrequency = (DWORD)(numberOfCycles / elapsedTime);
    return(dwCPUFrequency);
}
```

Казалось бы, чтобы получить время в миллисекундах, соответствующее числу тактов, которое вернула функция QueryProcessCycleTime, достаточно разделить это число на значение, возвращаемое, и умножить частное на 1000. Однако эти простые вычисления могут давать ошибочные результаты. Дело в том, что тактовая частота процессора непостоянна, она варьируется в зависимости от настроек, сделанных пользователем, и от источника питания (сеть ли это или, в случае ноутбука, встроенный аккумулятор). Наконец, на многопроцессорном компьютере поток может подключаться к разным процессором, тактовая частота которых может несколько отличаться.

Структура CONTEXT

К этому моменту вы должны понимать, какую важную роль играет структура CONTEXT в планировании потоков. Система сохраняет в ней состояние потока перед самым отключением его от процессора, благодаря чему его выполнение возобновляется с того места, где было прервано.

Вы, Наверное, удивитесь, но в документации Platform SDK структуре CONTEXT отведен буквально один абзац:

«В структуре CONTEXT хранятся данные о состоянии регистров с учетом специфики конкретного процессора. Она используется системой для выполнения различных внутренних операций? Соответствующие определения см. в заголовочном файле WinNT.h.»

В документации нет ни слова об элементах этой структуры, набор которых зависит от lima процессора. Фактически CONTEXT — единственная из всех структур Windows, специфичная для конкретного процессора.

Так из чего же состоит структура CONTEXT? Давайте посмотрим. Ее элементы четко соответствуют регистрам процессора. Например, для процессоров x86 в число элементов входят *Eax*, *Ebx*, *Ecx*, *Edx* и т. д. Структура CONTEXT для процессоров *x*86 выглядит так:

```
typedef struct _CONTEXT {

    //
    // Флаги, управляющие содержимым записи CONTEXT.
    //
    // Если запись контекста используется как входной параметр, тогда раздел,
    // управляемый флагом (когда он установлен), считается содержащим
    // действительные значения. Если запись контекста используется для
    // модификации контекста потока, то изменяются только те разделы, для
    // которых флаг установлен.
    //
    // Если запись контекста используется как входной и выходной параметр
    // для захвата контекста потока, возвращаются только те разделы контекста,
    // для которых установлены соответствующие флаги. Запись контекста никогда
    // не используется только как выходной параметр.
    //

    DWORD ContextFlags;

    //
    // Этот раздел определяется/возвращается, когда в ContextFlags установлен
    // флаг CONTEXT_DEBUG_REGISTERS. Заметьте, что CONTEXT_DEBUG_REGISTERS
    // не включаются в CONTEXT_FULL.
    //

    DWORD                                   Dr0;
    DWORD                                   Dr1;
    DWORD                                   Dr2;
    DWORD                                   Dr3;
    DWORD                                   Dr6;
    DWORD                                   Dr7;

    //
    // Этот раздел определяется/возвращается, когда в ContextFlags
    // установлен флаг CONTEXT_FLOATING_POINT.
    //

    FLOATING_SAVE_AREA FloatSave;
```

```
    //
    // Этот раздел определяется/возвращается, когда в ContextFlags
    // установлен флаг CONTEXT_SEGMENTS.
    //

    DWORD SegGs;
    DWORD SegFs;
    DWORD SegEs;
    DWORD SegDs;

    //
    // Этот раздел определяется/возвращается, когда в ContextFlags
    // установлен флаг CONTEXT_INTEGER.
    //

    DWORD Edl;
    DWORD Esi;
    DWORD Ebx;
    DWORD Edx;
    DWORD Ecx;
    DWORD Eax;

    //
    // Этот раздел определяется/возвращается, когда в ContextFlags
    // установлен флаг CONTEXT_CONTROL.
    //

    DWORD Ebp;
    DWORD Eip;
    DWORD SegCs;                          // следует очистить
    DWORD EFlags;                         // следует очистить
    DWORD Esp;
    DWORD SegSs;

    //
    // Этот раздел определяется/возвращается, когда в ContextFlags
    // установлен флаг COMTEXT_EXTENDED_REGISTERS.
    // Формат и смысл значений зависят от типа процессора.
    //

    BYTE  ExtendedRegistere[MAXIMUM_SUPPORTED_EXTENSION;]

} CONTEXT;
```

Эта структура разбита на несколько разделов. Раздел CONTEXT_ CONTROL содержит управляющие регистры процессора: указатель команд, указатель стека, флаги и адрес возврата функции. Раздел CONTEXT_

INTEGER соответствует целочисленным регистрам процессора, CONTEXT_FLOATING_POINT — регистрам с плавающей точкой, CONTEXT_SEGMENTS — сегментным регистрам, CONTEXT_DEBUG_REGISTERS — регистрам, предназначенным для отладки, а CONTEXT_EXTENDED_REGISTERS — дополнительным регистрам.

Windows фактически позволяет заглянуть внутрь объекта ядра «поток» и получить сведения о текущем состоянии регистров процессора. Для этого предназначена функция:

```
BOOL GetThreadContext(
    HANDLE hThread,
    PCONTEXT pContext);
```

Чтобы вызвать ее, создайте экземпляр структуры CONTEXT, инициализируйте нужные флаги (в элементе ContextFlags) и передайте функции GetThreadContext адрес этой структуры. Функция поместит значения в элементы, сведения о которых вы запросили.

Прежде чем обращаться к GetThreadContext, приостановите поток вызовом SuspendThread, иначе поток может быть подключен к процессору, и значения регистров существенно изменятся. На самом деле у потока есть два контекста: пользовательского режима и режима ядра. GetThreadContext возвращает лишь первый из них. Если вы вызываете SuspendThread, когда поток выполняет код операционной системы, пользовательский контекст можно считать достоверным, даже несмотря на то что поток еще не остановлен (он все равно не выполнит ни одной команды пользовательского кода до последующего возобновления).

Единственный элемент структуры CONTEXT, которому не соответствует какой-либо регистр процессора, — ContextFlags. Присутствуя во всех вариантах этой структуры независимо от типа процессора, он подсказывает функции GetThreadContext, значения каких регистров вы хотите узнать. Например, чтобы получить значения управляющих регистров для потока, напишите что-то вроде:

```
// создаем экземпляр структуры CONTEXT
CONTEXT Context;

// Сообщаем системе, что нас интересуют сведения
// только об управляющих регистрах.
Context.ContextFlags = CONTEXT_CONTROL;

// Требуем от системы информацию
// о состоянии регистров процессора для данного потока.
GetThreadContext(hThread, AContext);

// Действительные значения содержат элементы структуры CONTEXT,
// соответствующие управляющим регистрам, остальные значения
// не определены.
```

Перед вызовом *GetThreadContext* надо инициализировать элемент ContextFlags. Чтобы получить значения как управляющих, так и целочисленных регистров, инициализируйте его так:

```
// Сообщаем системе, что нас интересуют
// управляющие и целочисленные регистры.
Context.ContextFlags = CONTEXT_CONTROL | CONTEXT_INTEGER;
```

Есть еще один идентификатор, позволяющий узнать значения важнейших регистров (т. е. используемых, по мнению Майкрософт, чаще всего):

```
// Сообщаем системе, что нас интересуют
// управляющие и целочисленные регистры.
Context.ContextFlags = CONTEXT_FULL;
```

CONTEXT_FULL определен в файле WinNT.h, как CONTEXT_CONTROL CONTEXT_INTEGER | CONTEXT_SEGMENTS.

После возврата из *GetThreadContext* вы легко проверите значения любых регистров для потока, но помните, что такой код зависит от типа процессора. Так, у процессоров типа x86 в поле Eip хранится указатель на сегмент команд, а в поле *Esp* — указатель на стек.

Даже удивительно, какой мощный инструмент дает Windows в руки разработчика! Но есть вещь, от которой вы придете в полный восторг: значения элементов CONTEXT можно изменять и передавать объекту ядра «поток» с помощью функции *SetThreadContext*.

```
BOOL SetThreadContext(
    HANDLE hThread,
    CONST CONTEXT *pContext);
```

Перед этой операцией поток тоже нужно приостановить, иначе результаты могут быть непредсказуемыми.

Прежде чем обращаться к *SetThreadContext*, инициализируйте элемент *ContextFlags*, как показано ниже.

```
CONTEXT Context;

// приостанавливаем поток
SuspendThread read(hThread);

// получаем регистры для контекста потока
Context.ContextFlags = C0NTEXT_C0NTROL;
GetThreadContext(hThread, &Context);

// устанавливаем указатель команд по своему выбору;
// в нашем примере присваиваем значение 0x00010000
Context.Eip = 0x00010000;

// вносим изменения в регистры потока; ContextFlags
// можно и не инициализировать, так как это уже сделано
```

```
Context.ContextFlags = CONTEXT_CONTROL;
SetThreadContext(hThread, &Context);

// возобновляем выполнение потока;
// оно начнется с адреса 0x00010000
ResumeThread(hThread);
```

Этот код, вероятно, приведет к ошибке защиты (нарушению доступа) в удаленном потоке; система сообщит о необработанном исключении, и удаленный процесс будет закрыт. Все верно — не ваш, а удаленный. Вы благополучно обрушили другой процесс, оставив свой в целости и сохранности!

Функции *GetThreadContext* и *SetThreadContext* наделяют вас огромной властью над потоками, но пользоваться ею нужно с осторожностью. Вызывают их лишь считанные приложения. Эти функции предназначены для отладчиков и других инструментальных средств, хотя обращаться к ним можно из любых программ.

Подробнее о структуре CONTEXT мы поговорим в главе 24.

Приоритеты потоков

В начале главы я сказал, что поток получает доступ к процессору на 20 мс, после чего планировщик переключает процессор на выполнение другого потока. Так происходит, только если у всех потоков один приоритет, но на самом деле в системе существуют потоки с разными приоритетами, а это меняет порядок распределения процессорного времени.

Каждому потоку присваивается уровень приоритета — от 0 (самый низкий) до 31 (самый высокий). Решая, какому потоку выделить процессорное время, система сначала рассматривает только потоки с приоритетом 31 и подключает их к процессору по принципу карусели. Если поток с приоритетом 31 не исключен из планирования, он немедленно получает квант времени, по истечении которого система проверяет, есть ли еще один такой поток. Если да, он тоже получает свой квант процессорного времени.

Пока в системе имеются планируемые потоки с приоритетом 31, ни один поток с более низким приоритетом процессорного времени не получает. Такая ситуация называется «голоданием» (starvation). Она наблюдается, когда потоки с более высоким приоритетом так интенсивно используют процессорное время, что остальные практически не работают. Вероятность этой ситуации намного ниже в многопроцессорных системах, где потоки с приоритетами 31 и 30 могут выполняться одновременно. Система всегда старается, чтобы процессоры были загружены работой, и они простаивают только в отсутствие планируемых потоков.

На первый взгляд, в системе, организованной таким образом, у потоков с низким приоритетом нет ни единого шанса на исполнение. Но, как я уже говорил, зачастую потоки как раз и не нужно выполнять. Например, если первичный поток вашего процесса вызывает *GetMessage*, а система видит,

что никаких сообщений пока нет, она приостанавливает его выполнение, отнимает остаток неиспользованного времени и тут же подключает к процессору другой ожидающий поток. И пока в системе не появятся сообщения для потока вашего процесса, он будет простаивать — система не станет тратить на него процессорное время. Но вот в очереди этого потока появляется сообщение, и система сразу же подключает его к процессору (если только в этот момент не выполняется поток с более высоким приоритетом).

А теперь обратите внимание на еще один момент. Потоки с более высоким приоритетом всегда вытесняют потоки с более низким приоритетом независимо от того, исполняются последние или нет. Допустим, процессор исполняет поток с приоритетом 5, и тут система обнаруживает, что поток с более высоким приоритетом готов к выполнению. Что будет? Система остановит поток с более низким приоритетом — даже если не истек отведенный ему квант процессорного времени — и подключит к процессору поток с более высоким приоритетом (и, между прочим, выдаст ему полный квант времени).

Кстати, при загрузке системы создается особый поток — *поток обнуления страниц* (zero page thread), которому присваивается нулевой уровень приоритета. Ни один поток, кроме этого, не может иметь нулевой уровень приоритета. Он обнуляет свободные страницы в оперативной памяти при отсутствии других потоков, требующих внимания со стороны системы.

Абстрагирование приоритетов

Создавая планировщик потоков, разработчики из Майкрософт прекрасно понимали, что он не подойдет на все случаи жизни. Они также осознавали, что со временем «назначение» компьютера может измениться. Например, в момент выпуска Windows NT создание приложений с поддержкой OLE еще только начиналось. Теперь такие приложения — обычное дело. Кроме того, значительно расширилось применение игрового программного обеспечения, ну и, конечно же, Интернета.

Алгоритм планирования потоков существенно влияет на выполнение приложений. С самого начала разработчики Майкрософт понимали, что его придется изменять по мере того, как будут расширяться сферы применения компьютеров. Майкрософт гарантирует, что наши программы будут работать и в следующих версиях Windows. Как же ей удается изменять внутреннее устройство системы, не нарушая работоспособность наших программ? Дело в том, что:

- планировщик документируется не полностью;
- Майкрософт не разрешает в полной мере использовать все особенности планировщика;
- Майкрософт предупреждает, что алгоритм работы планировщика постоянно меняется, и не рекомендует писать программы в расчете на текущий алгоритм.

Windows API предоставляет слой абстрагирования от конкретного алгоритма работы планировщика, запрещая прямое обращение к планировщику. Вместо этого вы вызываете функции Windows, которые «интерпретируют» ваши параметры в зависимости от версии системы. Я буду рассказывать именно об этом слое абстрагирования.

Проектируя свое приложение, вы должны учитывать возможность параллельного выполнения других программ. Следовательно, вы обязаны выбирать класс приоритета, исходя из того, насколько «отзывчивой» должна быть ваша программа. Согласен, такая формулировка довольно туманна, но так и задумано: Майкрософт не желает обещать ничего такого, что могло бы нарушить работу вашего кода в будущем.

Windows поддерживает шесть классов приоритета: idle (простаивающий), below normal (ниже обычного), normal (обычный), above normal (выше обычного), high (высокий) и realtime (реального времени). Самый распространенный класс приоритета, естественно, — normal; его использует 99% приложений. Классы приоритета показаны в следующей таблице.

Приоритет idle идеален для программ, выполняемых, только когда системе больше нечего делать. Примеры таких программ — экранные заставки и средства мониторинга. Компьютер, не используемый в интерактивном режиме, может быть занят другими задачами (действуя, скажем, в качестве файлового сервера), и их потокам незачем конкурировать с экранной заставкой за доступ к процессору. Средства мониторинга, собирающие статистическую информацию о системе, тоже не должны мешать выполнению более важных задач.

Класс приоритета high следует использовать лишь при крайней необходимости. Классом приоритета real-time почти никогда не стоит пользоваться. На самом деле в ранних бета-версиях Windows NT 3.1 присвоение этого класса приоритета приложениям даже не предусматривалось, хотя операционная система поддерживала эту возможность. Real-time — чрезвычайно высокий приоритет, и, поскольку большинство потоков в системе (включая управляющие самой системой) имеет более низкий приоритет, процесс с таким классом окажет на них сильное влияние. Так, потоки реального времени могут заблокировать необходимые операции дискового и сетевого ввода-вывода и привести к несвоевременной обработке ввода от мыши и клавиатуры — пользователь может подумать, что система зависла. У вас должна быть очень веская причина для применения класса real-time — например, программе требуется реагировать на события в аппаратных средствах с минимальной задержкой или выполнять быстротечную операцию, которую нельзя прерывать ни при каких обстоятельствах.

Табл. 7-2. Классы приоритета процессов

Класс приоритета	Описание
Real-time	Потоки в этом процессе обязаны немедленно реагировать на события, обеспечивая выполнение критических по времени задач. Такие потоки вытесняют даже компоненты операционной системы. Будьте крайне осторожны с этим классом
High	Потоки в этом процессе тоже должны немедленно реагировать на события, обеспечивая выполнение критических по времени задач. Этот класс присвоен, например, Task Manager, что дает возможность пользователю закрывать больше неконтролируемые процессы
Above normal	Класс приоритета, промежуточный между normal и high. Это новый класс, введенный в Windows 2000
Normal	Потоки в этом процессе не предъявляют особых требований к выделению им процессорного времени
Below normal	Класс приоритета, промежуточный между normal и idle. Это новый класс, введенный в Windows 2000
Idle	Потоки в этом процессе выполняются, когда система не занята другой работой. Этот класс приоритета обычно используется для утилит, работающих в фоновом режиме, экранных заставок и приложений, собирающих статистическую информацию

Примечание. Процесс с классом приоритета real-time нельзя запустить, если пользователь не имеет привилегии Increase Scheduling Priority. По умолчанию такой привилегией обладает администратор и опытный пользователь.

Конечно, большинство процессов имеет обычный класс приоритета. В Windows 2000 появилось два новых промежуточных класса — below normal и above normal. Майкрософт добавила их, поскольку некоторые компании жаловались, что существующий набор классов приоритетов не дает должной гибкости.

Выбрав класс приоритета, забудьте о том, как ваша программа будет выполняться совместно с другими приложениями, и сосредоточьтесь на ее потоках. Windows поддерживает семь относительных приоритетов потоков: idle (Простаивающий), lowest (низший), below normal (ниже обычного), normal (обычный), above normal (выше обычного), highest (высший) и time-critical (критичный по времени). Эти приоритеты относительны классу приоритета процесса. Как обычно, большинство потоков использует обычный приоритет. Относительные приоритеты потоков описаны в таблице 7-3.

Итак, вы присваиваете процессу некий класс приоритета и можете изменять относительные приоритеты потоков в пределах процесса. Заметьте, что я не сказал ни слова об уровнях приоритетов 0-31. Разработчики прило-

жений не имеют с ними дела. Уровень приоритета формируется самой системой, исходя из класса приоритета процесса и относительного приоритета потока. А механизм его формирования - как раз то, чем Майкрософт не хочет себя ограничивать. И действительно, этот механизм меняется практически в каждой версии системы.

Табл. 7-3. Приоритеты потоков

Относительный приоритет потока	Описание
Time-critical	Поток выполняется с приоритетом 31 в классе real-time и с приоритетом 15 в других классах
Highest	Поток выполняется с приоритетом на два уровня выше обычного для данного класса
Above normal	Поток выполняется с приоритетом на один уровень выше обычного для данного класса
Normal	Поток выполняется с обычным приоритетом процесса для данного класса
Below normal	Поток выполняется с приоритетом на один уровень ниже обычного для данного класса
Lowest	Поток выполняется с приоритетом на два уровня ниже обычного для данного класса
Idle	Поток выполняется с приоритетом 16 в классе real-time и с приоритетом 1 в других классах

В таблице 7-4 показано, как формируется уровень приоритета в Windows Vista, но не забывайте, что в Windows NT и тем более в Windows 95/98 этот механизм действует несколько иначе. Учтите также, что в будущих версиях Windows он вновь изменится.

Например, обычный поток в обычном процессе получает уровень приоритета 8. Поскольку большинство процессов имеет класс normal, а большинство потоков — относительный приоритет normal, у основной части потоков в системе уровень приоритета равен 8.

Обычный поток в процессе с классом приоритета high получает уровень приоритета 13. Изменив класс приоритета процесса на idle, вы снизите уровень приоритета того же потока до 4. Вспомните, что приоритет потока всегда относителен классу приоритета его процесса. Изменение класса приоритета процесса не влияет на относительные приоритеты его потоков, но сказывается на уровне их приоритета.

Табл. 7-4. Формирование уровней процесса

Относительный приоритет потока	Класс приоритета процесса					
	Idle	Below normal	Normal	Above normal	High	Real-time
Time-critical (критичный по времени)	15	15	15	15	15	31
Highest (высший)	6	8	10	12	15	26
Above normal (выше обычного)	5	7	9	11	14	25
Normal (обычный)	4	6	8	10	13	24
Below normal (ниже обычного)	3	5	7	9	12	23
Lowest (низший)	2	4	6	8	11	22
Idle (простаивающий)	1	1	1	1	1	16

Обратите внимание, что в таблице не показано, как задать уровень приоритета 0. Это связано с тем, что нулевой приоритет зарезервирован для потока обнуления страниц, и никакой другой поток не может иметь такой приоритет. Кроме того, уровни 17-21 и 27-30 в обычном приложении тоже недоступны. Вы можете пользоваться ими, только если пишете драйвер устройства, работающий в режиме ядра. И еще одно: уровень приоритета потока в процессе с классом real-time не может опускаться ниже 16, а потока в процессе с любым другим классом — подниматься выше 15.

Примечание. Концепция класса приоритета вводит некоторых в заблуждение. Они делают отсюда вывод, будто процессы участвуют в распределении процессорного времени. Так вот, процессы никогда не получают процессорное время — оно выделяется лишь потокам. Класс приоритета процесса — сугубо абстрактная концепция, введенная Майкрософт с единственной целью: скрыть от разработчика внутреннее устройство планировщика.

Примечание. В общем случае поток с высоким уровнем приоритета должен быть активен как можно меньше времени. При появлении у него какой-либо работы он тут же получает процессорное время. Выполнив минимальное количество команд, он должен снова вернуться в ждущий режим. С другой стороны, поток с низким уровнем приоритета может оставаться активным и занимать процессор довольно долго. Следуя этим правилам, вы сохраните должную отзывчивость операционной системы на действия пользователя.

Программирование приоритетов

Так как же процесс получает класс приоритета? Очень просто. Вызывая *CreateProcess***,** вы можете указать в ее параметре *fdwCreate* нужный класс приоритета. Идентификаторы этих классов приведены в таблице 7-5.

Табл. 7-5. Классы приоритета процессов

Класс приоритета	Идентификатор
Real-time	REALTIME_PRIORITY_CLASS
High	HIGH_PRIORITY_CLASS
Above normal	ABOVENORMALPRIORITYCLASS
Normal	NORMAL_PRIORITY_CLASS
Below normal	BELOW_NORMAL_PRIORITY_CLASS
Idle	IDLE_PRIORITY_CLASS

Вам может показаться странным, что, создавая дочерний процесс, родительский сам устанавливает ему класс приоритета. За примером далеко ходить не надо — возьмем все тот же Explorer. При запуске из него какого-нибудь приложения новый процесс создается с обычным приоритетом. Но Explorer ведь не знает, что делает этот процесс и как часто его потокам надо выделять процессорное время. Поэтому в системе предусмотрена возможность изменения класса приоритета самим выполняемым процессом — вызовом функции *SetPriorityClass*:

```
BOOL SetPriorityClass(
    HANDLE hProcess,
    DWORD fdwPriority);
```

Эта функция меняет класс приоритета процесса, определяемого описателем *hProcess*, в соответствии со значением параметра *fdwPriority*. Последний должен содержать одно из значений, указанных в таблице выше. Поскольку *SetPriorityClass* принимает описатель процесса, вы можете изменить приоритет любого процесса, выполняемого в системе, — если его описатель известен и у вас есть соответствующие права доступа.

Обычно процесс пытается изменить свой класс приоритета. Вот как процесс может сам себе установить класс приоритета idle:

```
BOOL SetPriorityClass( GetCurrentProcess(), IDLE_PRI0RITY_CLA8S);
```

Парная ей функция *GetPriorityClass* позволяет узнать класс приоритета любого процесса:

```
DWORD GetPriorityClass(HANDLE hProcess);
```

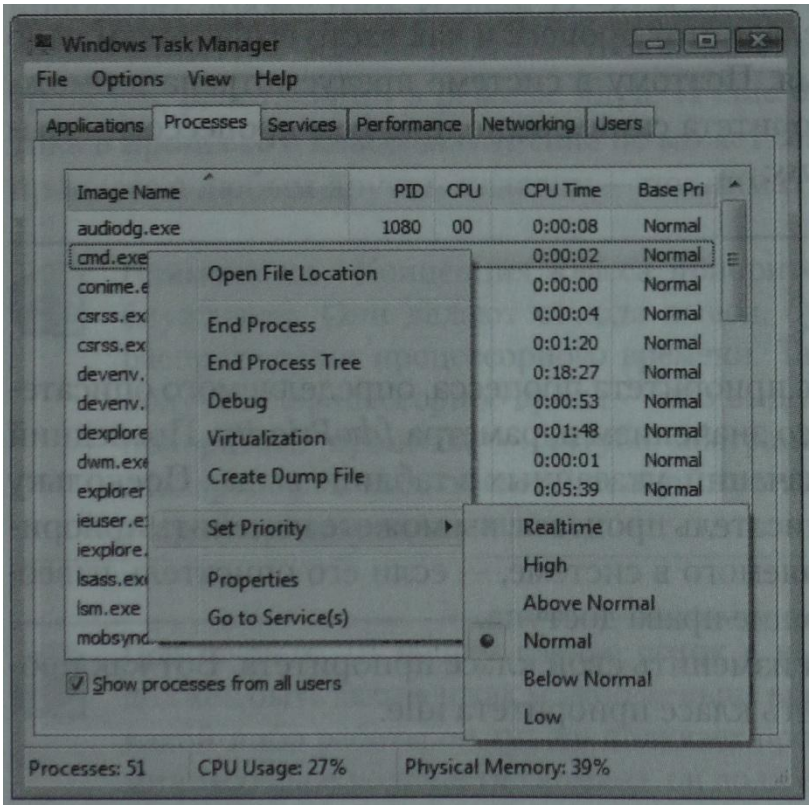
Она возвращает, как вы догадываетесь, один из ранее перечисленных флагов.

При запуске из оболочки командного процессора начальный приоритет программы тоже обычный. Однако, запуская ее командой Start, можно указать ключ, определяющий начальный приоритет. Так, следующая команда, введенная в оболочке командного процессора, заставит систему запустить приложение Calculator и присвоить ему приоритет idle:

```
C:\>START /LOW CALC.EXE
```

Команда Start допускает также ключи /BELOWNORMAL, /NORMAL, /ABOVENORMAL, /HIGH и /REALTIME, позволяющие начать выполнение программы с соответствующим классом приоритета. Разумеется, после запуска программа может вызвать *SetPriorityClass* и установить себе другой класс приоритета.

Task Manager в Windows дает возможность изменять класс приоритета процесса. На рисунке ниже показана вкладка Processes в окне Task Manager со списком выполняемых на данный момент процессов. В колонке Base Pri сообщается класс приоритета каждого процесса. Вы можете изменить его, выбрав процесс и указав другой класс в подменю Set Priority контекстного меню.



Только что созданный поток получает относительный приоритет normal. Почему *CreateThread* не позволяет задать относительный приоритет — для меня так и остается загадкой. Такая операция осуществляется вызовом функции:

```
B00L SetThreadPriority(
   HANDLE hThread,
   int  nPriority);
```

Разумеется, параметр *hTheard* указывает на поток, чей приоритет вы хотите изменить, а через *nPriority* передается один из идентификаторов (см. таблицу 7-6).

Табл. 7-6. Относительные приоритеты потоков

Относительный приоритет потока	Идентификатор
Time-critical	THREAD_PRIORITY_TIME_CRITICAL
Highest	THREAD_PRIORITY_HIGHEST
Above normal	THREAD_PRIORITY_ABOVE_NORMAL
Normal	THREAD_PRIORITY_NORMAL
Below normal	THREAD_PRIORITY_BELOW_NORMAL
Lowest	THREAD_PRIORITY_LOWEST
Idle	THREAD_PRIORITY_IDLE

Функция *GetThreadPriority*, парная *SetThreadPriority*, позволяет узнать относительный приоритет потока:

```
int GetThreadPriority(HANDLE hThread);
```

Она возвращает один из идентификаторов, показанных в таблице выше.

Чтобы создать поток с относительным приоритетом idle, сделайте, например, так:

```
DWORD dwThreadID;
HANDLE hThread = CreateThread(NULL, 0, ThreadFunc, NULL,
    CREATE_SUSPENDED, adwThreadlD);
SetThreadPriority(hThread, THREAD_PRIORITY_IDLE);
ResumeThread(hThread);
CloseHandle(hThread);
```

Заметьте, что *CreateThread* всегда создает поток с относительным приоритетом normal. Чтобы присвоить потоку относительный приоритет idle, создайте приостановленный поток, передав в *CreateThread* флаг CREATE_SUSPENDED, а потом вызовите *SetThreadPriority* и установите нужный приоритет. Далее можно вызвать *ResumeThread*, и поток будет включен в число планируемых. Сказать заранее, когда поток получит процессорное время, нельзя, но планировщик уже учитывает его новый приоритет. Выполнив эти операции, вы можете закрыть описатель потока, чтобы соответствующий объект ядра был уничтожен по завершении данного потока.

> **Примечание.** Ни одна Windows-функция не возвращает уровень приоритета потока. Такая ситуация создана преднамеренно. Вспомните, что Майкрософт может в любой момент изменить алгоритм распределения процессорного времени. Поэтому при разработке приложений не сто-

ит опираться на какие-то нюансы этого алгоритма. Используйте классы приоритетов процессов и относительные приоритеты потоков, и ваши приложения будут нормально работать как в нынешних, так и в следующих версиях Windows.

Динамическое изменение уровня приоритета потока

Уровень приоритета, получаемый комбинацией относительного приоритета потока и класса приоритета процесса, которому принадлежит данный поток, называют *базовым уровнем приоритета потока*. Иногда система изменяет уровень приоритета потока. Обычно это происходит в ответ на некоторые события, связанные с вводом-выводом (например, на появление оконных сообщений или чтение с диска).

Так, поток с относительным приоритетом normal, выполняемый в процессе с классом приоритета high, имеет базовый приоритет 13. Если пользователь нажимает какую-нибудь клавишу, система помещает в очередь потока сообщение WM_KEYDOWN. А поскольку в очереди потока появилось сообщение, поток становится планируемым. При этом драйвер клавиатуры может заставить систему временно поднять уровень приоритета потока с 13 до 15 (действительное значение может отличаться в ту или другую сторону).

Процессор исполняет поток в течение отведенного отрезка времени, а по его истечении система снижает приоритет потока на 1, до уровня 14. Далее потоку вновь выделяется квант процессорного времени, по окончании которого система опять снижает уровень приоритета потока на 1. И теперь приоритет потока снова соответствует его базовому уровню.

Текущий уровень приоритета не может быть ниже базового. Кроме того, драйвер устройства, «разбудивший» поток, сам устанавливает величину повышения приоритета. И опять же Майкрософт не документирует, насколько повышаются эти значения конкретными драйверами. Таким образом, она получает возможность тонко настраивать динамическое изменение приоритетов потоков в операционной системе, чтобы та максимально быстро реагировала на действия пользователя.

Система повышает приоритет только тех потоков, базовый уровень которых находится в пределах 1-15. Именно поэтому данный диапазон называется «областью динамического приоритета» (dynamic priority range). Система не допускает динамического повышения приоритета потока до уровней реального времени (более 15). Поскольку потоки с такими уровнями обслуживают системные функции, это ограничение не дает приложению нарушить работу операционной системы. И, кстати, система никогда не меняет приоритет потоков с уровнями реального времени (от 16 до 3-1).

Некоторые разработчики жаловались, что динамическое изменение приоритета системой отрицательно сказывается на производительности их приложений, и поэтому Майкрософт добавила две функции, позволяющие отключать этот механизм:

```
BOOL SetProcessPriorityBoost(
    HANDLE hProcess,
    BOOL bDisablePriorityBoost);
BOOL SetThreadPriorityBoost(
    HANDLE hThread,
    BOOL bDisablePriorityBoost);
```

SetProcessPriorityBoost заставляет систему включить или отключить изменение приоритетов всех потоков в указанном процессе, а *SetThreadPriorityBoost* действует применительно к отдельным потокам. Эти функции имеют свои аналоги, позволяющие определять, разрешено или запрещено изменение приоритетов:

```
BOOL GetProcessPriorityBoost(
    HANDLE hProcess,
    PBOOL pbDisablePriorityBoost);
BOOL GetThreadPriorityBoost(
    HANDLE hThread,
    PBOOL pbDisablePriorityBoost);
```

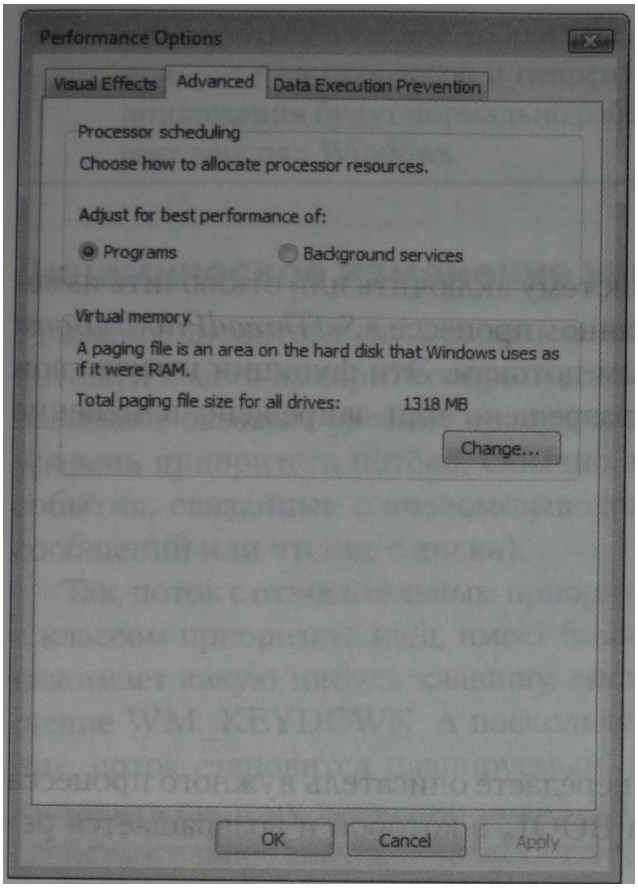
Каждой из этих двух функций вы передаете описатель нужного процесса или потока и адрес переменной типа BOOL, в которой и возвращается результат.

Есть еще одна ситуация, в которой система динамически повышает приоритет потока. Представьте, что поток с приоритетом 4 готов к выполнению, но не может получить доступ к процессору из-за того, что его постоянно занимают потоки с приоритетом 8. Это типичный случай «голодания» потока с более низким приоритетом. Обнаружив такой поток, не выполняемый на протяжении уже трех или четырех секунд, система поднимает его приоритет до 15 и выделяет ему двойную порцию времени. По его истечении потоку немедленно возвращается его базовый приоритет.

Подстройка планировщика для активного процесса

Когда пользователь работает с окнами какого-то процесса, последний считается *активным* (foreground process), а остальные процессы — *фоновыми* (background processes). Естественно, пользователь заинтересован в повышенной отзывчивости активного процесса по сравнению с фоновыми., Для этого Windows подстраивает алгоритм планирования потоков активного процесса. В Windows 2000, когда процесс становится активным, система выделяет его потокам более длительные кванты времени. Такая регулировка применяется только к процессам с классом приоритета normal.

Windows Vista позволяет модифицировать работу этого механизма подстройки. Щелкнув кнопку Performance Options на вкладке Advanced диалогового окна System Properties, вы открываете следующее окно.



Если пользователь выбирает оптимизацию работы программ (это настройка по умолчанию в Windows Vista), система выполняет подстройку. Если же пользователь предпочитает оптимизацию работы фоновых служб, никакой подстройки не выполняется.

Приоритеты запросов ввода-вывода

От расстановки приоритетов потоков зависит распределение между ними времени процессора. Но потоки исполняют, наряду с вычислениями, запросы дискового ввода-вывода. Так, поток с низким приоритетом, получивший время процессора, за очень короткое время может поставить в очередь тысячи запросов ввода-вывода. Поскольку операции ввода-вывода требуют сравнительно много времени, такой низкоприоритетный поток способен существенно снизить отзывчивость системы, приостановив исполнение высокоприоритетных потоков. Именно в этом причина «торможения» системы во время работы низкоприоритетных служб, таких как утилиты для дефрагментации диска, антивирусов, индексирующих программ и т.п.

Теперь в Windows Vista потоки могут задавать приоритеты запросов ввода-вывода. Так, можно сообщить Windows, что данный поток вправе генерировать только низкоприоритетные запросы ввода-вывода, вызвав SetTnreadPriority с передачей флага THREAD_MODE_BACKGROUND_BEGIN. Заметьте, что при этом понижается и приоритет потока в распределении процессорного времени. Восстановить способность потока генериро-

вать обычные запросы ввода-вывода (и вернуть ему обычный приоритет для планировщика) можно повторным вызовом *SetThreadPriority*, но с передачей THREAD_MODE_BACKGROUND_END. При вызове *SetThreadPriority* с любым из этих флагов необходимо также передавать описатель вызывающего потока (его можно получить, вызвав *GetCurrentThread*). Потокам запрещено изменять приоритет операций ввода-вывода других потоков.

Чтобы понизить приоритет в отношении ввода-вывода и распределении процессорного времени для всех потоков процесса, следует вызвать *SetPriorityClass* с передачей флага PROCESS_MODE_BACKGROUND_BEGIN. Вернуть процесс в исходное состояние можно вызовом *SetPriorityClass* с передачей PROCESS_MODE_BACKGROUND_END. При вызове *SetPriorityClass* с любым из этих флагов необходимо также передавать описатель вызывающего процесса (его можно получить, вызвав *GetCurrentProcess*). Потокам запрещено изменять приоритет операций ввода-вывода потоков в других процессах.

Возможна и более тонкая настройка приоритета запросов ввода-вывода. Так, поток, обладающий нормальным приоритетом, может генерировать низкоприоритетные запросы ввода-вывода к отдельному файлу. Вот как это делается:

```
FILE_IO_PRIORITY_HINT_INFO phi;
phi.PriorityHint = IoPriorityHintLow;
SetFileInformationByHandle(
    hFile, FileIoPriorityHintInfo, &phi, sizeof(PriorityHint));
```

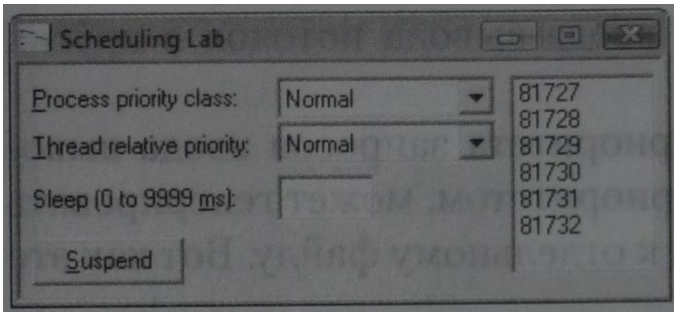
Приоритет, заданный вызовом *SetFileInformationByHandle*, заменяет приоритет, заданный на уровне процесс или потока вызовом *SetPriorityClass* или *SetThreadPriority*, соответственно.

Как разработчик, вы отвечаете за использование этих механизмов для поддержания отзывчивости активных приложений, а для этого необходимо избегать *инверсии приоритетов* (priority inversion). Так, если потоки с нормальным приоритетом выполняют много операций ввода-вывода, низкоприоритетные могут ожидать исполнения своих запросов на чтение и запись в течение нескольких *секунд*. Если же низкоприоритетный поток заблокирует ресурс и потокам с нормальным приоритетом придется ждать его освобождения, дело закончится тем, что приоритетные потоки будут ждать завершения запросов ввода-вывода, сгенерированных фоновыми потоками. Более того, такая ситуация может возникнуть, даже если фоновые потоки не сгенерировали ни одного запроса ввода-вывода. В силу этих причин совместное использование синхронизирующих объектов потоками с нормальным и низким уровнем приоритета следует свести к минимуму (а лучше вовсе избегать его) — так можно предотвратить инверсию приоритетов, когда потокам с нормальным приоритетом приходится ждать освобождения блокировок, установленных потоками с фоновыми уровнями приоритета.

Примечание Описанные выше механизмы используются функцией SuperFetch. Подробнее об этом см. По ссылке http://www.microsoft.com/whdc/driver/priorityio.mspx

Программа-пример Scheduling Lab

Эта программа, «07 SchedLab.exe» (см. листинг ниже), позволяет экспериментировать с классами приоритетов процессов и относительными приоритетами потоков и исследовать их влияние на общую производительность системы. Файлы исходного кода и ресурсов этой программы находятся в каталоге 07-SchedLab на компакт-диске, прилагаемом к книге. После запуска SchedLab открывается окно, показанное ниже.



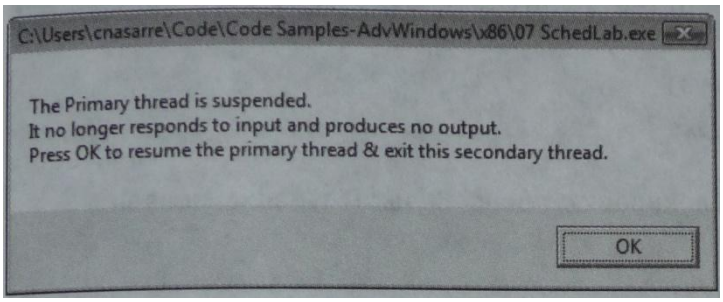
Изначально первичный поток работает очень активно, и степень использования процессора подскакивает до 100%. Все, чем он занимается, — постоянно увеличивает исходное значение на 1 и выводит текущее значение в крайнее справа окно списка. Все эти числа не несут никакой смысловой информации; их появление просто демонстрирует, что поток чем-то занят. Чтобы прочувствовать, как повлияет на него изменение приоритета, запустите по крайней мере два экземпляра программы. Можете также открыть Task Manager и понаблюдать за нагрузкой на процессор, создаваемой каждым экземпляром.

В начале теста процессор будет загружен на 100%, и вы увидите, что все экземпляры SchedLab получают примерно равные кванты процессорного времени. (Task Manager должен показать практически одинаковые процентные доли для всех ее экземпляров.) Как только вы поднимете класс приоритета одного из экземпляров до above normal или high, львиную долю процессорного времени начнет получать именно этот экземпляр, а аналогичные показатели для других экземпляров резко упадут. Однако они никогда не опустятся до нуля — это действует механизм динамического повышения приоритета «голодающих» процессов. Теперь вы можете самостоятельно поиграть с изменением классов приоритетов процессов и относительных приоритетов потоков. Возможность установки класса приоритета real-time я исключил намеренно, чтобы не нарушить работу операционной системы. Если вы все же хотите поэкспериментировать с этим приоритетом, вам придется модифицировать исходный текст моей программы.

Используя поле Sleep, можно приостановить первичный поток на заданное число миллисекунд в диапазоне от 0 до 9999. Попробуйте приостанав-

ливать его хотя бы на 1 мс и посмотрите, сколько процессорного времени это позволит сэкономить. На своем ноутбуке с 2,2-ГГц процессором Pentium я выиграл аж 99% — впечатляет!

Кнопка Suspend заставляет первичный поток создать дочерний поток, который приостанавливает родительский и выводит следующее окно.



Пока это окно открыто, первичный поток полностью отключается от процессора, а дочерний тоже не требует процессорного времени, так как ждет от пользователя дальнейших действий. Вы можете свободно перемещать это окно в пределах экрана или убрать его в сторону от основного окна программы. Поскольку первичный поток остановлен, основное окно не принимает оконных сообщений (в том числе WM_PAINT), Это еще раз доказывает, что поток задержан. Закрыв окно с сообщением, вы возобновите первичный поток, и нагрузка на процессор снова возрастет до 100%.

А теперь проведите еще один эксперимент. Откройте диалоговое окно Performance Options (я говорил о нем в предыдущем разделе) и выберите переключатель Background Services (или, наоборот, Application). Потом запустите несколько экземпляров моей программы с классом приоритета normal и выберите один из них, сделав его активным процессом. Вы сможете наглядно убедиться, как эти переключатели влияют на активные и фоновые процессы.

SchedLab.cpp

```
/******************************************************************************
Module:  SchedLab.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"
#include <StrSafe.h>


//////////////////////////////////////////////////////////////////////////////


DWORD WINAPI ThreadFunc(PVOID pvParam) {
```

```
   HANDLE hThreadPrimary = (HANDLE) pvParam;
   SuspendThread(hThreadPrimary);
   chMB(
      "The Primary thread is suspended.\n"
      "It no longer responds to input and produces no output.\n"
      "Press OK to resume the primary thread & exit this secondary thread.\n");
   ResumeThread(hThreadPrimary);
   CloseHandle(hThreadPrimary);

   // To avoid deadlock, call EnableWindow after ResumeThread.
   EnableWindow(
      GetDlgItem(FindWindow(NULL, TEXT("Scheduling Lab")), IDC_SUSPEND),
      TRUE);
   return(0);
}


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog (HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_SCHEDLAB);

   // Initialize process priority classes
   HWND hWndCtl = GetDlgItem(hWnd, IDC_PROCESSPRIORITYCLASS);

   int n = ComboBox_AddString(hWndCtl, TEXT("High"));
   ComboBox_SetItemData(hWndCtl, n, HIGH_PRIORITY_CLASS);

   // Save our current priority class
   DWORD dwpc = GetPriorityClass(GetCurrentProcess());

   if (SetPriorityClass(GetCurrentProcess(), BELOW_NORMAL_PRIORITY_CLASS)) {

      // This system supports the BELOW_NORMAL_PRIORITY_CLASS class

      // Restore our original priority class
      SetPriorityClass(GetCurrentProcess(), dwpc);

      // Add the Above Normal priority class
      n = ComboBox_AddString(hWndCtl, TEXT("Above normal"));
      ComboBox_SetItemData(hWndCtl, n, ABOVE_NORMAL_PRIORITY_CLASS);
```

```
   dwpc = 0;  // Remember that this system supports below normal
}

int nNormal = n = ComboBox_AddString(hWndCtl, TEXT("Normal"));
ComboBox_SetItemData(hWndCtl, n, NORMAL_PRIORITY_CLASS);

if (dwpc == 0) {

   // This system supports the BELOW_NORMAL_PRIORITY_CLASS class

   // Add the Below Normal priority class
   n = ComboBox_AddString(hWndCtl, TEXT("Below normal"));
   ComboBox_SetItemData(hWndCtl, n, BELOW_NORMAL_PRIORITY_CLASS);
}

n = ComboBox_AddString(hWndCtl, TEXT("Idle"));
ComboBox_SetItemData(hWndCtl, n, IDLE_PRIORITY_CLASS);

ComboBox_SetCurSel(hWndCtl, nNormal);

// Initialize thread relative priorities
hWndCtl = GetDlgItem(hWnd, IDC_THREADRELATIVEPRIORITY);

n = ComboBox_AddString(hWndCtl, TEXT("Time critical"));
ComboBox_SetItemData(hWndCtl, n, THREAD_PRIORITY_TIME_CRITICAL);

n = ComboBox_AddString(hWndCtl, TEXT("Highest"));
ComboBox_SetItemData(hWndCtl, n, THREAD_PRIORITY_HIGHEST);

n = ComboBox_AddString(hWndCtl, TEXT("Above normal"));
ComboBox_SetItemData(hWndCtl, n, THREAD_PRIORITY_ABOVE_NORMAL);

nNormal = n = ComboBox_AddString(hWndCtl, TEXT("Normal"));
ComboBox_SetItemData(hWndCtl, n, THREAD_PRIORITY_NORMAL);

n = ComboBox_AddString(hWndCtl, TEXT("Below normal"));
ComboBox_SetItemData(hWndCtl, n, THREAD_PRIORITY_BELOW_NORMAL);

n = ComboBox_AddString(hWndCtl, TEXT("Lowest"));
ComboBox_SetItemData(hWndCtl, n, THREAD_PRIORITY_LOWEST);

n = ComboBox_AddString(hWndCtl, TEXT("Idle"));
ComboBox_SetItemData(hWndCtl, n, THREAD_PRIORITY_IDLE);

ComboBox_SetCurSel(hWndCtl, nNormal);
```

```
   Edit_LimitText(GetDlgItem(hWnd, IDC_SLEEPTIME), 4);   // Maximum of 9999
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand (HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         PostQuitMessage(0);
         break;

      case IDC_PROCESSPRIORITYCLASS:
         if (codeNotify == CBN_SELCHANGE) {
            SetPriorityClass(GetCurrentProcess(), (DWORD)
               ComboBox_GetItemData(hWndCtl, ComboBox_GetCurSel(hWndCtl)));
         }
         break;

      case IDC_THREADRELATIVEPRIORITY:
         if (codeNotify == CBN_SELCHANGE) {
            SetThreadPriority(GetCurrentThread(), (DWORD)
               ComboBox_GetItemData(hWndCtl, ComboBox_GetCurSel(hWndCtl)));
         }
         break;

      case IDC_SUSPEND:
         // To avoid deadlock, call EnableWindow before creating
         // the thread that calls SuspendThread.
         EnableWindow(hWndCtl, FALSE);

         HANDLE hThreadPrimary;
         DuplicateHandle(GetCurrentProcess(), GetCurrentThread(),
            GetCurrentProcess(), &hThreadPrimary,
            THREAD_SUSPEND_RESUME, FALSE, DUPLICATE_SAME_ACCESS);
         DWORD dwThreadID;
         CloseHandle(chBEGINTHREADEX(NULL, 0, ThreadFunc,
            hThreadPrimary, 0, &dwThreadID));
         break;
   }
}
```

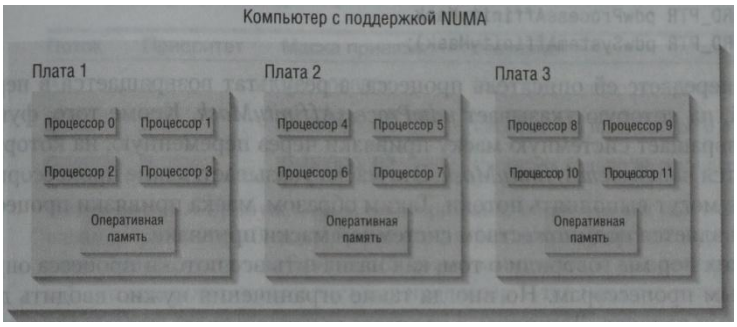
```
//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }

   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////

class CStopwatch {
public:
   CStopwatch() { QueryPerformanceFrequency(&m_liPerfFreq); Start();
}

   void Start() { QueryPerformanceCounter(&m_liPerfStart); }

   __int64 Now() const { // Возвращает время (в мс) с момента вызова Start
      LARGE_INTEGER liPerfNow;
      QueryPerformanceCounter(&liPerfNow);
      return(((liPerfNow.QuadPart - m_liPerfStart.QuadPart) * 1000)
         / m_liPerfFreq.QuadPart);
   }

private:
   LARGE_INTEGER m_liPerfFreq;    // отсчетов/с
   LARGE_INTEGER m_liPerfStart;   // начальный счет
};

__int64 FileTimeToQuadWord (PFILETIME pft) {
   return(Int64ShllMod32(pft->dwHighDateTime, 32) | pft->dwLowDateTime);
}

int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   HWND hWnd =
      CreateDialog(hInstExe, MAKEINTRESOURCE(IDD_SCHEDLAB), NULL, Dlg_Proc);
   BOOL fQuit = FALSE;
```

```
   while (!fQuit) {
      MSG msg;
      if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

         // IsDialogMessage allows keyboard navigation to work properly.
         if (!IsDialogMessage(hWnd, &msg)) {

            if (msg.message == WM_QUIT) {
               fQuit = TRUE;  // For WM_QUIT, terminate the loop.
            } else {
               // Not a WM_QUIT message. Translate it and dispatch it.
               TranslateMessage(&msg);
               DispatchMessage(&msg);
            }
         }  // if (!IsDialogMessage())
      } else {

         // Add a number to the listbox
         static int s_n = -1;
         TCHAR sz[20];
         StringCchPrintf(sz, _countof(sz), TEXT("%u"), ++s_n);
         HWND hWndWork = GetDlgItem(hWnd, IDC_WORK);
         ListBox_SetCurSel(hWndWork, ListBox_AddString(hWndWork, sz));

         // Remove some strings if there are too many entries
         while (ListBox_GetCount(hWndWork) > 100)
            ListBox_DeleteString(hWndWork, 0);

         // How long should the thread sleep
         int nSleep = GetDlgItemInt(hWnd, IDC_SLEEPTIME, NULL, FALSE);
         if (chINRANGE(1, nSleep, 9999))
            Sleep(nSleep);
      }
   }

   DestroyWindow(hWnd);
   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

Привязка потоков к процессорам

По умолчанию Windows Vista использует *нежесткую привязку* (soft affinity) потоков к процессорам. Это означает, что при прочих равных условиях, система пытается выполнять поток на том же процессоре, на котором он работал в последний раз. При таком подходе можно повторно использовать данные, все еще хранящиеся в кэше процессора.

В новой компьютерной архитектуре NUMA (Non-Uniform Memory Access) машина состоит из нескольких плат, на каждой из которых находятся четыре процессора и отдельный банк памяти. На следующей иллюстрации показана машина с тремя такими платами, в сумме содержащими 12 процессоров. Отдельный поток может выполняться на любом из этих процессоров.



Система NUMA достигает максимальной производительности, если процессоры используют память на своей плате. Если же они обращаются к памяти на другой плате, производительность резко падает. В такой среде желательно, чтобы потоки одного процесса выполнялись на процессорах 0-3, другого — на процессорах 4-7 и т. д. Windows 2000 позволяет подстроиться под эту архитектуру, закрепляя отдельные процессы и потоки за конкретными процессорами. Иначе говоря, вы можете контролировать, на каких процессорах будут выполняться ваши потоки. Такая привязка называется *жесткой* (hard affinity).

Количество процессоров система определяет при загрузке, и эта информация становится доступной приложениям через функцию *GetSystemInfo* (о ней — в главе 14). По умолчанию любой поток может выполняться на любом процессоре. Чтобы потоки отдельного процесса работали лишь на некоем подмножестве процессоров, используйте функцию *SetProcessAffinityMask*:

```
BOOL SetProcessAffinityMask(
    HANDLE hProcess,
    DW0RD_PTR dwProcessAffinityMask);
```

В *первом* параметре, *hProcess,* передается описатель процесса. Второй параметр, *dwProcessAfjHnityMask,* — это битовая маска, указывающая, на каких

процессорах могут выполняться потоки данного процесса. Передав, например, значение 0x00000005, мы разрешим процессу использовать только процессоры 0 и 2 (процессоры 1 и 3-31 ему будут недоступны).

Привязка к процессорам наследуется дочерними процессами. Так, если для родительского процесса задана битовая маска 0x00000005, у всех потоков его дочерних процессов будет идентичная маска, и они смогут работать лишь на тех же процессорах. Для привязки целой группы процессов к определенным процессорам используйте объект ядра «задание» (см. главу 5).

Ну и, конечно же, есть функция, позволяющая получить информацию о такой привязке:

```
B00L GetProcessAffinityMask(
    HANDLE hProcess,
    PDWORD_PTR pdwProcessAffinityMask,
    PDW0RD_PTR pdwSystemAffinityMask);
```

Вы передаете ей описатель процесса, а результат возвращается в переменной, на которую указывает *pdwProcessAffinityMask*. Кроме того, функция возвращает системную маску привязки через переменную, на которую ссылается *pdwSystemAffinityMask*. Эта маска указывает, какие процессоры в системе могут выполнять потоки. Таким образом, маска привязки процесса всегда является подмножеством системной маски привязки.

До сих пор мы говорили о том, как назначить все потоки процесса определенным процессорам. Но иногда такие ограничения нужно вводить для отдельных потоков. Допустим, в процессе имеется четыре потока, выполняемые на четырехпроцессорной машине. Один из потоков занимается особо важной работой, и вы, желая повысить вероятность того, что у него всегда будет доступ к вычислительным мощностям, запрещаете остальным потокам использовать процессор 0.

Задать маски привязки для отдельных потоков позволяет функция:

```
DW0RD_PTR SetThreadAffinityMask(
    HANDLE hThread,
    DW0RD_PTR dwThreadAffinityMask);
```

В параметре *hThread* передается описатель потока, а *dwThreadAffinityMask* определяет процессоры, доступные этому потоку. Параметр *dwThreadAffinityMask* должен быть корректным подмножеством маски привязки процесса, которому принадлежит данный поток. Функция возвращает предыдущую маску привязки потока. Вот как ограничить три потока из нашего примера процессорами 1,2 и 3:

```
// поток 0 выполняется только на процессоре 0
SetThreadAffinityMask(hThread0, 0x00000001);

// потоки 1, 2, 3 выполняются на процессорах 1, 2, 3
SetThreadAffinityMask(hThread1, 0x0000000E);
```

```
SetThreadAffinityMask(hThread2, 0x0000000E);
SetThreadAffinityMask(hThread3, 0x0000000E);
```

При загрузке система тестирует процессоры типа ***x86*** на наличие в них знаменитого «жучка» в операциях деления чисел с плавающей точкой (эта ошибка имеется в некоторых Pentium). Она привязывает поток, выполняющий потенциально сбойную операцию деления, к исследуемому процессору и сравнивает результат с тем, что должно быть на самом деле. Такая последовательность операций выполняется для каждого процессора в машине.

> **Примечание**. В большинстве сред вмешательство в системную привязку потоков нарушает нормальную работу планировщика, не позволяя ему максимально эффективно распределять вычислительные мощности. Рассмотрим один пример.
>
> | Поток | Приоритет | Маска привязки | Результат |
> |---|---|---|---|
> | A | 4 | 0x00000001 | Работает только на процессоре 0 |
> | B | 8 | 0x00000003 | Работает на процессоре 0 и 1 |
> | C | 6 | 0x00000002 | Работает только на процессоре 1 |
>
> Когда поток A пробуждается, планировщик, видя, что тот жестко привязан к процессору 0, подключает его именно к этому процессору. Далее активизируется поток B, который может выполняться на процессорах 0 и 1, и планировщик выделяет ему процессор 1, так как процессор 0 уже занят. Пока все нормально.
>
> Но вот пробуждается поток C, привязанный к процессору 1. Этот процессор уже занят потоком B с приоритетом 8, а значит, поток C, приоритет которого равен 6, не может его вытеснить. Он, конечно, мог бы вытеснить поток A (с приоритетом 4) с процессора 0, но у него нет прав на использование этого процессора. Вот как жесткая привязка может изменить схему приоритетов, принятую планировщиком.

Ограничение потока одним процессором не всегда является лучшим решением. Ведь может оказаться так, что три потока конкурируют за доступ к процессору 0, тогда как процессоры 1, 2 и 3 простаивают. Гораздо лучше сообщить системе, что поток желательно выполнять на определенном процессоре, но, если он занят, его можно переключать на другой процессор.

Указать предпочтительный (идеальный) процессор позволяет функция:

```
DWORD SetThreadIdealProcessor(
    HANDLE hThread,
    DWORD dwIdealProcessor);
```

В параметре *hThread* передается описатель потока. В отличие от функций, которые мы уже рассматривали, параметр *dwIdealProcessor* содержит не битовую маску, а целое значение в диапазоне 0-31, которое указывает предпочтительный процессор для данного потока. Передав в нем константу

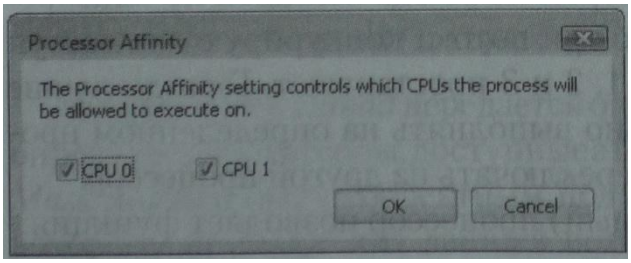
MAXIMUM_PROCESSORS (в WinNT.h она определена как 32 для 32-разрядных систем и 64 — для 64-разрядных), вы сообщите системе, что потоку не требуется предпочтительный процессор. Функция возвращает установленный ранее номер предпочтительного процессора или MAXIMUM_PROCESSORS, если таковой процессор не задан.

Привязку к процессорам можно указать в заголовке исполняемого файла. Как ни странно, но подходящего ключа компоновщика на этот случай, похоже, не предусмотрено. Тем не менее вы можете воспользоваться функциями, определенными в ImageHlp.h:

```
// загружаем EXE-файл в память
PLOADED_IMAGE pLoadedImage = ImageLoad(szExeName, NULL);

// получаем информацию о текущей загрузочной конфигурации EXE-файла
IMAGE_LOAD_CONFIG_DIRECTORY ilcd;
GetImageConfigInformation(pLoadedImage, &ilcd);

// изменяем маску привязки процесса
ilcd.ProcessAffinityMask = 0x00000003; // I desire CPUs 0 and 1

// сохраняем новую информацию о загрузочной конфигурации
SetImageConfigInformation(pLoadedImage, &ilcd);

// выгружаем EXE-файл из памяти
ImageUnload(pLoadedImage);
```

Детально описывать эти функции я не стану — при необходимости вы найдете их в документации Platform SDK.

Указав при запуске ImageCfg ключ -a, вы сможете изменить маску привязки для приложения. Конечно, все, что делает эта утилита, — вызывает функции, перечисленные в подсказке по ее применению. Обратите внимание на ключи, который сообщает системе, что исполняемый файл может выполняться исключительно на однопроцессорной машине.



При запуске Windows Vista на машине с процессорами типа x86 можно ограничить число процессоров, используемых системой. Во время загрузки система анализирует содержимое хранилища загрузочной конфигурации (boot configuration data, BCD), заменившего файл boot.ini. BCD предоставляет уровень абстрагирования для оборудования и микрокода компьютера. Подробные сведения об этой новинке см. по ссылке http://www.microsoft.com/whdc/system/platform/firmware/bcd.mspx.

Программное конфигурирование BCD осуществляется средствами Windows Management Instrumentation (WMI), но некоторые общие параметры можно настроить и через графический интерфейс. Так, чтобы ограничить число процессоров, доступных Windows, откройте Control Panel и щелкните Administrative Tools и выберите System Configuration. На вкладке Boot щелкните кнопку Advanced, пометьте флажком параметр Number Of Processors и введите нужное число процессоров в соответствующее поле.

Оглавление

ГЛАВА 8

Синхронизация потоков в пользовательском режиме

Windows лучше всего работает, когда все потоки могут заниматься своим делом, не взаимодействуя друг с другом. Однако такая ситуация очень редка. Обычно поток создается для выполнения определенной работы, о завершении которой, вероятно, захочет узнать другой поток.

Все потоки в системе должны иметь доступ к системным ресурсам — кучам, последовательным портам, файлам, окнам и т. д. Если один из потоков запросит монопольный доступ к какому-либо ресурсу, другим потокам, которым тоже нужен этот ресурс, не удастся выполнить свои задачи. А с другой стороны, просто недопустимо, чтобы потоки бесконтрольно пользовались ресурсами. Иначе может получиться так, что один поток пишет в блок памяти, из которого другой что-то

считывает. Представьте, вы читаете книгу, а в это время кто-то переписывает текст на открытой вами странице. Ничего хорошего из этого не выйдет.

Потоки должны взаимодействовать друг с другом в двух основных случаях:

- совместно используя разделяемый ресурс (чтобы не разрушить его);
- когда нужно уведомлять другие потоки о завершении каких-либо операций.

Синхронизации потоков — тематика весьма обширная, и мы рассмотрим ее в этой и следующих главах. Одна новость вас обрадует: в Windows есть масса средств, упрощающих синхронизацию потоков. Но другая огорчит: точно спрогнозировать, в какой момент потоки будут делать то-то и то-то, крайне сложно. Наш мозг не умеет работать асинхронно; мы обдумываем свои мысли старым добрым способом — одну за другой по очереди. Однако многопоточная среда ведет себя иначе.

С программированием для многопоточной среды я впервые столкнулся в 1992 г. Поначалу я делал уйму ошибок, так что в главах моих книг и

журнальных статьях хватало огрехов, связанных с синхронизацией потоков. Сегодня я намного опытнее и действительно считаю, что уж в этой-то книге все безукоризненно (хотя самонадеянности у меня вроде бы поубавилось). Единственный способ освоить синхронизацию потоков — заняться этим на практике. Здесь и в следующих главах я объясню, как работает система и как правильно синхронизировать потоки. Однако вам придется стоически переносить трудности: приобретая опыт, ошибок не избежать.

Атомарный доступ: семейство Intelocked-функций

Бóльшая часть синхронизации потоков связана с *атомарным доступом* (atomic access) — монопольным захватом ресурса обращающимся к нему потоком. Возьмем простой пример.

```
// определяем глобальную переменную
long g_x = 0;

DWORD WINAPI ThreadFunc1(PV0ID pvParam) {
    g_x++;
    return(0);
}

DWORD WINAPI ThreadFunc2(PV0ID pvParam) {
    g_x++;
    return(0);
}
```

Я объявил глобальную переменную *g_x* и инициализировал ее нулевым значением. Теперь представьте, что я создал два потока: один выполняет *ThreadFunc1*, другой — *ThreadFunc2*. Код этих функций идентичен: обе увеличивают значение глобальной переменной *g_x* на 1. Поэтому вы, наверное, подумали: когда оба потока завершат свою работу, значение *g_x* будет равно 2. Так ли это? Может быть. При таком коде заранее сказать, каким будет конечное значение *g_x*, нельзя. И вот почему. Допустим, компилятор сгенерировал для строки, увеличивающей *g_x* на 1, следующий код:

```
MOV EAX, [g_x]    ; значение из g_x помещается в регистр
INC EAX           ; значение регистра увеличивается на 1
MOV [g_x], EAX    ; значение из регистра помещается обратно в g_x
```

Вряд ли оба потока будут выполнять этот код в одно и то же время. Если они будут делать это по очереди — сначала один, потом другой, тогда мы получим такую картину:

```
MOV EAX, [g_x]    ; поток 1: в регистр помещается 0
INC EAX           ; поток 1: значение регистра увеличивается на 1
MOV [g_x], EAX    ; поток 1: значение 1 помещается в g_x
```

```
NOV EAX, [g_x]     ; поток 2: в регистр помещается 1
INC EAX            ; поток 2: значение регистра увеличивается до 2
NOV [g_x], EAX     ; поток 2: значение 2 помещается в g_x
```

После выполнения обоих потоков значение *g_x* будет равно 2. Это просто замечательно и как раз то, что мы ожидали: взяв переменную с нулевым значением, дважды увеличили ее на 1 и получили в результате 2. Прекрасно. Но постойте-ка, ведь Windows — это среда, которая поддерживает многопоточность и вытесняющую многозадачность. Значит, процессорное время в любой момент может быть отнято у одного потока и передано другому. Тогда код, приведенный мной выше, может выполняться и таким образом:

```
MOV EAX, [g_x]        ; поток 1: в регистр помещается 0
INC EAX               ; поток 1: значение регистра увеличивается на 1

MOV EAX, [g_x]        ; поток 2: в регистр помещается 0
INC EAX               ; поток 2: значение регистра увеличивается на 1
MOV [g_x], EAX        ; поток 2: значение 1 помещается в g_x

MOV [g_x], EAX        ; поток 1: значение 1 помещается в g_x
```

А если код будет выполняться именно так, конечное значение *g_x* окажется равным 1, а не 2, как мы думали! Довольно пугающе, особенно если учесть, как мало у нас рычагов управления планировщиком. Фактически, даже при сотне потоков, которые выполняют функции, идентичные нашей, в конечном итоге вполне можно получить в *g_x* все ту же единицу! Очевидно, что в таких условиях работать просто нельзя. Мы вправе ожидать, что, дважды увеличив 0 на 1, при любых обстоятельствах получим 2. Кстати, результаты могут зависеть от того, как именно компилятор генерирует машинный код, а также от того, как процессор выполняет этот код и сколько процессоров установлено в машине. Это объективная реальность, в которой мы не в состоянии что-либо изменить. Однако в Windows есть ряд функций, которые (при правильном их использовании) гарантируют корректные результаты выполнения кода.

Решение этой проблемы должно быть простым. Все, что нам нужно, — это способ, гарантирующий приращение значения переменной на уровне атомарного доступа, т. е. без прерывания другими потоками. Семейство *Interlocked*-функций как раз и дает нам ключ к решению подобных проблем. Большинство разработчиков программного обеспечения недооценивает эти функции, а ведь они невероятно полезны и очень просты для понимания. Все функции из этого семейства манипулируют переменными на уровне атомарного доступа. Взгляните на *InterlockedExchangeAdd* и ее 64-разрядную версию *InterlockedExchangeAdd64*, манипулирующую значениями типа LONGLONG:

```
LONG InterlockedExchangeAdd(
    PLOMG volatile plAddend,
    LONG lIncrement);
```

```
LONGLONG InterlockedExchangeAdd64(
    PLOMGLONG volatile pllAddend,
    LONGLONG llIncrement);
```

Что может быть проще? Вы вызываете эту функцию, передавая адрес переменной типа LONG, и указываете добавляемое значение. *InterlockedExchangeAdd* гарантирует, что операция будет выполнена атомарно. Перепишем наш код вот так:

```
// определяем глобальную переменную
long g_x = 0;
DWORD WINAPI ThreadFunc1(PVOID pvParam) {
    InterlockedExchangeAdd(&g_x, 1);
    return(0);
}

DWORD WINAPI ThreadFunc2(PVOID pvParam) {
    InterlockedExchangeAdd(&g_x, 1);
    return(0);
}
```

Теперь вы можете быть уверены, что конечное значение *g_x* будет равно 2. Ну, вам уже лучше? Заметьте: в любом потоке, где нужно модифицировать значение разделяемой (общей) переменной типа LONG, следует пользоваться лишь *Interlocked-* функциями и никогда не прибегать к стандартным операторам языка C:

```
// переменная типа LONG, используемая несколькими потоками
LONG g_x; …

// неправильный способ увеличения переменной типа LONG
G_x++; …

// правильный способ увеличения переменной типа LONG
InterlockedExchangeAdd(&g_x, 1);
```

Как же работают *Interlocked*-функции? Ответ зависит от того, какую процессорную платформу вы используете. На компьютерах с процессорами семейства *x86* эти функции выдают по шине аппаратный сигнал, не давая другому процессору обратиться по тому же адресу памяти.

Вовсе не обязательно вникать в детали работы этих функций. Вам нужно знать лишь одно: они гарантируют монопольное изменение значений переменных независимо от того, как именно компилятор генерирует код и сколько процессоров установлено в компьютере. Однако вы должны позаботиться о выравнивании адресов переменных, передаваемых этим функциям, иначе они могут потерпеть неудачу. (О выравнивании данных я расскажу в главе 13).

Примечание. Библиотека C поддерживает функций *_aligned_malloc*, которая служит для выделения надлежащим образом выровненных блоков памяти. Вот как выглядит прототип этой функции:

```
void * _aligned_malloc(size_t size, size_t alignment);
```

Аргумент *size* определяет размер блока в байтах, а *alignment* — границу (в байтах), по которой должен быть выровнен блок. Аргумент *alignment* может принимать значения, представляющие собой различные степени числа 2.

Другой важный аспект, связанный с *Interlocked*-функциями, состоит в том, что они выполняются чрезвычайно быстро. Вызов такой функции обычно требует не более 50 тактов процессора, и при этом не происходит перехода из пользовательского режима в режим ядра (а он отнимает не менее 1000 тактов).

Кстати, *InterlockedExchangeAdd* позволяет не только увеличить, но и уменьшить значение — просто передайте во втором параметре отрицательную величину. *InterlockedExchangeAdd* возвращает исходное значение в **plAddend*.

Вот еще три функции из этого семейства:

```
LONG InterlockedExchange(
    PL0NG volatile plTarget,
    LONG lValue);

L0NGL0NG InterlockedExchange64(
    PL0NGL0NG volatile plTarget,
    L0NGL0N6 lValue);

PV0ID InterlockedExchangePointer(
    PV0ID* volatile ppvTarget,
    PV0ID pvValue);
```

InterlockedExchange и *InterlockedExchangePointer* монопольно заменяют текущее значение переменной типа LONG, адрес которой передается в первом параметре, на значение, передаваемое во втором параметре. В 32-разрядном приложении обе функции работают с 32-разрядными значениями, но в 64-разрядной программе первая оперирует с 32-разрядными значениями, а вторая — с 64-разрядными. Все функции возвращают исходное значение переменной. *InterlockedExchange* чрезвычайно полезна при реализации спин-блокировки (spin-lock):

```
// глобальная переменная, используемая как индикатор того, занят ли
// разделяемый ресурс
B00L g_fResourceInUse = FALSE; …
void Func1() {
```

```
    // ожидаем доступа к ресурсу
    while (IntertockedExchange (&g_fResourceInUee, TRUE) == TRUE)
       Sleep(0);

    // получаем ресурс в свое распоряжение
    …
    // доступ к ресурсу больше не нужен
    InterlockedExchange(&g_fResourceInUse, FALSE);
}
```

В этой функции постоянно «крутится» цикл *while*, в котором переменной *g_fResourceInUse* присваивается значение TRUE и проверяется ее предыдущее значение. Если оно было равно FALSE, значит, ресурс не был занят, но вызывающий поток только что занял его; на этом цикл завершается. В ином случае (значение было равно TRUE) ресурс занимал другой поток, и цикл повторяется.

Если бы подобный код выполнялся и другим потоком, его цикл *while* работал бы до тех пор, пока значение переменной *g_fResourceInUse* вновь не изменилось бы на FALSE. Вызов *InterlockedExchange* в конце функции показывает, как вернуть переменной *g_fResourceInUse* значение FALSE.

Применяйте эту методику с крайней осторожностью, потому что процессорное время при спин-блокировке тратится впустую. Процессору приходится постоянно сравнивать два значения, пока одно из них не будет «волшебным образом» изменено другим потоком. Учтите: этот код подразумевает, что все потоки, использующие спин-блокировку, имеют одинаковый уровень приоритета. К тому же, вам, наверное, придется отключить динамическое повышение приоритета этих потоков (вызовом *SetProcessPriorityBoost* или *SetThreadPriorityBoost*).

Вы должны позаботиться и о том, чтобы переменная — индикатор блокировки и данные, защищаемые такой блокировкой, не попали в одну кэш-линию (о кэш-линиях я расскажу в следующем разделе). Иначе процессор, использующий ресурс, будет конкурировать с любыми другими процессорами, которые пытаются обратиться к тому же ресурсу. А это отрицательно скажется на быстродействии.

Избегайте спин-блокировки на однопроцессорных машинах. «Крутясь» в цикле, поток впустую транжирит драгоценное процессорное время, не давая другому потоку изменить значение переменной. Применение функции *Sleep* в цикле *while* несколько улучшает ситуацию. С ее помощью вы можете отправлять свой поток в сон на некий случайный отрезок времени и, после каждой безуспешной попытки обратиться к ресурсу, увеличивать этот отрезок. Тогда потоки не будут зря отнимать процессорное время. В зависимости от ситуации вызов *Sleep* можно убрать или заменить вызовом *SwitchToThread*. Очень жаль, но, по-видимому, вам придется действовать здесь методом проб и ошибок.

Спин-блокировка предполагает, что защищенный ресурс не бывает занят надолго. И тогда эффективнее делать так: выполнять цикл, переходить в режим ядра и ждать. Многие разработчики повторяют цикл некоторое число раз (скажем, 4000) и, если ресурс к тому времени не освободился, переводят поток в режим ядра, где он спит, ожидая освобождения ресурса (и не расходуя процессорное время). По такой схеме реализуются критические секции (critical sections).

Спин-блокировка полезна на многопроцессорных машинах, где один поток может «крутиться» в цикле, а второй — работать на другом процессоре. Но даже в таких условиях надо быть осторожным. Вряд ли вам понравится, если поток надолго войдет в цикл, ведь тогда он будет впустую тратить процессорное время. О спин-блокировке мы еще поговорим в этой главе.

Последняя пара Interlocked-функций выглядит так:

```
PV0ID InterlockedCompareExchange(
    PL0NG plDestination,
    LONG lExchange,
    LONG lComparand);

PV0ID InterlockedCompareExchangePointer(
    PV0ID* ppvDestination,
    PV0ID pvExchange,
    PV0ID pvComparand);
```

Они выполняют операцию сравнения и присвоения на уровне атомарного доступа. В 32-разрядном приложении обе функции работают с 32-разрядными значениями, но в 64-разрядном приложении *InterlockedCompareExchange* используется для 32-разрядных значений, а *InterlockedCompareExchangePointer* — для 64-разрядных. Вот как они действуют, если представить это в псевдокоде:

```
LONG InterlockedCompareExchange(PLONG plDestination,
    LONG lExchange, L0NQ lComparand) {

    L0Nfi lRet = *plDestination;            // исходное значение

    if (*plDestination == lComparand)
      *plDestination = lExchange;
    return(lRet);
}
```

Функция сравнивает текущее значение переменной типа LONG (на которую указывает параметр **plDestination*) со значением, передаваемым в параметре *lComparand*. Если значения совпадают, **plDestination* получает значение параметра *lExchange*; в ином случае **plDestination* остается без изменений. Функция возвращает исходное значение **plDestination*. И не забывайте, что все эти действия выполняются как единая атомарная операция.

```
L0NGL0NG InterlockedCompareExchange64(
    LONGLONG pllDestination,
    L0NGL0NG llExchange,
    LONGLONG llComparand);
```

Обратите внимание на отсутствие *Interlocked*-функции, позволяющей просто считывать значение какой-то переменной, не меняя его. Она и не нужна. Если один поток модифицирует переменную с помощью какой-либо *Interlocked*-функции в тот момент, когда другой читает содержимое той же переменной, ее значение, прочитанное вторым потоком, всегда будет достоверным. Он получит либо исходное, либо измененное значение переменной. Поток, конечно, не знает, какое именно значение он считал, но главное, что оно корректно и не является некоей произвольной величиной. В большинстве приложений этого вполне достаточно. *Interlocked*-функции можно также использовать в потоках различных процессов для синхронизации доступа к переменной, которая находится в разделяемой области памяти, например в проекции файла. (Правильное применение *Interlocked*-функций демонстрирует несколько программ-примеров из главы 9.)

В Windows есть и другие функции из этого семейства, но ничего нового по сравнению с тем, что мы уже рассмотрели, они не делают. Вот еще две из них:

```
LONG InterlockedIncrement(PLONG plAddend);

LONG InterlockedDecrement(PLONG plAddend);
```

InterlockedExchangeAdd полностью заменяет обе эти устаревшие функции. Новая функция умеет добавлять и вычитать произвольные значения, а функции *InterlockedIncrement* и *InterlockedDecrement* увеличивают и уменьшают значения только на 1.

Поддерживается также набор вспомогательных Interlocked-функций OR, AND и XOR, основанных на InterlockedCompareExchange64. В их реализации (см. файл WinBase.h) показанные выше спин-блокировки используются следующим образом:

```
LONGLONG InterlockedAnd64(
    LONGLONG* Destination,
    LONGLONG Value) {
    LONGLONG Old;

    do {
        Old = *Destination;
    } while (InterlockedCompareExchange64(Destination, Old & Value, Old) != Old);

    return Old;
}
```

В Windows XP и последующих версиях к вышеописанным атомарным операциям над целочисленными и булевыми значениями добавлен ряд функций, позволяющих без особого труда манипулировать стеком, известным также как *InterlockedSinglyLinkedList* (однонаправленный список с взаимоблокировкой). Все операции над таким стеком, такие как заталкивание в него значений и извлечение значений выполняются атомарно. Соответствующие функции перечислены в табл. 8-1.

Табл. 8-1. Функции для работы с однонаправленными списками с взаимоблокировкой

Функция	Описание
InitializeSListHead	Создает пустой стек
InterlockedPushEntrySList	Добавляет элемент в начало стека
InteHockedPopEntrySLisi	Возвращает элемент стека, попутно удаляя его
InterlockedFhishSList	Очищает стек
QueryDepthSList	Возвращает число элементов стека

Кэш-линии

Если вы хотите создать высокоэффективное приложение, работающее на многопроцессорных машинах, то просто обязаны уметь пользоваться кэш-линиями процессора (CPU cache lines). Когда процессору нужно считать из памяти один байт, он извлекает не только его, но и столько смежных байтов, сколько требуется для заполнения кэш-линии. Такие линии состоят из 32 или 64 байтов (в зависимости от типа процессора) и всегда выравниваются по границам, кратным 32 или 64 байтам. Кэш-линии предназначены для повышения быстродействия процессора. Обычно приложение работает с набором смежных байтов, и, если эти байты уже находятся в кэше, процессору не приходится снова обращаться к шине памяти, что обеспечивает существенную экономию времени.

Однако кэш-линии сильно усложняют обновление памяти в многопроцессорной среде. Вот небольшой пример:

1. Процессор 1 считывает байт, извлекая этот и смежные байты в свою кэш-линию.
2. Процессор 2 считывает тот же байт, а значит, и тот же набор байтов, что и процессор 1; извлеченные байты помещаются в кэш-линию процессора 2.
3. Процессор 1 модифицирует байт памяти, и этот байт записывается в его кэш-линию. Но эти изменения еще не записаны в оперативную память.
4. Процессор 2 повторно считывает тот же байт. Поскольку он уже помещен в кэш-линию этого процессора, последний не обращается к памяти и, следовательно, не «видит» новое значение данного байта.

Такой сценарий был бы настоящей катастрофой. Но разработчики чипов прекрасно осведомлены об этой проблеме и учитывают ее при проектировании своих процессоров. В частности, когда один из процессоров модифицирует байты в своей кэш-линии, об этом оповещаются другие процессоры, и содержимое их кэш-линий объявляется недействительным. Таким образом, в примере, приведенном выше, после изменения байта процессором 1, кэш процессора 2 был бы объявлен недействительным. На этапе 4 процессор 1 должен сбросить содержимое своего кэша в оперативную память, а процессор 2 — повторно обратиться к памяти и вновь заполнить свою кэш-линию. Как видите, кэш-линии, которые, как правило, увеличивают быстродействие процессора, в многопроцессорных машинах могут стать причиной снижения производительности.

Все это означает, что вы должны группировать данные своего приложения в блоки размером с кэш-линии и выравнивать их по тем же правилам, которые применяются к кэш-линиям. Ваша цель — добиться того, чтобы различные процессоры обращались к разным адресам памяти, отделенным друг от друга по крайней мере границей кэш-линии. Кроме того, вы должны отделить данные «только для чтения» (или редко используемые данные) от данных «для чтения и записи». И еще вам придется позаботиться о группировании тех блоков данных, обращение к которым происходит примерно в одно и то же время.

Вот пример плохо продуманной структуры данных:

```
struct CUSTINF0 {
    DWORD     dwCustomerID;                 // в основном "только для чтения"
    int       nBalanceDue;                  // для чтения и записи
    wchar_t   szName[100];                  // в основной "только для чтения"
    FILETIME  ftLastOrderDate;              // для чтения и записи
};
```

А это усовершенствованная версия той же структуры:

```
#define CACHE_ALIGN  64

// принудительно помещаем следующие элементы в другую кэш-линию
struct __declspec(align(CACHE_ALIGN)) CUSTINF0 {
    DWORD    dwCustomerID;                  // в основном "только для чтения"
    wchar_t  szName[100];                   // в основном "только для чтения"

    // принудительно помещаем следующие элементы в другую кэш-линию
    _declspac(align(CAGHE_ALIGN))
    int nBalanceDue;                        // для чтения и записи
    FILETIME ftLastOrderDate;               // для чтения и записи
};
```

Проще всего определить размер кэш-линии процессора вызовом Win32-функции *GetLogicalProcessorInformation*. Эта функция возвращает массив

структур SYSTEM_LOGICAL_PROGESS0R_INFORMATION. Узнать размер кэш-линии можно, проанализировав поле Cache этой структуры, содержащее ссылку на структуру CACHE_DESCRIPTOR, в которой, в свою очередь, имеется поле *LineSize*, представляющее размер кэш-линии. Получив эту информацию, можно управлять выравниванием полей с помощью директивы *__decfapec(align(#))* компилятора C/C++. Подробнее об использовании *__declspec(align(#))*, см. по ссылке http://msdn2.microsoftcom/en^is/library/83ythb65&spx.

> **Примечание**. Лучше всего, когда данные используются единственным потоком (самый простой способ добиться этого — применять параметры функций и локальные переменные) или одним процессором (это реализуется привязкой потока к определенному процессору). Если вы пойдете по такому пути, можете вообще забыть о проблемах, связанных с кэш-линиями.

Более сложные методы синхронизации потоков

Interlocked-функции хороши, когда требуется монопольно изменить всего одну переменную. С них и надо начинать. Но реальные программы имеют дело со структурами данных, которые гораздо сложнее единственной 32-или 64-битной переменной. Чтобы получить доступ на атомарном уровне к таким структурам данных, забудьте об *Interlocked*-функциях и используйте другие механизмы, предлагаемые Windows.

В предыдущем разделе я подчеркнул неэффективность спин-блокировки на однопроцессорных машинах и обратил ваше внимание на то, что со спин-блокировкой надо быть осторожным даже в многопроцессорных системах. Хочу еще раз напомнить, что основная причина связана с недопустимостью пустой траты процессорного времени. Так что нам нужен механизм, который позволил бы потоку, ждущему освобождения разделяемого ресурса, не расходовать процессорное время.

Когда поток хочет обратиться к разделяемому ресурсу или получить уведомление о некоем «особом событии», он должен вызвать определенную функцию операционной системы и передать ей параметры, сообщающие, чего именно он ждет. Как только операционная система обнаружит, что ресурс освободился или что «особое событие» произошло, эта функция вернет управление потоку, и тот снова будет включен в число планируемых. (Это не значит, что поток тут же начнет выполняться; система подключит его к процессору по правилам, описанным в предыдущей главе.)

Пока ресурс занят или пока не произошло «особое событие», система переводит поток в ждущий режим, исключая его из числа планируемых, и берет на себя роль агента, действующего в интересах спящего потока. Она

выведет его из ждущего режима, когда освободится нужный ресурс или произойдет «особое событие».

Большинство потоков почти постоянно находится в ждущем режиме. И когда система обнаруживает, что все потоки уже несколько минут спят, срабатывает механизм управления электропитанием.

Худшее, что можно сделать

Если бы синхронизирующих объектов не было, а операционная система не умела отслеживать особые события, потоку пришлось бы самостоятельно синхронизировать себя с ними, применяя метод, который я как раз и собираюсь продемонстрировать. Но поскольку в операционную систему встроена поддержка синхронизации объектов, никогда не применяйте этот метод.

Суть его в том, что поток синхронизирует себя с завершением какой-либо задачи в другом потоке, постоянно просматривая значение переменной, доступной обоим потокам. Возьмем пример:

```
volatile BOOL g_fFinishedCalculation = FALSE;

int WINAPI _tWinMain(…) {
    CreateThread(…, RecalcFunc, …);
    …
    // ждем завершения пересчета
    while (!g_fFinishedCalculation)
        ;
    …
}

DWORD WINAPI RecalcFunc(PVOID pvParam) {
    // выполняем пересчет
    … g_fFihlshedCalculation = TRUE;
    return(0);
}
```

Как видите, первичный поток (он исполняет функцию *_tWinMain*) при синхронизации по такому событию, как завершение функции *RecalcFunc*, никогда не впадает в спячку. Поэтому система по-прежнему выделяет ему процессорное время за счет других потоков, занимающихся чем-то более полезным.

Другая проблема, связанная с подобным методом опроса, в том, что булева переменная *g_fFinishedCalculation* может не получить значения TRUE — например, если у первичного потока более высокий приоритет, чем у потока, выполняющего функцию *RecalcFunc*. В этом случае система никогда не предоставит процессорное время потоку *RecalcFunc*, а он никогда не выполнит оператор, присваивающий значение TRUE переменной *g_fFinishedCalculation*. Если бы мы не опрашивали поток, выполняющий функцию *_tWinMain*, а просто отправили в спячку, это позволило бы системе от-

дать его долю процессорного времени потокам с более низким приоритетом, в частности потоку *RecalcFunc*.

Вполне допускаю, что опрос иногда удобен. В конце концов, именно это и делается при спин-блокировке. Но есть два способа его реализации: корректный и некорректный. Общее правило таково: избегайте применения спин-блокировки и опроса. Вместо этого пользуйтесь функциями, которые переводят ваш поток в состояние ожидания до освобождения нужного ему ресурса. Как это правильно сделать, я объясню в следующем разделе.

Прежде всего, позвольте обратить ваше внимание на одну вещь: в начале приведенного выше фрагмента кода я использовал спецификатор *volatile* — без него работа моей программы просто немыслима. Он сообщает компилятору, что переменная может быть изменена извне приложения — операционной системой, аппаратным устройством или другим потоком. Точнее, спецификатор *volatile* заставляет компилятор исключить эту переменную из оптимизации и всегда перезагружать ее значение из памяти. Представьте, что компилятор сгенерировал следующий псевдокод для оператора *while* из предыдущего фрагмента кода:

```
NOV    RegO, [g_fFinishedCalculation]   ; копируем значение в регистр
Label: TEST  RegO, 0                    ; равно ли оно нулю?
JMP    RegO == 0, Label                 ; в регистре находится 0, повторяем цикл
                                        ; в регистре находится ненулевое значение
```

Если бы я не определил булеву переменную как *volatile*, компилятор мог бы оптимизировать наш код на C именно так. При этом компилятор загружал бы ее значение в регистр процессора только раз, а потом сравнивал бы искомое значение с содержимым регистра. Конечно, такая оптимизация повышает быстродействие, поскольку позволяет избежать постоянного считывания значения из памяти; оптимизирующий компилятор, скорее всего, сгенерирует код именно так, как я показал. Но тогда наш поток войдет в бесконечный цикл и никогда не проснется. Кстати, если структура определена как volatile, таковыми становятся и все ее элементы, т. е. при каждом обращении они считываются из памяти.

Вас, наверное, заинтересовало, а не следует ли объявить как *volatile* и мою переменную *g_fResourceInUse* в примере со спин-блокировкой. Отвечаю: нет, потому что она передается *Interlocked*-функции по ссылке, а не по значению. Передача переменной по ссылке всегда заставляет функцию считывать ее значение из памяти, и оптимизатор никак не влияет на это.

Критические секции

Критическая секция (critical section) — это небольшой участок кода, требующий монопольного доступа к каким-то общим данным. Она позволяет

сделать так, чтобы единовременно только один поток получал доступ к определенному ресурсу. Естественно, система может в любой момент вытеснить ваш поток и подключить к процессору другой, но ни один из потоков, которым нужен занятый вами ресурс, не получит процессорное время до тех пор, пока ваш поток не выйдет за границы критической секции.

Вот пример кода, который демонстрирует, что может произойти без критической секции:

```
const int COUNT = 1000;
int   g_nSum = 0;

DWORD WINAPI FirstThread(PVOID pvParam) {
    g_nSum = 0;
    for (int n - 1; n <= COUNT; n++) {
      g_nSum += n;
    }
    return(g_nSum);
}

DWORD WINAPI SecondThread(PV0ID pvParam) {
    g_nSum = 0;
    for (int i = 1; n <= COUNT; n++) {
      g_nSum += n;
    }
    return(g_nSum);
}
```

Здесь предполагается, что функции обоих потоков дают одинаковый результат, хоть они и закодированы с небольшими различиями. Если бы исполнялась только функция *FirstThread*, она бы сложила все числа от 0 до значения COUNT. Это верно и в отношении *SecondThread* — если бы она тоже исполнялась независимо. Но в нашем коде возникает проблема: функции обоих потоков (возможно, работающих на разных процессорах) одновременно обращаются к одной и той же глобальной переменной (*g_nSum*), значение которой в результате изменяется непредсказуемым образом.

Согласен, пример довольно надуманный (к тому же сумму легко вычислить и без использования цикла, например, так: *g_nSum* = COUNT * (COUNT + 1) / 2), но, чтобы привести реалистичный, нужно минимум несколько страниц кода. Важно другое: теперь вы легко представите, что может произойти в действительности. Возьмем пример с управлением связанным списком объектов. Если доступ к связанному списку не синхронизирован, один поток может добавить элемент в список в тот момент, когда другой поток пытается найти в нем какой-то элемент. Ситуация станет еще более угрожающей, если оба потока одновременно добавят в список новые элементы. Так что, используя критические секции, можно и нужно координировать доступ потоков к структурам данных.

Теперь, когда вы видите все «подводные камни», попробуем исправить этот фрагмент кода с помощью критической секции:

```
const int COUNT = 10;
int g_nSum = 0;
CRITICAL_SECTION g_cs;

DW0R0 WINAPI FirstThread(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    g_nSum = 0;
    for (int n = 1; n <= C0UNT; n++) {
      g_nSum += n;
    }
    LeaveCriticalSection(&g_cs);
    return(g_nSum);
}

DWORD WINAPI SecondThread(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    g_nSum = 0;
    for (int n = 1; n <= COUNT; n++) {
      g_nSum += n;
    }
    LeaveCriticalSection(&g_cs);
    return(g_nSum);
}
```

Я создал экземпляр структуры данных CRITICAL_SECTION — *g_cs*, а потом «обернул» весь код, работающий с разделяемым ресурсом (в нашем примере это строки *cg_nSum*), вызовами *EnterCriticalSection* и *LeaveCriticalSection*. Заметьте, что при вызовах этих функций я передаю адрес *g_cs*.

Запомните несколько важных вещей. Если у вас есть ресурс, разделяемый несколькими потоками, вы должны создать экземпляр структуры CRITICAL_SECTION. Так как я пишу эти строки в самолете, позвольте провести следующую аналогию. Структура CRITICAL_SECTION похожа на туалетную кабинку в самолете, а данные, которые нужно защитить, — на унитаз. Туалетная кабинка (критическая секция) в самолете очень маленькая, и единовременно в ней может находиться только один человек (поток), пользующийся унитазом (защищенным ресурсом).

Если у вас есть ресурсы, всегда используемые вместе, вы можете поместить их в одну кабинку — единственная структура CRITICAL_SECTION будет охранять их всех. Но если ресурсы не всегда используются вместе (например, потоки 1 и 2 работают с одним ресурсом, а потоки 1 и 3 — с другим), вам придется создать им по отдельной кабинке, или структуре CRITICAL_SECTION.

Теперь в каждом участке кода, где вы обращаетесь к разделяемому ресурсу, вызывайте *EnterCriticalSection*, передавая ей адрес структуры CRITICAL_

SECTION, которая выделена для этого ресурса. Иными словами, поток, желая обратиться к ресурсу, должен сначала убедиться, нет ли на двери кабинки знака «занято». Структура CRITICAL_SECTION идентифицирует кабинку, в которую хочет войти поток, а функция *EnterCriticalSection* — тот инструмент, с помощью которого он узнает, свободна или занята кабинка. *EnterCriticalSection* допустит вызвавший ее поток в кабинку, если определит, что та свободна. В ином случае (кабинка занята) *EnterCriticalSection* заставит его ждать, пока она не освободится.

Поток, покидая участок кода, где он работал с защищенным ресурсом, должен вызвать функцию *LeaveCriticalSection*. Тем самым он уведомляет систему о том, что кабинка с данным ресурсом освободилась. Если вы забудете это сделать, система будет считать, что ресурс все еще занят, и не позволит обратиться к нему другим ждущим потокам. То есть вы вышли из кабинки и оставили на двери знак «занято».

> **Примечание**. Самое сложное — запомнить, что любой участок кода, работающего с разделяемым ресурсом, нужно заключить в вызовы функций *EnterCriticalSection* и *LeaveCriticalSection*. Если вы забудете сделать это хотя бы в одном месте, ресурс может быть поврежден. Так, если в *FirstThread* убрать вызовы *EnterCriticalSection* и *LeaveCriticalSection*, содержимое переменной *g_nSum* станет некорректным — даже несмотря на то что в *SecondThread* функции *EnterCriticalSection* и *LeaveCriticalSection* вызываются правильно.
>
> Забыв вызвать эти функции, вы уподобитесь человеку, который рвется в туалетную кабинку, не обращая внимания на то, есть в ней кто-нибудь или нет. Поток пробивает себе путь к ресурсу и берется им манипулировать. Как вы прекрасно понимаете, стоит лишь одному потоку проявить такую «грубость», и ваш ресурс станет кучкой бесполезных байтов.

Применяйте критические секции, если вам не удается решить проблему синхронизации за счет *Interlocked*-функций. Преимущество критических секций в том, что они просты в использовании и выполняются очень быстро, так как реализованы на основе *Interlocked*-функций. А главный недостаток — нельзя синхронизировать потоки в разных процессах.

Критические секции: важное дополнение

Теперь, когда у вас появилось общее представление о критических секциях (зачем они нужны и как с их помощью можно монопольно распоряжаться разделяемым ресурсом), давайте повнимательнее приглядимся к тому, как они устроены. Начнем со структуры CRITICAL_SECTION. Вы не найдете ее в Platform SDK — о ней нет даже упоминания. В чем дело?

Хотя CRITICAL_SECTION задокументирована, Microsoft полагает, что вам незачем знать, как она устроена. И это правильно. Для нас она является своего рода чёрным ящиком: сама структура известна, а ее элементы —

нет. Конечно, поскольку CRITICAL_SECTION — не более чем одна из структур, мы можем сказать, из чего она состоит, изучив заголовочные файлы. (CRITICAL_SECTION определена в файле WinBase.h как RTL_CRITICAL_SECTION, а тип структуры RTL_CRITICAL_SECTION определен в файле WinNT.h). Но никогда не пишите код, прямо ссылающийся на ее элементы.

Вы работаете со структурой CRITICAL_SECTION исключительно через функции Windows, передавая им адрес соответствующего экземпляра этой структуры. Функции сами знают, как обращаться с ее элементами, и гарантируют, что она всегда будет в согласованном состоянии. Так что теперь мы перейдем к рассмотрению этих функций.

Обычно структуры CRITICAL_SECTION создаются как глобальные переменные, доступные всем потокам процесса. Но ничто не мешает нам создавать их как локальные переменные или переменные, динамически размещаемые &куче. Есть только два условия, которые надо соблюдать. Во-первых, все потоки, которым может понадобиться ресурс, должны знать адрес структуры CRITICAL_SECTION, которая защищает этот ресурс. Вы можете получить ее адрес, используя любой из существующих механизмов. Во-вторых, элементы структуры CRITICAL_SECTION следует инициализировать до обращения какого-либо потока к защищенному ресурсу. Структура инициализируется вызовом:

```
VOID InitializeCriticalSection(PCRITICAL_SECTION pcs);
```

Эта функция инициализирует элементы структуры CRITICAL_SECTION, на которую указывает параметр *pcs*. Поскольку вся работа данной функции заключается в инициализации нескольких переменных-членов, она не дает сбоев и поэтому ничего не возвращает (void). *InitializeCriticalSection* должна быть вызвана до того, как один из потоков обратится к *EnterCriticalSection*. В документации Platform SDK недвусмысленно сказано, что попытка воспользоваться неинициализированной критической секцией даст непредсказуемые результаты.

Если вы знаете, что структура CRITICAL_SECTION больше не понадобится ни одному потоку, удалите ее, вызвав *DeleteCriticalSection*:

```
VOID DeleteCriticalSection(PCRITICAL_SECTION pcs);
```

Она сбрасывает все переменные-члены внутри этой структуры. Естественно, нельзя удалять критическую секцию в тот момент, когда ею все еще пользуется какой-либо поток. Об этом нас предупреждают и в документации Platform SDK.

Участок кода, работающий с разделяемым ресурсом, предваряется вызовом:

```
VOI0 EnterCriticalSection(PCRITICAL_SECTION pcs);
```

Первое, что делает *EnterCriticalSection,* — исследует значения элементов структуры CRITICAL_SECTION. Если ресурс занят, в них содержатся сведения о том, какой поток пользуется ресурсом. *EnterCriticalSection* выполняет следующие действия.

- Если ресурс свободен, *EnterCriticalSection* модифицирует элементы структуры, указывая, что вызывающий поток занимает ресурс, после чего немедленно возвращает управление, и поток продолжает свою работу (получив доступ к ресурсу).
- Если значения элементов структуры свидетельствуют, что ресурс уже захвачен вызывающим потоком, *EnterCriticalSection* обновляет их, отмечая тем самым, сколько раз подряд этот поток захватил ресурс, и немедленно возвращает управление. Такая ситуация бывает нечасто — лишь тогда, когда поток два раза подряд вызывает *EnterCriticalSection* вез промежуточного вызова *LeaveCriticalSection*.
- Если значения элементов структуры указывают на то, что ресурс занят другим потоком, *EnterCriticalSection* переводит вызывающий поток в режим ожидания. Это потрясающее свойство критических секций: поток, пребывая в ожидании, не тратит ни кванта процессорного времени! Система запоминает, что данный поток хочет получить доступ к ресурсу, и — как только поток, занимавший этот ресурс, вызывает *LeaveCriticalSection* — вновь начинает выделять нашему потоку процессорное время. При этом она передает ему ресурс, автоматически обновляя элементы структуры CRITICAL_SECTION.

Внутреннее устройство *EnterCriticalSection* не слишком сложно; она выполняет лишь несколько простых операций. Чем она действительно ценна, так это способностью выполнять их на уровне атомарного доступа. Даже если два потока на многопроцессорной машине одновременно вызовут *EnterCriticalSection*, функция все равно корректно справится со своей задачей: один поток получит ресурс, другой — перейдет в ожидание.

Поток, переведенный *EnterCriticalSection* в ожидание, может надолго лишиться доступа к процессору, а в плохо написанной программе — даже вообще не получить его. Когда именно так и происходит, говорят, что поток «голодает».

> **Примечание.** В действительности потоки, ожидающие освобождения критической секции, никогда не блокируются «навечно». *EnterCriticalSection* устроена так, что по истечении определенного времени генерирует исключение. После этого вы можете подключить к своей программе отладчик и посмотреть, что в ней случилось. Длительность времени ожидания функцией EnterCriticalSection определяется значением параметра CriticalSectionTimeout, который хранится в следующем разделе системного реестра:
>
> **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager**

Длительность времени ожидания измеряется в секундах и по умолчанию равна 2 592 000 секунд (что составляет ровно 30 суток). Не устанавливайте слишком малое значение этого параметра (например, менее 3 секунд), так как иначе вы нарушите работу других потоков и приложений, которые обычно ждут освобождения критической секции дольше трех секунд.

Вместо *EnterCriticalSection* вы можете воспользоваться:

```
B00L TryEnterCriticalSection(PCRITICAL_SECTION pcs);
```

Эта функция никогда не приостанавливает выполнение вызывающего потока. Но возвращаемое ею значение сообщает, получил ли этот поток доступ к ресурсу. Если при ее вызове указанный ресурс занят другим потоком, она возвращает FALSE. Во всех остальных случаях возвращается TRUE.

TryEnterCriticalSection позволяет потоку быстро проверить, доступен ли ресурс, и, если нет, заняться чем-нибудь другим. Если функция возвращает TRUE, значит, она обновила элементы структуры CRITICAL_SECTION так, чтобы они сообщали о захвате ресурса вызывающим потоком. Отсюда следует, что для каждого вызова функции *TryEnterCriticalSection*, где она возвращает TRUE, надо предусмотреть парный вызов *LeaveCriticalSection*.

В конце участка кода, использующего разделяемый ресурс, должен присутствовать вызов:

```
VOID LeaveCriticalSection(PCRITICAL_SECTION pcs);
```

Эта функция просматривает элементы структуры CRITICAL_SECTION и уменьшает счетчик числа захватов ресурса вызывающим потоком на 1. Если его значение больше 0, *LeaveCriticalSection* ничего не делает и просто возвращает управление.

Если значение счетчика достигло 0, *LeaveCriticalSection* сначала выясняет, есть ли в системе другие потоки, ждущие данный ресурс в вызове *EnterCriticalSection*. Если есть хотя бы один такой поток, функция настраивает значения элементов структуры, чтобы они сигнализировали о занятости ресурса, и отдает его одному из ждущих потоков (поток выбирается «по справедливости»). Если же ресурс никому не нужен, *LeaveCriticalSection* соответственно сбрасывает элементы структуры. Как и *EnterCriticalSection*, функция *LeaveCriticalSection* выполняет все действия на уровне атомарного доступа. Однако *LeaveCriticalSection* никогда не приостанавливает поток, а управление возвращает немедленно.

Критические секции и спин-блокировка

Когда поток пытается войти в критическую секцию, занятую другим потоком, он немедленно приостанавливается. А это значит, что поток переходит из пользовательского режима в режим ядра (на что затрачивается около 1000 тактов процессора). Цена такого перехода чрезвычайно высока. На много-

процессорной машине поток, владеющий ресурсом, может выполняться на другом процессоре и очень быстро освободить ресурс. Тогда появляется вероятность, что ресурс будет освобожден еще до того, как вызывающий поток завершит переход в режим ядра. В итоге уйма процессорного времени будет потрачена впустую.

Microsoft повысила быстродействие критических секций, включив в них спин-блокировку. Теперь, когда вы вызываете *EnterCriticalSection*, она выполняет заданное число циклов спин-блокировки, пытаясь получить доступ к ресурсу. И лишь в том случае, когда все попытки заканчиваются неудачно, функция переводит поток в режим ядра, где он будет находиться в состоянии ожидания.

Для использования спин-блокировки в критической секции нужно инициализировать счетчик циклов, вызвав:

```
BOOL InitializeCriticalSectionAndSpinCount(
    PCRITICAL_SECnON pcs,
    DWORD dwSpinCount);
```

Как и в *InitializeCriticalSection*, первый параметр этой функции — адрес структуры критической секции. Но во втором параметре, *dwSpinCount*, передается число циклов спин-блокировки при попытках получить доступ к ресурсу до перевода потока в состояние ожидания. Этот параметр может принимать значения от 0 до 0x00FFFFFF. Учтите, что на однопроцессорной машине значение параметра *dwSpinCount* игнорируется и считается равным 0. Дело в том, что применение спин-блокировки в такой системе бессмысленно: поток, владеющий ресурсом, не сможет освободить его, пока другой поток «крутится» в циклах спин-блокировки.

Вы можете изменить счетчик циклов спин-блокировки вызовом:

```
DWORD SetCriticalSectionSpinCount(
    PCRITICAL_SECTION pcs,
    DWORD dwSpinCount);
```

И в этой функции значение *dwSpinCount* на однопроцессорной машине игнорируется.

На мой взгляд, используя критические секции, вы должны всегда применять спин-блокировку — терять вам просто нечего. Могут возникнуть трудности в подборе значения *dwSpinCount*, но здесь нужно просто поэкспериментировать. Имейте в виду, что для критической секции, стоящей на страже динамической кучи вашего процесса, этот счетчик равен 4000.

Критические секции и обработка ошибок

Вероятность того, что *InitializeCriticalSection* потерпит неудачу, крайне мала, но все же существует. В свое время Microsoft не учла этого при разработке функции и определила ее возвращаемое значение как VOID, т. е. она ничего не возвращает. Однако функция может потерпеть неудачу, так как выделяет

блок памяти для внутрисистемной отладочной информации. Если выделить память не удается, генерируется исключение STATUS_NO_MEMORY. Вы можете перехватить его, используя структурную обработку исключений (см. главы 23,24 и 25).

Есть и другой, более простой способ решить эту проблему — перейти на новую функцию *InitializeCriticalSectionAndSpinCount*. Она, тоже выделяя блок памяти для отладочной информации, возвращает FALSE, если выделить память не удается.

В работе с критическими секциями может возникнуть еще одна проблема. Когда за доступ к критической секции конкурирует два и более потока, она использует объект ядра «событие». Поскольку такая конкуренция маловероятна, система не создает объект ядра «событие» до тех пор, пока он действительно не потребуется. Это экономит массу системных ресурсов — в большинстве критических секций конкуренция потоков никогда не возникает. Кстати, этот объект ядра освобождается только при вызове *DeleteCriticalSection*. Так что не забывайте вызывать эту функцию, закончив операции в критической секции.

Раньше (до Windows XP) в случаях, когда в условиях нехватки памяти конкуренция потоков за критическую секцию все же возникала, системе не удавалось создать нужный объект ядра. И тогда *EnterCriticalSection* генерировала исключение EXCEPTION_INVALID_HANDLE. Большинство разработчиков просто игнорирует вероятность такой ошибки, и не предусматривает для нее никакой обработки, поскольку она случается действительно очень редко. Но если вы хотите заранее подготовиться к такой ситуации, у вас есть две возможности.

Первая — использовать структурную обработку исключений и перехватывать ошибку. При этом вы либо отказываетесь от обращения к ресурсу, защищенному критической секцией, либо дожидаетесь появления свободной памяти, а затем повторяете вызов *EnterCriticalSection*.

Вторая возможность заключается в том, что вы создаете критическую секцию вызовом *InitializeCriticalSectionAndSpinCount*, передавая параметр *dwSpinCount* с установленным старшим битом. Тогда функция создает объект «событие» и сопоставляет его с критической секцией. Если создать объект не удается, она возвращает FALSE, и это позволяет корректнее обрабатывать такие ситуации. Но успешно созданный объект ядра «событие» гарантирует вам, что *EnterCriticalSection* выполнит свою задачу при любых обстоятельствах и никогда не вызовет исключение. (Всегда выделяя память под объекты ядра «событие», вы неэкономно расходуете системные ресурсы. Поэтому делать так следует лишь в нескольких случаях, а именно: если программа может рухнуть из-за неудачного завершения функции *EnterCriticalSection*, если вы уверены в конкуренции потоков при обращении к критической секции или если программа будет работать в условиях нехватки памяти.)

В Windows XP появился новый тип объекта ядра «событие» — т. н. событие с ключом (*keyed event*). Оно предназначено как раз для решения

проблемы создания объектов в условиях нехватки памяти. Вместе с процессом операционная система всегда создает один такой объект, его легко найти с помощью утилиты Process Explorer от Sysinternals (см. http://www.micwsoft.com/technet/sysinternals/utilities/ProcessExplorer.mspx; ищите объект \KernelObjects\CritSecOutOfMemoryEvent). Этот недокументированный объект ядра во всем похож на обычный объект «событие» за исключением одного. Он способен синхронизировать различные группы потоков, которые определяются и блокируются с помощью ключа (по сути, указателя). Если из-за нехватки памяти потоку, вошедшему в критическую секцию, не удается создать объект «событие», в качестве ключа используется адрес критической секции. Так что потоки, пытающиеся войти в данную конкретную критическую секцию, будут синхронизированы и, при необходимости, блокированы на этом объекте «событие с ключом».

«Тонкая» блокировка

У *«тонкой» блокировки чтения и записи* (slim reader-writer lock), называемой также *SRWLock*, то же назначение, что и у обычной критической секции: защита ресурса от одновременного доступа разных потоков. Однако, в отличие от критической секции, *SRWLock* различает потоки, обращающиеся к ресурсу для чтения и для записи (т.е. читающие и записывающие потоки). *SRWLock* разрешает нескольким читающим потокам одновременно обращаться к ресурсу, поскольку чтение не грозит повреждением ресурса. Потребность в синхронизации возникает только при попытке потока модифицировать ресурс. В этом случае нужен монопольный доступ, то есть запрет обращения к ресурсу всем другим потокам, как записывающим, так и читающим. Именно это позволяет сделать *SRWLock*, причем без особого труда.

Сначала следует объявить структуру SRWLOCK и инициализировать ее вызовом *InitializeSRWLock*:

```
VOID InitializeSRWLock(PSRWLOCK SRWLock);
```

Структура SRWLOCK определена как RTL_SRWLOCK в файле WinBase.h. В WinNT.h объявление этой структуры содержит лишь указатель, ссылающийся на что-то другое, незадокументированное и потому (в отличие от полей CRITICAL_SECTION) недоступное для использования в коде.

```
typedef struct _RTL_SRWLOCK {
    PVOID Ptr;
} RTL_SRWLOCK, *PRTL_SRWLOCK;
```

После инициализации *SRWLock* записывающий поток может попытаться получить монопольный доступ к ресурсу, защищенному *SRWLock*, вызвав функцию AcquireSRWLockExclusive и передав ей в качестве параметра адрес объекта SRWLOCK:

```
VOID AcquireSRWLockExclusive(PSRWLOCK SRWLock);
```

После модификации ресурса блокировку освобождают вызовом *ReleaseSRWLockExclusive* с передачей в качестве параметра адреса объекта SRWLOCK:

```
VOID ReleaseSRWLockExclusive(PSRWLOCK SRWLock);
```

В случае читающего потока все аналогично, только используются другие функции:

```
VOID AcquireSRWLockShared(PSRWLOCK SRWLock);
VOID ReleaseSRWLockShared(PSRWLOCK SRWLock);
```

Вот и все. Функций для удаления и разрушения объектов SRWLOCK нет, поскольку система выполняет эти операции автоматически.

У *SRWLock* отсутствует ряд функций критической секции:

- Вызовы типа *TryEnter*(*Shared*/*Exclusive*)*SRWLock* невозможны, поскольку вызовы функций вида *AcquireSRWLock*(*Shared*/Exclusive) блокируют вызывающий поток, если у него уже есть блокировка.
- Невозможно также рекурсивное получение SRWLOCK, то есть один поток не может получить сразу несколько блокировок для многократной записи, а затем освободить их, вызвав соответствующее число раз функцию *ReleaseSRWLock**.

Впрочем, если вы смиритесь с этими ограничениями, то получите реальный прирост производительности, заменив критические секции блокировками *SRWLock*. Чтобы убедиться в том, что эти механизмы сильно различаются по скорости работы, запустите на многопроцессорной машине проект 08-UserSyncCompare, доступный на веб-сайте поддержки этой книги.

Эта простая программа порождает один, два или четыре потока, повторно исполняющих одну и ту же задачу с использованием различных механизмов синхронизации. Я прогнал все тесты на своем двухпроцессорном компьютере и записал затраченное время. Результаты показаны в табл. 8-2.

Табл. 8-2. Сравнение производительности различных механизмов синхронизации

Число потоков	Время (мс)						
	Чтение без синхронизации	Запись без синхронизации	Приращение (Interlocked)	SRWLock критическая секция	SRWLock (общий доступ)	(общий доступ)	Мьютекс
1	8	8	35	66	66	67	1060
2	8	76	153	268	134	148	11082
4	9	145	361	768	244	307	23785

В ячейках табл. 8-2 показано время (в мс; измерено с помощью класса *StopWatch*, показанного в главе 7), прошедшее от запуска потоков до завершения исполнения последним потоком 1000000 следующих операций:

- Чтение (без синхронизации) значения типа long:

```
LONG lValue = gv_value;
```

  Эта операция — самая быстрая, поскольку не требует синхронизации и кэш-память у разных процессоров работает независимо. Обычно такой тест занимает примерно одинаковое время независимо от числа процессоров или потоков.
- Запись (без синхронизации) значения типа long:

```
gv_value = 0;
```

  Если используется один поток, операция занимает столько же времени — 8 мс. Можно думать, что в случае двух потоков время исполнения этой операции просто удвоится, но на деле двухпроцессорный компьютер тратит на нее намного больше времени (76 мс). Причина — в необходимости взаимодействия процессоров для поддержания когерентности кэша. При увеличении числа потоков до четырех время операции увеличивается почти вдвое (до 145 мс) просто за счет удвоения нагрузки. Однако затраты времени увеличиваются не так сильно, как могло бы, поскольку для обработки данных используется лишь два процессора. Чем больше процессоров в компьютере, тем сильнее снижается быстродействие, поскольку с числом процессоров увеличивается и объем взаимодействия, необходимый для поддержания когерентности кэша у каждого из процессоров.
- Безопасное приращение long-значения с помощью функции *InterlockedIncrement*.

```
InterlockedIncrement(&gv_value);
```

  InterlockedIncrement работает медленнее, чем чтение-запись без синхронизации. Дело в том, что при этом процессор должен заблокировать доступ к соответствующей области памяти, чтобы предотвратить обращение к ней других процессоров. Использование двух потоков еще больше замедляет эту операцию, поскольку процессорам приходится обмениваться данными для поддержания когерентности кэша. В случае с четырьмя потоками быстродействие снижается пропорционально объему вычислений, которые, правда, выполняются двумя процессорами. Вероятно, на четырех-процессорной машине производительности снизится еще сильнее, поскольку в синхронизации кэша нуждаются уже четыре процессора.
- Чтение long-значения с использованием критической секции:

```
EnterCriticalSection(&g_cs);
gv_value = 0;
LeaveCriticalSection(&g_cs);
```

  Критические секции еще медленнее, поскольку поток должен входить и выходить из них, то есть выполнять две дополнительные операции. Кроме того, во время этих операций происходит модификация множес-

тва полей структуры CRITICAL_SECTION. При возникновении конкуренции работа критических секций замедляется еще сильнее, как видно из табл. 8-2. Так, обработка с четырьмя потоками занимает 768 мс, то есть затраченное время увеличивается более чем вдвое по сравнению с двумя потоками (268 мс) по причине переключения контекста, повышающего вероятность конкуренции.

- Чтение long-значения с помощью *SRWLock*:

```
AcquireSRWLockShared/Exclusive(&g_srwLock);
gv_value = 0;
ReleaseSRWLockShared/Exclusive(&g_srwLock);
```

Скорость чтения и записи с использованием *SRWLock* практически не отличается, если эти операции выполняет единственный поток. Производительность чтения с участием двух потоков при использовании *SRWLock* несколько выше, чем записи, поскольку оба потока могут читать ресурс одновременно, а записывать — только по очереди. При использовании четырех потоков чтение с использованием *SRWLock* работает заметно быстрее записи. Причина та же: все потоки могут читать ресурс одновременно. Результаты теста могут показаться неожиданно низкими (см. табл. 8-2), но в нем используется простой код, который мало что делает после установки блокировки. При этом поля блокировки и данные, которые она изменяет, непрерывно модифицируютсянесколькими потоками, в результате процессорам приходится обмениваться данными для синхронизации кэша.

- Использование синхронизирующего объекта ядра «мьютекс» (см. главу 9) для чтения long-значения:

```
WaitForSingleObject(g_hMutex, INFINITE);
gv_value = 0;
ReleaseMutex(g_hMutex);
```

Мьютексы намного медленнее остальных механизмов из-за необходимости ожидания этого объекта. Кроме того, для освобождения мьютексов поток должен каждый раз переключаться из режима пользователя в режим ядра и обратно, что отнимает уйму процессорного времени. Конкуренция, возникающая при наличии двух и более потоков, снижает производительность еще сильнее.

Производительность *SRWLock* вполне сравнима с производительностью критической секции. Более того, результаты теста показывают, что во многих случаях *SRWLock* работает быстрое критической секции, поэтому я рекомендую использовать именно ее. Кроме того, *SRWLock* поддерживает одновременное чтение ресурсов несколькими потоками, что повышает общую скорость работы и масштабируемость в отношении числа читающих потоков (как правило, их в приложениях большинство).

В заключение я кратко расскажу, как добиться максимальной производительности приложений. Для начала попытайтесь обойтись вовсе без разделяемых ресурсов. Если это невозможно, используйте (в порядке предпочтительности) чтение-запись без синхронизации, *interlocked*-функции, SRWLock-блокировки либо критические секции. И только если ни один из этих механизмов не подходит для ваших задач, прибегайте к синхронизирующим объектам ядра (о них — в следующей главе).

Условные переменные

Из сказанного выше ясно, что *SRWLock* применяют в случаях, когда возможен одновременный доступ к ресурсу читающих потоков (потоков-потребителей данных), а записывающему потоку (потоку-создателю данных) требуется только монопольный доступ. Возможны ситуации, когда данные для потока-потребителя еще не готовы, и тогда он должен освободить блокировку и ждать, пока поток-создатель не запишет в ресурс какие-нибудь данные. Если записывающий поток «под завязку» заполнил своими данными структуру, играющую роль приемника данных, он также должен освободить блокировку и «заснуть», пока читающие потоки не опустошат структуру-приемник.

Условные переменные (condition variables) облегчают жизнь разработчика при реализации сценариев, в которых необходимо заставить поток атомарно освободить блокировку ресурса, а после заблокировать его при помощи показанных ниже функций *SleepConditionVariableCS* или *SleepConditionVariableSRW*, пока не будет выполнено некоторое условие.

```
BOOL SleepConditionVariableCS(
    PCONDITION_VARIABLE pConditionVariable,
    PCRITICAL_SECTION pCriticalSection,
    DWORD dwMilllseconds);

BOOL SleepConditionVariableSRW(
    PCONDITION_VARIABLE pConditionVariable,
    PSRWLOCK pSRWLock,
    DWORD dwMilliseconds,
    ULONG Flags);
```

Параметр *pConditionVariable* — это указатель на инициализированную условную переменную, на которой ожидает поток. Второй параметр — указатель на критическую секцию либо *SRWLock*, применяемую для синхронизации доступа к разделяемому ресурсу. Параметр *dwMilliseconds* указывает, как долго вызывающий поток должен ждать момента, когда условная переменная примет нужное значение (ожидание будет бесконечным, если передать значение INFINITE). Параметр *Flags* второй функции определяет, какую блокировку следует установить, когда условная переменная получит значение, заданное условием. Чтобы установить монопольную блокировку

(для записывающего потока), передайте 0; чтобы разрешить совместный доступ (читающим потокам), передайте значение CONDITION_VARIABLE_LOCK-MODE_SHARED. Эти функции возвращают FALSE, если период ожидания истек, а условной переменной так и не присвоено заданное значение; в противном случае возвращается TRUE. Естественно, ни блокировка, ни критическая секция не создается, если функция вернула FALSE.

Поток, заблокированный в этих *Sleep*-функциях, пробуждается, если другой поток вызывает функцию *WakeConditionVariable* или *WakeAllConditionVariable* и выполненная в результате вызова проверка подтвердит выполнение некоторого условия. Для потока-потребителя таким условием может быть наличие данных, записанных потоком-создателем, а для потока-создателя — наличие свободного места в структуре-приемнике данных. Разберем, чем отличаются эти функции:

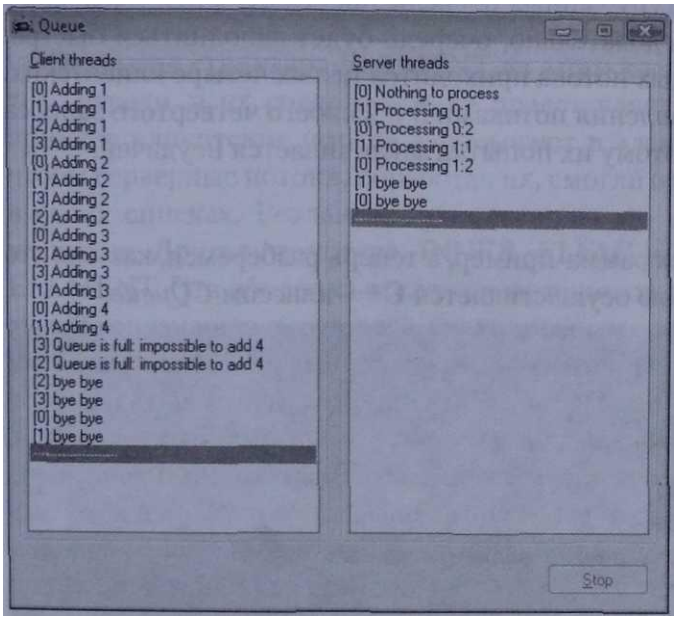
```
VOID WakeConditionVariable(
    PCONDITION_VARIABLE ConditionVariable);

VOID WakeAllConditionVariable(
    PCONDITION_VARIABLE ConditionVariable);
```

При вызове *WakeConditionVariable* поток, ожидающий переменной, заданной при вызове *SleepConditionVariable**, пробуждается с установкой соответствующей блокировки. Когда данный поток освободит эту блокировку, не проснется ни один из других потоков, спящих в ожидании той же самой условной переменной. Напротив, при вызове *WakeAllConditionVariable* просыпаются все потоки, ожидающие одной и той же условной переменной, заданной при вызове *SleepConditionVariable**. Это допустимо, поскольку в каждый момент времени только один записывающий поток может владеть монопольной блокировкой (если вы запросили такую блокировку). Либо, если в параметре *Flag* передано значение CONDITION_VARIABLE_LOCKMODE_SHARED, будет установлена блокировка, разрешающая всем потокам совместный доступ для чтения. Таким образом, в одних случаях просыпаются сразу все читающие потоки, либо сначала просыпается один читающий поток, затем один записывающий и т.д., пока каждый из заблокированных потоков не завладеет блокировкой. Те, кто работают с Microsoft .NET Framework, могут заметить сходство между классом Monitor и условными переменными. Оба механизма обеспечивают синхронизацию доступа за счет использования функций *SleepConditionVariable* / *Wait* и *Wake*ConditionVariable* / *Pulse (All)*. Подробнее о классе *Monitor* см. на сайте MSDN (http://msdn2.microsoft.com/en-us/library/hf5de04k.aspx) либо в моей книге *CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#, Русская редакция, Питер, 2006-2007*.

Приложение-пример Queue

Условные переменные всегда используют вместе с блокировками» реализованными с помощью критических секций либо *SRWLock*. В приложении

Queue (08-Queue.exe) для управления очередью запросов используется *SRWLock* и две условные переменные. Его исходный код и файлы ресурсов находятся в разделе 08-Queue на веб-сайте поддержки данной книги (см. выше). Если запустить это приложение и щелкнуть кнопку Stop, спустя некоторое время откроется следующее окно:



При инициализации Queue создается четыре клиентских потока (это записывающие потки) и два серверных (эти потоки являются читающими). Клиентские потоки помещают в очередь элементы-запросы, затем на некоторое время засыпают и, пробудившись, снова пытаются добавить запросы в очередь. После добавления элементов в очередь обновляется список Client Threads. Каждый элемент очереди содержит номер добавившего его клиентского потока. Например, первый элемент списка добавляет клиентский поток с номером 0, за ним добавляют свои элементы клиентские потоки 1,2 и 3, далее поток 0 помещает свой второй запрос и все повторяется.

Серверные потоки отвечают за обработку запросов. Серверный поток с номером 0 обрабатывает запросы с четными номерами, а поток номер 1 — запросы с нечетными номерами. Пока очередь пуста, потоки простаивают. Как только в очереди появляется хотя бы один элемент, пробуждается соответствующий серверный поток и обрабатывает его. При этом серверный поток помечает элемент очереди как прочитанный, уведомляет клиентские потоки о том, что они могут добавить в очередь новые элементы и засыпают до появления подходящих для него запросов.

Список Server Threads отображает состояние серверных потоков. Как видно из первой записи в списке, поток 0 пытался (безуспешно) найти запрос с четным номером. Вторая запись говорит, что серверный поток номер 1 обработал первый запрос, созданный клиентским потоком с номером 0.

Третья запись показывает, что серверный поток 0 обработал второй запрос клиентского потока 0 и т.д. После щелчка кнопки Stop потоки уведомляются о необходимости остановки и добавляют в свои списки записи «bye bye».

В этом примере серверные потоки недостаточно быстро обрабатывают клиентские запросы, поэтому через некоторое время очередь заполняется. Я инициализировал структуру данных очереди так, чтобы в ней умещалось не более 10 элементов, следовательно, очередь будет заполняться быстро. Кроме того, на два серверных потока приходится целых четыре клиентских. Видно, что к моменту добавления потоками 3 и 2 своего четвертого запроса очередь уже заполнена, поэтому их попытка заканчивается неудачей.

Реализация Queue

Вы увидели, что делает программа-пример, а теперь разберемся, как оно это делает. Управление очередью осуществляется C++-классом *CQueue*:

```
class CQueue
{
 public:
   struct ELEMENT {
       int  n_nTnreadNum;
       int  m_nRequestNum;
       // здесь должны быть другие элементы данных
   };
   typedef ELEMENT* PELEMENT;

 private:
   struct INNER_ELEMENT {
       int   m_nStamp;                      // 0 = пусто
       ELEMENT m_element;
   };
   typedef INNER_ELEMENT* PINNER_ELEMENT;

 private:
   PINNER_ELEMENT m_pElements;              // массив элементов, подлежащих
                                            // обработке

   int            m_nMaxElements;           // максимальное число элементов
                                            // в массиве
   int            m_nCurrentStamp;          // число добавленных элементов

 private:
   int GetFreeSlot();
   int GetNextSlot(int nThreadNum);

 public:
CQueue(int nMaxElements);
```

```
    ~CQueue();
    BOOL IsFull();
    BOOL IsEmpty(int nThreadNum);
    void AddElement(ELEMENT e);
    BOOL GetNewElement(int nThreadNum, ELEMENT& e);
}
```

Открытая структура ELEMENT из этого класса определяет вид элементов очереди, а их содержимое не представляет особой важности. В этом примере клиентские потоки записывают в элементы очереди свой номер, чтобы серверные потоки, обработав их, смогли отобразить эту информацию в своих списках. Реальным приложениям эта информация, как правило, не нужна. Другая структура, INNER_ELEMENT, является оболочкой для ELEMENT. Эта оболочка отслеживает порядок элементов с помощью поля *m_nStamp*, значение которого увеличивается при вставке каждого элемента. К закрытым членам структуры относится поле *m_pElements*, ссылающееся на массив структур INNER_ELEMENT фиксированного размера. Эти данные необходимо защитить от одновременного обращения клиентских и серверных потоков. Размер массива определяется значением, которое записывается в поле *m_nMaxElements* при создании объекта *CQueue*. Следующее поле, *m_nCurrentStamp*, содержит целочисленное значение, которое увеличивается каждый раз при добавлении элемента в очередь. Закрытая функция *GetFreeSlot* возвращает индекс первой структуры INNER_ELEMENT, хранящейся в поле *m_pElements* с *m_nStamp* = 0 (это означает, этот элемент уже прочитан либо пуст). Если подходящий элемент не найден, функция возвращает -1.

```
int CQueue::GetFreeSlot() {

    // поиск первого элемента с индексом 0
    for (int current = 0; current < m_nMaxElements; current++) {
        if (n_pElements[current].m_nStamp == 0)
           return(current);
    }

    // очередь заполнена
    return(-1);
}
```

Закрытая вспомогательная функция *GetNextSlot* возвращает индекс ft4NER_ELEMENT с наименьшим номером, отличным от нуля (то есть не пустого и не прочитанного еще элемента, добавленного первым) в массиве *m_pElements* of the INNER_ELEMENT. Если все элементы очереди были прочитаны (и получили отметку 0), возвращается -1.

```
int CQueue::GetNextSlot(int nThreadNum) {

   // по умолчанию данному потоку не разрешается добавлять элементы
   // в очередь
   int firstSlot = -1;

   // Индекс элемента не может быть больше, чем номер элемента,
   // добавленного последним
   int firstStamp = m_nCurrentStamp+1;

   // Поиск непрочитанных элементов с четными (для обработки потоком 0)

   // либо нечетными (для обработки потоком 1) номерами
   for (int current = 0; current < m_nMaxElements; current++) {

       // Следующий код нужен для отслеживания элементов, добавленными
       // первыми
       // (элементов с наименьшим номером), для реализации очереди типа
       // «первым вошел, первым вышел»
       if ((m_pElements[current].m_nStamp 1= 0) &&  // прочитанный
       // элемент
           ((m_pElements[current].m_element.m_nRequestNum % 2) == nThreadNum) &&
               (m_pElements[current].m_nStamp < firstStamp)) {

               firstStamp = m_pElements[current].m_nStamp;
               firstSlot = current;
       }
   }

   return(firstSlot);
}
```

Думаю, вы и сами без труда разберетесь в устройстве конструктора, деструктора, а также методов *IsFull* и *IsEmpty* объекта *CQueue*. Лучше поговорим о функции *AddElement*, которую вызывает клиентские потоки для добавления запросов в очередь:

```
void CQueue::AddElement(ELEMENT e) {
    // если очередь заполнена, ничего не предпринимаем
    int nFreeSlot = GetFreeSlot();
    if (nFreeSlot == -1)
        return;

    // копирование содержимое элемента
n_pElements[nFreeSlot].m_element = e;
```

```
    // Обновить номер элемента
    m_pElements[nFreeSlot].m_nStamp = ++m_nCurrentStamp;
}
```

Если в *m_pElements* есть место, в него записывается переданная как параметр структура ELEMENT, а значение в поле номера увеличивается и содержит текущее число элементов в очереди. Пытаясь обработать запрос, серверный поток вызывает функцию *GetNewElement*, передавая ей свой номер (0 или 1), а также структуру ELEMENT, в которую следует записать данные нового запроса:

```
B00L CQueue::GetNewElement(int nThreadNum, ELEMENT& e) {
    int nNewSlot = GetNextSlot(nThreadNum);
    if (nNewSlot == -1)
        return(FALSE);

    // Копируем содержимого элемента
    e = m_pElements[nNewSlot].m_element;

    // Помечаем элемент как прочитанный
    m_pElements[nNewSlot].m_nStamp = 0;

    return(TRUE);
}
```

Вспомогательная функция *GetNextSlot* выполняет основную работу по поиску первого элемента, подходящего для обработки данным серверным потоком. Если такой элемент есть в очереди, *GetNewElement* копирует его данные в предоставленную вызывающим потоком структуру и записывает в поле *m_nStamp* значение 0.

Здесь нет ничего сложного, поэтому вы можете заподозрить *CQueue* в незопасном поведении в многопоточном окружении. Да, ваши подозрения справедливы. В следующей главе я расскажу о том, как с помощью синхронизирующих объектов ядра создать версию объекта *CQueue*, безопасную в многопоточной среде. В приложении 08-Queue.exe за синхронизацию доступа к глобальному экземпляру очереди отвечают сами клиентские и серверные потоки:

```
CQueue  g_q(10);                          // Общая очередь
```

В приложении 08-Queue.exe используются три глобальные переменные, управляющие совместной работой клиентских (пишущих) и серверных (читающих) потоков во избежание повреждения очереди:

```
SRWLOCK             g_srwLock;            // SRWLock-блокировка, защищающая очередь
C0NDITI0N_VARIABLE  g_cvReadyToConsume;   // Условная переменная,
```

```
// устанавливается читающими потоками
CONDITION_VARIABLE  g_cvReadyToProduce;   // Условная переменная,
                                          // устанавливается читающими потоками
```

При каждой попытке обратиться к очереди поток вынужден получать *SRWLock* для совместного (в случае серверных, т.е. читающих потоков) либо монопольного доступа (в случае клиентских, т.е. записывающих потоков).

Реализация клиентских (записывающих) потоков

Рассмотрим реализацию клиентского потока:

```
DWORD WINAPI WriterThread(PVOID pvParam) {

    int nThreadNum = PtrToUlong(pvParam);
    HWND hWndLB = GetDlgItem(g_hWnd, IDC_CLIENTS);

    for (int nRequestNum = 1; !g_fShutdown; nRequestNum++) {

        CQueue::ELEMENT e = { nThreadNum, nRequestNum };

        // Запрашиваем доступ для записи
        AcquireSRWLockExclusive(&g_srwLock);

        // Если очередь заполнена, «усыпляем» поток, пока не изменится
        // значение условной переменной.
        // Примечание. В ожидании блокировки пользователь
        // может щелкнуть кнопку Stop
        if (g_q.IsFull() & !g_fShutdown) {
            // Очередь заполнена
            AddText(hWndW, TEXT("[Xd] Queue is full: impossible to add Xd"),
                nThreadNum, nRequestNum);

            // -> Необходимо дождаться, пока читающий поток не очистит
            // место в очереди, чтобы получить блокировку.
            SleepConditionVariableSRW(&g_cvReadyToProduce, &g_srwLock, INFINITE,
                0);
        }

        // Другие записывающие потоки могут ожидать получения блокировки.
        // --> Освобождаем блокировку и уведомляем записывающие потоки,
        // которые ожидают блокировки.
        if (g_fShutdown) {
            // Уведомляем о завершении текущего потока
            AddText(hWndLB, TEXT("[Xd] bye bye"), nThreadNum);
```

```
            // Удерживать блокировку больше не требуется
            ReleaseSRWLockExclusive(&g_srwLock);

            // Уведомить все остальные ожидающие потоки, если пора
            // завершения работу
            WakeAllConditionVa riable(&g_cvReadyToP roduce);

            // Завершение потока и запись строк «bye bye»
            return(0);
        } else {
            // Добавляем в очередь новый элемент
            g_q.AddElement(e);

            // Показываем результат обработки элемента
            AddText(hWndLB, TEXT("[Xd] Adding Xd"), nThreadNum, nRequestNum);

            // Удерживать блокировку больше не требуется
            ReleaseSRWLockExclusive(&g_srwLock);

            // Уведомляем читающие потоки о том, что в очереди появился
            // новый элемент
            WakeAllConditionVariable(&g_cvReadyToConsume);

            // Ожидаем добавления нового элемента
            Sleep(1500);
        }
    }

    // Уведомляем о завершении текущего потока
    AddText(hWndLB, TEXT("[Xd] bye bye"), nThreadNum);

    return(0);
}
```

Цикл *for* увеличивает значения счетчика запросов, генерируемых этим потоком. Поток завершается, если булева переменная *g_fShutdown* принимает значение TRUE в результате закрытия главного окна приложения либо щелчка кнопки Stop. Я еще вернусь к этому вопросу при обсуждении проблем, связанных с остановкой фоновых клиентских и серверных потоков командой их активного потока, обслуживающего пользовательский интерфейс.

Перед попыткой добавления нового элемента в очередь поток должен получить *SRWLock* для монопольного доступа, вызвав функцию *AcquireSRWLockExclusive*. Если блокировка уже получена другим клиентским или серверным потоком, вызывающий поток блокируется при вызове *AcquireSRWLockExchmve* в ожидании освобождения блокировки. Когда эта

функция возвращает управление, блокировка устанавливается, но кроме этого для постановки запроса в очередь должно выполняться некоторое условие, а именно: в очереди должно быть место. Если очередь заполнена, необходимо «усыпить» записывающий поток, пока один из читающих потоков не обработает запрос и не освободит в очереди место для размещения нового запроса. Однако блокировку необходимо освободить до того, как поток заснет, иначе случится взаимная блокировка потоков: ни один читающий поток не сможет освободить очередь: им будет отказано в доступе, поскольку блокировка будет еще занята. Именно это и делает функция *SleepConditionVariableSRW*, которая освобождает переданную ей в качестве параметра блокировку g_srwLock и «усыпляет» поток, пока функция *g_cvReadyToProduce* не присвоит условной переменной нужное значение вызовом *WakeConditionVariable*. Последнюю операцию выполняет серверный поток, как только в очереди освобождается место.

Когда функция *SleepConditionVariableSRW* вернет управление, будут выполняться два условия: будет установлена блокировка, а другой поток, присвоив условной переменной нужное значение, уведомит клиентские потоки о том, что в очереди появилось место. К этому моменту поток уже готов к постановке в очередь нового запроса. Однако перед этим он проверяет, не поступила ли во время его сна команда на завершение работы и, если нет, добавляет в очередь новый запрос. При этом клиентский поток получает уведомление о том, что он должен обновить свой список и происходит освобождение блокировки вызовом *ReleaseSRWLockExclusive*. Перед следующим оборотом цикла вызывается функция *WakeAllConditionVariable* с передачей *&g_cvReadyToConsume* в качестве параметра, чтобы пробудить все потоки для обработки данных.

Обработка запросов серверными потоками

При запуске приложения создается два серверных потока с идентичными функциями обратного вызова. Каждый из этих потоков обрабатывает запросы с четными либо нечетными номерами, вызывая функцию ConsumeElement в цикле, пока переменной *g_fShutdown* не будет присвоено значение TRUE. Вспомогательная функция возвращает TRUE после успешной обработки запроса и FALSE, если оказывается, что *g_fShutdown* = TRUE.

```
BOOL ConsumeEleroent(lnt nThreadNum, int nRequestNum, HWND hWndLB) {
    // Получаем доступ к очереди для обработки нового элемента
    AcquireSRWLockShared(&g_srwLock);

    // Усыпляем поток, пока не появятся запросы для обработки.
    // Проверяем, не отдана ли команда на прекращение работы,
    // пока поток спал.
    while (g_q.IsEmpty(nThreadNum) && !g_fShutdown) {
```

```
        // В очереди нет доступных элементов
        AddText(hWndLB, TEXT("[*d] Nothing to process"), nThreadNum);

        // Очередь пуста, ждем, пока записывающий поток не добавит
        // в очередь новые запросы, после чего функция вернет управление,
        // установив блокировку, разрешающую совместный доступ для чтения.
        SleepConditionVariableSRW(&g_cvReadyToConsume, &g_srwLock,
            INFINITE, CONDITION_VARIABLE_LOCKMODE_SHARED);
    }

    // Перед завершением работы необходимо освободить блокировку, чтобы
    // уведомить потоки через условные переменные.
    if (g_fShutdown) {
        // уведомляем о завершении текущего потока
        AddText(hWndLB, TEXT("[Xd] bye bye"), nThreadNum);

        // Другой записывающий поток может ожидать блокировки,
        // поэтому необходимо освободить ее перед завершением.
        ReleaseSRWLockShared(&g_srwLock);

        // Уведомляем читающие потоки и
        // пробуждаем их.
        WakeConditionVariable(&g_cvReadyToConsume);

        return(FALSE);
    }

    // Получаем первый элемент очереди
    CQueue::ELEMENT e;
    // Примечание. Нет нужды проверять результат, поскольку функция
    // IsEmpty вернула FALSE
    g_q.GetNewElement(nThreadNum, e);

    // Удерживать блокировку больше не требуется
    ReleaseSRWLockShared(&g_s rwLock);

    // Показываем результат обработки запроса
    AddText(hWndLB, TEXT("[%d] Processing %d:%d"),
        nThreadNum, e.m_nThreadNum, e.m_nRequestNum);
    // В очереди освободилось место для нового запроса,
    // поэтому пробуждаем записывающий поток.
    WakeConditionVariable(&g_cvReadyToProduce);

    return(TRUE);
}
```

```
DWORD WINAPI ReaderThread(PVOID pvParam) {

    int nThreadNum = PtrToUlong(pvParam);
    HWND hWndLB i GetDlgItem(g_hWnd, IDC_SERVERS);

    for (int nRequestNum = 1; !g_fShutdown; nRequestNum++) {

        if (!ConsumeElement(nThreadNum, nRequestNum, hWndLB))
            return(0);

        Sleep(2500);      // Ждем перед чтением следующего элемента
    }

    // Во время сна потока установлена переменная g_fShutdown,
    // поэтому сообщаем о завершении текущего потока.
    AddText(hWndLB, TEXT("[%d] bye bye"), nThreadNum);

    return(0);
}
```

Перед обработкой запроса поток вызывает *AcquireSRWLockShared*, чтобы получить *srwLock* в режиме совместного доступа. Если блокировка в режиме монопольного доступа уже установлена клиентским потоком, вызов блокируется. Если же блокировку установил в режиме совместного доступа другой серверный поток, функция сразу возвращает управление, позволяя обработать запрос. Даже если блокировка будет успешно получена, в очереди может и не оказаться новых запросов, пригодных для обработки данным потоком. Например, в очереди может быть необработанный запрос с нечетным номером, но поток 0 обрабатывает только запросы с четными номерами. В этом случае списку серверного потока отправляется сообщение, а сам поток блокируется при вызове *SleepConditionVariableSRW*, пока клиентский поток не поставит в очередь новый запрос и не присвоит условной переменной *g_cvReadyToConsume* значение, разрешающее его обработать. Когда функция *SteepConditionVariableSRW* вернет управление, блокировка *g_srwLock* будет установлена, а в очереди будет новый запрос. И в этом случае номер запроса может оказаться неподходящим, поэтому *SleepConditionVariableSRW* вызывается в цикле и проверяет наличие в очереди запроса с подходящим номером. Заметьте, что в этом примере используются две условных переменных, а не просто *wReadyToConsume*: одна управляет обработкой запросов с четными номерами, а другая — обработкой запросов с нечетными номерами. Это позволяет не будить зря серверные потоки, если в очереди нет запросов с подходящими номерами. Сейчас приложение реализовано так, что серверные потоки получают блокировку в режиме совместного доступа, даже если очередь обновляется во время вызова *GetNewElement*, поскольку поле *m_nStamp* в представляющей запрос структуре установлено в 0, показывая, что запрос

обрабатывается. В данном примере это не проблема, поскольку запросы «поделены» между серверными потоками: поток 0 обрабатывает только запросы с четными номерами, а поток 1 — только с нечетными номерами.

Если в очереди обнаруживается запрос с походящим номером, то он извлекается, вызывается функция *ReleaseSRWLockShared* и серверному списку направляется сообщение. Теперь пора разбудить клиентские потоки вызовом *WakeConditionVariable* с передачей *&g_cvReadyToProduce* в качестве параметра и уведомить их о наличии свободного места в очереди.

Взаимные блокировки при остановке потоков

Добавляя к окну приложения кнопку Stop, я не думал, что ее щелчок приведет к взаимной блокировке (deadlock). Для остановки клиентских и серверных потоков используется простой код следующего вида:

```
void StopProcessing() {

    if (!g_fShutdown) {
        // уведомляем потоки о необходимости  остановки
        InterlockedExchangePointer((PLONG*) &g_fShutdown, (LONG) TRUE);

        // пробуждаем все потоки, ожидающие на условных переменных
        WakeAllConditionVariable(&g_cvReadyToConsume);
        WakeAllConditionVariable(&g_cvReadyToProduce);

        // дождемся завершения всех потоков и выполним очистку
        WaitForMultipleObjects(g_nNumThreads, g_hThreads, TRUE, INFINITE);

        // Не забудем освободить ресурсы ядра.
        // Примечание. Это не обязательно, поскольку завершается весь
        // процесс.
        while {g_nNumTh reads-)
            CloseHandle(g_hThreads[g_nNumThreads]);

        // закрываем все списки
        AddText(GetDlgItem(g_hWnd, IDC_SERVERS), TEXT("---------------"));
        AddText(GetDlgItem(g_hWnd, IDC_CLIENTS), TEXT("---------------"));
    }
}
```

Флаг *g_fShutdown* устанавливается в TRUE, а обе условные переменные переводятся в состояние, разрешающее пробуждение потоков (вызовом *WakeAllConditionVariable*). После этого остается только вызвать *WaitForMultipleObjects*, используя массив описателей работающих потоков в качестве параметра. Когда *WaitForMultipleObjects* вернет управление, описатели потоков будут закрыты, а в списки будет добавлена последняя строка.

Предполагается, что когда вызов *WakeAllConditionVariabte* пробудит потоки ото сна, в который их погрузила функция *SleepConditionVariableSRW*, они начинают отслеживать значение флага *g_fShutdown* и после его установки просто завершаются, отправляя в списки строку «bye bye». Вот тут-то (при отправке сообщения списку) и может случиться взаимная блокировка. Если код, исполняющий функцию *StopProcessing*, работает с описателем сообщения WM_COMMAND, то поток пользовательского интерфейса, ответственный за обработку оконных сообщений, блокируется в функции *WaitForMultipleObjects*. Если при этом какой-нибудь клиентский или серверный поток вызовет *ListBox_SetCurSel* и *ListBox_AddString*, чтобы добавить к списку новый элемент, поток пользовательского интерфейса не сможет ответить на вызов, и... вот вам взаимная блокировка! Я решил деактивировать кнопку Stop на время исполнения обработчика сгенерированных ей сообщений и породить другой поток для вызова функции *StopProcessing*. При этом нет риска взаимной блокировки, поскольку обработчик сообщений сразу же возвращает управление:

```
DWORD WINAPI StoppingThread(PVOID pvParam) {

    StopProcessing();
    return(0);

}

void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify)
{
    switch (id) {
        case IDCANCEL:
            EndDialog(hWnd, id);
            break;

        case IDC_BTN_STOP:
        {
            // StopProcessing нельзя вызывать из UI-потока из-за
            // риска взаимной блокировки: для заполнения списков
            // используется SendMessage(),
            // следовательно, нужен отдельный поток.
            DWORD dwThreadID;
            CloseHandle(chBEGINTHREADEX(NULL, 0, StoppingThread,
                NULL, 0, &dwThreadID));

            // эту кнопку не удастся щелкнуть дважды
            Button_Enable(hWndCtl, FALSE);
        }
        break;
    }
}
```

Не забывайте, что опасность взаимной блокировки есть и при синхронизации потока пользовательского интерфейса с другими потоками, выполняющими чреватые блокировкой действия, такие как обращение к общим ресурсам. В следующем разделе я дам несколько советов о том, как избежать взаимной блокировки.

И последнее (но от этого не менее важное) замечание: строки добавляются к списку вызовом вспомогательной функции AddText, в которой используется *_vstprintf_s*, новая безопасная функция для работы со строками:

```
void AddText(HWND hWndLB, PCTSTR pszFormat,,..) {

    va_list argList;
    va_start(argList, pszFormat);

    TCHAR sz[20 * 1024];
    _vstprintf_s(sz, _countof(sz), pszFormat, argList);
    ListBox_SetCurSel(hWndLB, ListBox_AddString(hWndLB, sz));

    va_end(argList);
}
```

Несколько полезных приемов

Используя критические секции, желательно привыкнуть делать одни вещи и избегать других. Вот несколько полезных приемов, которые пригодятся вам в работе с критическими секциями. (Они применимы и к синхронизации потоков с помощью объектов ядра, о которой я расскажу в следующей главе.)

На каждый разделяемый ресурс используйте отдельную структуру CRITICAL_SECTION

Довольно часто встречаются единые «логические» ресурсы, составленные из нескольких объектов. Примером может быть ситуация, когда при добавлении элемента в набор необходимо также увеличить значение счетчика. Для этого достаточно одной блокировки, управляющей доступом для чтения и записи к логическому ресурсу.

У каждого логического ресурса в вашем приложении должна быть собственная блокировка, синхронизирующая доступ ко всем без исключения компонентам этого ресурса. Далее, не используйте одну и ту же блокировку для всех логических ресурсов — это снижает масштабируемость при обращении нескольких потоков к разным логическим ресурсам, ведь в этом случае потоки будут исполняться строго по очереди.

Одновременный доступ к нескольким ресурсам

Иногда нужен одновременный доступ сразу к двум и более логическим ресурсам. Например, приложению необходимо заблокировать один ресурс для

извлечения из него элемента, при этом другой ресурс должен быть заблокирован для добавления элемента. Если у каждого ресурса есть своя блокировка, то для атомарного исполнения подобной операции необходимо использовать обе блокировки. Соответствующую функцию можно реализовать так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {

    EnterCriticalSectlon(&g_csResource1);
    EnterCriticalSection(&g_csResource2);

    // извлекаем элемент из Resource1
    // добавляем элемент в Resource2
    LeaveCriticalSection(&g_csResource2);
    LeaveCriticalSection(&g_csResou rce1);
    return(0);
}
```

Предположим, доступ к обоим массивам требуется и другому потоку в данном процессе; при этом его функция написана следующим образом:

```
DWORD WINAPI OtherThreadFunc(PVOTD pvParam) {

    EnterCriticalSection(&g_csResource2);
    EnterCriticalSection(&g_csResource1);

    // извлекаем элемент из Resource1
    // добавляем элемент в Resource2
    LeaveCriticalSection(&g_csResou rce2);
    LeaveCriticalSection(&g_csResou rce1);
    return(0);
}
```

Я лишь поменял порядок вызовов *EnterCriticalSection* и *LeaveCriticalSection*. Но из-за того, что функции *ThreadFunc* и *OtherThreadFunc* написаны именно так, существует вероятность взаимной блокировки. Допустим, *ThreadFunc* начинает исполнение и занимает критическую секцию *g_csResource1*. Получив от системы процессорное время, поток с функцией *OtherThreadFunc* захватывает критическую секцию *g_csResource2*. Тут-то и происходит взаимная блокировка потоков. Какая бы из функций — *ThreadFunc* или *OtherThreadFunc* — ни пыталась продолжить исполнение, она не сумеет занять другую, необходимую ей критическую секцию.

Эту ситуацию легко исправить, написав код обеих функций так, чтобы они вызывали *EnterCriticalSection* в одинаковом порядке. Заметьте, что порядок вызовов *LeaveCriticalSection* несуществен, поскольку эта функция никогда не приостанавливает поток.

Не занимайте критические секции надолго

Надолго занимая критическую секцию, ваше приложение может блокировать другие потоки, что отрицательно скажется на его общей производительности. Вот прием, позволяющий свести к минимуму время пребывания в критической секции. Следующий код не дает другому потоку изменять значение в *g_s* до тех пор, пока в окно не будет отправлено сообщение WM_SOMEMSG:

```
SOMESTRUCT g_s;
CRITICAL_SECTION g_cs;

DWORD WINAPI SomeThread(PVOID pvParam) {
    EnterCriticalSection(&g_cs);

    // посылаеи в окно сообщение
    SendNessage(hNndSomeWnd, WM_SOMEMSG, &g_s, 0);

    LeaveCrlticalSection(&g_cs);
    return(0);
}
```

Трудно сказать, сколько времени уйдет на обработку WM_SOMEMSG оконной процедурой — может, несколько миллисекунд, а может, и несколько лет. В течение этого времени никакой другой поток не получит доступ к структуре *g_s*. Поэтому лучше составить код иначе:

```
SONESTRUCT g_s;
CRITICAL_SECTION g_cs;

DWORD WINAPI SomeThread(PVOID pvParam) {

    EnterCriticalSection(&g_cs);
    SOMESTRUCT sTemp = g_s;
    LeaveCriticalSection(&g_cs);

    // посылаем в окно сообщение
    SendMessage(hWndSomeWnd, WM_SOMEMSG, &sTemp, 0);
    return(0);
}
```

Этот код сохраняет значение элемента *g_s* во временной переменной *sTemp*. Нетрудно догадаться, что на исполнение этой строки уходит всего несколько тактов процессора. Далее программа сразу вызывает *LeaveCriticalSection* — защищать глобальную структуру больше не нужно. Так что вторая версия программы намного лучше первой, поскольку другие потоки «отлучаются» от структуры *g_s* лишь на несколько тактов процессора, а не на неопределенно долгое время. Такой подход предполагает, что «моментальный снимок» структуры вполне пригоден для чтения оконной процедурой, а также что оконная процедура не будет изменять элементы этой структуры.

Оглавление

ГЛАВА 9

Синхронизация потоков с использованием объектов ядра

В предыдущей главе мы обсудили, как синхронизировать потоки с применением механизмов, позволяющих вашим потокам оставаться в пользовательском режиме. Самое удивительное, что эти механизмы работают очень быстро. Поэтому, если вы озабочены быстродействием потока, сначала проверьте, нельзя ли обойтись синхронизацией в пользовательском режиме.

Хотя механизмы синхронизации в пользовательском режиме обеспечивают высокое быстродействие, им свойствен ряд ограничений, и во многих приложениях они просто не будут работать. Например, *Interlocked*-функции оперируют только с отдельными переменными и никогда не переводят поток в состояние ожидания. Последнюю задачу можно решить с помощью критических секций, но они подходят лишь в тех случаях, когда требуется синхронизировать потоки в рамках одного процесса. Кроме того, при использовании критических секций легко попасть в ситуацию взаимной блокировки потоков, потому что задать предельное время ожидания входа в критическую секцию нельзя.

В этой главе мы рассмотрим, как синхронизировать потоки с помощью объектов ядра. Вы увидите, что такие объекты предоставляют куда больше возможностей, чем механизмы синхронизации в пользовательском режиме. В сущности, единственный их недостаток — меньшее быстродействие. Дело в том, что при вызове любой из функций, упоминаемых в этой главе, поток должен перейти из пользовательского режима в режим ядра. А такой переход обходится очень дорого — в 200 процессорных тактов на платформе *x86*. Прибавьте сюда еще и время, которое необходимо на выполнение кода этих функций в режиме ядра. Но все это мелочь по сравнению с издержками, вызванными необходимостью планирования потока, сбросом и промахами кэша, измеряемыми десятками тысяч тактов.

К этому моменту я уже рассказал вам о нескольких объектах ядра, в том числе о процессах, потоках и заданиях. Почти все они годятся и для решения задач синхронизации. В случае синхронизации потоков о каждом из этих объектов говорят, что он находится либо в свободном (signaled state), либо в занятом состоянии (nonsignaled state). Переход из одного состояния в другое осуществляется по правилам, определенным Майкрософт для каждого из объектов ядра. Так, объекты ядра «процесс» сразу после создания всегда находятся в занятом состоянии. В момент завершения процесса операционная система автоматически освобождает его объект ядра «процесс», и он навсегда остается в этом состоянии.

Объект ядра «процесс» пребывает в занятом состоянии, пока выполняется сопоставленный с ним процесс, и переходит в свободное состояние, когда процесс завершается. Внутри этого объекта поддерживается булева переменная, которая при создании объекта инициализируется как FALSE («занято»). По окончании работы процесса операционная система меняет значение этой переменной на TRUE, сообщая тем самым, что объект свободен.

Если вы пишете код, проверяющий, выполняется ли процесс в данный момент, вам нужно лишь вызвать функцию, которая просит операционную систему проверить значение булевой переменной, принадлежащей объекту ядра «процесс». Тут нет ничего сложного. Вы можете также сообщить системе, чтобы та перевела ваш поток в состояние ожидания и автоматически пробудила его при изменении значения булевой переменной с FALSE на TRUE. Тогда появляется возможность заставить поток в родительском процессе, ожидающий завершения дочернего процесса, просто заснуть до освобождения объекта ядра, идентифицирующего дочерний процесс. В дальнейшем вы увидите, что в Windows есть ряд функций, позволяющих легко решать эту задачу.

Я только что описал правила, определенные Майкрософт для объекта ядра «процесс». Точно такие же правила распространяются и на объекты ядра «поток». Они тоже сразу после создания находятся в занятом состоянии. Когда поток завершается, операционная система автоматически переводит объект ядра «поток» в свободное состояние. Таким образом, используя те же приемы, вы можете определить, выполняется ли в данный момент тот или иной поток. Как и объект ядра «процесс», объект ядра «поток» никогда не возвращается в занятое состояние.

Следующие объекты ядра бывают в свободном или занятом состоянии:

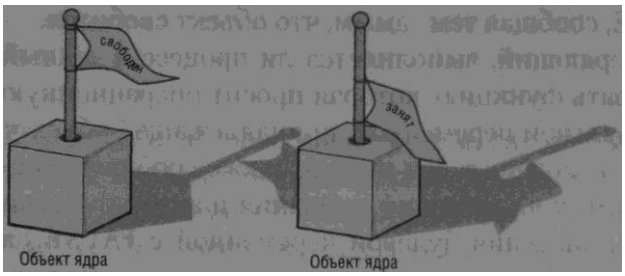
- процессы;
- потоки;
- задания;
- файлы;
- консольный ввод;
- мьютексы;
- семафоры;

- ожидаемые таймеры;
- события.

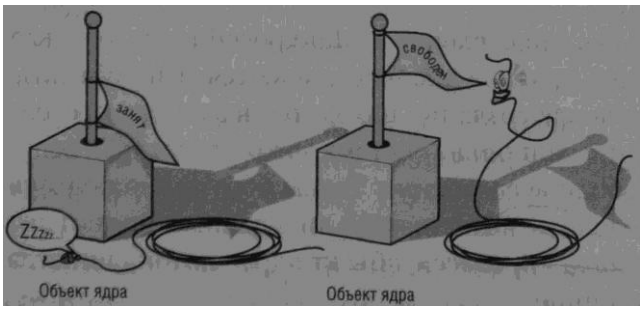
Потоки могут засыпать и в таком состоянии ждать освобождения какого-либо объекта. Правила, по которым объект переходит в свободное или занятое состояние, зависят от типа этого объекта. О правилах для объектов процессов и потоков я упоминал совсем недавно, а правила для заданий были описаны в главе 5.

В этой главе мы обсудим функции, которые позволяют потоку ждать перехода определенного объекта ядра в свободное состояние. Потом мы поговорим об объектах ядра, предоставляемых Windows специально для синхронизации потоков: событиях, ожидаемых таймерах, семафорах и мьютексах.

Когда я только начинал осваивать всю эту тематику, я предпочитал рассматривать понятия «свободен-занят» по аналогии с обыкновенным флажком. Когда объект свободен, флажок поднят, а когда он занят, флажок опущен.



Потоки спят, пока ожидаемые ими объекты заняты (флажок опущен). Как только объект освободился (флажок поднят), спящий поток замечает это, просыпается и возобновляет выполнение.



Wait-функции

Wait-функции позволяют потоку в любой момент приостановиться и ждать освобождения какого-либо объекта ядра. Заметьте, что поток не будет приостановлен, если в момент вызова *wait*-функции заданный объект ядра свободен. Из всего семейства этих функций чаще всего используется *WaitForSingleObject*:

```
DWORD WaitForSingleObject(
    HANDLE hObject,
    DWORD dwMilliseconds);
```

Когда поток вызывает эту функцию, первый параметр, *hObject*, идентифицирует объект ядра, поддерживающий состояния «свободен-занят». (То есть любой объект, упомянутый в списке из предыдущего раздела.) Второй параметр, *dwMilliseconds*, указывает, сколько времени (в миллисекундах) поток готов ждать освобождения объекта.

Следующий вызов сообщает системе, что поток будет ждать до тех пор, пока не завершится процесс, идентифицируемый описателем *hProcess*:

```
WaitForSingleObject(hProcess, INFINITE);
```

Второй параметр сообщает системе, что вызывающий поток намерен ждать вечно (либо до завершения процесса).

В данном случае константа INFINITE, передаваемая во втором параметре, подсказывает системе, что вызывающий поток готов ждать этого события хоть целую вечность. Именно эта константа обычно и передается функции *WaitForSingleObject*, но вы можете указать любое значение в миллисекундах. Кстати, константа INFINITE определена как 0xFFFFFFFF (или -1). Разумеется, передача INFINITE не всегда безопасна. Если объект так и не перейдет в свободное состояние, вызывающий поток никогда не проснется; одно утешение: тратить драгоценное процессорное время он при этом не будет.

Вот пример, иллюстрирующий, как вызывать *WaitForSingleObject* со значением таймаута, отличным от INFINITE:

```
DWORD dw = WaitForSingleObject(hProcess, 5000);
switch (dw) {

    case WAIT_0BJECT_0:
        // процесс завершается
        break;

    case WAIT_TIMEOUT:
        // процесс не завершился в течение 5000 мс
        break;

    case WAIT_FAILED:
        // неправильный вызов функции (неверный описатель?)
        break;
}
```

Данный код сообщает системе, что вызывающий поток нс должен получать процессорное время, пока не завершится указанный процесс или не пройдет 5000 мс (в зависимости от того, что случится раньше). Поэтому функция вернет управление либо до истечения 5000 мс, если процесс завершится, либо примерно через 5000 мс, если процесс к тому времени не закон-

чит свою работу. Заметьте, что в параметре *dwMilliseconds* можно передать 0, и тогда *WaitForSingleObject* немедленно вернет управление.

Возвращаемое значение функции *WaitForSingleObject* указывает, почему вызывающий поток снова стал планируемым. Если функция возвращает WAIT_OBJECT_0, объект свободен, а если WAIT_TIMEOUT — заданное время ожидания (таймаут) истекло. При передаче неверного параметра (например, недопустимого описателя) *WaitForSingleObject* возвращает WAIT_FAILED. Чтобы выяснить конкретную причину ошибки, вызовите функцию *GetLastError*.

Функция *WaitForMultipleObjects* аналогична *WaitForSingleObject* с тем исключением, что позволяет ждать освобождения сразу нескольких объектов или какого-то одного из списка объектов:

```
DWORD WaitForMultipleObjects(
    DWORD dwCount,
    CONST HANDLE* phObjects,
    B00L bWaitAll,
    DWORD dwMilliseconds);
```

Параметр *dwCount* определяет количество интересующих вас объектов ядра. Его значение должно быть в пределах от 1 до MAXIMUM_WAIT_OBJECTS (в заголовочных файлах Windows оно определено как 64). Параметр *phObjects* — это указатель на массив описателей объектов ядра.

WaitForMultipleObjects приостанавливает поток и заставляет его ждать освобождения либо всех заданных объектов ядра, либо одного из них. Параметр *fWaitAll* как раз и определяет, чего именно вы хотите от функции. Если он равен TRUE, функция не даст потоку возобновить свою работу, пока не освободятся все объекты.

Параметр *dwMilliseconds* идентичен одноименному параметру функции *WaitForSingleObject*. Если вы указываете конкретное время ожидания, то по его истечении функция в любом случае возвращает управление. И опять же, в этом параметре обычно передают INFINITE (будьте внимательны при написании кода, чтобы не создать ситуацию взаимной блокировки).

Возвращаемое значение функции *WaitForMultipleObjects* сообщает, почему возобновилось выполнение вызвавшего ее потока. Значения WAIT_FAILED и WAIT_TIMEOUT никаких пояснений не требуют. Если вы передали TRUE в параметре *fWaitAll* и все объекты перешли в свободное состояние, функция возвращает значение WAIT_OBJECT_0. Если же *fWaitAll* приравнен FALSE, она возвращает управление, как только освобождается любой из объектов. Вы, по-видимому, захотите выяснить, какой именно объект освободился. В этом случае возвращается значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + *dwCount* - 1. Иначе говоря, если возвращаемое значение не равно WAIT_TIMEOUT или WAIT_FAILED, вычтите из него значение WAIT_OBJECT_0, и вы получите индекс в массиве описателей, на который указывает второй параметр функции *WaitForMultipleObjects*. Индекс

подскажет вам, какой объект перешел в незанятое состояние. Поясню сказанное на примере.

```
HANDLE h [3];
h[0] = hProcess1;
h[1] = hProcess2;
h[2] = hProcess3;
DWORD dw = WaitForMultipleObjects(3, h, FALSE, 5000);
switch (dw) {
    case WAIT_FAILED:
        // неправильный вызов функции (неверный описатель?)
        break;

    case WAIT_TIMEOUT:
        // ни один из объектов не освободился в течение 5000 мс
        break;

    case WAIT_0BJECT_0 + 0:
        // завершился процесс, идентифицируемый h[0], т. е. описателем (hProcess1)
        break;

    case WAIT_0BJECT_0 + 1:
        // завершился процесс, идентифицируемый h[1], т. е. описателем (hProcess2)
        break;

    case WAIT_0BJECT.0 + 2:
        // завершился процесс, идентифицируемый h[2], т. е. описателем (hProcess3)
        break;
}
```

Если вы передаете FALSE в параметре *fWaitAll*, функция *WaitForMultipleObjects* сканирует массив описателей (начиная с нулевого элемента), и первый же освободившийся объект прерывает ожидание. Это может привести к нежелательным последствиям. Например, ваш поток ждет завершения трех дочерних процессов; при этом вы передали функции массив с их описателями. Если завершается процесс, описатель которого находится в нулевом элементе массива, *WaitForMultipleObjects* возвращает управление. Теперь поток может сделать то, что ему нужно, и вновь вызвать эту функцию, ожидая завершения другого процесса. Если поток передаст те же три описателя, функция немедленно вернет управление, и вы снова получите значение WAIT_OBJECT_0. Таким образом, пока вы не удалите описатели тех объектов, об освобождении которых функция уже сообщила вам, код будет работать некорректно.

Побочные эффекты успешного ожидания

Успешный вызов *WaitForSingleObject* или *WaitForMultipleObjects* на самом деле меняет состояние некоторых объектов ядра. Под успешным вызовом я имею в виду тот, при котором функция видит, что объект освободился, и возвращает значение, относительное WAIT_OBJECT_0. Вызов считается неудачным, если возвращается WAIT_TIMEOUT или WAIT_FAILED. В последнем случае состояние каких-либо объектов не меняется.

Изменение состояния объекта в результате вызова я называю *побочным эффектом успешного ожидания* (successful wait side effect). Например, поток ждет объект «событие с автосбросом» (auto-reset event object) (об этих объектах я расскажу чуть позже). Когда объект переходит в свободное состояние, функция обнаруживает это и может вернуть вызывающему потоку значение WAIT_OBJECT_0. Однако перед самым возвратом из функции событие переводится в занятое состояние — здесь сказывается побочный эффект успешного ожидания.

Объекты ядра «событие с автосбросом» ведут себя подобным образом, потому что таково одно из правил, определенных Майкрософт для объектов этого типа. Другие объекты дают иные побочные эффекты, а некоторые — вообще никаких. К последним относятся объекты ядра «процесс» и «поток», так что поток, ожидающий один из этих объектов, никогда не изменит его состояние. Подробнее о том, как ведут себя объекты ядра, я буду рассказывать при рассмотрении соответствующих объектов.

Чем ценна функция *WaitForMultipleObjects*, так это тем, что она выполняет все действия на уровне атомарного доступа. Когда поток обращается к этой функции, она ждет освобождения всех объектов и в случае успеха вызывает в них требуемые побочные эффекты; причем все действия выполняются как одна операция.

Возьмем такой пример. Два потока вызывают *WaitForMultipleObjects* совершенно одинаково:

```
HANDLE h[2];
h[0] = hAutoResetEvent1;                    // изначально занят
n[1] - hAutoResetEvent2;                    // изначально занят
WaitForMultipleObjects(2, h, TRUE, INFINITE);
```

На момент вызова *WaitForMultipleObjects* эти объекты-события заняты, и оба потока переходят в режим ожидания. Но вот освобождается объект *hAutoResetEvent1*. Это становится известным обоим потокам, однако ни один из них не пробуждается, так как объект *hAutoResetEvent2* по-прежнему занят. Поскольку потоки все еще ждут, никакого побочного эффекта для объекта *hAutoResetEvent1* не возникает.

Наконец освобождается и объект *hAutoResetEvent2*. В этот момент один из потоков обнаруживает, что освободились оба объекта, которых он ждал. Его ожидание успешно завершается, оба объекта снова переводятся в заня-

тое состояние, и выполнение потока возобновляется. А что же происходит со вторым потоком? Он продолжает ждать и будет делать это, пока вновь не освободятся оба объекта-события.

Как я уже упоминал, *WaitForMultipleObjects* работает на уровне атомарного доступа, и это очень важно. Когда она проверяет состояние объектов ядра, никто не может «у нее за спиной» изменить состояние одного из этих объектов. Благодаря этому исключаются ситуации со взаимной блокировкой. Только представьте, что получится, если один из потоков, обнаружив освобождение hAutoResetEvent1, сбросит его в занятое состояние, а другой поток, узнав об освобождении hAutoResetEvent2, тоже переведет его в занятое состояние. Оба потока просто зависнут: первый будет ждать освобождения объекта, захваченного вторым потоком, а второй — освобождения объекта, захваченного первым. *WaitForMultipleObjects* гарантирует, что такого не случится никогда.

Тут возникает интересный вопрос. Если несколько потоков ждет один объект ядра, какой из них пробудится при освобождении этого объекта? Официально Майкрософт отвечает на этот вопрос так: «Алгоритм действует честно». Что это за алгоритм, Майкрософт не говорит, потому что не хочет связывать себя обязательствами всегда придерживаться именно этого алгоритма. Она утверждает лишь одно: если объект ожидается несколькими потоками, то всякий раз, когда этот объект переходит в свободное состояние, каждый из них получает шанс на пробуждение.

Таким образом, приоритет потока не имеет значения: поток с самым высоким приоритетом не обязательно первым захватит объект. Не получает преимущества и поток, который ждал дольше всех. Есть даже вероятность, что какой-то поток сумеет повторно захватить объект. Конечно, это было бы нечестно по отношению к другим потокам, и алгоритм пытается не допустить этого. Но никаких гарантий нет.

На самом деле этот алгоритм просто использует популярную схему «первым вошел — первым вышел» (FIFO). В принципе, объект захватывается потоком, ждавшим дольше всех. Но в системе могут произойти какие-то события, которые повлияют на окончательное решение, и из-за этого алгоритм становится менее предсказуемым. Вот почему Майкрософт и не хочет говорить, как именно он работает. Одно из таких событий — приостановка какого-либо потока. Если поток ждет объект и вдруг приостанавливается, система просто забывает, что он ждал этот объект. А причина в том, что нет смысла планировать приостановленный поток. Когда он, в конце концов, возобновляется, система считает, что он только что начал ждать данный объект.

Учитывайте это при отладке, поскольку в точках прерывания (breakpoints) все потоки внутри отлаживаемого процесса приостанавливаются. Отладка делает алгоритм FIFO в высшей степени непредсказуемым из-за частых приостановки и возобновления потоков процесса.

События

События — самая примитивная разновидность объектов ядра. Они содержат счетчик числа пользователей (как и все объекты ядра) и две булевы переменные: одна сообщает тип данного объекта-события, другая — его состояние (свободен или занят).

События просто уведомляют об окончании, какой-либо операции. Объекты-события бывают двух типов: со сбросом вручную (manual-reset events) и с автосбросом (auto-reset events). Первые позволяют возобновлять выполнение сразу нескольких ждущих потоков, вторые — только одного.

Объекты-события обычно используют в том случае, когда какой-то поток выполняет инициализацию, а затем сигнализирует другому потоку, что тот может продолжить работу. Инициализирующий поток переводит объект «событие» в занятое состояние и приступает к своим операциям. Закончив, он сбрасывает событие в свободное состояние. Тогда другой поток, который ждал перехода события в свободное состояние, пробуждается и вновь становится планируемым.

Объект ядра «событие» создается функцией *CreateEvent*:

```
HANDLE CreateEvent(
    PSECURITY_ATTRIBUTES psa,
    BOOL bManualReset,
    BOOL bInitialState,
    PCTSTR pszName);
```

В главе 3 мы обсуждали общие концепции, связанные с объектами ядра, — защиту, учет числа пользователей объектов, наследование их описателей и совместное использование объектов за счет присвоения им одинаковых имен. Поскольку все это вы теперь знаете, я не буду рассматривать первый и последний параметры данной функции.

Параметр *bManualReset* (булева переменная) сообщает системе, хотите вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE). Параметр *bInitialState* определяет начальное состояние события — свободное (TRUE) или занятое (FALSE). После того как система создает объект-событие, *CreateEvent* возвращает описатель события, специфичный для конкретного процесса. В Windows Vista поддерживается новая функция для создания объектов-событий, *CreateEventExr*.

```
HANDLE CreateEventEx(
    PSECURITY_ATTRIBUTES psa,
    PCTSTR pszName,
    DWORD dwFlags,
    DWORD dwDesiredAccess);
```

Ее параметры *psa* и *pszName* идентичны одноименным параметрам функции CreateEvent. Параметр *dwFlags* принимает две битовые маски (см. табл. 9-1).

Табл. 9-1. Флаги функции *CreateEvent*

Константы, определенные в WinBase.h	Описание
CREATE_EVENT_INITIAL_SET (0x00000002)	Эквивалент параметра *bInitialState*, передаваемого функции *CreateEvent*. Если установлен соответствующий битовый флаг, событие инициализируется как свободное, а в противном случае — как занятое.
CREATE_EVENT_MANUAL_RESET (6x00000001)	Эквивалент параметра *bManualReset*, передаваемого функции *CreateEvent*. Если установлен соответствующий битовый флаг, событие инициализируется как событие с ручным сбросом, а в противном случае — как событие с автосбросом.

Параметр *dwDesiredAccess* позволяет во время создания объекта указать уровень доступа для описателя события, возвращаемого функцией. Новая функция может создавать описатели событий с пониженным уровнем доступа, а *CreateEvent* всегда возвращает описатели с правами полного доступа. Однако более полезно то, что *CreateEventEx* способна открывать существующие события, Запрашивая при этом пониженный уровень доступа, тогда как *CreateEvent* всегда запрашивает полный доступ. Например, флаг EVENT_MODIFY_STATE (0x0002) необходим для вызова функций *SetEvent, ResetEvent* и *PulseEvent* (о них — чуть позже). Подробнее об этом см. на сайте MSDN (http://msdn2.microsoft.com/en-us/library/ms686670.aspx)

Потоки из других процессов могут получать доступ к объектам, вызывая *CreateEvent* с передачей имени нужного объекта-события в параметре pszName; при помощи наследования; а также вызывая функции *DuplicateHandle* либо *OpenEvent* с передачей в параметре *pszName* имени, указанного при вызове *CreateEvent*.

```
HANDLE 0penEvent(
    DWORD dwDesiredAccess,
    B00L bInherit,
    PCTSTR pszName);
```

Ненужный объект ядра «событие» следует, как всегда, закрыть вызовом *CloseHandle*.

Создав событие, вы можете напрямую управлять его состоянием. Чтобы перевести его в свободное состояние, вы вызываете:

```
B00L S8etEvent(HANDLE hEvent);
```

А чтобы поменять его на занятое:

```
B00L ResetEvent(HANDLE hEvent);
```

Вот так все просто.

Для событий с автосбросом действует следующее правило. Когда его ожидание потоком успешно завершается, этот объект автоматически сбра

сывается в занятое состояние. Отсюда и произошло название таких объектов-событий. Для этого объекта обычно не требуется вызывать *ResetEvent*, поскольку система сама восстанавливает его состояние. А для событий со сбросом вручную никаких побочных эффектов успешного ожидания не предусмотрено.

Рассмотрим небольшой пример тому, как на практике использовать объекты ядра «событие» для синхронизации потоков. Начнем с такого кода:

```
// глобальный описатель события со сбросом вручную (в занятом состоянии)
HANDLE g_hEvent;

int WINAPI _tWinMain(…) {

    // создаем объект "событие со сбросом вручную" (в занятом состоянии)
    g_hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

    // порождаем три новых потока
    HANDLE hThread[3];
    DWORD dwThreadID;
    hThread[0] = _beginthreadex(NULL, 0, WordCount, NULL, 0, &dwThreadID);
    hThread[1] = _beginthreadex(NULL, 0, SpellCheck, NULL, 0, &dwThreadID);
    hThread[2] = _beginthreadex(NULL, 0, GrammarCheck, NULL, 0, &dwThreadID);

    OpenFileAndReadContentsIntoMemory(…);

    // разрешаем всем трем потокам обращаться к памяти
    SetEvent(g_hEvent);
    …
}

DWORD WINAPI WordCount(PV0ID pvParam) {

    // ждем, когда в память будут загружены данные из файла
    WaitForSingle0bject(g_hEvent, INFINITE);

    // обращаемся к блоку памяти
    …
    return(0);
}

DWORD WINAPI SpellCheck (PV0ID pvParam) {

    // ждем, когда в память будут загружены данные из файла
    WaitForSingleObject(g_hEvent, INFINITE);

    // обращаемся к блоку памяти
    …
```

```
    return(0);
}

DWORD WINAPI GrammarCheck (PVOID pvParam) {

    // ждем, когда в память будут загружены данные из файла
    WaitForSingleObject(g_hEvent, INFINITE);

    // обращаемся к блоку памяти
    …
    return(0);
}
```

При запуске этот процесс создает занятое событие со сбросом вручную и записывает его описатель в глобальную переменную. Это упрощает другим потокам процесса доступ к тому же объекту-событию. Затем порождается три потока. Они ждут, когда в память будут загружены данные (текст) из некоего файла, и потом обращаются к этим данным: один поток подсчитывает количество слов, другой проверяет орфографические ошибки, третий — грамматические. Все три функции потоков начинают работать одинаково: каждый поток вызывает *WaitForSingleObject*, которая приостанавливает его до тех пор, пока первичный поток не считает в память содержимое файла.

Загрузив нужные данные, первичный поток вызывает *SetEvent*, которая переводит событие в свободное состояние. В этот момент система пробуждает три вторичных потока, и они, вновь получив процессорное время, обращаются к блоку памяти. Заметьте, что они получают доступ к памяти в режиме только для чтения. Это единственная причина, по которой все три потока могут выполняться одновременно. Заметьте также, что на многопроцессорном компьютере эти потоки действительно будут исполняться одновременно, выполняя больше вычислений за меньшее время.

Если событие со сбросом вручную заменить событием с автосбросом, программа будет вести себя совершенно иначе. После вызова первичным потоком функции *SetEvent* система возобновит выполнение только одного из вторичных потоков. Какого именно — сказать заранее нельзя. Остальные два потока продолжат ждать.

Поток, вновь ставший планируемым, получает монопольный доступ к блоку памяти, где хранятся данные, считанные из файла. Давайте перепишем функции потоков так, чтобы перед самым возвратом управления они (подобно функции *_tWinMain*) вызывали *SetEvent*. Теперь функции потоков выглядят следующим образом:

```
DWORD WINAPI WordCount(PVOID pvParam) {

    // ждем, когда в память будут загружены данные из файла
    WaitForSingleObject(g_hEvent, INFINITE);
```

```
    // обращаемся к блоку памяти
    …
    SetEvent(g_hEvent);
    return(0);
}

DWORD WINAPI SpellCheck (PVOID pvParam) {

    // ждем, когда в память будут загружены данные из файла
    WaitForSingleObject(g_hEvent, INFINITE);

    // обращаемся к блоку памяти

    SetEvent(g_h Event);
    return(0);
}

DWORD WINAPI GrammarCheck (PVOID pvParam) {

    // ждем, когда в память будут загружены данные из файла
    WaitFprSingleObj ect(g_h Event, INFINITE);

    // обращаемся к блоку памяти
    …
    SetEvent(g_hEvent);
    return(0);
}
```

Закончив свою работу с данными, поток вызывает *SetEvent*, которая решает системе возобновить выполнение следующего из двух ждущих потоков. И опять мы не знаем, какой поток выберет система, но так или иначе кто-то из них получит монопольный доступ к тому же блоку памяти. Когда и этот поток закончит свою работу, он тоже вызовет *SetEvent*, после чего с блоком памяти сможет монопольно оперировать третий, последний поток. Обратите внимание, что использование события с автосбросом снимает проблему с доступом вторичных потоков к памяти как для чтения, так и для записи; вам больше не нужно ограничивать их доступ только чтением. Этот пример четко иллюстрирует различия в применении событий со сбросом вручную и с автосбросом.

Для полноты картины упомяну о еще одной функции, которую можно использовать с объектами-событиями:

```
BOOL PulseEvent(HANDLE hEvent);
```

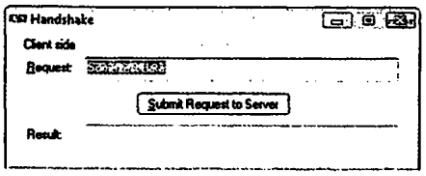
PulseEvent освобождает событие и тут же переводит его обратно в занятое состояние; ее вызов равнозначен последовательному вызову SetEvent и *ResetEvent*. Если вы вызываете *PulseEvent* для события со сбросом вручную,

любые потоки, ждущие этот объект, становятся планируемыми. При вызове этой функции применительно к событию с автосбросом пробуждается только один из ждущих потоков. А если ни один из потоков не ждет объект-событие, вызов функции не дает никакого эффекта.

Особой пользы от *PulseEvent* я не вижу. В сущности, я никогда не пользовался ею на практике, потому что абсолютно неясно, какой из потоков заметит этот импульс и станет планируемым. Наверное, в каких-то сценариях *PulseEvent* может пригодиться, но ничего такого мне в голову не приходит. Когда мы перейдем к рассмотрению функции *SignalObjectAndWait*, я расскажу о *PulseEvent* чуть подробнее.

Программа-пример Handshake

Эта программа, «09 Handshake.exe» (Файлы исходного кода и ресурсов этой программы находятся в каталоге 09-Handshake в архиве, доступном на веб-сайте поддержки этой книги, http://www.wintellect.com/Books.aspx), демонстрирует применение событий с автосбросом. После запуска *Handshake* открывается окно, показанное ниже.



Handshake принимает строку запроса, меняет в ней порядок всех символов и показывает результат в поле Result. Самое интересное в программе Handshake — то, как она выполняет эту героическую задачу.

Программа решает типичную проблему программирования. У вас есть клиент и сервер, которые должны как-то общаться друг с другом. Изначально серверу делать нечего, и он переходит в состояние ожидания. Когда клиент готов передать ему запрос, он помещает этот запрос в разделяемый блок памяти и переводит объект-событие в свободное состояние, чтобы поток сервера считал этот блок памяти и обработал клиентский запрос. Пока серверный поток занят обработкой запроса, клиентский должен ждать, когда будет готов результат. Поэтому клиент переходит в состояние ожидания и остается в нем до тех пор, пока сервер не освободит другой объект-событие, указав тем самым, что результат готов. Вновь пробудившись, клиент узнает, что результат находится в разделяемом блоке памяти, и выводит готовые данные пользователю.

При запуске программа немедленно создает два объекта-события с автосбросом в занятом состоянии. Один из них, *g_hevtRequestSubmitted*, используется как индикатор готовности запроса к серверу. Это событие ожидается серверным потоком и освобождается клиентским. Второй объект-событие,

g_hevtResultReturned, служит индикатором готовности данных для клиента. Это событие ожидается клиентским потоком, а освобождается серверным.

После создания событий программа порождает серверный поток и выполняет функцию *ServerThread*. Эта функция немедленно заставляет серверный поток ждать запроса от клиента. Тем временем первичный поток, который одновременно является и клиентским, вызывает функцию *DialogBox*, отвечающую за отображение пользовательского интерфейса программы. Вы вводите какой-нибудь текст в поле Request и, щелкнув кнопку Submit Request To Server, заставляете программу поместить строку запроса в буфер памяти, разделяемый между клиентским и серверным потоками, а также перевести событие *g_hevtRequestSubmitted* в свободное состояние. Далее клиентский поток ждет результат от сервера, используя объект-событие *g_hevtResultReturned*.

Теперь пробуждается серверный поток, обращает строку в блоке разделяемой памяти, освобождает событие *g_hevtResultReturned* и вновь засыпает, ожидая очередного запроса от клиента. Заметьте, что программа никогда не вызывает *ResetEvent*, так как в этом нет необходимости: события с автосбросом автоматически восстанавливают свое исходное (занятое) состояние в результате успешного ожидания. Клиентский поток обнаруживает, что событие *g_hevtResultReturned* освободилось, пробуждается и копирует строку из общего буфера памяти в поле Result.

Последнее, что заслуживает внимания в этой программе, — то, как она завершается. Вы закрываете ее окно, и это приводит к тому, что *DialogBox* в функции *_tWinMain* возвращает управление. Тогда первичный поток копирует в общий буфер специальную строку и пробуждает серверный поток, чтобы тот ее обработал. Далее первичный поток ждет от сервера подтверждения о приеме этого специального запроса и завершения его потока. Серверный поток, получив от клиента специальный запрос, выходит из своего цикла и сразу же завершается. Специальная строка не интерпретируется как команда на завершение, пока отображается главное окно программы, для *g_hMainDlg* проверяется на NULL-значение.

Я предпочел сделать так, чтобы первичный поток ждал завершения серверного вызовом *WaitForMultipleObjects*, — просто из желания продемонстрировать, как используется эта функция. На самом деле я мог бы вызвать и *WaitForSingleObject*, передав ей описатель серверного потока, и все работало бы точно так же.

Как только первичный поток узнает о завершении серверного, он трижды вызывает *CloseHandle* для корректного закрытия всех объектов ядра, которые использовались программой. Конечно, система могла бы закрыть их за меня, но как-то спокойнее, когда делаешь это сам. Я предпочитаю полностью контролировать все, что происходит в моих программах.

```
Handshake.cpp

/******************************************************************************
Module:   Handshake.cpp
Notices:  Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************A*******************/

#include "..\CommonFiles\CmnHdr.h"          /*см. Приложение A */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////

// это событие освобождается, когда клиент передает запрос серверу
HANDLE g_hevtRequestSubmitted;

// это событие освобождается, когда сервер готов сообщить результат
// клиенту
HAN0LE g_hevtResultReturned;

// это буфер, разделяемый между клиентским и серверным потоками
TCHAR g_szSharedRequestAndResultBuffer[1024];

// Специальное значение, посылаемое клиентом;
// оно заставляет серверный поток корректно завершиться.
TCHAR g_szServerShutdown[] = TEXT("Server Shutdown");

// При получении команды на завершение серверный поток проверяет,
// не открыто ли еще главное окно программы.
HWND   g_hHainDlg;

//////////////////////////////////////////////////////////////////////////////

// это код, выполняемый серверным потоком
DWORD WINAPI ServerThread(PV0I0 pvParam) {

   // предполагаем, что серверный поток будет выполняться вечно
   B00L fShutdown * FALSE;

   while (lfShutdown) {

      // ждем от клиента передачи запроса
      WaitForSingle0bj6Ct(g.hevtRequestSubmltted, INFINITE);

      // проверяем, не хочет ли клиент, чтобы сервер завершился
      fShutdown =
```

```
        (g_hMainDlg == NULL) &&
            (_tcscmp(g_szSharedRequestAndResultBuffer, g_ erShutdown) == 0);
        if{!fShutdown) {
            //,обрабатывавм клиентский запрое-(инвертиру^стрску)
            _tcsrev(g_szSharedRequestAndResultBuffer);
        }

        // разрешаем клиенту обработать результат запроса:
        SetEvent(g_hevtResultReturned);
    }

    // клиент хочет завершить программу — выходим
    return(0);
}

//////////////////////////////////////////////////////////////////////////////
    ///

BOOL Dlg_OnInitDialog (HWND hwnd, HNND hwndFocus,LPARAM lParam){

    chSETDLGICONS(hwnd, IDI_HANDSHAKE);

    // инициализируем поле ввода текстом запроса по умолчанию
    Edit_SetText(GetDlgItem(hwnd, IDC_REQUEST), TEXT{"Some test data"));

    // Сохраняем описатель главного окна
    g_hMainDlg = hwnd;

    return(TRUE»
}

//////////////////////////////////////////////////////////////////////////////
    ///

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)
{
    Switch (id) {

        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_SUBMIT:   // передаем запрос серверному потоку
```

```
        // копируем строку запроса в разделяемый блок памяти
        Edit_GetText(GetDlgItem(hwnd, IDC_REQUEST),
            g_szSharedRequestAndResultBuffer,
            _countof(g_szSharedRequestAndResultBuffer));

        // даем знать серверному потоку, что в буфере появился запрос
        SetEvent(g_hevtRequestSubmitted);

        // ждем, когда сервер обработает запрос и сообщит нам результат
        WaitForSingleObject(g_hevtResultReturned, INFINITE);

        // показываем результат пользователю
        Edit_SetText(GetDlgItem(hwnd, IDC_RESULT),
            g_szSharedRequestAndResultBuffer);

        break;
    }
}

//////////////////////////////////////////////////////////////////////////////
    ///

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMSG(hwnd, WM_COMMAND,    Dlg_OnCommand);
    }

    return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////
    ///

int WINAPI _tWinMain(HINSTANCE hInstanceExe, HINSTANCE, PTSTR, int) {

    // создаем и инициализируем два события с автосбросом в занятом
    // состоянии
    g_hevtRequestSubmitted = CreateEvent(NULL, FALSE, FALSE, NULL);
    g.hevtResultReturned   = CreateEvent(NULL, FALSE, FALSE, NULL);

    // порождаем серверный поток
    DWORD dwThreadID;
    HANDLE hThreadServer = chBEGINTHREADEX(NULL, 0, ServerThread, NULL,
        0, &dwThreadID);
```

```
    // отображаем пользовательский интерфейс клиентского потока
    DialogBox(hInstanceExe, MAKEINTRESOURCE(IDD_HANDSHAKE), NULL, Dlg_, Proc);
    g_hMainDlg = NULL;

    // пользовательский интерфейс клиента закрывается - надо завершить
    // серверный поток
    _tcscpy_s(g_szSharedRequestAndResultBuffer
        _countof(g_szSha redRequestAndResultBuffer), g_szServerShutdown);
    SetEvent(g_hevtRequestSubmitted);

    // ждем от серверного потока подтверждения и его завершения
    HANDLE h[2];
    h[0] = g_hevtResultReturned;
    h[1] = hThreadServer;
    WaitForMultipleObjects(2, h, TRUE, INFINITE);

    //проводим должную очистку
    CloseHandle(hThreadServer);
    CloseHandle(g_hevtRequestSubmitted);
    CloseHandle(g_hevtResultReturned);

    // клиентский поток завершается вместе со всем процессом
    return(0);
}

//////////////////////////////// End of File ////////////////////////////////
```

Ожидаемые таймеры

Ожидаемые таймеры (waitable timers) — это объекты ядра, которые самостоятельно переходят в свободное состояние в определенное время или через регулярные промежутки времени. Чтобы создать ожидаемый таймер, достаточно вызвать функцию *CreateWaitableTimer*.

```
HANDLE CreateWaitableTimer(
    PSECURITY_ATTRIBUTES psa,
    BOOL bManualReset,
    PCTSTR pszName);
```

О параметрах *psa* и *pszName* я уже рассказывал в главе 3. Разумеется, любой процесс может получить свой («процессозависимый») описатель существующего объекта «ожидаемый таймер», вызвав *OpenWaitableTimer*.

```
HANDLE OpenWaitableTimer(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

По аналогии с событиями параметр *bManualReset* определяет тип ожидаемого таймера: со сбросом вручную или с автосбросом. Когда освобождается таймер со сбросом вручную, возобновляется выполнение всех потоков, ожидавших этот объект, а когда в свободное состояние переходит таймер с автосбросом — лишь одного из потоков.

Объекты «ожидаемый таймер» всегда создаются в занятом состоянии. Чтобы сообщить таймеру, в какой момент он должен перейти в свободное состояние, вызовите функцию *SetWaitableTimer*.

```
BOOL SetWaitableTimer(
    HANDLE hTimer,
    const LARGE_INTEGER *pDueTime,
    LONG lPeriod,
    PTIMERAPCROUTINE pfnCompletionRoutine,
    PVOID pvArgToCompletionRoutine,
    BOOL bResume);
```

Эта функция принимает несколько параметров, в которых легко запутаться. Очевидно, что *hTimer* определяет нужный таймер. Следующие два параметра (*pDueTime* и *lPeriod*) используются совместно: первый из них задает, когда таймер должен сработать в первый раз, второй определяет, насколько часто это должно происходить в дальнейшем. Попробуем для примера установить таймер так, чтобы в первый раз он сработал 1 января 2008 года в 1:00 PM, а потом срабатывал каждые 6 часов:

```
// объявляем свои локальные переменные
HANDLE hTimer;
SYSTEMTIME st;
FILETIME ftLocal, ftUTC;
LARGE.INTEGER liUTC;

// создаем таймер с автосбросом
hTimer = CreateWaitableTimer(NULL, FALSE, NULL);

// таймер должен сработать в первый раз 1 января 2008 года в 1:00 PM
// по местному времени
st.wYear          = 2008;                    // год
st.wMonth         = 1;                       // январь
st.wDayOfWeek     = 0;                       // игнорируется
st.wDay           = 1;                       // первое число месяца
st.wHour          = 13;                      // 1 PM
st.wMinute        = 0;                       // 0 минут
8t.wSecond        = 0;                       // 0 секунд
st.wMilliseconds  = 0;                       // 0 миллисекунд

SystemTimeToFileTime(&st, &ftLocal);
```

```
// преобразуем местное время в UTC-время
LocalFileTimeToFileTime(&ftLocal, &ftUTC);
// преобразуем FILETIME в LARGE_INTEGER из-за различий в выравнивании данных
liUTC.LowPart = ftUTC.dwLowDateTime;
liUTC.HighPart = ftUTC.dwHighDateTime;

// устанавливаем таймер
SetWaitableTimer(hTimer, &liUTC, 6 * 60 * 60 * 1000,
    NULL, NULL, FALSE); …
```

Этот фрагмент кода сначала инициализирует структуру SYSTEMTIME, определяя время первого срабатывания таймера (его перехода в свободное состояние). Я установил это время как местное. Второй параметр представляется как *const* LARGE_INTEGER* и поэтому не позволяет напрямую использовать структуру SYSTEMTIME. Однако двоичные форматы структур FILETIME и LARGE_INTEGER идентичны: обе содержат по два 32-битных значения. Таким образом, мы можем преобразовать структуру SYSTEMTIME в FILETIME. Другая проблема заключается в том, что функция *SetWaitableTimer* ждет передачи времени в формате UTC (Coordinated Universal Time). Нужное преобразование легко осуществляется вызовом *LocalFileTimeToFileTime*.

Поскольку двоичные форматы структур FILETIME и LARGE_INTEGER идентичны, у вас может появиться искушение передать в *SetWaitableTimer* адрес структуры FILETIME напрямую:

```
// устанавливаем таймер
SetWaitableTimer(hTimer, (PLARGE_INTEGER) &ftUTC,
    6 * 60 * 60 * 1000, NULL, NULL, FALSE);
```

В сущности, разбираясь с этой функцией, я так и поступил. Но это большая ошибка! Хотя двоичные форматы структур FILETIME и LARGE_INTEGER совпадают, выравнивание этих структур осуществляется по-разному. Адрес любой структуры FILETIME должен начинаться на 32-битной границе, а адрес любой структуры LARGE_INTEGER — на 64-битной. Вызов *SetWaitableTimer* с передачей ей структуры FILETIME может сработать корректно, но может и не сработать — все зависит от того, попадет ли начало структуры FILETIME на 64-битную границу. В то же время компилятор гарантирует, что структура LARGE_INTEGER всегда будет начинаться на 64-битной границе, и поэтому правильнее скопировать элементы FILETIME в элементы LARGE_INTEGER, а затем передать в *SetWaitableTimer* адрес именно структуры LARGE_INTEGER.

> **Примечание.** Процессоры x86 всегда «молча» обрабатывают ссылки на невыровненные данные. Поэтому передача в *SetWaitableTimer* адреса структуры FILETIME будет срабатывать, если приложение выполняется на

машине с процессором x86. Однако другие процессоры (например, Alpha) в таких случаях, как правило, генерируют исключение EXCEPTION_DATATYPE_MISALIGNMENT, которое приводит к завершению вашего процесса. Ошибки, связанные с выравниванием данных, — самый серьезный источник проблем при переносе на другие процессорные платформы программного кода, корректно работавшего на процессорах x86. Так что, обратив внимание на проблемы выравнивания данных сейчас, вы сэкономите себе месяцы труда при переносе программы на другие платформы в будущем! Подробнее о выравнивании данных см. главу 13.

Чтобы разобраться в том, как заставить таймер срабатывать каждые 6 часов (начиная с 1Ю0 PM 1 января 2008 года), рассмотрим параметр *lPeriod* функции *SetWaitableTimer*. Этот параметр определяет последующую частоту срабатывания таймера (в мс). Чтобы установить 6 часов, я передаю значение, равное 21600 000 мс (т. е. 6 часов • 60 минут • 60 секунд • 1000 миллисекунд). Между прочим, вызов *SetWatiMeTimer* сработает и при передаче абсолютного времени и даты в прошлом, например 1:00 PM 1 января 1975 г.

Вместо того чтобы устанавливать время первого срабатывания таймера в абсолютных единицах, вы можете задать его в относительных единицах (в интервалах по 100 нс), просто передав отрицательное значение в параметре *pDueTime*. Поскольку мы обычно не пользуемся 100-нс интервалами, возможно вам пригодятся следующие соотношения: 1 с = 1000 мс = 1000 000 мс =10 000 000 интервалов по 100 нс.

Следующий код демонстрирует, как установить таймер на первое срабатывание через 5 секунд после вызова *SetWaitableTimer*.

```
// объявляем свои локальные переменные
HANDLE hTimer;
LARGE_INTEGER li;

// создаем таймер с автосбросом
hTimer * CreateWaitableTimer(NULL, FALSE. NULL);

// Таймер должен сработать через 5 секунд после вызова SetWaitableTimer.
// задаем время в интервалах по 100 нс.
const int nTimerUnitsPerSecond = 10000000;

// Делаем полученное значение отрицательным, чтобы SetWaitableTimer
// знала: нам нужно относительное, а не абсолютное время.
li.QuadPart =-(5 * nTimerUnitsPerSecond);

// устанавливаем таймер
SetWaitableTimer(hTimer. &li, 6 * 60 * 60 = 1000,
    NULL, NULL, FALSE); …
```

Обычно нужно, чтобы таймер сработал только раз — через определенное (абсолютное или относительное) время перешел в свободное состояние и

уже больше никогда не срабатывал. Для этого достаточно передать 0 в параметре *lPeriod*. Затем можно либо вызвать *CloseHandle*, чтобы закрыть таймер, либо перенастроить таймер повторным вызовом *SetWaitableTimer* с другими параметрами.

И о последнем параметре функции *SetWaitableTimer* — *fResume*. Он полезен на компьютерах с поддержкой режима сна. Обычно в нем передают FALSE, и в приведенных ранее фрагментах кода я тоже делал так. Но если вы, скажем, пишете программу-планировщик, которая позволяет устанавливать таймеры для напоминания о запланированных встречах, то должны передавать в этом параметре TRUE. Когда таймер сработает, машина выйдет из режима сна (если она находилась в нем), и пробудятся потоки, ожидавшие этот таймер. Далее программа сможет проиграть какой-нибудь WAV-файл и вывести окно с напоминанием о предстоящей встрече. Если же вы передадите FALSE в параметре *fResume*, объект-таймер перейдет в свободное состояние, но ожидавшие его потоки не получат процессорное время, пока компьютер не выйдет из режима сна (как правило, в результате действия пользователя).

Рассмотрение ожидаемых таймеров было бы неполным, пропусти мы функцию *CancelWaitableTimer*.

```
BOOL CancelWaitableTimer(HANDLE hTimer);
```

Эта очень простая функция принимает описатель таймера и отменяет его (таймер), после чего тот уже никогда не сработает, — если только вы не переустановите его повторным вызовом *SetWaitableTimer*. Кстати, если вам понадобится перенастроить таймер, то вызывать *CancelWaitableTimer* перед повторным обращением к *SetWaitableTimer* не требуется; каждый вызов *SetWaitableTimer* автоматически отменяет предыдущие настройки перед установкой новых.

Ожидаемые таймеры и APC-очередь

Теперь вы знаете, как создавать и настраивать таймер. Вы также научились приостанавливать потоки на таймере, передавая его описатель в *WaitForSingleObjects* или *WaitForMultipleObjects*. Однако у вас есть возможность создать очередь асинхронных вызовов процедур (asynchronous procedure call, APC) для потока, вызывающего *SetWaitableTimer* в момент, когда таймер свободен.

Обычно при обращении к функции *SetWaitableTimer* вы передаете NULL в параметрах *pfnCompletionRoutine* и *pvArgToCompletionRoutine*. В этом случае объект-таймер переходит в свободное состояние в заданное время. Чтобы таймер в этот момент поместил в очередь вызов APC-функции, нужно реализовать данную функцию и передать ее адрес в *SetWaitableTimer*. APC-функция должна выглядеть примерно так:

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompletionRoutine,
    DWORD dwTimerLowValue, DWORD dwTimerHighValue) {

    // здесь делаем то, что нужно
}
```

Я назвал эту функцию *TimerAPCRoutine*, но вы можете назвать ее как угодно. Она вызывается из того потока, который обратился к *SetWaitableTimer* в момент срабатывания таймера, — но только если вызывающий поток находится в «тревожном» (alertable) состоянии, т. е. ожидает этого в вызове одной из функций: *SleepEx*, *WaitForSingleObjectEx*, *WaitForMultipleObjectsEx*, *MsgWaitForMultipleObjectsEx* или *SignalObjectAndWait*. Если же поток этого не ожидает в любой из перечисленных функций, система не поставит в очередь APC-функцию таймера. Тем самым система не даст APC-очереди потока переполниться уведомлениями от таймера, которые могли бы впустую израсходовать колоссальный объем памяти. Подробнее о тревожном ожидании см. в главе 10.

Если в момент срабатывания таймера ваш поток находится в одной из перечисленных ранее функций, система заставляет его вызвать процедуру обратного вызова. Первый ее параметр совпадает с параметром *pvArgToCompletionRoutine*, передаваемым в функцию *SetWaitableTimer*. Это позволяет передавать в *TimerAPCRoutine* какие-либо данные (обычно указатель на определенную вами структуру). Остальные два параметра, *dwTimerLowValue* и *dwTimerHighValue*, задают время срабатывания таймера. Код, приведенный ниже, демонстрирует, как принять эту информацию и показать ее пользователю.

```
VOID APIENTRY TimerAPCRoutine(PVOID pvArgToCompletionRoutine,
    DWORD dwTimerLowValue, DWORD dwTimerHighValue) {

    FILETIME ftUTC, ftLocal;
    SYSTEMTIME st;
    TCHAR szBuf[256];

    // записываем время в структуру FILETIME
    ftUTC.dwLowDateTime = dwTimerLowValue;
    ftUTC.dwHighDateTime = dwTimerHighValue;

    // преобразуем UTC-время в местное
    FileTimeToLocalFileTime(&ftUTC. &ftLocal);

    // Преобразуем структуру FILETIME в структуру SYSTEMTIME,
    // как того требуют функции GetDateFormat и 6etTimeFormat.
    FileTimeToSystemTime(&ftLocal, &st);
    // Формируем строку с датой и временем, в которое
    // сработал таймер.
```

```
    GetDateFormat(LOCALE_USER_DEFAULT, DATE_LONGDATE,
        &st, NULL, szBuf, _countof(szBuf));
    _tcscat_s(szBuf, _countof(szBuf), TEXT(" "));
    GetTimeFormat(LOCALE_USER_DEFAULT, 0,
        &st, NULL, _tcschr(szBuf, TEXT('\0')),
        (int)(_countof(szBuf) - _tcslen(szBuf)));

    // Показываем время пользователю.
    MessageBox(NULL, szBuf, TEXT("Timer went off at…"), MB_OK);
}
```

Функция «тревожного ожидания» возвращает управление только после обработки всех элементов APC-очереди. Поэтому вы должны позаботиться о том, чтобы ваша функция *TimerAPCRoutine* заканчивала свою работу до того, как таймер вновь подаст сигнал (перейдет в свободное состояние). Иначе говоря, элементы не должны ставиться в APC-очередь быстрее, чем они могут быть обработаны.

Следующий фрагмент кода показывает, как правильно пользоваться таймерами и APC:

```
void SomeFunc() {

    // создаем таймер (его тип не имеет значения)
    HANDLE hTimer = CreateWaitableTimer(NULL, TRUE, NULL);

    // настраиваем таймер на срабатывание через 5 секунд
    LARGE_INTEGER li = { 0 };
    SetWaitableTimer(hTimer, &li, 5000, TimerAPCRoutine, NULL, FALSE);

    // ждем срабатывания таймера в "тревожном" состоянии
    SleepEx(INFINITE, TRUE);

    CloseHandle(hTimer);
}
```

И последнее. Взгляните на этот фрагмент кода:

```
HANDLE hTimer = CreateWaitableTimer(NULL, FALSE, NULL);
SetWaitableTimer(hTimer, …, TimerAPCRoutine, …);
WaitForSingleObjectEx(hTimer, INFINITE, TRUE);
```

Никогда не пишите такой код, потому что вызов *WaitForSingleObjectEx* на деле заставляет дважды ожидать таймер — по описателю *hTimer* и в «тревожном» состоянии. Когда таймер перейдет в свободное состояние, поток пробудится, что выведет его из «тревожного» состояния, и вызова APC-функции не последует. Правда, APC-функции редко используются совместно с ожидаемыми таймерами, так как всегда можно дождаться перехода таймера в свободное состояние, а затем сделать то, что нужно.

И еще кое-что о таймерах

Таймеры часто применяются в коммуникационных протоколах. Например, если клиент делает запрос серверу и тот не отвечает в течение определенного времени, клиент считает, что сервер не доступен. Сегодня клиентские машины взаимодействуют, как правило, со множеством серверов одновременно. Если бы объект ядра «таймер» создавался для каждого запроса, производительность системы снизилась бы весьма заметно. В большинстве приложений можно создавать единственный объект-таймер и по мере необходимости просто изменять время его срабатывания.

Постоянное отслеживание параметров таймера и его перенастройка довольно утомительны, из-за чего реализованы лишь в немногих приложениях. Однако в числе новых функций для операций с пулами потоков (о них — в главе 11) появилась *CreateTimerQueueTimer* — она как раз и берет на себя всю эту рутинную работу. Присмотритесь к ней, если в вашей программе приходится создавать несколько объектов-таймеров и управлять ими.

Конечно, очень мило, что таймеры поддерживают APC-очереди, но большинство современных приложений использует не APC, а порты завершения ввода-вывода. Как-то раз мне понадобилось, чтобы один из потоков в пуле (управляемом через порт завершения ввода-вывода) пробуждался по таймеру через определенные интервалы времени. К сожалению, такую функциональность ожидаемые таймеры не поддерживают. Для решения этой задачи мне пришлось создать отдельный поток, который всего-то и делал, что настраивал ожидаемый таймер и ждал его освобождения. Когда таймер переходил в свободное состояние, этот поток вызывал *PostQueuedCompletionStatus*, передавая соответствующее уведомление потоку в пуле.

Любой, мало-мальски опытный Windows-программист непременно поинтересуется различиями ожидаемых таймеров и таймеров User (настраиваемых через функцию *SetTimer*). Так вот, главное отличие в том, что ожидаемые таймеры реализованы в ядре, а значит, не столь тяжеловесны, как таймеры User. Кроме того, это означает, что ожидаемые таймеры — объекты разделяемые и защищенные.

Таймеры User генерируют сообщения WM_TIMER, посылаемые тому потоку, который вызвал *SetTimer* (в случае таймеров с обратной связью) или создал определенное окно (в случае оконных таймеров). Таким образом, о срабатывании таймера User уведомляется только один поток. А ожидаемый таймер позволяет ждать любому числу потоков, и, если это таймер со сбросом вручную, при его освобождении может пробуждаться сразу несколько потоков.

Если в ответ на срабатывание таймера вы собираетесь выполнять какие-то операции, связанные с пользовательским интерфейсом, то, по-видимому, будет легче структурировать код под таймеры User, поскольку применение ожидаемых таймеров требует от потоков ожидания не только сообщений, но и объектов ядра. (Если у вас есть желание переделать свой код, используйте функцию *MsgWaitForMultipleObjects*, которая как раз и рассчитана на

такие ситуации.) Наконец, в случае ожидаемых таймеров вы с большей вероятностью будете получать уведомления именно по истечении заданного интервала. Сообщения WM_TIMER всегда имеют наименьший приоритет и принимаются, только когда в очереди потока нет других сообщений. Но ожидаемый таймер обрабатывается так же, как и любой другой объект ядра: если он сработал, ждущий поток немедленно пробуждается.

Семафоры

Объекты ядра «семафор» используются для учета ресурсов. Как и все объекты ядра, они содержат счетчик числа пользователей, но, кроме того, поддерживают два 32-битных значения со знаком: одно определяет максимальное число ресурсов (контролируемое семафором), другое используется как счетчик текущего числа ресурсов.

Попробуем разобраться, зачем нужны все эти счетчики, и для примера рассмотрим программу, которая могла бы использовать семафоры. Допустим, я разрабатываю серверный процесс, в адресном пространстве которого выделяется буфер для хранения клиентских запросов. Размер этого буфера «зашит» в код программы и рассчитан на хранение максимум пяти клиентских запросов. Если новый клиент пытается связаться с сервером, когда эти пять запросов еще не обработаны, генерируется ошибка, которая сообщает клиенту, что сервер занят и нужно повторить попытку позже. При инициализации мой серверный процесс создает пул из пяти потоков, каждый из которых готов обрабатывать клиентские запросы по мере их поступления.

Изначально, когда запросов от клиентов еще нет, сервер не разрешает выделять процессорное время каким-либо потокам в пуле. Но как только серверу поступает, скажем, три клиентских запроса одновременно, три потока в пуле становятся планируемыми, и система начинает выделять им процессорное время. Для слежения за ресурсами и планированием потоков семафор очень удобен. Максимальное число ресурсов задается равным 5, что соответствует размеру буфера. Счетчик текущего числа ресурсов первоначально получает нулевое значение, так как клиенты еще не выдали ни одного запроса. Этот счетчик увеличивается на 1 в момент приема очередного клиентского запроса и на столько же уменьшается, когда запрос передается на обработку одному из серверных потоков в пуле.

Для семафоров определены следующие правила:

- когда счетчик текущего числа ресурсов становится больше 0, семафор переходит в свободное состояние;
- если этот счетчик равен 0, семафор занят;
- система не допускает присвоения отрицательных значений счетчику текущего числа ресурсов;
- счетчик текущего числа ресурсов не может быть больше максимального числа ресурсов.

Не путайте счетчик текущего числа ресурсов со счетчиком числа пользователей объекта-семафора.

```
HANDLE CreateSemaphore(
    PSECURITY_ATTRIBUTE psa,
    L0NG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName);
```

О параметрах *psa* и *pszName* я рассказывал в главе 3. Следующая функция позволяет напрямую предоставить права доступа, указав их в параметре *dwDesiredAccess*. Заметьте, что параметр *dwFlags* зарезервирован и должен быть установлен в 0.

```
HANDLE CreateSemaphoreEx(
    PSECURnY_ATTRIBUTES psa,
    L0NG lInitialCount,
    LONG lMaximumCount,
    PCTSTR pszName,
     DWORD dwFlags,
    DWORD dwDesiredAccess);
```

Разумеется, любой процесс может получить свой («процессозависимый») описатель существующего объекта «семафор», вызвав *OpenSemaphore*.

```
HANDLE OpenSemaphore(
    DWORD dwDesiredAccess,
    BOOL bInheritHandle,
    PCTSTR pszName);
```

Параметр *lMaximumCount* сообщает системе максимальное число ресурсов, обрабатываемое вашим приложением. Поскольку это 32-битное значение со знаком, предельное число ресурсов может достигать 2 147 483 647. Параметр *lInitialCount* указывает, сколько из этих ресурсов доступно изначально (наданный момент). При инициализации моего серверного процесса клиентских запросов нет, поэтому я вызываю *CreateSemaphore* так:

```
HANDLE hSemaphore = CreateSemaphore(NULL, 0, 5, NULL);
```

Это приводит к созданию семафора со счетчиком максимального числа ресурсов, равным 5, при этом изначально ни один ресурс не доступен. (Кстати, счетчик числа пользователей данного объекта ядра равен 1, так как я только что создал этот объект; не запутайтесь в счетчиках.) Поскольку счетчику текущего числа ресурсов присвоен 0, семафор находится в занятом состоянии. А это значит, что любой поток, ждущий семафор, просто засыпает.

Поток получает доступ к ресурсу, вызывая одну из *Wait*-функций и передавая ей описатель семафора, который охраняет этот ресурс. *Wait*-функция проверяет у семафора счетчик текущего числа ресурсов: если его значение больше 0 (семафор свободен), уменьшает значение этого счетчика на 1, и

вызывающий поток остается планируемым. Очень важно, что семафоры выполняют эту операцию проверки и присвоения на уровне атомарного доступа; иначе говоря, когда вы запрашиваете у семафора какой-либо ресурс, операционная система проверяет, доступен ли этот ресурс, и, если да, уменьшает счетчик текущего числа ресурсов, не позволяя вмешиваться в эту операцию другому потоку. Только после того как счетчик ресурсов будет уменьшен на 1, доступ к ресурсу сможет запросить другой поток.

Если *Wait*-функция определяет, что счетчик текущего числа ресурсов равен 0 (семафор занят), система переводит вызывающий поток в состояние ожидания. Когда другой поток увеличит значение этого счетчика, система вспомнит о ждущем потоке и снова начнет выделять ему процессорное время (а он, захватив ресурс, уменьшит значение счетчика на 1).

Поток увеличивает значение счетчика текущего числа ресурсов, вызывая функцию *ReleaseSemaphore*:

```
BOOL ReleaseSemaphore(
    HANDLE hSemaphore,
    LONG lReleaseCount,
    PLONG plPreviousCount);
```

Она просто складывает величину *lReleaseCount* со значением счетчика текущего числа ресурсов. Обычно в параметре *lReleaseCount* передают 1, о это вовсе не обязательно: я часто передаю в нем значения, равные или большие 2. Функция возвращает исходное значение счетчика ресурсов в **plPreviousCount*. Если вас не интересует это значение (а в большинстве программ так оно и есть), передайте в параметре **plPreviousCount* значение NULL.

Было бы удобнее определять состояние счетчика текущего числа ресурсов, не меняя его значение, но такой функции в Windows нет. Поначалу я думал, что вызовом *ReleaseSemaphore* с передачей ей во втором параметре нуля можно узнать истинное значение счетчика в переменной типа LONG, на которую указывает параметр **plPreviousCount*. Но не вышло: функция занесла туда нуль. Я передал во втором параметре заведомо большее число, и — тот же результат. Тогда мне стало ясно: получить значение этого счетчика, не изменив его, невозможно.

Мьютексы

Объекты ядра «мьютексы» гарантируют потокам взаимоисключающий доступ к единственному ресурсу. Отсюда и произошло название этих объектов (mutual exclusion, mutex). Они содержат счетчик числа пользователей, счетчик рекурсии и переменную, в которой запоминается идентификатор потока. Мьютексы ведут себя точно так же, как и критические секции. Но если последние являются объектами пользовательского режима (см. главу 8), то мьютексы — объектами ядра. Кроме того, единственный объект-мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из раз-

ных процессов; при этом можно задать максимальное время ожидания доступа к ресурсу.

Идентификатор потока определяет, какой поток захватил мьютекс, а счетчик рекурсий — сколько раз. У мьютексов много применений, и это наиболее часто используемые объекты ядра. Как правило, с их помощью защищают блок памяти, к которому обращается множество потоков. Если бы потоки одновременно использовали какой-то блок памяти, данные в нем были бы повреждены. Мьютексы гарантируют, что любой поток получает монопольный доступ к блоку памяти, и тем самым обеспечивают целостность данных.

Для мьютексов определены следующие правила:

- если его идентификатор потока равен 0 (у самого потока не может быть такой идентификатор), мьютекс нс захвачен ни одним из потоков и находится в свободном состоянии;
- если его идентификатор потока не равен 0, мьютекс захвачен одним из потоков и находится в занятом состоянии;
- в отличие от других объектов ядра мьютексы могут нарушать обычные правила, действующие в операционной системе (об этом — чуть позже).

Для использования объекта-мьютекса один из процессов должен сначала создать его вызовом *CreateMutex*.

```
HANDLE CreateMutex(
    PSECURITY_ATTRIBUTES psa,
    B00L MnitialOwner,
    PCTSTR pszName);
```

О параметрах *psa* и *pszName* я рассказывал в главе 3. Следующая функция позволяет напрямую передать права доступа, указав их в параметре *dwDesiredAccess*. Параметр *dwFlags* — замена параметра *bInitialOwned* функции *CreateMutex*: 0 означает FALSE, а CREATE_MUTEX_INITIAL_OWNER — TRUE.

```
HAN0LE CreateHutexEx(
    PSECURITY_ATTRIBUTES psa,
    PCTSTR pszName,
    DWORD dwFlags,
    DWORD dwDesiredAccess);
```

Разумеется, любой процесс может получить свой («процессозависимый») описатель существующего объекта «мьютекс», вызвав *OpenMutex*.

```
HANDLE OpenMutex(
    DWORD dwDesiredAccess,
    B00L bInheritHandle,
    PCTSTR pszName);
```

Параметр *bfInitialOwner* определяет начальное состояние мьютекса. Если в нем передается FALSE (что обычно и бывает), объект-мьютекс не прина-

длежит ни одному из потоков и поэтому находится в свободном состоянии. При этом его идентификатор потока и счетчик рекурсии равны 0. Если же в нем передается TRUE, идентификатор потока, принадлежащий мьютексу, приравнивается идентификатору вызывающего потока, а счетчик рекурсии получает значение 1. Поскольку теперь идентификатор потока отличен от 0, мьютекс изначально находится в занятом состоянии.

Поток получает доступ к разделяемому ресурсу, вызывая одну из Wait-функций и передавая ей описатель мьютекса, который охраняет этот ресурс. Wait-функция проверяет у мьютекса идентификатор потока: если его значение не равно 0, мьютекс свободен; в ином случае оно принимает значение идентификатора вызывающего потока, и этот поток остается планируемым.

Если *Wait*-функция определяет, что у мьютекса идентификатор потока не равен 0 (мьютекс занят), вызывающий поток переходит в состояние ожидания. Система запоминает это и, когда идентификатор обнуляется, записывает в него идентификатор ждущего потока, а счетчику рекурсии присваивает значение 1, после чего ждущий поток вновь становится планируемым. Все проверки и изменения состояния объекта-мьютекса выполняются на уровне атомарного доступа.

Для мьютексов сделано одно исключение в правилах перехода объектов ядра из одного состояния в другое. Допустим, поток ждет освобождения занятого объекта-мьютекса. В этом случае поток обычно засыпает (переходит в состояние ожидания). Однако система проверяет, не совпадает ли идентификатор потока, пытающегося захватить мьютекс, с аналогичным идентификатором у мьютекса. Если они совпадают, система по-прежнему выделяет потоку процессорное время, хотя мьютекс все еще занят. Подобных особенностей в поведении нет ни у каких других объектов ядра в системе. Всякий раз, когда поток захватывает объект-мьютекс, счетчик рекурсии в этом объекте увеличивается на 1. Единственная ситуация, в которой значение счетчика рекурсии может быть больше 1, — поток захватывает один и тот же мьютекс несколько раз, пользуясь упомянутым исключением из общих правил.

Когда ожидание мьютекса потоком успешно завершается, последний получает монопольный доступ к защищенному ресурсу. Все остальные потоки, пытающиеся обратиться к этому ресурсу, переходят в состояние ожидания. Когда поток, занимающий ресурс, заканчивает с ним работать, он должен освободить мьютекс вызовом функции *ReleaseMutex*.

```
BOOL ReleaseMutex(HANDLE hMutex);
```

Эта функция уменьшает счетчик рекурсии в объекте-мьютексе на 1. Если данный объект передавался во владение потоку неоднократно, поток обязан вызвать *ReleaseMutex* столько раз, сколько необходимо для обнуления счетчика рекурсии. Как только счетчик станет равен 0, переменная, хранящая идентификатор потока, тоже обнулится, и объект-мьютекс освободится. После этого система проверит, ожидают ли освобождения мьютекса какие-нибудь другие потоки. Если да, система «по-честному» выберет один из

ждущих потоков и передаст ему во владение объект-мьютекс. Естественно, это означает, что в него будет записан идентификатор выбранного потока, а счетчик рекурсии — установлен в 1. Если ждущих потоков нет, мьютекс остается свободным и будет немедленно передан первому же потоку, который станет его ожидать.

Отказ от объекта-мьютекса

Объект-мьютекс отличается от остальных объектов ядра тем, что занявшему его потоку передаются права на владение им. Прочие объекты могут быть либо свободны, либо заняты — вот, собственно, и все. А объекты-мьютексы способны еще и запоминать, какому потоку они принадлежат. Именно поэтому для мьютексов сделано исключение: поток может завладеть даже занятым мьютексом.

Особенности мьютексов касаются не только их захвата, но и освобождения. Если какой-то посторонний поток попытается освободить мьютекс вызовом функции *ReleaseMutex*, то она, проверив идентификаторы потоков и обнаружив их несовпадение просто вернет FALSE. Тут же вызвав *GetLastError*, вы получите значение ERROR_NOT_OWNER.

Отсюда возникает вопрос: а что будет, если поток, которому принадлежит мьютекс, завершится, не успев его освободить? В таком случае система считает, что произошел отказ от мьютекса, и автоматически переводит его в свободное состояние (сбрасывая при этом все его счетчики в исходное состояние). Система отслеживает все мьютексы и объекты ядра «поток», поэтому она точно знает, когда поток отказывается от мьютекса. У таких мьютексов система автоматически сбрасывает идентификатор потока-владельца и счетчик рекурсии до 0. Если этот мьютекс ждут другие потоки, система, как обычно, «по-честному» выбирает один из потоков и позволяет ему захватить мьютекс, записывая в мьютекс идентификатор этого потока и увеличивая счетчик рекурсии до 1. В результате этот поток становится планируемым.

Тогда *Wait*-функция возвращает потоку WAIT_ABANDONED вместо WAIT_OBJECT_0, и тот узнает, что мьютекс освобожден некорректно. Данная ситуация, конечно, не самая лучшая. Выяснить, что сделал с защищенными данными завершенный поток — бывший владелец объекта-мьютекса, увы, невозможно. Не исключено, что они необратимо повреждены. Что должно делать приложение в таких случаях — решать вам.

В реальности программы никогда специально нс проверяют возвращаемое значение на WAIT_ABANDONED, потому что такое завершение потоков происходит очень редко. (Вот, кстати, еще один яркий пример, доказывающий, что вы не должны пользоваться функцией *TerminateThread*.)

Мьютексы и критические секции

Мьютексы и критические секции одинаковы в том, как они влияют на планирование ждущих потоков, но различны по некоторым другим характеристикам. Эти объекты сравниваются в следующей таблице.

Табл. 9-2. Сравнение мьютексов и критических секций

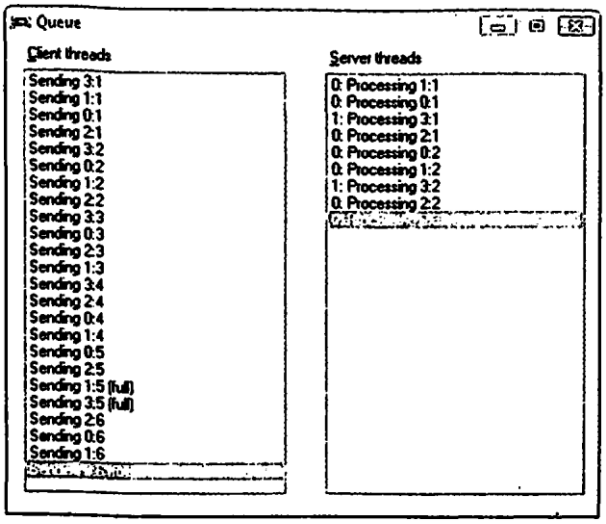
Характеристики	Объект-мьютекс	Объект — критическая секция
Быстродействие	Малое	Высокое
Возможность использования за границами процесса	Да	Нет
Объявление	*HANDLE hmtx*	*CRITICAL_SECTION cs;*
Инициализация	*hmtx = CreateMutex (NULL, FALSE, NULL);*	*InitializeCriticalSection (&cs);*
Очистка	*CloseHandle(hmtx);*	*DeleteCriticalSection(&cs);*
Бесконечное ожидание	*WaitForSingleObject (hmtx, INFINITE);*	*EnterCriticalSection(&cs);*
Ожидание в течение 0 мс	*WaitForSingleObject (hmtx, 0);*	*TryEnterCriticalSection(&cs);*
Ожидание в течение произвольного периода времени	*WaitForSingleObject (hmtx, dwMilliseconds);*	Невозможно
Освобождение	*ReleaseMutex(hmtx);*	*LeaveCriticalSection (&cs);*
Возможность параллельного ожидания других объектов ядра	Да (с помощью *WaitForMultipleObjects* или аналогичной функции)	Нет

Программа-пример Queue

В главе 8 на примере одноименной программы я продемонстрировал управление очередью с помощью *SRWLock* и условных переменных. В этой главе мы рассмотрим версию Queue (09 Queue.exe, исходный текст и файлы ресурсов см. в каталоге 09-Queue архива, доступного на веб-сайте поддержки книги). Эта программа безопасна в многопоточной среде, кроме того, ею проще манипулировать из других потоков. Она управляет очередью обрабатываемых элементов данных, используя мьютекс и семафор. Файлы исходного кода и ресурсов этой программы находятся в каталоге 09-Queue на компакт-диске, прилагаемом к книге. После запуска Queue открывается окно, показанное ниже.

Как и в примере из 8 главы, при инициализации Queue создает четыре клиентских и два серверных потока. Каждый клиентский поток засыпает на определенный период времени, а затем помещает в очередь элемент данных. Когда в очередь ставится новый элемент, содержимое списка Client Threads обновляется. Каждый элемент данных состоит из номера клиентского потока и порядкового номера запроса, выданного этим потоком. Например, первая запись в списке сообщает, что клиентский поток 0 поставил в очередь свой первый запрос. Следующие записи свидетельствуют, что далее свои

первые запросы выдают потоки 1-3, потом поток 0 помещает второй запрос, то же самое делают остальные потоки, и все повторяется.



Серверные потоки ничего не делают, пока в очереди не появится хотя бы один элемент данных. Как только он появляется, для его обработки пробуждается один из серверных потоков. Состояние серверных потоков отражается в списке Server Threads. Первая запись говорит о том, что первый запрос от клиентского потока 0 обрабатывается серверным потоком 0, вторая запись — что первый запрос от клиентского потока 1 обрабатывается серверным потоком 1, и т. д.

В этом примере серверные потоки не успевают обрабатывать клиентские запросы и очередь в конечном счете заполняется до максимума. Я установил максимальную длину очереди равной 10 элементам, что приводит к быстрому заполнению этой очереди. Кроме того, на четыре клиентских потока приходится лишь два серверных. В итоге очередь полностью заполняется к тому моменту, когда клиентский поток 3 пытается выдать свой пятый запрос.

Итак, что делает программа, вы поняли; теперь посмотрим — как она это делает (что гораздо интереснее). Очередью управляет C++-класс CQueue:

```
class CQueue {
public:
    Struct ELEMENT {
        int m_nThreadNum, m_nRequestNum;
        // Другие элементы данных должны быть определены здесь
    };
    typedef ELEMENT* PELEMENT;

private:
    PELEMENT m_pElements;              // массив элементов, подлежащих обработке
    int      m_nMaxElements;           // количество элементов в массиве
```

```
    HANDLE    m_h[2];                    // описатели мьютекса и семафора
    HANDLE    &m_hmtxQ;                  // ссылка на m_h[0]
    HANDLE    &m_hsemNumElements;        // ссылка на  m_h[1]

public:
    CQueue(int nMaxElements);
    ~CQueue();

    BOOL Append(PELEMENT pElement, DWORD dwMilliseconds);
    BOOL Remove(PELEMENT pElement, DWORD dwMilliseconds);
};
```

Открытая структура ELEMENT внутри этого класса определяет, что представляет собой элемент данных, помещаемый в очередь. Его реальное содержимое в данном случае не имеет значения. В этой программе-примере клиентские потоки записывают в элемент данных собственный номер и порядковый номер своего очередного запроса, а серверные потоки, обрабатывая запросы, показывают эту информацию в списке. В реальном приложении такая информация вряд ли бы понадобилась.

Что касается закрытых элементов класса, мы имеем *m_pElements*, который указывает на массив (фиксированного размера) структур ELEMENT. Эти данные как раз и нужно защищать от одновременного доступа к ним со стороны клиентских и серверных потоков. Элемент *m_nMaxElements* определяет размер массива при создании объекта CQueue. Следующий элемент, *m_h*, — это массив из двух описателей объектов ядра. Для корректной защиты элементов данных в очереди нам нужно два объекта ядра: мьютекс и семафор. Эти два объекта создаются в конструкторе CQueue; в нем же их описатели помещаются в массив *m_h*.

Как вы вскоре увидите, программа периодически вызывает *WaitForMultipleObjects*, передавая этой функции адрес массива описателей. Вы также убедитесь, что программе время от времени приходится ссылаться только на один из этих описателей. Чтобы облегчить чтение кода и его модификацию, я объявил два элемента, каждый из которых содержит ссылку на один из описателей, — *m_hmtxQ* и m_*hsemNumElements*. Конструктор CQueue инициализирует эти элементы содержимым *m_h[0]* и *m_h[1]* соответственно.

Теперь вы и сами без труда разберетесь в методах конструктора и деструктора CQueue, поэтому я перейду сразу к методу *Append*. Этот метод пытается добавить ELEMENT в очередь. Но сначала он должен убедиться, что вызывающему потоку разрешен монопольный доступ к очереди. Для этого метод *Append* вызывает *WaitForSingleObject*, передавая ей описатель объекта-мьютекса, *m_hmtxQ*. Если функция возвращает WAIT_OBJECT_0, значит, поток получил монопольный доступ к очереди.

Далее метод *Append* должен попытаться увеличить число элементов в очереди, вызвав функцию ReleaseSemaphore и передав ей счетчик числа освобождений (release count), равный 1. Если вызов *ReleaseSemaphore* прохо-

дит успешно, в очереди еще есть место, и в нее можно поместить новый элемент. К счастью, *ReleaseSemaphore* возвращает в переменной *lPreviousCount* предыдущее количество элементов в очереди. Благодаря этому вы точно знаете, в какой элемент массива следует записать новый элемент данных. Скопировав элемент в массив очереди, функция возвращает управление. По окончании этой операции *Append* вызывает *ReleaseMutex*, чтобы и другие потоки могли получить доступ к очереди. Остальной код в методе *Append* отвечает за обработку ошибок и неудачных вызовов.

Теперь посмотрим, как серверный поток вызывает метод *Remove* для выборки элемента из очереди. Сначала этот метод должен убедиться, что вызывающий поток получил монопольный доступ к очереди и что в ней есть хотя бы один элемент. Разумеется, серверному потоку нет смысла пробуждаться, если очередь пуста. Поэтому метод *Remove* предварительно обращается к *WaitForMultipteObjects*, передавая ей описатели мьютекса и семафора. И только после освобождения обоих объектов серверный поток может пробудиться.

Если возвращается WAIT_OBJECT_0, значит, поток получил монопольный доступ к очереди и в ней есть хотя бы один элемент. В этот момент программа извлекает из массива элемент с индексом 0, а остальные элементы сдвигает вниз на одну позицию. Это, конечно, не самый эффективный способ реализации очереди, так как требует слишком большого количества операций копирования в памяти, но наша цель заключается лишь в том, чтобы продемонстрировать синхронизацию потоков. По окончании этих операций вызывается *ReleaseMutex*, и очередь становится доступной другим потокам.

Заметьте, что объект-семафор отслеживает, сколько элементов находится в очереди. Вы, наверное, сразу же поняли, что это значение увеличивается, когда метод *Append* вызывает *ReleaseSemaphore* после добавления нового элемента к очереди. Но как оно уменьшается после удаления элемента из очереди, уже не столь очевидно. Эта операция выполняется вызовом *WaitForMultipleObjects* из метода *Remove*. Тут надо вспомнить, что побочный эффект успешного ожидания семафора заключается в уменьшении его счетчика на 1. Очень удобно для нас.

Теперь, когда вы понимаете, как работает класс CQueue, вы легко разберетесь в остальном коде этой программы.

```
Queue.cpp

/******************************************************************************

Module:   Queue.cpp
Notices:  Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..CommonFiles\CmhHdr.h"           /* см. приложение A */
#include <windowsx.h>
```

```
#include <tchar.h>
#include <StrSafe.h>
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////
    ///

class CQueue {
public:
    struct ELEMENT {
        int ro_nThreadNum, m_nRequestNum;
        // другие элементыданных должны быть определены здесь
    };
    Typedef ELEMENT* PELEMENT;

private:
    PELEMENT  m_pElements;              // массив элементов, подлежащих обра-
        ботке
    int       m_nMaxElements;           // количество элементов в массиве
    HANDLE    m_h[2];                   // описатели мьютекса и семафора
    HANDLE    &m_hmtxQ;                 // ссылка на m_h[0]
    HANDLE    &m_hsemNumElements;       // ссылка на m_h[1]

public:
    CQueue(int nMaxElements);
    ~CQueue();

    BOOL Append(PELEMENT pElement, DWORD dwMilliseconds);
    BOOL Remove(PELEMENT pElement, DWORD dwMilliseconds);
};

//////////////////////////////////////////////////////////////////////////
    ///

CQueue::CQueue(int nMaxElements) : m_hmtxQ(m_h[0]), m_hsemNumElements(m_h[1]) {

    m_pElements =
        (PELEMENT)HeapAlloc(GetProcessHeap(), 0, sizeof(ELEMENT) * nMaxEle-
        ments);
    m_nMaxElements = nMaxElements;
    m_hmtxQ = CreateMutex(NULL, FALSE, NULL);
    m_hsemNumElements = CreateSemaphore(NULL, 0, nMaxElements, NULL);
}

//////////////////////////////////////////////////////////////////////////
    ///

CQueue::~CQueue() {

    CloseHandle(m_hsemNumElements);
    CloseHandle(m_hmtxQ);
```

```
    HeapFree(GetProcessHeap(), 0, m_pElements);
}

//////////////////////////////////////////////////////////////////////////////
    ///

BOOL CQueue::Append(PELEMENT pElement, DWORD dwTimeout) {

    BOOL fOk = FALSE;
    DWORD dw = WaitForSingleObject(n_hmtxQ, dwTimeout);

    if (dw == WAIT_OBJECT_O) {
        // этот поток получил монопольный доступ к очереди

        // увеличиваем число элементов в очереди
        LONG lPrevCount;
        fOk = ReleaseSemaphore(m_hsemNumElements, 1, &lPrevCount);
        if (fOk) {
            // в очереди еще есть место; добавляем новый элемент
            nLpElements[lPrevCount] = *pElement;
        } else {

            // Очередь полностью заполнена; устанавливаем код ошибки
            // и сообщаем о неудачном завершении вызова.
            SetLastError(ERROR_DATABASE_FULL);
        }

        // Разрешаем другим потокам обращаться к очереди
        ReleaseMutex(m_hmtxQ);

    } else {

        // Время ожидания истекло; устанавливаем код ошибки
        // и сообщаем о неудачном завершении вызова.
        SetLastError(ERROR_TIMEOUT);
    }

    return(fOk);                  // GetLastError сообщит дополнительную инфор-
        мацию
}

//////////////////////////////////////////////////////////////////////////////
    ///

BOOL CQueue::Remove(PELEMENT pElement, DWORD dwTimeout) {

    // Ждем монопольного доступа к очереди
    // и появления в ней хотя бы одного элемента.
    BOOL fOk = (WaitForMultipleObjects(_countof(m_h), m_h, TRUE, dwTimeout)
        == WAIT_0BJECT_0);
```

```
    if (fOk) {
        // в очереди есть элемент; извлекаем его
        *pElement = m_pElements[0];

        // сдвигаем остальные элементы вниз на одну позицию
        MoveMemory(&m_pElements[0], &m_pElements[1],
            sizeof(ELEMENT) * (m_nMaxElements - 1));

        // разрешаем другим потокам обращаться к очереди
        ReleaseMutex(m_hmtxQ);

    } else {
        // Время ожидания истекло; устанавливаем код ошибки
        // и сообщаем о неудачном завершении вызова.
        SetLastError(ERROR_TIMEOUT);
    }

    return(fOk);                  //GetLastError сообщит дополнительную информа-
        цию
}

////////////////////////////////////////////////////////////////////////////
    ///

CQueue g_q(10);                             // совместно используемая очередь
volatile BOOL g_fShutdown = FALSE;          // сигнализирует клиентским
                                            // и серверным потокам,
                                            // когда им нужно завершаться;
HWND g_hwnd;                                // позволяет выяснять состояние
                                            // клиентских и серверных потоков.

// описатели и количество всех потоков (клиентских и серверных)
HANDLE g_hTh reads[MAXIMUM_WAIT_OBJECTS];
int    g_nNumThreads = 0;

////////////////////////////////////////////////////////////////////////////
    ///

DWORD WINAPI ClientThread(PVOID pvParam) {
    int nThreadNum = PtrToUlong(pvParam);
    HWND hwndLB = GetDlgItem(g_hwnd, IDC_CLIENTS);
    int nRequestNum = 0;
    while ((PV0ID)1 !=
        InterlockedCompareExchangePointer(
            (PV0ID*) &g_fShutdown, (PV0ID)0, (PV0ID)0)) {

        // Отслеживаем элемент, обрабатываемый в данное время
        nRequestNum++;
```

```
        TCHAR sz[1024];
        CQueue::ELEMENT e = { nThreadNum, nRequestNum };

        // пытаемся поместить элемент в очередь
        if (g_q.Append(&e, 200)) {

            // указываем номера потока и запроса
            StringCchPrintf(sz, _countof(sz), TEXT("Sending %d:%d"),
                nThreadNum, nRequestNum);
        } else {

            // поставить элемент в очередь не удалось
            StringCchPrintf(sz, _countof(sz), TEXT("Sending %d:%d (%s)"),
                nThreadNum, nRequestNum, (GetLastError() == ERROR_TIMEOUT)
                    ? TEXT("timeout") : TEXT("full"));
        }

        // показываем результат добавления элемента
        ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
        Sleep(2500);                          // интервал ожидания до добавления
                                              // следующего элемента
    }

    return(0);
}

//////////////////////////////////////////////////////////////////////////////
    ///

DWORD WINAPI ServerThread(PVOID pvParam) {

    int nThreadNum = PtrToUlong(pvParam);
    HWND hwndLB = GetDlgItem(g_hwnd, IDC_SERVERS);
    while ((PV0ID)1 !=
        InterlockedCompareExchangePointer(
            (PVOID*) &g_fShutdown, (PVOID)0, (PVOID)0)) {

        TCHAR sz[1024];
        CQueue::ELEMENT e;

        // пытаемся получить элемент из очереди
        if (g_q.Remove(&e, 5000)) {

            // Сообщаем, какой поток обрабатывает этот элемент,
            // какой лоток поместил его в очередь и какой он по счету.
            StringCchPrintf(sz, _countof(sz), TEXT("%: Processing %d:%d"),
                nThreadNum, e.m_nThreadNum, e.m_nRequestNum);
```

```
            // на обработку запроса серверу нужно какое-то время
            Sleep(2000 * e.m_nThreadNum);

        } else {
            // получить элемент из очереди не удалось
            StringCchPrintf(sz, _countof(sz), TEXT("%d: (timeout)"),
                nThreadNum);
        }

        // показываем результат обработки элемента
        ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz)};
    }

    return(0);
}

//////////////////////////////////////////////////////////////////////////////
    ///

BOOL Dlg__OnInitDialog(HWND hwnd, HWND hwndFocus, LPARAM lParam) {

    chSETDLGICONS(hwnd, IDI_QUEUE);

    g_hwnd = &hwnd;              Используется клиентскими и серверными потоками
                                 // для уведомления о своем состоянии.

    DWORD dwThreadID;

    // создаем клиентские потоки
    for (int x = 0; x < 4; x++)
        g_hThreads[g_nNumThreads++] =
            chBEGINTHREADEX(NULL, 0, CientThread, (PVOID) (INT_PTR) x,
            0, &dwThreadID);

    // создаем серверные потоки
    for (int x = 0; x < 2; x++)
        gJrThreadsEg_nNumThreads++] =
            chBEGINTHREADEX(NULL, 0, ServerThread, (PVOID) (INT_PTR) x,
            0, &dwThreadID);

    return(TRUE);
}
//////////////////////////////////////////////////////////////////////////////
    ///

void Dlg_OnCownmnd(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify)
{
```

```
    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;
    }
}

//////////////////////////////////////////////////////////////////////////
    ///

INT_PTR WINAPI Dlg_Proc(HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg.OnInitDialog);
        chHANDLE_DLGMSG(hwnd, WM_COMNAND,    Dlg_OnCommand);
    }
    return(FALSE);
}

//////////////////////////////////////////////////////////////////////////
    ///

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_QUEUE), NULL, Dlg_Proc);
    InterlockedExchangePointer(&g_fShutdown, TRUE);

    // ждем завершения всех потоков, а затем проводим очистку
    WaitForMultipleObjects(g_nNumThreads, g_hThreads, TRUE, INFINITE);
    while (g_nNumThreads--)
        CloseHandle(g.hThreads[g.nNuroThreads]);

    return(0);
}

//////////////////////////////// End of File ////////////////////////////////
```

Сводная таблица объектов, используемых для синхронизации потоков

В следующей таблице суммируются сведения о различных объектах ядра применительно к синхронизации потоков.

Табл. 9-3. Объекты ядра и синхронизация потоков

Объект	Находится в занятом состоянии, когда:	Переходит в свободное состояние, когда:	Побочный эффект успешного ожидания
Процесс	процесс еще активен	процесс завершается (*ExitProcess, TerminateProcess*)	Нет
Поток	поток еще активен	поток завершается (*ExitThread, TerminateThread*)	Нет
Задание	время, выделенное заданию, еще не истекло	время, выделенное заданию, истекло	Нет
Файл	выдан запрос на ввод-вывод	завершено выполнение запроса на ввод-вывод	Нет
Консольный ввод	ввода нет	ввод есть	Нет
Уведомление об изменении файла	в файловой системе нет изменений	файловая система обнаруживает изменения	Сбрасывается в исходное состояние
Событие с автосбросом	вызывается *ResetEvent, PulseEvent* или ожидание успешно завершилось	вызывается *SetEvent* или *PulseEvent*	Сбрасывается в исходное состояние
Событие со сбросом вручную	вызывается *ResetEvent* или *PulseEvent*	вызывается *SetEvent* или *PulseEvent*	Нет
Ожидаемый таймер с автосбросом	вызывается *CancelWaitableTimer* или ожидание успешно завершилось	наступает время срабатывания (*SetWaitableTimer*)	Сбрасывается в исходное состояние
Ожидаемый таймер со сбросом вручную	вызывается *CancelWaitableTimer*	наступает время срабатывания (*SetWaitableTimer*)	Нет
Семафор	ожидание успешно завершилось	счетчик > 0 (*ReleaseSemaphore*)	Счетчик уменьшается на 1
Мьютекс	ожидание успешно завершилось	поток освобождает мьютекс {ReleaseMutex)	Передается потоку во владение
Критическая секция (пользовательского режима)	ожидание успешно завершилось ((*Try*)*EnterCriticalSection*)	поток освобождает критическую секцию (*LeaveCriticalSection*)	Передается потоку во владение

Табл. 9-3. *(окончание)*

Объект	Находится в занятом состоянии, когда:	Переходит в свободное состояние, когда:	Побочный эффект успешного ожидания
SRWLock (пользовательского режима)	ожидание успешно завершилось (*AcquireSRWLock* (*Exclusive*))	поток освобождает SRWLock (*ReleaseSRWLock* (*Exclusive*))	Передается потоку во владение
Условная переменная (пользовательского режима)	ожидание успешно завершилось (*SleepConditionVariable*)	поток пробуждается (*Wake*(*All*)*ConditionVariable*)	Нет

Interlocked-функции (пользовательского режима) никогда не приводят к исключению потока из числа планируемых; они лишь изменяют какое-то значение и тут же возвращают управление.

Другие функции, применяемые в синхронизации потоков

При синхронизации потоков чаще всего используются функции *WaitForSingleObject* и *WaitForMultipleObjects*. Однако в Windows есть и другие, несколько отличающиеся функции, которые можно применять с той же целью. Если вы понимаете, как работают *Wait-ForSingleObject* и *WaitForMultipleObjects*, вы без труда разберетесь и в этих функциях.

Асинхронный ввод-вывод на устройствах

При асинхронном вводе-выводе поток начинает операцию чтения или записи и не ждет ее окончания. Например, если потоку нужно загрузить в память большой файл, он может «попросить» систему сделать это за него. И пока система грузит файл в память, поток спокойно занимается другими задачами — создает окна, инициализирует внутренние структуры данных и т. д. Закончив, поток приостанавливает себя и ждет уведомления от системы о том, что загрузка файла завершена.

Объекты устройств являются синхронизируемыми объектами ядра, а это означает, что вы можете вызывать *WaitForSingleObject* и передавать ей описатель какого-либо файла, сокета, коммуникационного порта и т.д. Пока система выполняет асинхронный ввод-вывод, объект устройства пребывает в занятом состоянии. Как только операция заканчивается, система переводит объект в свободное состояние, и поток узнает о завершении операции. С этого момента поток возобновляет выполнение.

Функция WaitForInputIdle

Поток может приостановить себя и вызовом *WaitForInputIdle*:

```
DWORD WaitForInputIdle(
    HANDLE hProcess,
    DWORD dwMilliseconds);
```

Эта функция ждет, пока у процесса, идентифицируемого описателем *hProcess*, не опустеет очередь ввода в потоке, создавшем первое окно приложения. *WaitForInputIdle* полезна для применения, например, в родительском процессе, который порождает дочерний для выполнения какой-либо нужной ему работы. Когда один из потоков родительского процесса вызывает *CreateProcess*, он продолжает выполнение и в то время, пока дочерний процесс инициализируется. Этому потоку может понадобиться описатель окна, создаваемого дочерним процессом. Единственная возможность узнать о моменте окончания инициализации дочернего процесса — дождаться, когда тот прекратит обработку любого ввода. Поэтому после вызова *CreateProcess* поток родительского процесса должен вызвать *WaitForInputldle*.

Эту функцию можно применить и в том случае, когда вы хотите имитировать в программе нажатие каких-либо клавиш. Допустим, вы асинхронно отправили в главное окно приложения следующие сообщения:

```
WM_KEYDOWN        с виртуальной клавишей VK_MENU
WM_KEYDOWN        с виртуальной клавишей VK_F
WM_KEYUP          с виртуальной клавишей VK_F
WM_KEYUP          с виртуальной клавишей VK_MENU
WM_KEYDOWN        с виртуальной клавишей VK_O
WM_KEYUP          с виртуальной клавишей VK_O
```

Эта последовательность дает тот же эффект, что и нажатие клавиш *Alt+F, O,* — в большинстве англоязычных приложений это вызывает команду Open из меню File. Выбор данной команды открывает диалоговое окно; но, прежде чем оно появится на экране, Windows должна загрузить шаблон диалогового окна из файла и «пройтись» по всем элементам управления в шаблоне, вызывая для каждого из них функцию *CreateWindow*. Разумеется, на это уходит какое-то время. Поэтому приложение, асинхронно отправившее сообщения типа WM_KEY*, теперь может вызвать *WaitForInputIdle* и таким образом перейти в режим ожидания до того момента, как Windows закончит создание диалогового окна и оно будет готово к приему данных от пользователя. Далее программа может передать диалоговому окну и его элементам управления сообщения о еще каких-то клавишах, что заставит диалоговое окно проделать те или иные операции.

С этой проблемой, кстати, сталкивались многие разработчики приложений для 16-разрядной Windows. Программам нужно было асинхронно передавать сообщения в окно, но получить точной информации о том, создано ли это окно и готово ли к работе, они не могли. Функция *WaitForInputIdle* решает эту проблему.

Функция MsgWaitForMultipleObjects(Ex)

При вызове *MsgWaitForMultipleObjects* или *MsgWaitForMultipleObjectsEx* поток переходит в ожидание своих (предназначенных этому потоку) сообщений:

```
DWORD MsgWaitForMultipleObjects(
    DWORD dwCount,
    PHANDLE phObjects,
    BOOL bWaitAll,
    DWORD dwMilliseconds,
    DWORD dwWakeMask);

DWORD MsgWaitForMultipleObjectsEx(
    DW0RD dwCount,
    PHANDLE phObjects,
    DWORD dwMilliseconds,
    DWORD dwWakeMask,
    DWORD dwFlags);
```

Эти функции аналогичны *WaitForMultipleObjects*. Единственное различие заключается в том, что они пробуждают поток, когда освобождается некий объект ядра или когда определенное оконное сообщение требует перенаправления в окно, созданное вызывающим потоком.

Поток, который создает окна и выполняет другие операции, относящиеся к пользовательскому интерфейсу, должен работать с функцией *MsgWaitForMultipleObjectsEx*, а не с *WaitForMultipleObjects*, так как последняя не дает возможности реагировать на действия пользователя.

Функция WaitForDebugEvent

В Windows встроены богатейшие отладочные средства. Начиная исполнение, отладчик подключает себя к отлаживаемой программе, а потом просто ждет, когда операционная система уведомит его о каком-нибудь событии отладки, связанном с этой программой. Ожидание таких событий осуществляется через вызов:

```
BOOL WaitForDebugEvent(
    PDEBUG_EVENT pde,
    DWORD dwMilliseconds);
```

Когда отладчик вызывает *WaitForDebugEvent*, его ноток приостанавливается. Система уведомит поток о событии отладки, разрешив функции *WaitForDebugEvent* вернуть управление. Структура, на которую указывает параметр pde, заполняется системой перед пробуждением потока отладчика. В ней содержится информация, касающаяся только что произошедшего события отладки. Подробнее о написании собственных отладчиков см. в статье «Escape from DLL Hell with Custom Debugging and Instrumentation Tools and Utilities, Part 2» в MSDN Magazine (http://msdn.microsoft.com/msdnmag/issues/02/08/EscapefromDLLHell/).

Функция SignalObjectAndWait

SignedObjectAndWait переводит в свободное состояние один объект ядра и ждет другой объект ядра, выполняя все это как одну операцию на уровне атомарного доступа:

```
DWORD SignalObjectAndWait(
    HANDLE hObjectToSignal,
    HANDLE hObjectToWaitOn,
    DWORD dwMilliseconds,
    BOOL bAlertable);
```

Параметр *hObjectToSignal* должен идентифицировать мьютекс, семафор или событие; объекты любого другого типа заставят *SignalObjectAndWait* вернуть WAIT_FAILED, а функцию *GetLastError* — ERROR_INVALID_HANDLE. Функция *SignalObjectAndWait* проверяет тип объекта и выполняет действия, аналогичные тем, которые предпринимают функции *ReleaseMutex*, *ReleaseSemaphore* (со счетчиком, равным 1) или *ResetEvent*.

Параметр *hObjectToWaitOn* идентифицирует любой из следующих объектов ядра: мьютекс, семафор, событие, таймер, процесс, поток, задание, уведомление об изменении файла или консольный ввод. Параметр *dwMilliseconds*, как обычно, определяет, сколько времени функция будет ждать освобождения объекта, а флаг *bAlertable* указывает, сможет ли поток в процессе ожидания обрабатывать посылаемые ему APC-вызовы.

Функция возвращает одно из следующих значений: WAIT_OBJECT_0, WAIT_TIMEOUT, WAIT_FAILED, WAIT_ABANDONED (см. раздел о мьютексах) или WAIT_IO_COMPLETION.

SignalObjectAndWait — удачное добавление к Windows API по двум причинам. Во-первых, освобождение одного объекта и ожидание другого — задача весьма распространенная, а значит, объединение двух операций в одной функции экономит процессорное время. Каждый вызов функции, заставляющей поток переходить из кода, который работает в пользовательском режиме, в код, работающий в режиме ядра, требует примерно 200 процессорных тактов, перепланировка потока также занимает немало тактов процессора (на платформах *x86*), и поэтому для выполнения, например, такого кода:

```
ReleaseMutex(hMutex);
WaitForSingleObject(hEvent, INFINITE);
```

понадобится около 400 тактов. В высокопроизводительных серверных приложениях *SignalObjectAndWait* дает заметную экономию процессорного времени. Использование функции *SignalObjectAndWait* в высокопроизводительных серверных приложениях экономит много времени.

Во-вторых, без функции *SignalObjectAndWait* ни у одного потока не было бы возможности узнать, что другой поток перешел в состояние ожидания. Знание таких вещей очень полезно для функций типа *PulseEvent*. Как я уже говорил в этой главе, *PulseEvent* переводит событие в свободное состояние

и тут же сбрасывает его. Если ни один из потоков не ждет данный объект, событие не зафиксирует этот импульс (pulse). Я встречал программистов, которые пишут вот такой код:

```
// выполняем какие-то операции … SetEvent(hEventWorkerThreadDone);
WaitForSingleObject(hEventKoreWorkToBeDone, INFINITE);
// выполняем еще какие-то операции …
```

Этот фрагмент кода выполняется рабочим потоком, который проделывает какие-то операции, а затем вызывает *SetEvent*, чтобы сообщить (другому потоку) об окончании своих операций. В то же время в другом потоке имеется код:

```
WaitForSingleObject(hEventWorkerThreadDone);
PulseEvent(hEventMoreWorkToBeDone);
```

Приведенный ранее фрагмент кода рабочего потока порочен по самой своей сути, так как будет работать ненадежно. Ведь вполне вероятно, что после того, как рабочий поток обратится к *SetEvent*, немедленно пробудится другой поток и вызовет *PulseEvent*. Проблема здесь в том, что рабочий поток уже вытеснен и пока еще не получил шанса на возврат из вызова *SetEvent*, не говоря уж о вызове *WaitForSingleObject*. В итоге рабочий поток не сможет своевременно освободить событие *hEventMoreWorkToBeDone*.

Но если вы перепишете код рабочего потока с использованием функции *SignalObjectAndWait*:

```
// выполняем какие-то операции … SignalObjectAndWait(hEventWorkerTnreadDone,
    hEventMoreWorkToBeDone, INFINITE, FALSE);
// выполняем еще какие-то операции …
```

то код будет работать надежно, поскольку освобождение и ожидание реализуются на уровне атомарного доступа. И когда пробудится другой поток, вы сможете быть абсолютно уверены, что рабочий поток ждет события *hEventMoreWorkToBeDone*, а значит, он обязательно заметит импульс, «приложенный» к событию.

Обнаружение взаимных блокировок с помощью Wait Chain Traversal API

Разработка многопоточных приложений — одна из сложнейших задач программирования, а поиск в них багов, возникающих из-за бесконечного ожидания и блокировок, особенно взаимных блокировок, еще сложнее. Функции для анализа т.н. цепочек ожидания, собранные в API Wait Chain Traversal (WCT), являются новинкой Windows Vista. Эти функции позволяют генерировать списки блокировок и выявлять взаимные блокировки потоков, принадлежащих не только к одному, но и к разным процессам. С помощью API-функций WCT Windows отслеживает синхронизирующие механизмы, а также потенциальные источники блокировок (см. табл. 9-4).

Табл. 9-4. Типы синхронизирующих механизмов, отслеживаемых WCT

Потенциальные источники блокировок	Описание
Критическая секция	Windows отслеживает критические секции и потоки, которые ими владеют
Мьютексы	Windows отслеживает мьютексы (включая те, от которых отказались владевшие им потоки) и потоки, которым они принадлежат
Процессы и потоки	Windows отслеживает потоки, ждущие завершения других потоков и процессов
Вызовы *SendMessage*	Система отслеживает потоки, ожидающие завершения вызова *SendMessage*
Вызовы и инициализация объектов COM	Вызовы *CoCreateInstance* и методов объектов COM также отслеживаются
Advanced Local Procedure Call (ALPC)	Механизма ALPC — новый незадокументированный механизм межпроцессной коммуникации, заменивший LPC в Windows Vista

Внимание! WCT не отслеживает синхронизирующие блокировки *SRWLock* (см. главу 8). Учтите также, что многие объекты ядра, включая события, семафоры и ожидаемые таймеры, также не отслеживаются, поскольку любой поток, освободив такой объект, может в произвольный момент времени пробудить другой поток, ждущий этот объект.

Программа-пример LockCop

Программа-пример LockCop (09-LockCop.exe) показывает, как с помощью WCT-функций написать весьма полезную утилиту. Файлы с исходным кодом и ресурсами этой программы см. в каталоге 09-LockCop архива, доступного на веб-сайте поддержки этой книги. Если запустить LockCop и выбрать в списке Processes отображение программ, в которых возникли взаимные блокировки, откроется окно (рис. 9-1) со списком потоков, попавших во взаимные блокировки.

Сначала LockCop перечисляет работающие в настоящее время процессы с использованием ToolHelp32 (см. главу 4), записывая идентификаторы и идентификаторы процессов в поле со списком Processes. При выборе процесса выводится список его потоков, попавших во взаимную блокировку, с указанием идентификаторов этих потоков, а также их *цепочки ожидания* (wait chain). На сайте MSDN цепочка ожидания определяется как «последовательность потоков и синхронизирующих объектов, в которой за потоком идет объект, ожидаемый этим потоком и принадлежащий следующему потоку в этой последовательности».

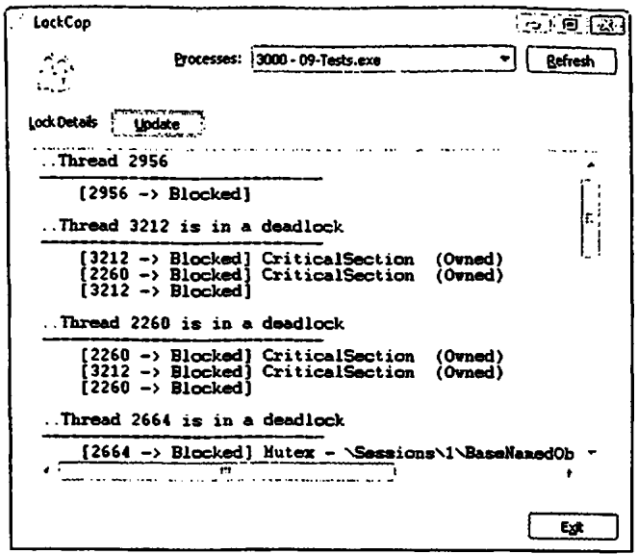


Рис. 9-1. *LockCop в действии*

Чтобы разобраться в работе цепочки ожидания, разберем ее на примере. На рис. 9-1 поток 3212 находится во взаимной блокировке, причины которой видны при анализе его цепочки ожидания:

- поток 3212 блокирован в ожидании критической секции (назовем ее CS1);
- эта критическая секция (CS1) принадлежит другому потоку (потоку с идентификатором 2260), ждущему другую критическую секцию (CS2);
- вторая критическая секция (CS2) принадлежит первому потоку с идентификатором 3212.

Подведем итоги: поток 3212 ждет, когда поток 2260 освободит нужную ему критическую секцию, в то время как поток 2260 ждет освобождения потоком 3212 другой критической секции — налицо типичная взаимная блокировка (см. стр. 238).

В сущности, отображаемые LockCop сведения генерируются различными WCT-функциями. Чтобы немного облегчить работу с ними, я создал C++-класс CWCT (см. файл WaitChainTraversal.h), упрощающий анализ цепочки ожидания. Вам нужно породить от CWCT свой класс и переопределить пару виртуальных методов (выделены на листинге жирным шрифтом). После этого можно будет во время выполнения вызвать функцию *ParseThreads* с передачей ей идентификатора нужного вам процесса:

```
class CWCT
{
    public:
        CWCT();
        ~CWCT();
```

```
        // Перебираем потоки, работающие в заданном процессе,
        // генерируя для каждого из них цепочку ожидания.
        void ParseThreads(DWORD PID);

protected:
    // Этот метод вызывается для анализа каждого потока.
    // Его следует переопределить.
    // Примечание. Если nodeCount = 0, проанализировать поток не удалось
    virtual void 0nThread(DW0RD TID, B00L bDeadlock, DWORD nodeCount);

    // Этот метод вызывается для каждого звена цепочки ожидания
    // Его следует переопределить.
    virtual void OnChainNodeInfo(DWORD rootTID, DWORD currentNode,
        WAITCHAIN_NODE_INF0 nodeInfo);

    // возвращаем число звеньев цепочки ожидания для текущего потока
    DWORD GetNodesInChain();

    //Получаем идентификатор анализируемого процесса
    DWORDGetPID();

private:
    void InitC0M();
    void ParseThread(DWORD TID);

private:
    // Описатель сеанса
    WCT HWCT _hWCTSession;

    // Описатель модуля 0LE32.DLL
    HM0DULE _hOLE32DLL;

    DWORD _PID;
    DWORD _dwNodeCount;
};
```

При создании экземпляра CWCT вызывается функция *RegisterWaitChain-COMCallback* для регистрации контекста COM у WCT (детали реализации см. в коде метода *InitCOM*), после чего открывается цепочка ожидания вызовом следующей функции:

```
HWCT OpenThreadWaitChainSession(
    DWORD dwFlags,
    PWAITCHAINCALLBACK callback);
```

Чтобы вызвать ее асинхронно, передайте в параметре *dwFlags* значение о, в противном случае передайте флаг WCT_ASYNC_OPEN_FLAG. В первом

случае также передается второй параметр с указателем на функцию обратного вызова. На сильно загруженном компьютере построение длинной цепочки ожидания может занять значительно время. В подобной ситуации имеет смысл использовать асинхронный вызов, поскольку при атом можно отменить построение цепочки вызовом *CloseThreadWaitChainSession*. Однако класс CWCT генерирует цепочку ожидания синхронно, поэтому в *dwFlags* передается 0, а второй параметр устанавливается в NULL. Деструктор CWCT закрывает сеанс WCT, передавая функции *CloseThreadWaitChainSession* описатель сеанса, полученный от *OpenThreadWaitChainSession*.

Перебор потоков заданного процесса выполняется функцией *ParseThreads*, в которой задействован ToolHelp API (см. главу 4):

```
void CWCT::ParseThreads(DWORD PID) {

    _PID = PID;

    // Создаем список всех потоков заданного процесса
    CToolhelp th(TH32CS_SNAPTHREAD, PID);
    THREADENTRY32 te = { sizeof(te) };
    BOOL fOk = th.ThreadFirst(&te);
    for (; fOk; fOk = th.ThreadNext(&te)) {
        // Анализируем только потоки заданного процесса
        if (te.th32OwnerProcessID == PID) {
            ParseThread(te.th32ThreadID);
        }
    }
}
```

Метод *ParseThread* — ключевой для анализа цепочек ожидания:

```
void CWCT::ParseThread(DWORD TID) {

    WAITCHAIN_NODE_INFO    chain[WCT_MAX.NODE_COUNT];
    DWORD                  dwNodesInChain;
    BOOL                   bOeadlock;

    dwNodesInChain = WCT_MAX_NODE_COUNT;

    // получаем цепочку ожидания для текущего потока
    if (!GetThreadWaitChain(_hWCTSession, NULL, WCTP_GETINF0_ALL_FLAGS,
        TID, &dwNodesInChain, chain, &bDeadlock)) {

        _dwNodeCount = 0;
        OnThread(TID, FALSE, 0);
        return;
    }
```

```
    // начинаем обработку цепочки ожидания для текущего потока
    _dwNodeCount = min(dwNodesInChain, WCT_MAX_NODE_COUNT);
    OnThread(TID, bDeadlock, dwNodesInChain);

    // для каждого узла цепочки вызываем виртуальный метод onChainNodeInfo

    for (
        DWORD current = 0;
        current < min(dwNodesInChain, WCT_MAX_N0DE_C0UNT);
        current++) {
        OnChainNodeInfo(TID. current, chain[current]);
    }
}
```

Функция *GetThreadWaitChain* заполняет массив структур WAITCHAIN_NODE_INFO, каждая из которых содержит описание заблокированного потока либо синхронизирующего объекта, вызвавшего блокировку:

```
B00L WINAPI GetThreadWaitChain(
    HWCT hWctSession,
    DW0RD_PTR pContext,
    DWORD dwFlags,
    DWORD TID,
    PDW0RD pNodeCount,
    PWAITCHAIN_N0DE_INF0 pNodeInfoArray,
    LPB00L pbIsCycle
);
```

Описатель, который вернула функция *OpenThreadWaitChainSession*, передается в параметре *hWctSession*. В случае асинхронного сеанса любая дополнительная информация передается через параметр *pContext*. Параметр *dwFlags* управляет работой функции при анализе потоков, принадлежащих разных процессам, и содержит один из флагов, перечисленных в таблице 9-5.

Табл. 9-5. Флаги функции GetThreadWaitChain

Значение dwFlags	Описание
WCT_OUT_OF_PROC_FLAG (0x1)	Если этот флаг не установлен, в цепочке ожидания не будет сведений о потоках и объектах из процессов, отличных от процесса, в котором работает текущий поток. Этот флаг устанавливают для приложений с несколькими процессами либо программ, порождающих процессы и ожидающих их завершения

Табл. 9-5. *(окончание)*

Значение *dwFlags*	Описание
WCT_OUT_OF_PROC_CS_FLAG (0x4)	Включает сбор сведений о критических секциях из процессов, отличных от процесса, в котором работает текущий поток. Этот флаг устанавливают для приложений с несколькими процессами либо программ, порождающих процессы и ожидающих их завершения
WCT_OUT_OF_PROC_COM_FLAG (0x2)	Важен для работы с серверами COM для многопоточного окружения
WCTP_GETINFO_ALL_FLAGS	Эквивалент одновременной установки все вышеперечисленных флагов

Параметр TID содержит идентификатор потока, с которого вы хотите начать цепочку ожидания. Описание цепочки ожидания возвращается в последних трех параметрах:

- DWORD, на которое ссылается параметр *pNodeCount*, содержит число звеньев цепочки;
- сами звенья хранятся в массиве, который передается через параметр *pNodeInfoArray*;
- при обнаружении взаимной блокировки булева переменная, на которую ссылается параметр *pbIsCycle*, устанавливается в TRUE.

При обработке функцией *ParseThread* каждого потока заданного процесса, вызывается переопределенный вами метод *OnThread* с передачей идентификатора потока в первом параметре; при обнаружении взаимной блокировки *bDeadLock* устанавливается в TRUE, а *nodeCount* содержит число звеньев цепочки ожидания этого потока (значение 0 присваивается этому параметру при сбоях, например, из-за отказа в доступе). Переопределенный вами метод *OnChainNodeInfo* вызывается для обработки каждого из звеньев цепочки ожидания, при этом в параметре *rootTID* передается идентификатор потока, переданный *OnThread*; в *currentNode* передается номер (начиная с 0) текущего звена, а параметре *nodeInfo* — описание этого звена в виде структуры WAITCHAW_NODE_INFO, объявленной в заголовочном файле wct.h.

```
typedef struct _WATTCHAIN_N0DE_INF0
{
    WCT_OBJECT_TYPE ObjectType;
    WCT_OBJECT_STATUS ObjectStatus;

    union {
        struct {
            WCHAR ObjectName[WCT_OBJNAME_LENGTH];
            LARGE_INTEGER Timeout;           // в этой версии не реализовано
```

```
            BOOL Alertable;                    // в этой версии не реализовано
        } LockObject;

        struct {
            DW0R0 ProcessId;
            DWORD ThreadId;
            DWORD WaitTime;
            DWORD ContextSwitches;
        } ThreadObject;
    };

} WAITCHAIN_NODE_INFO, *PWAITCHAIOODE_INFO;
```

Тип звена определяется полем *ObjectType*, принимающим значения перечислимого WCT_OBJECT_TYPE (см. табл. 9-6).

Табл. 9-6. Типы объектов, представляющих звенья цепи ожидания

WCT_OBJECT_TYPE	Описание
WctThreadType	Заблокированный поток
WctCriticalSectionType	Принадлежащий потоку объект — критическая секция
WctSendMessageType	Причина блокирования — вызов *SendMessage*
WctMutexType	Принадлежащий потоку объект — мьютекс
WctAlpcType	Причина блокирования — ALPC-вызов
WctComType	Ожидание завершения вызова COM
WctThreadWaitType	Ожидание завершения потока
WctProcessWaitType	Ожидание завершения процесса
WctComActivationType	Ожидание завершения вызова CoCreateInstance
WctUnknownType	Зарезервирован для будущих расширений API

Группа *ThreadObject* в составе этой структуры имеет смысл, только если *ObjectType* = *WctThreadType*, во всех остальных случае используется группа *LockObject*. Цепочка ожидания потока всегда начинается с звена типа *WctThreadType*, соответствующего значению параметра *rootTID*, переданному при вызове *OnChainNodeInfo*.

Поле *ObjectStatus* содержит сведения о состоянии потока, если *ObjectType* = *WctThreadType*. В противном случае это поле содержит информацию о состоянии блокировки — звене цепочки ожидания. Описание состояния определено как перечислимое WCT_OBJECT_STATUS:

```
typedef enum _WCT_OBJECT_STATUS
{
    WctStatusNoAccess = 1,                 // Отказано в доступе к объекту
    WctStatusRunning,                      // Состояние потока
```

```
   WctStatusBlocked,                   // Состояние потока
   WctStatusPidOnly,                   // Состояние потока
   WctStatusPidOnlyRpcss,              // Состояние потока
   WctStatusOwned,                     // Состояние синхронизирующего объекта
   WctStatusNotOwned,                  // Состояние синхронизирующего объекта
   WctStatusAbandoned,                 // Состояние синхронизирующего объекта
   WctStatusUnknown,                   // Для всех объектов
   WctStatusError,                     // Для всех объектов
   WctStatusMax
} WCT_OBJECT_STATUS;
```

Чтобы проиллюстрировать работу утилиты LockCop, я прилагаю к ней программу 09-BadLock, которая генерирует массу взаимных блокировок и бесконечно ожидающих потоков. Она поможет понять, как WCT заполняет структуру WAITCHAIN_NODE_INFO для разных типов блокировок.

> **Примечание**. Утилита LockCop удобна для диагностики взаимных блокировок и ситуаций с бесконечно ожидающими потоками. Однако Windows Vista налагает на LockCop одно ограничение: в этой операционной системе не поддерживается функция *WaitForMultipleObjects*. Если ваш код вызывает эту функцию, чтобы заставить поток ждать сразу несколько объектов, LockCop обнаружит бесконечные циклы, но, вызывая *OnThread* по завершении метода *GetThreadWaitChain*, он не «увидит» явные блокировки.

Оглавление

ГЛАВА 10

Синхронный и асинхронный ввод-вывод на устройствах

Сложно переоценить важность этой главы, посвященной технологиям Windows, позволяющим создавать быстрые, масштабируемые и надежные приложения. Масштабируемым считается приложение, способное выполнить множество одновременных операций не менее эффективно, чем обработку незначительной вычислительной нагрузки. В случае службы Windows такими операциями являются клиентские запросы, время поступления которых предсказать невозможно, как и количество вычислительных ресурсов, необходимое для обработки запросов. Как правило, эти запросы приходят с устройств ввода-вывода, таких как сетевые платы, а в обработке этих запросов часто участвуют другие устройства, такие как диски.

В Windows-приложениях удобнее всего распределять нагрузку между потоками. Каждый поток приписан к определенному процессору, что позволяет исполнять на многопроцессорном компьютере сразу несколько операций, что повышает общую производительность. Сгенерировав синхронный запрос ввода-вывода на устройстве, поток приостанавливается до завершения обработки этого запроса. При этом страдает производительность, поскольку в данном состоянии поток неспособен выполнять полезную работу, такую как обработка клиентских запросов. Короче говоря, ваша задача — заставить потоки выполнять как можно больше полезных операций, избегая их блокировки.

Чтобы потоки не простаивали, им необходимо взаимодействовать, обмениваться информацией о выполняемых ими операциях. Майкрософт затратила годы работы на исследования и разработки в этой области. Результат этих усилий — весьма совершенный механизм взаимодействия потоков, получивший название *порт завершения ввода-вывода* (I/O completion port). С помощью данного механизма разработчики могут создавать высокопроизводительные масштабируемые приложения. Использование портов завер-

шения ввода-вывода позволяет добиться феноменальной производительности приложений, поскольку оно освобождает их от необходимости ждать отклика устройств ввода-вывода.

Изначально порты завершения ввода предназначены для обработки ввода-вывода на устройствах, но со временем в операционных системах Майкрософт появлялось все больше и больше механизмов, использующих модель портов завершения ввода-вывода. Примером может быть объект ядра «задание», отслеживающий включенные в задание процессы и отправляющий уведомления о событиях в порты завершения ввода-вывода. Совместную работу объектов-заданий и портов завершения ввода-вывода иллюстрирует программа-пример Job Lab из главы 5.

Несмотря на свой многолетний опыт разработки для Windows, я постоянно нахожу новые применения для портов завершения ввода-вывода и считаю, что каждый Windows-разработчик должен досконально разбираться в работе этого механизма. В этой главе я демонстрирую работу портов завершения ввода-вывода при работе с устройствами, но этим их сфера применения далеко не исчерпывается. Проще говоря, порты завершения ввода-вывода — это отличное средства для организации взаимодействия между потоками с безграничной областью применения.

После такой патетики легко заключить, что я — большой фанат использования портов завершения ввода-вывода. Надеюсь, дочитав эту главу до конца, вы тоже поклонником этого средства. Однако я немного отложу детальный разбор портов завершения ввода-вывода, чтобы рассказать о том, какие средства ввода-вывода на устройствах в Windows были исходно доступны разработчикам, чтобы вы глубже осознали все достоинства этого механизма.

Открытие и закрытие устройств

Одна из сильных сторон Windows состоит в разнообразии устройств, поддерживаемых этой операционной системой. Устройством в контексте этого раздела мы будем называть любую сущность, взаимодействующую с системой. Типичные устройства с указанием их применения перечислены в таблице 10-1.

Табл. 10-1. Типичные устройства и их применение

Устройство	Применение
Файл	Постоянное хранилище для любых данных
Каталог	Назначение атрибутов и параметров сжатия файлов
Логический диск	Форматирование дисков
Физический диск	Доступ к таблице разделов
Последовательный порт	Передача данных по телефонным линиям
Параллельный порт	Передача данных на принтер
Почтовый ящик	Передача данных множеству адресатов, обычно Windows-компьютерам по сети

Табл. 10-1. *(окончание)*

Устройство	Применение
Именованный канал	Передача данных отдельному адресату, обычно Windows-компьютеру по сети
Неименованный канал	Обмен данными между парой адресатов на одном и том же компьютере (но ни в коем случае не по сети)
Сокет	Передача данных (потоковая или в виде дейтаграмм, обычно через сеть) на другие поддерживающие сокеты компьютеры (под управлением Windows или других операционных систем), обычно осуществляется через сеть
Консоль	Экранный буфер для текстового окна

В этой главе пойдет речь о том, как наладить взаимодействие потоков с перечисленными выше устройствами так, чтобы потокам не пришлось дожидаться их отклика. Windows пытается по максимуму скрыть от разработчиков различия между устройствами. Другими словами, открыв любое устройство, вы сможете использовать для чтения и записи данных одни и те же Windows-функции. Хотя некоторые функции чтения-записи работают на всех устройствах, устройства все различаются. Так, скорость передачи данных через последовательный порт задают в бод, но этот параметр не имеет никакого смысла для именованных каналов, применяемых для взаимодействия компьютеров сеть (либо компонентов локальной системы) настройка скорости передачи в бод. Я не буду обсуждать все тонкости, которыми устройства отличаются друг от друга, но поподробнее остановлюсь на файлах, поскольку с ними приходится работать очень часто. Для выполнения любых операций ввода-вывода прежде всего необходимо открыть устройство и получить его описатель. Способ получения описателя зависит от типа устройства. Функции, позволяющие открывать различные устройства, перечислены в таблице 10-2.

Табл. 10-2. Функции, открывающие различные устройства

Устройство	Функция
Файл	*CreateFile* (значение *pszName* — путь или UNC-путь)
Каталог	*CreateFile* (значение *pszName* — путь или UNC-путь к каталогу). Windows позволяет открывать каталоги, если при вызове *CreateFile* установлен флаг FILE_FLAG_BACKUP_SEMANTICS. Открыв каталог, можно изменять его атрибуты (например, сделать его скрытым) и временную отметку
Логический диск	*CreateFile* (значение pszName — "\\.\x"). В Windows можно открывать логические диски, ссылаясь на них в формате "\\.\x", где x — буква диска. Например, открыть диск A можно с помощью ссылки \\.\A: Открыв диск, вы сможете его отформатировать либо определить размер носителя

Табл. 10-2. *(окончание)*

Устройство	Функция
Физический диск	*CreateFile* (значение *pszName* — "\\.\PHYSICALDRIVEx"). В Windows можно открывать физические диски, используя ссылку в формате "\\.\PHYSICALDRIVEx", где x — номер физического диска. Например, прочитать физические сектора первого жесткого диска на компьютере пользователя позволит ссылка "\\.\PHYSICALDRIVE0". Открыв физический диск, вы сможете напрямую обращаться к его таблице разделов, правда, это опасно: некорректная модификация содержимого диска сделает его недоступным для операционной системы
Последовательный порт	*CreateFile* (значение *pszName* — "COMx")
Параллельный порт	*CreateFile* (значение *pszName* — "LPTx")
Сервер почтового ящика	*CreateMailslot* (значение pszName — "\\.\mailslot\имя_ящика")
Клиент почтового ящика	*CreateFile* (значение *pszName* — "\\имя_сервера\mailslot\имя_ящика")
Сервер именованного канала	*CreateNamedPipe* (значение *pszName* — "\\.\pipe\имя_канала")
Клиент именованного канала	*CreateFile* (значение *pszName* — "\\имя_сервера\гяре\имя_ канала")
Неименованный канал	*CreatePipe* (для клиента и сервера)
Сокет	*socket, accept* и *AcceptEx*
Консоль	*CreateConsoleScreenBuffer* и *GetStdHandle*

Все эти функции возвращают описатели, идентифицирующие устройства. Этот описатель передают различным функциям для взаимодействия с устройствами. Например, функция *SetCommConfig* устанавливается скорость последовательного порта (в бод):

```
BOOL SetCommConfig(
    HANDLE       hCommDev,
    LPCOMMCONFIG pCC,
    DWORD        dwSize);
```

а функция *SetMailslotInfo* — длительность ожидания чтения данных:

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,
    DWORD  dwReadTimeout);
```

Как видно, первым аргументом этих функций является описатель устройства. Закончив работу с устройством, описатель необходимо закрыть.

Для большинства устройств это можно сделать вызовом весьма популярной функции *CloseHandle*:

```
BOOL CloseHandle(HANDLE hobject);
```

Однако в случае сокета для этого необходимо вызывать функцию *closesocket*:

```
int closesocket(SOCKET s);
```

Кроме того, обладая описателем, можно узнать тип соответствующего устройства, вызвав *GetFileType*:

```
DWORD GetFileType(HANDLE hDevice);
```

Для этого достаточно передать функции *GetFileType* описатель устройства, и она вернет одно из значений, перечисленных в таблице 10-3.

Табл. 10-3. Значения, возвращаемые функцией GetFileType

Значение	Описание
FILE_TYPE_UNKNOWN	Тип заданного файла неизвестен
FILE_TYPE_DISK	Дисковый файл
FILE_TYPE_CHAR	Текстовый файл, консоль или устройство, подключенное к порту LPT
FILE_TYPE_PIPE	Именованный или неименованный канал

Близкое знакомство с функцией *CreateFile*

Естественно, функция *CreateFile* способна создавать и открывать файлы, но этим ее возможности далеко не исчерпываются — она способна открывать и множество других устройств:

```
HANDLE CreateFile(
    PCTSTR pszName,
    DWORD  dwDesiredAccess,
    DWORD  dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD  dwCreationDisposition,
    DWORD  dwFlagsAndAttributes,
    HANDLE hFileTemplate);
```

Как видно из этого листинга, *CreateFile* принимает изрядное число параметров, что обеспечивает ей немалую гибкость при работе с устройствами. Давайте разберем эти параметры подробно.

При вызове *CreateFile* параметр *pszName* определяет тип и экземпляр устройства. Параметр *dwDesiredAccess* указывает, как должен осуществляться обмен данными с устройством, его значения перечислены в табл. 10-4. Некоторые устройства поддерживают дополнительные флаги, управля-

ющие доступом. Подробнее о каждом из этих флагов см. в документации Platform SDK.

Табл. 10-4. Значения параметра dwDesiredAccess функции *CreateFile*

Значение	Описание
0	Это значение передается, если чтение-запись данных на устройстве не планируется, а требуется только изменить его конфигурацию, например временную отметку файла
GENERIC_READ	Разрешает доступ к устройству только для чтения
GENERIC_WRITE	Разрешает доступ к устройству только для записи. Данное значение используется для работы с принтером либо архивными носителями. Заметьте, что установка GENERIC_WRITE не предполагает автоматическую установку флага GENERIC_READ
GENERIC_READ | GENERIC_WRITE	Разрешает доступ к устройству для чтения и записи. Это значение используется чаще всего, поскольку оно обеспечивает свободный обмен данными

Параметр *dwShareMode* определяет совместным доступом к устройству. Он определяет, как устройство будет открываться другими вызовами *CreateFile*, выполненными, пока вы еще не закрыли это устройство (вызовом *CloseHandle*). Возможные значения параметра *dwShareMode* перечислены в табл. 10-5.

Табл. 10-5. Значения параметра dwShareMode

Значение	Описание
0	Запрашивается монопольный доступ к устройству. Если это устройство уже открыто, вызов *CreateFile* заканчивается неудачей. Если же открыть устройство удастся, неудачей закончатся все последующие вызовы *CreateFile*
FILE_SHARE_READ	Этот флаг запрещает модификацию данных на этом устройстве всем другим объектам ядра. Если устройство уже открыто для записи или для монопольного доступа, вызов *CreateFile* заканчивается неудачей. Если же открыть устройство удастся, неудачей закончатся все последующие вызовы *CreateFile* с флагом GENERIC_WRITE
FILE_SHARE_WRITE	Этот флаг запрещает чтение данных на этом устройстве всем другим объектам ядра. Если устройство уже открыто для чтения или для монопольного доступа, вызов *CreateFile* заканчивается неудачей. Если же открыть устройство удастся, неудачей закончатся все последующие вызовы *CreateFile* с флагом GENERIC_READ

Табл. 10-5. *(продолжение)*

Значение	Описание
FILE_SHARE_READ | FILE_SHARE_ WRITE	Всем объектам ядра разрешен доступ для чтения и записи к этому устройству. Если устройство уже открыто для чтения или для монопольного доступа, вызов *CreateFile* заканчивается неудачей. Если же удалить устройство удастся, неудачей закончатся все последующие вызовы *CreateFile* с запросом монопольного доступа для чтения, записи либо чтения и записи закончатся неудачей
FILE_SHARE_ DELETE	Этот флаг разрешает логическое удаление или перемещение файла, даже если он еще не закрыт. В действительности при этом Windows помечает данный файл для удаления, но удаляет его только когда все описатели этого файла будут закрыты

Примечание. Открывая файл, вы можете передать путь **к** нему, максимальная длина которого определяется значением MAX_PATH (не более 260 символов, согласно определению в файле WinDef.h). Однако, это ограничение можно обойти, вызывая *CreateFileW* (Unicode-версию функции *CreateFile***)** и предваряя путь строкой “*\\?\”. При вызове *CreateFileW* префикс удаляется, и максимальная длина пути составляет около 32 OOO Unicode-символов. Однако помните, что с данным префиксом разрешено использовать только полные пути, относительные ссылки, включая «.» и «..», не поддерживаются. При этом длина отдельных компонентов пути по-прежнему ограничена значением MAX_PATH. Не стоит также удивляться, обнаружив в различных исходных кодах константу _MAX_PATH, значение которой в стандартных библиотеках C/C++ определено как 260 (см. stdlib.h).

Параметр *psa* ссылается на структуру SECURITY_ATTRIBUTES, которая позволяет задать атрибуты системы безопасности, а также указать, должен ли быть наследуемым описатель, возвращаемый функцией *CreateFile*. Дескриптор защиты, который содержится в этой структуре, используется только при создании файлов в защищенных файловых системах, таких как NTFS, и игнорируется в остальных случаях. Обычно в параметре *psa* передают просто NULL. При этом создается файл с параметрами защиты по умолчанию, а функция возвращает описатель, не являющийся наследуемым.

Параметр *dwCreationDisposition* особо важен при вызове *CreateFile* для открытия именно файлов, а не других устройств. Возможные значения этого параметра приводятся в табл. 10-6.

Табл. 10-6. Значения параметра *dwCreationDisposition* функции *CreateFile*

Значение	Описание
CREATE_NEW	Вызов *CreateFile* создает новый файл либо заканчивается неудачей, если файл с таким же именем уже существует
CREATE_ALWAYS	Вызов *CreateFile* создает новый файл независимо от того, существует ли уже файл с таким же именем, и перезаписывает существующий файл
OPEN_EXISTING	Вызов *CreateFile* открывает существующий файл либо заканчивается неудачей, если, файл или устройство с таким же именем не существует
OPEN_ALWAYS	Вызов *CreateFile* открывает файл, если он существует, либо создает новый файл в противном случае
TRUNCATE_EXISTING	Вызов *CreateFile* открывает существующий файл, усекая его до 0 байтов, либо заканчивается неудачей, если заданный файл не существует

> **Примечание.** Вызывая *CreateFile*, чтобы открыть любое устройство кроме файла, в параметре *dwCreationDisposition* следует передавать значение OPEN_EXISTING.

Параметр *dwFlagsAndAttributes* функции *CreateFile* используется для установки флагов, оптимизирующих взаимодействие с устройством, а также для установки файловых атрибутов (если устройство является файлом). Большинство флагов-параметров этой функции сообщают системе, как вы планируете обращаться к устройству. Это позволяет оптимизировать алгоритм кэширования и повысить эффективность работы приложения. Начнем с описания этих флагов, а затем перейдем к атрибутам.

Флаги функции *CreateFile*, управляющие кэшированием

В этом разделе описаны различные флаги *CreateFile*, управляющие кэшированием, при этом особое внимание уделяется объектам файловой системы. Подробную информацию о других объектах ядра, таких как почтовые ящики, см. в документации MSDN.

Флаг FILE_FLAG_NO_BUFFERING

Этот флаг запрещает использование буферизации при обращении к файлу. Для повышения быстродействия система кэширует данные при чтении и записи их на диск, и этот флаг обычно не указывают, чтобы позволить диспетчеру кэша хранить в памяти недавно затребованные данные файловой системы. Так что если, прочитав из файла несколько байтов, вы попробуете прочитать из этого файла еще несколько байтов, скорее всего окажется, что система уже загрузила в оперативную память эти данные. В итоге удастся обойтись вместо двух обращений к диску одним, что сильно повышает быс-

тродействие. Однако в действительности при этом в памяти оказывается две копии содержимого файла: в буфере диспетчера кэша и в вашем собственном буфере, куда эти данные копируются вызванной вами функцией (такой как *ReadFile*).

При буферизации данных диспетчер кэша может использовать упреждающее чтение, так что к тому моменту, когда программа обратится к следующей порции данных (при последовательном чтении файла), эти данные, скорее всего, будут уже в памяти. Скорость работы повышается и за счет чтения из файла большего количества данных, чем реально требуется программе. Но если программе не потребуются следующие порции данных из этого файла, часть памяти будет потрачена зря. (Подробнее об упреждающем чтении см. в описании флагов ниже FILE_FLAG_SEQUENTIAL_SCAN и FILE_FLAG_RANDOM_ACCESS.)

Установив флаг FILE_FLAG_NO_BUFFERING, вы запретите диспетчеру кэша буферизацию данных, но знайте, что при этом ответственность за кэширование данных ложится на вас! В зависимости от задачи, этот флаг может повышать быстродействие и эффективность использования памяти. Поскольку драйвер устройства файловой системы пишет содержимое файла прямо в предоставленные вами буферы, придерживайтесь следующих правил:

- в работе с файлом всегда следует использовать только смещения, кратные размеру сектора дискового тома, на котором находится файл (определить его можно с помощью функции *GetDiskFreeSpace*);
- читать и записывать данные всегда следует порциями, размер которых в байтах кратен размеру сектора дискового тома;
- адрес начала буфера, выделенного в адресном пространстве процесса, должен быть кратным размеру диска.

Флаги FILE_FLAG_SEQUENTIAL_SCAN и FILE_FLAG_RANDOM_ACCESS

Эти флаги полезны, только если вы разрешаете системе управлять кэшированием содержимого файла за вас. Если установлен флаг FILE_FLAG_NO_BUFFERING, оба этих флага игнорируются.

Установив флаг FILE_FLAG_SEQUENTIAL_SCAN, вы сообщите системе, что собираетесь читать файл последовательно. В результате система будет читать содержимое файла в память до того, как вы реально обратитесь к нему, что уменьшает число обращений к жесткому диску и повышает быстродействие вашего приложения. Если вам придется обращаться к нужным участкам файла напрямую, система потратит зря небольшую часть процессорного времени и памяти, кэшируя данные, которые вам так и не потребуются. Это вполне нормально, но если такая ситуация повторяется часто, лучше указать флаг FILE_FLAG_RANDOM_ACCESS, запрещающий упреждающее чтение.

Для управления файлом диспетчеру кэша требуется ряд внутренних структур данных, и чем больше файл, тем больше таких структуру нужно

диспетчеру кэша. При работе с чрезвычайно большими файлами диспетчеру кэша может не хватить памяти для необходимых структур, в результате ему не удастся открыть такой файл. Так что при работе с гигантскими файлами устанавливайте флаг FILE_FLAG_NO_BUFFERING.

Флаг FILE_FLAG_WR1TE_THR0UGH

Последний флаг, управляющий кэшем, отключает буферизацию данных, предназначенных для записи в файл, снижая тем самым риск потери данных. Если установлен этот флаг, все модифицированные данные система сразу же записывает в файл. Тем не менее, система поддерживает внутренний кэш содержимого файла, по возможности читая его из кэша, а не напрямую с диска. Когда этот флаг используют, чтобы открыть файл с сетевого диска, Windows-функции для записи в файлы не возвращают управление вызывающему потоку, пока данные не будут записаны в файл на сетевом диске. Вот, собственно, и все о флагах, управляющих кэшированием.

Другие флаги функции *CreateFile*

В этом разделе мы поговорим о других флагах *CreateFile*, не связанных с кэшированием.

Флаг FILE_FLAG_DELETE_ON_CLOSE

Этот флаг заставляет систему удалить файл после того, как будет закрыт последний его описатель. Чаще всего данный флаг используют с флагом-атрибутом FILE_ATTRIBUTE_TEMPORARY. Одновременное использование этих двух флагов позволяет в приложении создавать временные файлы, записывать и читать в них, а затем закрывать временные файлы. Как только временный файл будет закрыт, система автоматически удалит его — как удобно!

Флаг FILE_FLAG_BACKUP_SEMANTICS

Этот флаг используют в программах для архивации и восстановления данных. Прежде чем открыть или создать файл, система обычно проверяет наличие у процесса, пытающегося выполнить это действие, достаточного уровня привилегий. Особенностью программ архивации является их способность обходить проверку прав доступа в отношении некоторых файлов. Если установлен флаг FILE_FLAG_BACKUP_SEMANTICS, система проверяет наличие у маркера доступа вызывающего потока привилегии Backup/Restore File and Directories. Если она имеется; система разрешает открыть файл. Флаг FILE_FLAG_BACKUP_SEMANTICS также используют, чтобы открывать описатели каталогов.

Флаг FILE_FLAG_POSIX_SEMANTICS

В Windows регистр символов в именах файлов сохраняется, но не учитывается при поиске файлов по имени. Однако подсистема POSIX требует

учитывать регистр символов при поиске файлов по имени. Флаг FILE_PLAG_POSIX_SEMANTICS заставляет *CreateFile* учитывать регистр при поиске во время создания или открытия файла. Флаг FILE_FLAG_POSIX_SEMANTICS следует применять чрезвычайно осторожно, поскольку файлы, созданные с таким флагом, могут оказаться недоступными Windows-приложениям.

Флаг FILE_FLAG_OPEN_REPARSE_POINT

По этому, этому флагу больше подходит имя FILE_FLAG_IGNORE_REPARSE_POINT, поскольку он заставляет систему игнорировать атрибут повторного разбора, если таковой есть у файла. Такие атрибуты позволяют фильтрам файловой системы по-другому открывать, читать, записывать и закрывать файлы. Как правило это делается с определенной целью, поэтому использовать флаг FILE_FLAG_OPEN_REPARSE_POINT не рекомендуется.

Флаг FILE_FLAG_OPEN_NO_RECALL

Этот флаг запрещает системе восстанавливать содержимое файла с архивного носителя (такого как картридж с магнитной пленкой) на подключенное к системе хранилище (например, на жесткий диск). Файлы, к которым долгое время никто не обращается, система может перенести на архивные носители, чтобы освободить место на жестком диске. При этом с жесткого диска удаляется только содержимое файла, а сам файл остается на диске. Когда кто-то пытается открыть этот файл, система автоматически восстанавливает его содержимое **с** архивного носителя. Флаг FILE_FLAG_OPEN_NO_RECALL запрещает системе автоматически выполнять такие операции.

FILE_FLAG_OVERLAPPED

Установив этот флаг, вы сообщаете системе, что хотите работать с устройством асинхронно. По умолчанию устройства открываются для синхронного ввода-вывода (без флага FILE_FLAG_OVERLAPPED). Большинство разработчиков привыкло именно к синхронному вводу-выводу, когда поток приостанавливается до завершения чтения данных из файла. После чтения необходимых данных поток вновь получает управления и его исполнение продолжается.

Поскольку операции ввода-вывода на устройства являются довольно медленным по сравнению с другими операциями, стоит подумать об использовании асинхронного ввода-вывода на отдельных устройствах. Вот как он работает: вы вызываете функцию, запрашивающую у системы чтение или запись данных, но вместо того, чтобы ждать завершения этой операции, вызванная функция немедленно возвращает управление, а операционная система сама завершает операцию ввода-вывода, используя собственные потоки. Закончив запрошенную операцию, система уведомляет об этом ваше приложение. Асинхронный ввод-вывод — ключ к созданию производительных, масштабируемых и надежных приложений, быстро реагирующих на

действия пользователя. Windows поддерживает несколько методов асинхронного ввода-вывода, о которых будет рассказано в этой главе.

Флаги файловых атрибутов

А теперь настало время вернуться к флагам атрибутов, которые передаются через параметр *dwFlagsAndAttributes* функции *CreateFile* (см. табл. 10-7). Эти флаги игнорируются во всех случаях кроме одного, когда вы создаете новый файл и передаете в параметре *hFileTemplate* NULL-значение. Многие из перечисленных атрибутов должны быть вам уже знакомы.

Табл. 10-7. Флаги файловых атрибутов (значения параметра *dwFlagsAndAttributes*)

Флаг	Описание
FILE_ATTRIBUTE_ ARCHIVE	Файл является архивным. Приложения помечают этим флагом файлы, подлежащие удалению. Для новых файлов функция CreateFile устанавливает этот флаг автоматически
FILE_ATTRIBUTE_ ENCRYPTED	Файл зашифрован
FILE_ATTRIBUTE_ HIDDEN	Файл является скрытым и не отображается при просмотре каталога обычными средствами
FILE_ATTRIBUTE_ NORMAL	Другие атрибуты не установлены. Имеет смысл, только если является единственным атрибутом файла
FILE_ATTRIBUTE_ NOT_CONTENT_ INDEXED	Файл обрабатывается службой индексирования
FILE_ATTRIBUTE_ OFFLINE	Файл существует, но его содержимое перемещено на архивный носитель. Этот флаг применяется в иерархических системах хранения данных
FILE_ATTRIBUTE_ READONLY	Файл доступен только для чтения, приложениям разрешено его чтение, но запрещены запись и удаление этого файла
FILE_ATTRIBUTE_ SYSTEM	Файл является часть операционной системы либо используется только ей
FILE_ATTRIBUTE_ TEMPORARY	Временный файл. Операционная система пытается хранить его содержимое в оперативной памяти, чтобы свести к минимуму время доступа

Атрибут FILE_ATTRIBUTE_TEMPORARY служит для создания временных файлов. Система стремится хранить файл с таким атрибутом в оперативной памяти, а не на диске, что существенно ускоряет доступ к нему. Если вы запишете в этот файл много данных, и система не сможет хранить его в RAM, ей придется сбросить часть содержимого этого файла на жесткий диск. Комбинируя флаги FILE_ATTRIBUTE_TEMPORARY и FILE_ FLAG_DELETE_ON_CLOSE (о нем см. выше) можно повысить быстродействие системы. Как правило, после закрытия файла система сбрасывает

его кэшированное содержимое на диск. Если же атрибуты файла говорят системе, что данный файл следует удалить сразу после закрытия, ей не приходится сбрасывать на диск его содержимое.

В дополнение к вышеописанным флагам система поддерживает ряд флагов, управляющих защитой и работой именованных каналов. Подробнее о них см. в описании функции CreateFile в документации Platform SDK.

Последний параметр *CreateFile*, *hFileTemplate*, содержит описатель открытого файла либо NULL. Если в *hFileTemplate* находится описатель файла, *CreateFile* игнорирует любые атрибуты, заданные флагами в *dwFlagsAndAttributes*, используя вместо них атрибуты файла, заданного параметром *hFileTemplate*. Чтобы все это работало, заданный параметром *hFileTemplate* файл должен быть открыт с флагом GENERIC_READ. Если функция *CreateFile* открывает существующий файл (а не создает новый), параметр *hFileTemplate* игнорируется.

Успешно создав или открыв файл либо устройство, функция *CreateFile* возвращает описатель файла или устройства. Если же вызов *CreateFile* заканчивается неудачей, возвращается INVALID_HANDLE_VALUE.

> **Примечание.** Большинство Windows-функций, возвращающих описатели; при неудачном вызове возвращают NULL. В отличие от них, *CreateFile* возвращает в этом случае значение INVALID_HANDLE_VALUE (определенное как -1). Мне часто приходилось видеть такой ошибочный код:
>
> ```
> HANDLE hFile = CreateFile(…);
> if (hFile == NULL) {
>    // программа никогда не окажется здесь
> } else {
>    // этот блок будет исполнен, даже если файл не создан
> }
> ```
>
> Вот как надо правильно обращаться с значением, возвращаемым функцией *CreateFile*:
>
> ```
> HANDLE hFile = CreateFile(…);
> if (hFile == INVALID_HANDLE_VALUE) {
>    // файл не создан
> } else {
>    // файл успешно создан
> }
> ```

Работа с файлами

Поскольку работать с файлами приходится очень часто, я решил подробнее остановиться на проблемах, характерных для файлов, которые в контексте этой главы тоже считаются устройствами. Ниже я расскажу, как устанавливают указатель в файле и изменяют размер файла.

Прежде всего, вам следует знать, что Windows поддерживает работу с чрезвычайно большими файлами. Разработчики в Майкрософт использовали для представления размера файлов 64-разрядные значения вместо 32-разрядных. Это означает, что теоретический максимальный размер файла в Windows составляет 16 экзабайт.

Обработка 64-разрядных значений в 32-разрядной операционной системе несколько затруднительна, поскольку многие Windows-функции требуют передачи 64-разрядных значений как пары 32-разрядных. Однако вы убедитесь, что это не трудно, как кажется. Кроме того, вам редко придется сталкиваться с обработкой файлов, размер которых превышает 4 Гб. Это означает, что 32 старших бита 64-разрядных значений чаще всего будут нулевыми.

Определение размера файла

При работе с файлами довольно часто приходится определять их размеры. Проще всего сделать это вызовом *GetFileSizeEr*.

```
BOOL GetFileSizeEx(
    HANDLE         hFile,
    PLARGE_INTEGER pliFileSize);
```

Первый параметр *hFile* — описатель открытого файла, а *pliFileSize* — адрес объединенной структуры LARGEJNTEGER. Эта структура позволяет ссылаться на 64-разрядное значение с знаком как на пару 32-разрядных либо одно 64-разрядное, что весьма удобно при работе с размерами файлов и смещениями. Вот как выглядит эта объединенная структура:

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;                  // Младшее 82-разрядное значение без знака
        LONG HighPart;                  // Старшее 32-разрядное значение со знаком
    };
    L0NGL0NG QuadPart;                  // Полное 64-разрядное значение со знаком
} LARGE_INTEGER, *PLARGE_INTEGER;
```

В дополнение к LARGE_INTEGER поддерживается структура ULARGE_INTEGER, представляющая 64-разрядное значение без знака:

```
typedef union _ULARGE_INTEGER {
    struct {
        DWORD LowPart;       // Младшее 32-разрядное значение без знака
        DWORD HighPart;      // Старшее 32-разрядное значение без знака
    };
    ULONGLONG QuadPart;      // Полное 64-разрядное значение без знака
} ULARGE_INTEGER, *PULARGE_INTEGER;
```

Есть еще одна весьма полезная функция для определения размера файла — *GetCompressedFileSize*:

```
DWORD GetCompressedFileSize(
    PCTSTR pszFileName,
    PDWORD pdwFileSizeHigh);
```

Эта функция возвращает физический размер файла, а *GetFileSizeEx* — логический размер. Возьмем для примера 100-Кб файл, сжатый до размера 85 Кб. Вызов *GetFileSizeEx* вернет логический размер файла (100 Кб), тогда как *GetCompressedFileSize* вернет реальное число байтов, занятое на диске этим файлом (85 Кб).

В отличие от ***GetFileSizeEx,*** первый параметр функции *GetCompressedFileSize* — это имя файла в виде строки, а не описателя. *GetCompressedFileSize* возвращает 64-разрядное значение, представляющее размер файла, но делает это необычным способом. Младшие 32 бита этого числа передаются в возвращаемом значении функции, а старшие 32 бита записываются в DWORD-значение, на которое ссылается параметр *pdwFileSizeHigh*. Удобнее использовать структуру ULARGEJNTEGER следующим образом:

```
ULARGE_INTEGER ulFileSize;
ulFileSize.LowPart = GetCompressedFileSize(TEXT("SomeFile.dat"),
    &ulFileSize.HighPart);

// Теперь 64-разрядное значение размера файла содержится в ulFileSize.LowPart
```

Установка указателя в файле

Вызов *CreateFile* заставляет систему создать объект ядра «файл», управляющий работой с файлом. Внутри этого объекта ядра содержится указатель, определяющий 64-разрядное смещение внутри файла, по которому будет выполнена следующая синхронная операция чтения или записи. Исходно этот указатель установлен на 0, следовательно, если вызвать *ReadFile* сразу после *CreateFile*, файл будет прочитан с начала (т.е. с нулевой позиции). Если при этом было прочитано 10 байтов, система перемещает на 10 байтов указатель, связанный с описателем файла, поэтому при следующем вызове *ReadFile* чтение файла начнется уже с 11-го байта. Рассмотрим следующий пример кода, считывающего из файла в буфер первые 10 байтов, а затем еще 10 байтов.

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile = CreateFile(TEXT("MyFile.dat"), …);   // Указатель установлен на 0
ReadFlle(hFile, pb, 10, &dwNumBytes, NULL);         // Чтение байтов  0 - 9
ReadFile(hFile, pb, 10, idwNumBytes, NULL);         // Чтение байтов 10 - 19
```

У каждого объекта ядра имеется свой собственный указатель, поэтому, открыв файл два раза подряд, вы получите немного неожиданные результаты:

```
BYTE pb[10];
```

```
DWORD dwNumBytes;
HANDLE hFile1 = CreateFile(TEXT( "MyFile.dat"), …);          // Указатель
                                                             // установлен на 0
HANDLE hFile2 = CreateFile(TEXT("MyFile.dat"), ...);         // Указатель
                                                             // установлен на 0
ReadFile(hFile1, pb, 10, MwNumBytes, NULL);                  // Чтение байтов 0-9
ReadFile(hFile2, pb, 10, &dwNumBytes, NULL);                 // Чтение байтов 0-9
```

В этом примере два объекта ядра управляют одним и тем же файлом. Поскольку у каждого объекта ядра «файл» есть свой указатель, манипулирование файлом с помощью одного из объектов никак не влияет на указатель в другом объекте. В итоге первые 10 байтов будут прочитаны дважды.

Приведу еще один пример, который поможет вам разобраться в этом:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile1 = CreateFile(TEXT("MyFile.dat"), …);           // Указатель
                                                             // установлен на 0
HANDLE hFile2;
DuplicateHandle(
    GetCurrentProcess(), hFile1,
    GetCurrentProcess(), &hFile2,
    0, FALSE, 0UPLICATE_SANE_ACCESS);
ReadFile(hFile1, pb, 10, &dwNumBytes, NULL);                 // Чтение байтов 0-9
ReadFile(hFile2, pb, 10, &dwNumBytes, NULL);                 // Чтение байтов 10-19
```

В этом примере один объект ядра «файл» связан с двумя описателями одного и того же файла. В результате положение указателя в файле будет обновляться независимо от того, какой из объектов ядра используется для манипулирования файлом.

Для случайного доступа к файлу требуется возможность модификации указателя, связанного с объектом ядра «файл». Делается это вызовом функции *SetFilePointerEx*:

```
B00L SetFilePointerEx(
    HANDLE          hFile,
    LARGE_INTEGER   liDistanceToMove,
    PLARGE_INTEGER pliNewFilePointer,
    DWORD           dwMoveMethod);
```

Параметр *hFile* определяет объект ядра «файл», указатель которого требуется изменить. Параметр *liDistanceToMove* сообщает системе, на сколько байт следует переместить указатель. Значение этого параметра прибавляется к текущему значению указателя, поэтому отрицательные значения *liDistanceToMove* перемещают указатель в обратном направлении (к началу файла). Последний параметр, *dwMoveMethod*, указывает, как функция *SetFilePointerEx* должна интерпретировать параметр *liDistanceToMove*. Возможные значения этого параметра перечислены в табл. 10-8.

Табл. 10-8. Значения параметра *dwMoveMethod* функции *SetFilePointerEx*

Значение	Описание
FILE_BEGIN	Указатель объекта «файл» перемещается в положение, заданное параметром *HDistanceTbMove*, который интерпретируется как 64-разрядное значение без знака
FILE_CURRENT	Значение параметра *HDistanceTbMove* прибавляется к текущему значению указателя объекта «файл». Учтите, что отрицательные значения интерпретируются как смещения к началу файла, что позволяет вести поиск в обоих направлениях
FILE_END	Новая позиция указателя вычисляется как сумма логического размера файла и значения пара метра *HDistanceToMove*, которое интерпретируется как 64-разрядное значение со знаком. Это позволяет вести поиск в файле в обоих направлениях

Обновив указатель в объекте «файл», функция *SetFilePointerEx* возвращает новое значение указателя в параметре *pliNewFilePointer*, содержащем ссылку на LARGEINTEGER. Можно отказаться от получения нового значение указателя, передайте NULL в параметре *pliNewFilePointer*.

Обратите внимание не несколько моментов, касающихся *SetFilePointerEx*:

- допускается устанавливать указатель дальше конца файла. При этом размер файла не увеличивается, если только вы не запишете в файл данные или не вызовете функцию *SetEndOfFile*;
- при использовании функции *SetFilePointerEx* с файлом, открытым с флагом FILE_FLAG_NO_BUFFERING, устанавливать указатель можно только на границах, выровненных по размеру сектору (как это делается, я покажу на примере программы *FileCopy* ниже в этой главе);
- Windows не поддерживает функцию *GetFilePointerEx* для определения положения указателя. Но можно получить желаемое значение, переместив указатель на 0 байтов вызовом *SetFilePointerEx*, как показано ниже:

```
LARGE_IHTEGER liCurrentPosition = { 0 };
SetFilePointerEx(hFile, liCurrentPosition, &liCurrentPosition, FILE_CURRENT);
```

Установка конца файла

Обычно система автоматически устанавливает конец файла при его закрытии. Однако в некоторых случаях требуется принудительно сделать файл больше или меньше. Для этого вызывают следующую функцию:

```
BOOL SetEndOfFile(HANDLE hFile);
```

Функция *SetEndOfFile* уменьшает или увеличивает файл до размера, заданного текущим положением указателя в файле. Так, если вы хотите принудительно назначить файлу размер 1024 байта, вызовите *SetEndOfFile* следующим образом:

```
HANDLE hFile = CreateFile(…);
LARGE_INTEGER liDistanceToMove;
liDistanceToMove.QuadPart = 1024;
SetFilePointerEx(hFile, liDistanceToMove, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
CloseHandle(hFile);
```

Если проверить свойства такого файла с помощью Windows Explorer, его размер окажется равным в точности 1024 байтам.

Синхронный ввод-вывод на устройствах

В этом разделе рассказывается о Windows-функциях для синхронного ввода-вывода на устройствах. Помните, что к устройствам относятся файлы, почтовые ящики, каналы, сокеты и пр. Независимо от типа устройства для синхронного ввода-вывода используются одни и те же функции.

Несомненно, самыми простыми и востребованными функциям для чтения-записи на устройствах являются *ReadFile* и *WriteFile*:

```
BOOL ReadFile(
    HANDLE       hFile,
    PVOID        pvBuffer,
    DWORD        nNumBytesToRead,
    PDWORD       pdwNumBytes,
    OVERLAPPED*  pOverlapped);

BOOL WriteFile(
    HANDLE       hFile,
    CONST VOID   *pvBuffer,
    DWORD        nNumBytesToWrite,
    PDWORD       pdwNumBytes,
    OVERLAPPED*  pOverlapped);
```

Параметр *hFile* задает описатель нужного устройства. Для открытого устройства нельзя задавать флаг FILE_FLAG_OVERLAPPED, иначе система подумает, что вы хотите работать с устройством асинхронно. Параметр *pvBuffer* ссылается на буфер для хранения прочитанных данных либо данных, подлежащих записи. Параметры *nNumBytesToRead* и *nNumBytesToWrite* сообщают функциям *ReadFile* и *WriteFile*, сколько байтов следует прочитать с устройства либо записать на него, соответственно.

Параметр *pdwNumBytes указывает адрес* DWORD-переменной, в которую функция записывает число байтов, успешно полученных с устройства либо переданных на него. Последний параметр, *pOverlapped*, в случае синхронного ввода-вывода устанавливают в NULL. Подробнее о нем — в разделе, посвященном асинхронному вводу-выводу.

При успешном завершении вызовы *ReadFile* и *WriteFile* возвращают TRUE. Кстати, *ReadFile* можно вызывать только для устройств, открытых с

флагом GENERIC_READ. Аналогично, *WriteFile* вызывают только для **уст**ройств, открытых с флагом GENERIC_WRITE.

Сброс данных на устройство

Вспомните, что функция *CreateFile* принимает целый ряд флагов, которые определяют, как система каптирует содержимое файлов. Другие устройства, такие как последовательные порты, почтовые ящики и канала, также способны кэшировать файлы. Чтобы заставить систему записать кэшированные данные на устройство, можно воспользоваться функцией *FlushFileBuffers:*

```
BOOL FlushFileBuffers(HANDLE hFile);
```

Эта функция принуждает систему сбросить все содержимое буферов на устройство, заданное параметром *hFile*. Для этого устройство должно быть открыто с флагом GENERIC_WRITE. При успешном завершении вызова функция возвращает TRUE.

Отмена синхронного ввода-вывода

Функции для синхронного ввода-вывода просты в использовании, но до завершения их вызова поток прекращает все остальные операции. Типичным примером может служить функция *CreateFile*. Когда пользователь вводит информацию с помощью мыши или клавиатуры происходит вставка оконных сообщений в очередь потока — создателя окна, в которое выполняется ввод. Если поток оказался заблокированным в ожидании завершения вызова *CreateFile,* обработка оконных сообщений останавливается и все окна, созданные этим потоком, «застывают». Блокирование потоков в ожидании завершения синхронного ввода-вывода — наиболее распространенная причина «зависаний» приложений!

Майкрософт сделала в Windows Vista ряд заметных нововведений, призванных решить эту проблему. Так, при зависании консольного приложения из-за синхронного ввода-вывода пользователь теперь может, нажав Ctrl+C, вернуть себе контроль над консолью и продолжить работу с ней, не «убивая» процесс консоли. Кроме того, диалоги сохранения и открытия файлов содержат кнопку Cancel, которой можно отменить чрезмерно затянувшуюся операцию (например, обращение к файлу, расположенному на сетевом сервере).

Чтобы приложение реагировало на действие пользователя без задержек, следует по максимуму использовать асинхронный ввод-вывод. Кроме того, этот подход позволяет обойтись в приложении минимумом потоков, что экономит ресурсы (такие как объекты ядра «поток» и «стек»). Для асинхронных операций также проще реализовать возможность отмены. Например, Internet Explorer позволяет отменять веб-запросы (щелчком красной кнопки с белым крестом либо нажатием клавиши Esc), если их исполнение слишком затянулось и пользователь больше не хочет ждать.

К сожалению, некоторые API-функции, такие как *CreateFile*, не поддерживают асинхронный вызов методов. Некоторые из этих методов могут завершаться по тайм-ауту (если их исполнение заняло слишком много времени, например, обращение к сетевому серверу), но лучше бы иметь функцию, принудительно завершающую ожидание или просто отменяющую синхронный ввод-вывод. В Windows Vista следующая функция позволяет отменить незаконченные операции синхронного ввода-вывода для заданного потока:

```
BOOL CancelSynchronouslo(HANDLE hThread);
```

Параметр *hThread* — это описатель потока, приостановленного в ожидании завершения синхронного запроса ввода-вывода. Данный описатель должен разрешать завершение (THREAD_TERMINATE) потока. В противном случае вызов *CancelSynchronousIo* заканчивается неудачей, а GetLastError возвращает ERROR_ACCESS_DENIED. Описатели созданных вами потоков разрешают полный доступ к ним (THREAD_ALL_ACCESS), в том числе для завершения (THREAD_TERMINATE). Если же вы используете поток из пула либо код для отмены вызывается по таймеру, как правило, приходится вызывать *OpenThread*, чтобы получить описатель потока с соответствующим идентификатором; при этом не забудьте передать THREAD_TERMINATE как первый параметр.

Если заданный поток приостановлен в ожидании завершения синхронной операции ввода-вывода, вызов *CancelSynchronousIo* пробудит этот поток, а операция, которую он ожидал, завершится неудачей. При этом вызов *GetLastError* вернет ERROR_OPERATION_ABORTED, а *CancelSynchronousIo* вернет TRUE вызвавшему его потоку.

Заметьте, что вызывающий *CancelSynchronousIo* поток не «знает», что именно делает поток, заблокированный на синхронной операции. Возможно, он просто был вытеснен, так и не успев обратиться к устройству. Этот поток также может ожидать ответа устройства либо устройство ответило, и поток уже выходит их синхронной операции. Если функция *CancelSynchronousIo* вызвана на потоке, который не ждал отклика устройства, вызов *CancelSynchronousIo* возвращает FALSE, а *GetLastError* — ERROR_NOT_FOUND.

По этой причине имеет смысл использовать дополнительные средства синхронизации потоков (см. главы 8 и 9), чтобы отменять только синхронные операции. Однако на практике это обычно не требуется, поскольку инициатором отмены обычно является пользователь, обнаруживший, что приложение «зависло». Кроме того, пользователь может попытаться еще раз отменить операцию, если ему покажется, что первая попытка не удалась. Кстати, Windows автоматически вызывает *CancelSynchronousIo*, чтобы вернуть пользователю контроль над консолью либо диалогом сохранения или открытия файла.

Внимание! Возможность отмены запросов ввода-вывода зависит от реализации соответствующего уровня в драйвере. Драйвер может и не поддерживать отмену. В этом случае *CancelSynchronousIo* все равно вернет TRUE, поскольку функция обнаружила запрос, который необходимо пометить для отмены. Собственно, отмена находится в компетенции драйвера. Примером обновленного драйвера, поддерживающего отмену синхронных операций в Windows Vista, может быть сетевой редиректор.

Асинхронный ввод-вывод на устройствах: основы

Ввод-вывод на устройствах — самая медленная и наименее предсказуемая категория операций, выполняемых компьютером. Процессор выполняет арифметические действия и даже закрашивает экран намного быстрее, чем чтение-запись данных в файлы, в том числе через сеть. Однако применение асинхронного ввода-вывода позволяет оптимизировать использование ресурсов и создавать более эффективные приложения.

Рассмотрим поток, генерирующий асинхронный запрос ввода-вывода. Этот запрос передается драйверу устройства, который в итоге берет на себя ответственность за исполнение» собственно, ввода-вывода. Пока драйвер ждет отклика устройства, поток приложения не будет приостановлен в ожидании завершения запроса ввода-вывода, а продолжит исполнение других полезных операций.

В какой-то момент драйвер устройства завершит обработку запросов ввода-вывода и ему придется уведомить приложение об успешной отправке или приеме данных либо об ошибке. Подробнее об этом рассказывается далее, а сейчас разберем постановку запросов ввода-вывода в очередь. Очереди запросов ввода-вывода — важнейший механизм для проектирования высокопроизводительных масштабируемых приложений, о котором и пойдет речь в остальных разделах этой главы.

Для асинхронного доступа к устройству последнее сначала необходимо открыть вызовом *CreateFile* с передачей флага FILE_FLAG_OVERLAPPED в параметре dwFlagsAndAttributes. Этот флаг уведомляет систему о том, что вы планируете работать с устройством асинхронно.

Для постановки запросов ввода-вывода в очередь драйвера устройства применяют уже известные вам функции *ReadFile* и *WriteFile* (см. раздел о синхронном вводе-выводе выше). Для удобства я снова приведу прототипы этих функций:

```
BOOL ReadFile(
   HANDLE       hFile,
   PVOID        pvBuffer,
   DWORD        nNumBytesToRead,
```

```
    PDWORD       pdwNumBytes,
    OVERLAPPED*  pOverlapped);

BOOL WriteFile(
    HANDLE       hFile,
    CONST VOID   *pvBuffer,
    DWORD        nNumBytesToWrite,
    PDWORD       pdwNumBytes,
    OVERLAPPED* pOverlapped);
```

При вызове любая из этих функций проверяет, не открыто ли заданное параметром *hFile* устройство с флагом FILE_FLAG_OVERLAPPED. Если это так, функция выполняет запрошенную операцию ввода-вывода асинхронно. Кстати, при вызове этих функций для асинхронного ввода-вывода в параметре *pdwNumBytes* можно передать NULL (обычно так и делается), что и понятно: этот вызов должен вернуть управление до завершения запроса ввода-вывода, следовательно, считать число переданных байтов в этот момент бессмысленно.

Структура OVERLAPPED

При выполнении асинхронного ввода-вывода на устройствах необходимо передавать адрес инициализированной структуры OVERLAPPED через параметр *pOverlapped*. В этом контексте название структуры (overlapped — по-английски «перекрывающийся») означает, что исполнение запросов ввода-вывода перекрывается по времени с исполнением потоком других операций. Вот как выглядит структура OVERLAPPED:

```
typedef struct _OVERLAPPED {
    DWORD  Internal;         // [вывод] код ошибки
    DWORD  InternalHigh;     // [вывод] число переданных байтов
    DWORD  Offset;           // [ввод]  смещение в файле, младшие 32 бита
    DWORD  OffsetHigh;       // [ввод]  смещение в файле, старшие 32 бита
    HANDLE hEvent;           // [ввод]  описатель события или данных
} OVERLAPPED, *LPOVERLAPPED;
```

Эта структура включает пять элементов. Три из них — *Offset, OffsetHigh* и *hEvent* — должны быть инициализированы до вызова *ReadFile или WriteFile.* Остальные два, Internal и *InternalHigh,* устанавливаются драйвером устройства и могут быть прочитаны по завершении операции ввода-вывода. Рассмотрим их поподробнее.

- *Offset* и *OffsetHigh*

  При обращении к файлу в эти элементы записывают 64-разрядное смещение, определяющее позицию в файле, с которой начнется операция ввода-вывода. Вспомните, что с каждым объектом ядра «файл» связан указатель. При синхронном запросе ввода этот указатель определяет, с какого места начнется обращение к файлу. По завершении операции система ав-

томатически обновляет указатель, чтобы начать следующую операцию с того места, на котором закончилась предыдущая. При асинхронных операциях ввода-вывода система игнорирует этот указатель. Представьте, что будет, если ваш код два раза асинхронно вызовет *ReadFile* (на одном и том же объекте ядра). В этом случае система не узнает, откуда следует начать чтение при втором вызове *ReadFile*. Ясно только, что читать область файла, уже прочитанную при первом вызове *ReadFile*, не нужно. Поэтому во избежание путаницы при повторных асинхронных вызовах на одном и том же объекте с каждым асинхронным запросом ввода-вывода необходимо передавать в составе структуры OVERLAPPED смещение в файле. Заметьте, что элементы *Offset* и *OffsetHigh* не игнорируются, даже если устройство не является файлом. Эти элементы необходимо установить в 0, в противном случае запрос ввода-вывода закончится неудачей, а *GetLastError* вернет ERROR_INVALID_PARAMETER.

- *hEvent*
  Этот элемент используется одним в одном из методов получения уведомлений о завершении ввода-вывода. В методе, основанном на вводе-выводе с оповещениями, это элемент можно использовать для собственных целей. Я знаю многих разработчиков, хранящих в *hEvent* адреса различных C++-объектов (подробнее об этом элементе см. далее в этой главе.)
- *Internal*
  Это поле хранит код ошибки, сгенерированный при обработке запроса ввода-вывода. В случае асинхронного запроса драйвер устанавливает для *Internal* значение STATUS_PENDING, свидетельствующее об отсутствии ошибок (поскольку ввод-вывод еще не начался). Проверить, завершена ли асинхронная операция ввода-вывода, можно при помощи макроса *HasOverlappedloCompleted*, определенного в WmBase.h. Если операция еще не закончилась, макрос вернет FALSE, а в противном случае — TRUE. Вот определение этого макроса:

```
#define HasOverlappedloCompleted(pOverlapped) \
   ((pOverlapped)->Internal != STATUS_PENDING)
```

- *InternalHigh*
  По завершении асинхронного запроса ввода-вывода в это поле записывается число переданных байтов.
  Изначально при разработке структуры OVERLAPPED Майкрософт решила не документировать поля *Internal* и *InternalHigh* (отсюда их названия). Со временем стало ясно, что хранящаяся в этих поля информация может оказаться полезной разработчикам, в результате поля были задокументированы. Несмотря на это, полям оставили прежние имена, поскольку они часто используются в коде операционной систем, а его в Майкрософт изменять не хотели.

Примечание. По завершении асинхронного запроса ввода-вывода вы получаете адрес структуры OVERLAPPED, использованной при генерации запроса. Сведения о контексте, переданные в OVERLAPPED, часто оказываются полезными. Например, в этой структуре можно сохранить описатель устройства, которому адресован запрос ввода-вывода. У этой структуры нет ни специального поля для хранения описателя устройства, ни других полей для хранения потенциально полезных сведений о контексте, но это проблему довольного легко решить.

Я часто создаю на основе структуры OVERLAPPED производные C++-классы, способные хранить любую необходимую мне информацию. Получив в своих приложениях адрес структуры OVERLAPPED, я просто привожу его к указателю на экземпляр моего C++-класса. Так я получаю доступ ко всем элементам OVERLAPPED и любой контекстной информации, необходимой моему приложению. Этот метод демонстрируется на примере программы *FileCopy*, показанной в конце этой главы (см. код C++-класса *FileCopy*).

Асинхронный ввод-вывод на устройствах: «подводные камни»

Используя асинхронный ввод-вывод, следует быть в курсе пары сложностей. Во-первых, драйверы устройств не обязаны обрабатывать очередь запросов ввода-вывода по принципу «первый вошел, первый вышел» (first-in first-out, FIFO). Так, в случае потока, исполняющего показанный ниже код, драйвер устройства вполне способен сначала записать данные файла, а потом прочитать этот файл:

```
OVERLAPPED o1 = { 0 };
OVERLAPPED o2 = { 0 };
BYTE bBuffer[100];
ReadFile (hFile, bBuffer, 100, NULL, &o1);
WriteFile(hFile, bBuffer, 100, NULL, &o2);
```

Как правило, драйверы устройств исполняют запросы в порядке, оптимальном с точки зрения производительности. Например, стремясь уменьшить число перемещений головки жесткого диска и времени доступа, драйвер файловой системы может выбрать из очереди запросы ввода-вывода, обращающихся к физически близким областям диска.

Вторая сложность связана с проверкой наличия ошибок. Большинство Windows-функций сообщают об ошибке, возвращая FALSE, а об успехе — возвращая значение, отличное от нуля. Однако функции *ReadFile* и *WriteFile* в этом плане отличаются. Чем — разберемся на примере.

При попытке поставить в очередь асинхронный запрос ввода-вывода драйвер устройства может исполнить этот запрос синхронно. Так бывает, например, если при чтении файла система обнаружила, что запрошенные данные уже находятся в кэше. В этом случае запрос ввода-вывода не попа-

дает в очередь драйвера: система просто копирует нужные данные из кэша в ваш буфер — и все, запрос обработаю. Некоторые операции, такие как сжатие NTFS, увеличение размеров и дописывание данных к файлам драйверы всегда выполняют синхронно (подробнее см. по ссылке http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B156932).

Функции *ReadFile* и *WriteFile* возвращают отличное от нуля значение, если запрос ввода-вывода был выполнен Синхронно. Если же он выполнен синхронно, либо при вызове *ReadFile* или *WriteFile* возникла ошибка, возвращается FALSE. Получив FALSE, вы должны вызвать *GetLastError*, чтобы выяснить, что именно случилось. Если *GetLastError* возвращает ERROR_IO_PENDING, запрос ввода вывода успешно поставлен в очередь и будет выполнен позже.

Если же *GetLastError* вернет значение, отличное от ERROR_IO_PENDING, поставить запрос ввода-вывода в очередь драйвера не удалось. Ниже приводятся коды ошибок при постановке запросов ввода-вывода в очередь драйвера устройства, которые *GetLastError* возвращает чаще всего:

- ERROR_INVALID_USER_BUFFER или ERROR_NOT_ENOUGH_MEMORY
  Каждый драйвер устройства поддерживает список незавершенных запросов ввода-вывода, хранящийся в невыгружаемом пуле и включающий фиксированное число элементов. Если этот список заполнен, добавить в него новые запросы невозможно, в результате *ReadFile* и *WriteFile* возвращают FALSE, а *GetLastError* — один из показанных выше кодов ошибок.
- ERROR_NOT_ENOUGH_QUOTA
  Некоторые устройства требуют блокировать в ОЗУ страницы памяти, составляющие выделенный вами буфер, во избежание выкачивания их на диск в то время, пока запрос стоит в очереди. Типичный пример — файловый ввод-вывод при использовании флага FILE_FLAG_NO_BUFFERING. Однако система ограничивает число страниц, которое может заблокировать отдельный процесс. Если функциям *ReadFile* и *WriteFile* не удается заблокировать нужное число страниц, эти функции возвращают FALSE, а *GetLastError* — ERROR_NOT_ENOUGH_QUOTA. Увеличить квоту процесса можно, вызвав *SetProcessWorkingSetSize*.

Как обрабатывать такие ошибки? В сущности, они возникают из-за превышения лимита необработанных запросов ввода-вывода, поэтому следует дождаться завершения некоторых запросов и повторно вызвать *ReadFile* или *WriteFile*.

И третье, что нужно знать об асинхронном вводе-выводе: буфер с данными и структуру OVERLAPPED нельзя ни перемещать, ни уничтожать до завершения соответствующего запроса ввода-вывода. Дело в том, что при постановке запроса в очередь драйвера устройства последний получает адреса буфера и структуры OVERLAPPED, именно адреса, а не соответствующие блоки памяти. Почему это так, вполне понятно: копирование блоков памяти отнимает массу ресурсов и процессорного времени.

Подготовившись к обработке очередного запроса, драйвер передает данные, которые находятся по адресу, заданному параметром *pvBuffer*, а затем обращается к полям, содержащим смещение в файле, и другим элементам структуры OVERLAPPED, адрес которой задан параметром *pOverlapped*. В частности, драйвер записывает в поле *Internal* код ошибки ввода-вывода, а в поле *InternalHigh* — число переданных байтов.

> **Примечание.** Во избежание повреждения памяти ни в коем случае не перемещайте и не разрушайте эти буферы до завершения запроса ввода-вывода. Кроме того, для каждого запроса ввода-вывода вы должны создать и инициализировать отдельную структуру OVERLAPPED.

Предыдущее замечание чрезвычайно важно, поскольку именно по этой причине разработчики чаще всего допускают ошибки, реализуя асинхронный ввод-вывод в приложениях. Вот пример того, как не нужно писать код:

```
VOID ReadData(HANDLE hFile) {
    OVERLAPPED o = { 0 };
    BYTE b[100];
    ReadFile(hFile, b. 100, NULL, &o);
}
```

Этот код выглядит довольно безобидно, и вызов *ReadFile* написан вроде бы безупречно. Проблема лишь в том, что эта функция возвращает управление после постановки в очередь асинхронного запроса ввода-вывода. При этом буфер и структура OVERLAPPED удаляются из стека потока, а драйвер устройства не «в курсе», что вызов *ReadData* завершен и по-прежнему использует переданные ему адреса двух блоков памяти в стеке потока. Когда запрос ввода-вывода завершится, драйвер запишет результаты в стек потока. В результате будут затерты все данные, оказавшиеся к моменту завершения запроса в переданных драйверу областях памяти. Диагностировать такие ошибки особенно сложно, поскольку модификация памяти происходит асинхронно. Иногда драйвер ввод-вывод синхронно, и тогда ошибка не проявляется. В других случаях ввод-вывод может завершиться сразу после вызова *ReadData*, а может через час и более — кто знает, что окажется в стеке к этому моменту?

Отмена запросов ввода-вывода, ожидающих в очереди

Иногда требуется отменить поставленный в очередь запрос ввода-вывода до того, как он будет обработан драйвером устройства. *Windows позволяет* сделать это несколькими способами:

- функция *CancelIo* позволяет отменить все запросы ввода-вывода, сгенерированные вызывающим потоком для заданного описателя (если только этот описатель не связан с портом завершения ввода-вывода):

```
BOOL Cancello(HANDLE hFile);
```

- закрыв описатель устройства, можно отменить все адресованные ему запросы ввода-вывода, независимо от сгенерировавшего их потока;
- при завершении потока система автоматически отменяет все сгенерированные им запросы ввода-вывода за исключением запросов на описатели, связанные с портами завершения ввода-вывода;
- отменить отдельный запрос ввода-вывода можно вызовом функции *CancelIoEx*:

```
BOOL CancelIoEx(HANDLE hFile, LPOVERLAPPED pOverlapped);
```

CancelIoEx позволяет отменять незавершенные запросы ввода-вывода, сгенерированные потоком, отличным от вызывающего потока. Эта функция помечает как отмененные все запросы, заданные значениями параметров *hFile* и *pOverlapped*. Поскольку каждому из незавершенных запросов ввода-вывода соответствует уникальная структура OVERLAPPED, вызов *CancelloEx* отменяет один-единственный запрос. Если же параметр pOverlapped содержит NULL, *CancelloEx* отменяет все незавершенные запросы ввода-вывода, адресованные устройству с описателем, заданным параметром *hFile*.

> **Примечание.** Отмененные запросы ввода-вывода завершаются с кодом ошибки ERROR_OPERATION_ABORTED.

Уведомление о завершении ввода-вывода

Итак, вы уже знаете, как поставить в очередь асинхронный запрос ввода-вывода, а теперь поговорим о том, как драйвер устройства уведомляет приложения о том, что он завершил обработку запросов ввода-вывода.

Windows поддерживает четыре способа уведомления о завершении ввода-вывода (см. табл. 10-9), ниже мы разберем каждый из них подробно. Эти методы осуждаются в порядке от простейшего в реализации (освобождение объекта ядра «устройство») до самого сложного (порты завершения ввода-вывода).

Табл. 10-9. Методы уведомления о завершении ввода-вывода

Метод	Описание
Освобождение объекта ядра	Неудобен в ситуациях, когда одному устройству адресовано «устройство» сразу несколько запросов ввода-вывода. Позволяет потоку обработать результаты запроса, сгенерированного другим потоком
Освобождение объекта ядра	Поддерживает ситуации, когда одному устройству «событие» адресовано сразу несколько запросов ввода-вывода. Позволяет потоку обработать результаты запроса, сгенерированного другим потоком

Табл. 10-9. (окончание)

Метод	Описание
Ввод-вывод с оповещением	Поддерживает ситуации, когда одному устройству адресовано сразу несколько запросов ввода-вывода. Результаты запроса ввода-вывода должен обработать поток, который сгенерировал этот запрос
Порты завершения ввода-вывода	Поддерживает ситуации, когда одному устройству адресовано сразу несколько запросов ввода-вывода. Позволяет потокам обрабатывать результаты запросов, сгенерированных другими потоками. Обеспечивает наибольшую масштабируемость и гибкость

Как сказано выше, порты завершения ввода-вывода — наилучший метод получения уведомлений о завершении ввода-вывода. Однако рекомендую вам изучить все четыре метода, чтобы понять, зачем Майкрософт добавила порты завершения ввода-вывода и этот механизм позволяет решать проблемы, характернее для менее совершенных методов.

Освобождение объекта ядра «устройство»

Сгенерировав асинхронный запрос ввода-вывода, поток продолжает работать, исполняя различные полезные операции. Однако в итоге поток должен синхронизироваться с исполнением запрошенной им операции ввода-вывода. Иными словами, рано или поздно настанет момент, когда поток не сможет продолжить работу, не получив в полном объеме с устройства необходимые ему данные.

В Windows объекты ядра «устройство» используются для синхронизации потоков, поэтому такой объект может быть свободен либо занят. Функции *ReadFile* и *WriteFile* переводят объект ядра устройство в состояние «занят» непосредственно перед постановкой в очередь запроса ввода-вывода. Обработав запрос ввода-вывода, драйвер устройства переводит этот объект ядра в состояние «свободен».

Поток может узнать, завершен ли запрос ввода-вывода, вызвав функцию *WaitForSingleObject* либо *WaitForMultipleObjects*. Вот простой пример:

```
HANDLE hFile = CreateFile(…, FILE_FLAG_OVERLAPPED, …);
BYTE bBuffer[100];
OVERLAPPED o = { 0 };
o.Offset = 345;

BOOL bReadDone = ReadFile(hFile, bBuffer, 100, NULL, &o);
DWORD dwError = GetLastError();

if ((bReadDone && (dwError == ERR0R_I0_PENDING)) {
    // Ввод-вывод выполняется синхронно, ждем его завершения
```

```
    WaitForSingleObject(hFile, INFINITE);
    bReadDone = TRUE;
}

if (bReadDone) {
    // o.Internal содержит код ошибки ввода-вывода
    // o.InternalHigh содержит число переданных байтов
    // bBuffer содержит прочитанные данные
} else {
    // An error occurred; see dwError
}
```

Этот код генерирует асинхронный запрос ввода-вывода, после чего ожидает его завершения, что абсолютно противоречит логике асинхронного ввода-вывода! Ясно, что вы никогда не напишете такой код для реального приложения, я же использую его как иллюстрацию важных моментов, о которых говорится ниже.

- Устройство должно быть открыто для асинхронного ввода-вывода с флагом FILE_FLAG_OVERLAPPED.
- Элементы *Offset, OffsetHigfr* и *hEvent* структуры OVERLAPPED должны быть инициализированы. В этом примере я установил их в 0 за исключением *Offset*, которому присвоено значение 345, чтобы *ReadFile* начала чтение файла с 346-го байта,
- Значение, возвращаемое функцией *ReadFile*, записывается в переменную *bReadDone*. Проверив ее, можно узнать, выполнен ли запрос ввода-вывода синхронно.
- Если запрос ввода-вывода не был выполнен синхронно, я пытаюсь выяснить, не возникла ли ошибка, либо запрос был выполнен асинхронно. Это делается путем сравнения результата *GetLastError* с ERROR_IO_PENDING.
- Чтобы поток дождался результатов исполнения запроса, я вызываю *WaitForSingleObject* и передаю ей описатель объекта ядра «устройство». Как сказано в главе 9, вызов этой функции приостанавливает поток до освобождения объекта ядра. Драйвер устройства освобождает этот объект ядра после завершения ввода-вывода. Когда *VMtForSingleObject* возвращает управление, ввод-вывод уже завершен, и я устанавливаю *bReadDone* в TRUE.
- Когда данные будут прочитаны, можно проверить данные в *bBuffer*, код ошибки и число переданных байтов, соответственно, в полях *Internal* и *InternalHigh* структуры OVERLAPPED.
- Если возникла ошибка, ее код и дополнительные сведения можно получить, проверив *dwError*.

Освобождение объекта ядра «событие»

Этот метод получения уведомлений о завершении ввода-вывода очень прост и напоминает предыдущий. Однако на практике он не очень удобен, так как плохо работает, когда запросов ввода-вывода много. Предположим, что ваша программа пытается выполнить сразу несколько асинхронных операций (например, записать и прочитать 10 байт) над одним и тем же файлом. Вот пример такой программы:

```
HANDLE hFile = CreateFile(…, FILE_FLAG_0VERLAPPED, …);

BYTE bReadBuffer[10];
OVERLAPPED oRead = { 0 };
oRead.Offset = 0;
ReadFile(hFile, bReadBuffer, 10, NULL, &oRead);

BYTE bWriteBuffer[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
OVERLAPPED oWrite = { 0 };
oWrite.Offset =10;
WriteFile(hFile, bWriteBuffer, _countof(bWriteBuffer), NULL, &oWrite);
…
WaitForSingle0bject(hFile, INFINITE);

// Неизвестно, какая операция завершилась: чтение, запись или обе операции?
```

Синхронизация потоков путем ожидания устройства невозможно, поскольку объект «устройство» освободиться по завершении любой из операций. Если вызвать функцию *WaitForSingleObject* и передать ей описатель устройства, сложно будет сказать, что стало причиной возврата управления этой функции: завершение чтения, записи или обеих операций. Ясно, что должен быть более эффективный способ координации одновременных запросов ввода-вывода, исполняемых асинхронно, и он, к счастью, существует.

Последний элемент структуры OVERLAPPED, *hEvent*, идентифицирует объект ядра «событие». Вы должны создать этот объект вызовом *CreateEvent*. По завершении асинхронного запроса ввода-вывода драйвер устройства проверяет, не установлен ли элемент *hEvent* структуры OVERLAPPED в NULL и, если нет, освобождает содержащийся в нем объект-событие, вызывая *SetEvent*. Кроме того, драйвер, как обычно, освобождает объект «устройство». Если проверка завершения операций на устройстве выполняется с помощью событий, следует ожидать освобождения не объекта «устройство», а соответствующего объекта «событие».

> **Примечание.** Можно немного повысить производительность, приказав Windows не освобождать объект «файл» сразу после завершения операции. Для это вызовите функцию *SetFileCompletionNotificationModes* следующим образом:

```
BOOL SetFileCompletionNotificationMode8(HANDLE hFile, UCHAR uFlags);
```

Параметр *hFile* содержит описатель файла, параметр *uFlags* определяет, что должна сделать Windows после завершения операции ввода-вывода. Если передать в этом параметре флаг FILE_SKIP_SET_EVENT_ON_HANDLE, Windows не освободит объект «файл» по завершении операции с файлом. К сожалению, флаг FILE_SKIP_SET_EVENT_ON_HANDLE назван очень неудачно, ему куда больше подошло бы имя вроде FILE_SKIP_SIGNAL.

Чтобы одновременно выполнить несколько асинхронных запросов ввода-вывода на устройстве, следует создать отдельный объект «событие» для каждого запрос, инициализировать элемент hEvent структуры OVERLAPPED у каждого запроса и вызвать *ReadFile* или *WriteFile*. Достигнув момента, когда продолжение работы невозможно без результатов запроса ввода-вывода, просто вызовите функцию *WaitForMultipleObjects*, передав ей описатели событий, связанные со структурами OVERLAPPED запросов, исполнения которых необходимо дождаться. Такой алгоритм позволяет просто и надежно выполнять одновременные асинхронные запросы ввода-вывода с использованием одного и того же объекта «устройство». Следующий пример иллюстрирует этот подход:

```
HANDLE hFile = CreateFile(…, FILE_FLAG_OVERLAPPED, …);

BYTE bReadBuffer[10];
OVERLAPPED oRead = { 0 };
oRead.Offset = 0;
oRead.hEvent = CreateEvent(…);
ReadFile(hFile, bReadBuffer, 10, NULL, &oRead);

BYTE bWriteBuffer[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
OVERLAPPED oWrite = { 0 };
oWrite.Offset = 10;
oWrite.hEvent = CreateEvent(…);
WriteFile(hFile, bWriteBuffer, .qountof(bWriteBuffer), NULL, AoWrite);
…

HANDLE h[2];
h[0] = oRead.hEvent;
h[1] = oWrite.hEvent;
DWORD dw = WaitForMultiple0bjects(2, h, FALSE, INFINITE);
switch (dw - WAIT_0BJECT_0) {
    case 0:    // чтение завершено
       break;

    case 1:    // запись завершена
       break;
}
```

В реальных приложениях такой код не используется, но зато он хорошо иллюстрирует мою мысль. Как правило, в реальных приложениях ожидание завершения запроса ввода-вывода реализовано в виде цикла. После завершения запроса поток выполняет нужное действие, ставит в очередь следующий асинхронный запрос ввода-вывода и снова крутится в цикле, пока не будет обработан следующий запрос.

GetOverlappedResult

Как я уже говорил, изначально Майкрософт не собиралась документировать поля *Internal* и *InternalHigh* структуры OVERLAPPED. Следовательно, нужно было как-то иначе сообщить разработчику, сколько байтов было передано в ходе операции ввода-вывода и передать ему код ошибки. Для этого Майкрософт предусмотрела функцию *GetOverlappedResult*:

```
BOOL GetOverlappedResult(
    HANDLE      hFile,
    OVERLAPPED* pOverlapped,
    PDWORD      pdwNumBytes,
    BOOL        bWalt);
```

Теперь поля *Internal* и *InternalHigh* задокументированы, поэтому проку от *GetOverlappedResult* немного. Однако во время знакомства с асинхронным вводом-выводом я решил выяснить внутреннее устройство этой функции, чтобы глубже изучить концепции, лежащие в основе этого механизма. Вот как выглядит внутренняя реализация функции *GetOverlappedResult*.

```
BOOL GetOverlappedResult(
    HANDLE hFile,
    OVERLAPPED* po,
    PDWORD pdwNumBytes,
    BOOL bWait) {

    if (po->Internal == STATUS_PENDING) {
        DWORD dwWaitRet = WAIT_TIME0UT;
        if (bWait) {
            // Ожидаем завершения ввода-вывода
                dwWaitRet = WaitForSingleObject(
                (po->hEvent != NULL) ? po->hEvent : hFile, INFINITE);
        }

        if (dwWaitRet == WAIT_TIMEOUT) {
            // Ввод-вывод не завершен и его ожидание не планируется
            SetLastError(ERR0R_I0_INC0MPLETE);
            return(FALSE);
        }
```

```
        if (dwWaitRet != WAIT_OBJECT_0) {
            // ошибка при вызове WaitForSingleObject
            return(FALSE);
        }
    }
    // Ввод-вывод завершен, возвращаем число переданных байтов
    pdwNumBytes = po->InternalHigh;

    if (SUCCEEDED(po->Internal)) {
        return(TRUE);    // Ввод-вывод выполнен без ошибок
    }

    // Устанавливаем последнюю ошибку согласно ошибке ввода-вывода
    SetLastError(po->Internal);
    return(FALSE);
}
```

Ввод-вывод с оповещением

Третий метод получения уведомлений о завершении ввода-вывода называется *ввод-вывод с оповещением* (alertable I/O). Майкрософт продвигала его как самый лучший механизм для создания высокопроизводительных масштабируемых приложений. Однако, начав использовать ввод-вывод с оповещением, разработчики вскоре поняли, что возможности этого механизма не оправдывают их ожиданий.

Мне пришлось довольно много применять ввод-вывод с оповещением, поэтому скажу вам сразу: этот механизм ужасен и лучше всячески избегать его. Однако для обеспечения его работоспособности Майкрософт добавила в свои операционные системы чрезвычайно полезную инфраструктуру с ценными возможностями. Так что, читая этот раздел, не старайтесь досконально разобраться в аспектах, связанных с вводом-выводом, а обращайте больше внимания на вспомогательную инфраструктуру.

Вместе с потоком система всегда создает очередь, называемая *очередью вызовов асинхронных процедур* (asynchronous procedure call, APC) или APC-очередью потока. При генерации запроса ввода-вывода можно приказать драйверу устройства добавить элемент в APC-очередь вызывающего потока. Чтобы уведомления о завершении ввода-вывода помещались в APC-очередь потока, следует вызвать функцию *ReadFileEx* или *WriteFileEx*:

```
BOOL ReadFileEx(
    HANDLE       hFile,
    PVOID        pvBuffer,
    DWORD        nNumBytesToRead,
    OVERLAPPED*  pOverlapped,
    LPOVERLAPPED_C0MPLETI0N_R0UTINE pfnCompletionRoutine);
```

```
BOOL WriteFileEx(
    HANDLE       hFile,
    CONST VOID   *pvBuffer,
    DWORD        nNumBytesToWrite,
    OVERLAPPED*  pOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine);
```

Подобно *ReadFile* и *WriteFile*, функции *ReadFileEx* и *WriteFileEx* отправляют запросы ввода-вывода драйверам устройств и немедленно возвращают управление. Функции *ReadFileEx* и *WriteFileEx* по набору параметров не отличаются от *ReadFile* с *WriteFile* за исключением двух моментов. Во-первых, **Ex*-функции не принимают указатель на DWORD-значение, в которое записывается число переданных байтов, получить эту информацию можно лишь с помощью функции обратного вызова. Во-вторых, **Ex*-функциям необходимо передавать адрес функции обратного вызова, называемой также *процедурой завершения* (completion routine). Прототип этой функции должен иметь следующий вид:

```
VOID WINAPI CompletlonRoutine(
    DWORD        dwError,
    DWORD        dwNumBytes,
    OVERLAPPED* po);
```

Генерируя асинхронный запрос ввода-вывода, функции *ReadFileEx* или *WriteFileEx* передают адрес функции обратного вызова драйверу устройства. Завершив обработку этого запроса, драйвер добавляет в APC-очередь вызывающего потока новый элемент. Этот элемент содержит адреса функции обратного вызова (процедуры завершения) и структуры OVERLAPPED, с использованием которой был инициирован запрос ввода-вывода.

Примечание Между прочим, при завершении ввода-вывода с оповещением, драйвер не пытается освободить объект-событие. В действительности устройство даже не ссылается на элемент *hEvent* структуры OVERLAPPED. Поэтому при желании можете использовать элемент *hEvent* для собственных нужд.

Когда поток находится в *тревожном состоянии* (alertable state), о котором будет рассказано чуть ниже, система анализирует его APC-очередь и для каждого ее элемента вызывает соответствующую процедуру завершения, передавая ей код ошибки ввода-вывода, число переданных байтов и адрес структуры OVERLAPPED. Заметьте, что код ошибки и число переданных байтов также хранятся в полях *Internal* и *InternalHigh* структуры OVERLAPPED (как сказано выше, причина этого — в первоначальном нежелании Майкрософт документировать эти поля).

Чуть позже мы вернемся к процедурам завершения, а пока рассмотрим, как система обрабатывает асинхронные запросы ввода-вывода. Следующий код ставит в очередь три асинхронных операции:

```
hFile = CreateFile(…, FILE_FLAG_OVERLAPPED, …);

ReadFileEx(hFile, …);                    // первый вызов ReadFileEx
WriteFileEx(hFile, …);                   // первый вызов WriteFileEx
ReadFileEx(hFile, …);                    // второй вызов ReadFileEx

SomeFunc();
```

Исполнение вызова *SomeFunc* занимает некоторое время, поскольку при этом система должна выполнить три операции. Пока поток исполняет функцию *SomeFunc*, драйвер устройства добавляет в его APC-очередь элементы, соответствующие завершенным операциям ввода-вывода. Вот как может выглядеть APC-очередь:

```
first WriteFileEx completed
second ReadFileEx completed
first ReadFileEx completed
```

Система автоматически поддерживает APC-очередь. Из этого примера также видно, что система может исполнять поставленные в очередь запросы в любом порядке: запросы, добавленные первыми, могут быть исполнены в последнюю очередь, и наоборот. Каждый элемент APC-очереди потока содержит адрес и параметры функции обратного вызова.

Выполненные запросы ввода-вывода просто добавляются в APC-очередь потока, то есть процедура завершения вызывается не сразу, поскольку поток может быть занят исполнение какой-нибудь важной операции, прервать которую невозможно. Для обработки элементов APC-очереди поток должен перевести себя в тревожное состояние. Это всего лишь означает, что исполнение потока достигло точки, в которой оно может быть прервано. Windows поддерживает шесть функций, способных перевести поток в тревожное состояние:

```
DWORD SleepEx(
    DWORD  dwMilliseconds,
    BOOL   bAlertable);

DWORD WaitForSingleObjectEx(
    HANDLE hObject,
    DWORD  dwMilliseconds,
    BOOL   bAlertable);

DWORD WaitForMultipleObjectsEx(
    DWORD  cObjects,
    CONST HANDLE* phObjects,
    BOOL   bWaitAll,
    DWORD  dwMilliseconds,
    BOOL   bAlertable);
```

```
BOOL SignalObjectAndWait(
    HANDLE hObjectToSignal,
    HANDLE hObjectToWaitOn,
    DWORD  dwMilliseconds,
    BOOL   bAlertable);

BOOL GetQueuedCompletionStatusEx(
    HANDLE hCompPort,
    LPOVERLAPPED_ENTRY pCompPo rtEnt ries,
    ULONG  ulCount,
    PULONG pulNumEntriesRemoved,
    DWORD  dwMilliseconds,
    BOOL   bAlertable);

DWORD MsgWaitForMultipleObjectsEx(
    DWORD  nCount,
    CONST HANDLE* pHandles,
    DWORD  dwMilliseconds,
    DWORD  dwWakeMask,
    DWORD  dwFlags);
```

Последним аргументом первых пяти функций является булево значение, которое указывает, должен ли поток перейти в тревожное состояние. Функции *MsgWaitForMultipleObjectsEx* следует передавать флаг MWMO_ALERTABLE, чтобы перевести поток в тревожное состояние. Знакомые с функциями *Sleep*, *WaitForSingleObject* и *WaitForMultipleObjects* не удивятся, узнав, что функции без суффикса *Ex* вызывают расширенные версии этих функций (с суффиксом *Ex* в именах), всегда передавая FALSE в параметре *bAlertable*.

После вызова одной из вышеперечисленных функций и перехода потока в тревожное состояние система прежде всего проверяет APC-очередь этого потока. Если в ней есть хотя бы один элемент, поток не засыпает. Вместо этого система извлекает элемент из APC-очереди и поток вызывает процедуру завершения, передавая ей код ошибки завершенной операции ввода-вывода, число переданных байтов и адрес структуры OVERLAPPED. Кода эта процедура возвращает управление, система ищет следующий элемент в APC-очереди. Если в очереди еще есть элементы, они обрабатываются, в противном случае функция, вызов которой перевел поток в тревожное состояние, возвращает управление. Учтите, что если APC-очередь потока окажется непустой в момент вызова функций, которые переводят поток в тревожное состояние, поток так никогда и не «заснет»!

Эти функции приостановят исполнение потока, только если в момент их вызова APC-очередь этого потока окажется пустой. Спящий (приостановленный) поток пробуждается при освобождении рбъекта (или объектов) ядра, которых он ждал, либо при добавлении элемента в его APC-очередь.

Поскольку при этом поток находится в тревожном состоянии, при появлении в APC-очереди новых элементов система тут же пробуждает поток и обрабатывает их, вызывая соответствующие процедуры завершения. Эти функции немедленно возвращают управление, так что поток не засыпает в ожидании освобождения необходимого ему объекта ядра.

Результат вызова вышеописанных функций можно узнать по значению, которое они возвращают. Если такая функция (либо *GetLastError*) вернула WAIT_IO_COMPLETION, знайте, что поток занят обработкой своей APC-очереди. Если причина завершения другая, поток пробудился, потому что истек его период ожидания, освободился нужный этому потоку объект ядра, либо другой поток отказался от мьютекса.

Плюсы и минусы ввода-вывода с оповещением

Итак, мы разобрались, как работает ввод-вывод с оповещением, а теперь я расскажу, почему я так не люблю этот механизм.

- Функции обратного вызова

  Для работы ввода-вывода с оповещением необходимо создавать функции обратного вызова, что сильно затрудняет написание кода. Как правило, контекстной информации, предоставляемой этими функциями, слишком мало для эффективной диагностики, поэтом в итоге все равно приходится хранить много сведений в глобальных переменных. К счастью, эти глобальные переменные не приходится синхронизировать, поскольку функцию для перехода в тревожное состояние и функцию обратного вызова исполняет один и тот же поток. Поскольку один поток не может исполнить сразу две функции, эти переменные вне опасности.

- Сложности с масштабируемостью

  Самая большая проблема ввода-вывода с оповещением состоит в следующем: уведомление о завершении запроса ввода-вывода должен обработать тот же самый поток, который инициировал этот запрос. Если поток генерирует несколько запросов, тот же самый поток должен обработать все уведомления о завершении этих запросов, даже если все остальные потоки в это время бездействуют. Поскольку балансировка нагрузки в этом случае не поддерживается, использующие этот механизм приложения плохо масштабируются.

Обе проблемы весьма серьезны, поэтому я настоятельно рекомендую избегать применения ввода-вывода с оповещением на устройствах. Думаю, вы уже догадываетесь, что механизм, о котором пойдет речь в следующем разделе, — порты завершения ввода-вывода — решает обе эти проблемы. Но я обещал рассказать о достоинствах инфраструктуры для ввода-вывода с оповещением, поэтому сначала я выполню свое обещание.

Windows поддерживает функцию, которая позволяет вручную добавить элемент в APC-очередь потока:

```
DWORD QueueUserAPC(
    PAPCFUNC  pfnAPC,
    HANDLE    hThread,
    ULONG_PTR dwData);
```

Первый ее параметр — указатель на APC-функцию с прототипом следующего вида:

```
VOID WINAPI APCFunc(ULONG_PTR dwParam);
```

Второй параметр — описатель потока, в очередь которого нужно добавить элемент. Это может быть любой из потоков в системе. Если *hThread* идентифицирует поток, принадлежащий другому процессу, в *pfnAPC* дожжен быть адрес функции в адресном пространстве соответствующего процесса. Последний параметр функции *QueueUserAPC, dwData,* — значение, которое передается функции обратного вызова.

Хотя прототип *QueueUserAPC* объявлен так, что эта функция должна возвращать DWORD, на самом деле она возвращает значение типа BOOL, свидетельствующее об успешном либо неудачном завершении вызова. G помощью функции *QueueUserAPC* позволяет наладить чрезвычайно эффективное взаимодействие между потоками, даже принадлежащими разным процессам. К сожалению, так допускается передавать одно-единственное значение.

QueueUserAPC также позволяет принудительно вывести поток из ожидания. Предположим, ваш поток вызвал *WaitForSingleObject* и заснул в ожидании освобождения объекта ядра. В это время пользователь отдал команду на завершение приложения. Вы знаете, что поток должен корректно уничтожить себя, но как заставить пробудить спящий поток, чтобы тот совершил «самоубийство»? Ответ — с помощью функции *QueueUserAPC*.

Следующий код демонстрирует, как принудительно пробудить поток, чтобы тот корректно завершил свою работу. Главная функция потока порождает новый поток и передает ему описатель какого-нибудь объекта ядра. Пока вторичный поток работает, работает и первичный поток. Далее вторичный поток (исполняющий функцию *ThreadFunc*) вызывает функцию *WaitForSingleObjectEx*, которая приостанавливает поток и переводит его в тревожное состояние. А теперь предположим, что пользователь приказывает первичного потоку завершить приложение. Естественно, первичный поток может завершиться и тогда система просто «убьет» весь процесс. Однако это не очень «чистое» решение, и зачастую требуется прервать отдельную операцию, сохранив процесс.

Итак, первичный поток вызывает функцию *QueueUserAPC*, добавляющую элемент в APC-очередь потока. Поскольку вторичный поток находится в тревожном состоянии, он просыпается и опустошает свою APC-очередь вызовом функции *APCFunc*. Эта функция не делает абсолютно ничего и просто возвращает управление. Поскольку APC-очередь теперь пуста, вызов *WaitForSingleObjectEx* возвращает управление потоку с кодом WAIT_IO_

COMPLETION. Функция *ThreadFunc* ждет именно этого значения, поскольку оно говорит о том, что поток был разбужен для корректного завершения.

```
// это функция обратного вызова APC, которая ничего не делает
VOID WINAPI APCFunc(ULONG_PTR dwParam) {
    // здесь нет никаких действий
}

UINT WINAPI ThreadFunc(PVOID pvParam) {
    HANDLE hEvent = (HANDLE) pvParam;    // описатель передается потоку

    // тревожное ожидание, из которого поток нужно вывести для корректного заверше-
       ния
    DWORD dw = WaitForSingleObjectEx(hEvent, INFINITE, TRUE);
    if (dw == WAIT_OBJECT_0) {
        // объект освободился
    }
    if (dw == WAIT_I0_C0MPLETI0N) {
        // вызов QueueUserAPC принудительно вывел поток из ожидания
        return(0);                          // поток завершается корректно
    }
    …
    return(0);
}

void main() {
    HANDLE hEvent = CreateEvent(…);
    HANDLE hThread = (HANDLE) _beginthreadex(NULL, 0,
        ThreadFunc, (PVOID) hEvent, 0, NULL);
    …

    // корректное принудительное завершение вторичного потока
    QueueUserAPC(APCFunc, hThread, NULL);
    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hTh read);
    CloseHandle(hEvent);
}
```

Некоторые могут подумать, что проблему можно решить, заменив *WaitForSingleObjeciEx* на *WaitForMultipleObjects* и создав объект ядра «событие», освободив которое, можно заставить вторичный поток корректно завершиться. В моем простом примере это сработает, но если вторичный поток вызовет *WaitForMultipleObjects* так, чтобы ждать освобождения всех объектов, использованием *QueueUserAPC* будет единственным решением для принудительного вывода потока из ожидания.

Порты завершения ввода-вывода

Windows создавалась как защищенная и надежная операционная система, в которой смогут работать приложения, обслуживающие тысячи пользователей. Традиционно разработчикам были доступны две модели построения приложений-служб:

- Модель на основе последовательной обработки
  Используется единственный поток, ожидающий поступления запросов от клиентов (обычно по сети). Получив запрос, поток пробуждается и обрабатывает его.
- Модель на основе параллельной обработки
  В этой модели используется несколько потоков. Один из потоков ожидает запросы от клиентов. Получив запрос, он порождает новый поток, обрабатывающий принятый запрос. В это время первый поток крутится в цикле в ожидании следующих клиентских запросов. Закончив обработку запроса, второй поток завершается.

Проблема с моделью, основанной на последовательной обработке, состоит в том, что она плохо справляется с обработкой множества одновременных запросов. Клиентские запросы, поступающие одновременно, могут быть обработаны только по очереди, друг за другом. Службы, спроектированные с использованием этой модели, неспособны использовать преимущества многопроцессорных компьютеров. Ясно, что модель, основанная на последовательной обработке, годится только для простейших серверных приложений, которым приходится обрабатывать мало клиентских запросов, которые не создают большой нагрузки. Типичным примером сервера, обрабатывающего запросы последовательно, может служить сервер Ping.

Ограничения последовательной модели обусловили чрезвычайно высокую популярность модели, основанной на параллельной обработке. Эта модель предусматривает создание отдельного потока для обработки каждого из клиентских запросов. Ее преимуществом является очень низкая загруженность потока, принимающего входящие запросы: большую часть времени он просто спит. При поступлении клиентского запроса этот поток пробуждается, порождает новый поток для обработки запроса и возвращается к ожиданию следующих запросов. Это означает, что клиентские запросы обрабатываются независимо друг от друга. Кроме того, поскольку для каждого запроса создается отдельный поток, такие серверные приложения прекрасно масштабируются и эффективно работают на многопроцессорных машинах. Таким образом, установка дополнительных процессоров позволяет повысить производительность серверных приложений, построенных с использованием параллельной модели.

Для Windows были разработаны службы, использовавшие параллельную обработку. Однако их производительность оказалась меньше желаемой. В частности, разработчики Windows заметили, что при одновременной обработке множества запросов в системе появляется множество потоков.

Поскольку все эти потоки являются готовыми к выполнению (т.е. они не приостановлены и не ожидают какого-либо события), стало ясно, что ядро Windows тратит слишком много времени на переключение контекста, поэтому потокам достается слишком мало процессорного времени для выполнения полезной работы. Чтобы сделать среду Windows более приспособленной для серверных приложений, Майкрософт должна была решить эту проблему. В результате появился объект ядра «порт завершения ввода-вывода».

Создание портов завершения ввода-вывода

Принцип портов завершения ввода-вывода заключается в ограничении числа одновременно исполняемых потоков. Так, невозможно создать 500 готовых к выполнению потоков для обработки 500 запросов. В этой связи возникает вопрос: а каково оптимальное число одновременно работающих потоков? Если вдуматься, вряд ли есть смысл иметь больше одного готового к выполнению потока в расчете на один процессор. Ведь как только число потоков становится больше числа процессоров, системе приходится тратить драгоценное процессорное время на переключение контекста потоков — в этом состоит один из недостатков модели, основанной на параллельной обработке.

К недостаткам этой модели относится и необходимость создания нового потока для каждого клиентского запроса. Создание потока обходится намного дешевле, чем целого процесса с его виртуальным адресным пространством, но оно, все же, далеко не бесплатно. Можно повысить производительность службы, если во время ее инициализации создать пул потоков, который будет доступен, пока служба работает. Порты завершения ввода-вывода предназначены для использования вместе с пулами потоков.

Порт завершения ввода-вывода является, наверное, самым сложным объектом ядра. Чтобы создать его, следует вызвать функцию *CreateIoCompletionPort*.

```
HANDLE CreateIoCompletionPort(
    HANDLE      hFile,
    HANDLE      hExistingCompletionPort,
    UL0NG_PTR CompletionKey,
    DWORD       dwNumbe rOfConcu r rentTh reads);
```

Эта функция выполняет две операции: создает порт завершения ввода-вывода и связывает с ним устройство. По-моему, эта функция слишком сложна и лучше бы Майкрософт разбила ее на две функции. Когда я работают с портами завершения ввода-вывода, я разделяю эти операции путем создания пары крошечных функций, абстрагирующих вызовы *CreateIoCompletionPort*. Первая функция, *CreateNewCompletionPort*, выглядит так:

```
HANDLE CreateNewCompletionPort(DWORD dwNumberOfConcurrentThreads) {
```

```
    return(CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0,
        dwNumberOfConcurrentThreads));
}
```

Она принимает единственный аргумент, *dwNumberOfCmmnentThreads,* и вызывает Windows-функцию *CreateIoCompletionPort* с параметрами, из которых первые три «зашиты» в коде, а четвертый является значением *dwNumberOfConcurrentThreads*. Как видите, первые три параметра *CreateIoCompletionPort* используются только для связывания устройства с портом завершения ввода-вывода (об этом — чуть ниже). Чтобы просто создать порт завершения ввода-вывода, я передаю функции *CreateIoCompletionPort* в первых трёх параметрах значения INVALID_HANDLE_VALUE, NULL и 0, соответственно.

Параметр *dwNumberOfConcurrentThreads* определяет максимальное число планируемых потоков для данного порта завершения ввода-вывода. Если установить этот параметр в 0, то по умолчанию число готовых к выполнению потоков для этого порта будет равно числу процессоров, установленных в системе. Как правило, это то, что надо, чтобы свести к минимуму лишние переключения контекста. Имеет смысл увеличить это значение, если обработка клиентских запросов требует интенсивных вычислений и редко приводит к блокированию потоков, но лучше этого не делать. Попробуйте поэкспериментировать с параметром *dwNumberOfConcurrentThreads,* чтобы узнать, как его значение влияет на производительность вашего приложения на имеющемся оборудовании.

Вероятно, вы заметили, что *CreateIoCompletionPort* — одна из немногих Windows-функций для создания объектов ядра, но не принимающих адрес структуры SECURITY_ATTRIBUTES. Дело в том, что порты ввода-вывода предназначены для использования в пределах одного процесса, почему — станет ясно, когда я объясню, как их используют.

Связывание устройства с портом завершения ввода-вывода

При создании порта завершения ввода-вывода ядро в действительности создает целых пять различных структур данных, как показано на рис. 10-1; продолжая чтение раздела, держите этот рисунок перед глазами.

Первая структура данных представляет собой список устройств, связанных с данным портом. Для связывания устройства с портом ввода-вывода необходимо вызвать функцию *CreateIoCompletionPort.* Для этого я тоже написал собственную функцию:

```
BOOL AssociateDeviceWithCompletionPort(
    HANDLE hCompletionPort, HANDLE hDevice, DWORD dwCompletionKey) {

    HANDLE h = CreateIoCompletionPort(hDevice, hCompletionPort,
        dwCompletionKey, 0);
    return(h == hCompletionPort);
}
```

Функция *AssociateDeviceWithCompletionPort* добавляет элемент в список устройств существующего порта завершения ввода-вывода. Она принимает описатель существующего порта (его возвращает вызов *CreateNewCompletionPort*), описатель устройства (файла, сокета, почтового ящика, канал и пр.) и ключ завершения (это значение, которое имеет смысл для разработчика, системе же безразлично, что передается в этом параметре). Каждый раз при связывании устройства с портом система добавляет эти сведения в список устройств порта завершения ввода-вывода.

> **Примечание.** Функция *CreateIoCompletionPort* слишком сложна, поэтому я рекомендую логически разделить две ее возможности. Однако у столь сложной функции есть одно преимущество: она позволяет в один прием создать порт завершения ввода-вывода и связать с ним устройство. Например, следующий код открывает файл и создает связанный с ним порт завершения ввода-вывода. У всех адресованных этому файлу запросов ввода-вывода будет ключ завершения CK_FILE, при этом возможно одновременное исполнение до двух потоков.

```
#define CK_FILE    1
HANDLE hFile = CreateFile(…);
HANDLE hCompletionPort = CreateIoCompletionPort(hFile, NULL, CK_FILE, 2);
```

Вторая структура данных — очередь завершения ввода-вывода. После завершения асинхронного запроса ввода-вывода на устройстве система проверяет, не связано ли это устройство с портом завершения ввода-вывода. Если это так, система добавляет в очередь завершения ввода-вывода этого порта элемент, представляющий обработанный запрос. В каждом элементе этой очереди содержатся сведения о числе переданных байтов, значение ключа завершения, заданное при связывании устройства с портом, указатель на структуру OVERLAPPED запроса и код ошибки. Об удалении элементов из этой очереди я расскажу чуть позже.

> **Примечание.** Возможна генерация запросов ввода-вывода на устройстве без постановки элементов в очередь порта завершения ввода-вывода. Обычно это не требуется, но иногда удобно отказаться от уведомлений, например при пересылке данных через сокеты.

Чтобы сгенерировать запрос ввода-вывода без постановки элемента в очередь завершения, необходимо записать в элемент *hEvent* структуры OVERLAPPED допустимый описатель события, а затем обработать побитовой операцией OR этот описатель и 1.

```
Overlapped.hEvent * CreateEvent(NULL, TRUE, FALSE, NULL);
Overlapped.hEvent = (HANDLE) ((DWORD_PTR) Overlapped.hEvent | 1);
ReadFile(…, &Overlapped);
```

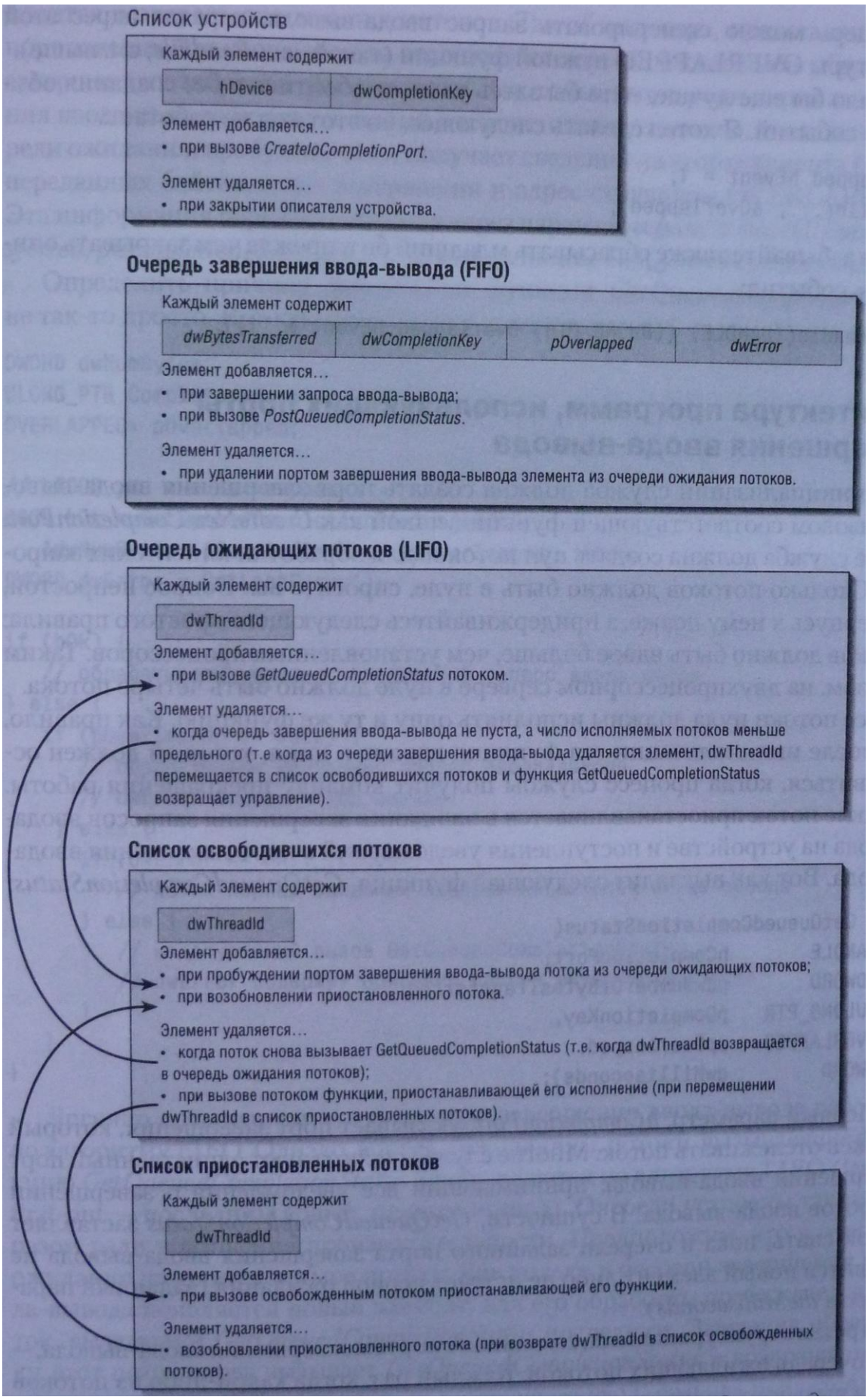


Рис. 10-1. *Внутреннее устройство порта завершения ввода-вывода*

Теперь можно сгенерировать запрос ввода-вывода, передав адрес этой структуры OVERLAPPED нужной функции (такой, как *ReadFile;* см. выше).

Было бы еще лучше, если бы здесь удалось обойтись и без создания объектов-событий. Я хотел сделать следующее, но этот код не работает:

```
Overlapped.hEvent = 1;
ReadFile(…, &Overlapped);
```

Не забывайте также сбрасывать младший бит, прежде чем закрывать описатель события:

```
CloseHandle((HANDLE) ((DWORD.PTR) Overlapped.hEvent & "1));
```

Архитектура программ, использующих порты завершения ввода-вывода

При инициализации служба должна создать порт завершения ввода-вывода вызовом соответствующей функции, такой как *CreateNewCompletionPort*. Далее служба должна создать пул потоков для обработки клиентских запросов. Сколько потоков должно быть в пуле, спросите вы. Вопрос непростой, и я вернусь к нему позже, а придерживайтесь следующего простого правила: потоков должно быть вдвое больше, чем установленных процессоров. Таким образом, на двухпроцессорном сервере в пуле должно быть четыре потока.

Все потоки пула должны исполнять одну и ту же функцию. Как правило, эта после инициализации эта функция входит в цикл, который должен остановиться, когда процесс службы получит команду прекращения работы. В цикле поток приостанавливается в ожидании завершения запросов ввода-вывода на устройстве и поступления уведомлений в порт завершения ввода-вывода. Вот как выглядит следующая функция, *GetQueuedCompletionStatus*:

```
BOOL GetQueuedCompletionStatus(
    HANDLE          hCompletionPort,
    PDWORD          pdwNumberOfBytesTransferred,
    PULONG_PTR      pCompletionKey,
    OVERLAPPED**    ppOverlapped,
    DWORD           dwMilliseconds);
```

Первый параметр, *hCompletionPort*, указывает порт завершения, который должен отслеживать поток. Многие службы используют единственный порт завершения ввода-вывода, принимающий все уведомления о завершении запросов ввода-вывода. В сущности, *GetQueuedCompletionStatus* заставляет поток спать, пока в очереди заданного порта завершения ввода-вывода не появится новый элемент либо не истечет период ожидания (заданный параметром *dwMilliseconds*).

Третья структура данных, связанная с портом завершения ввода-вывода, — это очередь ожидающих потоков. Каждый раз» когда какой-либо из потоков пула вызывает функцию *GetQueuedCompletionStatus*, его идентификатор за-

носится в очередь ожидающих потоков. Так объект ядра, представляющий порт завершения ввода-вывода отслеживает потоки, ожидающие обработки завершенных запросов ввода-вывода. Как только в очереди порта завершения ввода-вывода появится новый элемент, один из потоков, стоящих в очереди ожидания, пробуждается и получает сведения из этого элемента (число переданных байтов, ключ завершения и адрес структуры OVERLAPPED). Эта информация передается потоку через параметры *pdwNumberOfBytesTransferred*, *pCompletionKey* и *ppOverlapped* функции *GetQueuedCompletionStatus*.

Определить причину завершения функции *GetQueuedCompletionStatus* не так-то просто. Вот как это делается правильно:

```
DWORD dwNumBytes;
ULONG_PTR CompletionKey;
OVERLAPPED* pOverlapped;

// hIOCP уже инициализирована
BOOL bOk = GetQueuedCompletionStatus(hIOCP,
    &dwNumBytes, &CompletionKey, &pOverlapped, 1000);
DWORD dwError = GetLastError();

if (bOk) {
    // обрабатываем успешно завершенный запрос ввода-вывода
} else {
    if (pOverlapped != NULL) {
        // обрабатываем неудачный запрос ввода-вывода
        // dwError содержит код ошибки
    } else {
        if (dwError == WAJT_TIME0UT) {
            // истек период ожидания завершения запроса ввода-вывода
        } else {
            // недопустимый вызов GetQueuedCompletionStatus
            // dwError содержит описание причин неудачного вызова
        }
    }
}
```

Логично предположить, что очередь завершения ввода-вывода работает по алгоритму FIFO. Однако, вопреки ожиданиям, потоки, вызывающие функцию *GetQueuedCompletionStatus*, пробуждаются по алгоритму LIFO (last-in first-out — последним вошел, первым вышел). Очереди устроены таким образом ради повышения производительности. Предположим, что в очереди ожидания находится четыре потока. Как только в очереди завершения ввода-вывода появляется новый элемент, для его обработки пробуждается поток, вызвавший *GetQueuedCompletionStatus* последним. Завершив обработку, этот поток снова вызывает *GetQueuedCompletionStatus* и возвращается в очередь ожидания. Если в очереди завершения ввода-вывода появится еще один элемент, для его обработки проснется тот же самый поток.

Таким образом, если запросы ввода-вывода обрабатываются так медленно, что для их обработки достаточно единственного потока, система будет пробуждать один и тот же поток, а остальные три так и не проснутся. Благодаря использованию алгоритма LIFO, страницы памяти, занятые ресурсами (такими как стек) потоков, не получающих процессорное время, могут выкачиваться из оперативной памяти в страничный файл на жестком диске и сбрасываться из кэша процессора. Так что существование множества потоков, ожидающих поступления уведомлений в порт завершения ввода-вывода, не так уж сильно снижает быстродействие. Если ожидающих потоков много, а обработанных запросов ввода-вывода мало, большинство ресурсов простаивающих потоков, скорее всего, будет выгружено в страничный файл.

Если ожидается, что серверное приложение будет постоянно получать множество запросов ввода-вывода, в Windows Vista не обязательно плодить потоки, которые в основном ждут прихода уведомлений в порт завершения ввода-вывода и тратят системные ресурсы на переключения контекста. Вместо этого можно обрабатывать запросы группам, вызывая следующую функцию:

```
BOOL GetQueuedCompletionStatusEx(
    HANDLE hCompletionPort,
    LPOVERLAPPED_ENTRY pCompletionPortEntries,
    ULONG ulCount,
    PULONG pulNumEntriesRemoved,
    DWORD dwMilliseconds,
    BOOL bAlertable);
```

Первый параметр, *hCompletionPort*, указывает порт завершения ввода-вывода, который поток должен отслеживать. При вызове этой функции из очереди указанного порта извлекаются элементы, затем их содержимое копируется в массив, содержащийся в параметре *pCompletionPortEntries*. Параметр *ulCount* указывает число элементов этого массива, а в long-значение, указанное параметром *pulNumEntriesRemoved*, записывается, сколько запросов извлечено из очереди завершения ввода-вывода.

Элемент массива *pCompletionPortEntries* — это структура **OVERLAPPED_ENTRY**, в которой хранятся данные, составляющие элемент очереди завершения ввода-вывода: ключ завершения, адрес структуры OVERLAPPED, код ошибки запроса ввода-вывода и число переданных байтов.

```
typedef struct _OVERLAPPED_ENTRY {
    ULONG_PTR lpCompletionKey;
    LPOVERLAPPED lpOverlapped;
    ULONG_PTR Internal;
    DWORD dwNumberOfBytesTransferred;
} OVERLAPPED_ENTRY, *LPOVERLAPPED_ENTRY;
```

Поле *Internal* незадокументировано, поэтому не следует его использовать.

Последний параметр, *bAlertable,* устанавливается в FALSE. Функция ожидает постановки в очередь порта завершенных запросов ввода-вывода до истечения периода ожидания (заданного параметром *dwMilliseconds).* Если параметр *bAlertable* установлен в TRUE и в очереди нет завершенных запросов, поток переходит в тревожное состояние (об этом см. выше).

> **Примечание.** При генерации асинхронного запроса ввода-вывода на устройстве, связанном с портом завершения, Windows заносит результаты запроса в очередь этого порта. Windows поступает так, даже если асинхронный запрос был выполнен синхронно, дабы не нарушать логическое единство модели программирования. Однако за это приходится расплачиваться небольшим снижением производительности, поскольку информация о выполненном запросе должна быть передана в порт завершения ввода-вывода, а потому получена оттуда потоком.
>
> Можно немного повысить производительность, запретив Windows ставить в очередь порта завершения асинхронные запросы, выполненные синхронно. Для этого следует вызвать функцию *SetFileCompletionNotificationModes,* передав ей флаг FILE_SKIP_COMPLETION_PORT_ON_SUCCESS (см. выше). Разработчики, для которых производительность особенно важна, могут воспользоваться функцией *SetFileloOverlappedRange* (о ней можно прочитать в документации Platform SDK).

Как порт завершения ввода-вывода управляет пулом потоков

А теперь пришло время поговорить о достоинствах портов завершения ввода-вывода. При создании порта прежде всего необходимо указать предельное число одновременно работающих потоков. Как сказано выше, это число обычно равно числу процессоров компьютера, на котором предполагается использовать эту программу. После постановки в очередь выполненных запросов порт завершения ввода-вывода должен пробудить спящие потоки для их обработки. Однако число разбуженных потоков будет не больше, чем задано при создании порта. То есть, если в очередь будет поставлено четыре выполненных запроса ввода-вывода и четыре потока находятся в состоянии ожидания после вызова *GetQueuedCompletionStatus,* проснутся только два потока из четырех, остальные продолжат ожидание. Обработав выполненный запрос, поток снова вызывает *GetQueuedCompletionStatus.* Если система видит, что в очереди завершения ввода-вывода еще есть элементы, она пробуждает для их обработки те же самые потоки.

Если вдуматься в принцип работы этого механизма, он может показаться бессмысленным: порт завершения ввода-вывода разрешает одновременную работу ограниченного числа потоков, зачем тогда в пуле создаются дополнительные потоки, которые только и делают, что спят? Предположим для примера, что серверное приложение работает на компьютере с двумя про-

цессорами. Эта программа создает порт завершения ввода-вывода, разрешающий одновременную работу максимум двух потоков. Тем не менее, в пуле создается четыре потока (т.е. вдвое больше, чем процессоров). По-видимому, два из них так никогда и не проснутся для обработки данных.

Однако порты завершения ввода-вывода «умнее», чем вы думаете. Пробудив поток, порт завершения ввода-вывода заносит его идентификатор в четвертую структуру данных, составляющую этот порт, — список освобожденных потоков (см. рис. 10-1). Так порт отслеживает потоки, которые он пробуждает для обработки. Если освобожденный поток вызывает функцию, которая переводит его в состояние ожидания, это становится известно порту завершения, который переносит идентификатор потока из списка освобожденных потоков в список приостановленных потоков (это пятая и последняя из структур данных, составляющих порт завершения ввода-вывода).

Порт завершения ввода-вывода старается, чтобы число элементов списка освобожденных потоков было равно максимальному числу одновременно работающих потоков, указанному при создании порта. Если освобожденный поток по какой-либо причине переходит в состояние ожидания, число элементов списка освобожденных потоков уменьшается и порт пробуждает один из ожидающих потоков. Проснувшийся поток покидает список приостановленных и возвращается в список освобожденных потоков. В результате число элементов списка освобожденных потоков может превысить предельное число одновременно работающих потоков.

> **Примечание.** После вызова *GetQueuedCompletionStatus* поток «приписывается» к заданному порту завершения ввода-вывода. Система предполагает, что все приписанные к порту потоки будут обрабатывать его очередь. Однако число потоков пула, пробуждаемых портом для обработки, не больше предела, заданного при создании порта. Разорвать связь между потоком и портом завершения ввода-вывода можно:
> - завершив поток;
> - вызвав в потоке функцию *GetQueuedCompletionStatus*, передав ей описатель другого порта завершения ввода-вывода;
> - разрушив порт, к которому поток приписан в настоящее время.

А теперь подведем итоги. Допустим, что программа работает на двухпроцессорном компьютере, создает порт завершения ввода-вывода, допускающий одновременное пробуждение не более двух потоков, и пул с четырьмя потоками. При постановке в очередь трех выполненных запросов ввода-вывода в целях экономии времени на переключениях контекста порт пробудит только два потока. Если же один из активных потоков сгенерирует асинхронный запрос, вызовет *Sleep*, *WaitForSingleObject*, *WaitForMultipleObjects*, *SignalObjectAndWait* или другую функцию, которая выведет его из числа потоков, готовых к исполнению, порт заметит это и немедленно разбудит третий поток. Так порт завершения ввода-вывода выполняет свою задачу по предотвращении) простоя процессоров.

Рано или поздно первый поток вновь становится готовым к исполнению. При этом число готовых потоков становится больше числа процессоров в системе. Однако порт завершения ввода-вывода знает об этом и не позволит пробудить дополнительные потоки, пока число готовых потоков вновь не станет меньше числа процессоров. Порт завершения ввода-вывода спроектирован так, что число готовых к исполнению потоков недолго превышает предельное и быстро снижается по мере вызова потоками функции *Get Queued Completion Status*. Собственно, поэтому в пуле создается больше потоков, чем разрешено для одновременного исполнения портом завершения ввода-вывода.

Сколько потоков должно быть в пуле?

Сейчас уместно вернуться к вопросу о том, сколько потоков *должно* быть в пуле. При этом нужно учесть два момента. Во-первых, минимальное число потоков желательно создавать при инициализации службы, чтобы свести к минимуму затраты процессорного времени на создание и разрушение потоков. Во-вторых, необходимо ограничить число потоков для экономии системных ресурсов. Ресурсы простаивающих потоков выкачиваются из оперативной памяти в страничный файл, однако следует стремиться свести к минимуму бесполезную трату ресурсов, в том числе места в страничном файле.

Имеет смысл поэкспериментировать с числом потоков. Большинство служб (включая IIS) используют эвристические алгоритмы управления пулом потоков, я рекомендую вам тоже использовать подобные алгоритмы. Например, можно создать следующие переменные для управления пулом потоков:

```
LONG g_nThreadsMin;          // минимальное число потоков в пуле
LONG g_nThreadsMax;          // максимальное число потоков в пуле
LONG g_nThreadsCrnt;         // текущее число потоков в пуле
LONG g_nThreadsBusy;         // число занятых потоков в пуле
```

При инициализации вашего приложения можно создать столько потоков, сколько задано переменной *gjnThreadsMin*, все эти поток должны исполнять одну и ту же функцию. Ниже показан пример псевдокода функции потока из пула.

```
DWORD WINAPI ThreadPoolFunc(PVOID pv) {

    // добавляем поток в пул
    InterlockedIncrement(&g_nThreadsCrnt);
    InterlockedIncrement(&g_nThreadsBusy);

    for (BOOL bStaylnPool = TRUE; bStaylnPool;) {

        // приостанавливаем поток е ожидании выполненных запросов
        InterlockedDecrement(&m_nThreadsBusy);
        BOOL bOk = GetQueuedCompletionStatus(…);
```

```
        DWORD dwIOError = GetLastError();

        // поток занят
        int nThreadsBusy w InterlockedIncrement(&m_nThreadsBusy);

        // нужно ли добавить в пул еще один поток?
        if (nThreadsBusy == m_nThreadsCrnt) {      // все потоки заняты
            if (nThreadsBusy < m_nThreadsMax) {    // в пуле есть место для потока
                if (GetCPUUsage() < 75) {    // процессор загружен меньше, чем на 75X

                    // добавляем поток в пул
                    CloseHandle(chBEGINTHBEADEX(…));
                }
            }
        }

        if (!b0k && (dwIOError == WAIT_TIMEOUT)) {    // срок ожидания потока закон-
            чился
            // Сервер загружен слабо, следовательно этот поток можно завершить,
            // даже если не все запросы ввода-вывода обработана
            bStayInPool = FALSE;
        }

        if (bOk || (po != NULL)) {
            // пробуждаем поток и выполняем обработку
            …

            if (GetCPUUsage() > 90) {     // процессор загружен больше чем на 90X
                if (g_nThreadsCrnt > g_nThreadsMin)) {  // число потоков в пуле
                                                        // больше минимального
                    bStayInPool = FALSE;      // удаляем поток из пула
                }
            }
        }
    }

    // поток удаляется из пула
    InterlockedDecrement(&g_nThreadsBusy);
    InterlockedDecrement(&g_nThreadsCurrrent);
    return(0);
}
```

Этот пример демонстрирует возможности для творчества, которые открываются перед вами при использовании портов ввода-вывода. Функция *GetCPUUsage* не входит в Windows API, поэтому вы должны реализовать ее самостоятельно. Также необходимо следить, чтобы в пуле был хотя бы один поток, иначе клиенты не получат обслуживание. Используйте показанный

выше пример как образец при написании собственных служб. Однако чтобы добиться прироста производительности службы на ваших конкретных задачах, может потребоваться изменить его структуру.

> **Примечание.** Как сказано выше, при завершении потока система автоматически отменяет все необработанные запросы ввода-вывода, сгенерированные этим потоком. Действительно, в прежних версиях Windows (до Vista) поток, сгенерировавший запрос к устройству, связанному с портом завершения ввода-вывода, нельзя было завершить до исполнения этого запроса, иначе система отменяла все запросы этого потока. В Windows Vista поток может сгенерировать запросы ввода-вывода и спокойно завершиться, не дожидаясь его исполнения: его запросы будут исполнены и соответствующие уведомления будут направлены в порт завершения ввода-вывода.

У многих служб имеются утилиты, предоставляющие администраторам возможности по управлению пулом потоков. Они позволяют, например, устанавливать максимальное и минимальное число потоков, предельные значения загруженности процессора, а также максимальное число одновременно работающих потоков для порта завершения ввода-вывода.

Эмуляция выполненных запросов ввода-вывода

Вовсе не обязательно применять порты завершения ввода-вывода исключительно для работы с устройствами. Я уже говорил, что объект ядра «порт завершения ввода-вывода» — прекрасное средство для организации взаимодействия между потоками. Выше я показал функцию *QueueUserAPC*, которая позволяет потоку поставить APC-элемент в очередь другого потока. Порты завершения ввода-вывода поддерживают сходную функцию, *PostQueuedComptetionStatus*:

```
BOOL PostQueuedCompletionStatus(
    HANDLE      hCompletionPort,
    DWORD       dwNumBytes,
    ULONG_PTR   CompletionKey,
    OVERLAPPED* pOverlapped);
```

Эта функция ставит в очередь порта уведомление о завершении ввода-вывода. Первый ее параметр, *hCompletionPort*, определяет порт, в очередь которого следует добавить элемент. Остальные параметры, *dwNumBytes*, *CompletionKey* и *pOverlapped*, задают значения, возвращаемые вызовом *GetQueuedCompletionStatus*. Когда поток извлекает из очереди порта «эмулированный» элемент, функция *GetQueuedCompletionStatus* возвращает TRUE как при успешном исполнении запроса ввода-вывода.

Функция *PostQueuedCompletionStatus* невероятно полезное средство взаимодействия с потоками пула. Например, когда пользователь прекращает работу службы, все ее потоки должны корректно завершаться. Но потоки,

ждущие поступления уведомлений в порт ввода-вывода, не проснутся, пока не поступят новые запросы ввода-вывода. Вызвав *PostQueuedCompletionStatus* для каждого из потоков пула, можно пробудить спящие потоки и, если приложение прекращает работу (узнать об этом поможет функция *GetQueuedCompletionStatus*), заставить их освободить ресурсы и корректно завершиться.

Используйте только что показанный прием завершения работы потоков с осторожностью. Мой код работает, поскольку потоки пула завершаются и больше не вызывают *GetQueuedCompletionStatus*. Но если вам потребуется о чем-то уведомить потоки и заставить их снова вызывать *GetQueuedCompletionStatus*, вы столкнетесь с проблемой, поскольку потоки просыпаются по принципу LIFO. В таком случае придется воспользоваться дополнительными средствами синхронизации потоков, чтобы дать каждому потоку из пула обнаружить эмулированный элемент в очереди порта. Без этого потоки могут обнаруживать одни и те же уведомления по нескольку раз.

> **Примечание.** Если в Windows Vista вызвать функцию *CloseHandle* и передать ей описатель порта завершения ввода-вывода, все потоки, ожидающие после вызова *GetQueuedCompletionStatus*, проснутся и получат FALSE. При этом вызов *GetLastError* вернет ERROR_INVALID_HANDLE. Такое сообщение говорит потокам, что пришло время для их корректного завершения.

Программа-пример FileCopy

Показанная ниже программа FileCopy (10-FileCopy.exe) иллюстрирует использование портов завершения ввода-вывода. Исходный текст этой программы можно найти в каталоге 10-FileCopy внутри архива, доступного на веб-сайте поддержки этой книги. Эта просто копирует заданный пользователем файл в новый файл, FileCopy.cpy. При запуске FileCopy открывается окно, показанное на рис. **10-2.**

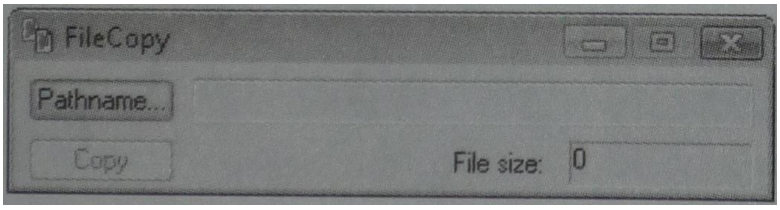


Рис. 10-2. *Окно программы FileCopy*

Пользователь щелкает кнопку и выбирает файл для копирования, после чего обновляются поля Pathname и File Size. По щелчку кнопки Copy программа вызывает функцию *FileCopy*, которая и выполняет основную работу. Познакомимся поближе с этой функцией.

При подготовке к копированию *FileCopy* открывает исходный файл и получает его размер в байтах. Я хотел, чтобы моя программ работала максимально быстро, поэтому файл открывается с флагом FILE_FLAG_NO_ BUFFERING. Так удается получить прямой доступ к файлу без дополни

тельных издержек, связанных с кэшированием, с помощью которого система «ускоряет» доступ к файлам. Естественно, выбрав прямой доступ к файлу, я взял на себя кое-какие дополнительные обязательства. Так, мне придется обращаться к файлу для чтения и записи только с использованием смещений, кратным размеру сектора дискового тома. Я решил передавать данные порциями по 64 Кб (это значение BUFFSIZE). Заметьте также, что исходный файл открывается с флагом FILE_FLAG_OVERLAPPED, что позволяет асинхронно исполнять адресованные ему запросы ввода-вывода.

Аналогичным образом открывается и целевой файл (с указанием флагов FILE_FLAG_NO_BUFFERING и FILE_FLAG_OVERLAPPED). При создании целевого файла я также передаю через параметр *hFileTemplate* функции *CreateFile* описатель исходного файла, чтобы новый файл получил атрибуты исходного.

> **Примечание.** После открытия обоих файлов для целевого файла сразу устанавливается максимальный размер (вызовами *SetFilePointerEx* и SetEndOfFile). Это очень важно, и вот почему: NTFS ставит особый указатель в месте завершения последней операции записи в файл. Попытка прочитать область за пределами этого указателя вернет нули. При записи в эту область система сначала заполнит область между старым и новым положениями указателя нулями, затем запишет в файл ваши данные и переставит указатель в новое положение. Это делается во избежание случайного доступа к старым данным, ранее записанным в этой области, согласно требованиям стандарта безопасности C2. Когда данные записываются рядом с концом файла, хранящегося в NTFS-разделе, система перемещает указатель записи. В этой ситуации NTFS приходится исполнять запросы ввода-вывода синхронно, даже если программа запрашивает асинхронный ввод-вывод. Так что если бы функция FileCopy не устанавливала бы размер целевого файла сразу, все асинхронные запросы исполнялись бы синхронно.

Когда нужные файлы уже открыты и подготовлены к обработке, *FileCopy* создает порт завершения ввода-вывода. Чтобы облегчить работу с портом, я написал маленький C++-класс, СЮСР. По сути, это простейшая оболочка для функций порта завершения ввода-вывода (см. файл IOCP.h и приложение A). Функция *FileCopy* создает порт завершения ввода-вывода путем создания экземпляра класса СЮСР (объекта с именем *iocp).*

Исходный и целевой файлы связываются с портом завершения ввода-вывода с помощью вызова функции *AssociateDevice* класса *СЮСР.* При связывании с портом каждое устройство получает ключ завершения. Исходному файлу назначается ключ CK_READ, свидетельствующий об успешном завершении чтения, а целевому файлу — CK_WRITE, поскольку в этот файл данные будут записываться. Теперь все готово для инициализации структур OVERLAPPED и буферов запросов ввода-вывода. Функция *FileCopy* устроена так, что число необработанных запросов ввода-вывода не превышает

четырех (согласно значению MAX_PENDING_IO_REQS). В ваших собственных приложениях можно реализовать динамическую подстройку этого значения. В программе *FileCopy* запросы ввода-вывода инкапсулированы в классе *CIOReq*. Это C++-класс, производный от структуры OVERLAPPED и содержащий дополнительную контекстную информацию. Функция *FileCopy* создает массив объектов *CIOReq* и вызывает метод *AllocBuffer*, который связывает буферы с объектами, представляющими запросы ввода-вывода. Буферы создаются с помощью функции *VirtualAlloc*, их размер определяется значением BUFFSIZE. Применение *VirtualAlloc* гарантирует, что размер выделенного блока памяти и его начальный адрес будет кратным размеру сектора тома, что необходимо при использовании флага FILE_FLAG_NO_BUFFERING.

Я начинаю генерацию запросов ввода-вывода с небольшой хитрости, отправляя в порт завершения ввода-вывода четыре уведомления об исполнении запросов с ключом CK_WRITE. В результате ожидающий порт поток считает, что завершилась операция записи, немедленно пробуждается и отправляет запрос на чтение исходного файла — вот тут-то и начинается настоящее копирование файла.

Главный цикл завершается после обработки всех запросов ввода-вывода. Если необработанные запросы еще остались, главный цикл совершает очередной оборот и поток пытается перейти в состояние ожидания порта завершения ввода-вывода, вызывая метод *GetStatus* класса CIOCP (который, в свою очередь, вызывает *GetQueuedCompletionStatus*). Таким образом» поток спит, пока в порт не придет уведомление о завершении запроса ввода-вывода. После возврата управления функцией *GetQueuedCompletionStatus* выполняется проверка ключа завершения (*CompletionKey*). Если CompletionKey = CK_READ, запрос чтения исходного файла выполнен, и я вызываю метод *Write* класса *CIOReq*, генерирующий запрос записи в целевой файл. Если же *CompletionKey* = CK_WRITE, то выполнен запрос записи. Если конец исходного файла еще не достигнут, я вызываю метод *Read* класса *CIOReq*, чтобы продолжить чтение.

Когда необработанные запросы ввода-вывода закончатся, цикл завершится. Далее программа выполняет очистку, закрывая описатели исходного и целевого файлов. Прежде, чем функция *FileCopy* вернет управление, нужно сделать еще кое-что, а именно исправить размер целевого файла, чтобы он стал равным размеру исходного файла. Для этого я еще раз открываю целевой файл, но на этот раз без флага FILE_FLAG_NO_BUFFERING. Это позволит мне оперировать порциями данных любого размера, а не только кратными размеру сектора. Следовательно, я смогу установить для целевого файла размер, в точности соответствующий размеру исходного файла.

```
/******************************************************************************
Module:  FileCopy.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"           // See Appendix A.
#include "..\CommonFiles\IOCompletionPort.h" // See Appendix A.
#include "..\CommonFiles\EnsureCleanup.h"    // See Appendix A.

#include <WindowsX.h>
#include "Resource.h"

// C RunTime Header Files
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>


//////////////////////////////////////////////////////////////////////////////


// Each I/O Request needs an OVERLAPPED structure and a data buffer
class CIOReq : public OVERLAPPED {
public:
   CIOReq() {
      Internal = InternalHigh = 0;
      Offset = OffsetHigh = 0;
      hEvent = NULL;
      m_nBuffSize = 0;
      m_pvData = NULL;
   }

   ~CIOReq() {
      if (m_pvData != NULL)
         VirtualFree(m_pvData, 0, MEM_RELEASE);
   }

   BOOL AllocBuffer(SIZE_T nBuffSize) {
      m_nBuffSize = nBuffSize;
      m_pvData = VirtualAlloc(NULL, m_nBuffSize, MEM_COMMIT, PAGE_READWRITE);
      return(m_pvData != NULL);
   }

   BOOL Read(HANDLE hDevice, PLARGE_INTEGER pliOffset = NULL) {
      if (pliOffset != NULL) {
         Offset     = pliOffset->LowPart;
         OffsetHigh = pliOffset->HighPart;
      }
      return(::ReadFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
   }

   BOOL Write(HANDLE hDevice, PLARGE_INTEGER pliOffset = NULL) {
      if (pliOffset != NULL) {
         Offset     = pliOffset->LowPart;
         OffsetHigh = pliOffset->HighPart;
      }
```

```
      return(::WriteFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
   }

private:
   SIZE_T m_nBuffSize;
   PVOID  m_pvData;
};


//////////////////////////////////////////////////////////////////////////////


#define BUFFSIZE              (64 * 1024) // The size of an I/O buffer
#define MAX_PENDING_IO_REQS   4           // The maximum # of I/Os


// The completion key values indicate the type of completed I/O.
#define CK_READ  1
#define CK_WRITE 2


//////////////////////////////////////////////////////////////////////////////


BOOL FileCopy(PCTSTR pszFileSrc, PCTSTR pszFileDst) {

   BOOL bOk = FALSE;    // Assume file copy fails
   LARGE_INTEGER liFileSizeSrc = { 0 }, liFileSizeDst;

   try {
      {
      // Open the source file without buffering & get its size
      CEnsureCloseFile hFileSrc = CreateFile(pszFileSrc, GENERIC_READ,
         FILE_SHARE_READ, NULL, OPEN_EXISTING,
         FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, NULL);
      if (hFileSrc.IsInvalid()) goto leave;

      // Get the file's size
      GetFileSizeEx(hFileSrc, &liFileSizeSrc);

      // Nonbuffered I/O requires sector-sized transfers.
      // I'll use buffer-size transfers since it's easier to calculate.
      liFileSizeDst.QuadPart = chROUNDUP(liFileSizeSrc.QuadPart, BUFFSIZE);

      // Open the destination file without buffering & set its size
      CEnsureCloseFile hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
         0, NULL, CREATE_ALWAYS,
```

```
         FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, hFileSrc);
      if (hFileDst.IsInvalid()) goto leave;

      // File systems extend files synchronously. Extend the destination file
      // now so that I/Os execute asynchronously improving performance.
      SetFilePointerEx(hFileDst, liFileSizeDst, NULL, FILE_BEGIN);
      SetEndOfFile(hFileDst);

      // Create an I/O completion port and associate the files with it.
      CIOCP iocp(0);
      iocp.AssociateDevice(hFileSrc, CK_READ);  // Read from source file
      iocp.AssociateDevice(hFileDst, CK_WRITE); // Write to destination file

      // Initialize record-keeping variables
      CIOReq ior[MAX_PENDING_IO_REQS];
      LARGE_INTEGER liNextReadOffset = { 0 };
      int nReadsInProgress  = 0;
      int nWritesInProgress = 0;

      // Prime the file copy engine by simulating that writes have completed.
      // This causes read operations to be issued.
      for (int nIOReq = 0; nIOReq < _countof(ior); nIOReq++) {

         // Each I/O request requires a data buffer for transfers
         chVERIFY(ior[nIOReq].AllocBuffer(BUFFSIZE));
         nWritesInProgress++;
         iocp.PostStatus(CK_WRITE, 0, &ior[nIOReq]);
      }

      BOOL bResult = FALSE;

      // Loop while outstanding I/O requests still exist
      while ((nReadsInProgress > 0) || (nWritesInProgress > 0)) {

         // Suspend the thread until an I/O completes
         ULONG_PTR CompletionKey;
         DWORD dwNumBytes;
         CIOReq* pior;
         bResult = iocp.GetStatus(&CompletionKey, &dwNumBytes, (OVERLAPPED**)
&pior, INFINITE);

         switch (CompletionKey) {
         case CK_READ:  // Read completed, write to destination
            nReadsInProgress--;
            bResult = pior->Write(hFileDst);  // Write to same offset read from
source
```

```
            nWritesInProgress++;
            break;

         case CK_WRITE: // Write completed, read from source
            nWritesInProgress--;
            if (liNextReadOffset.QuadPart < liFileSizeDst.QuadPart) {
               // Not EOF, read the next block of data from the source file.
               bResult = pior->Read(hFileSrc, &liNextReadOffset);
               nReadsInProgress++;
               liNextReadOffset.QuadPart += BUFFSIZE; // Advance source offset
            }
            break;
         }
      }
      bOk = TRUE;
      }
   leave:;
   }
   catch (...) {
   }

   if (bOk) {
      // The destination file size is a multiple of the page size. Open the
      // file WITH buffering to shrink its size to the source file's size.
      CEnsureCloseFile hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
         0, NULL, OPEN_EXISTING, 0, NULL);
      if (hFileDst.IsValid()) {

         SetFilePointerEx(hFileDst, liFileSizeSrc, NULL, FILE_BEGIN);
         SetEndOfFile(hFileDst);
      }
   }

   return(bOk);
}


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_FILECOPY);
```

```
   // Disable Copy button since no file is selected yet.
   EnableWindow(GetDlgItem(hWnd, IDOK), FALSE);
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   TCHAR szPathname[_MAX_PATH];

   switch (id) {
   case IDCANCEL:
      EndDialog(hWnd, id);
      break;

   case IDOK:
      // Copy the source file to the destination file.
      Static_GetText(GetDlgItem(hWnd, IDC_SRCFILE),
         szPathname, _countof(szPathname));
      SetCursor(LoadCursor(NULL, IDC_WAIT));
      chMB(FileCopy(szPathname, TEXT("FileCopy.cpy"))
         ? "File Copy Successful" : "File Copy Failed");
      break;

   case IDC_PATHNAME:
      OPENFILENAME ofn = { OPENFILENAME_SIZE_VERSION_400 };
      ofn.hwndOwner = hWnd;
      ofn.lpstrFilter = TEXT("*.*\0");
      lstrcpy(szPathname, TEXT("*.*"));
      ofn.lpstrFile = szPathname;
      ofn.nMaxFile = _countof(szPathname);
      ofn.lpstrTitle = TEXT("Select file to copy");
      ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST;
      BOOL bOk = GetOpenFileName(&ofn);
      if (bOk) {
         // Show user the source file's size
         Static_SetText(GetDlgItem(hWnd, IDC_SRCFILE), szPathname);
         CEnsureCloseFile hFile = CreateFile(szPathname, 0, 0, NULL,
            OPEN_EXISTING, 0, NULL);
         if (hFile.IsValid()) {
            LARGE_INTEGER liFileSize;
            GetFileSizeEx(hFile, &liFileSize);
            // NOTE: Only shows bottom 32 bits of size
            SetDlgItemInt(hWnd, IDC_SRCFILESIZE, liFileSize.LowPart, FALSE);
```

```
         }
      }
      EnableWindow(GetDlgItem(hWnd, IDOK), bOk);
      break;
   }
}


//////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
   chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
   chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_FILECOPY), NULL, Dlg_Proc);
   return(0);
}


//////////////////////////////// End of File ////////////////////////////////
```

Оглавление

ГЛАВА 11

Пулы потоков

В предыдущей главе мы обсудили организацию очереди запросов ввода-вывода с помощью объекта ядра «порт завершения ввода-вывода» и узнали, как рационально этот объект управляет потоками, обрабатывающими его очередь. Тем не менее, даже при использовании такого интеллектуального объекта за создание и уничтожение потоков отвечает разработчик.

Проблему того, как управлять созданием и уничтожением потоков, каждый решает по-своему. За прошедшие годы я создал несколько реализаций пулов потоков, рассчитанных на определенные сценарии. Однако в Windows поддерживаются пулы потоков (основанные на механизме порта завершения ввода-вывода), упрощающие создание, уничтожение и общий контроль за потоками. Конечно, эти функции носят общий характер и не годятся на все случаи жизни, но зачастую

их вполне достаточно, и они позволяют экономить массу времени при разработке многопоточного приложения.

Эти функции дают возможность вызывать другие функции:

- асинхронно;
- через определенные промежутки времени;
- при освобождении отдельных объектов ядра;
- при завершении запросов асинхронного ввода-вывода.

> **Примечание.** Первые API для управления пулами потоков появились в Windows 2000. В Windows Vista механизм пулов потоков был переработан, в результате появились новые API для управления пулами. Естественно Vista поддерживает прежние версии этих API для сохранения преемственной совместимости, однако в приложениях, предназначенных исключительно для Vista, рекомендуется использовать новые API В этой главе мы рассмотрим новые API-функции для управления пулами потоков в Windows Vista, информацию о прежних версиях API ищите в предыдущих изданиях этой книги.

При инициализации процесса никаких издержек, связанных с компонентами поддержки пула потоков, не возникает. Однако, как только вызывается одна из функций пула потоков, для процесса создается набор этих компонентов, и некоторые из них сохраняются до его завершения. Как видите, издержки от применения этих функций отнюдь не малые: частью вашего процесса становится целый набор потоков и внутренних структур данных. Так что, прежде чем пользоваться ими, тщательно взвесьте все «за» и «против».

Теперь, когда я вас предупредил, посмотрим, как все это работает.

Сценарий 1. Асинхронный вызов функций

Для асинхронного исполнения функций с использованием пула потоков статочно объявить функцию с прототипом следующего вида:

```
VOID NTAPI SimpleCallback(
    PTP_CALLBACK_INSTANCE plnstance,        // См.  раздел "Обработка завершения
                                            // обратного вызова"
    PVOID pvContext);
```

После этого можно запросить пул, чтобы один из его потоков исполнил вашу функцию. Для передачи запроса просто вызовите следующую функцию:

```
BOOL TrySubmitThreadpoolCallback(
    PTP_SIMPLE_CALLBACK pfnCallback,
    PVOID pvContext,
    PTP_CALLBACK_ENVIRON pcbe);   // См. раздел "Обработка завершения обрат вызова"
```

Эта функция добавляет рабочий элемент в очередь пула потоков (вызовом *PostQueuedCompletionStatus)* и возвращает TRUE, если операция завершится успешно, либо FALSE в противном случае. Функция *TrySubmitThreadpoolCallback* принимает несколько параметров. Параметр *pfnCallback* определяет вашу функцию, соответствующую прототипу SimpleCallback; параметр *pvContext* содержит аргументы вашей функции (значение ее параметра *pvContext),* а параметре PTPCALLBACKENVIRON можно передать просто NULL (подробнее об этом — в разделе о настройке пула потоков), параметр *pInstance* функции *SimpleCallback* я поясню позже (см. раздел «Обработка завершения обратного вызова»).

Заметьте: вам не требуется вызвать *CreateThread* самостоятельно. По умолчанию пул потоков создается для вашего процесса автоматически, и потоки пула вызывают заданную вами функцию обратного вызова. После обработки клиентского запроса поток не уничтожается, а возвращается в пул и обрабатывает другие элементы очереди пула. Таким образом, пул повторно использует одни и те же потоки вместо того, чтобы постоянно уничтожать старые и создавать новые. Это позволяет значительно повысить производительность приложений, поскольку создание и разрушение потоков отнимает много вре-

мени. Естественно, если пул обнаружит, что для вашего приложения лучше создать отдельный пул потоков он так и сделает. Кроме того, пул уничтожает свои лишние потоки, если таковые обнаружатся. Если вы не знаете наверняка, сколько потоков вам нужно, положитесь на внутренние алгоритмы пула потоков и позвольте ему автоматически автоматически подстраиваться под ваше приложение.

Явное управление рабочими элементами

В некоторых ситуациях, таких как нехватка памяти, вызов *TrySubmitThreadpoolCallback* может закончиться неудачей. Это неприемлемо, если требуется координированное исполнение ряда действий, например, при отмене некоторой операции по таймеру. Устанавливая таймер, вы надеетесь, что инициированная им операция отмены будет обработана одним из потоков пула. Однако к моменту срабатывания таймера свободная память может быть исчерпана, и вызов *TrySubmitThreadpoolCallback* завершится неудачей. В подобных случаях вы должны самостоятельно создать объект «рабочий элемент» с таймером, чтобы впоследствии явно поставить его в очередь пула потоков.

При каждом вызове *TrySubmitThreadpoolCallback* система создает для вас рабочий элемент. Если требуется поставить в очередь множество рабочих элементов, лучше создать рабочий элемент однократно, а затем поставить его в очередь нужное число раз — так удастся сэкономить память и повысить производительность. Рабочий элемент создают с помощью этой функции:

```
PTP_W0RK CreateThreadpoolWork(
    PTP_WORK_CALLBACK pfnWorkHandler,
    PVOID pvContext,
    PTP_CALLBACK_ENVIRON pcbe);            // см.  раздел "Настройка пула потоков"
```

Эта функция создает в памяти структуру пользовательского режима, в которой хранятся три параметра этой функции, и возвращает указатель на эту структуру. Параметр *pfnWorkHandler* — это указатель на функцию, которая будет в итоге вызвана потоком из пула при обработке рабочего элемента. Параметр *pvContext* может содержать любое значение, которое нужно передать функции обратного вызова. Имя функции, передаваемой через параметр *pfnWorkHandler*, должно соответствовать следующему прототипу:

```
VOID CALLBACK WorkCallback(
    PTP_CALLBACK_INSTANCE Instance,
    PVOID Context,
    PTP_W0RK Work);
```

Чтоб поставить запрос очередь пула потоков, вызовите функцию *SubmitThreadpoolWork*:

```
VOID SubmitThreadpoolWork(PTP_WORK pWork);
```

Теперь можно считать, что запрос поставлен в очередь успешно, и один из потоков пула исполнит функцию обратного вызова. Собственно, поэтому функция *SubmitThreadpoolWork* и возвращает значение типа VOID.

> **Внимание!** Если вы поставили в очередь несколько экземпляров одного и того же рабочего элемента, то указанная вами функция будет вызываться с одними и теми же параметрами, заданными значением *pvContext* при создании рабочего элемента. Имейте это в виду, пытаясь выполнить несколько операций с помощью одного рабочего элемента. Операции, требующие индивидуальной настройки, придется выполнять с помощью разных рабочих элементов.

Если другому потоку требуется отменить «чужой» рабочий элемент либо приостановить свою работу до то завершения обработки некоторого рабочего элемента, можно вызвать следующую функцию:

```
VOID WaitForThreadpoolWorkCallbacks(
    PTP_WORK pWork,
    BOOL bCancelPendingCallbacks);
```

Параметр *pWork* — это указатель на рабочий элемент, ранее созданный и поставленный в очередь вызовами функций *CreateThreadpoolWork* и *SubmitThreadpoolWork*. Если заданный рабочий элемент еще не поставлен в очередь, *WaitForThreadpoolWork Callbacks* тут же возвращает управление, не выполняя никаких действий.

Если передать TRUE в параметре *bCancelPendingCallbacks*, функция *WaitForThreadpoolWorkCallbacks* попытается отменить рабочие элементы, ранее поставленные в очередь. Обработка текущего рабочего элемента не прерывается, *WaitForThreadpoolWorkCallbacks* дождется ее завершения, и только потом вернет управление. Если рабочий элемент уже поставлен в очередь, но еще не обработан, он помечается как отмененный и функция *WaitForThreadpoolWorkCallbacks* возвращает управление. Получив помеченный таким образом рабочий элемент, поток знает, что делать обратный вызов не нужно, и обработка этого элемента даже не начнется.

Если же передать FALSE в параметре *bCancelPendingCallbacks*, *WaitForThread-poolWorkCallbacks* приостановит вызывающий поток до завершения обработки заданного рабочего элемента и возврата в пул выполнявшего ее потока.

> **Примечание.** Если в очередь поставлено несколько рабочих элементов с использованием одного и того же объекта PTP_WORK и при вызове *WaitForThreadpoolWorkCallbacks* передан параметр *bCancelPendingCallbacks* = FALSE, эта функция будет ждать завершения обработки всех таких рабочих элементов. Если же в этом параметре передано значение TRUE, *WaitForThreadpoolWorkCallbacks* дождется завершения обработки только текущих элементов.

Рабочие элементы, ставшие ненужными, следует освободить вызовом функции *CloseThreadpoolWork*, принимающей единственный параметр — указатель на рабочий элемент:

```
VOID CloseThreadpoolWork(PTP_WORK pwk);
```

Программа-пример Batch

Программа Batch(11-Batch.exe,см. листинг ниже), иллюстрирует применение функции рабочих элементов пула для реализации пакетной обработки. После выполнения каждой операции интерфейс программы получает уведомление — сообщение с префиксом в виде идентификатора потока, выполнившего эту операцию. В этой программе также реализовано простое решение, которое позволяет узнать о том, что все операции выполнены (см. рис. 11-1).

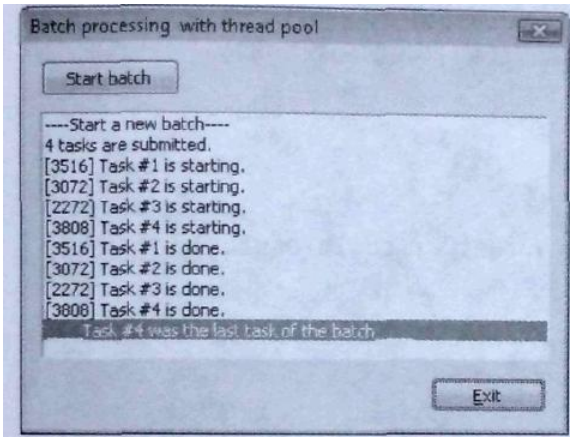


Рис. 11-1. *Вывод программы Batch*

Исходный код и файлы ресурсов для этой программы находятся в каталоге 11-Batch внутри архива, доступного на веб-сайте поддержки этой книги.

```
/******************************************************************************
Module:  Batch.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <Windowsx.h>
#include <WinBase.h>
#include <WinNT.h>

// C RunTime Header Files
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>
#include <strsafe.h>

#include "Batch.h"

///////////////////////////////////////////////////////////////////////////////

// Global variables
HWND     g_hDlg = NULL;
PTP_WORK g_pWorkItem = NULL;
volatile LONG g_nCurrentTask = 0;

// Global definitions
#define WM_APP_COMPLETED (WM_APP+123)

///////////////////////////////////////////////////////////////////////////////

void AddMessage(LPCTSTR szMsg) {
```

```
   HWND hListBox = GetDlgItem(g_hDlg, IDC_LB_STATUS);
   ListBox_SetCurSel(hListBox, ListBox_AddString(hListBox, szMsg));
}


//////////////////////////////////////////////////////////////////////////


void NTAPI TaskHandler(PTP_CALLBACK_INSTANCE Instance, PVOID Context, PTP_WORK
Work) {

   LONG currentTask = InterlockedIncrement(&g_nCurrentTask);

   TCHAR szMsg[MAX_PATH];
   StringCchPrintf(
      szMsg, _countof(szMsg),
      TEXT("[%u] Task #%u is starting."), GetCurrentThreadId(), currentTask);
   AddMessage(szMsg);

   // Simulate a lot of work
   Sleep(currentTask * 1000);

   StringCchPrintf(
      szMsg, _countof(szMsg),
      TEXT("[%u] Task #%u is done."), GetCurrentThreadId(), currentTask);
   AddMessage(szMsg);

   if (InterlockedDecrement(&g_nCurrentTask) == 0)
   {
      // Notify the UI thread for completion.
      PostMessage(g_hDlg, WM_APP_COMPLETED, 0, (LPARAM)currentTask);
   }
}


//////////////////////////////////////////////////////////////////////////


void OnStartBatch() {

   // Disable Start button
   Button_Enable(GetDlgItem(g_hDlg, IDC_BTN_START_BATCH), FALSE);

   AddMessage(TEXT("----Start a new batch----"));

   // Submit 4 tasks by using the same work item
   SubmitThreadpoolWork(g_pWorkItem);
```

```
   SubmitThreadpoolWork(g_pWorkItem);
   SubmitThreadpoolWork(g_pWorkItem);
   SubmitThreadpoolWork(g_pWorkItem);

   AddMessage(TEXT("4 tasks are submitted."));
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   switch (id) {
      case IDOK:
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;

      case IDC_BTN_START_BATCH:
         OnStartBatch();
         break;
   }
}


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   // Keep track of main dialog window for error messages
   g_hDlg = hWnd;

   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
      case WM_APP_COMPLETED: {
         TCHAR szMsg[MAX_PATH+1];
         StringCchPrintf(
            szMsg, _countof(szMsg),
            TEXT("____Task #%u was the last task of the batch____"), lParam);
         AddMessage(szMsg);
```

```
         // Don't forget to enable the button
         Button_Enable(GetDlgItem(hWnd, IDC_BTN_START_BATCH), TRUE);
      }
      break;
   }

   return(FALSE);
}



int APIENTRY _tWinMain(HINSTANCE hInstance, HINSTANCE, LPTSTR pCmdLine, int) {

   // Create the work item that will be used by all tasks
   g_pWorkItem = CreateThreadpoolWork(TaskHandler, NULL, NULL);
   if (g_pWorkItem == NULL) {
      MessageBox(NULL, TEXT("Impossible to create the work item for tasks."),
         TEXT(""), MB_ICONSTOP);
      return(-1);
   }

   DialogBoxParam(hInstance, MAKEINTRESOURCE(IDD_MAIN), NULL, Dlg_Proc,
      _ttoi(pCmdLine));

   // Don't forget to delete the work item
   CloseThreadpoolWork(g_pWorkItem);

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

Перед созданием главного окна создается один рабочий элемент. Если эта операция заканчивается неудачей, программа завершается, предварительно отображая сообщение с описанием ошибки. По щелчку кнопки Start один и тот же рабочий элемент ставится вызовом функции *SubmitThreadpoolWork* в очередь пула потоков по умолчанию. При этом кнопка Start деактивируется, чтобы пользователь не запустил обработку еще одного пакета. Функция обратного вызова, исполняемая потоками пула, атомарно увеличивает значение глобального счетчика операций с помощью *Interlockedlncrement* (см. главу 8) и заносит в журнал записи о начале и завершении обработки.

Непосредственно перед завершением функции *TaskHandler* значение счетчика операций атомарно уменьшается с помощью функции *InterlockedDecrement.* При обработке последнего элемента главному окну отправляется специальное сообщение, в результате чего в журнал записыва-

ется завершающее сообщение, после чего вновь активируется кнопка Start. Узнать о том, что обработка пакета завершена, можно и другим способом. При обработке последнего элемента порождается поток, просто вызывающий функцию *WmtForThreadpooWorkCallbacks(g_pWorkItem,* FALSE) Когда эта функция вернет управление, можно считать, что пул потоков обработал все элементы очереди.

Сценарий 2. Вызов функций через определенные интервалы времени

Иногда какие-то операции приходится выполнять через определенные промежутки времени. В Windows имеется объект ядра «ожидаемый таймер», который позволяет легко получать уведомления по истечении заданного времени. Многие программисты создают такой объект для каждой привязанной к определенному времени задаче, но это ошибочный путь, ведущий к пустой трате системных ресурсов. Вместо этого вы можете создать единственный ожидаемый таймер и каждый раз перенастраивать его на другое время ожидания. Однако такой код весьма непрост. К счастью, теперь эту работу можно поручить новым функциям пула потоков. Чтобы исполнить некоторую операцию в заданное время, определите функцию обратного вызова с прототипом следующего вида:

```
VOID CALLBACK TimeoutCallback(
    PTP_CALLBACK_INSTANCE plnstance,        // см.  раздел "Обработка завершения
                                            // обратного вызова"
    PVOID pvContext,
    PTP_TIMER pTimer);
```

Далее сообщите пулу потоков, когда следует вызвать эту функцию:

```
PTP_TIMER CreateThreadpooiTimer(
    PTP_TIMER_CALLBACK pfnTimerCallback,
    PVOID pvContext,
    PTP_CALLBACK_ENVIRON pcbe);             // см.  раздел "Настройка пула потоков
```

Эта функция работает аналогично *CreateThreadpoolWork* о которой говорилось в предыдущем разделе. Параметр *pfnTimerCallback* содержит адрес функции, объявленной в соответствии с прототипом *TtmeoutCallback,* значение параметра pvContext передается функции обратного вызова. При вызове вашей функции *TimerCallback* в ее параметр *pTimer* записывается указатель на объект, созданный функцией *CreateThreadpoolTimer.*

Чтобы зарегистрировать таймер у пула потоков, вызовите функцию *SetThreadpoolTimer*.

```
VOID SetThreadpoolTimer(
    PTP_TIMER pTimer,
    PFILETIME pftDueTime,
```

```
DWORD msPeriod,
DWORD msWindowLength);
```

Ее параметр *pTimer* задает объект TP_TIMER, созданный функцией *CreateThreadpoolTimer*. Параметр *pftDueTime* определяет, когда функция обратного вызова будет исполнена в первый раз. Чтобы указать относительный интервал срабатывания таймера (он будет отсчитываться от времени первого срабатывания), следует задать отрицательное значение (в миллисекундах). Специальное значение -1 вызывает немедленное срабатывание таймера. Абсолютные интервалы задают в виде положительных значений (с шагом в 100 наносекунд, время отсчитывается от 1 января 1600 г).

Чтобы таймер сработал только один раз, установите параметр *msPeriod* в 0. Если же требуется, чтобы поток периодически вызвал вашу функцию, присвойте параметру *msPeriod* значение, отличное от нуля (оно задает интервал между вызовами функции *TimerCallback).* Параметр *msWindowLength* вносит небольшой элемент случайности в исполнение функции обратного вызова. Эта функция вызывается в любой момент интервала, длительность которого задана параметром *msWindowLength;* отсчитывается этот интервал от текущего времени. Это удобно, когда в программе есть несколько таймеров, срабатывающих примерно с одинаковой частотой, и требуется избежать одновременного срабатывания таймеров. Таким образом, *msWindowLength* позволяет обойтись без вызова *Sleep* со случайным аргументом в функциях обратного вызова.

Кроме этого, параметр *msWindowLength* позволяет группировать таймеры. Если у вас имеется множество таймеров, срабатывающих примерно в одно время, имеет смысл сгруппировать их во избежание излишних переключений контекста. Предположим, что таймер А срабатывает через 5 мс, а таймер В — через 6 мс. Через 5 мс поток исполняет функцию обратного вызова таймера А, снова засыпает и тут же просыпается, чтобы исполнить функцию обратного вызова таймера В, и т.д. Чтобы избежать переключения контекста и добавления-удаления потоков из пула, можно установить *msWindowLength* = 2 для таймеров А и В. Тогда пул потоков будет знать, что функция обратного вызова таймера будет выполнена в течение 5-7 мс, начиная с текущего момента, а функция таймера В - в интервале 6-8 мс. В этом случае пул потоков предпочтет сгруппировать эти таймеры и установить для них время срабатывания 6 мс. Таким образом, при срабатывании этих таймеров достаточно будет пробудить один поток, который выполнит функцию обратного вызова сначала таймера А, затем таймера В, и только после этого вернется к ожиданию в пуле. Такая оптимизация особенно важна в случае таймеров с очень близкими временами срабатывания. Учтите также, что чем чаще срабатывают таймеры, тем выше издержки на пробуждение и «усыпление» потоков.

Я хочу подчеркнуть, что после установки таймеры можно изменять, вызывая функцию *SetThreadpoolTimer*. При этом в параметре *pTimer* передается указатель на существующий таймер, а в параметрах *pftDueTime, msPeriod* и

msWindowLength — новые значения. На самом деле, в *pftDueTime* можно передать и NULL тогда поток перестанет вызывать вашу функцию *TimerCallback.* Это хороший способ приостановить таймер, не уничтожая его, особенно удобно это делать из функции обратного вызова.

Функция *IsThreadpoolTimerSet* позволяет узнать, установлен ли таймер (т.е. не содержится ли в *pftDueTime* NULL-значение):

```
BOOL IsThreadpoolTimerSet(PTP_TIMER pti);
```

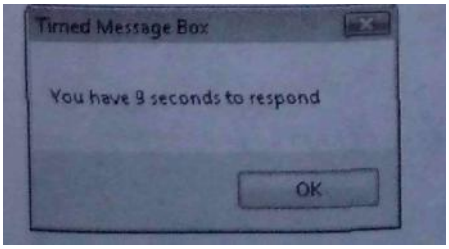
Наконец, можно заставить поток ждать срабатывания таймера, вызвав функцию *WaitForThreadpoolTimerCallbacks,* а уничтожить объект «таймер» — вызовом функции *CloseThreadpoolTimer.* Эти функции работают аналогично функциям *WaitForThreadpoolWork* и *CloselhreadpoolWork,* описанным выше в этой главе.

Программа-пример TimedMsgBox

Эта программа, 11-TimedMsgBox.exe, показывает, как пользоваться таймерными функциями пула потоков для создания окна, автоматически закрываемого через заданное время в отсутствие реакции пользователя. Файлы исходного кода и ресурсов этой программы находятся в каталоге 11-TimedMsgBox внутри архива, доступного на веб-сайте поддержки этой книги.

При запуске программа присваивает глобальной переменной *g_nSecLeft* значение 10. Эта переменная определяет, сколько времени (в секундах) программа ждет реакции пользователя на сообщение, показанное в окне. Далее вызывается функция *CreateThreadpoolTimer,* создающая таймер пула потоков. Затем этот таймер передается функции *SetThreadpoolTimer,* настраивающей пул на ежесекундный вызов *MsgBoxTimeout* Инициализировав все необходимые переменные, программа обращается к *MessageBox* и выводит окно, показанное ниже.

Пока ожидается ответ от пользователя, один из потоков пула каждую секунду вызывает функцию *MsgBoxTimeout,* которая находит описатель этого окна, уменьшает значение глобальной переменной *g_nSecLeft* на 1 и обновляет строку в окне. При первом вызове *MsgBoxTimeout* окно выглядит так.



При девятом вызове *MsgBoxTimeout* переменная *g_nSecLeft* получает значение 1, и тогда *MsgBoxTimeout* вызывает *EndDialog,* чтобы закрыть окно. После этого функция *MessageBox,* вызванная первичным потоком, возвращает управление, и вызывается *CloseThreadpoolTimer,* заставляющая пул прекратить вызовы *MsgBoxTimeout.* В результате открывается другое окно, где сообщается о том, что никаких действий в отведенное время не предпринято.

Если же пользователь успел отреагировать на первое сообщение, на экране появляется то же окно, но с другим текстом.


```
/******************************************************************************
Module:  TimedMsgBox.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
#include <StrSafe.h>


//////////////////////////////////////////////////////////////////////////////


// The caption of our message box
TCHAR g_szCaption[100];


// How many seconds we'll display the message box
int g_nSecLeft = 0;


// This is STATIC window control ID for a message box
#define ID_MSGBOX_STATIC_TEXT    0x0000ffff


//////////////////////////////////////////////////////////////////////////////


VOID CALLBACK MsgBoxTimeoutCallback(
   PTP_CALLBACK_INSTANCE   pInstance,
```

```
   PVOID                   pvContext,
   PTP_TIMER               pTimer
   )
{
   // NOTE: Due to a thread race condition, it is possible (but very unlikely)
   // that the message box will not be created when we get here.
   HWND hwnd = FindWindow(NULL, g_szCaption);

   if (hwnd != NULL) {
      if (g_nSecLeft == 1) {
         // The time is up; force the message box to exit.
         EndDialog(hwnd, IDOK);
         return;
      }

      // The window does exist; update the time remaining.
      TCHAR szMsg[100];
      StringCchPrintf(szMsg, _countof(szMsg),
         TEXT("You have %d seconds to respond"), --g_nSecLeft);
      SetDlgItemText(hwnd, ID_MSGBOX_STATIC_TEXT, szMsg);
   } else {

      // The window does not exist yet; do nothing this time.
      // We'll try again in another second.
   }
}

int WINAPI _tWinMain(HINSTANCE, HINSTANCE, PTSTR, int) {

   _tcscpy_s(g_szCaption, _countof(g_szCaption), TEXT("Timed Message Box"));

   // How many seconds we'll give the user to respond
   g_nSecLeft = 10;

   // Create the threadpool timer object
   PTP_TIMER lpTimer =
      CreateThreadpoolTimer(MsgBoxTimeoutCallback, NULL, NULL);

   if (lpTimer == NULL) {
      TCHAR szMsg[MAX_PATH];
      StringCchPrintf(szMsg, _countof(szMsg),
         TEXT("Impossible to create the timer: %u"), GetLastError());
      MessageBox(NULL, szMsg, TEXT("Error"), MB_OK | MB_ICONERROR);

      return(-1);
   }

   // Start the timer in one second to trigger every 1 second
```

```
   ULARGE_INTEGER ulRelativeStartTime;
   ulRelativeStartTime.QuadPart = (LONGLONG) -(10000000);  // start in 1 second
   FILETIME ftRelativeStartTime;
   ftRelativeStartTime.dwHighDateTime = ulRelativeStartTime.HighPart;
   ftRelativeStartTime.dwLowDateTime  = ulRelativeStartTime.LowPart;
   SetThreadpoolTimer(
      lpTimer,
      &ftRelativeStartTime,
      1000, // Triggers every 1000 milliseconds
      0
      );

   // Display the message box
   MessageBox(NULL, TEXT("You have 10 seconds to respond"),
      g_szCaption, MB_OK);

   // Clean up the timer
   CloseThreadpoolTimer(lpTimer);

   // Let us know if the user responded or if we timed out
   MessageBox(
      NULL, (g_nSecLeft == 1) ? TEXT("Timeout") : TEXT("User responded"),
      TEXT("Result"), MB_OK);

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

Прежде чем перейти к разбору следующего сценария, позвольте мне обратить ваше внимание на пару моментов. Установка периодически срабатывающего таймера гарантирует добавление рабочих элементов в очередь пула потоков с заданным интервалом. Если таймер срабатывает каждые 10 секунд, то и ваша функция обратного вызова будет исполняться каждые 10 секунд. Учтите, что в этом может быть задействовано сразу несколько потоков пула, поэтому исполнение отдельных частей функции обратного вызова может требовать синхронизации. Имейте также в виду, что при перегрузке пула, например из-за нехватки потоков, обработка элементов очереди может задерживаться.

Если такая ситуация вас не устраивает, и вы хотите, чтобы следующий элемент ставился в очередь не раньше чем через 10 секунд после обработки предыдущего, можно пойти другим путем и создать интеллектуальный таймер однократного срабатывания следующим образом.

■ Создайте таймер как обычно вызовом функции *CreateThreadpoolTimer*.

- Вызовите *SetThreadpoolTimer,* при этом в параметре *msPeriod* передайте 0, чтобы настроить таймер на однократное срабатывание.
- После завершения обработки можно перезапустить таймер, повторив шаг 2
- И последнее: чтобы корректно остановить таймер, перед вызовом *CloseThreadpoolTimer* вызовите функцию *WaitForThreadpoolTimerCallbacks,* передав TRUE в последнем параметре - так вы запретите пулу потоков дальнейшую обработку элементов для этого таймера. Если вы забудете сделать это, будет исполнена функция обратного вызова, и при вызове *SetThreadpoolTimer* вы получите исключение.

Заметьте, что для создания таймера однократного срабатывания можно вызвать *SetThreadpoolTimer* с параметром *msPeriod,* равным 0. Функция обратного вызова должна уничтожать таймер вызовом *CloseThreadpoolTimer* до своего завершения, чтобы наверняка освободить ресурсы, занятые пулом потоков.

Сценарий 3. Вызов функций при освобождении отдельных объектов ядра

Майкрософт обнаружила, что во многих приложениях потоки порождаются только для того, чтобы ждать на тех или иных объектах ядра. Как только объект освобождается, поток посылает уведомление и снова переходит к ожиданию того же объекта. Некоторые разработчики умудряются писать программы так, что в них создается несколько потоков, ждущих один объект. Это невероятное расточительство системных ресурсов. Конечно, издержки от создания потоков существенно меньше, чем от создания процессов, но и потоки не воздухом питаются. У каждого из них свой стек, не говоря уж об огромном количестве команд, выполняемых процессором при создании и уничтожении потока. Поэтому надо стараться сводить любые издержки к минимуму.

Если вы хотите зарегистрировать рабочий элемент так, чтобы он обрабатывался при освобождении какого-либо объекта ядра, используйте еще одну новую функцию пула потоков:

```
VOID CALLBACK WaitCallback(
    PTP_CALLBACK_INSTANCE pInstance,        // см. раздел "Обработка завершения
                                            // обратного вызова"
    PVOID Context,
    PTP_WAIT Wait,
    TP_WAIT_RESULT WaitResult);
```

Далее создайте объект, на котором пул будет ждать, вызвав *CreateThreadpoolWait.*

```
PTP_WAIT CreateThreadpoolWait(
```

```
    PTP_WAIT_CALLBACK    pfnWaitCallback,
    PVOID                pvContext,
    PTP_CALLBACK_ENVIRON pcbe);           // см. раздел "Настройка пула потоков"
```

Подготовив все необходимое, свяжите объект ядра с только что созданным объектом, вызвав функцию:

```
VOID SetThreadpoolWait(
    PTP_WAIT  pWaitItem,
    HANDLE    hObject,
    PFILETIME pftTimeout);
```

Ясно, что параметр идентифицирует объект, созданный функцией *CreateThreadpoolWait*, а параметр *hObject* — некоторый объект ядра, при освобождении которого пул потоков вызовет вашу функцию *WaitCallback*. Последний параметр, *pftTimeout*, определяет длительность ожидания пулом освобождения объекта ядра. Если установить его в 0, пул вовсе не будет ждать, отрицательные значения задают относительное время, положительные — абсолютное время, NULL заставит пул ждать бесконечно.

«За кулисами» пул потоков вызывает функцию *WaitForMultipleObjects* (см. главу 9), передавая ей набор описателей, зарегистрированных при помощи *SetThreadpoolWa.it*, и FALSE в параметре *bWait All*. Таким образом, поток в пуле будет пробуждаться при освобождении любого из объектов, описатели которых были переданы *WaitForMultipleObjects*. Поскольку *WaitForMultipleObjects* принимает до 64 описателей, т.е. может ждать максимально на 64 объектах (как задано значением MAXIMUM_WAIT_OBJECTS, см. главу 9), получается, что пул может использовать один поток в расчете на 64 объекта ядра, т.е. работает довольно эффективно.

Кроме того, *WaitForMultipleObjects* запрещает многократную передачу одного и того же описателя, поэтому каждый описатель следует регистрировать с помощью *SetThreadpoolWait* не более одного раза. Однако можно создать с помощью *DuplicateHandle* копию описателя и зарегистрировать ее независимо от оригинала.

При освобождении объекта ядра или истечении срока ожидания один из потоков пула пробуждается и вызывает вашу функцию *WaitCallback* (см. выше). Большинство параметров этой функции понятно без комментариев, за исключением последнего, *WaitResult*. Его тип — TP_WAIT_RESULT (он определен как DWORD-значение). Этот параметр указывает причину вызова функции *WaitCallback* и принимает одно из значений, перечисленных в табл. 11-1.

Табл. 11-1. Значения параметра WaltResuit

Значение	Описание
WAIT_OBJECT_0	Ваша функция обратного вызова получает эти значение, если объект ядра, переданный функции SetThreadpoolWait, не освободится до истечения заданного периода ожидания
WAIT_TIMEOUT	Ваша функция обратного вызова получает эти значение, если объект ядра, переданный функции SetThreadpoolWait, освободится до истечения заданного периода ожидания
WAIT_ABANDONED_0	Ваша функция обратного вызова получает эти значение, если объект ядра, переданный функции SetThreadpoolWait, ссылается на мьютекс, от которого отказался другой поток (см. стр. 267)

После исполнения потоком вашей функции обратного вызова соответствующий объект ожидания становится неактивным. Чтобы еще раз исполнить эту функцию обратного вызова при освобождении того же самого объекта ядра, необходимо снова зарегистрировать объект ожидания, вызвав функцию *SetThreadpoolWait.*

Предположим, вы зарегистрировали объект, позволяющий ожидать на объекте ядра «процесс». Но объект «процесс», однажды освободившись, остается свободным. В этом случае не имеет смысла повторно регистрировать тот же объект ожидания с описателем одного и того же процесса. Однако способы повторного использования объекта ожидания все же существуют: вызовите *SetThreadpoolWait,* передав описатель другого объекта ядра либо просто NULL, чтобы удалить объект из пула.

И последнее об этом. Заставить поток ждать на объекте ожидания можно при помощи функции *WaitForThreadpoolWaitCallbacks,* а чтобы освободить занятую объектом ожидания память, следует вызвать *CloseThreadpoolWait.* Эти функции работают аналогично функциям *WaitForThreadpoolWork* и *CloseThreadpodlWork,* о которых шла речь выше.

> **Примечание.** Ваша функция обратного вызова ни в коем случае не должна вызвать *WaitForThreadpoolWork* и передавать ей ссылку на свой собственный рабочий элемент, иначе вы получите взаимную блокировку. Дело в том, что поток блокируется в ожидании своего завершения, а завершиться он в данный момент не сможет. Также нельзя закрывать | описать объекта ядра, переданный *SetThreadpoolWait,* пока пул потоков ожидает на нем. И последнее: не стоит освобождать зарегистрированное событие вызовом *PulseEvent,* поскольку нет гарантии, что пул потоков действительно будет ждать это событие в момент вызова *PubeEvent.*

Сценарий 4. Вызов функций по завершении запросов асинхронного ввода-вывода

В предыдущей главе рассказывалось об эффективном выполнении асинхронных операций ввода-вывода в Windows с помощью портов завершения ввода-вывода. В этой главе я покажу, как создать пул, потоки которого ожидают на объекте «порт завершения ввода-вывода». Преимущество этого решения в том, что пул потоков берет управление созданием и разрушением объектов на себя, а порт является объектом, на котором ожидают автоматически созданные потоки пула. Открывая файл или устройство, свяжите его с портом завершения ввода-вывода, приписанным к пулу потоков. Далее нужно сообщить потоку, какую функцию он должен вызвать после завершения асинхронной операции ввода-вывода на связанном с портом устройстве. Прежде всего, напишите функцию, соответствующую прототипу следующего вида:

```
VOID CALLBACK OverlappedCompletionRoutine(
    PTP_CALLBACK_INSTANCE pInstance,       // см.  раздел "Обработка завершения
                                           // обратного вызова"
    PVOID                 pvContext,
    PVOID                 pOverlapped,
    ULONG                 IoResult,
    ULONG.PTR             NumberOfBytesTransferred,
    PTP_I0                plo);
```

По завершении операции ввода-вывода этой функции будет передан (через *napaMerppOverlapped)* указатель на структуру OVERLAPPED, использованную для вызова *ReadFile* или *WriteFile,* инициировавшего операцию ввода-вывода Результат операции ввода-вывода передается через параметр *IoResult.* Если операция завершилась успешно, этот параметр содержит значение NO_ERROR. Число переданных байтов передается через параметр *NumberOfBytesTransferred*, а указатель на элемент ввода-вывода этого пула потоков — через параметр *plo.* Параметр *pInstance* подробнее разъясняется в следующем разделе. I Далее вы должны создать для пула объект ввода-вывода. Вызовите функцию *CreateThreadpoolIo* и передайте ей описатель файла или другого устройства (открытого вызовом *CreateFile* с флагом FILE_FLAG_OVERLAPPED), которое нужно связать с портом завершения ввода-вывода пула потоков:

```
PTP_IO CreateThreadpoolIo(
   HANDLE                hDevice,
   PTP_WIN32_I0_CALLBACK pfnloCallback,
   PVOID                 pvContext,
   PTP_CALLBACK_ENVIRON  pcbe);   // см.  раздел "Настройка пула потоков"
```

Подготовив все необходимое, свяжите файл или устройство, инкапсулированное в объекте ввода-вывода, с портом пула потоков. Для этого вызовите следующую функцию:

```
VOID StartThreadpoolIo(PTP_IO pio);
```

Учтите, что функцию *StartThreadpoolIo* необходимо вызывать перед каждым вызовом *ReadFile* или *WriteFile*, иначе обратный вызов (*OveriappedCom pletzonRoutine*) не состоится.

Чтобы отменить обратный вызов после генерации запроса ввода-вывода, вызовите следующую функцию:

```
VOID CancelThreadpoolIo(PTP_IO pio);
```

Эту функцию также следует вызвать, если сгенерировать запрос ввода-вывода не удалось (функция *ReadFile* или *WriteFile* вернула FALSE либо *GetLastError* вернула значение, отличное от ERROR_IO_PENDING).

Закончив работу с файлом или устройством, вызовите *CloseHandle*, чтобы закрыть его описатель и разорвать его связь с пулом потоков. Это делается вызовом следующей функции:

```
VOID CloseThreadpoolIo(PTP_IO pio);
```

Поток также можно заставить ждать исполнения запроса ввода-вывода, вызвав такую функцию:

```
VOID WaitForThreadpoolIoCallbacks(
    PTP_I0 pio,
    BOOL bCancelPendingCallbacks);
```

Если передать TRUE в параметре *bCancelPendingCallbacks*, процедура обратного вызова вызвана не будет (если она уже не вызвана). Достигнутый результат будет аналогичен вызову *CancelThreadpoolIo*.

Обработка завершения обратного вызова

Пул потоков позволяет задать для функции обратного действия, кото-рые должны быть выполнены после ее завершения. Функция обратного вызова принимает недокументированный параметр *pInstance* типа PTP_CALLBACK_INSTANCE, который используется одной из следующих функции:

```
VOID LeaveCriticalSectionWhenCallbackReturns(
    PTP_CALLBACK_INSTANCE pci, PCRITICAL_SECTION pci);
VOID ReleaseMutexWhenCallbackReturns(PTP_CALLBACK_INSTANCE pci, HANDLE mut);
VOID ReleaseSemaphoreWhenCallbackReturns(PTP_CALLBACK_INSTANCE pci,
    HANDLE sem, DWORD crel);
VOID SetEventWhenCallbackReturns(PTP_CALLBACK_INSTANCE pci, HANDLE evt);
VOID FreeLibraryWhenCallbackReturns(PTP_CALLBACK_INSTANCE pci, HMODULE mod);
```

Несложно догадаться, что параметр *pInstance* идентифицирует экземпляр рабочего элемента, таймера или объекта ввода-вывода, который пул потоков обрабатывается в настоящее время. В таблице 11-2 для перечисленных функций приводятся соответствующие действия, которые потоки пула выполняют по завершении обратного вызова.

Табл. 11 -2. Обработчики завершения обратного вызова и их действия

Функция	По завершении обратного вызова поток пула...
LeaveCriticalSectionWhen-CallbackReturns	…вызывает функцию *LeaveCriticalSection* и передает ей структуру CRITICAL_SECTION
ReleaseMutexWhenCall-backReturns	…вызывает функцию *ReleaseMutex* и передает ей описатель заданный параметром HANDLE
ReleaseSemaphoreWhen-CallbackReturns	…вызывает функцию *ReleaseSemaphore* и передает ей описатель заданный параметром HANDLE
SetEventWhenCallbackRe-turns	…вызывает функцию *SetEvent* и передает ей описатель заданный параметром HANDLE
FreeLibraryWhenCallback-Returns	…вызывает функцию *FreeLibrary* и передает ей значение параметра HMODULE

Первые четыре функции позволяют уведомить поток о том, что другой поток пула закончил некоторую операцию. Последняя функция *(FreeLibraryWhenCallbackReturns)* дает возможность выгрузить динамически подключаемую библиотеку (DLL) после возврата управления функцией обратного вызова. Это особенно удобно, когда функция обратного вызова реализована в DLL, которую следует выгрузить, когда функция обратного вызова закончит свою работу. Естественно, функция обратного вызова не может вызвать *FreeLibrary* сама: ее код будет выгружен из адресного пространства процесса, и, когда *FreeLibrary* вернет управление функции обратного вызова, возникнет нарушение доступа.

> **Внимание!** После завершения функции обратного вызова поток пула может выполнить только одно действие. Поэтому нельзя, например, сразу освободить событие и мьютекс после исполнения функции обратного вызова. В данной ситуации последний вызов отменяет эффект предыдущего.

Существуют еще две функции, обрабатывающие завершение обратного вызова:

```
BOOL CallbackMayRunLong(PTP_CALLBACK_INSTANCE pel);
VOID DisassociateCurrentThreadFromCallback(PTP_CALLBACK_INSTANCE pci);
```

Функция *CallbackMayRunLong* не обрабатывает завершение функции обратного вызова, а, скорее, уведомляет пул потоков о том, как эта функция будет исполнена. Функция обратного вызова должна вызвать *CallbackMayRunLong*, если предполагается, что ее исполнение займет много времени. Пул потоков неохотно создает новые потоки, поэтому исполнение длительных операций в очереди пула потоков может страдать от нехватки свободных потоков. Если *CallbackMayRunLong* возвращает TRUE, в пуле есть потоки, которые смогут обработать стоящие в очереди элементы. Если же эта функция возвращает FALSE, в пуле нет потоков, способных обработать

длительную операцию, и для более эффективной обработки ее лучше разбить на несколько более коротких операций, а затем поставить их в очередь по отдельности. Первая из них может быть исполнена текущим потоком.

Функция обратного вызова вызывает довольно сложную функцию *DisassociateCurrentThreadFromCallback,* чтобы уведомить пул потоков о логическом завершении своей работы. Это позволяет любым потокам пула, заблокированным при вызове функций *WaitForThreadpoolWorkCallbacks, WaitForThreadpoolTimerCattbacks, WaitForThreadpoolWaitCallbacks* или *WaitForThreadpoolIoCallbacks,* не ждать реального завершения функции обратного вызова.

Настройка пула потоков

Функции *CreateThreadpoolWork, CreateThreadpoolTimer, CreateThreadpoolWait* и *CreateThreadpoolIo* принимают параметр PTP_CALLBACK_ENVIRON. Если передать в нем NULL-значение, рабочие элементы будут ставиться в очередь пула потоков по умолчанию, оптимально настроенного для работы с большинством приложений.

Однако иногда для приложений требуется пул потоков, настроенный особым образом. Например, одним приложениями требуется пул с другим максимальным числом потоков, а другим — несколько пулов, в которых потоки создаются и уничтожаются независимо друг от друга.

Чтобы создать новый пул потоков для своего приложения, вызовите следующую функцию:

```
PTP_P00L CreateThreadpool(PVOID reserved);
```

Несложно догадаться, что параметр reserved зарезервирован, и в нем следует передавать NULL. Возможно, он будет использоваться в следующих версиях Windows. Эта функция возвращает значение PTP_POOL со ссылкой на новый пул потоков. Теперь можно задать максимальное и минимальное число потоков в пуле вызовом следующих функций:

```
BOOL SetThreadpoolThreadMinimum(PTP_POOL pThreadPool. DWORD cthrdMin);
BOOL SetThreadpoolThreadMaximum(PTP_POOL pThreadPool, DWORD cthrdMost);
```

Пул поддерживает заданное число потоков и следит, чтобы их число не 1 превышало максимума. По умолчанию минимальное число потоков в пуле

равно 1, а максимальное — 500.

В редких случаях Windows отменяет запрос, например, если исполняющий его поток завершается. Возьмем функцию *RegNottfyChangeKeyWtue.* Вызывая ее, поток передает описатель события, которое Windows освобождает при изменении системного реестра. Если поток, вызвавший, *RegNotifyChangeKeyValue,* завершится, Windows перестанет освобождать это событие.

Пул создает и разрушает потоки по своему усмотрению для максимально эффективной работы Так есть вероятность (и немалая), вызвавший функцию *RegNotifyChangeKeyValue,* будет уничтожен пулом, после чего

Windows перестанет уведомлять приложение об изменениях реестра. Наверное, лучшее решение этой проблемы состоит в создании специального потока (вызовом *CreateThread),* который вызовет функцию *RegNotifyChangeKeyValue* и продолжит работу. Есть и другое решение. Оно заключается в создании пула потоков, у которого максимальное и минимальное число потоков идентично. Потоки такого пула никогда не уничтожаются. Один из потоков такого пула вызывает функцию *RegNotifyChangeKeyValue,* и Windows бесперебойно уведомляет ваше приложение об изменениях реестра.

Если настроенный пул потоков больше не нужен вашему приложению, следует уничтожить этот пул вызовом функции *CloseThreadpooh*

```
VOID CloseThreadpool(PTP_POOL pThreadPool);
```

После вызова *CloseThreadpool* постановка новых элементов в очередь этого пула невозможна. Все его потоки заканчивают текущие операции и завершаются сами, а все элементы очереди пула, обработка которых еще не началась, отменяются.

Создав собственный пул и указав для него максимальное и минимальное число потоков, инициализируйте структуру данных, представляющую *среду обратных вызовов* (callback environment). Эта структура содержит ряд дополнительных параметров настройки рабочих элементов.

В заголовочном файле WinNT.h структура данных, представляющая среду обратных вызовов потока, определена следующим образом:

```
typedef struct _TP_CALLBACK_ENVIRON {
    TP_VERSION                      Version;
    PTP_P00L                        Pool;
    PTP_CLEANUP_GROUP               CleanupGroup;
    PTP_CLEANUP_GROUP_CANCEL_CALLBACK  CleanupGroupCancelCallback;
    PVOID                           RaceDll;
    struct _ACTIVATION_CONTEXT      *ActivationContext;
    PTP_SIMPLE_CALLBACK             FinalizationCallback;
    union {
        DWORD                       Flags;
        struct {
            DWORD                   LongFunction :  1;
            DWORD                   Private      : 31;
        } s;
    } u;
} TP_CALLBACK_ENVIRON, *PTP_CALLBACK_ENVIRON;
```

Технически вы можете анализировать эту структуру и вручную манипулировать ее полями, но делать этого не следует. Считайте ее недокументированной и используйте для работы с ее полями соответствующие функции, объявленные в файле WinBase.h. Чтобы инициализировать эту структуру, вызовите следующую функцию:

```
VOID InitializeThreadpoolEnvironment(PTP_CALLBACK_ENVIRON pcbe);
```

Эта встраиваемая (inline) функция записывает 0 во все поля за исключением *Version,* в которое записывается 1. Среду обратных вызовов, ставшую ненужной, как и любую другую структуру, необходимо корректно разрушить вызовом функции *DestroyThreadpoolEnvironment*

```
VOID DestroyThreadpoolEnvironment(PTP_CALLBACK_ENVIRON pcbe);
```

В среде обратных вызовов должно быть указано, какой из пулов должен обрабатывать рабочие элементы, которые ставятся в очередь. Сделать это можно вызовом функции *SetThreadpoolCallbackPool* с передачей значения PTP_POOL (возвращаемого функцией *CreateThreadpoot):*

```
VOID SetThreadpoolCallbackPool(PTP_CALLBACK_ENVIRON pcbe,  PTP_P00L pThreadPool);
```

Если *SetThreadpoolCallbackPool* не вызвана, то в поле *Pool* структуры TP_CALLBACK_ENVIRON остается NULL, и рабочие элементы попадают в очередь пула потоков по умолчанию. Функция *SetThreadpoolCallbackRunsLong* «сообщает» среде обратных вызовов, что обработка элементов очереди, как правило, занимает длительное время. В результате пул быстрее создает потоки, стараясь полностью обработать элементы и жертвуя ради этого производительностью:

```
VOID SetThreadpoolCallbackRunsLong(PTP_CALLBACK_ENVIRON pcbe);
```

Функция *SetThreadpoolCallbackLibrary* гарантирует, что заданная DLL не будет выгружена из адресного пространства процесса, пока у пула потоков остаются необработанные элементы:

```
VOID SetThreadpoolCallbackLibrary(PTP_CALLBACK_ENVIRON pcbe,  PVOID mod);
```

В сущности, *SetThreadpoolCallbackLibrary* предотвращает потенциальные взаимные блокировки. Подробнее об этой довольно сложной функции см. в документации Platform SDK.

Корректное разрушение пула потоков и группы очистки

Пулы потоков обрабатывают множество рабочих элементов, поступающих в их очереди из различных источников. По этой причине сложно узнать, когда пул потоков уже завершил обработку своей очереди может быть корректно разрушен. Для координации уничтожения пулов потоков предназначен механизм под названием *«группы очистки»* (cleanup groups). Учтите, что сказанное в этом разделе не касается пула потоков по умолчанию, поскольку уничтожить его нельзя: он живет, пока жив процесс, а после завершения процесса Windows автоматически уничтожает пул по умолчанию и освобождает все занятые им ресурсы.

Выше уже говорилось о том, как инициализировать структуру TP_CALLBACK_ENVIRON, которая используется для постановки элемен-

тов в очередь пользовательского пула потоков. Чтобы корректно уничтожить такой пул, необходимо сначала создать группу очистки, вызвав функцию *CreateThreadpoolCleanupGroup:*

```
PTP_CLEANUP_GROUP CreateThreadpoolCleanupGroup();
```

Далее следует связать группу очистки со структурой TP_CALLBACK_ENVIRON, уже связанной с пулом потоков. Для этого вызовите следующую функцию:

```
VOID SetThreadpoolCallbackCleanupGroup(
    PTP_CALLBACK_ENVIRON pcbe,
    PTP_CLEANUP_GROUP ptpcg,
    PTP_CLEANUP_GROUP_CANCEL_CALLBACK pfng);
```

Эта функция устанавливает поля *CleanupGroup* и *CleanupGroupCancelCallback* структуры PTP_CALLBACK_ENVIRON. Параметр *pfng* функции *SetThreadpoolCallbackCleanupGroup* идентифицирует адрес функции обратного вызова, которая исполняется в случае отмены группы очистки. Если этому параметру присвоено значение, отличное от NULL, заданная функция обратного вызова должна соответствовать следующему прототипу:

```
VOID CALLBACK CleanupGroupCancelCallback(
    PVOID pvObjectContext,
    PVOID pvCleanupContext);
```

Если при вызове *CreateThreadpoolWork, CreateThreadpoolTimer*, *CreateThreadpoolWait* или *CreateThreadpoolIo* в последнем параметре для структуры PTP_CALLBACK_ENVIRON передается значение, отличное от NULL, происходит следующее. Созданный этим вызовом объект добавляется в группу очистки среды обратных вызовов соответствующего пула, уведомляя этот пул о потенциальном добавлении рабочего элемента в его очередь. Если после обработки этих элементов будет вызвана функция *CloseThreadpoolWork, CloseThreadpoolTimer, CloseThreadpoolWait* или *CloseThreadpoolIo*, соответствующий объект неявно удаляется из соответствующей группы очистки.

Теперь для уничтожения пула потоков приложение может вызвать функцию:

```
VOID CloseThreadpoolCleanupGroupMembers(
    PTP_CLEANUP_GROUP ptpcg,
    BOOL bCancelPendingCallbacks,
    PVOID pvCleanupContext);
```

Она работает аналогично различным *WaitForThreadpool*-функциям (таким, как *WaitForThreadpoolWork*), о которых рассказывалось выше в этой главе. Когда поток вызывает *CloseThreadpoolCleanupGroupMembers*, он блокируется, пока не будут разрушены все объекты в группе очистки пула, к которому этот поток относится. При желании, передав TRUE в параметре

bCancelPendingCallbacks, можно просто отменить все необработанные элементы. Функция вернет управление, как только завершится обработка текущих элементов. Если в параметре *bCancelPendingCallbacks* передано значении TRUE, а в параметре pfng функции *SetThreadpoolCallbackCleanupGroup* — адрес функции *CleanupGroupCancelCallback*, ваша функция обратного вызова будет вызвана по разу для каждого из отменяемых элементов. Для определения контекста используется параметр *pvContextCreateThreadpool*-функций. У функции *CleanupGroupCancelCallback* этот параметр содержит текст, переданный одноименный параметр функции *CloseThreadpoolCleanupGroupMembers*.

Если вызвать *CloseThreadpoolCleanupGroupMembers*, передав ей FALSE в параметре *bCancelPendingCallbacks*, функция будет ждать завершения обработки всех оставшихся в очереди элементов. Заметьте также, что в параметре *pvCleanupContext* можно передать NULL, ведь ваша функция *CleanupGroupCancelCallback* все равно не будет вызвана.

После обработки либо отмены всех рабочих элементов вызовите функцию *CloseThreadpoolCleanupGroup*, чтобы освободить ресурсы группы очистки:

```
VOID WINAPI CloseThreadpoolCleanupGroup(PTP_CLEANUP_GROUP ptpcg);
```

В завершение вызовите*DestroyThreadpoolEnvironment* и *CloseThreadpool* — пул потоков будет корректно уничтожен.

Оглавление

ГЛАВА 12

Волокна

Майкрософт добавила в Windows поддержку волокон (fibers), чтобы упростить портирование (перенос) существующих серверных приложений из UNIX в Windows. С точки зрения терминологии, принятой в Windows, такие серверные приложения следует считать однопоточными, но способными обслуживать множество клиентов. Иначе говоря, разработчики UNIX-приложений создали свою библиотеку для организации многопоточности и с ее помощью эмулируют истинные потоки. Она создает набор стеков, сохраняет определенные регистры процессора и переключает контексты при обслуживании клиентских запросов.

Разумеется, чтобы добиться большей производительности от таких UNIX-приложений, их следует перепроектировать, заменив библиотеку, эмулирующую потоки, на настоящие потоки, используемые в Windows. Но переработка может занять несколько месяцев, и поэтому компании сначала просто переносят существующий UNIX-код в Windows — это позволяет быстро предложить новый продукт на рынке Windows-приложений.

Но при переносе UNIX-программ в Windows могут возникнуть проблемы. В частности, механизм управления стеком потока в Windows куда сложнее простого выделения памяти. В Windows стеки начинают работать, располагая сравнительно малым объемом физической памяти, и растут по мере необходимости (об этом я расскажу в разделе «Стек потока» главы 16). Перенос усложняется и наличием механизма структурной обработки исключений (см. главы 23,24 и 25).

Стремясь помочь быстрее (и с меньшим числом ошибок) переносить UNIX-код в Windows, Майкрософтдобавила в операционную систему механизм поддержки волокон. В этой главе мы рассмотрим концепцию волокон и функции, предназначенные для операций с ними. Кроме того, я покажу, как эффективнее работать с такими функциями. Но, конечно, при разработке новых приложений следует использовать настоящие потоки.

Работа с волокнами

Во-первых, потоки и Windows реализуются на уровне ядра операционной системы, которое отлично осведомлено об их существовании и «коммутирует» их в соответствии с созданным Майкрософт алгоритмом. В то же время волокна реализованы на уровне кода пользовательского режима, ядро ничего не знает о них, и процессорное время распределяется между волокнами по алгоритму, определяемому нами. А раз так, то о вытеснении волокон говорить не приходится — по крайней мере, когда дело касается ядра.

Второе, о чем следует помнить, — в потоке может быть одно или несколько волокон. Для ядра ноток — все то, что можно вытеснить и что выполняет код. Единовременно поток будет выполнять код лишь одного волокна — какого, решать вам (соответствующие концепции я поясню позже). Приступая к работе с волокнами, прежде всего преобразуйте существующий поток в волокно. Это делает функция *ConvertThreadToFiber*:

```
PV0ID ConvertThreadToFiber(PVOID pvParam);
```

Она создает в памяти контекст волокна (размером около 200 байтов). В него входят следующие элементы:

- определенное программистом значение; оно получает значение параметра *pvParam*, передаваемого в *ConvertThreadToFiber*;
- заголовок цепочки структурной обработки исключении;
- начальный и конечный адреса стека волокна; при преобразовании потока в волокно оп служит и стеком потока;
- регистры процессора, включая указатели стека и команд.

По умолчанию, на компьютерах с архитектурой *x*86 сведения о состоянии вычислении с плавающей точкой не хранятся в регистрах процессора, содержимое которых хранит каждое волокно. В результате при выполнении волокном операций с плавающей точкой возможно повреждение памяти. Чтобы избежать этого, следует вызывать новую функцию *ConvertThreadToFiber*, которая позволяет передавать в параметре *dwFlags* флаг FIBER_FLAG_FLOAT_SWITCH:

```
PV0ID ConvertThreadToFiberEx(
    PV0ID pvParam,
    DWORD dwFlags);
```

Создав и инициализировав контекст волокна, вы сопоставляете его адрес с потоком, преобразованным в волокно, и теперь оно выполняется в этом потоке. *ConvertThreadToFiber(Ex)* возвращает адрес, по которому расположен контекст волокна. Этот адрес еще понадобится вам, но ни считывать, ни записывать по нему напрямую ни в коем случае нельзя — с содержимым этой структуры работают только функции, управляющие волокнами. При вызове *ExitThread* завершаются и волокно, и ноток.

Нет смысла преобразовывать поток в волокно, если вы не собираетесь создавать дополнительные волокна в том же потоке. Чтобы создать другое волокно, поток (выполняющий в данный момент волокно), вызывает функцию *CreateFiber*:

```
PVOID CreateFiber(
    DW0RD dwStackSize,
    PFIBER_START_ROUTINE pfnStartAddress,
    PVOID pvParam);
```

Сначала она пытается создать новый стек, размер которого задан в параметре *dwStackSize*. Обычно передают 0, и тогда максимальный размер стека ограничивается 1 Мб, а изначально ему передается две страницы памяти. Если вы укажете ненулевое значение, то для стека будет зарезервирован и передан именно такой объем памяти. При использовании множества волокон можно сэкономить память, необходимую для хранения их стеков. Для этого вместо *CreateFiber* воспользуйтесь следующей функцией:

```
PVOID CreateFiberEx(
    SIZE_T dwStackCommitSize,
    SIZE_T dwStackReserveSize,
    DWORD dwFlags,
    PFIBER_START_ROUTINE pStartAddress,
    PVOID pvParam);
```

Параметр *dwStackCommitSize* устанавливает размер памяти, изначально выделяемой для стека; параметр *dwStackReserveSize* позволяет зарезервировать для стека некоторое количество виртуальной памяти; параметр *dwFlags* принимает то же значение FIBER_FLAG_FLOAT_SWITCH, что и *ConvertThreadToEberEx*, добавляя состояние вычислений с плавающей точкой к контексту волокна. Остальные параметры — те же, что и для функции *CreateFiber*.

Функция *CreateFtber(Ex)* создает и инициализирует новую структуру, представляющую контекст исполнения волокна. При этом пользовательское значение устанавливается по значению параметрариРанш, сохраняются начальный и конечный адреса памяти нового стека, а также адрес функции волокна (переданный в параметре *pfnStartAddress*).

Аргумент *pfnStartAddress* задает адрес функции волокна, которую вам придется реализовать самостоятельно. Эта функция должна соответствовать следующему прототипу:

```
VOID WINAPI FiberFunc(PVOID pvParam);
```

Когда волокно получает процессорное время в первый раз, эта функция вызывается и получает значение *pvParam*, исходно переданное функции *CreateFiber*. В этой функции вы можете делать что угодно, но в прототипе тип возвращаемого ею значения определен как VOID — не потому, что это значение бессмысленно, а просто потому, что функция волокна не должна

завершаться, пока существует волокно! Как только функция волокна завершится, поток и все созданные в нем волокна тут же будут уничтожены.

Подобно *ConvertThreadToFiber(Ex)*, функция *CreateFiber(Ex)* возвращает адрес контекста исполнения волокна. Но, в отличие от *ConvertThreadToFiber(Ex)*, исполнение созданного функцией *CreateBber(Ex)* волокна не начинается, пока исполняется текущее волокно. Дело в том, что исполняться может только одно волокно потока одновременно. Чтобы запустить новое волокно, вызовите функцию *SwitchToFiber*.

```
VOID SwitchToFiber(PVOID pvFiberExecutionContext);
```

Эта функция принимает единственный параметр (*pvFiberExecutionContext*) — адрес контекста волокна, полученный в предшествующем вызове *ConvertThreadToFiber(Ex)* или *CreateFiber(Ex)*. По этому адресу она определяет, какому волокну предоставить процессорное время. *SwitchToFiber* осуществляет такие операции:

1. Сохраняет в контексте выполняемого в данный момент волокна ряд текущих регистров процессора, включая указатели команд и стека.
2. Загружает в регистры процессора значения, ранее сохраненные в контексте волокна, подлежащего выполнению. В их число входит указатель стека, и поэтому при переключении на другое волокно используется именно его стек.
3. Связывает контекст волокна с потоком, и тот выполняет указанное волокно.
4. Восстанавливает указатель команд. Поток (волокно) продолжает выполнение с того места, на каком волокно было прервано в последний раз.

Применение *SwitchToFuber* — единственный способ выделить волокну процессорное время. Поскольку ваш код должен явно вызывать эту функцию в нужные моменты, вы полностью управляете распределением процессорного времени для волокон. Помните: такой вид планирования не имеет ничего общего с планированием потоков. Поток, в рамках которого выполняются волокна, всегда может быть вытеснен операционной системой. Когда поток получает процессорное время, выполняется только выбранное волокно, и никакое другое не получит управление, нока вы сами не вызовете *SwitchToFiber*.

Для уничтожения волокна предназначена функция *DeieteFiber*:

```
VOID DeleteFiber(PVOIO pvFiberExecutionContext);
```

Она удаляет волокно, чей адрес контекста определяется параметром *pvFiberExecutionContext*, освобождает память, занятую стеком волокна, и уничтожает его контекст. Но, если вы передаете адрес волокна, связанного в данный момент с потоком, *DeleteFiber* сама вызывает *ExitThread* — в результате поток и все созданные в нем волокна «погибают».

DeleteFiber обычно вызывается волокном, чтобы удалить другое волокно. Стек удаляемого волокна уничтожается, а его контекст освобождается.

И здесь обратите внимание на разницу между волокнами и потоками: потоки, как правило, уничтожают себя сами, обращаясь к *ExitThread*. Использование с этой целью *TerminateThread* считается плохим тоном — ведь тогда система не уничтожает стек потока. Так вот, способность волокна корректно уничтожать другие волокна нам еще пригодится — как именно, я расскажу, когда мы дойдем до программы-примера.

После удаления всех волокон также можно удалить их состояние из потока, исходного вызвавшего *ConvertThreadToFiber(Ex)*, с помощью *ConvertFiberToThread*, — так удастся полностью освободить память, использованную для преобразования потока в волокно.

Для хранения информации, специфичной для волокна, используется локальная память волокна (Fiber Local Storage, FLS) и функции, предназначенные для работы с ней. Эти функции выполняют те же операции, что и аналогичные функции локальной памяти потока (см. главу 6). Сначала следует вызвать *FlsAlloc*, чтобы выделить слот FLS, доступный всем волокнам, работающим в данном процессе. Эта функция принимает единственный параметр: указатель на функцию обратного вызова, исполняемую при уничтожении волокна либо освобождении FLS-слота вызовом *FlsFree*. Записать специфичные для волокна данные в FLS-слот можно вызовом функции *FlsSetValue*, а прочитать их оттуда — вызовом *FlsGetValue*. Чтобы узнать, находитесь ли вы в контексте исполнения некоторого волокна, достаточно проверить булево значение, возвращаемое функцией *IsThreadAFiber*.

Для удобства предусмотрено еще две функции, управляющие волокнами. В каждый момент потоком выполняется лишь одно волокно, и операционная система всегда знает, какое волокно связано сейчас с потоком. Чтобы получить адрес контекста текущего волокна, вызовите *GetCurrentFiber*.

```
PVOID GetCurrentFiber();
```

Другая полезная функция — *GetHberData*:

```
PV0ID GetFiberData();
```

Как я уже говорил, контекст каждого волокна содержит определяемое программистом значение. Оно инициализируется значением параметра *pvParam*, передаваемого функции *ConvertThreadToFiber(Ex)* или *CreateFiber(Ex)*, и служит аргументом функции волокна. *GetFiberData* просто «заглядывает» в контекст текущего волокна и возвращает хранящееся там значение.

Обе функции — *GetCurrentEber* и *GetFiberData* — работают очень быстро и обычно реализуются компилятором как встраиваемые (т. е. вместо вызовов этих функций он подставляет их код).

Программа-пример Counter

Эта программа, «12 Counter.exe», демонстрирует применение волокон для реализации фоновой обработки. Запустив ее, вы увидите диалоговое окно,

показанное ниже. (Настоятельно советую запустить программу Counter; тогда вам будет легче понять, что происходит в пей и как она себя ведет.)

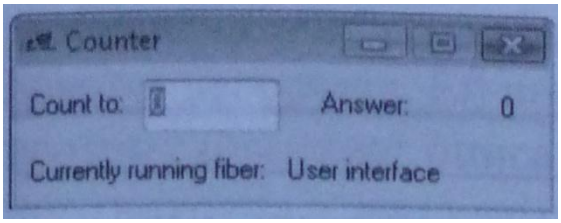


Рис. 12-1. *Окно программы Counter*

Считайте эту программу сверхминиатюрной электронной таблицей, состоящей всего из двух ячеек. В первую из них можно записывать — она реализована как поле, расположенное за меткой Count To. Вторая ячейка доступна только для чтения и реализована как статический элемент управления, размещенный за меткой Answer. Изменив число в поле, вы заставите программу пересчитать значение в ячейке Answer. В этом простом примере пересчет заключается в том, что счетчик, начальное значение которого равно 0, постепенно увеличивается до максимума, заданного в ячейке Count To. Для наглядности статический элемент управления, расположенный в нижней части диалогового окна, показывает, какое из волокон — пользовательского интерфейса или расчетное — выполняется в данный момент.

Чтобы протестировать программу, введите в поле число 5 — строка Currently Running Fiber будет заменена строкой Recalculation, а значение в поле Answer постепенно возрастет с 0 до 5. Когда пересчет закончится, текущим волокном вновь станет интерфейсное, а поток заснет. Теперь введите число 50 и вновь понаблюдайте за пересчетом — на этот раз, перемещая окно по экрану. При этом вы заметите, что расчетное волокно вытесняется, а интерфейсное вновь получает процессорное время, благодаря чему программа продолжает реагировать на действия пользователя. Оставьте окно в покое, и вы увидите, что расчетное волокно снова получило управление и возобновило работу с того значения, на котором было прервано.

Остается проверить лишь одно. Давайте изменим число в поле ввода в момент пересчета. Заметьте: интерфейс отреагировал на ваши действия, но после ввода данных пересчет начинается заново. Таким образом, программа ведет себя как настоящая электронная таблица.

Обратите внимание и на то, что в программе не задействованы ни критические секции, ни другие объекты, синхронизирующие потоки, — все сделано на основе двух волокон в одном потоке.

Теперь обсудим внутреннюю реализацию программы Counter. Когда первичный поток процесса приступает к выполнению *_tWinMain*, вызывается функция *ConvertThreadToFiber*, преобразующая поток в волокно, которое впоследствии позволит нам создать другое волокно. Затем мы создаем немодальное диалоговое окно, выступающее в роли главного окна программы. Далее инициализируем переменную — индикатор состояния фоновой обработки (background processing state, BPS). Она реализована как элемент *bps* в

глобальной переменной *g_FiberInfo*. Ее возможные состояния описываются в следующей таблице.

Табл. 12-1. Варианты состояния программы Counter

Состояние	Описание
BPS_DONE	Пересчет завершен, пользователь ничего не изменял, новый пересчет не нужен
BPS_STARTOVER	Пользователь внес изменения, требуется пересчет с самого начала
BPS_CONTFNUE	Пересчет еще продолжается, пользователь ничего не изменял, пересчет заново не нужен

Индикатор *bps* проверяется внутри цикла обработки сообщений потока, который здесь сложнее обычного. Вот что делает этот цикл.

- Если поступает оконное сообщение (активен пользовательский интерфейс), обрабатываем именно его. Своевременная обработка действий пользователя всегда приоритетнее пересчета.
- Если пользовательский интерфейс простаивает, проверяем, не нужен ли пересчет (т. е. не присвоено ли переменной *bps* значение BPS_STARTOVER или BPS_CONTINUE).
- Если вычисления не нужны (BPS_DONE), приостанавливаем поток, вызывая *WaitMessage*, — только событие, связанное с пользовательским интерфейсом, может потребовать пересчета.
- Если диалоговое окно закрывается, он останавливает расчетное волокно вызовом *DeleteFiber*, после чего вызывается *ConvertFuberToThread* для освобождения ресурсов интерфейсного волокна и возврата потока в обычный режим до завершения функции *_WinMain*.

Если интерфейсному волокну делать нечего, а пользователь только что изменил значение в поле ввода, начинаем вычисления заново (BPS_STARTOVER). Главное, о чем здесь надо помнить, — волокно, отвечающее за пересчет, может уже работать. Тогда это волокно следует удалить и создать новое, которое начнет все с начала. Чтобы уничтожить выполняющее пересчет волокно, интерфейсное вызывает *DeleteFiber*. Именно этим и удобны волокна. Удаление волокна, занятого пересчетом, — операция вполне допустимая, стек волокна и его контекст корректно уничтожаются. Если бы мы использовали потоки, а не волокна, интерфейсный поток не смог бы корректно уничтожить поток, занятый пересчетом, — нам пришлось бы задействовать какой-нибудь механизм межпоточного взаимодействия и ждать, пока поток пересчета не завершится сам. Зная, что волокна, отвечающего за пересчет, больше нет, мы вправе создать новое волокно для тех же целей, присвоив переменной *bps* значение BPS_CONTINUE.

Когда пользовательский интерфейс простаивает, а волокно пересчета чем-то занято, мы выделяем ему процессорное время, вызывая *SwitchToFiber*.

Последняя не вернет упранление, пока волокно пересчета тоже не обратится к *SwilchToFiber,* передан ей адрес контекста интерфейсного волокна.

FiberFunc является функцией волокна и содержит код, выполняемый волокном пересчета. Ей передается адрес глобальной структуры *g_FiberInfo,* и поэтому она знает описатель диалогового окна, адрес контекста интерфейсного волокна и текущее состояние индикатора фоновой обработки. Конечно, раз это глобальная переменная, то передавать ее адрес как параметр необязательно, но я решил показать, как в функцию волокна передаются параметры. Кроме того, передача адресов позволяет добиться того, чтобы код меньше зависел от конкретных переменных, — именно к этому и следует стремиться.

Первое, что делает функция волокна, — обновляет диалоговое окно, сообщая, что сейчас выполняется волокно пересчета. Далее функция получает значение, введенное в поле, и запускает цикл, считающий от 0 до указанного значения. Перед каждым приращением счетчика вызывается *GetQueueStatus* — эта функция проверяет, не появились ли в очереди потока новые сообщения. (Все волокна, работающие в рамках одного потока, делят его очередь сообщений.) Если сообщение появилось, значит, интерфейсному волокну есть чем заняться, и мы, считая его приоритетным по отношению к расчетному, сразу же вызываем *SwitchToFiber,* давая ему возможность обработать поступившее сообщение. Когда сообщение (или сообщения) будет обработано, интерфейсное волокно передаст управление волокну, отвечающему за пересчет, и фоновая обработка возобновится.

Если сообщений нет, расчетное волокно обновляет поле Answer диалогового окна и засыпает на 200 мс. В коде настоящей программы вызов *Sleep* надо, естественно, убрать — я поставил его, только чтобы «потянуть» время.

Когда волокно, отвечающее за пересчет, завершает свою работу, статус фоновой обработки устанавливается как BPS_DONE, и управление передается (через *SxwitchToFiber*) интерфейсному волокну. В этот момент ему делать нечего, и оно вызывает *WaitMessage,* которая приостанавливает поток, чтобы не тратить процессорное время понапрасну.

Заметьте, что каждое волокно записывает в FLS-слот строку-идентификатор («User interface» или «Computation»), которая выводится при регистрации различных событий, таких как удаление волокна или FLS-слота. Это делает функция обратного вызова, которая задается при создании FLS-слота; для проверки возможности использования FLS-слота эта функция использует функцию *IsThreadAFiber*.

ЧАСТЬ III

УПРАВЛЕНИЕ ПАМЯТЬЮ

Оглавление

ГЛАВА 13

Архитектура памяти в Windows

Архитектура памяти, используемая в операционной системе, — ключ к пониманию того, как система делает то, что она делает. Когда начинаешь работать с новой операционной системой, всегда возникает масса вопросов. Как разделить данные между двумя приложениями? Где хранится та или иная информация? Как оптимизировать свою программу? Список вопросов можно продолжить.

Обычно знание того, как система управляет памятью, упрощает и ускоряет поиск ответов на эти вопросы. Поэтому здесь мы рассмотрим архитектуру памяти, применяемую в Microsoft Windows.

Виртуальное адресное пространство процесса

Каждому процессу выделяется собственное виртуальное адресное пространство. Для 32-разрядных процессов его размер составляет 4 Гб. Соответственно 32-битный указатель может быть любым числом от 0x00000000 до 0xFFFFFFFF. Всего, таким образом, указатель может принимать 4 294 967 296 значений, что как раз и перекрывает четырехгигабайтовый диапазон. Для 64-разрядных процессов размер адресного пространства равен 16 экзабайтам, поскольку 64-битный указатель может быть любым числом от 0x00000000 00000000 до 0xFFFFFFFF FFFFFFFF и принимать 18 446 744 073 709 551 616 значений, охватывая диапазон в 16 экзабайтов. Весьма впечатляюще!

Поскольку каждому процессу отводится закрытое адресное пространство, то, когда в процессе выполняется какой-нибудь поток, он получает доступ только к той памяти, которая принадлежит его процессу. Память, отведенная другим процессам, скрыта от этого потока и недоступна ему.

> **Примечание**. В Windows память, принадлежащая собственно операционной системе, тоже скрыта от любого выполняемого потока. Иными словами, ни один поток не может случайно повредить ее данные.

Итак, как я уже говорил, адресное пространство процесса закрыто. Отсюда вытекает, что процесс Л в своем адресном пространстве может хранить какую-то структуру данных по адресу 0x12345678, и одновременно у процесса В по тому же адресу — но уже в его адресном пространстве — может находиться совершенно иная структура данных. Обращаясь к памяти по адресу 0x12345678, потоки, выполняемые в процессе А, получают доступ к структуре данных процесса А. Но, когда по тому же адресу обращаются потоки, выполняемые в процессе В, они получают доступ к структуре данных процесса В. Иначе говоря, потоки процесса А не могут обратиться к структуре данных в адресном пространстве процесса В, и наоборот.

А теперь, пока вы не перевозбудились от колоссального объема адресного пространства, предоставляемого вашей программе, вспомните, что оно — *виртуальное*, а не физическое. Другими словами, адресное пространство — всего лишь диапазон адресов памяти. И, прежде чем вы сможете обратиться к каким-либо данным, не вызвав нарушения доступа, придется спроецировать нужную часть адресного пространства на конкретный участок физической памяти (об этом мы поговорим чуть позже).

Как адресное пространство разбивается на разделы

Виртуальное адресное пространство каждого процесса разбивается на разделы. Их размер и назначение в какой-то мере зависят от конкретного ядра Windows (таблица 13-1).

Как видите, ядра 32- и 64-разрядной Windows создают разделы, почти одинаковые по назначению, но отличающиеся по размеру и расположению. Давайте рассмотрим, как система использует каждый из этих разделов.

Табл. 13-1. Так адресное пространство процесса разбивается на разделы

Раздел	32-разрядная Windows (на *x86*)	32-разрядная Windows (на *x86* с ключом /3GB)	64-разрядная Windows (на *x64*)	64-разрядная Windows (на IA-64)
Для выявления нулевых указателей	0x00000000 0x0000FFFF	0x00000000 0x0000FFFF	0x00000000 00000000 0x00000000 0000FFFF	0x00000000 00000000 0x00000000 0000FFFF
Для кода и данных пользовательского режима	0x00010000 0x7FFEFFFF	0x00010000 0xBFFEFFFF	0x00000000 00010000 0x000007FF FFFEFFFF	0x00000000 00010000 0x000006FB FFFEFFFF

Табл. 13-1. *(окончание)*

Раздел	32-разрядная Windows (на *x86*)	32-разрядная Windows (на *x86* с ключом /3GB)	64-разрядная Windows (на *x64*)	64-разрядная Windows (на IA-64)
Закрытый, размером 64 Кб	0x7FFF0000 0x7FFFFFFF	0xBFFF0000 0xBFFFFFFF	0x000007FF FFFF0000 0x000007FF FFFFFFFF	0x000006FB FFFF0000 0x000006FB FFFFFFFF
Для кода и данных ре жима ядра	0x800000000 0xFFFFFFFF	0xC0000000 0xFFFFFFFF	0x00000800 00000000 0xFFFFFFFF FFFFFFFF	0x000006FC 00000000

Раздел для выявления нулевых указателей

Раздел адресного пространства, охватывающий адреса от 0x00000000 до 0x0000FFFF, резервируется для того, чтобы программисты могли выявлять нулевые указатели. Любая попытка чтения или записи в память по этим адресам вызывает нарушение доступа.

Довольно часто в программах, написанных на C/C++, отсутствует скрупулезная обработка ошибок. Например, в следующем фрагменте кода такой обработки вообще нет:

```
int* pnSomeInteger = (int*) malloc(sizeof(int));
*pnSomeInteger = 5;
```

При нехватке памяти *malloc* вернет NULL. Но код не учитывает эту возможность и при ошибке обратится к памяти по адресу 0x00000000. А поскольку этот раздел адресного пространства заблокирован, возникнет нарушение доступа и данный процесс завершится. Эта особенность помогает программистам находить «жучков» в своих приложениях. Заметьте, что в этом разделе запрещено даже резервировать память с помощью функций Win32 API.

Раздел для кода и данных пользовательского режима

В этом разделе находятся адресные пространства процессов. Доступный для использования диапазон адресов и примерный размер раздела пользовательского режима зависит от архитектуры процессора (13-2).

Табл. 13-2. Размер и адреса раздела пользовательского режима для процессоров с разной архитектурой

Архитектура ЦП	Диапазон адресов размера пользовательского режима	Размер раздела пользовательского режима
x86	0x00010000-0x7FFEFFFF	-2 Гб
x86 (с ключом /3GB)	0x00010000-0xBFFFFFFF	-3 Гб
x64	0x00000000’00010000-0x000007FF’FFFEFFFF	-8192 Гб
IA-64	0x00000000’00010000-0x000006FB’FFFEFFFF	-7152 Гб

В этом разделе располагается закрытая (неразделяемая) часть адресного пространства процесса. Ни один процесс не может получить доступ к данным другого процесса, размещенным и этом разделе. Основной объем данных, принадлежащих процессу, хранится именно здесь (это касается всех приложений). Поэтому приложения менее зависимы от взаимных «капризов», и вся система функционирует устойчивее.

> **Примечание.** В Windows сюда загружаются все EXE- и DLL-модули. В каждом процессе эти DLL можно загружать по разным адресам в пределах данного раздела, но так делается крайне редко. На этот же раздел отображаются все проецируемые в память файлы, доступные данному процессу.

Впервые увидев адресное пространство своего 32-разрядного процесса, я был удивлен тем, что его полезный объем чуть ли не вдвое меньше. Неужели раздел для кода и данных режима ядра должен занимать столько места? Оказывается — да. Это пространство нужно системе для кода ядра, драйверов устройств, кэш-буферов ввода-вывода, областей памяти, не сбрасываемых в файл подкачки, таблиц, используемых для контроля страниц памяти в процессе и т. д. По сути, Майкрософт едва-едва втиснула ядро в эти виртуальные два гигабайта. В 64-разрядной Windows ядро наконец получит то пространство, которое ему нужно на самом деле.

Увеличение раздела для кода и данных пользовательского режима до 3 Гб на процессорах *x86*

Некоторые приложения, такие как Microsoft SQL Server, быстрее работают и лучше масштабируются, если расширить доступное им адресное пространство за границы 2 Гб. С этой целью в версии Windows для x86-компьютеров можно увеличивать до 3 Гб. Чтобы отвести процессам раздел пользовательского режима, размер которого больше 2 Гб, и раздел режима ядра размером меньше 1 Гб, следует настроить загрузочную конфигурацию (boot configuration data, BCD) Windows и перезагрузить компьютер (подробнее о BCD см. в официальной статье по ссылке http://www.microsoft.com/whdc/system/platform/firmware/bcd.mspx).

Для настройки BCD следует запустить BCDEdit.exe с ключом /set IncreaseUserVA. Так, команда bcdedit /set IncreaseUserVa 3072 заставляет Windows зарезервировать для процессов раздел пользовательского режима размером 3 Гб и 1-Гб раздел режима ядра (см. табл. 13-2). Минимальное значение параметра IncreaseUserVa — 2048, что соответствует установленному по умолчанию размеру 2 Гб. Явно сбросить этот параметр позволяет команда bcdedit /deletevalue IncreaseUserVa.

> **Совет**. Чтобы узнать текущие параметры BCD, просто выполните в командной строке команду bcdedit /enum (о параметрах BCDEdit см. по ссылке http://msdn2.microsoft.com/en-us/library/aa906211.aspx).

Раньше, когда такого ключа не было, программа не видела адресов памяти по указателю с установленным старшим битом. Некоторые изобретательные разработчики самостоятельно использовали этот бит как флаг, который имел смысл только в их приложениях. При обращении программы по адресам за пределами 2 Гб предварительно выполнялся специальный код, который сбрасывал старший бит указателя. Но, как вы понимаете, когда приложение на свой страх и риск создает себе трехгигабайтовую среду пользовательского режима, оно может с треском рухнуть.

Майкрософт пришлось придумать решение, которое позволило бы подобным приложениям работать в трехгигабайтовой среде. Теперь система в момент запуска приложения проверяет, не скомпоновано ли оно с ключом /LARGEADDRESSAWARE. Если да, приложение как бы заявляет, что обязуется корректно обращаться с этими адресами памяти и действительно готово к использованию трехгигабайтового адресного пространства пользовательского режима. А если нет, операционная система резервирует область памяти размером 1 Гб между 2-Гб регионом пользовательского режима и регионом режима ядра. Это предотвращает выделение памяти по адресам с установленным старшим битом.

Заметьте, что ядро и так с трудом умещается в двухгигабайтовом разделе. Но при использовании ключа /3GB ядру остается всего 1 Гб. Тем самым уменьшается количество потоков, стеков и других ресурсов, которые система могла бы предоставить приложению. Кроме того, система в этом случае способна задействовать максимум 64 Гб оперативной памяти против 128 Гб в нормальных условиях — из-за нехватки виртуального адресного пространства для кода и данных режима ядра, необходимого для управления дополнительной оперативной памятью.

> **Примечание.** Флаг LARGEADDRESSAWARE в исполняемом файле проверяется в тот момент, когда операционная система создает адресное пространство процесса. Для DLL этот флаг игнорируется. При написании DLL вы должны сами позаботиться об их корректном поведении в трехгигабайтовом разделе пользовательского режима.

Уменьшение раздела для кода и данных пользовательского режима до 2 Гб в 64-разрядной Windows

Майкрософт понимает, что многие разработчики захотят как можно быстрее перенести свои 32-разрядные приложения в 64-разрядную среду. Но в исходном коде любых программ полно таких мест, где предполагается, что указатели являются 32-разрядными значениями. Простая перекомпиляция исходного кода приведет к ошибочному усечению указателей и некорректному обращению к памяти.

Однако, если бы система как-то гарантировала, что память никогда не будет выделяться по адресам выше 0x00000000 7FFFFFFF, приложение работало бы нормально. И усечение 64-разрядного адреса до 32-разрядного,

когда старшие 33 бита равны 0, не создало бы никаких проблем. Так вот, система дает такую гарантию при запуске приложения в «адресной песочнице» (address space sandbox), которая ограничивает полезное адресное пространство процесса до нижних 2 Гб.

По умолчанию, когда вы запускаете 64-разрядиое приложение, система резервирует все адресное пространство пользовательского режима, начиная с 0x0000000 80000000, что обеспечивает выделение памяти исключительно в нижних 2 Гб 64-разрядного адресного пространства. Это и есть «адресная песочница». Большинству приложений этого пространства более чем достаточно. А чтобы 64-разрядиое приложение могло адресоваться ко всему разделу пользовательского режима (объемом 4 Тб), его следует скомпоновать с ключом /LARGEADDRESSAWARE.

> **Примечание.** Флаг LARGEADDRESSAWARE в исполняемом файле проверяется в тот момент, когда операционная система создает адресное пространство 64-разрядного процесса. Для DLL этот флаг игнорируется. При написании DLL вы должны сами позаботиться об их корректном поведении в четырехтерабайтовом разделе пользовательского режима.

Раздел для кода и данных режима ядра

В этот раздел помещается код операционной системы, в том числе драйверы устройств и код низкоуровневого управления потоками, памятью, файловой системой, сетевой поддержкой. Все, что находится здесь, доступно любому процессу. В Windows 2000 эти компоненты полностью защищены. Поток, который попытается обратиться по одному из адресов памяти в этом разделе, вызовет нарушение доступа, а это приведет к тому, что система в конечном счете просто закроет его приложение. (Подробнее на эту тему см. главы 23,24 и 25.)

> **Примечание.** В 64-разрядной Windows раздел пользовательского режима (4 Тб) выглядит непропорционально малым по сравнению с 16 777 212 Тб, отведенными под раздел для кода и данных режима ядра. Дело не в том, что ядру так уж необходимо все это виртуальное пространство, а просто 64-разрядное адресное пространство настолько огромно, что его бóльшая часть не задействована. Система разрешает нашим программам использовать 4 Тб, а ядру — столько, сколько ему нужно. К счастью, какие-либо внутренние структуры данных для управления незадействованными частями раздела для кода и данных режима ядра не требуются.

Регионы в адресном пространстве

Адресное пространство, выделяемое процессу в момент создания, практически все *свободно* (незарезервировано). Поэтому, чтобы воспользоваться какой-нибудь его частью, нужно выделить в нем определенные регионы че-

рез функцию *VirtualAlloc* (о ней — в главе 15). Операция выделения региона называется *резервированием* (reserving).

При резервировании система обязательно выравнивает начало региона с учетом так называемой *гранулярности выделения памяти* (allocation granularity). Последняя величина в принципе зависит от типа процессора, но для процессоров, рассматриваемых в книге, она одинакова и составляет 64 Кб.

Резервируя регион в адресном пространстве, система обеспечивает еще и кратность размера региона размеру *страницы* (page). Так называется единица объема памяти, используемая системой при управлении памятью. Как и гранулярность выделения ресурсов, размер страницы зависит от типа процессора. В частности, для процессоров *x86* и *x64* (*x86* с поддержкой EMT) он равен 4 Кб, а для 64-разрядных IA-64 — 8 Кб.

> **Примечание.** Иногда система сама резервирует некоторые регионы адресного пространства в интересах вашего процесса, например, для хранения блока переменных окружения процесса (process environment block, PEB). Этот блок — небольшая структура данных, создаваемая, контролируемая и разрушаемая исключительно операционной системой. Выделение региона под PEB-блок осуществляется в момент создания процесса.
>
> Кроме того, для управления потоками, существующими на данный момент в процессе, система создает блоки переменных окружения потоков (thread environment blocks, TEBs). Регионы под эти блоки резервируются и освобождаются по мере создания и разрушения потоков в процессе.
>
> Но, требуя от вас резервировать регионы с учетом гранулярности выделения памяти (а эта гранулярность на сегодняшний день составляет 64 Кб), сама система этих правил не придерживается. Поэтому вполне вероятно, что границы региона, зарезервированного под PEB- и TEB-блоки, не будут кратны 64 Кб. Тем не менее размер такого региона обязательно кратен размеру страниц, характерному для данного типа процессора.

Если вы попытаетесь зарезервировать регион размером 10 Кб, система автоматически округлит заданное вами значение до большей кратной величины. А это значит, что на *x86* и *x64* будет выделен регион размером 12 Кб, а на IA-64 — 16Кб.

И последнее в этой связи. Когда зарезервированный регион адресного пространства становится не нужным, его следует вернуть в общие ресурсы системы. Эта операция — *освобождение* (releasing) региона — осуществляется вызовом функции *VirtualFree*.

Передача региону физической памяти

Чтобы зарезервированный регион адресного пространства можно было использовать, вы должны выделить физическую память и спроецировать ее на этот регион. Такая операция называется *передачей физической памяти* (committing physical storage). Чтобы передать физическую память зарезервированному региону, вы обращаетесь все к той же функции *VirtualAlloc*.

Передавая физическую память регионам, нет нужды отводить ее целому региону. Можно, скажем, зарезервировать регион размером 64 Кб и передать физическую намять только его второй и четвертой страницам. На рис. 13-1 представлен пример того, как может выглядеть адресное пространство процесса. Как видите, структура адресного пространства зависит от архитектуры процессора. Слева показано, что происходит с адресным пространством на процессоре *x86*/*x64* (страницы по 4 Кб), а справа — на процессоре IA64 (страницы по 8 Кб).

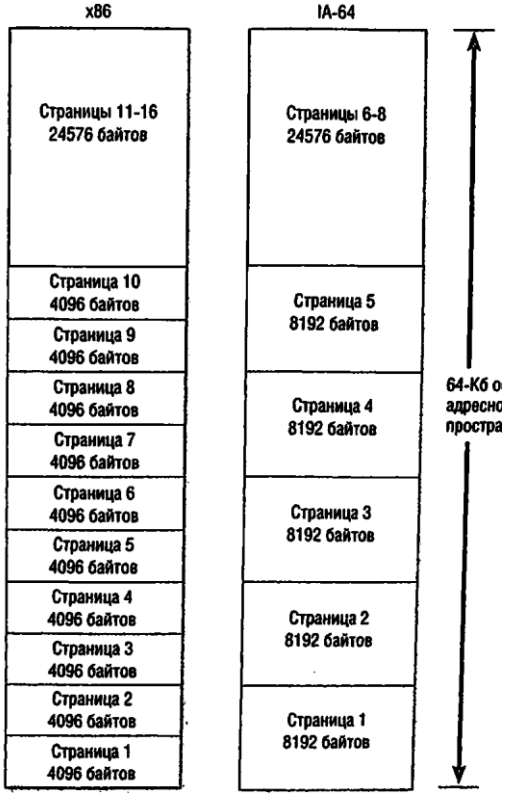


Рис. 13-1. *Примеры адресных пространств процессов для разных типов процессоров*

Когда физическая память, переданная зарезервированному региону, больше не нужна, ее освобождают. Эта операция — *возврат физической памяти* (decommitting physical storage) — выполняется вызовом функции *VirtualFree*.

Физическая память и страничный файл

В старых операционных системах физической памятью считалась вся оперативная память (RAM), установленная в компьютере. Иначе говоря, если в вашей машине было 16 Мб оперативной памяти, вы могли загружать и выполнять приложения, использующие вплоть до 16 Мб памяти. Современные операционные системы умеют имитировать память за счет дискового пространства. При этом на диске создается страничный файл (paging file), который и содержит виртуальную память, доступную всем процессам.

Разумеется, операции с виртуальной памятью требуют соответствующей поддержки от самого процессора. Когда поток пытается обратиться к какому-то байту, процессор должен знать, где находится этот байт — в оперативной памяти или на диске.

С точки зрения прикладной программы, страничный файл просто увеличивает объем памяти, которой она может пользоваться. Если в вашей машине установлено 1 Гб оперативной памяти, а размер страничного файла на жестком диске составляет также 1 Гб, приложение считает, что объем оперативной памяти равен 2 Гб.

Конечно, 2 Гб оперативной памяти у вас на самом деле нет. Операционная система в тесной координации с процессором сбрасывает содержимое части оперативной памяти в страничный файл и по мере необходимости подгружает его порции обратно в память. Если такого файла нет, система просто считает, что приложениям доступен меньший объем памяти, — вот и все. Но, поскольку страничный файл явным образом увеличивает объем памяти, доступный приложениям, его применение весьма желательно. Это позволяет приложениям работать с большими наборами данных.

Физическую память следует рассматривать как данные, хранимые в дисковом файле со страничной структурой. Поэтому, когда приложение передает физическую память какому-нибудь региону адресного пространства (вызывая *VirtualAlloc*), она на самом деле выделяется из файла, размещенного на жестком диске. Размер страничного файла в системе — главный фактор, определяющий количество физической памяти, доступное приложениям. Реальный объем оперативной памяти имеет гораздо меньшее значение.

Теперь посмотрим, что происходит, когда поток пытается получить доступ к блоку данных в адресном пространстве своего процесса. А произойти может одно из двух (рис. 13-2).

В первом сценарии данные, к которым обращается поток, находятся в оперативной памяти. В этом случае процессор проецирует виртуальный адрес данных на физический, и поток получает доступ к нужным ему данным.

Во втором сценарии данные, к которым обращается поток, отсутствуют в оперативной памяти, но размещены где-то в страничном файле. Попытка доступа к данным генерирует ошибку страницы (page fault), и процессор таким образом уведомляет операционную систему об этой попытке. Тогда операционная система начинает искать свободную страницу в оперативной памяти;

если таковой нет, система вынуждена освободить одну из занятых страниц. Если занятая страница нс модифицировалась, она просто освобождается; в ином случае она сначала копируется из оперативной памяти в страничный файл. После этого система переходит к страничному файлу, отыскивает в нем запрошенный блок данных, загружает этот блок на свободную страницу оперативной памяти и, наконец, отображает (проецирует) адрес данных в виртуальной памяти на соответствующий адрес в физической памяти.

Чем чаще системе приходится копировать страницы памяти в страничный файл и наоборот, тем больше нагрузка на жесткий диск и тем медленнее работает операционная система. (При этом может получиться так, что операционная система будет тратить все свое время на подкачку страниц вместо выполнения программ.) Поэтому, добавив компьютеру оперативной памяти, вы снизите частоту обращения к жесткому диску и тем самым увеличите общую производительность системы. Кстати, во многих случаях увеличение оперативной памяти дает больший выигрыш в производительности, чем замена старого процессора на новый.

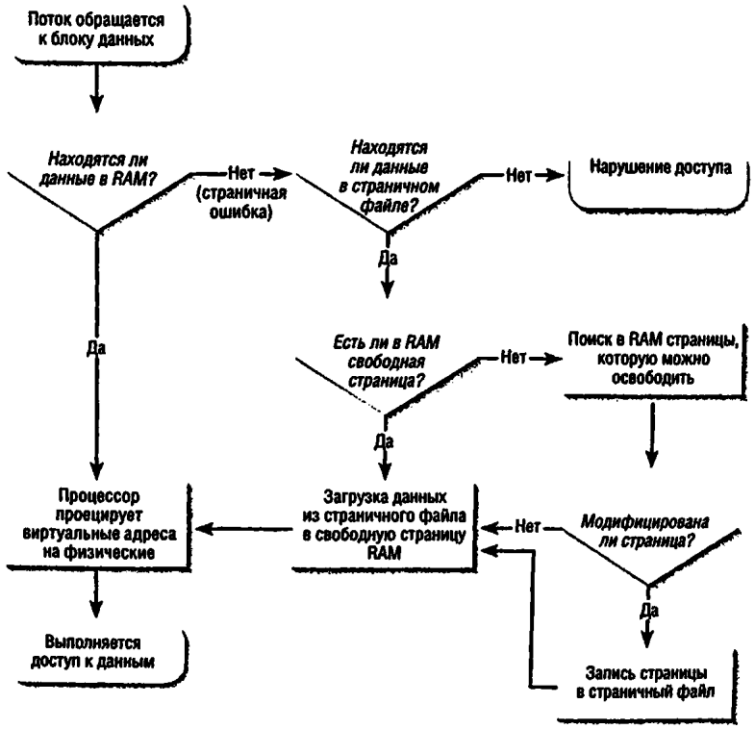


Рис. 13-2. *Трансляция виртуального адреса на физический*

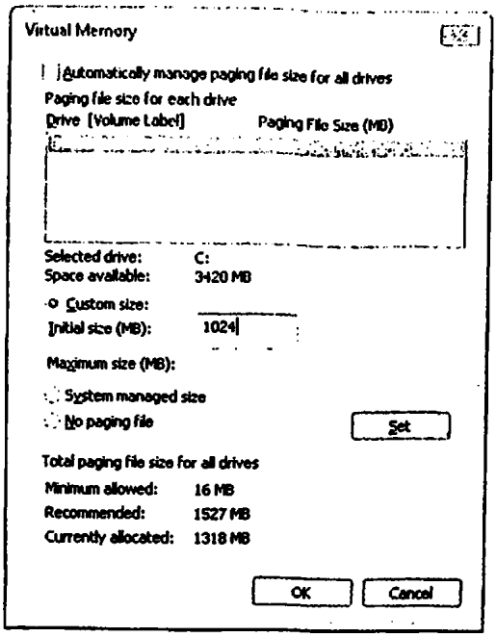
Физическая память в страничном файле не хранится

Прочитав предыдущий раздел, вы, должно быть, подумали, что страничный файл сильно разбухнет при одновременном выполнении в системе нескольких программ, — особенно если вы сочли, будто при каждом запуске приложения система резервирует регионы адресного пространства для кода и данных процесса, передает им физическую память, а затем копирует код и данные из файла программы (расположенного на жестком диске) в физическую память, переданную из страничного файла.

Однако система действует не так, иначе на загрузку и подготовку программы к запуску уходило бы слишком много времени. На самом деле происходит вот что: при запуске приложения система открывает его исполняемый файл и определяет объем кода и данных. Затем резервирует регион адресного пространства и помечает, что физическая память, связанная с этим регионом, — сам EXE-файл. Да-да, правильно: вместо выделения какого-то пространства из страничного файла система использует истинное содержимое, или *образ* (image) EXE-файла как зарезервированный регион адресного пространства программы. Благодаря этому приложение загружается очень быстро, а размер страничного файла удается заметно уменьшить.

Образ исполняемого файла (т. е. EXE- или DLL-файл), размещенный на жестком диске и применяемый как физическая память для того или иного региона адресного пространства, называется *проецируемым в память файлом* (memory-mapped file). При загрузке EXE или DLL система автоматически резервирует регион адресного пространства и проецирует на него образ файла. Помимо этого, система позволяет (с помощью набора функций) проецировать на регион адресного пространства еще и файлы данных. (О проецируемых в память файлах мы поговорим в главе 17.)

> **Примечание.** Windows может использовать несколько страничных файлов, и, если они расположены на разных физических дисках, операционная система работает гораздо быстрее, поскольку способна вести запись одновременно на нескольких дисках. Чтобы добавить или удалить страничный файл, откройте в Control Panel аплет System, щелкните ссылку Advanced, Adjust The Appearance And Performance Of Windows. На вкладке Advanced щелкните в секции Virtual Memory кнопку Change. На экране появится следующее диалоговое окно:



Примечание. Когда EXE- или DLL-файл загружается с дискеты, Windows 98 и Windows 2000 целиком копируют его в оперативную память, а в страничном файле выделяют такое пространство, чтобы в нем мог уместиться образ загружаемого файла. Если нагрузка на оперативную память в системе невелика, EXE- или DLL-файл всегда запускается непосредственно из оперативной памяти.

Так сделано для корректной работы программ установки. Обычно программа установки запускается с первой дискеты, потом поочередно вставляются следующие диски, на которых собственно и содержится устанавливаемое приложение. Если системе понадобится какой-то фрагмент кода EXE- или DLL-модуля программы установки, на текущей дискете его, конечно же, нет. Но, поскольку система скопировала файл в оперативную память (и предусмотрела для него место в страничном файле), у нее не возникнет проблем с доступом к нужной части кода программы установки.

Система не копирует в оперативную память образы файлов, хранящихся на других съемных носителях (CD-ROM или сетевых дисках), если только требуемый файл не скомпонован с использованием ключа /SWAPRUN:CD или /SWAPRUN:NET.

Атрибуты защиты

Отдельным страницам физической памяти можно присвоить свои атрибуты защиты, показанные в следующей таблице.

Табл. 13-3. Атрибуты защиты страниц памяти

Атрибут защиты	Описание
PAGE_NOACCESS	Попытки чтения, записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_READONLY	Попытки записи или исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_READWRITE	Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа
PAGE_EXECUTE	Попытки чтения или записи на этой странице вызывают нарушение доступа
PAGE_EXECUTE_READ	Попытки записи на этой странице вызывают нарушение доступа
PAGE_EXECUTE_ READWRITE	На этой странице возможны любые операции
PAGE_WRITECOPY	Попытки исполнения содержимого памяти на этой странице вызывают нарушение доступа; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы
PAGE_EXECUTE_ WRITECOPY	На этой странице возможны любые операции; попытка записи приводит к тому, что процессу предоставляется «личная» копия данной страницы

Некоторые вредоносные программы записывают код в области памяти, предназначенные для хранения данных (например, в стек потока), а затем исполняют его. Для защиты от подобных атак предназначен механизм Windows, названный Data Execution Prevention (DEP). Если DEP включен, система назначает атрибуты PAGE_EXECUTE_* только тем регионам памяти, в которых хранится предназначенный для исполнения код. Остальные атрибуты (чаще всего PAGE_READWRITE) назначаются регионам, предназначенных для хранения данных (например, регионам, занятым кучами и стеками потоков). При попытке исполнения кода, хранящегося в странице, не имеющей атрибута защиты PAGE_EXECUTE_*, процессор генерирует исключение «нарушение доступа».

У механизма структурной обработки исключений Windows (см. главы 23-25) защита еще надежнее. При компоновке приложения с ключом /SAFESEH обработчики исключений регистрируются в специальной таблице, хранящейся в файле образа. При вызове обработчика исключений система проверяет, зарегистрирован ли он в таблице, и разрешает исполнение только зарегистрированных обработчиков.

Подробнее о DEP см. в официальной статье «03_CIF_Memory_Protection. DOC» по ссылке http://go.microsoft.com/fwlink/?LinkId=28022.

Защита типа «копирование при записи»

Атрибуты защиты, перечисленные и предыдущей таблице, достаточно понятны, кроме двух последних: PAGE_WRITECOPY и PAGE_EXECUTE_ WRITECOPY. Они предназначены специально для экономного расходования оперативной памяти и места в страничном файле. Windows поддерживает механизм, позволяющий двум и более процессам разделять один и тот же блок памяти. Например, если вы запустите 10 экземпляров программы Notepad, все экземпляры будут совместно использовать одни и те же страницы с кодом и данными этой программы. И обычно никаких проблем не возникает — пока процессы ничего не записывают в общие блоки памяти. Только представьте, что творилось бы в системе, если потоки из разных процессов начали бы одновременно записывать в один и тот же блок памяти!

Чтобы предотвратить этот хаос, операционная система присваивает общему блоку памяти атрибут защиты «копирование при записи» (copy-on-write). Когда .exe- или .dll-модуль, проецируется в адресное пространство процесса, система подсчитывает число страниц, доступных для записи (хранящие код страницы обычно помечаются атрибутом PAGE_EXECUTE_READ, а страницы, хранящие данные — атрибутом PAGE_READWRITE) и выделяет в страничном файле место для соответствующего числа страниц. Однако это место реально используется, только если в эти страницы будет что-то записано.

Когда поток в одном процессе попытается что-нибудь записать в общий блок памяти, в дело тут же вступит система и проделает следующие операции:

1. Найдет свободную страницу в оперативной памяти. Заметьте, что при первом проецировании модуля на адресное пространство процесса эта страница будет скопирована на одну из страниц, выделенных в страничном файле. Поскольку система выделяет нужное пространство в страничном файле еще при первом проецировании модуля, сбои на этом этапе маловероятны.
2. Скопирует страницу с данными, которые поток пытается записать в общий блок памяти, на свободную страницу оперативной памяти, полученную на этапе 1. Последней присваивается атрибут защиты PAGE_READWRITE ИЛИ PAGE_EXECUTE_READWRITE. Атрибут защиты и содержимое исходной страницы не меняются.
3. Отобразит виртуальный адрес этой страницы в процессе на новую страницу в оперативной памяти.

Когда система выполнит эти операции, процесс получит свою копию нужной страницы памяти. Подробнее о совместном использовании памяти и о защите типа «копирование при записи» я расскажу в главе 17.

Кроме того, при резервировании адресного пространства или передаче физической памяти через *VirtualAlloc* нельзя указывать атрибуты PAGE_WRITECOPY или PAGE_EXECUTE_WRITECOPY. Иначе вызов *VirtualAlloc* даст ошибку, а *GetLastError* вернет код ERROR_INVALID_PARAMETER.

Дело в том, что эти два атрибута используются операционной системой, только когда она проецирует образы EXE- или DLL-файлов.

Специальные флаги атрибутов защиты

Кроме рассмотренных атрибутов защиты, существует три флага атрибутов защиты: PAGE_NOCACHE, PAGE_WRITECOMBINE и PAGE_GUARD. Они комбинируются с любыми атрибутами защиты (кроме PAGE_NOACCESS) побитовой операцией OR.

Флаг PAGE_NOCACHE отключает кэширование переданных страниц. Как правило, использовать этот флаг не рекомендуется; он предусмотрен главным образом для разработчиков драйверов устройств, которым нужно манипулировать буферами памяти.

Флаг PAGE_WRITECOMBINE тоже предназначен для разработчиков драйверов устройств. Он позволяет объединять несколько операций записи на устройство в один пакет, что увеличивает скорость передачи данных.

Флаг PAGE_GUARD позволяет приложениям получать уведомление (через механизм исключений) в тот момент, когда на страницу записывается какой-нибудь байт. Windows использует этот флаг при создании стека потока. Подробнее на эту тему см. раздел «Стек потока» в главе 16.

Подводя итоги

А теперь попробуем осмыслить понятия адресных пространств, разделов, регионов, блоков и страниц как единое целое. Лучше всего начать с изучения карты виртуальной памяти, на которой изображены все регионы адресного пространства в пределах одного процесса. В качестве примера мы воспользуемся программой VMMap из главы 14. Чтобы в полной мере разобраться в адресном пространстве процесса, рассмотрим его в том виде, в каком оно формируется при запуске VMMap (см. главу 14) под управлением Windows на 32-разрядной процессорной платформе x86. Образец карты адресного пространства VMMap показан в таблице 13-4.

Табл. 13-2. Образец карты адресного пространства процесса в Windows 2000 на 32-разрядном процессоре типа x86

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00000000	Free	65536			
00010000	Mapped	65536	1	-RW-	
00020000	Private	4096	1	-RW-	
00021000	Free	61440			
00030000	Private	1048576	3	-RW-	Стек потока
00130000	Mapped	16384	1	-R-	

Табл. 13-2. (*продолжение*)

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00134000 "	Free	49152			
00140000	Mapped	12288	1	-R-	
00143000	Free	53248			
00150000	Mapped	819200	4	-R-	
00218000	Free	32768			
00220000	Mapped	1060864	1	-R-	
00323000	Free	53248			
00330000	Private	4096	1	-RW-	
00331000	Free	61440			
00340000	Mapped	20480	1	-RWC	\Device\ HarddiskVolume1\ Windows\System32\ en-US\user32.dll.mui
00345000	Free	45056			
00350000	Mapped	8192	1	-R-	
00352000	Free	57344			
00360000	Mapped	4096	1	-RW-	
00361000	Free	61440			
00370000	Mapped	8192	1	-R-	
00372000	Free	450560			
003E0000	Private	65536	2	-RW-	
003F0000	Free	65536			
00400000	Image	126976	7	ERWC	C:\Apps\14 VMMap.exe
0041F000	Free	4096			
00420000	Mapped	720896	1	-R-	
004D0000	Free	458752			
00540000	Private	65536	2	-RW-	
00550000	Free	196608			
00580000	Private	65536	2 ~"	-RW-	
00590000	Free	196608			
005C0000	Private	65536	2	-RW-	
005D0000	Free	262144			
00610000	Private	1048576	2	-RW-	
00710000	Mapped	3661824	1	-R-	\Device\ HarddiskVolume1\ Windows\System32\ locale.nls
00A8E000	Free	8192			

Табл. 13-2. *(продолжение)*

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00A90000	Mapped	3145728	2	-R--	
00D90000	Mapped	3661824	1	-R--	\Device\ HarddiskVolume1\ Windows\System32\ locale.nls
0110E000	Free	8192			
01110000	Private	1048576	2	-RW-	
01210000	Private	524288	2	-RW-	
01290000	Free	65536			
012A0000	Private	262144	2	-RW-	
012E0000	Free	1179648			
01400000	Mapped	2097152	1	-R--	
01600000	Mapped	4194304	1	-R--	
01A00000	Free	1900544			
01BD0000	Private	65536	2	-RW-	
01BE0000	Mapped	4194304	1	-R--	
01FE0000	Free	235012096			
739B0000	Image	634880	9	ERWC	C:\Windows\WinSxS\ x86_microsoft.vc80. crt_lfc8b3b9alel8e3b_ 8.0.50727.312_none_ 10b2ee7b9bffc2c7\ MSVCR80.dll
73A4B000	Free	24072192			
75140000	Image	1654784	7	ERWC	C:\Windows\ WinSxS\x86_ microsoft.windows. common-controls_ 6595b64144ccfldf_ 6.0.6000.16386_none_ 5d07289e07eldl00\ comctl32.dll
752D4000	Free	1490944			
75440000	Image	258048	5	ERWC	C:\Wlndows\ system32\uxtheme.dll
7547F000	Free	15208448			
76300000	Image	28672	1	ERWC	C:\Windows\ system32\PSAPI.dll
76307000	Free	626688			
763A0000	Image	512000	7	ERWC	C:\Wmdows\ system32\USP10.dll
7641D000	Free	12288			

Табл. 13-2. *(продолжение)*

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
76420000	Image	307200	1	ERWC	C:\Windows\ system32\GDI32.dll
7646B000	Free	20480			
76470000	Image	36864	4	ERWC	C:\Windows\ system32\LPK.dll
76479000	Free	552960			
76500000	Image	348160	4	ERWC	C:\Windows\system32\ SHLWAPI.dll
76555000	Free	1880064			
76720000	Image	696320	7	ERWC	C:\Windows\ system32\msvcrt.dll
767CA000	Free	24576			
767D0000	Image	122880	4	ERWC	C:\Windows\ system32\IMM32.dll
767EE000	Free	8192			
767F0000		647168	5	ERWC	C:\Windows\ system32\USER32.dll
7688E000	Free	8192			
76890000	Image	815104	4	ERWC	C:\Windows\ system32\MSCTF.dll
76957000	Free	36864			
76960000	Image	573440	4	ERWC	C:\Windows\system32\ OLEAUT32.dll
769EC000	Free	868352			
76AC0000	Image	798720	4	ERWC	C:\Windows\ system32\RPCRT4.dll
76B83000	Free	2215936			
76DA0000	Image	884736	5	ERWC	C:\Windows\ system32\kernel32.dll
76E7S000	Free	32768			
76E80000	Image	1327104	5	ERWC	C:\Windows\ system32\ole32.dll
76FC4000	Free	11649024			
77AE0000	Image	1171456	9	ERWC	C:\Windows\ system3Antdll,dH
77BFE000	Free	8192			
77C00000	Image	782336	7	ERWC	C:\Windows\system32\ ADVAPI32.dll
77CBF000	Free	128126976			
7F6F0000	Mapped	1048576	2	-R--	
7F7F0000	Free	8126464			

Табл. 13-2. *(окончание)*

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
7FFB0000	Mapped	143360	1	-R--	
7FFD3000	Free	4096			
7FFD4000	Private	4096	1	-RW-	
7FFD5000	Free	40960			
7FFDF000	Private	4096	1	-RW-	
7FFE0000	Private	65536	2	-R--	

Карта в таблице 13-4 показывает регионы, расположенные в адресном пространстве процесса. Каждому региону соответствует своя строка в таблице, а каждая строка состоит из шести полей.

В первом (крайнем слева) поле проставляется базовый адрес региона. Наверное, вы заметили, что просмотр адресного пространства мы начали с региона по адресу 0x00000000 и закончили последним регионом используемого адресного пространства, который начинается по адресу 0x7FFE0000. Все регионы непрерывны. Почти все базовые адреса занятых регионов начинаются со значений, кратных 64 Кб. Это связано с гранулярностью выделения памяти в адресном пространстве. А если вы увидите какой-нибудь регион, начало которого не выровнено по значению, кратному 64 Кб, значит, он выделен кодом операционной системы для управления вашим процессом.

Во втором поле показывается тип региона: Free (свободный), Private (закрытый), Image (образ) или Mapped (проецируемый). Эти типы описаны в следующей таблице.

Табл. 13-5. Типы регионов памяти

Тип	Описание
Free	Этот диапазон виртуальных адресов не сопоставлен ни с каким типом физической памяти. Его адресное пространство не зарезервировано; приложение может зарезервировать регион по указанному базовому адресу или в любом месте в границах свободного региона
Private	Этот диапазон виртуальных адресов сопоставлен со страничным файлом
Image	Этот диапазон виртуальных адресов изначально был сопоставлен с образом EXE- или DLL-файла, проецируемого в память, но теперь, возможно, уже нет. Например, при записи в глобальную переменную из образа модуля механизм поддержки «копирования при записи» выделяет соответствующую страницу памяти из страничного файла, а не исходного образа файла
Mapped	Этот диапазон виртуальных адресов изначально был сопоставлен с файлом данных, проецируемым в память, но теперь, возможно, уже нет. Например, файл данных мог быть спроецирован с использованием механизма поддержки «копирования при записи». Любые операции записи в этот файл приведут к тому, что соответствующие страницы памяти будут выделены из страничного файла, а не из исходного файла данных

Способ вычисления этого ноля моей программой VMMap может давать неправильные результаты. Поясню почему. Когда регион занят, VMMap пытается «прикинуть», к какому из трех оставшихся типов он может относиться, — в Windows нет функций, способных подсказать точное предназначение региона. Я определяю это сканированием всех блоков в границах исследуемого региона, но результатам которого программа делает обоснованное предположение. Но предположение есть предположение. Если вы хотите получше разобраться в том, как это делается, изучите исходный код VMMap, приведенный в главе 14.

В третьем поле сообщается размер региона в байтах. Например, система спроецировала образ User32.dll по адресу 0x767F0000. Когда она резервировала адресное пространство для этого образа, ей понадобилось 647 168 байтов. Не забудьте, что в третьем поле всегда содержатся значения, кратные размеру страницы, характерному для данного процессора (4096 байтов для *x86*). Возможно, вы заметили, что размер дискового файла и число байтов, необходимое для его проецирования в память, различаются. Генерируемый компоновщиком PE-файл сжимается по максимуму в целях экономии места на диске. Однако при проецировании PE-файла в виртуальное адресное пространство процесса и адреса и размеры всех разделы выравниваются по размеру страницы. В этом и заключается причина «разбухания» PE-файла в виртуальной памяти по сравнению с дисковым файлом.

В четвертом поле показано количество блоков в зарезервированном регионе. Блок — это неразрывная группа страниц с одинаковыми атрибутами защиты, связанная с одним и тем же типом физической памяти (подробнее об этом мы поговорим в следующем разделе). Для свободных регионов это значение всегда равно 0, так как им не передается физическая память. (Поэтому в четвертой графе никаких данных для свободных регионов не приводится.) Но для занятых регионов это значение может колебаться в пределах от 1 до максимума (его вычисляют делением размера региона на размер страницы). Скажем, у региона, начинающегося с адреса 0x767F0000, размер — 647 168 байтов. Поскольку процесс выполняется на процессоре x86 (страницы памяти по 4096 байтов), максимальное количество блоков в этом регионе равно 158 (647 168/4096); ну а, судя по карте, в нем содержится 5 блоков.

В четвертом поле показано количество блоков в зарезервированном регионе. Блок — это неразрывная группа страниц с одинаковыми атрибутами защиты, связанная с одним и тем же типом физической памяти (подробнее об этом мы поговорим в следующем разделе). Для свободных регионов это значение всегда равно 0, так как им не передается физическая память. (Поэтому в четвертой графе никаких данных для свободных регионов не приводится.) Но для занятых регионов это значение может колебаться в пределах от 1 до максимума (его вычисляют делением размера региона на размер страницы). Скажем, у региона, начинающегося с адреса 0x77E20000, размер — 401408 байтов. Поскольку процесс выполняется на процессоре *x86* (страницы памяти по 4096 байтов), максимальное количество блоков в этом регионе равно 98 (401 408/4096); ну а, судя по карте, в нем содержится 4 блока.

В пятом поле — атрибуты защиты региона. Здесь используются следующие сокращения: *E* = execute (исполнение), *R* = read (чтение), *W* = *write* (запись), C = copy-on-write (копирование при записи). Если ни один из атрибутов в этой графе не указан, регион доступен без ограничений. Атрибуты защиты не присваиваются и свободным регионам. Кроме того, здесь вы никогда не увидите флагов атрибутов защиты PAGE_GUARD или PAGE_NOCACHE — они имеют смысл только для физической памяти, а не для зарезервированного адресного пространства. Атрибуты защиты присваиваются регионам только эффективности ради и всегда замещаются атрибутами защиты, присвоенными физической памяти.

В шестом (и последнем) поле кратко описывается содержимое текущего региона. Для свободных регионов оно всегда пустое, а для закрытых — обычно пустое, так как у VMMap нет возможности выяснить, зачем приложение зарезервировало данный закрытый регион. Однако VMMap все же распознает назначение тех закрытых регионов, в которых содержатся стеки потоков. Стеки потоков выдают себя тем, что содержат блок физической памяти с флагом атрибутов защиты PAGE_GUARD. Если же стек полностью заполнен, такого блока у него нет, и тогда VMMap не в состоянии распознать стек потока.

Для регионов, содержащих образы, VMMap отображает полный путь к файлу, спроецированному в этот регион. VMMap получает эту информацию с помощью функций PSAPI (см. главу 4). VMMap отображает сведения о регионах, с которыми сопоставлены файлы данных, вызывая функцию *GetMappedFileName*, а сведения о регионах, с которыми сопоставлены файлы образов исполняемых файлов, помогают получить функции ToolHelp API (они также описаны в главе 4).

Блоки внутри регионов

Попробуем увеличить детализацию адресного пространства (по сравнению с тем, что показано в таблице 13-4). Например, таблица 13-6 показывает ту же карту адресного пространства, но в другом «масштабе»: по ней можно узнать, из каких блоков состоит каждый регион.

Табл. 13-6. Образец карты адресного пространства процесса (с указанием блоков внутри регионов) в Windows на 32-разрядном процессоре типа x86

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00000000	Free	65536			
00010000	Mapped	65536	1	-RW-	
00010000	Mapped	65536		-RW- ---	
00020000	Private	4096	1	-RW-	
00020000	Private	4096		-RW- ---	

Табл. 13-6. *(продолжение)*

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
00021000	Free	61440			
00030000	Private	1048576	3	-RW-	Стек потока
00030000	Reserve	774144		-RW- ---	
…	…	…	…	…	…
00330000	Private	4096	1	-RW-	
00330000	Private	4096		-RW- ---	
00331000	Free	61440			
00340000	Mapped	20480	1	-RWC	\Device\ HarddiskVolume1\ Windows\System32\en-US\user32.dll.mui
00340000	Mapped	20480		-RWC ---	
…	…	…	…	…	…
003F0000	Free	65536			
00400000	Image	126976	7	ERWC	C:\Apps\14 VMMap.exe
00400000	Image	4096		-R-- ---	
00401000	Image	8192		ERW- ---	
00403000	Image	57344		ERWC ---	
00411000	Image	32768		ER-- ---	
00419000	Image	8192		-R-- ---	
0041B000	Image	8192		-RW- ---	
0041D000	Image	8192		-R-- ---	
0041F000	Free	4096			
…	…	…	…	…	…
739B0000	Image	634880	9	ERWC	C:\Windows\WinSxS\ x86_microsoft.vc80. crt_lfc8b3b9alel8e3b_ 8.0.50727.312_none_ 10b2ee7b9bffc2c7\ MSVCR80.dll
739B0006	Image	4096		-R-- ---	
739B1000	Image	405504		ER-- ---	
73A14000	Image	176128		-R-- ---	
73A3F000	Image	4096		-RW- ---	
73A40000	Image	4096		-RWC ---	
73A41000	Image	4096		-RW- ---	
73A42000	Image	4096		-RWC ---	
73A43000	Image	12288		-RW- ---	
73A46000	Image	20480		-R-- ---	

Табл. 13-6. *(продолжение)*

Базовый адрес	Тип	Размер	Блоки	Атрибут(ы) защиты	Описание
73A4B000	Free	24072192			
75140000	Image	1654784	7	ERWC	C:\Windows\WinSxS\ x86_microsoft.windows. common-controls_6595b 64144ccf1df_6.0.6000. 16386_none_5d07289e0 7e 1d100\comctl32.dll
75140000	Image	4096		-R-- ---	
75141000	Image	1273856		ER-- ---	
75278000	Image	4096		-RW- ---	
75279000	Image	4096		-RWC ---	
7527A000	Image	8192		-RW- ---	
7527C000	Image	40960		-RWC ---	
75286000	Image	319488		-R-- ---	
752D4000	Free	1490944			
…	…	…	…	…	…
767F0000	Image	647168	5	ERWC	C:\Windows\system32\ USER32.dll
767F0000	Image	4096		-R-- ---	
767F1000	Image	430080		ER-- ---	
7685A000	Image	4096		-RW- ---	
7685B000	Image	4096		-RWC ---	
7685C000	Image	204800		-R-- ---	
7688E000	Free	8192			
…	…	…	…	…	…
76DA0000	Image	884736	5	ERWC	C:\Windows\system32\ kernel32.dll
76DA0000	Image	4096		-R-- ---	
76DA1000	Image	823296		ER-- ---	
76E6A000	Image	8192		-RW- ---	
76E6C000	Image	4096		-RWC ---	
76E6D000	Image	45056		-R-- ---	
76E78000	Free	32768			
…	…	…	…	…	…
7FFDF000	Private	4096	1	-RW-	
7FFDF000	Private	4096		-RW- ---	
7FFE0000	Private	65536	2	-R--	
7FFE0000	Private	4096		-R-- ---	
7FFE1000	Reserve	61440		-R-- ---	

Разумеется, в свободных регионах блоков нет, поскольку им не переданы страницы физической памяти. Строки с описанием блоков состоят из пяти полей.

В первом поле показывается адрес группы страниц с одинаковыми состоянием и атрибутами защиты. Например, по адресу 0x767F0000 передана единственная страница (4096 байтов) физической памяти с атрибутом защиты, разрешающим только чтение. А по адресу 0x767F1000 присутствует блок размером 105 страниц (430 080 байтов) переданной памяти с атрибутами, разрешающими и чтение, и исполнение. Если бы атрибуты защиты этих блоков совпадали, их можно было бы объединить, и тогда на карте памяти появился бы единый элемент размером в 106 страниц (434 176 байтов).

Во втором поле сообщается тип физической памяти, с которой связан тот или иной блок, расположенный в границах зарезервированного региона. В нем появляется одно из пяти возможных значений: Free (свободный), Private (закрытый), Mapped (проецируемый), Image (образ) или Reserve (резервный). Значения Private, Mapped и Image говорят о том, что блок поддерживается физической памятью соответственно из страничного файла, файла данных, загруженного EXE- или DLL-модуля. Если же в поле указано значение Free или Reserve, блок вообще не связан с физической памятью.

Чаще всего блоки в пределах одного региона связаны с однотипной физической памятью. Однако регион вполне может содержать несколько блоков, связанных с физической памятью разных типов. Например, образ файла, проецируемого в память, может быть связан с EXE- или DLL-файлом. Если вам понадобится что-то записать на одну из страниц в таком регионе с атрибутом защиты PAGE_WRITECOPY или PAGE_EXECUTE_WRITECOPY, система подсунет вашему процессу закрытую копию, связанную со страничным файлом, а не с образом файла. Эта новая страница получит те же атрибуты, что и исходная, но без защиты по типу «копирование при записи».

В третьем поле проставляется размер блока. Все блоки непрерывны в границах региона, и никаких разрывов между ними быть не может.

В четвертом поле показывается количество блоков внутри зарезервированного региона.

В пятом поле выводятся атрибуты защиты и флаги атрибутов защиты текущего блока. Атрибуты защиты блока замещают атрибуты защиты региона, содержащего данный блок. Их допустимые значения идентичны применяемым для регионов; кроме того, блоку могут быть присвоены флаги PAGE_GUARD, PAGE_WRITECOMBINJE и PAGE_NOCACHE, недопустимые для региона.

Выравнивание данных

Здесь мы отвлечемся от виртуального адресного пространства процесса и обсудим такую важную тему, как выравнивание данных. Кстати, выравнивание данных — не столько часть архитектуры памяти в операционной системе, сколько часть архитектуры процессора.

Процессоры работают эффективнее, когда имеют дело с правильно выровненными данными. Например, значение типа WORD всегда должно начинаться с четного адреса, кратного 2, значение типа DWORD — с четного адреса, кратного 4, и т. д. При попытке считать невыровненные данные процессор сделает одно из двух: либо возбудит исключение, либо считает их в несколько приемов.

Вот фрагмент кода, обращающийся к невыровненным данным:

```
VOID SomeFunc(PVOID pvDataBuffer) {

   // первый байт в буфере содержит значение типа
   BYTE char c = * (PBYTE) pvDataBuffer;

   // увеличиваем указатель для перехода за этот байт
   pvDataBuffer = (PVOID)((PBYTE) pvDataBuffer + 1);

   // байты 2-5 в буфере содержат значение типа DWORD
   DWORD dw = * (DWORD *) pvDataBuffer;

   // на некоторых процессорах предыдущая строка приведет к исключению
   // из-за некорректного выравнивания данных
…
```

Очевидно, что быстродействие программы снизится, если процессору придется обращаться к памяти в несколько приемов. В лучшем случае система потратит на доступ к невыровненному значению в 2 раза больше времени, чем на доступ к выровненному! Так что, если вы хотите оптимизировать работу своей программы, позаботьтесь о правильном выравнивании данных.

Рассмотрим, как справляется с выравниванием данных процессор типа *x86*. Такой процессор в регистре EFLAGS содержит специальный битовый флаг, называемый флагом AC (alignment check). По умолчанию, при первой подаче питания на процессор он сброшен. Когда этот флаг равен 0, процессор автоматически выполняет инструкции, необходимые для успешного доступа к невыровненным данным. Однако если этот флаг установлен (равен 1), то при каждой попытке доступа к невыровненным данным процессор инициирует прерывание INT 17h. Версия Windows для процессоров типа *x86* никогда не изменяют этот битовый флаг процессора. Поэтому в программе, работающей на процессоре типа x86, исключения, связанные с попыткой доступа к невыровненным данным, никогда не возникают. То же верно для процессоров AMD *x*86-64, по умолчанию использующих аппаратное выравнивание данных.

Теперь обратим внимание на процессор IA-64. Он не умеет оперировать с невыровненными данными. Когда происходит попытка доступа к таким данным, этот процессор уведомляет операционную систему. Далее Windows решает, что делать — генерировать соответствующее исключение или самой устранить возникшую проблему, выдав процессору дополнительные инс-

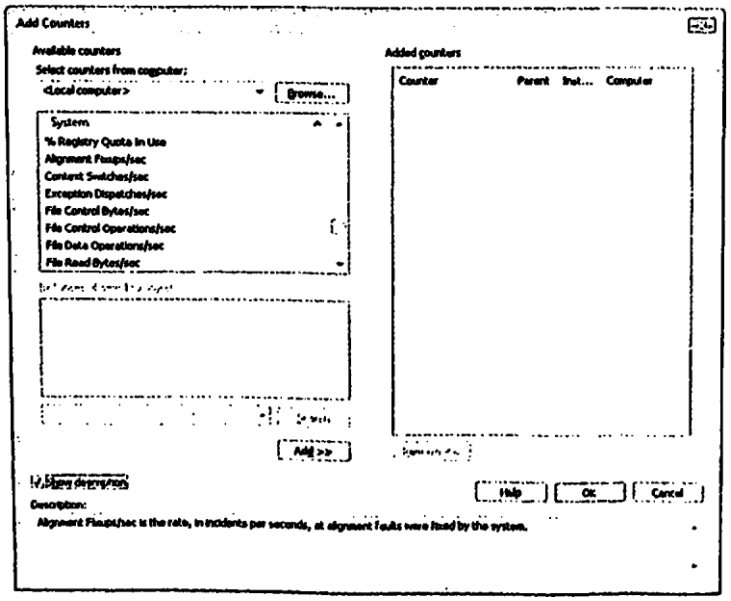
трукции. По умолчанию Windows, установленная на компьютере с процессором IA-64, автоматически преобразует ошибки обращения к невыровненным данным в исключения EXCEPTION_DATATYPE_MISALIGNMENT. Однако вы можете изменить ее поведение, заставив один из потоков процесса автоматически выравнивать данные для остальных потоков этого процесса, вызывая функцию SetErrorMode:

```
UINT SetErrorMode(UINT fuErrorMode);
```

В данном случае вам нужен флаг SEM_NOALIGNMENTFAULTEXCEPT. Когда он установлен, система автоматически исправляет ошибки обращения к невыровненным данным, а когда он сброшен, система вместо этого генерирует соответствующие исключения. Заметьте, что изменение этого флага влияет на потоки только того процесса, из которого была вызвана функция *SetErrorMode*. Иначе говоря, его модификация не отражается на потоках других процессов. Учтите также, что после установки или сброса изменить этот флаг не удастся, пока жив процесс.

Также учтите, что любые флаги режимов обработки ошибок наследуются всеми дочерними процессами. Поэтому перед вызовом функции *CreateProcess* вам может понадобиться временно сбросить этот флаг.

SetErrorMode можно вызывать с флагом SEM_NOALIGNMENTFAULTEXCEPT независимо от того, на какой платформе выполняется Ваше приложение. Но результаты ее вызова не всегда одинаковы. На платформах *x86* и *x64* сбросить этот флаг просто нельзя. Для наблюдения за частотой возникновения ошибок, связанных с доступом к невыровненным данным, в Windows можно использовать Performance Monitor, подключаемый к MMC. На следующей иллюстрации показано диалоговое окно Add Counters, которое позволяет добавить нужный показатель в Performance Monitor.



Этот показатель сообщает, сколько раз в секунду процессор уведомляет операционную систему о доступе к невыровненным данным. На компьютере с процессором типа *x86* он всегда равен 0. Это связано с тем, что такой процессор сам справляется с проблемами обращения к невыровненным данным и не уведомляет об этом операционную систему. А поскольку он обходится без помощи со стороны операционной системы, падение производительности при частом доступе к невыровненным данным не столь значительно, как на процессорах, требующих с той же целью участия операционной системы.

Компилятор Microsoft C/C++ для процессоров IA-64 поддерживает ключевое слово *__unaligned*. Этот модификатор используется так же, как *const* или *volatile*, но применим лишь для переменных-указателей. Когда вы обращаетесь к данным через невыровненный указатель (unaligned pointer), компилятор генерирует код, исходя из того, что данные скорее всего не выровнены, и вставляет дополнительные машинные инструкции, необходимые для доступа к таким данным. Ниже показан тот же фрагмент кода, что и в начале раздела, но с использованием ключевого слова *__unaligned*.

```
V0ID SomeFunc(PVOID pvDataBuffer) {
   // первый байт в буфере содержит значение типа BYTE
   char c = * (PBYTE) pvDataBuffer;

   // увеличиваем указатель для перехода за этот байт
   pvDataBuffer = (PVOID)((PBYTE) pvDataBuffer + 1);

   // байты 2-5 в буфере содержат значение типа DW0RD
   DWORD dw = * (__unaligned DWORD *) pvDataBuffer;

   // Предыдущая строка заставит компилятор сгенерировать дополнительные
   // машинные инструкции, которые позволят считать значение типа DW0RD
   // в несколько приемов. При этом исключение из-за попытки доступа
   // к невыровненный данным не возникнет.
   …
```

Но если я уберу ключевое слово *__unaligned*, то получу всего 3 машинные инструкции. Как видите, модификатор *__unaligned* на процессорах IA-64 приводит к увеличению числа генерируемых машинных инструкций более чем в 2 раза. Но инструкции, добавляемые компилятором, все равно намного эффективнее, чем перехват процессором попыток доступа к невыровненным данным и исправление таких ошибок операционной системой. Используя счетчик Alignment Fixups/sec counter, несложно увидеть, что обращения через невыровненные указатели почти не отражаются на показаниях счетчика. Кстати, компилятор будет генерировать дополнительные инструкции, даже если структуры будут выровнены, что снизит эффективность кода.

И последнее. Ключевое слово *__unaligned* на процессорах типа *x86* компилятором Visual C/C++ не поддерживается. На этих процессорах оно прос-

то не нужно. Но это означает, что версия компилятора для процессоров x86 встретив в исходном коде ключевое слово *__unaligned*, сообщит об ошибке. Поэтому, если вы хотите создать единую базу исходного кода приложения UNALIGNED или UNALIGNED64. Он определен в файле WinNT.h так:

```
#if defined(_M_MRX000) || defined(_M_ALPHA) || defined(_M_PPC) ||
    defined(_M_IA64) || defined(_M_AMD64)
    #define ALIGNMENT_MACHINE
    #define UNALIGNED __unaligned
    #if defined(_WIN64)
        #define UNALIGNED64 __unaligned
    #else
        #define UNALIGNED64
    #endif
#else
    #undef ALIGNMENT_MACHINE
    #define UNALIGNED
    #define UNALIGNED64
#endif
```

Оглавление

ГЛАВА 14

Исследование виртуальной памяти

В предыдущей главе мы выяснили, как система управляет виртуальной памятью, как процесс получает свое адресное пространство и что оно собой представляет. А сейчас мы перейдем от теории к практике и рассмотрим некоторые Windows-функции, сообщающие о состоянии системной памяти и виртуального адресного пространства в том или ином процессе.

Системная информация

Многие параметры операционной системы (размер страницы, гранулярность выделения памяти и др.) зависят от используемого в компьютере процессора. Поэтому нельзя жестко «зашивать» их значения в исходный код программ. Эту информацию надо считывать в момент инициализации процесса с помощью функции *GetSystemInfo*:

```
VOID GetSystemInfo(LPSYSTEM.INFO psi);
```

Вы должны передать в *GetSystemInfo* адрес структуры SYSTEM_WFO, и функция инициализирует элементы этой структуры:

```
typedef struct _SYSTEM_INFO {
    union {
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
    DWORD       dwPageSize;
    LPVOID      lpMinimumApplicationAddress;
    LPVOID      lpMaximuwApplicationAddress;
    DWORD_PTR   dwActiveProcessorMask;
```

```
    DWORD       dwNumberOfProcessors;
    DWORD       dwProcessorType;
    DWORD       dwAllocationGranularity;
    WORD        wProcessorLevel;
    WORD        wProcessorRevision;
} SYSTEM_INFO, *LPSYSTEM_INFO;
```

При загрузке система определяет значения элементов этой структуры; для конкретной системы их значения постоянны. Функция *GetSystemInfo* предусмотрена специально для того, чтобы и приложения могли получать эту информацию. Из всех элементов структуры лишь четыре имеют отношение к памяти. Они описаны в следующей таблице.

Табл. 14-1. Элементы структуры SYSTEM_INFO

Элемент	Описание
dwPageSize	Размер страницы памяти. На процессорах x86 это значение равно 4096, а на процессорах IA-64 — 8192 байтам
lpMinimumApplicationAddress	Минимальный адрес памяти доступного адресного пространства для каждого процесса. В Windows это значение равно 65 536, или 0x00010000, так как в системе резервируются лишь первые 64 Кб адресного пространства каждого процесса
lpMaximumApplicationAddress	Максимальный адрес памяти доступного адресного пространства, отведенного в «личное пользование> каждому процессу
dwAllocationGranularity	Гранулярность резервирования регионов адресного пространства. На момент написания книги это значение составляет 64 Кб для всех платформ Windows.

Остальные элементы этой структуры показаны в таблице ниже.

Табл. 14-2. Элементы структуры SYSTEM_INFO, не связанные с управлением памятью

Элемент	Описание
wReserved	Зарезервирован на будущее; пока не используется
dwNumberOfProcessors	Число процессоров в компьютере. На компьютере с двухъядерным процессором значение этого поля равно двум
dwActiveProcessorMask	Битовая маска, которая сообщает, какие процессоры активны (выполняют потоки)
dwProcessorType	Устарел, больше не используется
wProcessorArchitecture	Сообщает тип архитектуры процессора, например x86, x64 или IA-64

Табл. 14-2. *(окончание)*

Элемент	Описание
wProcessorLevel	Сообщает дополнительные подробности об архитектуре процессора, например Intel Pentium III или Pentium IV. Вместо него для определения возможностей процессора лучше использовать функцию *IsProcessorFeaturePresent*
wProcessorRevision	Сообщает дополнительные подробности об уровне данной архитектуры процессора

Совет. Более подробную информацию об установленных на компьютере процессорах можно получить с помощью функции *GetLogicalProcessorInformation*, как показано на листинге ниже.

```
void ShowProcessors() {
    PSYSTEM_LOGICAL_PROCESSOR_INFORMATION pBuffer = NULL;
    DWORD dwSize = 0;
    DWORD procCoreCount;

    B00L bResult = GetLogicalProcessorInformation(pBuffer, &dwSize);
    if (GetLastError() != ERROR_INSUFFICIENT_BUFFER) {
        _tprintf(TEXT("Impossible to get processor information\n"));
        return;
    }

    pBuffer = (PSYSTEM_LOGICAL_PROCESSOR_INFORMATON)malloc(dwSize);
    bResult = GetLogicalProcessorInformation(pBuffer, &dwSize);
    if (!bResult) {
        free(pBuffer);

        _tprintf(TEXT("Impossible to get processor information\n"));
        return;
    }

    procCoreCount = 0;
    DWORD lpiCount = dwSize / sizeof(SYSTEM_L0GICAL_PR0CESS0R_INFORMATION);
    for(DW0RD current = 0; current < lpiCount; current++) {
        if (pBuffer[current].Relationship = RelationProcessorCore) {
            if (pBuffer[current].ProcessorCpre.Flags == 1) {
                _tprintf(TEXT("   + one CPU core (HyperThreading)\n"));
            } else {
                _tprintf(TEXT("   + one CPU socket\n"));
            }
            procCoreCount++;
        }
    }
```

```
        _tprintf(TEXT("    -> %d active CPU(s)\n"), procCoreCount);

        free(pBuffer);
    }
```

Чтобы 32-разрядныс приложения работали в 64-разрядных версиях Windows, Майкрософт разработан уровень эмуляции под названием *Windows32-bit On Windows 64-bit* или WOW64. В случае 32-разрядного приложения, работающего под WOW64, возвращаемые *GetSystemInfo* значения могут отличаться от таковых в случае «родных» 64-разрядных приложений. Например, поле *dwPageSize* структуры SYSTEM_INFO на компьютере с архитектурой IA-64 в первом случае будет содержать значение 4 Кб, а во втором — 8 Кб. Чтобы выяснить, не запущена ли программа под WOW64:

```
BOOL IsWow64Process(
    HANDLE hProcess,
    PB00L pbWow64Process);
```

Первый параметр — это описатель нужного вам процесса, для исполняемого приложения он может быть получен вызовом *GetCurrentProcess*. Функция *IsWow64Process* обычно возвращает FALSE, если ей переданы недопустимые параметры. Если же она возвращает TRUE, булево значение, на которое указывает параметр *pbWow64Process*, в случае 32-разрядного приложения работающего в 32-разрядной версии Windows устанавливается в FALSE, как и в случае 64-разрядиого приложения в 64-разрядной версии Windows. Это значение устанавливается в TRUE только для 32-разрядного приложения, работающего в WOW64. В этом случае нужно вызвать *GetNativeSystemInfo*, чтобы получить структуру SYSTEM_INFO с «родными» (а не эмулированными) значениями параметров:

```
void GetNativeSystemInfo(
     LPSYSTEM_INFO pSystemInfo);
```

Вместо *IsWow64Process* следует вызвать функцию *IsOS*, объявленную в файле ShlWApi.h, передав ей в качестве параметра значение OS_WOW6432. Если *IsOS* вернет TRUE, вызывающее 32-разрядное приложение работает под WOW64. Если же возвращается значение FALSE, вызывающе 32-разрядное приложение работает в «родной» 32-разрядной среде Windows.

> **Примечание.** Подробнее об эмуляции 32-разрядной среды в 64-разрядных версиях Windows см. в официальной статье «Best Practices for WOW64» по ссылке http://www.microsoft.com/whdc/system/platform/64bit/WoW64_bestprac.mspx.

Программа-пример SysInfo

Эта программа (см. листинг SysInfo.cpp), весьма проста; она вызывает функцию *GetSystemInfo* и выводит на экран информацию, возвращенную в структуре SYSTEM_INFO. Файлы исходного кода и ресурсов этой программы находятся в каталоге 14-SysInfo внутри архива, доступного на веб-сайте поддержки этой книжке. Диалоговые окна с результатами выполнения программы SysInfo на разных процессорных платформах показаны ниже.

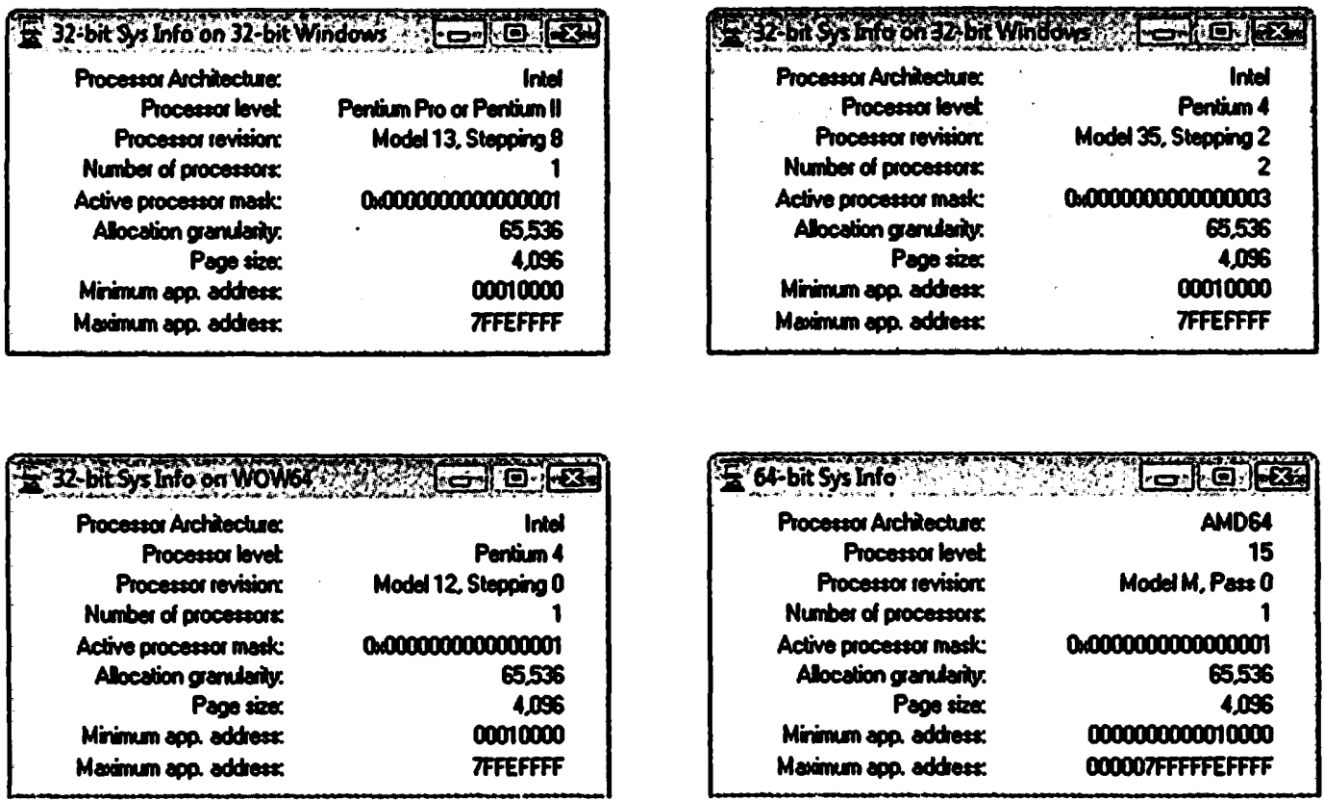


Рис. 14-1. *Окна приложения SysInfo: 32-разрядная версия в 32-разрядной Windows (вверху слева); 32-разрядная версия в 32-разрядной версии Windows на двухъядерном процессоре (вверху справа); 32-разрядная версия в 64-разрядной Windows (внизу слева) и 64-разрядная версия в 64-разрядной Windows (внизу слева)*

```
SysInfo.cpp

/******************************************************************************
    ***

Module:  SysInfo.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************
    **/

#include "..\CmnHdr.h"                     /*см. приложение A*/
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h>
#include "Recourse.h"
#include <StrSafe>
```

```
//////////////////////////////////////////////////////////////////////////////

// Эта функция принимает число и преобразует его в строку,
// вставляя в нужных местах запятые.
PTSTR BigNumToString(LONG lNum, PTSTR szBuf, 0W0R0 chBufSize) {

    TCHAR szNum[100];
    StringCchPrintf(szNum, _countof(szNum), TEXT("%d"), lNun);
    NUNBERFMT nf;
    nf.NumDigits = 0;
    nf.LeadingZero = FALSE;
    nf.Grouping = 3;
    nf.lpDecimalSep = TEXT(".");
    nf.lpThousandSep = TEXT(",");
    nf.NegativeOrder = 0;
    GetNumberFormat(LOCALE_USER_DEFAULT, 0, szNum, &nf, szBuf, chBufSize);
    return(szBuf);
}

//////////////////////////////////////////////////////////////////////////////

void ShowCPUInfo(HWND hWnd, WORD wProcessorArchitecture, WORD wProcessorLevel,
    W0RD wProcessorRevision) {

    TCHAR szCPUArch[64]  = TEXT("(unknown)");
    TCHAR szCPULevel[64] = TEXT("(unknown)");
    TCHAR szCPURev[64]   = TEXT("(unknown)");

    switch (wProcessorArchitecture) {
        // Notice that AMD processors are seen as PROCESSOR_ARCHITECTURE_INTEL.
        // In the Registry, the content of the "VendorIdentifier" key under
        // HKEY_LOCAL_MACHIME\HARDWARE\DESCRIPTIOM\System\CentralProcessor\0
        // is either "GenuineIntel" or "AuthenticAMD"
        //
        // Read http://download.intel.com/design/Xeon/applnots/24161831.pdf
        // for Model numeric codes.
        // http://www.amd.com/us-
           en/assets/content type/white papers and tech docs/
        // 20734.pdf should be used for AMD processors Model numeric codes.
        //
        case PROCESSOR_ARCHITECTURE_INTEL:
            _tcscpy_s(szCPUArch, _countof(szCPUArch), TEXT("Intel"));
```

```
            switch (wProcessorLevel) {
            case 3: case 4:
                StringCchPrintf(szCPULevel, _countof(szCPULevel),
                    TEXT("80%c86"),
                    wProcessorLevel + TEXT('0'));
                StringCchPrintf(szCPURev, _countof(szCPURev), TEXT("%c%d"),
                    HIBYTE(wProcessorRevision) + TEXT('A'),
                    LOBYTE(wProcessorRevision));
                break;

            case 5:
                _tcscpy_s(szCPULevel, _countof(szCPULevel), TEXT("Pentium"));
                StringCchPrintf(szCPURev, _countof(szCPURev),
                    TEXT("Model %d, Stepping %d"),
                    HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
                break;

            case 6:
                switch (HIBYTE(wProcessorRevision)) {//Model
                    case 1:
                        _tcscpy_s(szCPULevel, _countof(szCPULevel),
                            TEXT("Pentium Pro"));
                        break;

                    case 3:
                    case 5:
                        _tcscpy_s(szCPULevel, _countof(szCPULevel),
                            TEXT("Pentium II"));
                        break;

                    case 6:
                        _tcscpy_s(szCPULevel, _countof(szCPULevel),
                            TEXT("Celeron"));
                        break;

                    case 7:
                    case 8:
                    case 11:
                        _tcscpy_s(szCPULevel, _countof(szCPULevel),
                            TEXT("Pentium III"));
                        break;

                    case 9:
                    case 13:
                        _tcscpy_s(szCPULevel, _countof(szCPULevel),
```

```
                    TEXT("Pentium III"));
                    break;

                case 10:
                    _tcscpy_s(szCPULevel, _countof( szCPULevel),
                        TEXT("Pentium Xeon"));
                    break;

                case 15:
                    _tcscpy_s(szCPULevel, _countof(szCPULevel),
                        TEXT("Core 2 Duo"));
                    break;

                default:
                    _tcscpy_s(szCPULevel, _countof(szCPULevel),
                        TEXT("Unknown Pentium"));
                    break;
            }
            StringCchPrintf(szCPURev, _countof(szCPURev),
                TEXT("Model %d, Stepping %d"),
                HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
            break;

        case 15:
            _tcscpy_s(szCPULevel, _countof(szCPULevel),TDCT("Pentium4"));
            StringCchPrintf(szCPURev, _countof(szCPURev),
                TEXT("Model %d, Stepping %d"),
                HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
            break;
        }
        break;

    case PR0CESS0R_ARCHITECTURE_IA64:
        _tcscpy_s(szCPUArch, _countof(szCPUArch), TEXT("IA-64"));
        StringCchPrintf(szCPULevel, _countof(szCPULevel), TEXT("%d"),
            wProcessorLevel);
        StringCchPrintf(szCPURev, _countof(szCPURev), TEXT("Model %c, Pass
            %d"),
            HIBYTE(wProcessorRevislon) + TEXT('A'),
            LOBYTE(wProcessorRevision));
        break;

    case PR0CESS0R_ARCHITECTURE_AMD64:
        _tcscpy_s(szCPUArch, _countof(szCPUArch),TEXT("AMD64"));
        StringCchPrintf(szCPULevel, _countof(szCPULevel), TEXT("%d"),
            wProcessorLevel);
```

```
            StringCchPrintf(szCPURev, _countof(szCPURev), TEXT("Model %c, Pass
                %d"),
                HIBYTE(wProcessorRevision) + TEXT('A'),
                LOBYTE(wProcessorRevision));
            break;

        case PROCESSOR_ARCHITECTURE_UNKNOWN:
        default:
            _tcscpy_s (szCPUArch, _countof (szCPUArch), TEXT("Unknown"));
            break;
    }
    SetDlgItemText (hWnd, IDC_PROCARCH,  szCPUArch);
    SetDlgItemText (hWnd, IDC_PROCLEVEL, szCPULevel);
    SetDlgItemText (hWnd, IDC_PROCREV,   szCPURev);
}


void ShowBitness(HWND hWnd) {
    TCHAR szFullTitle[100];
    TCHAR szTitle[32];
    GetWindowText(hWnd, szTitle, _countof(szTitle));

#if defined(_WIN32)
    BOOL bIsWow64 = FALSE;
    if (!IsWow64Process(GetCurrentProcess(), &bIsWow64)) {
        chFAIL("Failed to get W0W64 state.");
        return;
    }

    if (bIsWow64) {
        StringCchPrintf(szFullTitle, _countof(szFullTitle),
            TEXT("32-bit Xs on W0W64"), szTitle);
    } else {
        StringCchPrintf(szFullTitle, _countof(szFullTitle),
            TEXT("32-bit %s on 32-bit Windows"), szTitle);
    }

#elif defined(_WIN64)
// 64-bit applications can only run on 64-bit Windows,
// so there is nothing special to check except the
// _WIN64 symbol set by the compiler.
    StringCchPrintf(szFullTitle, _countof(szFullTitle),
        TEXT("64-bit %s"), szTitle);
#endif

    SetWindowText(hWnd, szFullTitle);
}
```

```
//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hWnd, HWHD hWndFocus, LPARAM lParam) {

    chSETDLGICONS(hWnd, IDI_SYSINFO);

    SYSTEM_INFO sinf;
    GetSystemInfo(&sinf);

    ShowCPUInfo(hWnd, sinf.wProcessorArchitecture,
        sinf.wProcessorLevel, sinf.wProcessorRevision);

    TCHAR szBuf[50];
    SetDlgItemText(hWnd, IDC_PAGESIZE,
        BigNumToString(sinf.dwPageSize, szBuf, _countof(szBuf)));

    StringCchPrintf(szBuf, _countof(szBuf), TEXT("%p"),
        sinf.lpMinimumApplicationAddress);
    SetDlgItemText(hWnd, IDC_MINAPPADDR, szBuf);

    StringCchPrintf(szBuf, _countof(szBuf), TEXT("%p"),
        sinf.lpMaximumApplicationAddress);
    SetDlgItemText(hWnd, IDC_MAXAPPADDR, szBuf);

    StringCchPrintf(szBuf, _countof(szBuf), TEXT("0x%016I64X"),
        (__int64) sinf.dwActiveProcessorMask);
    SetDlgItemText(hWnd, IDC_ACTIVEPROCMASK, szBuf);

    SetDlgItemText(hWnd, IDC_NUHOFPROCS,
        BigNumToString(sinf.dwNumberOfProcessors, szBuf, _countof(szBuf)));

    SetDlgItemText(hWnd, IDC_ALLOCGRAN,
        BigNumToString(sinf.dwAllocationGranularity, szBuf, _countof(szBuf)));

    ShowBitness(hWnd);

    return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify)
{

    switch (id) {
        case IDCANCEL:
            EndDialog(hWnd, id);
```

```
            break;
    }
}

//////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hDlg, UINT uMsg, WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        chHANDLE_DLGMSG(hDlg, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMSG(hDlg, WM_COMMAND,    Dlg_OnCommand);
    }
    return(FALSE);
}

//////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR, int) {

    DialogBox(hInstExe, MAKEINTRES0URCE(IDD_SYSINFO), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////// End of File ////////////////////////////////
```

Статус виртуальной памяти

Windows-функция *GlobalMemoryStatus* позволяет отслеживать текущее состояние памяти:

```
VOID GlobalMemoryStatus(LPMEMORYSTATUS prost);
```

На мой взгляд, она названа крайне неудачно; имя *GlobalMemoryStatus* подразумевает, что функция каким-то образом связана с глобальными кучами в 16-разрядной Windows. Мне кажется, что лучше было бы назвать функцию *GlobalMemoryStatus* по-другому — скажем, *VirtualMemoryStatus*.

При вызове функции *GlobalMemoryStatus* вы должны передать адрес структуры MEMORYSTATUS. Вот эта структура:

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;
    DWORD dwMenoryLoad;
    SIZE_T dwTotalPhys;
    SIZE_T dwAvailPhys;
    SIZE_T dwTotalPageFile;
    SIZE_T dwAvailPageFile;
```

```
    SIZE_T dwTotalVirtual;
    SIZE_T dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Перед вызовом *GlobalMemoryStatus* надо записать в элемент dwLength размер структуры в байтах. Такой принцип вызова функции дает возможность Майкрософт расширять эту структуру в будущих версиях Windows, не нарушая работу существующих приложений. После вызова *GlobalMemoryStatus* инициализирует остальные элементы структуры и возвращает управление. Назначение элементов этой структуры вы узнаете из следующего раздела, в котором рассматривается программа-пример VMStat.

Если вы полагаете, что ваше приложение будет работать на машинах с объемом оперативной памяти более 4 Гб или файлом подкачки более 4 Гб, используйте новую функцию *GlobalMemoryStatusEx*.

```
BOOL GlobalMemoryStatusEx(LPHEHORYSTATUSEX pmst);
```

Вы должны передать ей адрес новой структуры MEMORYSTATUSEX:

```
typedef struct _MEMORYSTATUSEX {
    DWORD dwLength;
    0W0RD dwMemoryLoad;
    DW0RDL0NG ullTotalPhys;
    DW0RDL0N6 ullAvailPhys;
    DW0RDL0NG ullTotalPageFile;
    DW0RDL0N6 ullAvailPageFile;
    DW0RDL0NG ullTotalVirtual;
    DW0RDL0NG ullAvailVirtual;
    DW0RDL0NG ullAvallExtendedVirtual;
} MEMORYSTATUSEX, *LPMEMORYSTATUSEX;
```

Эта структура идентична первоначальной структуре MEMORYSTATUS с одним исключением: все ее элементы имеют размер по 64 бита, что позволяет оперировать со значениями, превышающими 4 Гб. Последний элемент, *ullAvailExtendedVirtual*, указывает размер незарезервированной памяти в самой большой области памяти виртуального адресного пространства вызывающего процесса. Этот элемент имеет смысл только для процессоров определенных архитектур при определенных конфигурациях.

Управление памятью на компьютерах с архитектурой NUMA

Как сказано в главе 7, процессоры компьютера с архитектурой NUMA используют память, установленною как на своей, так и на других платах. Однако обращение к «своей» памяти происходит намного быстрее, чем к «чужой». По умолчанию при выделении физической памяти потоком операционная система в целях повышения быстродействия пытается сопоставить

ей оперативную память, смонтированную на той же плате, что и процессор. И только при нехватке оперативной памяти для сопоставления может быть использована память с «чужой» платы.

Вызов функции *GlobalMemoryStatusEx* возвращает в параметре *ullAvailPhus* суммарное количество доступной памяти на всех платах. Узнать размер памяти, установленной на заданной плате, позволят следующая функция:

```
BOOL GetNumaAvailableMerooryNode(
    UCHAR uNode,
    PUL0NGL0NG pulAvailableBytes);
```

В переменную типа LONGLONG, на которую указывает параметр *pulAvailableBytes*, записывается объём памяти на плате, заданной параметром *uNode*. Чтобы узнать, на какой плате смонтирован процессор, достаточно вызвать функцию *GetNumaProcessorNode*:

```
BOOL WINAPI GetNumaProcessorNode(
    UCHAR Processor,
    PUCHAR NodeNumber);
```

Общее число установленных в NUMA-системе плат возвращает следующая функция:

```
BOOL GetNumaHighestNodeNumber(PULONG pulHighestNodeNumber);
```

Указав номер платы (от 0 до значения параметра *pulHighestNodeNumber*), можно получить список смонтированных на ней процессоров, вызвав следующую функцию:

```
BOOL GetNumaNodeProcessorNask(
    UCHAR uNode,
    PULONGLONG pulProcessorMask);
```

Параметр *uNode* — это числовой идентификатор платы; в переменную типа LONGLONG, на которую ссылается параметр *pulProcessorMask*, записывается битовая маска. Процессоры, смонтированные на заданной плате, соответствуют установленным битам в этой маске.

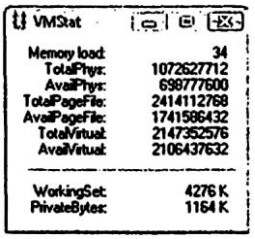
Как сказано выше, Windows старается, чтобы потоки использовали оперативную память «своей» платы, чтобы повысить производительность. Однако Windows также поддерживает функции для ручного управления распределением памяти между потоками (подробнее об этом — в главе 15).

Дополнительные сведения о NUMA в Windows см. в статьях MSDN «Application Software Considerations for NUMA-Based Systems» (http://www.microsoft.com/whdc/system/platform/server/datacenter//numa_isv.aspx) и «NUMA Support» (http://msdn2.microsoft.com/en-us/library/aa363804.aspx).

Программа-пример VMStat

Эта программа, «14 VMStat.exe» (14-VMStat.exe), выводит на экран окно с результатами вызова *GlobalMemoryStatus*. Информация в окне обновляется

каждую секунду, так что VMStat вполне пригодна дли мониторинга памяти в системе. Файлы исходного кода и ресурсов этой программы находятся в каталоге 14-VMStat внутри архива, доступного на веб-сайте поддержки этой книги. Окно этой программы после запуска в Windows Vista на машине с 1 Гб оперативной памяти показано ниже.



Элемент *dwMemoryLoad* (показываемый как Memory Load) позволяет оценить, насколько занята подсистема управления памятью. Это число может быть любым в диапазоне от 0 до 100. Кроме того, в будущих версиях операционных систем этот алгоритм почти наверняка придется модифицировать. Но, честно говоря, на практике от значения этого элемента толку немного.

Элемент *dwTotalPhys* (показываемый как TotalPhys) отражает общий объем физической (оперативной) памяти в байтах. На данной машине с 1 Гб оперативной памяти его значение составляет 1 072 627 712, что на 1 114 112 байта меньше 1 Гб. Причина, по которой *GlobalMemoryStatus* не сообщает о полном 1 Гб, кроется в том, что система при загрузке резервирует небольшой участок оперативной памяти, недоступный даже ядру. Этот участок никогда не сбрасывается на диск. А элемент *dwAvailPhys* (показываемый как AvailPhys) дает число байтов свободной физической памяти.

Элемент *dwTotalPageFile* (показываемый как TotalPageFile) сообщает максимальное количество байтов, которое может содержаться в страничном файле (файлах) на жестком диске (дисках). Хотя VMStat показывает, что текущий размер страничного файла составляет 2 414 112 768 байтов, система может варьировать его по своему усмотрению. Элемент *dwAvailPageFile* (показываемый как AvailPageFile) подсказывает, что в данный момент 1 741 586 432 байта в страничном файле свободно и может быть передано любому процессу.

Элемент *dwTotalVirtual* (показываемый как TotalVirtual) отражает общее количество байтов, отведенных под закрытое адресное пространство процесса. Значение 2 147 352 576 ровно на 128 Кб меньше 2 Гб. Два раздела недоступного адресного пространства — от 0x00000000 до 0x0000FFFF и от 0x7FFF0000 до 0x7FFFFFFF — как раз и составляют эту разницу в 128 Кб.

И, наконец, *dwAvailVirtual* (показываемый как AvailVirtual) — единственный элемент структуры, специфичный для конкретного процесса, вызывающего *GlobalMemoryStatus* (остальные элементы относятся исключительно к самой системе и не зависят от того, какой именно процесс вызывает эту функцию). При подсчете значения *dwAvailVirtual* функция суммирует размеры

всех свободных регионов в адресном пространстве вызывающего процесса. В данном случае его значение говорит о том, что в распоряжении программы VMStat имеется 2 106 437 632 байтов свободного адресного пространства. Вычтя из значения *dwTotalVirtual* величину *dwAvailVirtual*, вы получите 40 914 944 байтов — такой объем памяти VMStat зарезервировала в своем виртуальном адресном пространстве. Отдельного элемента, который сообщал бы количество физической памяти, используемой процессом в данный момент, не предусмотрено. Страницы адресного пространства процесса, загруженные в оперативную память, называются *рабочим набором* (working set) процесса. Следующая функция, объявленная в файле psapi.h, позволяет определить текущий и максимальный размер рабочего набора для процесса:

```
B00L GetProcessMemoryInfo(
    HANDLE hProcess,
    PPR0CESS_MEM0RY_C0UNTERS ppmc,
    DWORD cbSize);
```

Параметр *hProcess* — это описатель нужного вам процесса, у этого описателя должны быть права доступа PROCESS_QUERY_INFORMATION и PROCESS_VM_READ. Функция *GetCurrentProcess* возвращает для текущего процесса псевдоописатель, соответствующий этим требованиям. Параметр *ppmc* указывает на структуру PROCESS_MEMORY_COUNTERS_EX, размер которой задается параметром *cbSize*. Если *GetProcessMemoryInfo* возвращает TRUE, в элементы следующей структуры будут записаны сведения о заданном процессе:

```
typedef struct _PR0CESS_MEM0RY_C0UNTERS_EX {
    DWORD cb;
    DWORD PageFaultCount;
    SIZE_T PeakWorkingSetSize;
    SIZE_T WorkingSetSize;
    SIZE_T QuotaPeakPagedPoolUsage;
    SIZE_T QuotaPagedPoolUsage;
    SIZE_T QuotaPeakNonPagedPoolUsage;
    SIZE_T QuotaNonPagedPoolUsage;
    SIZE_T PageflleUsage;
    SIZE_T PeakPagefileUsage;
    SIZE_T PrlvateUsage;
} PROCESS_MEMORY_COUNTERS_EX,
*PPR0CESS_MEM0RY_C0UNTERS_EX;
```

Поле *WorkingSetSize* содержит число байтов RAM, используемых процессом, заданным параметром *hProcess* при вызове *GetProcessMemoryInfo*. Поле *PeakWorkingSetSize* содержит пиковое число байтов, занятых процессом в RAM, за весь период его работы.

Знать размер рабочего набора процесса чрезвычайно полезно, поскольку эта цифра отражает среднее количество оперативной памяти, необходимое

вашей программе. Чем меньше рабочий набор приложения, тем выше его производительность. Вероятно, вы знаете, что лучшая мера для повышения быстродействия Windows-приложений — наращивания размера оперативной памяти. Хотя Windows может хранить содержимое оперативной памяти в страничном файле, такая «оперативная» намять работает намного медленнее настоящей RAM. Чем больше размер физической RAM, тем меньше Windows использует страничный файл, а значит, производительность будет выше. Разработчики также могут управлять размером той части данных приложения, которая в каждый конкретный момент времени должна находиться в оперативной памяти. Чем меньше эта часть, тем выше производительность.

Помимо уменьшения рабочего набора, при «настройке» приложения желательно знать, сколько памяти оно явно выделяет для себя с помощью функций *new, malloc* и *VirtualAlloc*. Эту информацию предоставляет поле *PrivateUsage* (PrivateBytes). В остальных разделах этой главы я расскажу о функциях, позволяющих получить дополнительную информацию об адресном пространстве процесса.

Определение состояния адресного пространства

В Windows имеется функция, позволяющая запрашивать определенную информацию об участке памяти по заданному адресу (в пределах адресного пространства вызывающего процесса): размер, тип памяти и атрибуты защиты. В частности, с ее помощью программа VMMap выводит карты виртуальной памяти, с которыми мы познакомились в главе 13. Вот эта функция:

```
DW0RD VirtualQuery( LPCVOID pvAddress,
    PHEMORY_BASIC_INFORMATION pmbi,
    DWORD dwLength);
```

Парная ей функция, *VirtualQueryEx*, сообщает ту же информацию о памяти, но в другом процессе:

```
DW0R0 VirtualQueryEx(
    HANDLE hProcess,
    LPCVOID pvAddress,
    PMEMORY_BASIC_INFORMATION pmbi,
    DWORD dwLength);
```

Эти функции идентичны с тем исключением, что *VirtualQueryEx* принимает описатель процесса, об адресном пространстве которого вы хотите получить информацию. Чаще всего функцией *VirtualQueryEx* пользуются отладчики и системные утилиты — остальные приложения обращаются к *VirtualQuery*. При вызове *VirtualQuery(Ex)* параметр *pvAddress* должен содержать адрес виртуальной памяти, о которой вы хотите получить информацию. Параметр *pmbi* — это адрес структуры MEMORY_BASIC_INFORMATION,

которую надо создать перед вызовом функции. Данная структура определена в файле WinNT.h так:

```
typedef struct _MEMORY_BASIC.INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DW0R0 Type;
} MEMORY_BASIC_INFORMATION, *PHEMORY_BASIC_INFORMATION;
```

Параметр *dwLength* задает размер структуры MEMORY_BASIC_INFORMATION. Функция *VirtualQuery(Ex)* возвращает число байтов, скопированных в буфер.

Используя адрес, указанный вами в параметре *pvAddress*, функция *VirtualQuery(Ex)* заполняет структуру информацией о диапазоне смежных страниц, имеющих одинаковые состояние, атрибуты защиты и тип. Описание элементов структуры приведено в таблице ниже.

Табл. 144. Элементы структуры MEMORY_BASIC_INFORMATION

Элемент	Описание
BaseAddress	Сообщает то же значение, что и параметр *pvAddress*, но округленное до ближайшего меньшего адреса, кратного размеру страницы.
AllocationBase	Идентифицирует базовый адрес региона, включающего в себя адрес, указанный в параметре *pvAddress*.
AllocationProtect	Идентифицирует атрибут защиты, присвоенный региону при его резервировании.
RegionSize	Сообщает суммарный размер (в байтах) группы страниц, которые начинаются с базового адреса *BaseAddress* и имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *pvAddress*.
State	Сообщает состояние (MEM_FREE, MEM_RESERVE или MEM_ COMMГТ) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре pvAddress. При MEM_FREE элементы *AllocationBase*, *AllocationProtect*, *Protect* и *Type* содержат неопределенные значения, а при MEM_RESERVE неопределенное значение содержит элемент *Protect*.
Protect	Идентифицирует атрибут защиты (PAGE_*) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *pvAddress*.

Табл. 14-3. *(окончание)*

Элемент	Описание
Type	Идентифицирует тип физической памяти (MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE), связанной с группой смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре *pvAddress*. В Windows 98 этот элемент всегда даст MEM_PRIVATE.

Функция VMQuery

Начиная изучать архитектуру памяти в Windows, я пользовался функцией *VirtualQuery* как «поводырем». Если вы читали первое издание моей книги, то заметите, что программа VMMap была гораздо проще ее нынешней версии, представленной в следующем разделе. Прежняя была построена на очень простом цикле, из которого периодически вызывалась функция *VirtualQuery*, и для каждого вызова я формировал одну строку, содержавшую элементы структуры MEMORYJ-BASIC_№FORMATION. Изучая полученные дампы и сверяясь с документацией из SDK (в то время весьма неудачной), я пытался разобраться в архитектуре подсистемы управления памятью. Что ж, с тех пор я многому научился и теперь знаю, что функция *VirtualQuery* и структура MEMORY_BASIC_INFORMATION не дают полной картины.

Проблема в том, что в MEMORY_BASIC_INFORMATION возвращается отнюдь не вся информация, имеющаяся в распоряжении системы. Если вам нужны простейшие данные о состоянии памяти по конкретному адресу, *VirtualQuery* действительно незаменима. Она отлично работает, если вас интересует, передана ли по этому адресу физическая память и доступен ли он для операций чтения или записи. Но попробуйте с ее помощью узнать общий размер зарезервированного региона и количество блоков в нем или выяснить, не содержит ли этот регион стек потока, — ничего не выйдет.

Чтобы получать более полную информацию о памяти, я создал собственную функцию и назвал ее *VMQuery*:

```
BOOL VMQuery(
    HANDLE hProcess,
    LPCVOID pvAddress,
    PVMQUERY pVMQ);
```

По аналогии с *VirtualQueryEx* она принимает в *hProcess* описатель процесса, в *pvAddress* — адрес памяти, а в *pVMQ* — указатель на структуру, заполняемую самой функцией. Структура VMQUERY (тоже определенная мной) представляет собой вот что:

```
typedef struct {
    // Region information
    PVOID pvRgnBaseAddress;
    DWORD dwRgnProtection;                  // PAGE_*
```

```
    SIZE_T RgnSize;
    DWORD dwRgnStorage;             // MEM_*: Free, Image, Napped, Private
    DWORD dwRgnBlocks;
    DWORD dwRgnGuardBlks;           // если > 0, регион содержит стек потока
    BOOL bRgnIsAStack;              // TRUE, если регион содержит стек потока

    // Block information
    PVOID pvBlkBaseAddress;
    DWORD dwBlkProtection;          // PAGE_*
    SIZE_T BlkSize;
    DWORD dwBlkStorage;             // NEN_*: Free, Reserve, Image, Napped, Private
} VMQUERY, *PVMQUERY;
```

С первого взгляда заметно, что моя структура VMQUERY содержит куда больше информации, чем MEMORY_BASIC_INFORMATION. Она разбита (условно, конечно) на две части: в одной — информация о регионе, в другой — информация о блоке (адрес которого указан в параметре *pvAddress*). Элементы этой структуры описываются в следующей таблице.

Табл. 14-4. Элементы структуры VMQUERY

Элемент	Описание
pvRgnBaseAddress	Идентифицирует базовый адрес региона виртуального адресного пространства, включающего адрес, указанный в параметре *pvAddress*
dwRgnProtection	Сообщает атрибут защиты, присвоенный региону при его резервировании
RgnSize	Указывает размер (в байтах) зарезервированного региона
dwRgnStorage	Идентифицирует тип физической памяти, используемой группой блоков данного региона: MEM_FREE, MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE
dwRgnBlocks	Содержит значение — число блоков в указанном регионе
dwRgnGuardBlks	Указывает число блоков с установленным флагом атрибутов защиты PAGE_GUARD. Обычно это значение либо 0, либо 1. Если оно равно 1, то регион, скорее всего, зарезервирован под стек потока
fRgnIsAStack	Сообщает, есть ли в данном регионе стек потока. Результат определяется на основе взвешенной оценки, так как невозможно дать стопроцентной гарантии тому, что в регионе содержится стек
pvBlkBaseAddress	Идентифицирует базовый адрес блока, включающего адрес, указанный в параметре *pvAddress*
dwBlkProtection	Идентифицирует атрибут защиты блока, включающего адрес, указанный в параметре *pvAddress*
BlkSize	Содержит значение — размер блока (в байтах), включающего адрес, указанный в параметре *pvAddress*

Табл. 14-4. *(окончание)*

Элемент	Описание
dwBlkStorage	Идентифицирует содержимое блока, включающего адрес, указанный в параметре *pwAddress*. Принимает одно из значений: MEM_FREE, MEM_RESERVE, MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE.

Чтобы получить нею эту информацию, *VMQuerry*, естественно, приходится выполнять гораздо больше операции (в том числе многократно вызывать *VirtualQueryEx*), а потому она работает значительно медленнее *VirtualQueryEx*. Так что вы должны все тщательно взвесить, прежде чем остановить свой выбор на одной на этих функции. Если вам не нужна дополнительная информация, возвращаемая *VMQuery*, используйте *VirtualQuery* или *VirtualQueryEx*.

Листинг файла VMQuery.cpp (рис. 14-3) показывает, как я получаю и обрабатываю данные, необходимые для инициализации элементов структуры VMQUERY. (Файлы VMQuery.cpp и VMQuery.h содержатся в каталоге 14-VMMap на компакт-диске, прилагаемом к книге.) Чтобы не объяснять подробности обработки данных «на пальцах», я снабдил тексты программ массой комментарием, вольно разбросанных по всему коду.

```
VMOuery.cpp

/******************************************************************************
Module:  VMQuery.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..\CommonFiles\CmnHdr.h"     /* см. приложение A. */
#include <windowsx.h>
#include "VMQuery.h"

//////////////////////////////////////////////////////////////////////////////

// вспомогательная структура
typedef struct {
   SIZE_T RgnSize;
   DWORD  dwRgnStorage;     // MEM_*: Free, Image, Mapped, Private
   DWORD  dwRgnBlocks;
   DWORD  dwRgnGuardBlks;   // если > 0, в регионе содержится стек потока
   BOOL   bRgnIsAStack;     // TRUE, если в регионе содержится стек потока
} VMQUERY_HELP;


// глобальная статическая переменная, содержащая значение -
// гранулярность выделения
```

```
// памяти на данном типе процессора; инициализируется при первом
// вызове VWQuery
static DWORD gs_dwAllocGran = 0;

//////////////////////////////////////////////////////////////////////////////

// эта функция проходит по всем блокам в регионе, возвращая результаты в виде
    VMQUERY_HELP
static BOOL VMQueryHelp(HANDLE hProcess, LPCVOID pvAddress,
   VMQUERY_HELP *pVMQHelp) {

   ZeroMemory(pVMQHelp, sizeof(*pVMQHelp));

   // получаем базовый адрес региона, включающего переданный адрес памяти
   MEMORY_BASIC_INFORMATION mbi;
   BOOL bOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi))
      == sizeof(mbi));

   if (!bOk)
      return(bOk);   // неверный адрес памяти, сообщаем об ошибке

   // проходим по региону, начиная с его базового адреса
   // (который никогда не изменяется)
   PVOID pvRgnBaseAddress = mbi.AllocationBase;

   // начинаем с первого блока в регионе
   // (соответствующая переменная будет изменяться в цикле)
   PVOID pvAddressBlk = pvRgnBaseAddress;

   // запоминаем тип физической памяти, переданной данному блоку
   pVMQHelp->dwRgnStorage = mbi.Type;

   for (;;) {
      // получаем информацию о текущем блоке
      bOk = (VirtualQueryEx(hProcess, pvAddressBlk, &mbi, sizeof(mbi))
         == sizeof(mbi));
      if (!bOk)
         break;   // не удалось получить информацию; прекращаем цикл

      // проверяем, принадлежит ли текущий блок запрошенному региону
      if (mbi.AllocationBase != pvRgnBaseAddress)
         break;   // блок принадлежит следующему региону; прекращаем цикл

      // блок принадлежит запрошенному региону

      pVMQHelp->dwRgnBlocks++;           // добавляем к региону еще один блок
```

```
      pVMQHelp->RgnSize += mbi.RegionSize; // увеличиваем счетчик блоков
                                           // в этом регионе на 1

      // если блок имеет флаг PAGE_GUARD, добавляем 1 к счетчику блоков
      // с этим флагом
      if ((mbi.Protect & PAGE_GUARD) == PAGE_GUARD)
         pVMQHelp->dwRgnGuardBlks++;

      // делаем наиболее вероятное предположение о типе физической памяти,
      // переданной данному блоку. Стопроцентной гарантии дать нельзя,
      // потому что некоторые блоки могли быть преобразованы MEM_IMAGE
      // в MEM_PRIVATE или из MEM_MAPPED в MEM_PRIVATE; MEM_PRIVATE в любой
      // момент может быть замещен наMEM_IMAGE или MEM_MAPPED.
      if (pVMQHelp->dwRgnStorage == MEM_PRIVATE)
         pVMQHelp->dwRgnStorage = mbi.Type;

      // получаем адрес следующего блока
      pvAddressBlk = (PVOID) ((PBYTE) pvAddressBlk + mbi.RegionSize);
   }

   // обследовать регион, думаем: не стек ли это?
   // Windows Vista: да - если в регионе содержится хотя бы 1 блок
   // с флагом PAGE_GUARD
   pVMQHelp->bRgnIsAStack = (pVMQHelp->dwRgnGuardBlks > 0);

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

BOOL VMQuery(HANDLE hProcess, LPCVOID pvAddress, PVMQUERY pVMQ) {

   if (gs_dwAllocGran == 0) {
      // если это первый вызов, надо выяснить гранулярность
      // выделения памяти в данной системе
      SYSTEM_INFO sinf;
      GetSystemInfo(&sinf);
      gs_dwAllocGran = sinf.dwAllocationGranularity;
   }

   ZeroMemory(pVMQ, sizeof(*pVMQ));

   // получаем MEMORY_BASIC_INFORMATION для переданного адреса
   MEMORY_BASIC_INFORMATION mbi;
   BOOL bOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi))
      == sizeof(mbi));

   if (!bOk)
```

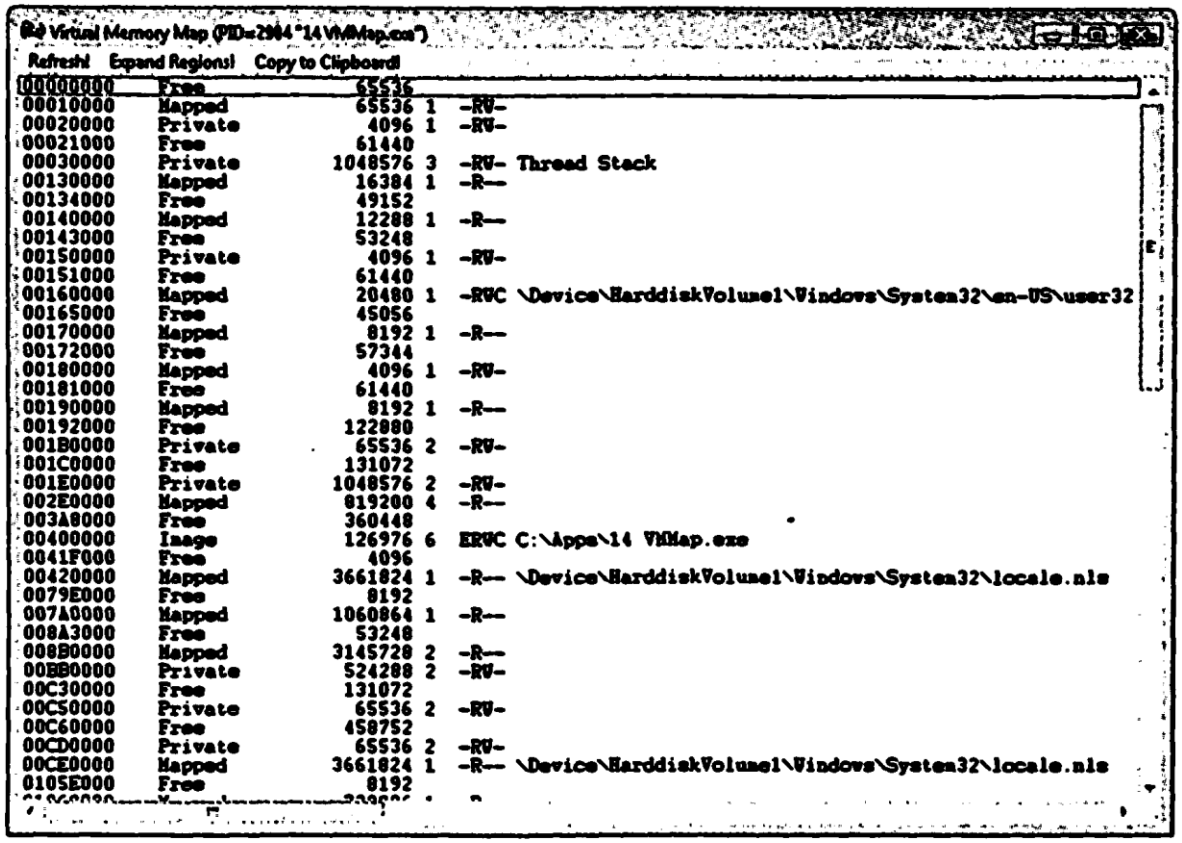
```
      return(bOk);   // неверный адрес памяти, сообщаем об ошибке

   // структура MEMORY_BASIC_INFORMATION содержит действительную
   // информацию - пора заполнить элементы нашей структуры VMQUERY

   // во-первых, заполним элементы, описывающие состояния блока;
   // данные по региону получим позже
   switch (mbi.State) {
      case MEM_FREE:                    // свободный блок (незарезервирован-
    ный)
         pVMQ->pvBlkBaseAddress = NULL;
         pVMQ->BlkSize = 0;
         pVMQ->dwBlkProtection = 0;
         pVMQ->dwBlkStorage = MEM_FREE;
         break;

      case MEM_RESERVE:                 // зарезервированный блок, которому
                                        // не передана физическая память
         pVMQ->pvBlkBaseAddress = mbi.BaseAddress;
         pVMQ->BlkSize = mbi.RegionSize;

         // для блока, которому не передана физическая память,
         // элемент mbi.Protect недействителен. Поэтому мы покажем,
         // что зарезервированный блок унаследовал атрибут защиты
         // того региона, в котором он содержатся
         pVMQ->dwBlkProtection = mbi.AllocationProtect;
         pVMQ->dwBlkStorage = MEM_RESERVE;
         break;

      case MEM_COMMIT:                  // зарезервированный блок, которому
                                        // передана физическая память
         pVMQ->pvBlkBaseAddress = mbi.BaseAddress;
         pVMQ->BlkSize = mbi.RegionSize;
         pVMQ->dwBlkProtection = mbi.Protect;
         pVMQ->dwBlkStorage = mbi.Type;
         break;

      default:
          DebugBreak();
          break;
   }

   // теперь заполняем элементы, относящиеся к региону
   VMQUERY_HELP VMQHelp;
   switch (mbi.State) {
      case MEM_FREE:       // свободный блок (незарезервированный)
         pVMQ->pvRgnBaseAddress = mbi.BaseAddress;
```

```
         pVMQ->dwRgnProtection  = mbi.AllocationProtect;
         pVMQ->RgnSize          = mbi.RegionSize;
         pVMQ->dwRgnStorage     = MEM_FREE;
         pVMQ->dwRgnBlocks      = 0;
         pVMQ->dwRgnGuardBlks   = 0;
         pVMQ->bRgnIsAStack     = FALSE;
         break;

      case MEM_RESERVE:                  // зарезервированный блок, которому
                                         // не передана физическая память
         pVMQ->pvRgnBaseAddress = mbi.AllocationBase;
         pVMQ->dwRgnProtection  = mbi.AllocationProtect;

         // чтобы получить полную информацию по региону, нам придется
         // пройти по всем его блокам
         VMQueryHelp(hProcess, pvAddress, &VMQHelp);

         pVMQ->RgnSize          = VMQHelp.RgnSize;
         pVMQ->dwRgnStorage     = VMQHelp.dwRgnStorage;
         pVMQ->dwRgnBlocks      = VMQHelp.dwRgnBlocks;
         pVMQ->dwRgnGuardBlks   = VMQHelp.dwRgnGuardBlks;
         pVMQ->bRgnIsAStack     = VMQHelp.bRgnIsAStack;
         break;

      case MEM_COMMIT:                   // зарезервированный блок, которому
                                         // передана физическая память
         pVMQ->pvRgnBaseAddress = mbi.AllocationBase;
         pVMQ->dwRgnProtection  = mbi.AllocationProtect;

         // чтобы получить полную информацию по региону, нам придется
         // пройти по всем его блокам
         VMQueryHelp(hProcess, pvAddress, &VMQHelp);

         pVMQ->RgnSize          = VMQHelp.RgnSize;
         pVMQ->dwRgnStorage     = VMQHelp.dwRgnStorage;
         pVMQ->dwRgnBlocks      = VMQHelp.dwRgnBlocks;
         pVMQ->dwRgnGuardBlks   = VMQHelp.dwRgnGuardBlks;
         pVMQ->bRgnIsAStack     = VMQHelp.bRgnIsAStack;
         break;

      default:
          DebugBreak();
          break;
   }

   return(bOk);
```

```
}

//////////////////////////////// End of File //////////////////////////////////
```

Программа-пример VMMap

Эта программа, «14-VMMap.exe», просматривает свое адресное пространство и показывает содержащиеся в нем регионы и блоки, присутствующие в регионах. Файлы исходного кода и ресурсов этой программы находятся в каталоге 14-VMMap внутри архива, доступного на сайте поддержке этой книги. После запуска VMMap на экране появляется следующее окно.



Карты виртуальной памяти, представленные в главе 13 в таблицах 13-2 и 13-3, созданы с помощью именно этой программы. Каждый элемент в списке — результат вызова моей функции *VMQuery*. Основной цикл программы VMMap (в функции *Refresh*) выглядит так:

```
BOOL bOk = TRUE;
PVOID pvAddress = NULL;
…

while (bOk) {
    VMQUERY vmq;
    bOk = VMQuery(hProcess, pvAddress, &vmq);
    if (bOk) {
        // формируем строку для вывода на экран
        // и добавляем ее в окно списка
```

```
        TCHAR szLine[1024];
        ConstructRgnInfoLine(hProcess, &vroq, szLlne, slzeof(szLine));
        ListBox_AddSt ring(hWndLB. szLine);

        if (bExpandRegions) {
            for (DWORD dwBlock = 0; bOk && (dwBlock < vmq.dwRgnBlocks);
                dwBlock++) {

                ConstructBlkInfoLine(&vmq. szLlne, sizeof(szLine));
                ListBox_AddString(hWndLB, szLine);

                // получаем адрес следующего региона
                pvAddress = ((PBYTE) pvAddress + vroq.BlkSize);
                if (dwBlock < vmq.dwRgnBlocks - 1) {
                // нельзя запрашивать информацию о памяти за последним блоком
                    bOk = VMQuery(hProcess, pvAddress, &vroq);
                }
            }
        }

        // получаем адрес следующего региона
        pvAddress = ((PBYTE) vroq.pvRgnBaseAddress + vmq.RgnSize);
    }
}
```

Этот цикл начинает работу с виртуального адреса NULL и заканчивается, когда *VMQuery* возвращает FALSE, что указывает на невозможность дальнейшего просмотра адресного пространства процесса. На каждой итерации цикла вызывается функция *ConstructRgnInfoLine*; она заполняет символьный буфер информацией о регионе. Потом эти данные вносятся в список.

В основной цикл вложен еще один цикл — он позволяет получать информацию о каждом блоке текущего региона. На каждой итерации из данного цикла вызывается функция *ConstructBlkInfoLine*, заполняющая символьный буфер информацией о блоках региона. Эти данные тоже добавляются к списку. В общем, с помощью функции *VMQpery* просматривать адресное пространство процесса очень легко.

Если запустить VMMap на компьютере с Windows Vista после перезагрузки (или сравнить результаты программы на разных компьютерах с Vista), можно заметить, что разные DLL-библиотеки каждый раз загружаются по разным адресам. Так работает новая функция Windows под названием Address Space Layout Randomization (ASLR). Цель случайного выбора базовых адресов — затруднить хакерам поиск известных DLL в памяти, чтобы они не смогли использовать их в своих целях.

Например, хакеры часто используют переполнение буфера или стека, чтобы вызвать стандартную функцию из системной DLL В системе с ASLR у хакера только один шанс из 256 (а то и меньше) найти нужную ему фун-

кцию по стандартному адресу. В результате хакерам будет намного сложнее воспользоваться ошибками из-за переполнения для обхода защиты.

При загрузке DLL ядро изменяет и выравнивает ее базовый адрес, после чего всем процессам передается уже измененный базовый адрес этой библиотеки. Это повышает эффективность использования памяти, поскольку освобождает от необходимости выравнивания адресов для каждого процесса по отдельности.

> **Примечание.** В Visual Studio 2005 SP1 и выше вы можете использовать ASLR для своих DLL- и EXE-файлов, включив ключ */dynamicbase* при компоновке. Я также рекомендую использовать этот ключ, если ваш модуль загружается по адресу, отличному от базового, — так процессы получат уже выровненный адрес вашей библиотеки, что повысит эффективность использования памяти.

Оглавление

ГЛАВА 15

Использование виртуальной памяти в приложениях

В Windows три механизма работы с памятью:

- виртуальная память — наиболее подходящая для операций с большими массивами объектов или структур;
- проецируемые в память файлы — наиболее подходящие для операций с большими потоками данных (обычно из файлов) и для совместного использования данных несколькими процессами на одном компьютере;
- кучи — наиболее подходящие для работы с множеством малых объектов.

В этой главе мы обсудим первый метод — виртуальную память. Остальные два метода (проецируемые в память файлы и кучи) рассматриваются соответственно в главах 17 и 18.

Функции, работающие с виртуальной памятью, позволяют напрямую резервировать регион адресного пространства, передавать ему физическую память (из страничного файла) и присваивать любые допустимые атрибуты защиты.

Резервирование региона в адресном пространстве

Для этого предназначена функция *VirtualAlloc:*

```
PVOID VirtualAlloc(
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD fdwAllocationType,
    DWORD fdwProtect);
```

В первом параметре, *pvAddress,* содержится адрес памяти, указывающий, где именно система должна зарезервировать адресное пространство. Обычно

в этом параметре передают NULL, тем самым сообщая функции *VirtualAlloc,* что система, ведущая учет свободных областей, должна зарезервировать регион там, где, по ее мнению, будет лучше. Поэтому нет никаких гарантий, что система станет резервировать регионы, начиная с нижних адресов или, наоборот, с верхних. Однако с помощью флага MEM_TOP_DOWN (о нем речь впереди) вы можете сказать свое веское слово.

Для большинства программистов возможность выбора конкретного адреса резервируемого региона — нечто совершенно новое. Вспомните, как это делалось раньше: операционная система просто находила подходящий по размеру блок памяти, выделяла этот блок и возвращала его адрес. Но поскольку каждый процесс владеет собственным адресным пространством, у вас появляется возможность указывать операционной системе желательный базовый адрес резервируемого региона.

Допустим, нужно выделить регион, начиная с «отметки» 50 Мб в адресном пространстве процесса. Тогда параметр *pvAddress* должен быть равен 52 428 800 (50 x 1024 x 1024). Если по этому адресу можно разместить регион требуемого размера, система зарезервирует его и вернет соответствующий адрес. Если же по этому адресу свободного пространства недостаточно или просто нет, система не удовлетворит запрос, и функция *VirtualAlloc* вернет NULL. Адрес, передаваемый в *pv Address,* должен укладываться в границы раздела пользовательского режима вашего процесса, так как иначе *VirtualAlloc* потерпит неудачу и вернет NULL.

Как я уже говорил в главе 13, регионы всегда резервируются с учетом гранулярности выделения памяти (64 Кб для существующих реализаций Windows). Поэтому, если вы попытаетесь зарезервировать регион по адресу 19 668 992 (300 x 65 536 x 8192), система округлит этот адрес до ближайшего меньшего числа, кратного 64 Кб, и на самом деле зарезервирует регион по адресу 19 660 800 (300 x 65 536).

Если *VirtualAlloc* в состоянии удовлетворить запрос, она возвращает базовый адрес зарезервированного региона. Если параметр *pvAddress* содержал конкретный адрес, функция возвращает этот адрес, округленный при необходимости до меньшей величины, кратной 64 Кб.

Второй параметр функции *VirtualAlloc* — *dwSize* — указывает размер резервируемого региона в байтах. Поскольку система резервирует регионы только порциями, кратными размеру страницы, используемой данным процессором, то попытка зарезервировать, скажем, 62 Кб даст регион размером 64 Кб (если размер страницы составляет 4,8 или 16 Кб).

Третий параметр, *fdwAllocationType,* сообщает системе, что именно вы хотите сделать: зарезервировать регион или передать физическую память. (Такое разграничение необходимо, поскольку *VirtualAlloc* позволяет не только резервировать регионы, но и передавать им физическую память.) Поэтому, чтобы зарезервировать регион адресного пространства, в этом параметре нужно передать идентификатор MEM_RESERVE.

Если вы хотите зарезервировать регион и не собираетесь освобождать его в ближайшее время, попробуйте выделить его в диапазоне самых старших — насколько это возможно — адресов. Тогда регион не окажется где-нибудь в середине адресного пространства процесса, что позволит не допустить вполне вероятной фрагментации этого пространства. Чтобы зарезервировать регион по самым старшим адресам, при вызове функции *VirtualAlloc* в параметре *pwAddress* передайте NULL, а в параметре *fdwAllocationType* — флаг MEM_RESERVE, скомбинированный с флагом MEM_TOP_DOWN.

Последний параметр, *fdwProtect,* указывает атрибут защиты, присваиваемый региону. Заметьте, что атрибут защиты, связанный с регионом, не влияет на переданную память, отображаемую на этот регион. Но если ему не передана физическая память, то — какой бы атрибут защиты у него ни был — любая попытка обращения по одному из адресов в этом диапазоне приведет к нарушению доступа для данного потока.

Резервируя регион, присваивайте ему тот атрибут защиты, который будет чаще всего использоваться с памятью, передаваемой региону. Скажем, если вы собираетесь передать региону физическую память с атрибутом защиты PAGE_READWRITE (этот атрибут самый распространенный), то и резервировать его следует с тем же атрибутом. Система работает эффективнее, когда атрибут зашиты региона совпадает с атрибутом зашиты передаваемой памяти.

Вы можете использовать любой из следующих атрибутов защиты: PAGE_NOACCESS, PAGE_READWRITE, PAGE_READONLY, PAGE_EXECUTE, PAGE_EXECUTE_READ или PAGE_ EXECUTE_READWRITE. Но указывать атрибуты PAGE_WRITECOPY или PAGE_EXECUTE_WRITECOPY нельзя: иначе функция *VirtualAlloc* не зарезервирует регион и вернет NULL. Кроме того, при резервировании региона флаги PAGE_GUARD, PAGE_WRITECOMBINE или PAGE_NOCACHE применять тоже нельзя — они присваиваются только передаваемой памяти.

> **Примечание.** Повысить производительность приложений, работающих на NUMA-компьютерах (см. главы 4 и 14), можно путем. «Привязки» виртуальной памяти процесса к физической RAM, смонтированной на определенной плате такого компьютера. Для этого служит следующая функция:

```
PVOID VirtualAllocExNuma(
    HANDLE hProcess,
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD fdwAllocationType,
    DWORD fdwProtect,
    DWORD dwPreferredNumaNode);
```

> Эта функция отличается от *VirtualAlloc* лишь парой дополнительных параметров, hProcess и *dwPreferredNumaNode*. Параметр *hProcess* идентифицирует процесс, которому следует передать физическую память (ма-

нипулировать виртуальной памятью текущего процесса позволяет функция *GetCurrentProcess*). Параметр *dwPreferredNumaNode* задает плату, на которой смонтирована физическая оперативная память, предназначенная для сопоставления с виртуальной памятью заданного процесса. Подробнее о Windows-функциях для работы с памятью и процессорами на NUMA-компьютерах см. в предыдущей главе.

Передача памяти зарезервированному региону

Зарезервировав регион, вы должны, прежде чем обращаться по содержащимся в нем адресам, передать ему физическую память. Система выделяет региону физическую память из страничного файла на жестком диске. При этом она, разумеется, учитывает свойственный данному процессору размер страниц и передает ресурсы постранично.

Для передачи физической памяти вызовите *VirtualAlloc* еще раз, указав в параметре *fdwAllocationType* не MEM_RESERVE, а MEM_COMMIT. Обычно указывают тот же атрибут защиты (чаще всего PAGE_READWRITE), что и при резервировании региона, хотя можно задать и другой.

Затем *обязательно* сообщите функции *VirtualAlloc,* по какому адресу и сколько физической памяти следует передать. Для этого в параметр *pvAddress* запишите желательный адрес, а в параметр *dwSize* — размер физической памяти в байтах. Передавать физическую память сразу всему региону необязательно.

Посмотрим, как это делается на практике. Допустим, программа работает на процессоре x86 и резервирует регион размером 512 Кб, начиная с адреса 5 242 880. Затем вы передаете физическую память блоку размером 6 Кб, отстоящему от начала зарезервированного региона на 2 Кб. Тогда вызовите *VirtualAlloc* с флагом MEM_COMMIT так:

```
VirtttalAlloc((PV0ID) (5242880 + (2 * 1024)), 6 * 1024,
   MEM_C0MMIT, PAGE_READWRITE);
```

В этом случае система передаст 8 Кб физической памяти в диапазоне адресов от 5 242 880 до 5 251 071 (т. е. 5 242 880 + 8 Кб - 1 байт), и обе переданные страницы получат атрибут зашиты PAGE_READWRITE. Страница является минимальной единицей памяти, которой можно присвоить собственные атрибуты защиты. Следовательно, в регионе могут быть страницы с разными атрибутами защиты (скажем, одна — с атрибутом PAGE_READWRITE, другая — с атрибутом PAGE_READONLY).

Резервирование региона с одновременной передачей физической памяти

Иногда нужно одновременно зарезервировать регион и передать ему физическую память. В таком случае *VirtualAlloc* можно вызвать следующим образом:

```
PVOID pvMem = VirtualAlloc(NULL, 99 * 1024,
   MEM_RESERVE | MEM_C0MMIT, PAGE_READWRITE);
```

Этот вызов содержит запрос на выделение региона размером 99 Кб и передачу ему 99 Кб физической памяти. Обрабатывая этот запрос, система сначала просматривает адресное пространство вашего процесса, пытаясь найти непрерывную незарезервированную область размером не менее 100 Кб (на машинах с 4-килобайтовыми страницами) или 104 Кб (на машинах с 8-килобайтовыми страницами).

Система просматривает адресное пространство потому, что в *pvAddress* указан NULL. Если бы он содержал конкретный адрес памяти, система проверила бы только его — подходит ли по размеру расположенное за ним адресное пространство. Окажись он недостаточным, функция *VirtualAlloc* вернула бы NULL.

Если системе удается зарезервировать подходящий регион, она передает ему физическую память. И регион, и переданная память получают один атрибут защиты — в данном случае PAGE_READWRITE.

Windows поддерживает большие страницы, что ускоряет манипуляции с большими областями памяти. При выделении памяти вместо размера страницы (гранулярности), возвращаемого функцией *GetSystemInfo* в поле *dwPageSize* структуры SYSTEM_INFO, можно использовать страницы большего размера. Определить этот размер позволяет следующая функция:

```
SIZE_T GetLargePageMinimum();
```

Заметьте, что GetLargePageMinimum вернет 0, если процессор не поддерживает большие страницы. Большие страницы можно использовать для выделения блоков памяти, размер которых как минимум равен результату вызова *GetLargePageMinimum.* Для выделения памяти достаточно вызвать *VirtualAlloc* с флагом MEM_LARGE_PAGE, скомбинированным (операцией OR) с значением параметра *fdwAllocationType.* Также необходимо соблюдать следующие условия:

- Размер выделяемого блока должен быть кратен значению *dwSize,* которое возвращает функция *GetLargePageMinimum*;
- При вызове VirtualAlloc флаг MEM_RESERVE | MEM_COMMIT необходимо скомбинировать (операцией OR) с параметром *fdwAllocationType.* Другими словами, резервирование и передача памяти осуществляется только одновременно. Нельзя зарезервировать регион, а потом выборочно передавать ему физическую память;

- При выделении памяти необходимо передавать флаг PAGE_READWRITE в параметре *fdwProtect* функции *VirtualAUoc*.

Windows считает память, выделенную с флагом MEM_LARGE_PAGE, невыгружаемой и всегда хранит ее в оперативной памяти. Это одна из причин, по которым выделенная таким образом память работает быстрее. Однако из-за дефицита физической памяти для вызова *VirtualAlloc* с флагом MEM_LARGE_PAGE необходимо наличие права Lock Pages In Memory, иначе вызов закончится неудачей. По умолчанию такого права нет ни у одного пользователя или группы. Чтоб интерактивное приложение смогло работать с большими страницами памяти, администратор должен предварительно предоставить это право пользователю, от имени которого данное приложение будет запущено.

Это делается так:

1. Щелкните меню Start | Administrative Tools | Local Security Policy.
2. На левой панели окна консоли раскройте двойным щелчком элементы Security Settings и Local Policies. Выберите элемент User Rights Assignment.
3. На правой панели щелкните атрибут) Lock Pages In Memory.
4. В меню Action выберите элемент Properties — откроется окно Lock Pages In Memory Properties. Щелкните кнопку Add User Or Group и в окне Select Users Or Groups добавьте учетные записи пользователей и групп, которым следует предоставить право Lock Pages In Memory. Закройте все окна, щелкая кнопку ОК.

Права назначаются пользователям при входе в систему. Если вы предоставили право Lock Pages la Memory самому себе, необходимо выйти из системы и снова войти в нее, чтобы это действие возымело эффект. Кстати, приложение, желающее работать большими страницами, нуждается и в повышении привилегий (см. стр. 110).

Наконец, функция *VirtualAlloc* возвращает виртуальный адрес этого региона, который потом записывается в переменную *pvMem*. Если же система не найдет в адресном пространстве подходящую область или не сумеет передать ей физическую память, *VirtualAUoc* вернет NULL.

Конечно, при резервировании региона с одновременной передачей ему памяти можно указать в параметре *pvAddress* конкретный адрес или запросить систему подобрать свободное место в верхней части адресного пространства процесса. Последнее реализуют так: в параметр *pvAddress* заносят NULL, а значение параметра *fdwAllocationType* комбинируют с флагом MEM_TOP_DOWN.

В какой момент региону передают физическую память

Допустим, вы разрабатываете программу — электронную таблицу, которая поддерживает до 200 строк при 256 колонках. Для каждой ячейки необходима своя структура CELLDATA, описывающая ее (ячейки) содержимое.

Простейший способ работы с двухмерной матрицей ячеек, казалось бы, -взять и объявить в программе такую переменную:

```
CELLDATA CellData [200][256];
```

Но если размер структуры CELLDATA будет хотя бы 128 байтов, матрица потребует 6 553 600 (200 x 256 x 128) байтов физической памяти. Не многовато ли? Тем более что большинство пользователей заполняет данными всего несколько ячеек. Выходит, матрицы здесь крайне неэффективны.

Поэтому электронные таблицы реализуют на основе других методов управления структурами данных, используя, например, связанные списки. В этом случае структуры CELLDATA создаются только для ячеек, содержащих какие-то данные. И поскольку большая часть ячеек в таблице остается незадействованной, вы экономите колоссальные объемы памяти. Но это значительно усложняет доступ к содержимому ячеек. Чтобы, допустим, выяснить содержимое ячейки на пересечении строки 5 и колонки 10, придется пройти по всей цепочке связанных списков. В итоге метод связанных списков работает медленнее, чем метод, основанный на объявлении матрицы.

К счастью, виртуальная память позволяет найти компромисс между «лобовым» объявлением двухмерной матрицы и реализацией связанных списков. Тем самым можно совместить простоту и высокую скорость доступа к ячейкам, предлагаемую «матричным» методом, с экономным расходованием памяти, заложенным в метод связанных списков.

Вот что надо сделать в своей программе.

1. Зарезервировать достаточно большой регион, чтобы при необходимости в него мог поместиться весь массив структур CELLDATA. Для резервирования региона физическая память не нужна.
2. Когда пользователь вводит данные в ячейку, вычислить адрес в зарезервированном регионе, по которому должна быть записана соответствующая структура CELLDATA. Естественно, физическая память на этот регион пока не отображается, и поэтому любое обращение к памяти по данному адресу вызовет нарушение доступа.
3. Передать по адресу, полученному в п. 2, физическую память, необходимую для размещения одной структуры CELLDATA. (Так как система допускает передачу памяти отдельным частям зарезервированного региона, в нем могут находиться и отображенные, и не отображенные на физическую память участки.)
4. Инициализировать элементы новой структуры CELLDATA.

Теперь, спроецировав физическую память на нужный участок зарезервированного региона, программа может обратиться к нему, не вызвав при этом нарушения доступа. Таким образом, метод, основанный на использовании виртуальной памяти, самый оптимальный, поскольку позволяет передавать физическую память только по мере ввода данных в ячейки электронной таблицы. И ввиду того, что большая часть ячеек в электронной таблице обычно

пуста, то и большая часть зарезервированного региона физическую память не получает.

Но при использовании виртуальной памяти все же возникает одна проблема: приходится определять, когда именно зарезервированному региону надо передавать физическую память. Если пользователь всего лишь редактирует данные, уже содержащиеся в ячейке, в передаче физической памяти необходимости нет — это было сделано в момент первого заполнения ячейки.

Нельзя забывать и о размерности страниц памяти. Попытка передать физическую память для единственной структуры CELLDATA (как в п. 2 предыдущего списка) приведет к передаче полной страницы памяти. Но в этом, как ни странно, есть свое преимущество: передав физическую память под одну структуру CELLDATA, вы одновременно выделите ее и следующим структурам CELLDATA. Когда пользователь начнет заполнять следующую ячейку (а так обычно и бывает), вам, может, и не придется передавать дополнительную физическую память.

Определить, надо ли передавать физическую память части региона, можно четырьмя способами.

- Всегда пытаться передавать физическую память. Вместо того чтобы проверять, отображен данный участок региона на физическую память или нет, заставьте программу передавать память при каждом вызове функции *VirtualAlloc*. Ведь система сама делает такую проверку и, если физическая память спроецирована на данный участок, повторной передачи не допускает. Это простейший путь, но при каждом изменений структуры CELLDATA придется вызывать функцию *VirtualAlloc,* что, естественно, скажется на скорости работы программы.
- Определять (с помощью *ViriuialQuery),* передана ли уже физическая память адресному пространству, содержащему структуру CELLDATA. Если да, больше ничего не делать; нет — вызвать *VirtualAlloc* для передачи памяти. Этот метод на деле еще хуже, чем первый: он не только замедляет выполнение, но и увеличивает размер программы из-за дополнительных вызовов *VirtualQuery*.
- Вести учет, каким страницам передана физическая память, а каким — нет. Это повысит скорость работы программы: вы избежите лишних вызовов *VirtualAlloc,* а программа сможет — быстрее, чем система — определять, передана ли память. Недостаток этого метода в том, что придется отслеживать передачу страниц; иногда это просто, но может быть и очень сложно — все зависит от конкретной задачи.
- Самое лучшее — использовать структурную обработку исключений (SEH). SEH — одно из средств операционной системы, с помощью которого она уведомляет приложения о возникновении определенных событий. В общем и целом, вы добавляете в программу обработчик исключений, после чего любая попытка обращения к участку, которому не передана физическая память, заставляет систему уведомлять программу

о возникшей проблеме. Далее программа передает память нужному участку и сообщает системе, что та должна повторить операцию, вызвавшую исключение. На этот раз доступ к памяти пройдет успешно, и программа, как ни в чем не бывало, продолжит работу. Таким образом, ваша задача заметно упрощается (а значит, упрощается и код); кроме того, программа, не делая больше лишних вызовов, выполняется быстрее. Но подробное рассмотрение механизма структурной обработки исключений мы отложим до глав 23,24 и 25. Программа-пример Spreadsheet «главе 25 продемонстрирует именно этот способ использования виртуальной памяти.

Возврат физической памяти и освобождение региона

Для возврата физической памяти, отображенной на регион, или освобождения всего региона адресного пространства используется функция *VirtualFree:*

```
BOOL VirtualFree(
    LPVOID pvAddress,
    SIZE_T dwSize,
    DWORD fdwFreeType);
```

Рассмотрим простейший случай вызова этой функции — для освобождения зарезервированного региона. Когда процессу больше не нужна физическая память, переданная региону, зарезервированный регион и всю связанную с ним физическую память можно освободить единственным вызовом *VirtualFree.*

В этом случае в параметр *pvAddress* надо поместить базовый адрес региона, т. е. значение, возвращенное функцией *VirtualAlloc* после резервирования данного региона. Системе известен размер региона, расположенного по указанному адресу, поэтому в параметре *dwSize* можно передать 0. Фактически вы даже обязаны это сделать, иначе вызов *VirtualFree* не даст результата. В третьем параметре *(fdwFreeType)* передайте идентификатор MEM_RELEASE; это приведет к возврату системе всей физической памяти, отображенной на регион, и к освобождению самого региона. Освобождая регион, вы должны освободить и зарезервированное под него адресное пространство. Нельзя выделить регион размером, допустим, 128 Кб, а потом освободить только 64 Кб: надо освобождать все 128 Кб.

Если вам нужно, не освобождая регион, вернуть в систему часть физической памяти, переданной региону, для этого тоже следует вызвать *VirtualFree.* При этом ее параметр *pvAddress* должен содержать адрес, указывающий на первую возвращаемую страницу. Кроме того, в параметре *dwSize* задайте количество освобождаемых байтов, а в параметре *fdwFreeType* — идентификатор MEMDECOMMIT.

Как и передача, возврат памяти осуществляется с учетом размерности страниц. Иначе говоря, задание адреса, указывающего на середину страни-

цы, приведет к возврату всей страницы. Разумеется, то же самое произойдет, если суммарное значение параметров *pvAddress* и *dwSize* выпадет на середину страницы. Так что системе возвращаются все страницы, попадающие в диапазон от *pvAddress* до *pvAddress + dwSize.*

Если же *dwSize* равен 0, а *pvAddress* указывает на базовый адрес выделенного региона, *VirtualFree* вернет системе весь диапазон выделенных страниц. После возврата физической памяти освобожденные страницы доступны любому другому процессу, а попытка обращения к адресам, уже не связанным с физической памятью, приведет к нарушению доступа.

В какой момент физическую память возвращают системе

На практике уловить момент, подходящий для возврата памяти, — штука непростая. Вернемся к примеру с электронной таблицей. Если программа работает на машине с процессором x86, размер каждой страницы памяти — 4 Кб, т. е. на одной странице умещается 32 (4096 /128) структуры CELLDATA. Если пользователь удаляет содержимое элемента *CellData[0][1],* вы можете вернуть страницу памяти, но только при условии, что ячейки в диапазоне от *CellData[0][0]* до *CellData[0][31]* тоже не используются. Как об этом узнать? Проблема решается несколькими способами.

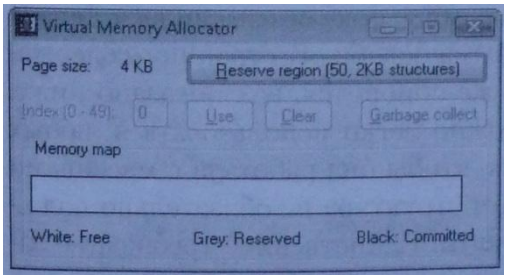
- Несомненно, простейший выход — сделать структуру CELLDATA такой, чтобы она занимала ровно одну страницу. Тогда, как только данные в какой-либо из этих структур больше не нужны, вы могли бы просто возвращать системе соответствующую страницу. Даже если бы структура данных занимала не одну, а несколько страниц, возврат памяти все равно был бы делом несложным. Но кто же пишет программы, подгоняя размер структур под размер страниц памяти — у разных процессоров они разные.
- Гораздо практичнее вести учет используемых структур данных. Для экономии памяти можно применить битовую карту. Так, имея массив из 100 структур, вы создаете дополнительный массив из 100 битов. Изначально все биты сброшены (обнулены), указывая тем самым, что ни одна структура не используется. По мере заполнения структур вы устанавливаете соответствующие биты (т. е. приравниваете их единице). Отпала необходимость в какой-то структуре — сбросьте ее бит и проверьте биты соседних структур, расположенных в пределах той же страницы памяти. Если и они не используются, страницу можно вернуть системе.
- В последнем варианте реализуется функция сбора мусора. Как известно, система при первой передаче физической памяти обнуляет все байты на переданной странице. Чтобы воспользоваться этим обстоятельством, предусмотрите в своей структуре элемент типа BOOL (назвав его, скажем, *fInUse*) и всякий раз, когда структура записывается в переданную память, устанавливайте его в TRUE.

- При выполнении программы вы будете периодически вызывать функцию сбора мусора, которая должна просматривать все структуры. Для каждой структуры (и существующей, и той, которая может быть создана) функция сначала определяет, передана ли под нее память; если да, то проверяет значение *fInUse*. Если он равен 0, структура не используется; TRUE — структура занята. Проверив все структуры, расположенные в пределах заданной страницы, функция сбора мусора вызывает *VirtualFree,* чтобы освободить память, — если, конечно, на этой странице нет используемых структур.
- Функцию сбора мусора можно вызывать сразу после того, как необходимость в одной из структур отпадет, но делать так не стоит, поскольку функция каждый раз просматривает все структуры — и существующие, и те, которые могут быть созданы. Оптимальный путь — реализовать эту функцию как поток с более низким уровнем приоритета. Это позволит не отнимать время у потока, выполняющего основную программу. А когда основная программа будет простаивать или ее поток займется файловым вводом-выводом, вот тогда система и выделит время функции сбора мусора.

Лично я предпочитаю первые два способа. Однако, если ваши структуры компактны (меньше одной страницы памяти), советую применять последний метод.

Программа-пример VMAlloc

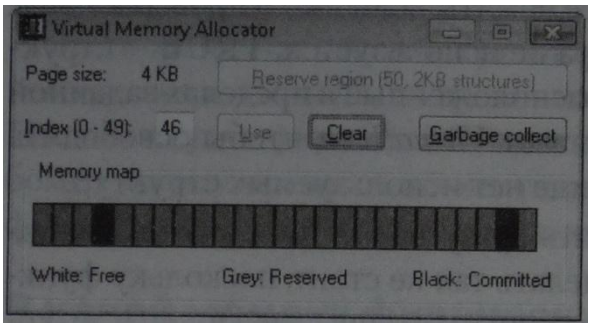
Эта программа (см. листинг VMAlloc.cpp), демонстрирует применение механизма виртуальной памяти для управления массивом структур. Файлы исходного кода и ресурсов этой программы находятся в каталоге 15-VMAlloc внутри архива, доступного на сайте поддержки этой книги. После запуска VMAlloc на экране появится диалоговое окно, показанное ниже.



Изначально для массива не резервируется никакого региона, и все адресное пространство, предназначенное для него, свободно, что и отражено на карте памяти. Если щелкнуть кнопку Reserve Region (50,2KB Structures), программа VMAlloc вызовет *VirtualAUoc* для резервирования региона, что сразу отразится на карте памяти. После этого станут активными и остальные кнопки в диалоговом окне.

Теперь в поле можно ввести индекс и щелкнуть кнопку Use. При этом по адресу, где должен располагаться указанный элемент массива, передается физическая память. Далее карта памяти вновь перерисовывается и уже отражает состояние региона, зарезервированного под весь массив. Когда вы,

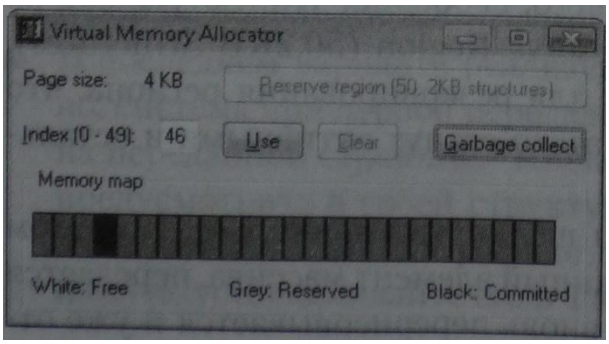
зарезервировав регион, вновь щелкнете кнопку Use, чтобы пометить элементы 7 и 46 как занятые, окно (при выполнении программы на процессоре с размером страниц по 4 Кб) будет выглядеть так:



Любой элемент массива, помеченный как занятый, можно освободить щелчком кнопки Clear. Но это не приведет к возврату физической памяти, переданной под элемент массива. Дело в том, что каждая страница содержит несколько структур и освобождение одной структуры не влечет за собой освобождения других. Если бы память была возвращена, то пропали бы и данные, содержащиеся в остальных структурах. И поскольку выбор кнопки Clear никак не сказывается на физической памяти региона, карта памяти после освобождения элемента не меняется.

Однако освобождение структуры приводит к тому, что ее элемент *fInUse* устанавливается в FALSE. Это нужно для того, чтобы функция сбора мусора могла вернуть не используемую больше физическую память. Кнопка Garbage Collect, если вы еще не догадались, заставляет программу VMAlloc выполнить функцию сбора мусора. Для упрощения программы я не стал выделять эту функцию в отдельный поток.

Чтобы посмотреть, как работает функция сбора мусора, очистите элемент массива с индексом 46. Заметьте, что карта памяти пока не изменилась. Теперь щелкните кнопку Garbage Collect. Программа освободит страницу, содержащую 46-й элемент, и карта памяти сразу же обновится, как показано ниже. Заметьте, что функцию *GarbageCollect* можно легко использовать в любых других приложениях. Я реализовал ее так, чтобы она работала с массивами структур данных любого размера; при этом структура не обязательно должна полностью умещаться на странице памяти. Единственное требование заключается в том, что первый элемент структуры должен быть значением типа BOOL, которое указывает, задействована ли данная структура.



И, наконец, хоть это и не видно на экране, закрытие окна приводит к возврату всей переданной памяти и освобождению зарезервированного региона.

Но есть в этой программе еще одна особенность, о которой я пока не упоминал. Программе приходится определять состояние памяти в адресном пространстве региона в трех случаях.

- После изменения индекса. Программе нужно включить кнопку Use и отключить кнопку Clear (или наоборот).
- В функции сбора мусора. Программа, прежде чем проверять значение флага *fInUse*, должна определить, была ли передана память.
- При обновлении карты памяти. Программа должна выяснить, какие страницы свободны, какие — зарезервированы, а какие — переданы.

Все эти проверки VMAlloc осуществляет через функцию *VirtualQuery*, рассмотренную в предыдущей главе.

VMAlloc.cpp

```
/******************************************************************************
Module:  VMAlloc.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"
#include <StrSafe.h>


//////////////////////////////////////////////////////////////////////////////


// The number of bytes in a page on this host machine.
UINT g_uPageSize = 0;


// A dummy data structure used for the array.
typedef struct {
   BOOL bInUse;
   BYTE bOtherData[2048 - sizeof(BOOL)];
} SOMEDATA, *PSOMEDATA;


// The number of structures in the array
#define MAX_SOMEDATA    (50)

// Pointer to an array of data structures
PSOMEDATA g_pSomeData = NULL;

// The rectangular area in the window occupied by the memory map
RECT g_rcMemMap;
```

```
//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_VMALLOC);

   // Initialize the dialog box by disabling all the nonsetup controls.
   EnableWindow(GetDlgItem(hWnd, IDC_INDEXTEXT),      FALSE);
   EnableWindow(GetDlgItem(hWnd, IDC_INDEX),          FALSE);
   EnableWindow(GetDlgItem(hWnd, IDC_USE),            FALSE);
   EnableWindow(GetDlgItem(hWnd, IDC_CLEAR),          FALSE);
   EnableWindow(GetDlgItem(hWnd, IDC_GARBAGECOLLECT), FALSE);

   // Get the coordinates of the memory map display.
   GetWindowRect(GetDlgItem(hWnd, IDC_MEMMAP), &g_rcMemMap);
   MapWindowPoints(NULL, hWnd, (LPPOINT) &g_rcMemMap, 2);

   // Destroy the window that identifies the location of the memory map
   DestroyWindow(GetDlgItem(hWnd, IDC_MEMMAP));

   // Put the page size in the dialog box just for the user's information.
   TCHAR szBuf[10];
   StringCchPrintf(szBuf, _countof(szBuf), TEXT("%d KB"), g_uPageSize / 1024);
   SetDlgItemText(hWnd, IDC_PAGESIZE, szBuf);

   // Initialize the edit control.
   SetDlgItemInt(hWnd, IDC_INDEX, 0, FALSE);

   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnDestroy(HWND hWnd) {

   if (g_pSomeData != NULL)
      VirtualFree(g_pSomeData, 0, MEM_RELEASE);
}


//////////////////////////////////////////////////////////////////////////////


VOID GarbageCollect(PVOID pvBase, DWORD dwNum, DWORD dwStructSize) {
```

```
   UINT uMaxPages = dwNum * dwStructSize / g_uPageSize;
   for (UINT uPage = 0; uPage < uMaxPages; uPage++) {
      BOOL bAnyAllocsInThisPage = FALSE;
      UINT uIndex     = uPage  * g_uPageSize / dwStructSize;
      UINT uIndexLast = uIndex + g_uPageSize / dwStructSize;

      for (; uIndex < uIndexLast; uIndex++) {
         MEMORY_BASIC_INFORMATION mbi;
         VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));
         bAnyAllocsInThisPage = ((mbi.State == MEM_COMMIT) &&
            * (PBOOL) ((PBYTE) pvBase + dwStructSize * uIndex));

         // Stop checking this page, we know we can't decommit it.
         if (bAnyAllocsInThisPage) break;
      }

      if (!bAnyAllocsInThisPage) {
         // No allocated structures in this page; decommit it.
         VirtualFree(&g_pSomeData[uIndexLast - 1], dwStructSize, MEM_DECOMMIT);
      }
   }
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   UINT uIndex = 0;

   switch (id) {
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;

      case IDC_RESERVE:
         // Reserve enough address space to hold the array of structures.
         g_pSomeData = (PSOMEDATA) VirtualAlloc(NULL,
            MAX_SOMEDATA * sizeof(SOMEDATA), MEM_RESERVE, PAGE_READWRITE);

         // Disable the Reserve button and enable all the other controls.
         EnableWindow(GetDlgItem(hWnd, IDC_RESERVE),        FALSE);
         EnableWindow(GetDlgItem(hWnd, IDC_INDEXTEXT),      TRUE);
         EnableWindow(GetDlgItem(hWnd, IDC_INDEX),          TRUE);
```

```
         EnableWindow(GetDlgItem(hWnd, IDC_USE),            TRUE);
         EnableWindow(GetDlgItem(hWnd, IDC_GARBAGECOLLECT), TRUE);

         // Force the index edit control to have the focus.
         SetFocus(GetDlgItem(hWnd, IDC_INDEX));

         // Force the memory map to update
         InvalidateRect(hWnd, &g_rcMemMap, FALSE);
         break;

      case IDC_INDEX:
         if (codeNotify != EN_CHANGE)
            break;

        uIndex = GetDlgItemInt(hWnd, id, NULL, FALSE);
         if ((g_pSomeData != NULL) && chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {
            MEMORY_BASIC_INFORMATION mbi;
            VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));
            BOOL bOk = (mbi.State == MEM_COMMIT);
            if (bOk)
               bOk = g_pSomeData[uIndex].bInUse;

            EnableWindow(GetDlgItem(hWnd, IDC_USE),  !bOk);
            EnableWindow(GetDlgItem(hWnd, IDC_CLEAR), bOk);

         } else {
            EnableWindow(GetDlgItem(hWnd, IDC_USE),   FALSE);
            EnableWindow(GetDlgItem(hWnd, IDC_CLEAR), FALSE);
         }
         break;


      case IDC_USE:
         uIndex = GetDlgItemInt(hWnd, IDC_INDEX, NULL, FALSE);
         // NOTE: New pages are always zeroed by the system
         VirtualAlloc(&g_pSomeData[uIndex], sizeof(SOMEDATA),
            MEM_COMMIT, PAGE_READWRITE);

         g_pSomeData[uIndex].bInUse = TRUE;

         EnableWindow(GetDlgItem(hWnd, IDC_USE),   FALSE);
         EnableWindow(GetDlgItem(hWnd, IDC_CLEAR), TRUE);

         // Force the Clear button control to have the focus.
         SetFocus(GetDlgItem(hWnd, IDC_CLEAR));
```

```
         // Force the memory map to update
         InvalidateRect(hWnd, &g_rcMemMap, FALSE);
         break;

      case IDC_CLEAR:
         uIndex = GetDlgItemInt(hWnd, IDC_INDEX, NULL, FALSE);
         g_pSomeData[uIndex].bInUse = FALSE;
         EnableWindow(GetDlgItem(hWnd, IDC_USE),   TRUE);
         EnableWindow(GetDlgItem(hWnd, IDC_CLEAR), FALSE);

         // Force the Use button control to have the focus.
         SetFocus(GetDlgItem(hWnd, IDC_USE));
         break;

      case IDC_GARBAGECOLLECT:
         GarbageCollect(g_pSomeData, MAX_SOMEDATA, sizeof(SOMEDATA));

         // Force the memory map to update
         InvalidateRect(hWnd, &g_rcMemMap, FALSE);
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnPaint(HWND hWnd) {    // Update the memory map

   PAINTSTRUCT ps;
   BeginPaint(hWnd, &ps);

   UINT uMaxPages = MAX_SOMEDATA * sizeof(SOMEDATA) / g_uPageSize;
   UINT uMemMapWidth = g_rcMemMap.right - g_rcMemMap.left;

   if (g_pSomeData == NULL) {

      // The memory has yet to be reserved.
      Rectangle(ps.hdc, g_rcMemMap.left, g_rcMemMap.top,
         g_rcMemMap.right - uMemMapWidth % uMaxPages, g_rcMemMap.bottom);

   } else {

      // Walk the virtual address space, painting the memory map
      for (UINT uPage = 0; uPage < uMaxPages; uPage++) {

         UINT uIndex = uPage * g_uPageSize / sizeof(SOMEDATA);
         UINT uIndexLast = uIndex + g_uPageSize / sizeof(SOMEDATA);
         for (; uIndex < uIndexLast; uIndex++) {
```

```
            MEMORY_BASIC_INFORMATION mbi;
            VirtualQuery(&g_pSomeData[uIndex], &mbi, sizeof(mbi));

            int nBrush = 0;
            switch (mbi.State) {
               case MEM_FREE:    nBrush = WHITE_BRUSH; break;
               case MEM_RESERVE: nBrush = GRAY_BRUSH;  break;
               case MEM_COMMIT:  nBrush = BLACK_BRUSH; break;
            }

            SelectObject(ps.hdc, GetStockObject(nBrush));
            Rectangle(ps.hdc,
               g_rcMemMap.left + uMemMapWidth / uMaxPages * uPage,
               g_rcMemMap.top,
               g_rcMemMap.left + uMemMapWidth / uMaxPages * (uPage + 1),
               g_rcMemMap.bottom);
         }
      }
   }

   EndPaint(hWnd, &ps);
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
      chHANDLE_DLGMSG(hWnd, WM_PAINT,      Dlg_OnPaint);
      chHANDLE_DLGMSG(hWnd, WM_DESTROY,    Dlg_OnDestroy);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR, int) {

   // Get the page size used on this CPU.
   SYSTEM_INFO si;
   GetSystemInfo(&si);
   g_uPageSize = si.dwPageSize;
```

```
   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_VMALLOC), NULL, Dlg_Proc);
   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

Изменение атрибутов защиты

Хоть это и не принято, но атрибуты защиты, присвоенные странице или страницам переданной физической памяти, можно изменять. Допустим, вы разработали код для управления связанным списком, узлы (вершины) которого хранятся в зарезервированном регионе. При желании можно написать функции, которые обрабатывали бы связанные списки и изменяли бы атрибуты защиты переданной памяти при старте на PAGE_READWRITE, а при завершении — обратно на PAGE_NOACCESS.

Сделав так, вы защитите данные в связанном списке от возможных «жучков», скрытых в программе. Например, если какой-то блок кода в вашей программе из-за наличия «блуждающего» указателя обратится к данным в связанном списке, возникнет нарушение доступа. Поэтому такой подход иногда очень полезен — особенно когда пытаешься найти трудноуловимую ошибку в своей программе.

Атрибуты защиты страницы памяти можно изменить вызовом *Virtual-Protect*

```
BOOL VirtualProtect(
    PVOID pvAddress,
    SIZE_T dwSlze,
    DWORD flNewProtect,
    PDWORD pflOldProtect);
```

Здесь *pvAddress* указывает на базовый адрес памяти (который должен находиться в пользовательском разделе вашего процесса), *dwSize* определяет число байтов, для которых вы изменяете атрибут защиты, а *flNewProtect* содержит один из идентификаторов PAGE_*, кроме PAGE_WRITECOPY и PAGE_EXECUTE_WRITECOPY.

Последний параметр, *pflOldProtect*, содержит адрес переменной типа DWORD, в которую *VirtualProtect* заносит старое значение атрибута защиты для данной области памяти. В этом параметре (даже если вас не интересует такая информация) нужно передать корректный адрес, иначе функция приведет к нарушению доступа.

Естественно, атрибуты защиты связаны с целыми страницами памяти и не могут присваиваться отдельным байтам. Поэтому, если на процессоре с четырехкилобайтовыми страницами вызвать *VirtualProtect*, например, так:

```
VirtualProtect(pvRgnBase + (3 * 1024), 2 * 1024,
    PAGE_NOACCESS, &flOldProtect);
```

то атрибут защиты PAGE_NOACCESS будет присвоен двум страницам памяти.

Функцию *VirtualProtect* нельзя использовать для изменения атрибутов защиты страниц, диапазон которых охватывает разные зарезервированные регионы. В таких случаях *VirtualProtect* надо вызывать для каждого региона отдельно.

Сброс содержимого физической памяти

Когда вы модифицируете содержимое страниц физической памяти, система пытается как можно дольше хранить эти изменения в оперативной памяти. Однако, выполняя приложения, система постоянно получает запросы на загрузку в оперативную память страниц из EXE-файлов, DLL и/или страничного файла. Любой такой запрос заставляет систему просматривать оперативную память и выгружать модифицированные страницы в страничный файл.

Windows позволяет программам увеличить свою производительность за счет сброса физической памяти, вы сообщаете системе, что данные на одной или нескольких страницах памяти не изменялись. Если система в процессе поиска свободной страницы в оперативной памяти выбирает измененную страницу, то должна сначала записать ее в страничный файл. Эта операция отнимает довольно много времени и отрицательно сказывается на производительности. Поэтому в большинстве приложений желательно, чтобы система как можно дольше хранила модифицированные страницы в страничном файле.

Однако некоторые программы занимают блоки памяти на очень малое время, а потом им уже не требуется их содержимое. Для большего быстродействия программа может попросить систему не записывать определенные страницы в страничный файл. И тогда, если одна из этих страниц понадобится для других целей, системе не придется сохранять ее в страничном файле, что, естественно, повысит скорость работы программы. Такой отказ от страницы (или страниц) памяти называется *сбросом физической памяти* (resetting of physical storage) и инициируется вызовом функции *VirtualAlloc* с передачей ей в третьем параметре флага MEM_RESET.

Если страницы, на которые вы ссылаетесь при вызове *VirtualAlloc,* находятся в страничном файле, система их удалит. Когда в следующий раз программа обратится к памяти, она получит новые страницы, инициализированные нулями. Если же вы сбрасываете страницу, находящуюся в оперативной памяти, система помечает ее как неизменявшуюся, и она не записывается в страничный файл. Но, хотя ее содержимое *не обнуляется,* читать такую страницу памяти уже нельзя. Если системе не понадобится эта страница оперативной памяти, ее содержимое останется прежним. В ином случае система может забрать ее в свое распоряжение, и тогда обращение к этой странице приведет к тому, что система предоставит программе новую страницу, заполненную нулями. А поскольку этот процесс нам не подвластен, лучше считать, что после сброса страница содержит только мусор.

При сбросе физической памяти надо учитывать и несколько других моментов. Во-первых, когда вы вызываете *VirtualAlloc, базовый* адрес обычно округляется до ближайшего меньшего значения, кратного размеру страниц, а количество байтов — до ближайшего большего значения, кратного той же величине. Такой механизм округления базового адреса и количества байтов был бы очень опасен при сбросе физической памяти; поэтому *VirtualAlloc* при передаче ей флага MEM_RESET округляет эти значения прямо наоборот. Допустим, в вашей программе есть следующий исходный код:

```
PINT pnData = (PINT) VirtualAlloc(NULL, 1024,
   MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
pnData[0] = 100;
pnData[1] = 200;
VirtualAlloc((PV0ID) pnData, sizeof(int), MEM.RESET, PAGE_READWRITE);
```

Этот код передает одну страницу памяти, а затем сообщает, что первые четыре байта (*sizeof (int)*) больше не нужны и их можно сбросить. Однако, как и при любых других действиях с памятью, эта операция *выполняется только* над блоками памяти, размер которых кратен размеру страниц. В данном случае вызов завершится неудачно (*VirtualAlloc* вернет NULL, а GetLastError — ошибку ERROR_INVALID_ADDRESS, определенную в WinError.h как ошибка 487). Почему? Дело в том, что при вызове VirtualAlloc вы указали флаг MEM_RESET и базовый адрес, переданный функции, теперь округляется до ближайшего большего значения, кратного размеру страниц, а количество байтов — до ближайшего меньшего значения, кратного той же величине. Так делается, чтобы исключить случайную потерю важных данных. В предыдущем примере округление количества байтов до ближайшего меньшего.

Второе, о чем следует помнить при сбросе памяти, — флаг MEM_RESET нельзя комбинировать (логической операцией OR) ни с какими другими флагами. Следующий вызов всегда будет заканчиваться неудачно:

```
PV0ID pvMem = VirtualAlloc(NULL, 1024,
   MEM_RESERVE | MEM_COMMIT | MEM_RESET, PAGE_READWRITE);
```

Впрочем, комбинировать флаг MEM_RESET с другими флагами все равно бессмысленно.

И, наконец, последнее. Вызов *VirtualAlloc* с флагом MEM_RESET требует передачи корректного атрибута защиты страницы, даже несмотря на то что он не будет использоваться данной функцией.

Программа-пример MemReset

Эта программа (см. листинг MemReset.cpp), демонстрирует, как работает флаг MEM RESET. Файлы исходного кода и ресурсов этой программы находятся в каталоге 15-MemReset внутри архива, доступного на веб-сайте поддержи этой книги.

Первое, что делает код этой программы» — резервирует регион и передает ему физическую память. Поскольку размер региона, переданный

в *VirtualAlloc*, равен 1024 байтам, система автоматически округляет это значение до размера страницы. Затем функция *_tcscpy_s* копирует в этот буфер строку, и содержимое страницы оказывается измененным. Если система впоследствии сочтет, что ей нужна страница, содержащая наши данные, она запишет эту страницу в страничный файл. Когда наша программа попытается считать эти данные, система автоматически загрузит страницу из страничного файла в оперативную память.

После записи строки в страницу памяти наша программа спрашивает у пользователя, понадобятся ли еще эти данные. Если пользователь выбирает отрицательный ответ (щелчком кнопки No), программа сообщает системе, что страница не изменялась, для чего вызывает *VirtualAlloc* с флагом MEM_RESET.

Для демонстрации того факта, что память действительно сброшена, смоделируем высокую нагрузку на оперативную память, для чего:

1. Получим общий размер оперативной памяти на компьютере вызовом *GlobalMemoryStatus*.
2. Передадим эту память вызовом VirtualAlloc. Данная операция выполняется очень быстро, поскольку система не выделяет оперативную память до тех пор, пока процесс не изменит какие-нибудь страницы.
3. Изменим содержимое только что переданных страниц через функцию ZeroMemory. Это создает высокую нагрузку на оперативную память, и отдельные страницы выгружаются в страничный файл.

Если пользователь захочет оставить данные, сброс не осуществляется, и при первой же попытке доступа к ним соответствующие страницы будут подгружаться в оперативную память из страничного файла. Если же пользователь откажется от этих данных, мы выполняем сброс памяти, система не записывает их в страничный файл, и это ускоряет выполнение программы.

После вызова *ZeroMemory* я сравниваю содержимое страницы данных со строкой, которая была туда записана. Если данные не сбрасывались, содержимое идентично, а если сбрасывались — то ли идентично, то ли нет. В моей программе содержимое никогда не останется прежним, поскольку я заставляю систему выгрузить все страницы оперативной памяти в страничный файл. Но если бы размер выгружаемой области был меньше общего объема оперативной памяти, то не исключено, что исходное содержимое все равно осталось бы в памяти. Так что будьте осторожны!

MemReset.cpp

```
/******************************************************************************
Module:  MemReset.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
```

```
//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE, HINSTANCE, PTSTR, int) {

   TCHAR szAppName[]  = TEXT("MEM_RESET tester");
   TCHAR szTestData[] = TEXT("Some text data");

   // Commit a page of storage and modify its contents.
   PTSTR pszData = (PTSTR) VirtualAlloc(NULL, 1024,
      MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
   _tcscpy_s(pszData, 1024, szTestData);

   if (MessageBox(NULL, TEXT("Do you want to access this data later?"),
      szAppName, MB_YESNO) == IDNO) {

      // We want this page of storage to remain in our process but the
      // contents aren't important to us anymore.
      // Tell the system that the data is not modified.

      // Note: Because MEM_RESET destroys data, VirtualAlloc rounds
      // the base address and size parameters to their safest range.
      // Here is an example:
      //    VirtualAlloc(pvData, 5000, MEM_RESET, PAGE_READWRITE)
      // resets 0 pages on CPUs where the page size is greater than 4 KB
      // and resets 1 page on CPUs with a 4 KB page. So that our call to
      // VirtualAlloc to reset memory below always succeeds, VirtualQuery
      // is called first to get the exact region size.
      MEMORY_BASIC_INFORMATION mbi;
      VirtualQuery(pszData, &mbi, sizeof(mbi));
      VirtualAlloc(pszData, mbi.RegionSize, MEM_RESET, PAGE_READWRITE);
   }

   // Commit as much storage as there is physical RAM.
   MEMORYSTATUS mst;
   GlobalMemoryStatus(&mst);
   PVOID pvDummy = VirtualAlloc(NULL, mst.dwTotalPhys,
      MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

   // Touch all the pages in the dummy region so that any
   // modified pages in RAM are written to the paging file.
   if (pvDummy != NULL)
      ZeroMemory(pvDummy, mst.dwTotalPhys);

   // Compare our data page with what we originally wrote there.
   if (_tcscmp(pszData, szTestData) == 0) {
```

```
      // The data in the page matches what we originally put there.
      // ZeroMemory forced our page to be written to the paging file.
      MessageBox(NULL, TEXT("Modified data page was saved."),
         szAppName, MB_OK);
   } else {

      // The data in the page does NOT match what we originally put there
      // ZeroMemory didn't cause our page to be written to the paging file
      MessageBox(NULL, TEXT("Modified data page was NOT saved."),
         szAppName, MB_OK);
   }

   // Don't forget to release part of the address space.
   // Note that it is not mandatory here since the application is exiting.
   if (pvDummy != NULL)
      VirtualFree(pvDummy, 0, MEM_RELEASE);
   VirtualFree(pszData, 0, MEM_RELEASE);

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

Механизм Address Windowing Extensions

Жизнь идет вперед, и приложения требуют все больше и больше памяти — особенно серверные. Чем выше число клиентов, обращающихся к серверу, тем меньше его производительность. Для увеличения быстродействия серверное приложение должно хранить как можно больше своих данных в оперативной памяти и сбрасывать их на диск как можно реже. Другим классам приложений (базам данных, программам для работы с трехмерной графикой, математическими моделями и др.) тоже нужно манипулировать крупными блоками памяти. И всем этим приложениям уже тесно в 32-разрядном адресном пространстве.

Для таких приложений Windows предлагает новый механизм — Address Windowing Extensions (AWE). Создавая AWE, Майкрософт стремилась к тому, чтобы приложения могли:

- работать с оперативной памятью, никогда не выгружаемой на диск операционной системой;
- обращаться к таким объемам оперативной памяти, которые превышают размеры соответствующих разделов в адресных пространствах их процессов.

AWE дает возможность приложению выделять себе один и более блоков оперативной памяти, невидимых в адресном пространстве процесса. Сделав это, приложение резервирует регион адресного пространства (с помощью *VirtualAlloc),* и он становится адресным окном (address window). Далее программа вызывает функцию, которая связывает адресное окно с одним из выделенных блоков оперативной памяти. Эта операция выполняется чрезвычайно быстро (обычно за пару миллисекунд).

Через одно адресное окно единовременно доступен лишь один блок памяти. Это, конечно, усложняет программирование, так как при обращении к другому блоку приходится явно вызывать функции, которые как бы переключают адресное окно на очередной блок.

Вот пример, демонстрирующий использование AWE:

```
// сначала резервируем для адресного окна регион размером 1 Мб
ULONG_PTR ulRAMBytes = 1024 * 1024;
PVOID pvWindow = VirtualAlloc(NULL, ulRAMBytes,
    MEM_RESERVE | MEM_PHYSICAL, PAGE_READWRITE);

// получаем размер страниц на данной процессорной платформе
SYSTEM_INFO sinf;
GetSystemlnfo(&sinf);

// вычисляем, сколько страниц памяти нужно для нашего количества байтов
UL0NG_PTR ulRAMPages = (ulRAMBytes + sinf.dwPageSize - 1) / sinf.
    dwPageSize;

// создаем соответствующий массив для номеров фреймов страниц
ULONG_PTR* aRAMPages = (UL0NG_PTR*) new UL0NG_PTR[ulRAMPages];

// выделяем страницы оперативной памяти (в полномочиях пользователя
// должна быть разрешена блокировка страниц в памяти)
AllocateUserPhysicalPages(
    GetCurrentProcessO,             // выделяем память для нашего процесса
    &ulRAMPages,                    // на входе: количество запрошенных страниц
    RAM,
    aRAMPages);                     // на выходе: количество выделенных страниц RAM

// назначаем страницы оперативной памяти нашему окну
MapUserPhysicalPages(pvWindow,      // адрес адресного окна
    ulRAMPages,                     // число элементов в массиве
    aRAMPages);                     // массив страниц RAM

// обращаемся к этим страницам через виртуальный адрес pvWindow
…
// освобождаем блок страниц оперативной памяти
FreeUserPhysicalPages(
    GetCurrentProcessO,         // освобождаем RAM, выделенную нашему процессу
    &ulRAMPages,                // на входе: количество страниц RAM,
```

```
                                // на выходе: количество освобожденных страниц
    RAN aRAMPages);             // на входе: массив, идентифицирующий освобождаемые
                                // страницы RAM
                                // уничтожаем адресное окно
VirtualFree(pvWindow, 0, MEM_RELEASE);
delete[] aRAMPages;
```

Как видите, пользоваться AWE несложно. А теперь хочу обратить ваше внимание на несколько интересных моментов, связанных с этим фрагментом кода.

Вызов *VirtualAlloc* резервирует адресное окно размером 1 Мб. Обычно адресное окно гораздо больше. Вы должны выбрать его размер в соответствии с объемом блоков оперативной памяти, необходимых вашему приложению. Но, конечно, размер такого окна ограничен размером самого крупного свободного (и непрерывного!) блока в адресном пространстве процесса. Флаг MEM_RESERVE указывает, что я просто резервирую диапазон адресов, а флаг MEMPHYSICAL — что в конечном счете этот диапазон адресов будет связан с физической (оперативной) памятью. Механизм AWE требует, чтобы вся память, связываемая с адресным окном, была доступна для чтения и записи; поэтому в данном случае функции *VirtualAlloc* можно передать только один атрибут защиты — PAGE_RE ADWRITE. Кроме того, нельзя пользоваться функцией *VirtualProtect* и пытаться изменять тип защиты этого блока памяти.

Для выделения блока в физической памяти надо вызвать функцию *AllocateUserPhysicalPages:*

```
BOOL AllocateUserPhysicalPages(
    HANDLE hProcess,
    PULONG_PTR pulRAMPages,
    PULONG_PTR aRAMPages);
```

Она выделяет количество страниц оперативной памяти, заданное в значении, на которое указывает параметр *pulRAMPages,* и закрепляет эти страницы за процессом, определяемым параметром *hProcess.*

Операционная система назначает каждой странице оперативной памяти *номер фрейма страницы* (page frame number). По мере того как система отбирает страницы памяти, выделяемые приложению, она вносит соответствующие данные (номер фрейма страницы для каждой страницы оперативной памяти) в массив, на который указывает параметр *aRAMPages.* Сами по себе эти номера для приложения совершенно бесполезны; вам не следует просматривать содержимое этого массива и тем более что-либо менять в нем. Вы не узнаете, какие страницы оперативной памяти будут выделены под запрошенный блок, да это и не нужно. Когда эти страницы связываются с адресным окном, они появляются в виде непрерывного блока памяти. А что там система делает для этого, вас не должно интересовать.

Когда функция *AllocateUserPhysicalPages* возвращает управление, значение в *pulRAMPages* сообщает количество фактически выделенных страниц.

Обычно оно совпадает с тем, что вы передаете функции, но может оказаться и поменьше.

Страницы оперативной памяти выделяются только процессу, из которого была вызвана данная функция; AWE не разрешает проецировать их на адресное пространство другого процесса. Поэтому такие блоки памяти нельзя разделять между процессами.

> **Примечание.** Конечно, оперативная память — ресурс драгоценный, и приложение может выделить лишь ее незадействованную часть. Не злоупотребляйте механизмом AWE: если ваш процесс захватит слишком много оперативной памяти, это может привести к интенсивной перекачке страниц на диск и резкому падению производительности всей системы. Кроме того, это ограничит возможности системы в создании новых процессов, потоков и других ресурсов. (Мониторинг степени использования физической памяти можно реализовать через функцию *GlobalMemoryStatusEx*.)
>
> *AllocateUserPhysicalPages* требует также, чтобы приложению была разрешена блокировка страниц в памяти (т. е. у пользователя должно быть право «Lock Pages in Memory»), а иначе функция потерпит неудачу. По умолчанию таким правом пользователи или их группы не наделяются. Оно назначается учетной записи Local System, которая обычно используется различными службами. Если вы хотите запускать интерактивное приложение, вызывающее *AllocateUserPhysicalPages,* администратор должен предоставить вам соответствующее право еще до того, как вы зарегистрируетесь в системе.
>
> Для вызова функции *AllocateUserPhysicalPages* необходимо право Lock Pages In Memory. О том, как предоставить и активировать его, рассказывается выше, в разделе «Резервирование региона с одновременной передачей физической памяти».

Теперь, создав адресное окно и выделив блок памяти, я связываю этот блок с окном вызовом функции *MapUserPhysicalPages:*

```
BOOL MapUserPhysicalPages(
    PVOID pvAddressWindow,
    ULONG_PTR ulRAMPages,
    PULONG_PTR aRAMPages);
```

Ее первый параметр, *pvAddressWindow,* определяет виртуальный адрес адресного окна, а последние два параметра, *ulRAMPages* и *aRAMPages,* сообщают, сколько страниц оперативной памяти должно быть видимо через адресное окно и что это за страницы. Если окно меньше связываемого блока памяти, функция потерпит неудачу. Приоритетной задачей Майкрософт при разработке этой функции была скорость, поэтому *MapUserPhysicalPages* тратит на проецировании блока в среднем не *более нескольких* микросекунд.

Примечание. Функция *MapUserPhysicalPages* отключает текущий блок оперативной памяти от адресного окна, если вместо параметра aRAMPages передается NULL. Вот пример:

```
// Отключает текущий блок оперативной памяти от адресного окна
MapUserPhysicalPages(pvWindow, ulRAMPages, NULL);
```

Связав блок оперативной памяти с адресным окном, вы можете легко обращаться к этой памяти, просто ссылаясь на виртуальные адреса относительно базового адреса адресного окна (в моем примере это *pvWindow*).

Когда необходимость в блоке памяти отпадет, освободите его вызовом функции *FreeUserPkysicalPagesr*.

```
BOOL FreeUserPhysicalPages(
    HANDLE hProcess,
    PULONG_PTR pulRAMPages,
    PULONG_PTR aRAMPages);
```

Ее первый параметр, *hProcess*, идентифицирует процесс, владеющий данными страницами памяти, а последние два параметра сообщают, сколько страниц оперативной памяти следует освободить и что это за страницы. Если освобождаемый блок в данный момент связан с адресным окном, он сначала отключается от этого окна.

И, наконец, завершая очистку, я освобождаю адресное окно. Для этого я вызываю *VirtualFree* и передаю ей базовый виртуальный адрес окна, нуль вместо размера региона и флаг MEM_RELEASE.

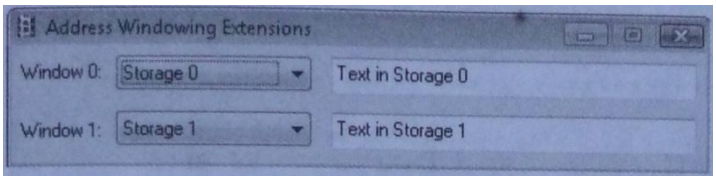
В моем простом примере создается одно адресное окно и единственный блок памяти. Это позволяет моей программе обращаться к оперативной памяти, которая никогда не будет сбрасываться на диск. Однако приложение может создать несколько адресных окон и выделить несколько блоков памяти. Эти блоки разрешается связывать с любым адресным окном, но операционная система не позволит связать один блок сразу с двумя окнами.

64-разрядная Windows полностью поддерживает AWE, так что перенос 32-разрядных приложений, использующих этот механизм, не вызывает никаких проблем. Однако AWE не столь полезен для 64-разрядных приложений, поскольку размеры их адресных пространств намного больше. Но все равно он дает возможность приложению выделять физическую память, которая никогда не сбрасывается на диск.

Программа-пример AWE

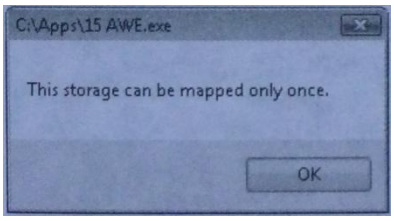
Эта программа (15-AWE.exe) демонстрирует, как создавать несколько адресных окон и связывать с ними разные блоки памяти. Файлы исходного кода и ресурсов этой программы находятся в каталоге 15-AWE внутри архива, доступного на вебсайте поддержки этой книги. Сразу после запуска программы AWE создается два адресных окна и выделяется два блока памяти.

Изначально первый блок занимает строка «Text in Storage 0», второй — строка «Text in Storage 1». Далее первый блок связывается с первым адресным окном, а второй — со вторым окном. При этом окно программы выглядит так, как показано ниже.



Оно позволяет немного поэкспериментировать. Во-первых, эти блоки можно назначить разным адресным окнам, используя соответствующие поля со списками. В них, кстати, предлагается и вариант No Storage, при выборе которого память отключается от адресного окна. Во-вторых, *любое* изменение текста немедленно отражается на блоке памяти, связанном с текущим адресным окном.

Если вы попытаетесь связать один и тот же блок памяти с двумя адресными окнами одновременно, то, поскольку механизм AWE это не разрешает, на экране появится следующее сообщение.



Исходный код этой программы-примера предельно ясен. Чтобы облегчить работу с механизмом AWE, я создал три C++-класса, которые содержатся в файле AddrWindows.h. Первый *класс, CSystemInfo,* — очень простая оболочка функции *GetSystemInfo.* По одному его экземпляру создают остальные два класса.

Второй C++-класс, CAddrWindow, инкапсулирует адресное окно. Его метод *Create* резервирует адресное окно, метод *Destroy* уничтожает это окно, метод *UnmapStorage* отключает от окна связанный с ним блок памяти, а метод оператора приведения PVOID просто возвращает виртуальный адрес адресного окна.

Третий C++-класс, CAddrWmdowStorage, инкапсулирует блок памяти, который можно назначить объекту класса CAddrWindow. Метод *Allocate* разрешает блокировать страницы в памяти, выделяет блок памяти, а затем отменяет право на блокировку. Метод *Free* освобождает блок памяти. Метод *HowManyPagesAllocated* возвращает количество фактически выделенных страниц. Наконец, метод *MapStorage* связывает блок памяти с объектом класса CAddrWindow, а *UnmapStorage* отключает блок от этого объекта.

Применение C++-классов существенно упростило реализацию программы AWE. Она создает по два объекта классов CAddrWindow и CAddr-

WindowStorage. Остальной код просто вызывает нужные методы в нужное время. Обратите также внимание на добавленный к программе манифест, заставляющий систему выводить окно с запросом повышения привилегий.

AWE.cpp

```
/******************************************************************************
Module:  AWE.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>
#include "AddrWindow.h"
#include "Resource.h"
#include <StrSafe.h>


//////////////////////////////////////////////////////////////////////////////


CAddrWindow g_aw[2];              // 2 memory address windows
CAddrWindowStorage g_aws[2];      // 2 storage blocks
const ULONG_PTR g_nChars = 1024; // 1024 character buffers
const DWORD g_cbBufferSize = g_nChars * sizeof(TCHAR);


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_AWE);

   // Create the 2 memory address windows
   chVERIFY(g_aw[0].Create(g_cbBufferSize));
   chVERIFY(g_aw[1].Create(g_cbBufferSize));

   // Create the 2 storage blocks
   if (!g_aws[0].Allocate(g_cbBufferSize)) {
      chFAIL("Failed to allocate RAM.\nMost likely reason: "
         "you are not granted the Lock Pages in Memory user right.");
   }
   chVERIFY(g_aws[1].Allocate(g_nChars * sizeof(TCHAR)));

   // Put some default text in the 1st storage block
   g_aws[0].MapStorage(g_aw[0]);
   _tcscpy_s((PTSTR) (PVOID) g_aw[0], g_cbBufferSize, TEXT("Text in Storage 0"));
```

```
   // Put some default text in the 2nd storage block
   g_aws[1].MapStorage(g_aw[0]);
   _tcscpy_s((PTSTR) (PVOID) g_aw[0], g_cbBufferSize, TEXT("Text in Storage 1"));

   // Populate the dialog box controls
   for (int n = 0; n <= 1; n++) {
      // Set the combo box for each address window
      int id = ((n == 0) ? IDC_WINDOW0STORAGE : IDC_WINDOW1STORAGE);
      HWND hWndCB = GetDlgItem(hWnd, id);
      ComboBox_AddString(hWndCB, TEXT("No storage"));
      ComboBox_AddString(hWndCB, TEXT("Storage 0"));
      ComboBox_AddString(hWndCB, TEXT("Storage 1"));

      // Window 0 shows Storage 0, Window 1 shows Storage 1
      ComboBox_SetCurSel(hWndCB, n + 1);
      FORWARD_WM_COMMAND(hWnd, id, hWndCB, CBN_SELCHANGE, SendMessage);
      Edit_LimitText(GetDlgItem(hWnd,
         (n == 0) ? IDC_WINDOW0TEXT : IDC_WINDOW1TEXT), g_nChars);
   }

   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   switch (id) {

   case IDCANCEL:
      EndDialog(hWnd, id);
      break;

   case IDC_WINDOW0STORAGE:
   case IDC_WINDOW1STORAGE:
      if (codeNotify == CBN_SELCHANGE) {

         // Show different storage in address window
         int nWindow  = ((id == IDC_WINDOW0STORAGE) ? 0 : 1);
         int nStorage = ComboBox_GetCurSel(hWndCtl) - 1;

         if (nStorage == -1) {   // Show no storage in this window
            chVERIFY(g_aw[nWindow].UnmapStorage());
         } else {
```

```
            if (!g_aws[nStorage].MapStorage(g_aw[nWindow])) {
               // Couldn't map storage in window
               chVERIFY(g_aw[nWindow].UnmapStorage());
               ComboBox_SetCurSel(hWndCtl, 0);  // Force "No storage"
               chMB("This storage can be mapped only once.");
            }
         }

         // Update the address window's text display
         HWND hWndText = GetDlgItem(hWnd,
            ((nWindow == 0) ? IDC_WINDOW0TEXT : IDC_WINDOW1TEXT));
         MEMORY_BASIC_INFORMATION mbi;
         VirtualQuery(g_aw[nWindow], &mbi, sizeof(mbi));
         // Note: mbi.State == MEM_RESERVE if no storage is in address window
         EnableWindow(hWndText, (mbi.State == MEM_COMMIT));
         Edit_SetText(hWndText, IsWindowEnabled(hWndText)
            ? (PCTSTR) (PVOID) g_aw[nWindow] : TEXT("(No storage)"));
      }
      break;

   case IDC_WINDOW0TEXT:
   case IDC_WINDOW1TEXT:
      if (codeNotify == EN_CHANGE) {
         // Update the storage in the address window
         int nWindow = ((id == IDC_WINDOW0TEXT) ? 0 : 1);
         Edit_GetText(hWndCtl, (PTSTR) (PVOID) g_aw[nWindow], g_nChars);
      }
      break;
   }
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }

   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////
```

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR, int) {

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_AWE), NULL, Dlg_Proc);
   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

AddrWindow.h

```
/******************************************************************************
Module:  AddrWindow.h
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#pragma once


//////////////////////////////////////////////////////////////////////////////


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>


//////////////////////////////////////////////////////////////////////////////


class CSystemInfo : public SYSTEM_INFO {
public:
   CSystemInfo() { GetSystemInfo(this); }
};


//////////////////////////////////////////////////////////////////////////////


class CAddrWindow {
public:
   CAddrWindow()  { m_pvWindow = NULL; }
   ~CAddrWindow() { Destroy(); }

   BOOL Create(SIZE_T dwBytes, PVOID pvPreferredWindowBase = NULL) {
      // Reserve address window region to view physical storage
      m_pvWindow = VirtualAlloc(pvPreferredWindowBase, dwBytes,
         MEM_RESERVE | MEM_PHYSICAL, PAGE_READWRITE);
      return(m_pvWindow != NULL);
   }

   BOOL Destroy() {
      BOOL bOk = TRUE;
      if (m_pvWindow != NULL) {
```

```
         // Destroy address window region
         bOk = VirtualFree(m_pvWindow, 0, MEM_RELEASE);
         m_pvWindow = NULL;
      }
      return(bOk);
   }

   BOOL UnmapStorage() {
      // Unmap all storage from address window region
      MEMORY_BASIC_INFORMATION mbi;
      VirtualQuery(m_pvWindow, &mbi, sizeof(mbi));
      return(MapUserPhysicalPages(m_pvWindow,
         mbi.RegionSize / sm_sinf.dwPageSize, NULL));
   }

   // Returns virtual address of address window
   operator PVOID() { return(m_pvWindow); }

private:
   PVOID m_pvWindow;    // Virtual address of address window region
   static CSystemInfo sm_sinf;
};


////////////////////////////////////////////////////////////////////////////


CSystemInfo CAddrWindow::sm_sinf;


////////////////////////////////////////////////////////////////////////////


class CAddrWindowStorage {
public:
   CAddrWindowStorage()  { m_ulPages = 0; m_pulUserPfnArray = NULL; }
   ~CAddrWindowStorage() { Free(); }

   BOOL Allocate(ULONG_PTR ulBytes) {
      // Allocate storage intended for an address window

      Free();  // Clean up this object's existing address window

      // Calculate number of pages from number of bytes
      m_ulPages = (ulBytes + sm_sinf.dwPageSize - 1) / sm_sinf.dwPageSize;

      // Allocate array of page frame numbers
      m_pulUserPfnArray = (PULONG_PTR)
         HeapAlloc(GetProcessHeap(), 0, m_ulPages * sizeof(ULONG_PTR));
```

```
      BOOL bOk = (m_pulUserPfnArray != NULL);
      if (bOk) {
         // The "Lock Pages in Memory" privilege must be enabled
         EnablePrivilege(SE_LOCK_MEMORY_NAME, TRUE);
         bOk = AllocateUserPhysicalPages(GetCurrentProcess(),
            &m_ulPages, m_pulUserPfnArray);
         EnablePrivilege(SE_LOCK_MEMORY_NAME, FALSE);
      }
      return(bOk);
   }

   BOOL Free() {
      BOOL bOk = TRUE;
      if (m_pulUserPfnArray != NULL) {
         bOk = FreeUserPhysicalPages(GetCurrentProcess(),
            &m_ulPages, m_pulUserPfnArray);
         if (bOk) {
            // Free the array of page frame numbers
            HeapFree(GetProcessHeap(), 0, m_pulUserPfnArray);
            m_ulPages = 0;
            m_pulUserPfnArray = NULL;
         }
      }
      return(bOk);
   }

   ULONG_PTR HowManyPagesAllocated() { return(m_ulPages); }

   BOOL MapStorage(CAddrWindow& aw) {
      return(MapUserPhysicalPages(aw,
         HowManyPagesAllocated(), m_pulUserPfnArray));
   }

   BOOL UnmapStorage(CAddrWindow& aw) {
      return(MapUserPhysicalPages(aw,
         HowManyPagesAllocated(), NULL));
   }

private:
   static BOOL EnablePrivilege(PCTSTR pszPrivName, BOOL bEnable = TRUE) {

      BOOL bOk = FALSE;    // Assume function fails
      HANDLE hToken;

      // Try to open this process' access token
      if (OpenProcessToken(GetCurrentProcess(),
```

```
         TOKEN_ADJUST_PRIVILEGES, &hToken)) {

         // Attempt to modify the "Lock pages in Memory" privilege
         TOKEN_PRIVILEGES tp = { 1 };
         LookupPrivilegeValue(NULL, pszPrivName, &tp.Privileges[0].Luid);
         tp.Privileges[0].Attributes = bEnable ? SE_PRIVILEGE_ENABLED : 0;
         AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
         bOk = (GetLastError() == ERROR_SUCCESS);
         CloseHandle(hToken);
      }
      return(bOk);
   }

private:
   ULONG_PTR  m_ulPages;         // Number of storage pages
   PULONG_PTR m_pulUserPfnArray; // Page frame number array

private:
   static CSystemInfo sm_sinf;
};


//////////////////////////////////////////////////////////////////////////////


CSystemInfo CAddrWindowStorage::sm_sinf;


//////////////////////////////// End of File /////////////////////////////////
```

Оглавление

ГЛАВА 16

Стек потока

Иногда система сама резервирует какие-то регионы в адресном пространстве вашего процесса. Я уже упоминал в главе 13, что это делается для размещения блоков переменных окружения процесса и его потоков. Еще один случай резервирования региона самой системой — создание стека потока.

Всякий раз, когда в процессе создается поток, система резервирует регион адресного пространства для стека потока (у каждого потока свой стек) и передает этому региону какой-то объем физической памяти. По умолчанию система резервирует 1 Мб адресного пространства и передает ему всего две страницы памяти. Но стандартные значения можно изменить, указав при сборке программы параметр компоновщика /STACK либо параметр /F при компиляции:

```
/Freserve
/STACK:reserve[,commit]
```

При компоновке приложений размер стека, заданный параметром /STACK или /F, внедряется в PE-заголовок .exe- или .dll-файла. Кроме того, объем изначально передаваемой памяти можно переопределить вызовом *CreateThread* или *_beginthreadex*. У обеих функций есть параметр, который позволяет изменять объем памяти, изначально передаваемой региону стека. Если в нем передать 0, система будет использовать значение, внедренное в PE-загловок. Далее я исхожу из того, что стек создается со стандартными параметрами.

На рис. 16-1 показано, как может выглядеть регион стека (зарезервированный по адресу 0x08000000) в системе с размером страниц по 4 Кб. Регион стека и вся переданная ему память имеют атрибут защиты PAGE_READWRITE.

Зарезервировав регион, система передает физическую память двум верхним его страницам. Непосредственно перед тем, как приступить к выполнению потока, система устанавливает регистр указателя стека на конец верхней страницы региона стека (адрес, очень близкий к 0x08100000). Это та

страница, с которой поток начнет использовать свой стек. Вторая страница сверху называется *сторожевой* (guard page).

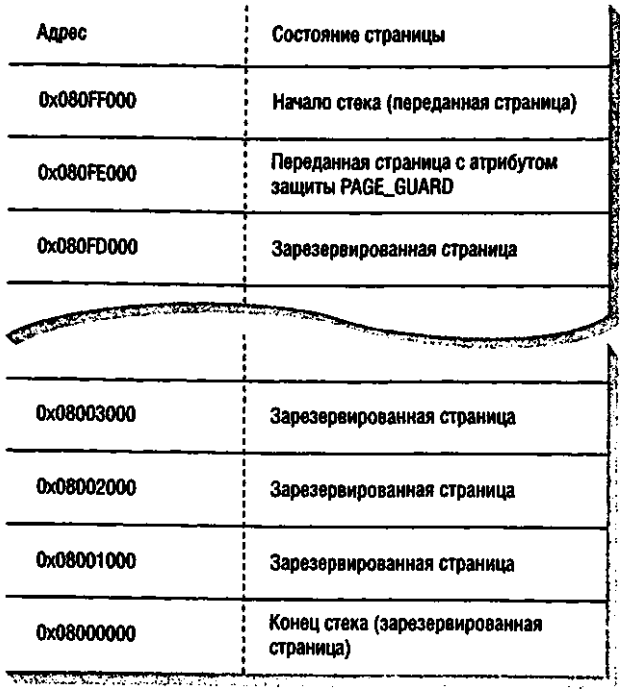


Рис. 16-1. *Так выглядит регион стека потока сразу после его создания*

По мере разрастания дерева вызовов (одновременного обращения ко все большему числу функций) потоку, естественно, требуется и больший объем стека. Как только поток обращается к следующей странице (а она сторожевая), система уведомляется об этой попытке. Тогда система передает память еще одной странице, расположенной как раз за сторожевой. После чего флаг PAGE_GUARD, как эстафетная палочка, переходит от текущей сторожевой к той странице, которой только что передана память. Благодаря такому механизму объем памяти, занимаемой стеком, увеличивается только по необходимости. Если дерево вызовов у потока будет расти и дальше, регион стека будет выглядеть примерно так, как показано на рис. 16-2.

Допустим, стек потока практически заполнен (как на рис. 16-2) и регистр указателя стека указывает на адрес 0x08003004. Тогда, как только поток вызовет еще одну функцию, система, по идее, должна передать дополнительную физическую память. Но когда система передает память странице по адресу 0x08001000, она делает это уже по-другому. Регион стека теперь выглядит, как на рис. 16-3.

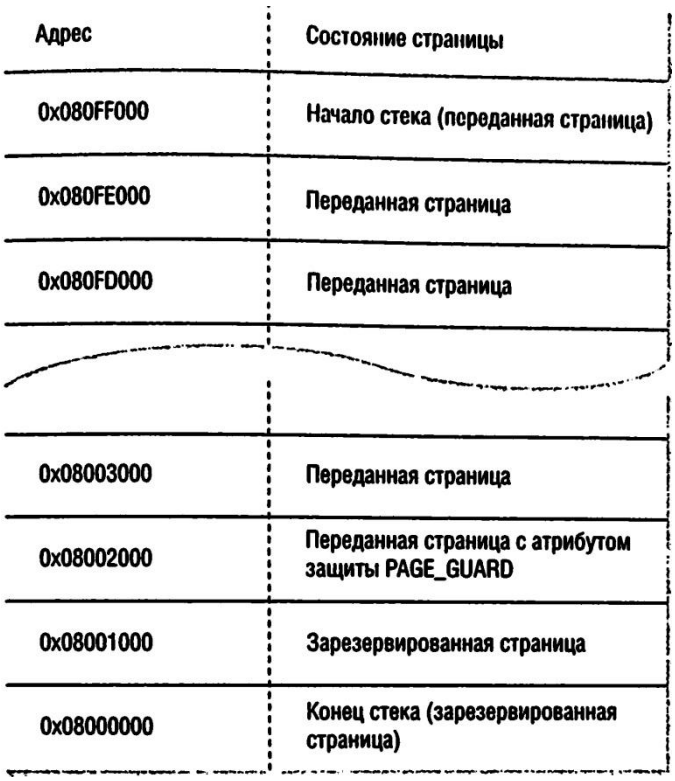


Рис. 16-2. *Почти заполненный регион стека потока*

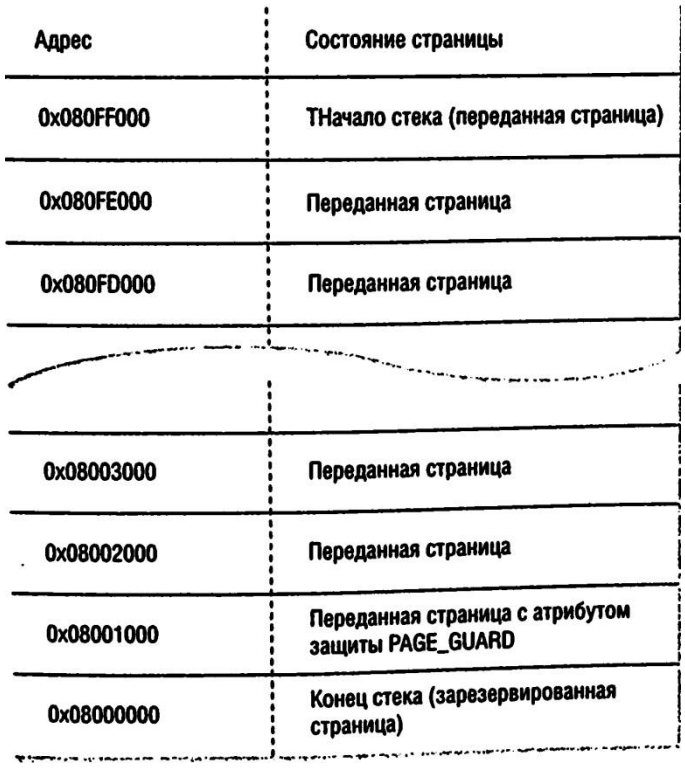
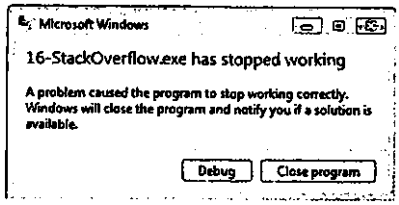


Рис. 16-3. *Целиком заполненный регион стека потока*

Как и можно было предполагать, флаг PAGE_GUARD со страницы по адресу 0x08002000 удаляется, а странице по адресу 0x08001000 передается физическая память. Но этой странице не присваивается флаг PAGE_GUARD. Это значит, что региону адресного пространства, зарезервированному под стек потока, теперь передана вся физическая память, которая могла быть ему передана. Самая нижняя страница остается зарезервированной, физическая память ей никогда не передается. Чуть позже я поясню, зачем это сделано.

Передавая физическую память странице по адресу 0x08001000, система выполняет еще одну операцию: генерирует исключение EXCEPTION_STACK_OVERFLOW (в файле WinNT.h оно определено как 0xC00000FD). При использовании структурной обработки исключений (SEH) ваша программа получит уведомление об этой ситуации и сможет корректно обработать ее. Подробнее о SEH см. главы 23, 24 и 25, а также листинг программы Summation, приведенный в конце этой главы.

Если поток продолжит использовать стек даже после исключения, связанного с переполнением стека, будет задействована вся память на странице по адресу 0x08001000, и поток попытается получить доступ к странице по адресу 0x08000000. Поскольку эта страница лишь зарезервирована (но не передана), возникнет исключение — нарушение доступа. Если это произойдет в момент обращения потока к стеку, вас ждут крупные неприятности. Система передаст управление службе Windows Error Reporting service, которая покажет следующее сообщение и завершит весь процесс (а не только поток, в котором возникла ошибка).



Приложение может заставить систему сгенерировать исключение EXCEPTION_STACK_OVERFLOW и раньше, вызвав функцию *SetThreadStackGuarantee*. Это позволяет гарантировать наличие свободной области заданного размера перед сторожевой страницей стека. Таким образом, в распоряжении приложения будет несколько страниц, прежде чем следующая попытка записи в стек заставит службу Windows Error Reporting прервать работу процесса.

> **Внимание!** Исключение EXCEPTION_STACK_OVERFLOW генерируется при обращении потока к сторожевой странице. Если поток перехватит это исключение и продолжит работу, повторных исключений не будет, поскольку других сторожевых страниц не предусмотрено. Чтобы в дальнейшем получить исключение EXCEPTION_STACK_OVERFLOW, поток должен сбросить сторожевую страницу вызовом функции _resetstkoflw из библиотеки (см. malloc.h).

Теперь объясню, почему нижняя страница стека всегда остается зарезервированной. Это позволяет защищать другие данные процесса от случайной перезаписи. Видите ли, по адресу 0x07FFF000 (на 1 страницу ниже, чем 0x08000000) может быть передана физическая память для другого региона адресного пространства. Если бы странице по адресу 0x08000000 была передана физическая память, система не сумела бы перехватить попытку потока расширить стек за пределы зарезервированного региона. А если бы стек расползся за пределы этого региона, поток мог бы перезаписать другие данные в адресном пространстве своего процесса — такого «жучка» выловить очень сложно.

Блок перед стеком предназначен для перехвата его переполнения, а блок после стека — для перехвата обращений к несуществующим областям стека. Чтобы понять, какая польза от последнего блока, рассмотрим такой фрагмент кода:

```
int WINAPI WinMain (HINSTANCE hInstExe, HINSTANCE,
    PTSTR pszCmdLine, int nCmdShow) {

    BYTE aBytes[100];
    aBytes[10000] = 0;                          // Stack underflow

    return(0);
}
```

Когда выполняется оператор присвоения, происходит попытка обращения за конец стека потока. Разумеется, ни компилятор, ни компоновщик не уловят эту ошибку в приведенном фрагменте кода, а нарушения доступа может и не случиться, если сразу за стеком потока окажется другой регион. И если вы случайно обратитесь за пределы стека, вы можете испортить содержимое области памяти, принадлежащей другой части вашего процесса, — система ничего *не заметит*. Вот пример кода, в котором обращение к области за пределами сетка неизбежно вызывает повреждение памяти, т.к. приложение выделяет блок памяти, прилегающий к нижней границе стека:

```
DWORD WINAPI ThreadFunc(PV0ID pvParam) {

    BYTE aBytes[0x10];

    // Определяем, где именно в виртуальном адресном пространстве находится стек;
    // подробнее о VirtualQuery см. в главе 14.
    MEM0RY_BASIC_INF0RMATI0N mbi;
    SIZE_T size = VirtualQuery(aBytes, &mbi, sizeof(mbi));

    // выделяем блок сразу после 1-Мб региона стека
    SIZE_T s = (SIZE_T)mbi.AllocationBase + 1024*1024;
    PBYTE pAddress = (PBYTE)s;
    BYTE* pBytes = (BYTE*)VirtualAlloc(pAddress, 0x10000,
       MEM_C0MMIT | MEM_RESERVE, PAGE_READWRITE);
```

```
    // Имитируем ошибку из-за обращения к памяти за пределами стека,
    // которая останется незамеченной.
    aBytes[0x10000] = 1; // записываем в блок, расположенный после стека
    …
    return(0);
}
```

Функция из библиотеки C/C++ для контроля стека

Библиотека C/C++ содержит функцию, позволяющую контролировать стек. Транслируя исходный код программы, компилятор при необходимости генерирует вызовы этой функции. Она обеспечивает корректную передачу страниц физической памяти стеку потока.

Возьмем, к примеру, небольшую функцию, требующую массу памяти под свои локальные переменные:

```
void SomeFunction () {
    int nValues[4000];

    // здесь что-то делаем с массивом
    nValues[0] = 0;                          // а тут что-то присваиваем
}
```

Для размещения целочисленного массива функция потребует минимум 16 000 байтов стекового пространства, так как каждое целое значение занимает 4 байта. Код, генерируемый компилятором, обычно выделяет такое пространство в стеке простым уменьшением указателя стека процессора на 16 000 байтов. Однако система не передаст физическую память этой нижней области стека, пока не произойдет обращения по данному адресу.

В системе с размером страниц по 4 или 8 Кб это могло бы создать проблему. Если первое обращение к стеку проходит по адресу, расположенному ниже сторожевой страницы (как в показанном выше фрагменте кода), поток обратится к зарезервированной памяти, и возникнет нарушение доступа. Поэтому, чтобы можно было спокойно писать функции вроде приведенной выше, компилятор и вставляет в код вызовы библиотечной функции для контроля стека.

При трансляции программы компилятору известен размер страниц памяти, используемых целевым процессором (4 Кб для *x*86 и 8 Кб для IA-64). Встречая в программе ту или иную функцию, компилятор определяет требуемый для нее объем стека и, если он превышает размер одной страницы, вставляет вызов функции, контролирующей стек.

Ниже показан псевдокод, который иллюстрирует, что именно делает функция, контролирующая стек. (Я говорю «псевдокод» потому, что обычно эта функция реализуется поставщиками компиляторов на языке ассемблера.)

```
// стандартной библиотеке C "известен" размер страницы в целевой системе
#ifdef _M_IA64
#define PAGESIZE     (8 * 1024)              // страницы по 8 Кб
#else
#define PAGESIZE     (4 * 1024)              // страницы по 4 Кб
#endif

void StackCheck(int nBytesNeededFromStack) {
    // Получим значение указателя стека. В этом месте указатель стека
    // еще НЕ был уменьшен для учета локальных переменных функции.
    PBYTE pbStackPtr = (указатель стека процессора);

    while (nBytesNeededFromStack >= PAGESIZE) {
        // смещаем страницу вниз по стеку - должна быть сторожевой
        pbStackPtr -= PAGESIZE;

        // обращаемся к какому-нибудь байту на сторожевой странице, вызывая
        // тем самым передачу новой страницы и сдвиг сторожевой страницы вниз
        pbStackPtr[0] = 0;

        // уменьшаем требуемое количество байтов в стеке
        nBytesNeededFromStack -= PAGESIZE;
    }

    // перед возвратом управления функция StackCheck устанавливает регистр
    // указателя стека на адрес, следующий за локальными переменными функции
}
```

В компиляторе Microsoft Visual C++ предусмотрен параметр **/GS**, позволяющий контролировать пороговый предел числа страниц, начиная с которого компилятор автоматически вставляет в программу вызов функции *StackCheck* (см. http://msdn2.microsoft.com/en-us/library/9598wk25(VS.80).aspx). Используйте этот параметр, только если вы точно знасте, что делаете, и если это действительно нужно. В 99,99999 процентах из ста приложения и DLL нс требуют применения упомянутого параметра.

> **Примечание.** Компилятор Microsoft C/C++ также поддерживает параметры для обнаружения повреждения стека во время выполнения. При отладочной (DEBUG) сборке C++-проекта по умолчанию активен параметр /RTCsu компилятора (см. http://msdn2.micwsoft.com/en-us/library/8wtf2dfz(VS.80).aspx). Если во время выполнения случится переполнение массива локальных переменных, вставленный компилятором код обнаружит это и уведомит вас при возврате управления сбойной функцией. Параметр /RTC поддерживается только для отладочной сборки. Для окончательной (RELEASE) сборки необходимо включить параметр /GS компилятора. Этот параметр заставляет компилятор добавить код, который записывает состояние стека в cookie-файл перед вызовом функций, а после вызова проверяет целостность функций. Такие предосто-

рожности блокируют попытки вредоносных программ инициировать переполнение стека с целью перехвата управления путем перезаписи адреса возврата в стеке. Сверка содержимого стека с cookie-файлом выявляет повреждение и работа приложения прерывается. Весьма детальный разбор параметра /GS compiler см. в статье http://www.symantec.com/avcenter/reference/GS_Protections_in_Vista.pdf.

Программа-пример Summation

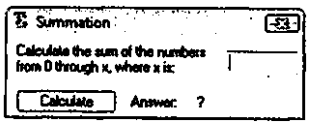
Эта программа, «16 Summation.exe» (см. листинг на рис. 16-6), демонстрирует использование фильтров и обработчиков исключений для корректного восстановления после переполнения стека. Файлы исходного кода и ресурсов этой программы находятся в каталоге 16-Summation внутри архива, доступного на сайте поддержки этой книги. Возможно, вам придется сначала прочесть главы по SEH, чтобы понять, как работает эта программа.

Она суммирует числа от 0 до x, где x — число, введенное пользователем. Конечно, проще было бы написать функцию с именем Sum, которая вычисляла бы по формуле:

```
Sum = (x * (x + 1)) / 2;
```

Но для этого примера я сделал функцию *Sum* рекурсивной, чтобы она использовала большое стековое пространство.

При запуске программы появляется диалоговое окно, показанное ниже.



В этом окне вы вводите число и щелкаете кнопку Calculate. Программа создает поток, единственная обязанность которого — сложить все числа от 0 до x. Пока он выполняется, первичный поток программы, вызвав *WaitForSingleObject*, просит систему не выделять ему процессорное время. Когда новый поток завершается, система вновь выделяет процессорное время первичному потоку. Тот выясняет сумму, получая код завершения нового потока вызовом *GetExitCodeThread*, и — это очень важно — закрывает свой описатель нового потока, так что система может уничтожить объект ядра «поток», и утечки ресурсов не произойдет.

Далее первичный поток проверяет код завершения суммирующего потока. Если он равен UINT_MAX, значит, произошла ошибка: суммирующий поток переполнил стек при подсчете суммы; тогда первичный поток выведет окно с соответствующим сообщением. Если же код завершения отличен от UINT_MAX, суммирующий поток отработал успешно; код завершения и есть искомая сумма. В этом случае первичный поток просто отображает результат суммирования в диалоговом окне.

Теперь обратимся к суммирующему потоку. Его функция — *SumThreadFunc*. При создании этого потока первичный поток передает ему в единственном параметре *pvParam* количество целых чисел, которые следует просуммировать. Затем его функция инициализирует переменную *uSum* значением UINT_MAX, т. е. изначально предполагается, что работа функции не завершится успехом. Далее *SumThreadFunc* активизирует SEH так, чтобы перехватывать любое исключение, возникающее при выполнении потока. После чего для вычисления суммы вызывается рекурсивная функция Sum.

Если сумма успешно вычислена, *SumThreadFunc* просто возвращает значение переменной *uSum*; оно и будет кодом завершения потока. Но, если при выполнении Sum возникает исключение, система сразу оценивает выражение в фильтре исключений. Иначе говоря, система вызывает *FilterFunc*, передавая ей код исключения. В случае переполнения стека этим кодом будет EXCEPTION_STACK_OVERFLOW. Чтобы увидеть, как программа обрабатывает исключение, вызванное переполнением стека, дайте ей просуммировать числа от 0 до 44000.

Моя функция *FilterFunc* очень проста. Сначала она проверяет, произошло ли исключение, связанное с переполнением стека. Если нет, возвращает EXCEPTION_CONTINUE_SEARCH, а если да — EXCEPTION_EXECUTE_HANDLER. Это подсказывает системе, что фильтр готов к обработке этого исключения и что надо выполнить код в блоке *except*. В данном случае обработчик исключения ничего особенного не делает, просто закрывая поток с кодом завершения UINT_MAX. Родительский поток, получив это специальное значение, выводит пользователю сообщение с предупреждением.

И последнее, что хотелось бы обсудить: почему я выделил функцию *Sum* в отдельный поток вместо того, чтобы просто создать SEH-фрейм в первичном потоке и вызывать *Sum* из его блока *try*. На то есть несколько причин.

Во-первых, всякий раз, когда создается поток, он получает стек размером 1 Мб. Если бы я вызывал *Sum* из первичного потока, часть стекового пространства уже была бы занята, и функция не смогла бы использовать весь объем стека. Согласен, моя программа очень проста и, может быть, не займет слишком большое стековое пространство. А если программа посложнее? Легко представить ситуацию, когда *Sum* подсчитывает сумму целых чисел от 0 до 1000 и стек вдруг оказывается чем-то занят, — тогда его переполнение произойдет, скажем, еще при вычислении суммы от 0 до 750. Таким образом, работа функции *Sum* будет надежнее, если предоставить ей полный стек, не используемый другим кодом.

Вторая причина в том, что поток уведомляется об исключении «переполнение стека» лишь однажды. Если бы я вызывал *Sum* из первичного потока и произошло бы переполнение стека, то это исключение было бы перехвачено и корректно обработано. Но к тому моменту физическая память была бы передана под все зарезервированное адресное пространство стека, и в нем уже не осталось бы страниц с флагом защиты. Начни пользователь новое

суммирование, и функция *Sum* переполнила бы стек, а соответствующее исключение не было бы возбуждено. Вместо этого возникло бы исключение «нарушение доступа», и корректно обработать эту ситуацию уже не удалось бы. Естественно, проблему можно было 6ы устранить вызовом библиотечной функции *_resetstkoflw*.

В-третьих, физическую память, отведенную под его стек, можно освободить. Рассмотрим такой сценарий: пользователь просит функцию *Sum* вычислить сумму целых чисел от 0 до 30 000. Это требует передачи региону стека весьма ощутимого объема памяти. Затем пользователь проводит несколько операций суммирования — максимум до 5000. И окажется, что стеку передан порядочный объем памяти, который больше не используется. А ведь эта физическая память выделяется из страничного файла. Так что лучше бы освободить ее и вернуть системе. И поскольку программа завершает поток *SumThreadFunc*, система автоматически освобождает физическую память, переданную региону стека.

И последнее, почему я использую отдельный поток: если поручить всю работу одному потоку, придется использовать синхронизацию, чтобы скоординировать исполнение потока и возврат результатов вычислений. Для такой простой программы, как в этом примере, проще всего породить для расчетов новый поток, передать ему входные данные для суммирования и дождаться результатов.

```
Summation.cpp
/******************************************************************************
Module:  Summation.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <limits.h>
#include <tchar.h>
#include "Resource.h"


//////////////////////////////////////////////////////////////////////////////


// программа вызывает Sum для uNum = 0 до 9
// uNum: 0 1 2 3  4  5  6  7  8  9 ...
// Sum:  0 1 3 6 10 15 21 28 36 45 ...
UINT Sum(UINT uNum) {

   // рекурсивный вызов Sum
   return((uNum == 0) ? 0 : (uNum + Sum(uNum - 1)));
}
```

```
//////////////////////////////////////////////////////////////////////////////


LONG WINAPI FilterFunc(DWORD dwExceptionCode) {

   return((dwExceptionCode == STATUS_STACK_OVERFLOW)
      ? EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}


//////////////////////////////////////////////////////////////////////////////


// отдельный поток, отвечающий за вычисление суммы.
// Я использую его по следующим причинам:
//    1. Отдельный поток получает собственный мегабайт стекового пространства.
//    2. Поток уведомляется о переполнении стека лишь однажды.
//    3. Память, выделенная для стека, освобождается по завершении потока.
DWORD WINAPI SumThreadFunc(PVOID pvParam) {

   // параметр pvParam определяет количество суммируемых чисел.
   UINT uSumNum = PtrToUlong(pvParam);

   // uSum содержит сумму чисел от 0 до uSumNum.
   // если сумму вычислить не удалось, возвращается значение UINT_MAX.
   UINT uSum = UINT_MAX;

   __try {
      // для перехвата исключения «переполнение стека»
      // функцию Sum надо выполнять в SEH-фрейме.
      uSum = Sum(uSumNum);
   }
   __except (FilterFunc(GetExceptionCode())) {
      // Если мы попали сюда, то это потому, что перехватили переполнение
      // стека. Здесь можно сделать все, что надо для корректного
      // возобновления работы. Но, так как от этого примера больше ничего
      // не требуется, кода в блоке обработчика нет.
   }

   // Кодом завершения потока является либо сумма первых uSumNum
   // чисел, либо UINT_MAX в случае переполнения стека.
   return(uSum);
}


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_SUMMATION);
```

```
   // мы принимаем не более чем девятизначные числа
   Edit_LimitText(GetDlgItem(hWnd, IDC_SUMNUM), 9);

   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;

      case IDC_CALC:
         // получаем количество целых числе, которые
         // пользователь хочет просуммировать
         BOOL bSuccess = TRUE;
         UINT uSum = GetDlgItemInt(hWnd, IDC_SUMNUM, &bSuccess, FALSE);
         if (!bSuccess) {
            MessageBox(hWnd, TEXT("Please enter a valid numeric value!"),
               TEXT("Invalid input..."), MB_ICONINFORMATION | MB_OK);
            SetFocus(GetDlgItem(hWnd, IDC_CALC));
            break;
         }

         // создаем поток (с собственным стеком), отвечающий за суммирование
         DWORD dwThreadId;
         HANDLE hThread = chBEGINTHREADEX(NULL, 0,
            SumThreadFunc, (PVOID) (UINT_PTR) uSum, 0, &dwThreadId);

         // ждем завершения потока.
         WaitForSingleObject(hThread, INFINITE);

         // код завершения — результат суммирования.
         GetExitCodeThread(hThread, (PDWORD) &uSum);

         // закончив, закрываем описатель потока,
         // чтобы система могла разрушить объект ядра «поток»
         CloseHandle(hThread);

         // обновляем содержимое диалогового окна.
         if (uSum == UINT_MAX) {
            // если код завершения равен UINT_MAX,
```

```
            // произошло переполнение стека.
            SetDlgItemText(hWnd, IDC_ANSWER, TEXT("Error"));
            chMB("The number is too big, please enter a smaller number");
         } else {
            // сумма вычислена успешно
            SetDlgItemInt(hWnd, IDC_ANSWER, uSum, FALSE);
         }
         break;
   }
}

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR, int) {

   DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SUMMATION), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

Оглавление

ГЛАВА 17

Проецируемые в память файлы

Операции с файлами — это то, что рано или поздно приходится делать практически во всех программах, и всегда это вызывает массу проблем. Должно ли приложение просто открыть файл, считать и закрыть его, или открыть, считать фрагмент в буфер и перезаписать его в другую часть файла? В Windows многие из этих проблем решаются очень изящно — с помощью проецируемых в память файлов (memory-mapped files).

Как и виртуальная память, проецируемые файлы позволяют резервировать регион адресного пространства и передавать ему физическую память. Различие между этими механизмами состоит в том, что в последнем случае физическая память не выделяется из страничного файла, а берется из файла, уже находящегося на диске. Как только файл спроецирован в память, к нему можно обращаться так, будто он целиком в нее загружен.

Проецируемые файлы применяются для:

- загрузки и выполнения EXE- и DLL-файлов. Это позволяет существенно экономить как на размере страничного файла, так и на времени, необходимом для подготовки приложения к выполнению;
- доступа к файлу данных, размещенному на диске. Это позволяет обойтись без операций файлового ввода-вывода и буферизации его содержимого;
- разделения данных между несколькими процессами, выполняемыми на одной машине. (В Windows есть и другие методы для совместного доступа разных процессов к одним данным — но все они так или иначе реализованы на основе проецируемых в память файлов.)

Эти области применения проецируемых файлов мы и рассмотрим в данной главе.

Проецирование в память EXE- и DLL-файлов

При вызове из потока функции *CreateProcess* система действует так:

1. Отыскивает EXE-файл, указанный при вызове *CreateProcess*. Если файл не найден, новый процесс не создается, а функция возвращает FALSE.
2. Создает новый объект ядра «процесс».
3. Создает адресное пространство нового процесса.
4. Резервирует регион адресного пространства — такой, чтобы в него поместился данный EXE-файл. Желательное расположение этого региона указывается внутри самого EXE-файла. По умолчанию базовый адрес EXE-файла — 0x00400000 (в 64-разрядном приложении под управлением 64-разрядной Windows этот адрес может быть другим). При создании исполняемого файла приложения базовый адрес может быть изменен через параметр компоновщика /BASE.
5. Отмечает, что физическая память, связанная с зарезервированным регионом, — EXE-файл на диске, а не страничный файл.

Спроецировав EXE-файл на адресное пространство процесса, система обращается к разделу EXE-файла со списком DLL, содержащих необходимые программе функции. После этого система, вызывая *LoadLibrary*, поочередно загружает указанные (а при необходимости и дополнительные) DLL-модули. Всякий раз, когда для загрузки DLL вызывается *LoadLibrary*, система выполняет действия, аналогичные описанным выше в пп. 4 и 5:

1. Резервирует регион адресного пространства — такой, чтобы в него мог поместиться заданный DLL-файл. Желательное расположение этого региона указывается внутри самого DLL-файла. По умолчанию Microsoft Visual C++ присваивает DLL-модулям базовый адрес 0x10000000 (в 64-разрядной DLL под управлением 64-разрядной Windows 2000 этот адрес может быть другим). При компоновке DLL это значение можно изменить с помощью параметра /BASE. У всех стандартных системных DLL, поставляемых с Windows, разные базовые адреса, чтобы не допустить их перекрытия при загрузке в одно адресное пространство.
2. Если зарезервировать регион по желательному для DLL базовому адресу не удается (из-за того, что он слишком мал либо занят каким-то еще EXE-или DLL-файлом), система пытается найти другой регион. Но по двум причинам такая ситуация весьма неприятна. Во-первых, если в DLL нет информации о возможной переадресации (relocation information), загрузка может вообще не получиться. (Такую информацию можно удалить из DLL при компоновке с параметром /FIXED. Это уменьшит размер DLL-файла, но тогда модуль *должен* грузиться только по указанному базовому адресу.) Во-вторых, системе приходится выполнять модификацию адресов (relocations) внутри DLL. В Windows на это уходит дополнительная физическая память, выделяемая из страничного файла, да и загрузка такой DLL займет больше времени.

3. Отмечает, что физическая память, связанная с зарезервированным регионом, — DLL-файл на диске, а не страничный файл. Если Windows пришлось выполнять модификацию адресов из-за того, что DLL не удалось загрузить по желательному базовому адресу, она запоминает, что часть физической памяти для DLL связана со страничным файлом.

Если система почему-либо не свяжет EXE-файл с необходимыми ему DLL, на экране появится соответствующее сообщение, а адресное пространство процесса и объект «процесс» будут освобождены. При этом *CreateProcess* вернет FALSE; прояснить причину сбоя поможет функция *GetLastError*.

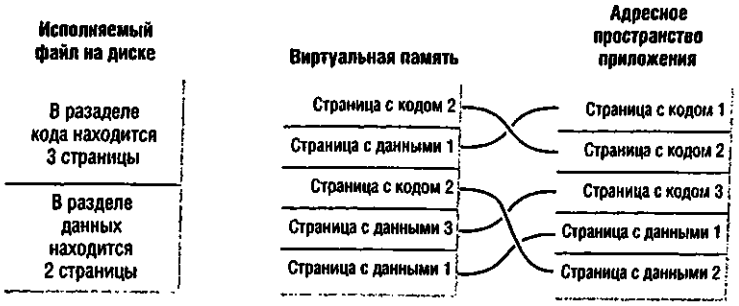
После увязки EXE- и DLL-файлов с адресным пространством процесса начинает исполняться стартовый код EXE-файла. Подкачку страниц, буферизацию и кэширование система берет на себя. Например, если код в EXE-файле переходит к команде, не загруженной в память, возникает ошибка. Обнаружив ее, система перекачивает нужную страницу кода из образа файла на страницу оперативной памяти. Затем отображает страницу оперативной памяти на должный участок адресного пространства процесса, тем самым позволяя потоку продолжить выполнение кода. Все эти операции скрыты от приложения и периодически повторяются при каждой попытке процесса обратиться к коду или данным, отсутствующим в оперативной памяти.

Статические данные не разделяются несколькими экземплярами EXE или DLL

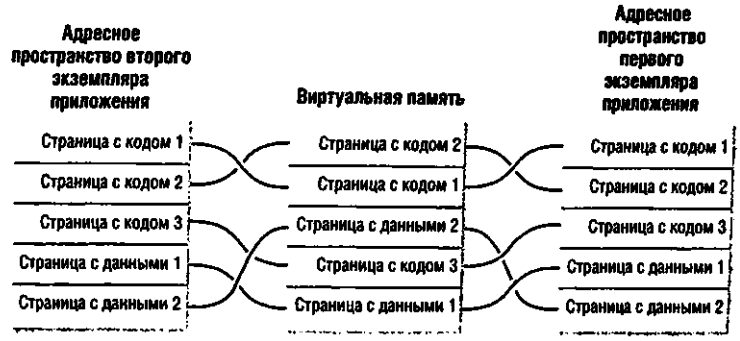
Когда вы создаете новый процесс для уже выполняемого приложения, система просто открывает другое проецируемое в память представление (view) объекта «проекция файла» (file-mapping object), идентифицирующего образ исполняемого файла, и создает новые объекты «процесс» и «поток» (для первичного потока). Этим объектам присваиваются идентификаторы процесса и потока. С помощью проецируемых в память файлов несколько одновременно выполняемых экземпляров приложения может совместно использовать один и тот же код, загруженный в оперативную память.

Здесь возникает небольшая проблема. Процессы используют линейное (flat) адресное пространство. При компиляции и компоновке программы весь ее код и данные объединяются в нечто, так сказать, большое и цельное. Данные, конечно, отделены от кода, но только в том смысле, что они расположены вслед за кодом в EXE-файле[1]. Вот упрощенная иллюстрация того, как код и данные приложения загружаются в виртуальную память, а затем отображаются на адресное пространство процесса:

[1] Надо найти



Теперь допустим, что запущен второй экземпляр программы. Система просто-напросто проецирует страницы виртуальной памяти, содержащие код и данные файла, на адресное пространство второго экземпляра приложения:

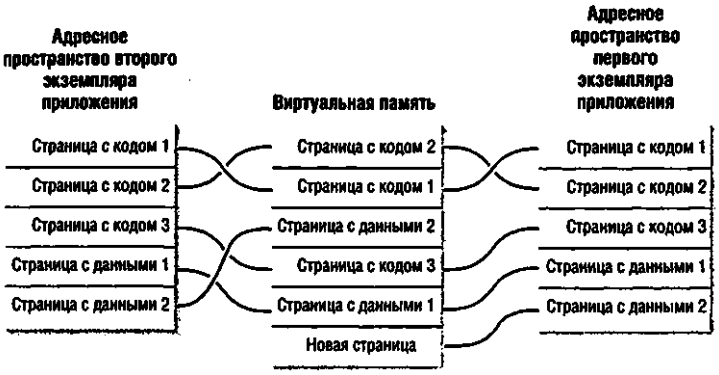


> **Примечание.** В действительности содержимое файла разбивается на два раздела. В одном разделе находится код, а в другом — глобальные переменные. Эти секции выравниваются по размеру страниц. Приложение может определить размер страницы, вызвав *GetSystemInfo*. В .exe- или . dll-файле раздела кода обычно предшествует разделу данных.

Если один экземпляр приложения модифицирует какие-либо глобальные переменные, размещенные на странице данных, содержимое памяти изменяется для всех экземпляров этого приложения. Такое изменение могло бы привести к катастрофическим последствиям и поэтому недопустимо.

Система предотвращает подобные ситуации, применяя механизм копирования при записи. Всякий раз, когда программа пытается записывать что-то в файл, спроецированный в память, система перехватывает эту попытку, выделяет новый блок памяти, копирует в него нужную программе страницу и после этого разрешает запись в новый блок памяти. Благодаря этому работа остальных экземпляров программы не нарушается. Вот что получится,

когда первый экземпляр программы попытается изменить какую-нибудь глобальную переменную на второй странице данных:



Система выделяет новую страницу и копирует на нее содержимое страницы данных 2. Адресное пространство первого экземпляра изменяется так, чтобы отобразить новую страницу данных на тот же участок, что и исходную. Теперь процесс может изменить глобальную переменную, не затрагивая данные другого экземпляра.

Аналогичная цепочка событий происходит и при отладке приложения. Например, запустив несколько экземпляров программы, вы хотите отладить только один из них. Вызвав отладчик, вы ставите в строке исходного кода точку прерывания. Отладчик модифицирует ваш код, заменяя одну из команд на языке ассемблера другой — заставляющей активизировать сам отладчик. И здесь вы сталкиваетесь с той же проблемой. После модификации кода все экземпляры программы, доходя до исполнения измененной команды, приводили бы к его активизации. Чтобы этого избежать, система вновь использует копирование при записи. Обнаружив попытку отладчика изменить код, она выделяет новый блок памяти, копирует туда нужную страницу и позволяет отладчику модифицировать код на этой копии.

> **Примечание.** При загрузке процесса система просматривает все страницы образа файла. Физическая память из страничного файла передается сразу только тем страницам, которые должны быть защищены атрибутом копирования при записи. При обращении к такому участку образа файла в память загружается соответствующая страница. Если ее модификации не происходит, она может быть выгружена из памяти и при необходимости загружена вновь. Если же страница файла модифицируется, система перекачивает ее на одну из ранее переданных страниц в страничном файле.

Статические данные разделяются несколькими экземплярами EXE или DLL

По умолчанию для большей безопасности глобальные и статические данные нс разделяются несколькими проекциями одного и того же EXE или DLL. Но иногда удобнее, чтобы несколько проекций EXE разделяли единственный экземпляр переменной. Например, в Windows не так-то просто определить, запущено ли несколько экземпляров приложения. Если бы у вас была переменная, доступная всем экземплярам приложения, она могла бы отражать число этих экземпляров. Тогда при запуске нового экземпляра приложения его поток просто проверил бы значение глобальной переменной (обновленное другим экземпляром приложения) и, будь оно больше 1, сообщил бы пользователю, что запустить можно лишь один экземпляр; после чего эта копия приложения была бы завершена.

В этом разделе мы рассмотрим метод, обеспечивающий совместное использование переменных всеми экземплярами EXE или DLL. Но сначала вам понадобятся кое-какие базовые сведения.

Любой образ EXE- или DLL-файла состоит из группы разделов. По соглашению имя каждого стандартного раздела начинается с точки. Например, при компиляции программы весь код помещается в раздел *.text*, неинициализированные данные — в раздел *.bss*, а инициализированные — в раздел *.data*.

С каждым разделом связана одна из комбинаций атрибутов, перечисленных в следующей таблице.

Табл. 17-1. Атрибуты разделов

Атрибут	Описание
READ	Разрешает чтение из раздела
WRITE	Разрешает запись в раздел
EXECUTE	Содержимое раздела можно исполнять
SHARED	Раздел доступен нескольким экземплярам приложения (этот атрибут отключает механизм копирования при записи)

Запустив утилиту DumpBin из Microsoft Visual Studio (с ключом /Headers), вы увидите список разделов в файле образа EXE или DLL Пример такого списка, показанный ниже, относится к EXE-файлу.

```
SECTION HEADER #1
    .text name
    11A70 virtual size
     1000 virtual address
    12000 size of raw data
     1000 file pointer to raw data
        0 file pointer to relocation table
        0 file pointer to line numbers
        0 number of relocations
```

```
       0 number of line numbers
60000020 flags
         Code
         Execute Read

SECTION HEADER #2
  .rdata name
     1F6 virtual size
   13000 virtual address
    1000 size of raw data
   13000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
40000040 flags
         Initialized Data
         Read 0nly

SECTION HEADER #3
   .data name
     560 virtual size
   14000 virtual address
    1000 size of raw data
   14000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write

SECTION HEADER #4
  .idata name
     58D virtual size
   15000 virtual address
    1000 size of raw data
   15000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write
```

```
SECTION HEADER 05
  .didat name
     7A2 virtual size
   16000 virtual address
    1000 size of raw data
   16000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
C0000040 flags
         Initialized Data
         Read Write

SECTION HEADER #6
  .reloc name
     26D virtual size
   17000 virtual address
    1000 size of raw data
   17000 file pointer to raw data
       0 file pointer to relocation table
       0 file pointer to line numbers
       0 number of relocations
       0 number of line numbers
42000040 flags
         Initialized Data
         Discardable
         Read Only

Summary
    1000 .data
    1000 .didat
    1000 .idata
    1000 .rdata
    1000 .reloc
   12000 .text
```

Некоторые из часто встречающихся разделов перечислены в таблице ниже.

Табл. 17-2. Общие разделы исполняемых файлов

Имя раздела	Описание
.bss	Неинициализированные данные
.CRT	Неизменяемые данные библиотеки C
.data	Инициализированные данные

Табл. 17-2. *(окончание)*

Имя раздела	Описание
.debug	Отладочная информация
.didat	Таблица имен для отложенного импорта (delay imported names table)
.edata	Таблица экспортируемых имен
.idata	Таблица импортируемых имен
.rdata	Неизменяемые данные периода выполнения
.reloc	Настроечная информация — таблица переадресации (relocation table)
.rsrc	Ресурсы
.text	Код EXE или DLL
.tls	Локальная память потока
.xdata	Таблица для обработки исключений

Кроме стандартных разделов, генерируемых компилятором и компоновщиком, можно создавать свои разделы в EXE- или DLL-файле, используя директиву компилятора:

```
#pragma data_seg("sectionname")
```

Например, можно создать раздел Shared, в котором содержится единственная переменная типа LONG:

```
#pragma data_seg("Shared")
L0NG g_lInstanceCount = 0;
#pragma data_seg()
```

Обрабатывая этот код, компилятор создаст раздел *Shared* и поместит в него все *инициализированные* переменные, встретившиеся после директивы *#pragma*. В нашем примере в этом разделе находится переменная *g_lInstanceCount*. Директива *#pragma data_seg()* сообщает компилятору, что следующие за ней переменные нужно вновь помещать в стандартный раздел данных, а не в Shared. Важно помнить, что компилятор помещает в новый раздел только инициализированные переменные. Если из предыдущего фрагмента кода исключить инициализацию переменной, она будет включена в другой раздел:

```
#pragma data_seg("Shared")
L0NG g_lInstanceCount;
#pragma data_seg()
```

Однако в компиляторе Microsoft Visual C++ предусмотрен спецификатор *allocate*, который позволяет помещать неинициализированные данные в любой раздел. Взгляните на этот код:

```
// создаем раздел Shared и заставляем компилятор
// поместить в него инициализированные данные
#pragma data_seg("Shared")

// инициализированная переменная, по умолчанию помещается в раздел Shared
int a = 0;

// неинициализированная переменная, по умолчанию помещается в другой раздел
int b;

// приказываем компилятору прекратить включение инициализированных данных
// в раздел Shared
#pragma data_seg()

// инициализированная переменная, принудительно помещается в раздел Shared
__declspec(allocate("Shared")) int c = 0;

// неинициализированная переменная, принудительно помещается в раздел Shared
__declspec(allocate("Shared")) int d;

// инициализированная переменная, по умолчанию помещается в другой раздел
int e = 0;

// неинициализированная переменная, по умолчанию помещается в другой раздел
int f;
```

Чтобы спецификатор *allocate* работал корректно, сначала должен быть создан соответствующий раздел. Так что, убрав из предыдущего фрагмента кода первую строку *#pragma data_seg*, вы не смогли бы его скомпилировать.

Чаще всего переменные помещают и собственные разделы, намереваясь сделать их разделяемыми между несколькими проекциями EXE или DLL. По умолчанию каждая проекция получает свой набор переменных. Но можно сгруппировать в отдельном разделе переменные, которые должны быть доступны всем проекциям EXE или DLL; тогда система не станет создавать новые экземпляры этих переменных для каждой проекции EXE или DLL.

Чтобы переменные стали разделяемыми, одного указания компилятору выделить их в какой-то раздел мало. Надо также сообщить компоновщику, что переменные в этом разделе должны быть общими. Для этого предназначен ключ /SECTION компоновщика:

```
/SECTI0N:имя,атрибуты
```

За двоеточием укажите имя раздела, атрибуты которого вы хотите изменить. В нашем примере нужно изменить атрибуты раздела Shared, поэтому ключ должен выглядеть так:

```
/SECTION:Shared,RWS
```

После запятой мы задаем требуемые атрибуты. При этом используются такие сокращения: R (READ), W(WRITE), E (EXECUTE) и S (SHARED). В данном случае мы указали, что раздел Shared должен быть ««читаемым», «записываемым» и «разделяемым». Если вы хотите изменить атрибуты более чем у одного раздела, указывайте ключ /SECTION для каждого такого раздела.

Соответствующие директивы для компоновщика можно вставлять прямо в исходный код:

```
#pragma comment(linker, "/SECTION:Shared,RWS")
```

Эта строка заставляет компилятор включить строку «/SECTION: Shared,RWS» в особый раздел *.drectve*. Компоновщик, собирая OBJ-модули, проверяет этот раздел в каждом OBJ-модуле и действует так, словно все эти строки переданы ему как аргументы в командной строке. Я всегда применяю этот очень удобный метод: перемещая файл исходного кода в новый проект, не надо изменять никаких параметров в диалоговом окне Project Settings в Visual C++.

Хотя создавать общие разделы можно, Майкрософт не рекомендует это делать. Во-первых, разделение памяти таким способом может нарушить защиту. Во-вторых, наличие общих переменных означает, что ошибка в одном приложении повлияет на другое, так как этот блок данных не удастся защитить от случайной записи.

Представьте, вы написали два приложения, каждое из которых требует от пользователя вводить пароль. При этом вы решили чуть-чуть облегчить жизнь пользователю: если одна из программ уже выполняется на момент запуска другой, то вторая считывает пароль из общей памяти. Так что пользователю не нужно повторно вводить пароль, если одно из приложений уже запущено.

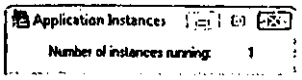
Все выглядит вполне невинно. В конце концов, только ваши приложения загружают данную DLL, и только они знают, где искать пароль, содержащийся в общем разделе памяти. Но хакеры не дремлют, и если им захочется узнать ваш пароль, то максимум, что им понадобится, — написать небольшую программу, загружающую вашу DLL, и понаблюдать за общим блоком памяти. Когда пользователь введет пароль, хакерская программа тут же его узнает.

Трудолюбивая хакерская программа может также предпринять серию попыток угадать пароль, записывая его варианты в общую память. А угадав, сможет посылать любые команды этим двум приложениям. Данную проблему можно было бы решить, если бы существовал какой-нибудь способ разрешать загрузку DLL только определенным программам. Но пока это невозможно — любая программа, вызвав *LoadLibrary*, способна явно загрузить любую DLL.

Программа-пример AppInst

Эта программа (17 AppInst.exe), демонстрирует, как выяснить, сколько экземпляров приложения уже выполняется в системе. Файлы исходного кода

и ресурсов этой программы находятся в каталоге 17-AppInst внутри архива, доступного на веб-сайте поддержки этой книги. После запуска AppInst на экране появляется диалоговое окно, в котором сообщается, что сейчас выполняется только один се экземпляр.



Если вы запустите второй экземпляр, оба диалоговых окна сообщат, что теперь выполняется два экземпляра.

Вы можете запускать и закрывать сколько угодно экземпляров этой программы — окно любого из них всегда будет отражать точное количество выполняемых экземпляров.

Где-то в начале файла AppInst.cpp вы заметите следующие строки:

```
// указываем компилятору поместить эту инициализированную переменную
// в раздел Shared, чтобы она стала доступной всем экземплярам программы
#pragma data_seg("Shared")
volatile LONG g_lApplicationInstances = 0;
#pragma data_seg()

// указываем компоновщику, что раздел Shared должен быть
// читаемым, записываемым и разделяемым
#pragma comment(linker, "/Section:Shared,RWS")
```

В этих строках кода создается раздел Shared с атрибутами защиты, которые разрешают его чтение, запись и разделение. Внутри него находится одна переменная, *g_lApplicationInstances*, доступная всем экземплярам программы. Заметьте, что для этой переменной указан спецификатор *volatile*, чтобы оптимизатор не слишком с ней умничал.

При выполнении функции *_tWinMain* каждого экземпляра значение переменной *g_lApplicationInstances* увеличивается на 1, а перед выходом из *_tWinMain* — уменьшается на 1. Я изменяю ее значение с помощью функции *InterlockedExchangeAdd*, так как эта переменная является общим ресурсом для нескольких потоков.

Когда на экране появляется диалоговое окно каждого экземпляра программы, вызывается функция *Dlg_OnInitDialog*. Она рассылает всем окнам верхнего уровня зарегистрированное оконное сообщение (идентификатор которого содержится в переменной *g_uMsgAppInstCountUpdate*):

```
PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);
```

Это сообщение игнорируется всеми окнами в системе, кроме окон AppInst. Когда его принимает одно из окон нашей программы, код в *Dlg_Proc* просто обновляет в диалоговом окне значение, отражающее текущее количество экземпляров (а эта величина хранится в переменной *g_lApplicationInstances*).

AppInst.cpp

```
/******************************************************************************
Module:  AppInst.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////

// общесистемное оконное сообщение с уникальным идентификатором
UINT g_uMsgAppInstCountUpdate = WM_APP+123;

//////////////////////////////////////////////////////////////////////////////

// указывает компилятору поместить эту инициализированную переменную
// в раздел Shared, чтобы она стала доступной всем экземплярам программы.
#pragma data_seg("Shared")
volatile LONG g_lApplicationInstances = 0;
#pragma data_seg()

// указываем компоновщику, что раздел Shared должен быть
// читаемым, записываемым и разделяемым.
#pragma comment(linker, "/Section:Shared,RWS")

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_APPINST);

   // Force the static control to be initialized correctly.
   PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);
   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////
```

```
void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;
   }
}

///////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   if (uMsg == g_uMsgAppInstCountUpdate) {
      SetDlgItemInt(hWnd, IDC_COUNT, g_lApplicationInstances, FALSE);
   }

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}

///////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR, int) {

   // получаем числовое значение из общесистемного оконного сообщения,
   // которое применяется для уведомления всех окон верхнего уровня
   // об изменении счетчика числа пользователей данного модуля.
   g_uMsgAppInstCountUpdate =
      RegisterWindowMessage(TEXT("MsgAppInstCountUpdate"));

   // запущен еще один экземпляр этой программы
   InterlockedExchangeAdd(&g_lApplicationInstances, 1);

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_APPINST), NULL, Dlg_Proc);

   // данный экземпляр закрывается
   InterlockedExchangeAdd(&g_lApplicationInstances, -1);

   // сообщаем об этом остальным экземплярам программы.
   PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);
```

```
   return(0);
}

//////////////////////////////// End of File ///////////////////////////////////
```

Файлы данных, проецируемые в память

Операционная система позволяет проецировать на адресное пространство процесса и файл данных. Это очень удобно при манипуляциях с большими потоками данных.

Чтобы представить всю мощь такого применения механизма проецирования файлов, рассмотрим четыре возможных метода реализации программы, меняющей порядок следования всех байтов в файле на обратный.

Метод 1: один файл, один буфер

Первый (и теоретически простейший) метод — выделение блока памяти, достаточного для размещения всего файла. Открываем файл, считываем его содержимое в блок памяти, закрываем. Располагая в памяти содержимым файла, можно поменять первый байт с последним, второй — с предпоследним и т. д. Этот процесс будет продолжаться, пока мы не поменяем местами два смежных байта, находящихся в середине файла. Закончив эту операцию, вновь открываем файл и перезаписываем его содержимое.

Этот довольно простой в реализации метод имеет два существенных недостатка. Во-первых, придется выделить блок памяти такого же размера, что и файл. Это терпимо, если файл небольшой. А если он занимает 2 Гб? Система просто не позволит приложению передать такой объем физической памяти. Значит, к большим файлам нужен совершенно иной подход.

Во-вторых, если перезапись вдруг прервется, содержимое файла будет испорчено. Простейшая мера предосторожности — создать копию исходного файла (потом ее можно удалить), но это потребует дополнительного дискового пространства.

Метод 2: два файла, один буфер

Открываем существующий файл и создаем на диске новый — нулевой длины. Затем выделяем небольшой внутренний буфер размером, скажем, 8 Кб. Устанавливаем указатель файла в позицию 8 Кб от конца, считываем в буфер последние 8 Кб содержимого файла, меняем в нем порядок следования байтов на обратный и переписываем буфер в только что созданный файл. Повторяем эти операции, пока не дойдем до начала исходного файла. Конечно, если длина файла не будет кратна 8 Кб, операции придется немного усложнить, но это не страшно. Закончив обработку, закрываем оба файла и удаляем исходный файл.

Этот метод посложнее первого, зато позволяет гораздо эффективнее использовать память, так как требует выделения лишь 8 Кб. Но и здесь не без

проблем, и вот две главных. Во-первых, обработка идет медленнее, чем при керном методе: на каждой итерации перед считыванием приходится находить нужный фрагмент исходного файла. Во-вторых, может понадобиться огромное пространство па жестком диске. Если длина исходного файла 1 Гб, новый файл постепенно вырастет до этой величины, и перед самым удалением исходного файла будет занято 2 Гб, т. е. на 1 Гб больше, чем следовало бы. Так что все пути ведут... к третьему методу.

Метод 3: один файл, два буфера

Программа инициализирует два раздельных буфера, допустим, по 8 Кб и считывает первые 8 Кб файла в один буфер, а последние 8 Кб — в другой. Далее содержимое обоих буферов обменивается в обратном порядке и первый буфер записывается в конец, а второй — в начало того же файла. На каждой итерации программа перемещает восьмикилобайтовые блоки из одной половины файла в другую. Разумеется, нужно предусмотреть какую-то обработку на случай, если длина файла не кратна 16 Кб, и эта обработка будет куда сложнее, чем в предыдущем методе. Но разве это испугает опытного программиста?

По сравнению с первыми двумя этот метод позволяет экономить пространство на жестком диске, так как все операции чтения и записи протекают в рамках одного файла. Что же касается памяти, то и здесь данный метод довольно эффективен, используя всего 16 Кб. Однако он, по-видимому, самый сложный в реализации. И, кроме того, как и первый метод, он может испортить файл данных, если процесс вдруг прервется.

Ну а теперь посмотрим, как тот же процесс реализуется, если применить файлы, проецируемые в память.

Метод 4: один файл и никаких буферов

Вы открываете файл, указывая системе зарезервировать регион виртуального адресного пространства. Затем сообщаете, что первый байт файла следует спроецировать на первый байт этого региона, и обращаетесь к региону так, будто он на самом деле содержит файл. Если в конце файла есть отдельный нулевой байт, можно вызвать библиотечную функцию *_strrev* и поменять порядок следования байтов на обратный.

Огромный плюс этого метода в том, что всю работу по кэшированию файла выполняет сама система: не надо выделять память, загружать данные из файла в память, переписывать их обратно в файл и т. д. и т. п. Но, увы, вероятность прерывания процесса, например из-за сбоя в электросети, по-прежнему сохраняется, и от порчи данных вы не застрахованы.

Использование проецируемых в память файлов

Для этого нужно выполнить три операции:

1. Создать или открыть объект ядра «файл», идентифицирующий дисковый файл, который вы хотите использовать как проецируемый в память.

2. Создать объект ядра «проекция файла», чтобы сообщить системе размер файла и способ доступа к нему.
3. Указать системе, как спроецировать в адресное пространство вашего процесса объект «проекция файла» — целиком или частично.

Закончив работу с проецируемым в память файлом, следует выполнить тоже три операции:

1. Сообщить системе об отмене проецирования на адресное пространство процесса объекта ядра «проекция файла».
2. Закрыть этот объект.
3. Закрыть объект ядра«файл».

Детальное рассмотрение этих операций — в следующих пяти разделах.

Этап 1: создание или открытие объекта ядра «файл»

Для этого вы должны применять только функцию *CreateFile*:

```
HANDLE CreateFile(
    PCSTR pszFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDisposition,
    DWORD dwFlagsAndAttributes,
    HANDLE hTemplateFile);
```

Как видите, у функции *CreateFile* довольно много параметров. Здесь я сосредоточусь только на первых трех: *pszFileName*, *dwDesiredAccess* и *dwShareMode*.

Как вы, наверное, догадались, первый параметр, *pszFileName*, идентифицирует имя создаваемого или открываемого файла (при необходимости вместе с путем). Второй параметр, *dwDesiredAccess*, указывает способ доступа к содержимому файла. Здесь задается одно из четырех значений, показанных в таблице ниже.

Табл. 17-3. Права доступа к файлу

Значение	Описание
0	Содержимое файла нельзя считывать или записывать; указывайте это значение, если вы хотите всего лишь получить атрибуты файла
GENERIC_READ	Чтение файла разрешено
GENERIC_WRITE	Запись в файл разрешена
GENERIC_READ | GENERIC_WRITE	Разрешено и то и другое

Создавая или открывая файл данных с намерением использовать его в качестве проецируемого в память, можно установить либо флаг GENERIC_

READ (только для чтения), либо комбинированный флаг GENERIC_READ | GENERIC_WRITE (чтение/запись).

Третий параметр, *dwShareMode*, указывает тип совместного доступа к данному файлу (см. следующую таблицу).

Табл. 17-4. Режимы совместного доступа к файлу

Значение	Описание
0	Другие попытки открыть файл закончатся неудачно
FILE_SHARE_READ	Попытка постороннего процесса открыть файл с флагом GENERIC_WRITE не удастся
FILE_SHARE_WRITE	Попытка постороннего процесса открыть файл с флагом GENERIC_READ не удастся
FILE_SHARE_READ| FILE_SHARE_WRITE	Посторонний процесс может открывать файл без ограничений

Создав или открыв указанный файл, *CreateFile* возвращает его описатель, в ином случае — идентификатор INVALID_HANDLE_VALUE.

> **Примечание.** Большинство функций Windows, возвращающих те или иные описатели, при неудачном вызове дает NULL. Но *CreateFile* — исключение и в таких случаях возвращает идентификатор INVALID_HANDLE_VALUE, определенный как ((HANDLE) - 1).

Этап 2: создание объекта ядра «проекция файла»

Вызвав *CreateFile*, вы указали операционной системе, где находится физическая память для проекции файла: на жестком диске, в сети, на CD-ROM или в другом месте. Теперь сообщите системе, какой объем физической памяти нужен проекции файла. Для этого вызовите функцию *CreateFileMapping*.

```
HANDLE CreateFileMapping(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD fdwProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaximumSizeLow,
    PCTSTR pszName);
```

Первый параметр, *hFile*, идентифицируетописательфайла, проецируемого на адресное пространство процесса. Этот описатель вы получили после вызова *CreateFile*. Параметр *psa* — указатель на структуру SECURITY_ATTRIBUTES, которая относится к объекту ядра «проекция файла»; для установки защиты по умолчанию ему присваивается NULL.

Как я уже говорил в начале этой главы, создание файла, проецируемого в намять, аналогично резервированию региона адресного пространства с пос-

ледующей передачей ему физической памяти. Разница лишь в том, что физическая память для проецируемого файла — сам файл на диске, и для него не нужно выделять пространство в страничном файле. При создании объекта «проекция файла» система не резервирует регион адресного пространства и не увязывает его с физической памятью из файла (как это сделать, я расскажу в следующем разделе). Но, как только дело дойдет до отображения физической памяти на адресное пространство процесса, системе понадобится точно знать атрибут защиты, присваиваемый страницам физической памяти. Поэтому в *fdwProtect* надо указать желательные атрибуты защиты. Обычно используется один из перечисленных в таблице 17-5.

Кроме рассмотренных выше атрибутов защиты страницы, существует еще и четыре атрибута раздела; их можно ввести в параметр *fdwProtect* функции *CreateFileMapping* побитовой операцией OR. Раздел (section) — всего лишь еще одно название проекции памяти, отображаемое программой Process Explorer от Sysinternals (см. http://www.microsoft.com/technet/sysinternals/Security/ProcessExplorer.mspx).

Первый из этих атрибутов, SEC_NOCACHE, сообщает системе, что никакие страницы файла, проецируемого в память, кэшировать не надо. В результате при записи данных в файл система будет обновлять данные на диске чаще обычного. Этот флаг, как и атрибут защиты PAGE_NOCACHE, предназначен для разработчиков драйверов устройств и обычно в приложениях не используется.

Табл. 17-5. Атрибуты защиты страниц

Атрибут защиты	Описание
PAGE_READONLY	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла. При этом вы должны были передать в *CreateFile* флаг GENERIC_READ
PAGE_READWRITE	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. При этом вы должны были передать в *CreateFile* комбинацию флагов GENERIC_READ | GENERIC_WRITE
PAGE_WRITECOPY	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. Запись приведет к созданию закрытой копии страницы. При этом вы должны были передать в *CreateFile* либо GENERIC_READ, либо GENERIC_READ | GENERIC_WRITE
PAGE_EXECUTE_READ	Отобразив объект «проекция файла» на адресное пространство, можно считывать данные файла, а также запускать файл. При этом вы должны были передать в *CreateFile* комбинацию флагов GENERIC_READ и GENERIC_EXECUTE

Табл. 17-5. *(окончание)*

Атрибут защиты	Описание
PAGE_EXECUTE_READWRITE	Отобразив объект «проекция файла» па адресное пространство, можно считывать данные из файла, записывать их, а также запускать файл. При этом мы должны были передать в *CreateFile* комбинацию флагов GENERIC_READ, GENERIC_WRITE и GENERIC_EXECUTE

Второй атрибут, SEC_IMAGE, указывает системе, что данный файл является переносимым исполняемым файлом (portable executable, PE). Отображая его на адресное пространство процесса, система просматривает содержимое файла, чтобы определить, какие атрибуты защиты следует присвоить различным страницам проецируемого образа (mapped image). Например, раздел кода PE-файла (*.text*) обычно проецируется с атрибутом PAGE_EXECUTE_READ, тогда как раздел данных этого же файла (*.data*) — с атрибутом PAGE_READWRITE. Атрибут SEC_IMAGE заставляет систему спроецировать образ файла и автоматически подобрать подходящие атрибуты защиты страниц.

Следующие два атрибута (SEC_RESERVE и SEC_COMMIT) исключают друг друга и неприменимы для проецирования в память файла данных. Эти флаги мы рассмотрим ближе к концу главы. *CreateFileMapping* их игнорирует.

Последний атрибут, SEC_LARGE_PAGES, приказывает Windows использовать больших страницы для проецирования в память файлов образов. Этот атрибут применим только к файлам образов в формате PE, использовать его для отображения в память ваших файлов с данными не удастся. Как сказано выше в описании функции VirtualAlloc, для использования больших страниц необходимо соблюдать следующие условия:

- при передаче физической памяти необходимо комбинировать (с помощью операции OR) флаг SEC_COMMIT с параметром *fdwAllocationType* во время вызова *CreateFileMapping*;
- размер отображаемого файла должен быть больше значения, возвращаемого функцией *GetLargePageMinimum* (см. описание параметров *dwMaximumSizeHigh* и *dwMaximumSizeLow* ниже);
- переданная память должна быть помечена атрибутом защиты PAGE_READWRITE;
- у пользователя, вызывающего *CreateFileMapping*, должно быть право Lock Pages in Memory.

Следующие два параметра этой функции (*dwMaximumSizeHigh* и *dwMaximumSizeLow*) самые важные. Основное назначение *CreateFileMapping* — гарантировать, что объекту «проекция файла» доступен нужный объем физической памяти. Через эти параметры мы сообщаем системе максимальный размер файла в байтах. Так как Windows позволяет работать с файлами, размеры

которых выражаются 64-разрядными числами, в параметре *dwMaximumSizeHigh* указываются старшие 32 бита, а в *dwMaximumSizeLow* — младшие 32 бита этого значения. Для файлов размером менее 4 Гб *dwMaximurnSizeHigh* всегда равен 0. Наличие 64-разрядного значения подразумевает, что Windows способна обрабатывать файлы длиной до 16 экзабайтов.

Для создания объекта «проекция файла» таким, чтобы он отражал текущий размер файла, передайте в обоих параметрах нули. Так же следует поступить, если вы собираетесь ограничиться считыванием или как-то обработать файл, не меняя его размер. Для дозаписи данных в файл выбирайте его размер максимальным, чтобы оставить пространство «для маневра». Если в данный момент файл на диске имеет нулевую длину, в параметрах *dwMaximurnSizeHigh* и *dwMaximumSizeLow* нельзя передавать нули. Иначе система решит, что вам нужна проекция файла с объемом памяти, равным 0. А это ошибка, и *CreateFileMapping* вернет NULL

Если вы еще следите за моими рассуждениями, то, должно быть, подумали: что-то тут не все ладно. Очень, конечно, мило, что Windows поддерживает файлы и их проекции размером вплоть до 16 экзабайтов, но как, интересно, спроецировать такой файл на адресное пространство 32-разрядного процесса, ограниченное 4 Гб, из которых и использовать-то можно только 2 Гб? На этот вопрос я отвечу в следующем разделе. (Конечно, адресное пространство 64-разрядного процесса, размер которого составляет 16 экзабайтов, позволяет работать с еще бóльшими проекциями файлов, но аналогичное ограничение существует и там.)

Чтобы досконально разобраться, как работают функции *CreateFile* и *CreateFileMapping*, предлагаю один эксперимент. Возьмите код, приведенный ниже, соберите его и запустите под отладчиком. Пошагово выполняя операторы, переключитесь в окно командного процессора и запросите содержимое каталога «C:\» командой dir. Обратите внимание на изменения, происходящие в каталоге при выполнении каждого оператора.

```
int WINAPI _tWinMain(HINSTANCE, HINSTANCE. PTSTR, int) {

    // перед выполнением этого оператора, в каталоге C:\
    // еще нет файла "MMFTest.dat" HANDLE
    hFile = CreateFile(TEXT("C:\\MMFTest.Dat"),
        GENERIC_READ | GENERIC_WRITE,
        FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL, NULL);

    // перед выполнением этого оператора файл MMFTest.dat существует,
    // но имеет нулевую длину
    HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE,
        0, 100, NULL);

    // после выполнения предыдущего оператора размер файла MMFTest.dat
    // возрастает до 100 байтов
```

```
    // очистка
    CloseHandle(hFileHap);
    CloseHandle(hFile);

    // по завершении процесса файл MMFTest.dat останется
    // на диске и будет иметь длину 100 байтов
    return(0);
}
```

Вызов *CreateFileMapping* с флагом PAGE_READWRITE заставляет систему проверять, чтобы размер соответствующего файла данных на диске был не меньше, чем указано в параметрах *dwMaximumSizeHigh* и *dwMaximumSizeLow*. Если файл окажется меньше заданного, *CreateFileMapping* увеличит его размер до указанной величины. Это делается специально, чтобы выделить физическую память перед использованием файла в качестве проецируемого в память. Если объект «проекция файла» создан с флагом PAGE_READONLY или PAGE_WRITECOPY, то размер, переданный функции *CreateFileMapping*, не должен превышать физический размер файла на диске (так как вы не сможете что-то дописать в файл).

Последний параметр функции *CreateFileMapping* — *pszName* — строка с нулевым байтом в конце; в ней указывается имя объекта «проекция файла», которое используется для доступа к данному объекту из другого процесса (пример см. в главе 3). Но обычно совместное использование проецируемого в память файла не требуется, и поэтому в данном параметре передают NULL

Система создает объект «проекция файла» и возвращает его описатель в вызвавший функцию поток. Если объект создать не удалось, возвращается нулевой описатель (NULL). И здесь еще раз обратите внимание на отличительную особенность функции *CreateFile* — при ошибке она возвращает не NULL, а идентификатор INWALID_HANDLE_VALUE (определенный как -1).

Этап 3: проецирование файловых данных на адресное пространство процесса

Когда объект «проекция файла» создан, нужно, чтобы система, зарезервировав регион адресного пространства подданные файла, передала их как физическую память, отображенную на регион. Это делает функция *MapViewOfFile*:

```
PV0ID MapViewOfFile(
    HANDLE hFileMappingObject,
    DW0RD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    SIZE_T dwNumberOfBytesToMap);
```

Параметр *hFileMappingObject* идентифицирует описатель объекта «проекция файла», возвращаемый предшествующим вызовом либо *CreateFileMapping*, либо *OpenFileMapping* (ее мы рассмотрим чуть позже). Параметр

dwDesiredAccess идентифицирует вид доступа к данным. Все правильно: придется опять указывать, как именно мы хотим обращаться к файловым данным. Можно задать одно из четырех значений, описанных в следующей таблице.

Табл. 17-6. Права доступа к файлу, спроецированному в память

Значение	Описание
FILE_MAP_WRITE	Файловые данные можно считывать и записывать; вы должны были передать функции *CreateFileMapping* атрибут PAGE_READWRITE
FILE_MAP_READ	Файловые данные можно только считывать; вы должны были вызвать *CreateFileMapping* с любым из следующих атрибутов: PAGE_READONLY, PAGE_ READWRITE или PAGE_WRITECOPY
FILE_MAP_ALL_ACCESS	То же, что и FILE_MAP_WRITE
FILE_MAP_COPY	Файловые данные можно считывать и записывать, но запись приводит к созданию закрытой копии страницы; вы должны были вызвать *CreateFileMapping* с любым из следующих атрибутов: PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY
FILE_MAP_EXECUTE	Содержимое файла может быть исполнено как код; вы должны были вызвать *CreateFileMapping* с атрибутом PAGE_EXECUTE_READWRITE или PAGE_EXECUTE_READ

Кажется странным и немного раздражает, что Windows требует бесконечно указывать все эти атрибуты защиты. Могу лишь предположить, что это сделано для того, чтобы приложение максимально полно контролировало защиту данных.

Остальные три параметра относятся к резервированию региона адресного пространства и к отображению на него физической памяти. При этом необязательно проецировать на адресное пространство весь файл сразу. Напротив, можно спроецировать лишь малую его часть, которая в таком случае называется представлением (view) — теперь-то вам, наверное, понятно, откуда произошло название функции *MapViewOfFile*.

Проецируя на адресное пространство процесса представление файла, нужно сделать две вещи. Во-первых, сообщить системе, какой байт файла данных считать в представлении первым. Для этого предназначены параметры *dwFileOffsetHigh* и *dwFileOffsetLow*. Поскольку Windows поддерживает файлы длиной до 16 экзабайтов, приходится определять смещение в файле как 64-разрядное число: старшие 32 бита передаются в параметре *dwFileOffsetHigh*, а младшие 32 бита—в параметре *dwFileOffsetLow*. Заметьте, что смещение в файле должно быть кратно гранулярности выделения памяти в данной системе. (В настоящее время во всех реализациях Windows она

составляст64 Кб.) О гранулярности выделения памяти см. раздел «Системная информация» в главе 14.

Во-вторых, от вас потребуется указать размер представления, т. е. сколько бантов файла данных должно быть спроецировано на адресное пространство. Это равносильно тому, как если бы вы задали размер региона, резервируемого в адресном пространстве. Размер указывается в параметре *dwNumberOfBytesToMap*. Если этот параметр равен 0, система попытается спроецировать представление, начиная с указанного смещения и до конца файла. В Windows функция *MapViewOfFile* ищет регион, достаточно большой для размещения запрошенного представления, не обращая внимания на размер самого объекта «проекция файла».

Если при вызове *MapViewOfFile* указан флаг FILE_MAP_COPY, система передаст физическую память из страничного файла. Размер передаваемого пространства определяется параметром *dwNumberOfBytesToMap*. Пока вы лишь считываете данные из представления файла, страницы, переданные из страничного файла, не используются. Но стоит какому-нибудь потоку в вашем процессе совершить попытку записи по адресу, попадающему в границы представления файла, как система тут же берет из страничного файла одну из переданных страниц, копирует на нее исходные данные и проецирует ее на адресное пространство процесса. Так что с этого момента потоки вашего процесса начинают обращаться к локальной копии данных и теряют доступ к исходным данным.

Создав копию исходной страницы, система меняет ее атрибут защиты с PAGE_WRITECOPY на PAGE_READWRITE. Рассмотрим пример:

```
// открываем файл, который мы собираемся спроецировать
HANDLE hFile = CreateFile(pszFileName, GENERIC_READ | GENERIC,
    WRITE, 0, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

// создаем для файла объект "проекция файла"
HANDLE hFileMapping = CreateFileMapping(hFile, NULL, PAGE_WRITECOPY,
    0, 0, NULL);

// Проецируем представление файла с атрибутом "копирование при записи";
// система передаст столько физической памяти из страничного файла,
// сколько нужно для размещения всего файла. Первоначально все страницы
// в представлении получат атрибут PAGE_WRITECOPY.
PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_COPY,
    0, 0, 0);

// считываем байт из представления файла
BYTE bSomeByte = pbFile[0];
// при чтении система не трогает страницы, переданные из страничного файла;
// страница сохраняет свой атрибут PAGE_WRITEC0PY
```

```
// записываем байт в представление файла
pbFile[0] = 0;
// При первой записи система берет страницу, переданную из страничного файла,
// копирует исходное содержимое страницы, расположенной по запрашиваемому адресу
// в памяти, и проецирует новую страницу (копию) на адресное пространство процес-
    са.
// Новая страница получает атрибут PAGE_READWRITE.

// записываем еще один байт в представление файла
pbFile[1] = 0;
// поскольку теперь байт располагается на странице с атрибутом PAGE_ READWRITE,
// система просто записывает его на эту страницу (она связана со страничным фай-
    лом)

// закончив работу с представлением проецируемого файла, прекращаем проецирование;
// функция UnmapViewOfFile обсуждается в следующем разделе
UnmapViewOfFile(pbFile);
// вся физическая память, взятая из страничного файла, возвращается системе;
// все, что было записано на эти страницы, теряется

// "уходя, гасите свет"
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

Примечание. Если ваше приложение работает на компьютере с архитектурой NUMA, можно повысить быстродействие, если передавать потоку физическую память, смонтированную на той же плате, что и исполняющий его процессор. Когда поток создает представление для спроецированного файла, Windows по умолчанию так и делает. Если же известно, что поток может подключиться к другому процессору, можно изменить заданное по умолчанию поведение, вызвав функцию *CreateFileMappingNuma* и явно указав плату NUMA-компьютера (в последнем параметре *dwPreferredNumaNode*), память которой следует передавать потоку:

```
HANDLE CreateFileMappingNuma(
    HANDLE hFile,
    PSECURITY_ATTRIBUTES psa,
    DWORD fdwProtect,
    DWORD dwMaximumSizeHigh,
    DWORD dwMaxirounSizeLow,
    PCTSTR pszName,
    DWORD dwPreferredNumaNode
);
```

Теперь при вызове *MapViewOfFile* поток получит память из банка, заданного при вызове *CreateFileMappingNuma*. Windows также поддерживает

функцию *MapViewOfFileExNuma*, которая позволяет переопределить банк памяти, заданный при вызове функции. Это делается с помощью се параметра *dwPreferredNumaNode*;

```
PVOID MapViewOfFileExNuma(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    SIZE_T dwNumberOfBytesToMap,
    LPVOID lpBaseAddress,
    DWORD dwPreferredNumaNode
);
```

Подробнее о Windows-функциях для управления памятью на NUMA-компьютерах см. в разделах «Управление памятью на компьютерах с архитектурой NUMA» и «Резервирование региона в адресном пространстве» в главах 14 и 15.

Этап 4: отключение файла данных от адресного пространства процесса

Когда необходимость в данных файла (спроецированного на регион адресного пространства процесса) отпадет, освободите регион вызовом:

```
BOOL UnmapViewOfFile(PVOID pvBaseAddress);
```

Ее единственный параметр, *pvBaseAddress*, указывает базовый адрес возвращаемого системе региона. Он должен совпадать со значением, полученным после вызова *MapViewOfFile*. Вы обязаны вызывать функцию *UnmapViewOfFile*. Если вы не сделаете этого, регион не освободится до завершения вашего процесса. И еще: повторный вызов *MapViewOfFile* приводит к резервированию нового региона в пределах адресного пространства процесса, но ранее выделенные регионы *не освобождаются*.

Для повышения производительности при работе с представлением файла система буферизует страницы данных в файле и не обновляет немедленно дисковый образ файла. При необходимости можно заставить систему записать измененные данные (все или частично) в дисковый образ файла, вызвав функцию *FlushViewOfFile*:

```
BOOL FlushViewOfFile(
    PVOID pvAddress,
    SIZE_T dwNumberOfBytesToFlush);
```

Ее первый параметр принимает адрес байта, который содержится в границах представления файла, проецируемого в память. Переданный адрес округляется до значения, кратного размеру страниц. Второй параметр определяет количество байтов, которые надо записать в дисковый образ файла. Если *FlushViewOfFile* вызывается в отсутствие измененных данных, она просто возвращает управление.

В случае проецируемых файлов, физическая память которых расположена на сетевом диске, *MushViewOfFile* гарантирует, что файловые данные будут перекачаны с рабочей станции. Но она не гарантирует, что сервер, обеспечивающий доступ к этому файлу, запишет данные на удаленный диск, так как он может просто кэшировать их. Для подстраховки при создании объекта «проекция файла» и последующем проецировании его представления используйте флаг FILE_FLAG_WRITE_THROUGH. При открытии файла с этим флагом функция *FlushViewOfFile* вернет управление только после сохранения на диске сервера всех файловых данных.

У функции *UnmapViewOfFile* есть одна особенность. Если первоначально представление было спроецировано с флагом FILE_MAP_COPY, любые изменения, внесенные вами в файловые данные, на самом деле производятся над копией этих данных, хранящихся в страничном файле. Вызванной в этом случае функции *UnmapViewOfFile* нечего обновлять в дисковом файле, и она просто инициирует возврат системе страниц физической памяти, выделенных из страничного файла. Все изменения в данных на этих страницах теряются.

Поэтому о сохранении измененных данных придется заботиться самостоятельно. Например, для уже спроецированного файла можно создать еще один объект «проекция файла» с атрибутом PAGE_READWRITE и спроецировать его представление на адресное пространство процесса с флагом FILE_MAP_WRITE. Затем просмотреть первое представление, отыскивая страницы с атрибутом PAGE_READWRITE. Найдя страницу с таким атрибутом, вы анализируете ее содержимое и решаете: записывать ее или нет. Если обновлять файл не нужно, вы продолжаете просмотр страниц. А для сохранения страницы с измененными данными достаточно вызвать *MoveMemory* и скопировать страницу из первого представления файла во второе. Поскольку второе представление создано с атрибутом PAGE_READWRITE, функция *MoveMemory* обновит содержимое дискового файла. Так что этот метод вполне пригоден для анализа изменений и сохранения их в файле.

Этапы 5 и 6: закрытие объектов «проекция файла» и «файл»

Закончив работу с любым открытым вами объектом ядра, вы должны его закрыть, иначе в процессе начнется утечка ресурсов. Конечно, по завершении процесса система автоматически закроет объекты, оставленные открытыми. Но, если процесс поработает еще какое-то время, может накопиться слишком много незакрытых описателей. Поэтому старайтесь придерживаться правил хорошего тона и пишите код так, чтобы открытые объекты всегда закрывались, как только они станут не нужны. Для закрытия объектов «проекция файла» и «файл» дважды вызовите функцию *CloseHandle*.

Рассмотрим это подробнее на фрагменте псевдокода:

```
HANDLE hFile = CreateFile(…);
HANDLE hFileMapping = CreateFileMapping(hFile, …);
PVOID pvFile = MapViewOfFile(hFileMapping, …);
```

```
// работаем с файлом, спроецированным в память

UnmapViewOfFile(pvFile);
CloseHandle(hFileMapping);
CloseHandle(hFile);
```

Этот фрагмент иллюстрирует стандартный метод управления проецируемыми файлами. Но он не отражает того факта, что при вызове *MapViewOfFile* система увеличивает счетчики числа пользователей объектов «файл» и «проекция файла». Этот побочный эффект весьма важен, так как позволяет переписать показанный выше фрагмент кода следующим образом:

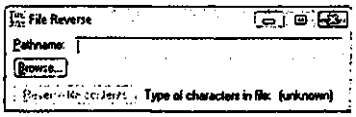
```
HANDLE hFile = CreateFile(…);
HANDLE hFileMapping = CreateFileMapping(hFile, …);
CloseHandle(hFile);
PVOID pvFile = MapViewOfFile(hFileMapping, …);
CloseHandle(hFileMapping);

// работаем с файлом, спроецированным в память

UnmapViewOfFile(pvFile);
```

При операциях с проецируемыми файлами обычно открывают файл, создают объект «проекция файла» и с его помощью проецируют представление файловых данных на адресное пространство процесса. Поскольку система увеличивает внутренние счетчики объектов «файл» и «проекция файла», их можно закрыть в начале кода, тем самым исключив возможную утечку ресурсов.

Если вы будете создавать из одного файла несколько объектов «проекция файла» или проецировать несколько представлений этого объекта, применить функцию *CloseHandle* в начале кода не удастся — описатели еще понадобятся вам для дополнительных вызовов *CreateFileMapping* и *MapViewOfFile*.

Программа-пример FileRev

Эта программа (17 FileRev.exe) демонстрирует, как с помощью механизма проецирования записать в обратном порядке содержимое текстового ANSI-или Unicode-файла. Файлы исходного кода и ресурсов этой программы находятся в каталоге 17-FileRev внутри архива, доступного на веб-сайте поддержки этой книги. После запуска FileRev на экране появляется диалоговое окно, показанное ниже.



Выбрав имя файла и щелкнув кнопку Reverse File Contents, вы активизируете функцию, которая меняет порядок символов в файле на обратный.

Программа корректно работает только с текстовыми файлами. В какой кодировке создан текстовый файл (ANSI или Unicode), FileRev определяет вызовом *IsTextUnicode* (см. главу 2).

После щелчка кнопки Reverse File Contents программа создает копию файла с именем FileRev.dat. Делается это для того, чтобы не испортить исходный файл, изменив порядок следования байтов на обратный. Далее программа вызывает функцию *FileReverse* — она меняет порядок байтов на обратный и после этого вызывает CreateFile, открывая FileRev.dat для чтения и записи.

Как я уже говорил, простейший способ «перевернуть» содержимое файла — вызвать функцию *_strrev* из библиотеки C. Но для этого последний символ в строке должен быть нулевой. И поскольку текстовые файлы не заканчиваются нулевым символом, программа FileRev подписывает его в конец файла. Для этого сначала вызывается функция *GetFileSize*.

```
dwFileSize = GetFileSize(hFile, NULL);
```

Теперь, вооружившись знанием длины файла, можно создать объект «проекция файла», вызвав *CreateFileMapping*. При этом размер объекта равен *dwFileSize* плюс размер «широкого» символа, чтобы учесть дополнительный нулевой символ в конце файла. Создав объект «проекция файла», программа проецирует на свое адресное пространство представление этого объекта. Переменная *pvFile* содержит значение, возвращенное функцией *MapViewOfFile*, и указывает на первый байт текстового файла.

Следующий шаг — запись нулевого символа в конец файла и реверсия строки:

```
PSTR pchANSI = (PSTR) pvFile;
pchANSI[dwFileSize / sizeof(CHAR)] = 0;
```

В текстовом файле каждая строка завершается символами возврата каретки ('\r') и перевода строки ('\n'). К сожалению, после вызова функции *_strrev* эти символы тоже меняются местами. Поэтому для загрузки преобразованного файла в текстовый редактор придется заменить все пары «\n\r» на исходные «\r\n». В программе этим занимается следующий цикл:

```
while (pchANSI != NULL) {
// "переворачиваем" содержимое файла
   *pchANSI++ = '\r';                    // заменяем '\n' на '\r'
   *pchANSI++ = '\n';                    // заменяем '\r' на '\n'
   pchANSI = strstr(pchANSI, "\n\r");    // ищем следующее вхождение
}
```

Закончив обработку файла, программа прекращает отображение на адресное пространство представления объекта «проекция файла» и закрывает описатели всех объектов ядра. Кроме того, программа должна удалить нулевой символ, добавленный в конец файла (функция *_strrev* не меняет позицию этого символа). Если бы программа не убрала нулевой символ, то полу-

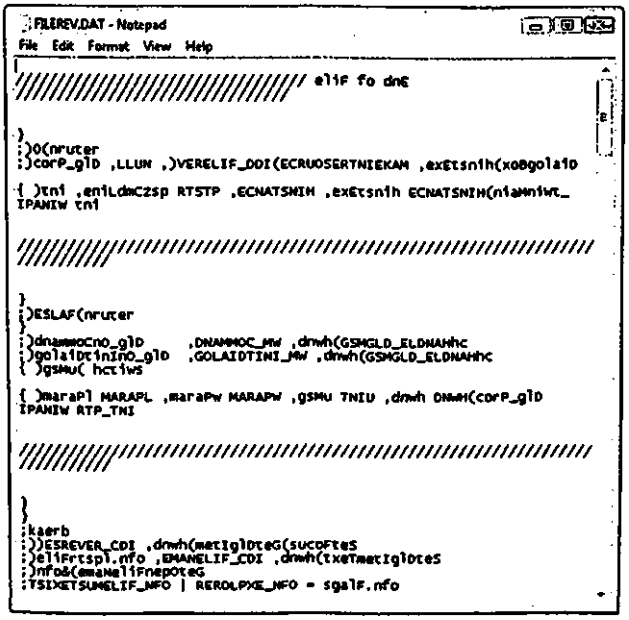
ченный файл оказался бы на 1 символ длиннее, и тогда повторный запуск программы FileRev нс позволил бы вернуть этот файл в исходное состояние. Чтобы удалить концевой нулевой символ, надо спуститься на уровень ниже и воспользоваться функциями, предназначенными для работы непосредственно с файлами на диске.

Прежде всего установите указатель файла в требуемую позицию (в данном случае — в конец файла) и вызовите функцию *SetEndOfFile*:

```
SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
```

> **Примечание.** Функцию *SetEndOfFile* нужно вызывать после отмены проецирования представления и закрытия объекта «проекция файла», иначе она вернет FALSE, а функция *GetLastError* — ERROR_USER_MAPPED_FILE. Данная ошибка означает, что операция перемещения указателя в конец файла невозможна, пока этот файл связан с объектом «проекция файла».

Последнее, что делает FileRev, — запускает экземпляр Notepad, чтобы вы могли увидеть преобразованный файл. Вот как выглядит результат работы программы FileRev применительно к собственному файлу FileRev.cpp.



FileRev.cpp

```
/******************************************************************************
Module:  FileRev.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..\CommonFiles\CmnHdr.h"     /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <commdlg.h>
#include <string.h>        // для доступа к _strrev
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////

#define FILENAME  TEXT("FileRev.dat")

//////////////////////////////////////////////////////////////////////////////

BOOL FileReverse(PCTSTR pszPathname, PBOOL pbIsTextUnicode) {

   *pbIsTextUnicode = FALSE;  // предполагаем, что текст в Unicode

   // открываем файл для чтения и записи
   HANDLE hFile = CreateFile(pszPathname, GENERIC_WRITE | GENERIC_READ, 0,
      NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

   if (hFile == INVALID_HANDLE_VALUE) {
      chMB("File could not be opened.");
      return(FALSE);
   }

   // получаем размер файла (я предполагаю, что спроецировать можно весь файл)
   DWORD dwFileSize = GetFileSize(hFile, NULL);

   // создаем объект «проекция файла». Он на 1 символ больше, чем сам
   // файл, чтобы можно было дописать нулевой символ для корректного
   // завершения строки. Поскольку пока еще неизвестно, содержит файл
   // ANSI- или Unicode-символы, я предлагаю худшее и добавляю
   // размер WCHAR вместо CHAR.
   HANDLE hFileMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE,
      0, dwFileSize + sizeof(WCHAR), NULL);

   if (hFileMap == NULL) {
      chMB("File map could not be opened.");
```

```
      CloseHandle(hFile);
      return(FALSE);
   }

   // получаем адрес, по которому проецируется в память первый байт файла
   PVOID pvFile = MapViewOfFile(hFileMap, FILE_MAP_WRITE, 0, 0, 0);

   if (pvFile == NULL) {
      chMB("Could not map view of file.");
      CloseHandle(hFileMap);
      CloseHandle(hFile);
      return(FALSE);
   }

   // что содержит буфер: ANSI- или Unicode-символы?
   int iUnicodeTestFlags = -1;   // Try all tests
   *pbIsTextUnicode = IsTextUnicode(pvFile, dwFileSize, &iUnicodeTestFlags);

   if (!*pbIsTextUnicode) {
      // при дальнейших операциях с файлом явно используется ANSI-функция,
      // так как мы имеет дело с ANSI-файлом

      // записываем в самый конец файла нулевой символ.
      PSTR pchANSI = (PSTR) pvFile;
      pchANSI[dwFileSize / sizeof(CHAR)] = 0;

      // «переворачиваем» содержимое файла.
      _strrev(pchANSI);

      // преобразуем все комбинации "\n\r" обратно в to "\r\n", чтобы
      // сохранить нормальную последовательность кодов завершения строки
      // в текстовый файл
      pchANSI = strstr(pchANSI, "\n\r"); // Find first "\r\n".

      while (pchANSI != NULL) {
         // вхождение найдено…
         *pchANSI++ = '\r';   // заменяем '\n' на '\r'.
         *pchANSI++ = '\n';   // заменяем '\r' на '\n'.
         pchANSI = strstr(pchANSI, "\n\r"); // ищем следующее вхождение
      }

   } else {
      // при дальнейших операциях с файлом явно используем
      // Unicode-функции, так как мы имеем дело с Unicode-файлом
```

```
      // записываем в самый конец файла нулевой символ
      PWSTR pchUnicode = (PWSTR) pvFile;
      pchUnicode[dwFileSize / sizeof(WCHAR)] = 0;

      if ((iUnicodeTestFlags & IS_TEXT_UNICODE_SIGNATURE) != 0) {
         // если первый символ – Unicode-маркер порядка байтов (0xFEFF),
         // то оставим этот символ в начале файла.
         pchUnicode++;
      }

      // «переворачиваем» содержимое файла
      _wcsrev(pchUnicode);

      // преобразуем все комбинации "\n\r" обратно в "\r\n", чтобы
      // сохранить нормальную последовательность кодов завершения
      // строки в текстовом файле
      pchUnicode = wcsstr(pchUnicode, L"\n\r"); // ищем первое вхождение '\n\r'

      while (pchUnicode != NULL) {
         // вхождение найдено…
         *pchUnicode++ = L'\r';   // заменяем '\n' на '\r'.
         *pchUnicode++ = L'\n';   // заменяем '\r' на '\n'.
         pchUnicode = wcsstr(pchUnicode, L"\n\r"); // ищем следующее вхождение
      }
   }

   // очищаем все перед завершением
   UnmapViewOfFile(pvFile);
   CloseHandle(hFileMap);

   // удаляем добавленный ранее концевой нулевой байт
   SetFilePointer(hFile, dwFileSize, NULL, FILE_BEGIN);
   SetEndOfFile(hFile);
   CloseHandle(hFile);

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_FILEREV);
```

```
   // инициализируем диалоговое окно
   EnableWindow(GetDlgItem(hWnd, IDC_REVERSE), FALSE);
   return(TRUE);
}

///////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   TCHAR szPathname[MAX_PATH];

   switch (id) {
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;

      case IDC_FILENAME:
         EnableWindow(GetDlgItem(hWnd, IDC_REVERSE),
            Edit_GetTextLength(hWndCtl) > 0);
         break;

      case IDC_REVERSE:
         GetDlgItemText(hWnd, IDC_FILENAME, szPathname, _countof(szPathname));

         // делаем копию исходного файла, чтобы случайно не повредить его
         if (!CopyFile(szPathname, FILENAME, FALSE)) {
            chMB("New file could not be created.");
            break;
         }

         BOOL bIsTextUnicode;
         if (FileReverse(FILENAME, &bIsTextUnicode)) {
            SetDlgItemText(hWnd, IDC_TEXTTYPE,
               bIsTextUnicode ? TEXT("Unicode") : TEXT("ANSI"));

            // запускаем Notepad, чтобы увидеть плоды своих трудов
            STARTUPINFO si = { sizeof(si) };
            PROCESS_INFORMATION pi;
            TCHAR sz[] = TEXT("Notepad ") FILENAME;
            if (CreateProcess(NULL, sz,
               NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi)) {

               CloseHandle(pi.hThread);
               CloseHandle(pi.hProcess);
            }
```

```
         }
         break;

      case IDC_FILESELECT:
         OPENFILENAME ofn = { OPENFILENAME_SIZE_VERSION_400 };
         ofn.hwndOwner = hWnd;
         ofn.lpstrFile = szPathname;
         ofn.lpstrFile[0] = 0;
         ofn.nMaxFile = _countof(szPathname);
         ofn.lpstrTitle = TEXT("Select file for reversing");
         ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST;
         GetOpenFileName(&ofn);
         SetDlgItemText(hWnd, IDC_FILENAME, ofn.lpstrFile);
         SetFocus(GetDlgItem(hWnd, IDC_REVERSE));
         break;
   }
}

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG,  Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,     Dlg_OnCommand);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR, int) {

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_FILEREV), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

Обработка больших файлов

Я обещал рассказать, как спроецировать на небольшое адресное пространство файл длиной 16 экзабайтов. Так вот, этого сделать нельзя. Вам придется проецировать не весь файл, а его представление, содержащее лишь некую часть данных. Вы начнете с того, что спроецируете представление са́мого на-

чала файла. Закончив обработку данных в этом представлении, вы отключите его и спроецируете представление следующей части файла — и так до тех пор, пока не будет обработан весь файл. Конечно, это делает работу с большими файлами, проецируемыми в намять, не слишком удобной, но утешимся тем, что длина большинства файлов достаточно мала.

Рассмотрим сказанное на примере файла размером 8 Гб. Ниже приведен текст подпрограммы, позволяющей в несколько этапов подсчитывать, сколько раз встречается нулевой байт в том или ином двоичном файле данных.

```
__int64 CountOs(void) {

    // начальные границы представлений всегда начинаются по адресам,
    // кратным гранулярности выделения памяти
    SYSTEM_INFO sinf;
    GetSystemInfo(&sinf);

    // открываем файл данных
    HAMDLE hFile = CreateFile(TEXT("C:\\HugeFile.Big"), GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_FLAG_SEQUENTIAL_SCAN, NULL);

    // создаем объект "проекция файла"
    HANDLE hFileMapping = CreateFileMapping(hFile, NULL,
        PAGE_READONLY, 0, 0, NULL);

    DWORD dwFileSizeHigh;
    __int64 qwFileSize = GetFileSize(hFile, &dwFileSizeHigh);
    qwFileSize += (((_int64) dwFileSizeHigh) << 32);

    // доступ к описателю объекта "файл" нам больше не нужен
    CloseHandle(hFile);

    __int64 qwFileOffset = 0, qwNum0f0s = 0;

    while (qwFileSize > 0) {

        // определяем, сколько байтов надо спроецировать
        DW0RD dwBytesInBlock = sinf.dwAllocationGranularity;
        if (qwFileSize < sinf.dwAllocationGranularity)
            dwBytesInBlock = (DWORD) qwFileSize;

        PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_HAP_READ,
            (DWORD) (qwFileOffset >> 32),        // начальный байт
            (DW0RD) (qwFileOffset & 0xFFFFFFFF), // в файле
            dwBytesInBlock);                     // # число проецируемых байтов
```

```
        // подсчитываем количество нулевых байтов в этом блоке
        for (DWORD dwByte = 0; dwByte < dwBytesInBlock; dwByte++) {
            lf (pbFile[dwByte] == 0)
            qwNumOfOs++;
        }

        // прекращаем проецирование представления, чтобы в адресном
        // пространстве не образовалось несколько представлений одного файла
        UnmapViewOfFile(pbFile);

        // переходим к следующей группе байтов в файле
        qwFileOffset += dwBytesInBlock;
        qwFileSize -= dwBytesInBlock;
    }

    CloseHandle(hFileMapping);
    return(qwNumOfOs);
}
```

Этот алгоритм проецирует представления по 64 Кб (в соответствии с гранулярностью выделения памяти) или менее. Кроме того, функция *MapViewOfFile* требует, чтобы передаваемое ей смещение в файле тоже было кратно гранулярности выделения памяти. Подпрограмма проецирует на адресное пространство сначала одно представление, подсчитывает в нем количество нулей, затем переходит к другому представлению, и все повторяется. Спроецировав и просмотрев все 64-килобайтовые блоки, подпрограмма закрывает объект «проекция файла».

Проецируемые файлы и когерентность

Система позволяет проецировать сразу несколько представлений одних и тех же файловых данных. Например, можно спроецировать в одно представление первые 10 Кб файла, а затем — первые 4 Кб того же файла в другое представление. Пока вы проецируете один и тот же объект, система гарантирует *когерентность* (согласованность) отображаемых данных. Скажем, если программа изменяет содержимое файла в одном представлении, это приводит к обновлению данных и в другом. Так происходит потому, что система, несмотря на многократную проекцию страницы на виртуальное адресное пространство процесса, хранит данные на единственной странице оперативной памяти. Поэтому, если представления одного и того же файла данных создаются сразу несколькими процессами, данные попрежнему сохраняют когерентность — ведь они сопоставлены только с одним экземпляром каждой страницы в оперативной памяти. Все это равносильно тому, как если бы страницы оперативной памяти были спроецированы на адресные пространства нескольких процессов одновременно.

> **Примечание.** Windows позволяет создавать несколько объектов «проекция файла», связанных с одним и тем же файлом данных. Но тогда у вас *не будет* гарантий, что содержимое представлений этих объектов когерентно. Такую гарантию Windows даст только для нескольких представлений одного объекта «проекция файла».

Кстати, функция *CreateFile* позволяет вашему процессу открывать файл, проецируемый в память другим процессом. После этого ваш процесс сможет считывать или записывать данные в файл (с помощью функций *ReadFile* или *WriteFile*). Разумеется, при вызовах упомянутых функций ваш процесс будет считывать или записывать данные не в файл, а в некий буфер памяти, который должен быть создан именно этим процессом; буфер не имеет никакого отношения к участку памяти, используемому для проецирования данного файла. Но надо учитывать, что, когда два приложения открывают один файл, могут возникнуть проблемы. Дело в том, что один процесс может вызвать *ReadFile*, считать фрагмент файла, модифицировать данные и записать их обратно в файл с помощью *WriteFile*, а объект «проекция файла», принадлежащий второму процессу, ничего об этом не узнает. Поэтому, вызывая для проецируемого файла функцию *CreateFile*, всегда указывайте нуль в параметре *dwShareMode*. Тем самым вы сообщите системе, что вам нужен монопольный доступ к файлу и никакой посторонний процесс не должен его открывать.

Файлы с доступом «только для чтения» не вызывают проблем с когерентностью — значит, это лучшие кандидаты на отображение в память. Ни в коем случае не используйте механизм проецирования для доступа к записываемым файлам, размещенным на сетевых дисках, так как система не сможет гарантировать когерентность представлений данных. Если один компьютер обновит содержимое файла, то другой, у которого исходные данные содержатся в памяти, не узнает об изменении информации.

Базовый адрес файла, проецируемого в память

Помните, как вы с помощью функции *VirtualAlloc* указывали базовый адрес региона, резервируемого в адресном пространстве? Примерно так же можно указать системе спроецировать файл по определенному адресу — только вместо функции *MapViewOfFile* нужна *MapViewOfFileEx*:

```
PVOID MapViewOfFileEx(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    SIZE_T dwNumberOfBytesToMap,
    PVOID pvBaseAddress);
```

Все параметры и возвращаемое этой функцией значение идентичны применяемым в *MapViewOfFile*, кроме последнего параметра — *pvBaseAddress*. В нем можно задать начальный адрес файла, проецируемого в память. Как и в случае *VirtualAlloc*, базовый адрес должен быть кратным гранулярности выделения памяти в системе (обычно 64 Кб), иначе *MapViewOfFileEx* вернет NULL, сообщив тем самым об ошибке. Если вы укажете базовый адрес, не кратный гранулярности выделения памяти, то *MapViewOfFileEx* в Windows 2000 завершится с ошибкой, и *GetLastError* вернет код 1132 (ERROR_MAPPED_ALIGNMENT).

Если система не в состоянии спроецировать файл по этому адресу (чаще всего из-за того, что файл слишком велик и мог бы перекрыть другие регионы зарезервированного адресного пространства), функция также возвращает NULL В этом случае она не пытается подобрать диапазон адресов, подходящий для данного файла. Но если вы укажете NULL в параметре *pvBaseAddress*, она поведет себя идентично *MapViewOfFile*.

MapViewOfFileEx удобна, когда механизм проецирования файлов в память применяется для совместного доступа нескольких процессов к одним данным. Поясню. Допустим, нужно спроецировать файл в память по определенному адресу; при этом два или более приложений совместно используют одну группу структур данных, содержащих указатели на другие структуры данных. Отличный тому пример — связанный список. Каждый узел, или элемент, такого списка хранит адрес другого узла списка. Для просмотра списка надо узнать адрес первого узла, а затем сделать ссылку на то его поле, где содержится адрес следующего узла. Но при использовании файлов, проецируемых в память, это весьма проблематично.

Если один процесс подготовил в проецируемом файле связанный список, а затем разделил его с другим процессом, не исключено, что второй процесс спроецирует этот файл в своем адресном пространстве на совершенно иной регион. А дальше будет вот что. Попытавшись просмотреть связанный список, второй процесс проверит первый узел списка, прочитает адрес следующего узла и, сделав на него ссылку, получит совсем не то, что ему было нужно, — адрес следующего элемента в первом узле некорректен для второго процесса.

У этой проблемы два решения. Во-первых, второй процесс, проецируя файл со связанным списком на свое адресное пространство, может вызвать *MapViewOfFileEx* вместо *MapViewOfFile*. Для этого второй процесс должен знать адрес, по которому файл спроецирован на адресное пространство первого процесса на момент создания списка. Если оба приложения разработаны с учетом взаимодействия друг с другом (а так чаще всего и делают), нужный адрес может быть просто заложен в код этих программ или же один процесс как-то уведомляет другой (скажем, посылкой сообщения в окно).

А можно и так. Процесс, создающий связанный список, должен записывать в каждый узел смещение следующего узла в пределах адресного пространства. Тогда программа, чтобы получить доступ к каждому узлу, будет суммировать

это смещение с базовым адресом проецируемого файла. Несмотря на простоту, этот способ не лучший: дополнительные операции замедлят работу программы и увеличат объем ее кода (гак как компилятор для выполнения всех вычислений, естественно, сгенерирует дополнительный код). Кроме того, при этом способе вероятность ошибок значительно выше. Тем не менее он имеет право на существование, и поэтому компиляторы Майкрософт поддерживают указатели со смещением относительно базового значения (based-pointers), для чего предусмотрено ключевое слово *__based*.

> **Примечание.** При вызове *MapViewOfFileEx* следует указывать адрес в границах пользовательского раздела адресного пространства процесса, иначе функция вернет NULL

Особенности проецирования файлов

Windows для доступа к файловым данным в адресном пространстве требует вызова *MapViewOfFile*. При обращении к этой функции система резервирует для проецируемого файла закрытый регион адресного пространства, и никакой другой процесс не получает к нему доступ автоматически. Чтобы посторонний процесс мог обратиться к данным того же объекта «проекция файла», его поток тоже должен вызвать *MapViewOfFile*, и система отведет регион для представления объекта в адресном пространстве второго процесса. Здесь важно заметить, что адреса, которые *MapViewOfFile* вернет первому и второму процессу, скорее всего, будут разными. Это верно, даже когда процессы работают с представлениями одного и того спроецированного файла.

Взгляните на текст программы, проецирующей два представления единственного объекта «проекция файла».

```
int WINAPI _tWinMain (HINSTANCE, HINSTANCE, PTSTR, int) {
    // открываем существующий файл; он должен быть больше 64 Кб
    HANDLE hFile = CreateFile(pszCmdLine, GENERIC_READ | GENERIC_WRITE,
        0, NULL, OPEN_EXIST_NG, FILE_ATTRIBUTE_NORMAL, NULL);

    // создаем объект "проекция файла", связанный с файлом данных
    HANDLE hFileMapping = CreateFileMapping(hFile, NULL,
        PAGE_READWRITE, 0, 0, NULL);

    // проецируем представление всего файла на наше адресное пространство
    PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping,
        FILE_MAP_WRITE, 0, 0, 0);

    // проецируем второе представление файла, начиная со смещения 64 Кб
    PBYTE pbFile2 = (PBYTE) MapViewOfFile(hFileMapping,
        FILE_MAP_WRTTE, 0, 65536, 0);
```

```
        // если адреса перекрываются, оба представления проецируются на один
        // регион, и мы работаем в Windows 98
    int iDifference = int(pbFile2 - pbFile);
    TCHAR szMsg[100];
    StringCchPrintf(szMsg, _countof(szMsg),
        TEXT("Pointers difference = %d KB"), iDifference / 1024);
    MessageBox(NULL, szNsg, NULL, MB_OK);

    UnmapViewOfFile(pbFile2);
    UnmapViewOfFile(pbFile);
    CloseHandle(hFileHapplng);
    CloseHandle(hFile);

    return(0);
}
```

Два вызова функции *MapViewOfFile* (как в показанном выше коде) приведут к тому, что будут зарезервированы два региона адресного пространства. Объем первого будет равен размеру объекта «проекция файла», объем второго — размеру объекта минус 64 Кб. Хотя регионы — разные, система гарантирует когерентность данных, так как оба представления созданы на основе одного объекта «проекция файла».

Совместный доступ процессов к данным через механизм проецирования

В Windows всегда было много механизмов, позволяющих приложениям легко и быстро разделять какие-либо данные. К этим механизмам относятся RPC, COM, OLE, DDE, оконные сообщения (особенно WM_COPYDATA), буфер обмена, почтовые ящики, сокеты и т. д. Самый низкоуровневый механизм совместного использования данных на одной машине — проецирование файла в память. На нем так или иначе базируются все перечисленные мной механизмы разделения данных. Поэтому, если вас интересует максимальное быстродействие с минимумом издержек, лучше всего применять именно проецирование.

Совместное использование данных в этом случае происходит так: два или более процесса проецируют в память представления одного и того же объекта «проекция файла», т. е. делят одни и те же страницы физической памяти. В результате, когда один процесс записывает данные в представление общего объекта «проекция файла», изменения немедленно отражаются на представлениях в других процессах. Но при этом все процессы должны использовать одинаковое имя объекта «проекция файла».

А вот что происходит при запуске приложения. При открытии EXE-файла на диске система вызывает *CreateFile*, с помощью *CreateFileMapping* создает объект «проекция файла» и, наконец, вызывает *MapViewOfFileEx* (с флагом SEC_IMAGE) для отображения EXE-файла на адресное пространство

только что созданного процесса. *MapViewOfFileFx* вызывается вместо *MapViewOfFile*, чтобы представление файла было спроецировано но базовому адресу, значение которого хранится n самом EXE-файле. Потом создается первичный ноток процесса, адрес первого байта исполняемого кода в спроецированном представлении заносится в регистр указателя команд (IP), и процессор приступает к исполнению кода.

Если пользователь запустит второй экземпляр того же приложения, система увидит, что объект «проекция файла» для нужного EXE-файла уже существует и не станет создавать новый объект. Она просто спроецирует еще одно представление файла — на этот раз в контексте адресного пространства только что созданного второго процесса, т. е. одновременно спроецирует один и тот же файл на два адресных пространства. Это позволяет эффективнее использовать память, так как оба процесса делят одни и те же страницы физической памяти, содержащие порции исполняемого кода.

Как и все объекты ядра, проекции файлов можно совместно использовать из нескольких процессов тремя методами: наследованием описателей, именованием и дублированием описателей. Подробное объяснение этих трех методов см. в главе 3.

Файлы, проецируемые на физическую память из страничного файла

До сих пор мы говорили о методах, позволяющих проецировать представление файла, размещенного на диске. В то же время многие программы при выполнении создают данные, которые им нужно разделять с другими процессами. А создавать файл на диске и хранить там данные только с этой целью очень неудобно.

Прекрасно понимая это, Майкрософт добавила возможность проецирования файлов непосредственно на физическую память из страничного файла, а не из специально создаваемого дискового файла. Этот способ даже проще стандартного — основанного на создании дискового файла, проецируемого в память. Во-первых, не надо вызывать *CreateFile*, так как создавать или открывать специальный файл не требуется. Вы просто вызываете, как обычно, *CreateFileMapping* и передаете INVALID_HANDLE_VALUE в параметре *hFile*. Тем самым вы указываете системе, что создавать объект «проекция файла», физическая память которого находится на диске, не надо; вместо этого следует выделить физическую память из страничного файла. Объем выделяемой памяти определяется параметрами *dwMaxhnumSizeHigh* и *dwMaximumSizeLow*.

Создав объект «проекция файла» и спроецировав его представление на адресное пространство своего процесса, его можно использовать так же, как и любой другой регион памяти. Если вы хотите, чтобы данные стали доступны другим процессам, вызовите *CreateFileMapping* и передайте в параметре *pszName* строку с нулевым символом в конце. Тогда посторонние процессы — если им понадобится сюда доступ — смогут вызвать *CreateFileMapping* или *OpenFileMapping* и передать ей то же имя.

Когда необходимость в доступе к объекту «проекция файла» отпадет, процесс должен вызвать *CloseHandle*. Как только все описатели объекта будут закрыты, система освободит память, переданную из страничного файла.

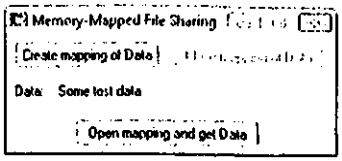
> **Примечание.** Есть одна интересная ловушка, в которую может попасть неискушенный программист. Попробуйте догадаться, что неверно в этом фрагменте кода:
>
> ```
> HANDLE hFile = CreateFile(…);
> HANDLE hMap = CreateFileNapping(hFile, …);
> if (hMap == NULL)
>    return(GetLastError());
> …
> ```
>
> Если вызов *CreateFile* не удастся, она вернет INVALID_HANDLE_VALUE. Но программист, написавший этот код, не дополнил его проверкой на успешное создание файла. Поэтому, когда в дальнейшем код обращается к функции *CreateFileMapping*, в параметре *hFile* ей передается INVALID_HANDLE_VALUE, что заставляет систему создать объект «проекция файла» из ресурсов страничного файла, а не из дискового файла, как предполагалось в программе. Весь последующий код, который использует проецируемый файл, будет работать правильно. Но при уничтожении объекта «проекция файла» все данные, записанные в спроецированную память (страничный файл), пропадут. И разработчик будет долго чесать затылок, пытаясь понять, в чем дело! Всегда проверяйте значение, которое возвращает *CreateFile*, так как причин, по которым ее вызов может закончиться неудачей, множество.

Программа-пример MMFShare

Эта программа (17-MMFShare.exe) демонстрирует, как происходит обмен данными между двумя и более процессами с помощью файлов, проецируемых в память. Файлы исходного кода и ресурсов этой программы находятся в каталоге 17-MMFShare внутри архива, доступного на веб-сайте поддержки этой книги.

Чтобы понаблюдать за происходящим, нужно запустить минимум две копии MMFShare. Каждый экземпляр программы создаст свое диалоговое окно.

Чтобы переслать данные из одной копии MMFShare в другую, наберите какой-нибудь текст в поле Data. Затем щелкните кнопку Create Mapping Of Data. Программа вызовет функцию *CreateFileMapping*, чтобы создать объект «проекция файла» размером 4 Кб и присвоить ему имя *MMFSharedData* (ресурсы выделяются объекту из страничного файла). Увидев, что объект с таким именем уже существует, программа выдаст сообщение, что не может создать объект. А если такого объекта нет, программа создаст объект, спроецирует представление файла на адресное пространство процесса и скопирует данные из поля Data в проецируемый файл.



Далее MMFShare прекратит проецировать представление файла, отключит кнопку Create Mapping Of Data и активизирует кнопку Close Mapping Of Data. На этот момент проецируемый в память файл с именем *MMFSharedData* будет просто «сидеть» где-то в системе. Никакие процессы пока не проецируют представление на данные, содержащиеся в файле.

Если вы теперь перейдете в другую копию MMFShare и щелкнете там кнопку Open Mapping And Get Data, программа попытается найти объект «проекция файла» с именем *MMFSharedData* через функцию *OpenFileMapping*. Если ей не удастся найти объект с таким именем, программа выдаст соответствующее сообщение. В ином случае она спроецирует представление объекта на адресное пространство своего процесса и скопирует данные из проецируемого файла в поле Data. Вот и все! Вы переслали данные из одного процесса в другой.

Кнопка Close Mapping Of Data служит для закрытия объекта «проекция файла», что высвобождает физическую память, занимаемую им в страничном файле. Если же объект «проекция файла» не существует, никакой другой экземпляр программы MMFShare не сможет открыть этот объект и получить от него данные. Кроме того, если один экземпляр программы создал объект «проекция файла», то остальным повторить его создание и тем самым перезаписать данные, содержащиеся в файле, уже не удастся.

```
MMFShare.cpp
/******************************************************************************
Module:  MMFShare.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_MMFSHARE);
```

```
   // инициализируем поле ввода текстовыми данными
   Edit_SetText(GetDlgItem(hWnd, IDC_DATA), TEXT("Some test data"));

   // отключаем кнопку Close, так как файл нельзя закрыть,
   // если он не создан или не открыт
   Button_Enable(GetDlgItem(hWnd, IDC_CLOSEFILE), FALSE);
   return(TRUE);
}

///////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   // описатель открытого файла, проецируемого в память
   static HANDLE s_hFileMap = NULL;

   switch (id) {
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;

      case IDC_CREATEFILE:
         if (codeNotify != BN_CLICKED)
            break;

         // создаем в памяти проецируемый файл с данными, набранными
         // в поле ввода; он занимает 4 KB и называется MMFSharedData
         // (память выделяется из страничного файла)
         s_hFileMap = CreateFileMapping(INVALID_HANDLE_VALUE, NULL,
            PAGE_READWRITE, 0, 4 * 1024, TEXT("MMFSharedData"));

         if (s_hFileMap != NULL) {

            if (GetLastError() == ERROR_ALREADY_EXISTS) {
               chMB("Mapping already exists - not created.");
               CloseHandle(s_hFileMap);

            } else {
               // создание проецируемого файла завершилось успешно;
               // проецируем представление файла на адресное пространство
               PVOID pView = MapViewOfFile(s_hFileMap,
                  FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

               if (pView != NULL) {
                  // поместим содержимое поля ввода в проецируемый файл
```

```
                Edit_GetText(GetDlgItem(hWnd, IDC_DATA),
                   (PTSTR) pView, 4 * 1024);

                // прекращаем проецирование; это защитит
                // данные от «блуждающих» указателей
                UnmapViewOfFile(pView);

                // пользователь не может создать сейчас еще один файл
                Button_Enable(hWndCtl, FALSE);

                // пользователь закрыл файл
                Button_Enable(GetDlgItem(hWnd, IDC_CLOSEFILE), TRUE);

             } else {
                chMB("Can't map view of file.");
             }
          }

       } else {
          chMB("Can't create file mapping.");
       }
       break;

    case IDC_CLOSEFILE:
       if (codeNotify != BN_CLICKED)
          break;

       if (CloseHandle(s_hFileMap)) {
          // пользователь закрыл файл; новый файл создать можно,
          // но закрыть его нельзя
          Button_Enable(GetDlgItem(hWnd, IDC_CREATEFILE), TRUE);
          Button_Enable(hWndCtl, FALSE);
       }
       break;

    case IDC_OPENFILE:
       if (codeNotify != BN_CLICKED)
          break;

       // смотрим: ну существует ли проецируемый в память файл
       // с именем MMFSharedData.
       HANDLE hFileMapT = OpenFileMapping(FILE_MAP_READ | FILE_MAP_WRITE,
          FALSE, TEXT("MMFSharedData"));

       if (hFileMapT != NULL) {
          // такой файл есть; проецируем его представление
```

```
            // на адресное пространство процесса
            PVOID pView = MapViewOfFile(hFileMapT,
               FILE_MAP_READ | FILE_MAP_WRITE, 0, 0, 0);

            if (pView != NULL) {

               // помещаем содержимое файла в поле ввода
               Edit_SetText(GetDlgItem(hWnd, IDC_DATA), (PTSTR) pView);
               UnmapViewOfFile(pView);
            } else {
               chMB("Can't map view.");
            }

            CloseHandle(hFileMapT);

         } else {
            chMB("Can't open mapping.");
         }
         break;
   }
}

////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}

////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR, int) {

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_MMFSHARE), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File ///////////////////////////////
```

Частичная передача физической памяти проецируемым файлам

До сих пор мы видели, что система требует передавать проецируемым файлам иск» физическую память либо из файла данных на диске, либо из страничного файла. Это значит, что намять используется не очень эффективно. Давайте вспомним то, что я говорил в разделе «В какой момент региону передают физическую память» главы 15. Допустим, вы хотите сделать всю таблицу доступной другому процессу. Если применить для этого механизм проецирования файлов, придется передать физическую память целой таблице:

```
CELLDATA CellData[200][256];
```

Если структура CELLDATA занимает 128 байтов, показанный массив потребует 6 553 600 (200x256x128) байтов физической памяти. Это слишком много — тем более, что в таблице обычно заполняют всего несколько строк.

Очевидно, что в данном случае, создав объект «проекция файла», желательно не передавать ему заранее всю физическую память. Функция *CreateFileMapping* предусматривает такую возможность, для чего в параметр *fdwProtect* нужно передать один из флагов: SEC_RESERVE или SEC_COMMIT.

Эти флаги имеют смысл, только если нм создаете объект «проекция файла», использующий физическую память из страничного файла. Флаг SEC_COMMIT заставляет *CreateFileMapping* сразу же передать память из страничного файла. (То же самое происходит, если никаких флагов не указано.) Но когда вы задаете флаг SEC_RESERVE, система не передает физическую память из страничного файла, а просто возвращает описатель объекта «проекция файла». Далее, вызвав *MapViewOfFile* или *MapViewfFielEx*, можно создать представление этого объекта. При этом *MapViewOfFile* или *MapViewOfFileEx* резервирует регион адресного пространства, не передавая ему физической памяти. Любая попытка обращения по одному из адресов зарезервированного региона приведет к нарушению доступа.

Таким образом, мы имеем регион зарезервированного адресного пространства и описатель объекта «проекция файла», идентифицирующий этот регион. Другие процессы могут использовать данный объект для проецирования представления того же региона адресного пространства. Физическая намять региону попрежнему не передается, гак что, если потоки в других процессах попытаются обратиться по одному из адресов представлении в своих регионах, они тоже вызовут нарушение доступа.

А теперь самое интересное. Оказывается, все, что нужно для передачи физической памяти общему (совместно используемому) региону, — вызвать функцию *VirtualAlloc*:

```
PVOID VirtualAlloc(
   PVOID pvAddross,
   SIZE_T dwSize,
```

```
DWORD fdwAllocationType,
DWORD fdwProtect);
```

Эту функцию мы уже рассматривали (и очень подробно) в главе 15. Вызвать *VirtualAlloc* для передачи физической памяти представлению региона — то же самое, что вызвать *VirtualAlloc* для передачи памяти региону, ранее зарезервированному вызовом *VirtualAlloc* с флагом MEM_RESERVE. Получается, что региону, зарезервированному функциями *MapViewOfFile* или *MapViewOfFileEx*, — как и региону, зарезервированному функцией *VirtualAlloc*, — тоже можно передавать физическую память порциями, а не всю сразу. И если вы поступаете именно так, учтите, что все процессы, спроецировавшие на этот регион представление одного и того же объекта «проекция файла», теперь тоже получат доступ к страницам физической памяти, переданным региону.

Итак, флаг SEC_RESERVE и функция *VirtualAlloc* позволяют сделать табличную матрицу *CellData* «общедоступной» и эффективнее использовать память.

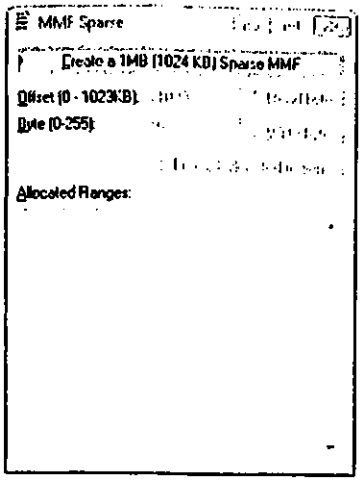
> **Примечание.** Функция *VirtualFree* не годится для возврата физической памяти, переданной в свое время проецируемому файлу (созданному с флагом SEC_RESERVE).

Файловая система NTFS поддерживает так называемые разреженные файлы (sparse files). Это потрясающая новинка. Она позволяет легко создавать и использовать разреженные проецируемые файлы (sparse memory-mapped files), которым физическая память предоставляется не из страничного, а из обычного дискового файла.

Вот пример того, как можно было бы воспользоваться этой новинкой. Допустим, вы хотите создать проецируемый в память файл (MMF) для записи аудиоданных. При этом вы должны записывать речь в виде цифровых аудиоданных в буфер памяти, связанный с дисковым файлом. Самый простой и эффективный способ решить эту задачу — применить разреженный MMF. Все дело в том, что вам заранее не известно, сколько времени будет говорить пользователь, прежде чем щелкнет кнопку Stop. Может, пять минут, а может, пять часов — разница большая! Однако при использовании разреженного MMF это не проблема.

Программа-пример MMFSparse

Эта программа (17-MMFSparse.exe), демонстрирует, как создать проецируемый в память файл, связанный с разреженным файлом NTFS. Файлы исходного кода и ресурсов этой программы находятся в каталоге 17-MMFSparse внутри архива, доступного на веб-сайте поддержки этой книги. После запуска MMFSparse на экране появляется окно, показанное ниже.

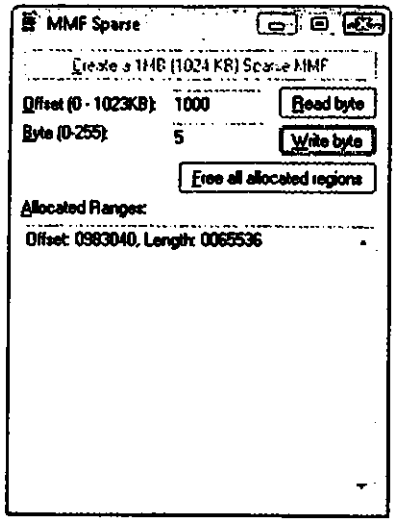


Когда вы щелкнете кнопку Create a 1MB (1024 KB) Sparse MMF, программа попытается создать разреженный файл «C:\MMFSparse». Если ваш диск C не является томом NTFS, у программы ничего не получится, и ее процесс завершится. А если вы создали том NTFS на каком-то другом диске, модифицируйте мою программу и перекомпилируйте ее, чтобы посмотреть, как она работает.

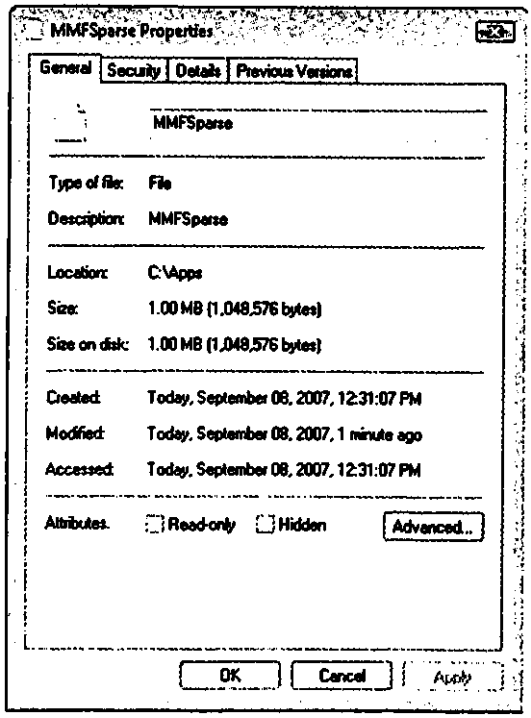
После создания разреженный файл проецируется на адресное пространство процесса. В поле Allocated Ranges (внизу окна) показывается, какие части файла действительно связаны с дисковой памятью. Изначально файл не связан ни с какой памятью, и в этом поле сообщается «No allocated ranges in the file» («В файле лет выделенных диапазонов»).

Чтобы считать байт, просто введите число в поле Offset и щелкните кнопку Read Byte. Введенное вами число умножается на 1024 (1 Кб), и программа, считав байт по полученному адресу, выводит его значение в поле Byte. Если адрес попадает в область, не связанную с физической памятью, в этом поле всегда показывается нулевой байт.

Для записи байта введите число в поле Offset, а значение байта (0-255) — в поле Byte. Потом, когда вы щелкнете кнопку Write Byte, смещение будет умножено на 1024, и байт по соответствующему адресу получит новое значение. Операция записи может заставить файловую систему передать физическую память какой-либо части файла. Содержимое поля Allocated Ranges обновляется после каждой операции чтения или записи, показывая, какие части файла связаны с физической памятью на данный момент. Вот как выглядит окно программы после записи всего одного байта по смещению 1 024 000 (1000x1024).



На этой иллюстрации видно, что физическая память выделена только одному диапазону адресов — размером 65 536 байтов, начиная с логического смещения 983 040 от начала файла. С помощью Explorer вы можете просмотреть свойства файла MMFSparse, как показано ниже.



Заметьте: на этой странице свойств сообщается, что длина файла равна 1 Мб (это виртуальный размер файла), но на деле он занимает на диске только 64 Кб.

Последняя кнопка, Free All Allocated Regions, заставляет программу высвободить всю физическую память, выделенную для файла; таким образом, соответствующее дисковое пространство освобождается, а все байты в файле обнуляются.

Теперь поговорим о том, как работает эта программа. Чтобы упростить ее исходный код, я создал C++-класс CSparseStream (который содержится в файле SparseStream.h). Этот класс инкапсулирует поддержку операций с разреженным файлом или потоком данных (stream). В файле MMFSparse.cpp я создал другой C++-класс, CMMFSparse, производный от CParseStream. Так что объект класса CMMFSparse обладает не только функциональностью CSparseStream, но и дополнительной, необходимой для использования разреженного потока данных как проецируемого в память файла. В процессе создается единственный глобальный экземпляр класса CMMFSparse — переменная *g_mmf*. Манипулируя разреженным проецируемым файлом, программа часто ссылается на эту глобальную переменную. Если файловая система тома не поддерживает разреженные файлы (это проверяется вызовом *CSparseStream::DoesFileSystemSupportSparseStreams*), обработчик сообщения WM_INITDIALOG говорит об ошибке, и работа программы завершается.

Когда пользователь щелкает кнопку Create a 1MB (1024 KB) Sparse MMF, программа вызывает *CreateFile* для создания нового файла в дисковом разделе NTFS 5. Пока что это обычный, самый заурядный файл. Но потом я вызываю метод Initialize глобального объекта *g_mmf*, передавая ему описатель и максимальный размер файла (1 Мб). Метод *Initialize* в свою очередь обращается к *CreateFileMapping* и создает объект ядра «проекция файла» указанного размера, а затем вызывает *MapViewOfFile*, чтобы сделать разреженный файл видимым в адресном пространстве данного процесса.

Когда *Initialize* возвращает управление, вызывается функция *Dlg_ShowAllocatedRanges*. Используя Windows-функции, она перечисляет диапазоны логических адресов в разреженном файле, которым передана физическая память. Начальное смещение и длина каждого такого диапазона показываются в нижнем поле диалогового окна. В момент инициализации объекта *g_mmf* файлу на диске еще не выделена физическая память, и данное поле отражает этот факт.

Теперь пользователь может попытаться считать или записать какие-то байты в пределах разреженного проецируемого файла. При записи программа извлекает значение байта и смещение из соответствующих полей, а затем помещает этот байт по вычисленному адресу в объект *g_mmf*. Такая операция может потребовать от файловой системы передачи физической памяти логическому блоку файла, но программа не принимает в этом участия.

При чтении объекта *g_mmf* возвращается либо реальное значение байта, если данному диапазону адресов передана физическая память, либо 0, если память не передана.

Моя программа также демонстрирует, как вернуть файл в исходное состояние, высвободив все выделенные ему диапазоны адресов (после этого он фактически не занимает места на диске). Реализуется это так. Пользователь щелкает кнопку Free All Allocated Regions. Однако освободить все диапазоны адресов, выделенные файлу, который проецируется в память, нельзя. Поэтому первое, что делает программа, — вызывает метод *ForceClose* объекта

g_mmf. Этот метод обращается к *UnmapViewOfFile*, а потом — к *CloseHandle*, передавая описатель объекта ядра «проекция файла».

Далее вызывается метод *DecommitPortionOfStream*, который освобождает всю память, выделенную логическим байтам в файле. Наконец, программа вновь обращается к методу *Initialize* объекта *g_mmf*, и тот повторно инициализирует файл, проецируемый на адресное пространство данного процесса. Чтобы подтвердить освобождение всей выделенной памяти, программа вызывает функцию *Dlg_ShowAllocatedRanges*, которая выводит в поле строку «No allocated ranges in the file».

И последнее. Используя разреженный проецируемый файл в реальном приложении, вы, наверное, захотите при закрытии файла урезать его логический размер до фактического. Отсечение концевой части разреженного файла, содержащей нулевые байты, не влияет на занимаемый им объем дискового пространства, но позволяет Explorer и команде dir сообщать точный размер файла. С этой целью вы должны после вызова метода *ForceClose* использовать функции *SetFilePointer* и *SetEndOfFile*.

> **Примечание.** Детали реализации проецируемых файлов, размер которых может увеличиваться, см. по ссылке (http://www.microsoft.com/msj/0499/win32/win320499.aspx).

MMFSparse.cpp

```
/******************************************************************************
Module:  MMFSparse.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..\CommonFiles\CmnHdr.h"     /* см. приложение A. */
#include <tchar.h>
#include <WindowsX.h>
#include <WinIoCtl.h>
#include "SparseStream.h"
#include <StrSafe.h>
#include "Resource.h"

//////////////////////////////////////////////////////////////////////////////

// этот класс упрощает работу с разреженными проецируемыми файлами
class CMMFSparse : public CSparseStream {
private:
   HANDLE m_hFileMap;      // объект «проекция файла»
   PVOID  m_pvFile;        // адрес начала проецируемого файла
```

```
public:
   // создает разреженный MMF и проецирует его на адресное
   // пространство процесса.
   CMMFSparse(HANDLE hStream = NULL, DWORD dwStreamSizeMaxLow = 0,
      DWORD dwStreamSizeMaxHigh = 0);

   // закрывает разреженный MMF
   virtual ~CMMFSparse() { ForceClose(); }

   // создает разреженный MMF и проецирует его на адресное
   // пространство процесса
   BOOL Initialize(HANDLE hStream, DWORD dwStreamSizeMaxLow,
      DWORD dwStreamSizeMaxHigh = 0);

   // оператор приведения MMF к BYTE возвращает адрес первого байта
   // в разреженном MMF
   operator PBYTE() const { return((PBYTE) m_pvFile); }

   // позволяет явно закрывать MMF, не дожидаясь вызова деструктора
   VOID ForceClose();
};

//////////////////////////////////////////////////////////////////////////////

CMMFSparse::CMMFSparse(HANDLE hStream, DWORD dwStreamSizeMaxLow,
   DWORD dwStreamSizeMaxHigh) {

   Initialize(hStream, dwStreamSizeMaxLow, dwStreamSizeMaxHigh);
}

//////////////////////////////////////////////////////////////////////////////

BOOL CMMFSparse::Initialize(HANDLE hStream, DWORD dwStreamSizeMaxLow,
   DWORD dwStreamSizeMaxHigh) {

   if (m_hFileMap != NULL)
      ForceClose();

   // инициализируем значение NULL на случай, если что-то пойдет не так
   m_hFileMap = m_pvFile = NULL;

   BOOL bOk = TRUE;  // предполагаем, что все будет хорошо

   if (hStream != NULL) {
      if ((dwStreamSizeMaxLow == 0) && (dwStreamSizeMaxHigh == 0)) {
         DebugBreak();  // недопустимый размер потока
      }
```

```
      CSparseStream::Initialize(hStream);
      bOk = MakeSparse();  // недопустимый размер потока данных
      if (bOk) {
         // создаем объект «проекция файла»
         m_hFileMap = ::CreateFileMapping(hStream, NULL, PAGE_READWRITE,
            dwStreamSizeMaxHigh, dwStreamSizeMaxLow, NULL);

         if (m_hFileMap != NULL) {
            // проецируем поток данных на адресное пространство процесса
            m_pvFile = ::MapViewOfFile(m_hFileMap,
               FILE_MAP_WRITE | FILE_MAP_READ, 0, 0, 0);
         } else {
            // спроецировать файлы не удалось; проводим очистку
            CSparseStream::Initialize(NULL);
            ForceClose();
            bOk = FALSE;
         }
      }
   }
   return(bOk);
}

//////////////////////////////////////////////////////////////////////////////

VOID CMMFSparse::ForceClose() {

   // очищаем все, что было успешно создано
   if (m_pvFile != NULL) {
      ::UnmapViewOfFile(m_pvFile);
      m_pvFile = NULL;
   }
   if (m_hFileMap != NULL) {
      ::CloseHandle(m_hFileMap);
      m_hFileMap = NULL;
   }
}

//////////////////////////////////////////////////////////////////////////////

#define STREAMSIZE      (1 * 1024 * 1024)    // 1 MB (1024 KB)
HANDLE g_hStream = INVALID_HANDLE_VALUE;
CMMFSparse g_mmf;
TCHAR g_szPathname[MAX_PATH] = TEXT("\0");

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {
```

```
   chSETDLGICONS(hWnd, IDI_MMFSPARSE);

   // инициализируем элементы управления в диалоговом окне
   EnableWindow(GetDlgItem(hWnd, IDC_OFFSET), FALSE);
   Edit_LimitText(GetDlgItem(hWnd, IDC_OFFSET), 4);
   SetDlgItemInt(hWnd, IDC_OFFSET, 1000, FALSE);

   EnableWindow(GetDlgItem(hWnd, IDC_BYTE), FALSE);
   Edit_LimitText(GetDlgItem(hWnd, IDC_BYTE), 3);
   SetDlgItemInt(hWnd, IDC_BYTE, 5, FALSE);

   EnableWindow(GetDlgItem(hWnd, IDC_WRITEBYTE), FALSE);
   EnableWindow(GetDlgItem(hWnd, IDC_READBYTE),  FALSE);
   EnableWindow(GetDlgItem(hWnd, IDC_FREEALLOCATEDREGIONS), FALSE);

   // записать файл в папку, доступную для записи
   GetCurrentDirectory(_countof(g_szPathname), g_szPathname);
   _tcscat_s(g_szPathname, _countof(g_szPathname), TEXT("\\MMFSparse"));

   // проверяем, поддерживает ли том разреженные файлы
   TCHAR szVolume[16];
   PTSTR pEndOfVolume = _tcschr(g_szPathname, _T('\\'));
   if (pEndOfVolume == NULL) {
      chFAIL("Impossible to find the Volume for the default document folder.");
      DestroyWindow(hWnd);
      return(TRUE);
   }
   _tcsncpy_s(szVolume, _countof(szVolume),
      g_szPathname, pEndOfVolume - g_szPathname + 1);
   if (!CSparseStream::DoesFileSystemSupportSparseStreams(szVolume)) {
      chFAIL("Volume of default document folder does not support sparse MMF.");
      DestroyWindow(hWnd);
      return(TRUE);
   }

   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_ShowAllocatedRanges(HWND hWnd) {

   // заполняем поле Allocated Ranges
   DWORD dwNumEntries;
```

```
   FILE_ALLOCATED_RANGE_BUFFER* pfarb =
      g_mmf.QueryAllocatedRanges(&dwNumEntries);

   if (dwNumEntries == 0) {
      SetDlgItemText(hWnd, IDC_FILESTATUS,
         TEXT("No allocated ranges in the file"));
   } else {
      TCHAR sz[4096] = { 0 };
      for (DWORD dwEntry = 0; dwEntry < dwNumEntries; dwEntry++) {
         StringCchPrintf(_tcschr(sz, _T('\0')), _countof(sz) - _tcslen(sz),
            TEXT("Offset: %7.7u, Length: %7.7u\r\n"),
            pfarb[dwEntry].FileOffset.LowPart, pfarb[dwEntry].Length.LowPart);
      }
      SetDlgItemText(hWnd, IDC_FILESTATUS, sz);
   }
   g_mmf.FreeAllocatedRanges(pfarb);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         if (g_hStream != INVALID_HANDLE_VALUE)
            CloseHandle(g_hStream);
         EndDialog(hWnd, id);
         break;

      case IDC_CREATEMMF:
         {
         g_hStream = CreateFile(g_szPathname, GENERIC_READ | GENERIC_WRITE,
            0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
         if (g_hStream == INVALID_HANDLE_VALUE) {
            chFAIL("Failed to create file.");
            return;
         }

         // используя этот файл, создаем MMF размером 1Мб (1024 Кб)
         if (!g_mmf.Initialize(g_hStream, STREAMSIZE)) {
            chFAIL("Failed to initialize Sparse MMF.");
            CloseHandle(g_hStream);
            g_hStream = NULL;
```

```
            return;
         }
         Dlg_ShowAllocatedRanges(hWnd);

         // активизируем или отключаем остальные элементы управления
         EnableWindow(GetDlgItem(hWnd, IDC_CREATEMMF), FALSE);
         EnableWindow(GetDlgItem(hWnd, IDC_OFFSET),    TRUE);
         EnableWindow(GetDlgItem(hWnd, IDC_BYTE),      TRUE);
         EnableWindow(GetDlgItem(hWnd, IDC_WRITEBYTE), TRUE);
         EnableWindow(GetDlgItem(hWnd, IDC_READBYTE),  TRUE);
         EnableWindow(GetDlgItem(hWnd, IDC_FREEALLOCATEDREGIONS), TRUE);

         // переводим фокус в поле Offset
         SetFocus(GetDlgItem(hWnd, IDC_OFFSET));
         }
         break;

      case IDC_WRITEBYTE:
         {
         BOOL bTranslated;
         DWORD dwOffset = GetDlgItemInt(hWnd, IDC_OFFSET, &bTranslated, FALSE);
         if (bTranslated) {
            g_mmf[dwOffset * 1024] = (BYTE)
               GetDlgItemInt(hWnd, IDC_BYTE, NULL, FALSE);
            Dlg_ShowAllocatedRanges(hWnd);
         }
         }
         break;

      case IDC_READBYTE:
         {
         BOOL bTranslated;
         DWORD dwOffset = GetDlgItemInt(hWnd, IDC_OFFSET, &bTranslated, FALSE);
         if (bTranslated) {
            SetDlgItemInt(hWnd, IDC_BYTE, g_mmf[dwOffset * 1024], FALSE);
            Dlg_ShowAllocatedRanges(hWnd);
         }
         }
         break;

      case IDC_FREEALLOCATEDREGIONS:
         // обычно проекцию файла закрывает деструктор, но в данном
         // случае мы хотим сами закрыть ее, чтобы можно было вернуть
         // часть файла в исходное состояние
         g_mmf.ForceClose();
```

```
         // мы вызываем ForceClose, потому что попытка обнуления
         // части файла при его  проецировании приводит к провалу
         // вызова DeviceIoControl с ошибкой ERROR_USER_MAPPED_FILE
         // ("запрошенная операция с файлом невозможна, пока открыт
         // раздел, проецируемый пользователем")
         g_mmf.DecommitPortionOfStream(0, STREAMSIZE);

         // для сброса разреженного состояние закроем описатель файла
         // и снова откроем его.
         CloseHandle(g_hStream);
         g_hStream = CreateFile(g_szPathname, GENERIC_READ | GENERIC_WRITE,
            0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
         if (g_hStream == INVALID_HANDLE_VALUE) {
            chFAIL("Failed to create file.");
            return;
         }

         // Reset the MMF wrapper for the new file handle.
         g_mmf.Initialize(g_hStream, STREAMSIZE);

         // Update the UI.
         Dlg_ShowAllocatedRanges(hWnd);
         break;
   }
}

//////////////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_MMFSPARSE), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////////
```

SparseStream.h

```
/******************************************************************************
Module:  SparseStream.h
Notices: Copyright (c) 2007 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..\CommonFiles\CmnHdr.h"     /* см. приложение A */
#include <WinIoCtl.h>

//////////////////////////////////////////////////////////////////////////////

#pragma once

//////////////////////////////////////////////////////////////////////////////

class CSparseStream {
public:
   static BOOL DoesFileSystemSupportSparseStreams(PCTSTR pszVolume);
   static BOOL DoesFileContainAnySparseStreams(PCTSTR pszPathname);

public:
   CSparseStream(HANDLE hStream = INVALID_HANDLE_VALUE) {
      Initialize(hStream);
   }

   virtual ~CSparseStream() { }

   void Initialize(HANDLE hStream = INVALID_HANDLE_VALUE) {
      m_hStream = hStream;
   }

public:
   operator HANDLE() const { return(m_hStream); }

public:
   BOOL IsStreamSparse() const;
   BOOL MakeSparse();
   BOOL DecommitPortionOfStream(
      __int64 qwFileOffsetStart, __int64 qwFileOffsetEnd);

   FILE_ALLOCATED_RANGE_BUFFER* QueryAllocatedRanges(PDWORD pdwNumEntries);
   BOOL FreeAllocatedRanges(FILE_ALLOCATED_RANGE_BUFFER* pfarb);

private:
   HANDLE m_hStream;
```

```
private:
   static BOOL AreFlagsSet(DWORD fdwFlagBits, DWORD fFlagsToCheck) {
      return((fdwFlagBits & fFlagsToCheck) == fFlagsToCheck);
   }
};

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::DoesFileSystemSupportSparseStreams(
   PCTSTR pszVolume) {

   DWORD dwFileSystemFlags = 0;
   BOOL bOk = GetVolumeInformation(pszVolume, NULL, 0, NULL, NULL,
      &dwFileSystemFlags, NULL, 0);
   bOk = bOk && AreFlagsSet(dwFileSystemFlags, FILE_SUPPORTS_SPARSE_FILES);
   return(bOk);
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::IsStreamSparse() const {

   BY_HANDLE_FILE_INFORMATION bhfi;
   GetFileInformationByHandle(m_hStream, &bhfi);
   return(AreFlagsSet(bhfi.dwFileAttributes, FILE_ATTRIBUTE_SPARSE_FILE));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::MakeSparse() {

   DWORD dw;
   return(DeviceIoControl(m_hStream, FSCTL_SET_SPARSE,
      NULL, 0, NULL, 0, &dw, NULL));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::DecommitPortionOfStream(
   __int64 qwOffsetStart, __int64 qwOffsetEnd) {

   // примечание: эта функция не сработает, если файл спроецирован в память
   DWORD dw;
   FILE_ZERO_DATA_INFORMATION fzdi;
   fzdi.FileOffset.QuadPart = qwOffsetStart;
   fzdi.BeyondFinalZero.QuadPart = qwOffsetEnd + 1;
```

```
   return(DeviceIoControl(m_hStream, FSCTL_SET_ZERO_DATA, (PVOID) &fzdi,
      sizeof(fzdi), NULL, 0, &dw, NULL));
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::DoesFileContainAnySparseStreams(
   PCTSTR pszPathname) {

   DWORD dw = GetFileAttributes(pszPathname);
   return((dw == 0xffffffff)
      ? FALSE : AreFlagsSet(dw, FILE_ATTRIBUTE_SPARSE_FILE));
}

//////////////////////////////////////////////////////////////////////////////

inline FILE_ALLOCATED_RANGE_BUFFER* CSparseStream::QueryAllocatedRanges(
   PDWORD pdwNumEntries) {

   FILE_ALLOCATED_RANGE_BUFFER farb;
   farb.FileOffset.QuadPart = 0;
   farb.Length.LowPart =
      GetFileSize(m_hStream, (PDWORD) &farb.Length.HighPart);

   // правильно определить размер блока памяти до попыток сбора этих
   // данных нельзя, и я просто беру 100 * sizeof(*pfarb)
   DWORD cb = 100 * sizeof(farb);
   FILE_ALLOCATED_RANGE_BUFFER* pfarb = (FILE_ALLOCATED_RANGE_BUFFER*)
      HeapAlloc(GetProcessHeap(), HEAP_ZERO_MEMORY, cb);

   DeviceIoControl(m_hStream, FSCTL_QUERY_ALLOCATED_RANGES,
      &farb, sizeof(farb), pfarb, cb, &cb, NULL);
   *pdwNumEntries = cb / sizeof(*pfarb);
   return(pfarb);
}

//////////////////////////////////////////////////////////////////////////////

inline BOOL CSparseStream::FreeAllocatedRanges(
   FILE_ALLOCATED_RANGE_BUFFER* pfarb) {

   // освобождаем выделенную память
   return(HeapFree(GetProcessHeap(), 0, pfarb));
}

//////////////////////////////// End Of File /////////////////////////////////
```

Оглавление

ГЛАВА 18

Динамически распределяемая память

Третий, и последний, механизм управления памятью — динамически распределяемые области памяти, или кучи (heaps). Они весьма удобны при создании множества небольших блоков данных. Например, связанными списками и деревьями проще манипулировать, используя именно кучи, а не виртуальную память (глава 15) или файлы, проецируемые в память (глава 17). Преимущество динамически распределяемой памяти в том, что она позволяет вам игнорировать гранулярность выделения памяти и размер страниц и сосредоточиться непосредственно на своей задаче. А недостаток — выделение и освобождение блоков памяти проходит медленнее, чем при использовании других механизмов, и, кроме того, вы теряете прямой контроль над передачей физической памяти и ее возвратом системе.

Куча — это регион зарезервированного адресного пространства. Первоначально большей его части физическая память не передается. По мере того, как программа занимает эту область под данные, специальный диспетчер, управляющий кучами (heap manager), постранично передает ей физическую память (из страничного файла). А при освобождении блоков в куче диспетчер возвращает системе соответствующие страницы физической памяти.

Майкрософт не документирует правила, по которым диспетчер передает или отбирает физическую память. Майкрософт постоянно проводит стрессовое тестирование своих операционных систем и прогоняет разные сценарии, чтобы определить, какие правила в большинстве случаев работают лучше. Их приходится менять по мере появления как нового программного обеспечения, так и оборудования. Если эти правила важны вашим программам, использовать динамически распределяемую память не стоит — работайте с функциями виртуальной памяти (т. е. *VirtualAlloc* и *VirtualFree),* и тогда вы сможете сами контролировать эти правила.

Стандартная куча процесса

При инициализации процесса система создает n сто адресном пространстве стандартную кучу (process's default heap). Не размер по умолчанию — 1 Мб. Но система позволяет увеличивать этот размер, для чего надо указать компоновщику при сборке программы ключ /HEAP. (Однако при сборке DLL этим ключом пользоваться нельзя, так как для DLL куча нс создастся.)

```
/HEAP:reserve[, commit]
```

Стандартная куча процесса необходима многим Windows-функциям. Например, функции ядра Windows выполняют все операции с использованием Unicode-символов и строк. Если вызвать ANSI-версию какой-нибудь Windows-функции, ей придется, преобразовав строки из ANSI в Unicode, вызывать свою Unicode-версию. Для преобразования строк ANSI-функции нужно выделить блок памяти, в котором она размещает Unicode-версию строки. Этот блок памяти заимствуется из стандартной кучи вызывающего процесса. Есть и другие функции, использующие временные блоки памяти, которые тоже выделяются из стандартной кучи процесса. Из нее же черпают себе память и функции 16-разрядной Windows, управляющие кучами (*LocalAlloc* и *GlobalAlloc*).

Поскольку стандартную кучу процесса используют многие Windows-функции, а потоки вашего приложения могут одновременно вызвать массу таких функций, доступ к этой куче разрешается только по очереди. Иными словами, система гарантирует, что в каждый момент времени только один поток сможет выделить или освободить блок памяти в этой куче. Если же два потока попытаются выделить в ней блоки памяти одновременно, второй поток будет ждать, пока первый поток не выделит свой блок. Принцип последовательного доступа потоков к куче немного снижает производительность многопоточной программы. Если в программе всего один поток, для быстрейшего доступа к куче нужно создать отдельную кучу и не использовать стандартную. Но Windows-функциям этого, увы, не прикажешь — они работают с кучей только последнего типа.

Как я уже говорил, куч у одного процесса может быть несколько. Они создаются и разрушаются в период его существования. Но стандартная куча процесса создается в начале его исполнения и автоматически уничтожается по его завершении — сами уничтожить ее вы не можете. Каждую кучу идентифицирует свой описатель, и все Windows-функции, которые выделяют и освобождают блоки в ее пределах, требуют передавать им этот описатель как параметр.

Описатель стандартной кучи процесса возвращает функция *GetProcessHeap*:

```
HANDLE GetProcessHeap();
```

Дополнительные кучи в процессе

В адресном пространстве процесса допускается создание дополнительных куч. Для чего они нужны? Тому может быть несколько причин:

- защита компонентов;
- более эффективное управление памятью;
- локальный доступ;
- исключение издержек, связанных с синхронизацией потоков;
- быстрое освобождение всей памяти в куче. Рассмотрим эти причины подробнее.

Защита компонентов

Допустим, программа должна обрабатывать два компонента: связанный список структур NODE и двоичное дерево структур BRANCH. Представим также, что у вас есть два файла исходного кода: LnkLst.cpp, содержащий функции для обработки связанного списка, и BinTree.cpp с функциями для обработки двоичного дерева.

Если структуры NODE и BRANCH хранятся в одной куче, то она может выглядеть примерно так, как показано на рис. 18-1.

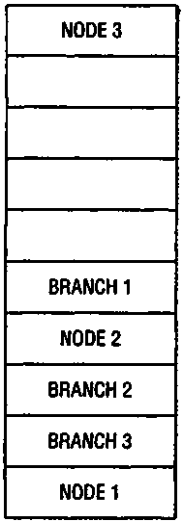


Рис. 18-1. *Единая куча, в которой размещены структуры NODE и BRANCH*

Теперь предположим, что в коде, обрабатывающем связанный список, «сидит жучок», который приводит к случайной перезаписи 8 байтов после NODE 1. А это в свою очередь влечет порчу данных в BRANCH 3. Впоследствии, когда код из файла BinTree.cpp пытается «пройти» по двоичному дереву, происходит сбой из-за того, что часть данных в памяти испор-

чена. Можно подумать, что ошибка возникает из-за «жучка» в коде двоичного дерева, тогда как на самом деле он — в коде связанного списка. А поскольку разные типы объектов смешаны в одну кучу (в прямом и переносном смысле), то отловить «жучков» в коде становится гораздо труднее.

Создав же две отдельные кучи — одну для NODE, другую для BRANCH, — вы локализуете место возникновения ошибки. И тогда «жучок» в коде связанного списка нс испортит целостности двоичного дерева, и наоборот. Конечно, всегда остается вероятность такой фатальной ошибки в коде, которая приведет к записи данных в постороннюю кучу, но это случается значительно реже.

Более эффективное управление памятью

Кучами можно управлять гораздо эффективнее, создавая в них объекты одинакового размера. Допустим, каждая структура NODE занимает 24 байта, а каждая структура BRANCH — 32. Память для всех этих объектов выделяется из одной кучи. На рис. 18-2 показано, как выглядит полностью занятая куча с несколькими объектами NODE и BRANCH. Если объекты NODE 2 и NODE 4 удаляются, память в куче становится фрагментированной. И если после этого попытаться выделить в ней память для структуры BRANCH, ничего не выйдет — даже несмотря на то что в куче свободно 48 байтов, а структура BRANCH требует всего 32.

Если бы в каждой куче содержались объекты одинакового размера, удаление одного из них позволило бы в дальнейшем разместить другой объект того же типа.

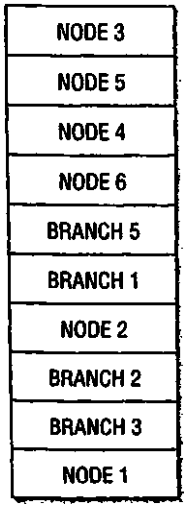


Рис. 18-2. *Фрагментированная куча, содержащая несколько объектов NODE* и *BRANCH*

Локальный доступ

Перекачка страницы из оперативной памяти в страничный файл занимает ощутимое время. Та же задержка происходит и в момент загрузки страницы данных обратно в оперативную память. Обращаясь в основном к памяти, локализованной в небольшом диапазоне адресов, вы снизите вероятность перекачки страниц между оперативной памятью и страничным файлом.

Поэтому при разработке приложения старайтесь размещать объекты, к которым необходим частый доступ, как можно плотнее друг к другу. Возвращаясь к примеру со связанным списком и двоичным деревом, отмечу, что просмотр списка не связан с просмотром двоичного дерева. Разместив все структуры NODE друг за другом в одной куче, вы, возможно добьетесь того, что по крайней мере несколько структур NODE уместятся в пределах одной страницы физической памяти. И тогда просмотр связанного списка не потребует от процессора при каждом обращении к какой-либо структуре NODE переключаться с одной страницы на другую.

Если же «свалить» оба типа структур в одну кучу, объекты NODE необязательно будут размещены строго друг за другом. При самом неблагоприятном стечении обстоятельств на странице окажется всего одна структура NODE, а остальное место займут структуры BRANCH. В этом случае просмотр связанного списка будет приводить к ошибке страницы (page fault) при обращении к каждой структуре NODE, что в результате может чрезвычайно замедлить скорость выполнения вашего процесса.

Исключение издержек, связанных с синхронизацией потоков

Доступ к кучам упорядочивается по умолчанию, поэтому при одновременном обращении нескольких потоков к куче данные в ней никогда не повреждаются. Однако для этого функциям, работающим с кучами, приходится выполнять дополнительный код. Если вы интенсивно манипулируете с динамически распределяемой памятью, выполнение дополнительного кода может заметно снизить быстродействие вашей программы. Создавая новую кучу, вы можете сообщить системе, что единовременно к этой куче обращается только один поток, и тогда дополнительный код выполняться не будет. Но берегитесь: теперь вы берете всю ответственность за целостность этой кучи на себя. Система не станет присматривать за вами.

Быстрое освобождение всей памяти в куче

Наконец, использование отдельной кучи для какой-то структуры данных позволяет освобождать всю кучу, не перебирая каждый блок памяти. Например, когда Windows Explorer перечисляет иерархию каталогов на жестком диске, он формирует их дерево в памяти. Получив команду обновить эту информацию, он мог бы просто разрушить кучу, содержащую это дерево,

и начать все заново (если бы, конечно, он использовал кучу, выделенную только для информации о дереве каталогов). Во многих приложениях это было бы очень удобно, да и быстродействие тоже возросло бы.

Создание дополнительной кучи

Дополнительные кучи в процессе создаются вызовом *HeapCreate:*

```
HANDLE HeapCreate(
    DWORD fdwOptions,
    SIZE_T dwInitialSize,
    SIZE_T dwHaximumSize);
```

Параметр *fdwOptions* модифицирует способ выполнения операций над кучей. В нем можно указать 0, HEAP_NO_SERIALIZE, HEAP_GENERATE_EXCEPTIONS, HEAP_CREATE_ENABLE_EXECUTE или комбинацию ЭТИХ флагов.

По умолчанию действует принцип последовательного доступа к куче, что позволяет не опасаться одновременного обращения к ней сразу нескольких потоков. При попытке выделения из кучи блока памяти функция *HeapAlloc* (ее параметры мы обсудим чуть позже) делает следующее:

1. Просматривает связанный список выделенных и свободных блоков памяти.
2. Находит адрес свободного блока.
3. Выделяет новый блок, помечая свободный как занятый.
4. Добавляет новый элемент в связанный список блоков памяти.

Флаг HEAP_NO_SERIALIZE использовать не следует, и вот почему. Допустим, два потока одновременно пытаются выделить блоки памяти из одной кучи. Первый поток выполняет операции по пп. 1 и 2 и получает адрес свободного блока памяти. Но только он соберется перейти к третьему этапу, как его вытеснит второй поток и тоже выполнит операции по пп. 1 и 2. Поскольку первый поток не успел дойти до этапа 3, второй ноток обнаружит тот же свободный блок памяти.

Итак, оба потока считают, что они нашли свободный блок памяти в куче. Поэтому поток 1 обновляет связанный список, помечая новый блок как занятый. После этого и поток 2 обновляет связанный список, помечая *тот же* блок как занятый. Ни один из потоков пока ничего не подозревает, хотя оба получили адреса, указывающие на один и тот же блок памяти.

Ошибку такого рода обнаружить очень трудно, поскольку она проявляется не сразу. Но в конце концов сбой произойдет и, будьте уверены, это случится в самый неподходящий момент. Вот какие проблемы это может вызвать:

- Повреждение связанного списка блоков памяти. Эта проблема не проявится до попытки выделения или освобождения блока.
- Оба потока делят один и тот же блок памяти. Оба записывают в него свою информацию. Когда поток 1 начнет просматривать содержимое блока, он не поймет данные, записанные потоком 2.

- Один из потоков, закончив работу с блоком, освобождает его, и это приводит к тому, что другой поток записывает данные в невыделенную память. Происходит повреждение кучи.

Решение этих проблем — предоставить одному из потоков монопольный доступ к куче и ее связанному списку (пока он не закончит все необходимые операции с кучей). Именно так и происходит в отсутствие флага HEAP_NO_SERIALIZE. Этот флаг можно использовать без опаски только при выполнении следующих условий:

- в процессе существует лишь один поток;
- в процессе несколько потоков, но с кучей работает лишь один из них;
- в процессе несколько потоков, но он сам регулирует доступ потоков к куче, применяя различные формы взаимоисключения, например критические секции, объекты-мьютексы или семафоры (см. главы 8 и 9).

Если вы не уверены, нужен ли вам флаг HEAP_NO_SERIALIZE, лучше не пользуйтесь им. В его отсутствие скорость работы многопоточной программы может чуть снизиться из-за задержек при вызовах функций, управляющих кучами, но зато вы избежите риска повреждения кучи и ее данных.

Другой флаг, HEAP_GENERATE_EXCEPTIONS, заставляет систему генерировать исключение при любом провале попытки выделения блока в куче. Исключение (см. главы 23,24 и 25) — еще один способ уведомления программы об ошибке. Иногда приложение удобнее разрабатывать, полагаясь на перехват исключений, а не на проверку значений, возвращаемых функциями.

> **Примечание.** По умолчанию, если при вызове *Heap**-функций операционная система обнаружит повреждение кучи (например, запись за пределами выделенного блока), не произойдет ничего особенного, если вы не отлаживаете программу. Эти недостатки часто использовались для хакерских атак, что вынудило Майкрософт реализовать дополнительные механизмы для обнаружения повреждений кучи.
>
> Теперь диспетчер кучи может генерировать исключение, если при вызове любой из *Heap**-функций будет обнаружено повреждение кучи. Для этого следует исполнить следующий код:
>
> ```
> HeapSetInformation(NULL, HeapEnableTerminationOnCorruption, NULL, 0);
> ```
>
> Первый параметр *HeapSetInformation* игнорируется, если во втором параметре передан флаг *HeapEnableTerminationOnCorruption*. В результате ко всем кучам процесса применяется указанная строгая политика. Кстати, активировав эту политику для процесса, вы уже не сможете отменить ее.

Для хранения в куче исполняемого кода служит последний флаг — HEAP_CREATE_ENABLE_EXECUTE. Это особенно важно, если активен защитный механизм Data Execution Prevention (см. главу 13). Если вы не установите этот флаг для блока с кодом, хранящегося в куче, Windows сгенерирует исключение EXCEPTION_ACCESS_VIOLATION.

Второй параметр функции *HeapCreate* — *dwInitiallSize* — определяет количество байтов, первоначально передаваемых куче. При необходимости функция округляет это значение до ближайшей большей величины, кратной размеру страниц. И последний параметр, *dwMaximumSize*, указывает максимальный объем, до которого может расширяться куча (предельный объем адресного пространства, резервируемого под кучу). Если он больше 0, вы создадите кучу именно такого размера и нс сможете его увеличить. А если этот параметр равен 0, система резервирует регион и, если надо, расширяет его до максимально возможного объема. При успешном создании кучи *HeapCreate* возвращает описатель, идентифицирующий новую кучу. Он используется и другими функциями, работающими с кучами.

Выделение блока памяти из кучи

Для этого достаточно вызвать функцию *HeapAlloc*:

```
PV0ID HeapAlloc(
    HANDLE hHeap,
    DWORD fdwFlags,
    SIZE_T dwBytes);
```

Параметр *hHeap* идентифицирует описатель кучи, из которой выделяется память. Параметр *dwBytes* определяет число выделяемых в куче байтов, а параметр *fdwFlags* позволяет указывать флаги, влияющие на характер выделения памяти. В настоящее время поддерживается только три флага: HEAP_ZERO_MEMORY, HEAP_GENERATE_EXCEPTIONS и HEAP_NO_SERIALIZE.

Назначение флага HEAP_ZERO_MEMORY очевидно. Он приводит к заполнению содержимого блока нулями перед возвратом из *HeapAlloc*. Второй флаг заставляет эту функцию генерировать программное исключение, если в куче не хватает памяти для удовлетворения запроса. Вспомните, этот флаг можно указывать и при создании кучи функцией *HeapCreate*; он сообщает диспетчеру, управляющему кучами, что при невозможности выделения блока в куче надо генерировать соответствующее исключение. Если вы включили данный флаг при вызове *HeapCreate*, то при вызове *HeapAlloc* указывать его уже не нужно. С другой стороны, вы могли создать кучу без флага HEAP_GENERATE_EXCEPTIONS. В таком случае, если вы укажете его при вызове *HeapAlloc*, он повлияет лишь на данный ее вызов.

Если функция *HeapAlloc* завершилась неудачно и при этом разрешено генерировать исключения, она может вызвать одно из двух исключений, перечисленных в следующей таблице.

Табл. 18-1. Исключения, генерируемые функцией *HeapAlloc*

Идентификатор	Описание
STATUS_NO_MEMORY	Попытка выделения памяти не удалась из-за ее нехватки
STATUS_ACCESS_VIOLATION	Попытка выделения памяти не удалась из-за повреждения кучи или неверных параметров функции

При успешном выделении блока *HeapAlloc* возвращает его адрес. Если памяти недостаточно и флаг HEAP_GENERATE_EXCEPTIONS не указан, функция возвращает NULL.

Флаг HEAP_NO_SERIALIZE заставляет *HeapAlloc* при данном вызове не применять принцип последовательного доступа к куче. Этим флагом нужно пользоваться с величайшей осторожностью, так как куча (особенно стандартная куча процесса) может быть повреждена при одновременном доступе к ней нескольких потоков.

> **Примечание.** Для выделения больших блоков памяти (от 1 Мб) рекомендуется использовать функцию *VirtualAlloc*, а не функции, оперирующие с кучами.

При выделении множества блоков различного размера алгоритм, который диспетчер кучи использует по умолчанию, может привести к фрагментации адресного пространства кучи. При этом в куче может не оказаться свободного блока подходящего размера, хотя вполне может быть несколько блоков, меньших по размеру. В Windows XP и Windows Server 2003 разработчик может заставить систему использовать для выделения памяти в куче алгоритм, снижающий фрагментацию кучи. Такая куча особенно быстро работает на многопроцессорных компьютерах. Задействовать это алгоритм поможет следующий код:

```
ULONG HeapInformationValue = 2;
if (HeapSetInformation(
    hHeap, HeapCompatibilityInformation,
    &HeapInformationValue, sizeof(HeapInformationValue)) {
    // переключаем hHeap в режии низкой фрагментации
} else {
    // hHeap невозможно переключить в режим низкой фрагментации,
    // возможно, из-за того, что куча создана с флагом HEAP_NO_SERIALIZE.
}
```

Если передать описатель, который вернула *GetPivcessHeap*, функции *HeapSetInformation*, можно переключить кучу в режим низкой фрагментации. Вызов *HeapSetInformation* окончится неудачей, если передать ей флаг HEAP_NO_SERIALIZE. Заметьте, что некоторые параметры отладчика также блокируют переключение кучи в режим низкой фрагментации. Чтобы

отключить их, установите переменную окружения _NO_DEBUG_HEAP в 1. Заметьте также, что диспетчер кучи отслеживает выделяемую память и оптимизирует кучу. Например, диспетчер может автоматически переключить кучу в режим низкой фрагментации, если сочтет, что это повысит производительность вашей программы.

Изменение размера блока

Часто бывает необходимо изменить размер блока памяти. Некоторые приложения изначально выделяют больший, чем нужно, блок, а затем, разместив в нем данные, уменьшают его. Но некоторые, наоборот, сначала выделяют небольшой блок памяти и потом увеличивают его по мере записи новых данных. Для изменения размера блока памяти вызывается функция *HeapReAlloc*.

```
PVOID HeapReAlloc(
    HANDLE hHeap,
    DWORD fdwFlags,
    PVOID pvMem,
    SIZE_T dwBytes);
```

Как всегда, параметр *hHeap* идентифицирует кучу, в которой содержится изменяемый блок. Параметр *fdwHags* указывает флаги, используемые при изменении размера блока: HEAP_GENERATE_EXCEPTIONS, HEAP_NO_SERIALIZE, HEAP_ZERO_MEMORY или HEAP_REALLOCK_IN_PLACE_ONLY.

Первые два флага имеют тот же смысл, что и при использовании с *HeapAlloc*. Флаг HEAP_ZERO_MEMORY полезен только при увеличении размера блока памяти, в этом случае дополнительные байты, включаемые в блок, предварительно обнуляются. При уменьшении размера блока этот флаг не действует.

Флаг HEAP_REALLOC_IN_PLACE_ONLY сообщает *HeapReAlloc*, что данный блок памяти перемещать внутри кучи не разрешается (а именно это и может попытаться сделать функция при расширении блока). Если функция сможет расширить блок без его перемещения, она расширит его и вернет исходный адрес блока. С другой стороны, если для расширения блока его надо переместить, она возвращает адрес нового, большего по размеру блока. Если блок затем снова уменьшается, функция вновь возвращает исходный адрес первоначального блока. Флаг HEAP_REALLOC_IN_PLACE_ONLY имеет смысл указывать, когда блок является частью связанного списка или дерева, в этом случае в других узлах списка или дерева могут содержаться указатели на данный узел, и его перемещение в куче непременно приведет к нарушению целостности связанного списка.

Остальные два параметра (*pvMem* и *dwBytes*) определяют текущий адрес изменяемого блока и его новый размер (в байтах). Функция *HeapReAlloc* возвращает либо адрес нового, измененного блока, либо NULL, если размер блока изменить не удалось.

Определение размера блока

Выделив блок памяти, можно вызвать *HeapSize* и узнать его истинный размер:

```
SIZE_T HeapSize(
    HANDLE hHeap,
    DWORD fdwFlags,
    LPCVOID pvMem);
```

Параметр *hHeap* идентифицирует кучу, а параметр *pvMem* сообщает адрес блока. Параметр *fdwFlags* принимает два значения: 0 или HEAP_NO_SERIALIZE.

Освобождение блока

Для этого служит функция *HeapFree*:

```
BOOL HeapFree(
    HANDLE hHeap,
    DWORD fdwFlags,
    PVOID pvMem);
```

Она освобождает блок памяти и при успешном вызове возвращает TRUE. Параметр *fdwFtegs* принимает два значения: 0 или HEAP_NO_SERIALIZE. Обращение к этой функции может привести к тому, что диспетчер, управляющий кучами, вернет часть физической памяти системе, но это не обязательно.

Уничтожение кучи

Кучу можно уничтожить вызовом *HeapDestroy*:

```
BOOL HeapDestroy(HANDLE hHeap);
```

Обращение к этой функции приводит к освобождению всех блоков памяти внутри кучи и возврату системе физической памяти и зарезервированного региона адресного пространства, занятых кучей. При успешном выполнении функция возвращает TRUE. Если при завершении процесса вы не уничтожаете кучу, это делает система, но — подчеркну еще раз — только в момент завершения процесса. Если куча создана потоком, она будет уничтожена лишь при завершении всего процесса.

Система не позволит уничтожить стандартную кучу процесса — она разрушается только при завершении процесса. Если вы передадите описатель этой кучи функции *HeapDestroy*, система просто проигнорирует ваш вызов.

Использование куч в программах на C++

Чтобы в полной мере использовать преимущества динамически распределяемой памяти, следует включить ее поддержку в существующие программы, написанные на C++. В этом языке выделение памяти для объекта класса выполняется вызовом оператора *new*, а не функцией *malloc*, как в обычной биб-

лиотеке C. Когда необходимость в данном объекте класса отпадает, вместо библиотечной C-функции *free* следует применять оператор *delete*. Скажем, у нас есть класс CSomcClass, и мы хотим создать экземпляр этого класса. Для этого нужно написать что-то вроде:

```
CSomeClass* pSomeClass = new CSomeClass;
```

Дойдя до этой строки, компилятор C++ сначала проверит, содержит ли класс CSomeClass функцию-член, переопределяющую оператор *new*. Если да, компилятор генерирует код для вызова этой функции. Нет — создает код для вызова стандартного C++-оператора *new*.

Созданный объект уничтожается обращением к оператору *detete*:

```
delete pSomeClass;
```

Переопределяя операторы *new* и *delete* для нашего C++-класса, мы получаем возможность использовать преимущества функций, управляющих кучами. Для этого определим класс CSomeClass в заголовочном файле, скажем, так:

```
class CSomeClass {
private:

    static HANDLE s_hHeap;
    static UINT s_uNuroAllocsInHeap;

    // здесь располагаются закрытые данные и функции-члены
    …
public:
    void* operator new (size_t size);
    void operator delete (void* p);
    // здесь располагаются открытые данные и функции-члены
    …
};
```

Я объявил два элемента данных, *s_hHeap* и *s_hNumAllocsInHeap*, как статические переменные. А раз так, то компилятор C++ заставит все экземпляры класса CSomeClass использовать одни и те же переменные. Иначе говоря, он не станет выделять отдельные переменные *s_hHeap* и *s_uNumAllocsInHeap* для каждого создаваемого экземпляра класса. Это очень важно: ведь мы хотим, чтобы все экземпляры класса CSomeClass были созданы в одной куче.

Переменная *s_hHeap* будет содержать описатель кучи, в которой создаются объекты CSomeClass. Переменная *s_uNumAllocsInHeap* — просто счетчик созданных в куче объектов CSomeClass. Она увеличивается на 1 при создании в куче нового объекта CSomeClass и соответственно уменьшается при его уничтожении. Когда счетчик обнуляется, куча освобождается. Для управления кучей в CPP-файл следует включить примерно такой код:

```
HANDLE CSomeClass::s_hHeap = NULL;
UINT CSomeClass::s_uNumAllocsInHeap = 0;
```

```
void* CSomeClass::operator new (size_t size) {
    if (s_hHeap == NULL) {
        // куча не существует; создаем ее
        s.hHeap = HeapCreate(HEAP_NO_SERIALIZE, 0, 0);

        if (s.hHeap == NULL)
            return(NULL);
    }
    // куча для объектов CSomeClass существует
    void* p = HeapAlloc(s_hHeap, 0, size);

    if (p != NULL) {
        // память выделена успешно; увеличиваем счетчик объектов CSomeClass в куче
        s_uNumAllocsInHeap++;
    }

    // возвращаем адрес созданного объекта CSomeClass
    return(p);
}
```

Заметьте, что сначала я объявил два статических элемента данных, *s_hHeap* и *s_uNumAllocsInHeap,* а затем инициализировал их значениями NULL и 0 соответственно.

Оператор *new* принимает один параметр — *size*, указывающий число байтов, нужных для хранения CSomeClass. Первым делом он создает кучу, если таковой нет. Для проверки анализируется значение переменной *s_hHeap*: если оно NULL, кучи нет, и тогда она создается функцией *HeapCreate*, а описатель, возвращаемый функцией, сохраняется в переменной *s_hHeap*, чтобы при следующем вызове оператора *new* использовать существующую кучу, а не создавать еще одну.

Вызывая *HeapCreate,* я указал флаг HEAP_NO_SERIALIZE, потому что данная программа построена как однопоточная. Остальные параметры, указанные при вызове *HeapCreate*, определяют начальный и максимальный размер кучи. Я подставил на их место по нулю. Первый нуль означает, что у кучи нет начального размера, второй — что куча должна расширяться по мере необходимости.

Не исключено, что вам показалось, будто параметр *size* оператора *new* стоит передать в *HeapCreate* как второй параметр. Вроде бы тогда можно инициализировать кучу так, чтобы она была достаточно большой для размещения одного экземпляра класса. И в таком случае функция *HeapAlloc* при первом вызове работала бы быстрее, так как не пришлось бы изменять размер кучи под экземпляр класса. Увы, мир устроен не так, как хотелось бы. Из-за того, что с каждым выделенным внутри кучи блоком памяти связан свой заголовок, при вызове *HeapAlloc* все равно пришлось бы менять размер кучи, чтобы в нее поместился не только экземпляр класса, но и связанный с ним заголовок.

После создания кучи из нее можно выделять намять под новые объекты CSomeClass с помощью функции *HeapAlloc*. Первый параметр — описатель кучи, второй — размер объекта CSomeClass. Функция возвращает адрес выделенного блока.

Если выделение прошло успешно, я увеличиваю переменную-счетчик *s_uNumAllocsInHeap*, чтобы знать число выделенных блоков в куче. Наконец, оператор *new* возвращает адрес только что созданного объекта CSomeClass.

Вот так происходит создание нового объекта CSomeClass. Теперь рассмотрим, как этот объект разрушается, — если он больше не нужен программе. Эта задача возлагается на функцию, переопределяющую оператор *delete*:

```
void CSomeClass::operator delete (void* p) {
    if (HeapFree(s_hHeap, 0, p)) {
        // объект удален успешно
        s_uNumAllocsInHeap--;
    }

    if (s_uNumAllocsInHeap == 0) {
        // если в куче больше нет объектов, уничтожаем ее
        if (HeapDestroy(s_hHeap)) {
            // описатель кучи приравниваем NULL, чтобы оператор new
            // мог создать новую кучу при создании нового объекта CSomeClass
            s_hHeap = NULL;}
        }
    }
}
```

Оператор *delete* принимает только один параметр: адрес удаляемого объекта. Сначала он вызывает *HeapFree* и передает ей описатель кучи и адрес высвобождаемого объекта. Если объект освобожден успешно, *s_uNumAltocsInHeap* уменьшается, показывая, что одним объектом CSomeClass в куче стало меньше. Далее оператор проверяет: не равна ли эта переменная 0, и, если да, вызывает *HeapDestroy*, передавая ей описатель кучи. Если куча уничтожена, *s_hHeap* присваивается NULL. Это важно: ведь в будущем наша программа может попытаться создать другой объект CSomeClass. При этом будет вызван оператор *new*, который проверит значение *s_hHeap*, чтобы определить, нужно ли использовать существующую кучу или создать новую.

Данный пример иллюстрирует очень удобную схему работы с несколькими кучами. Этот код легко подстроить и включить в ваши классы. Но сначала, может быть, стоит поразмыслить над проблемой наследования. Если при создании нового класса вы используете класс CSomeClass как базовый, то производный класс унаследует операторы *new* и *delete*, принадлежащие классу CSomeClass. Новый класс унаследует и его кучу, а это значит, что применение оператора *new* к производному классу повлечет выделение памяти для объекта этого класса из той же кучи, которую использует и класс CSomeClass. Хорошо это или нет, зависит от конкретной ситуации. Если

объекты сильно различаются размерами, это может привести к фрагментации кучи, что затруднит выявление таких ошибок в коде, о которых я рассказывал в разделах «Защита компонентов» и «Более эффективное управление памятью».

Если вы хотите использовать отдельную кучу для производных классов, нужно продублировать все, что я сделал для класса CSomeClass. А конкретнее — включить еще один набор переменных *s_hHeap* и *s_uNumAllocsInHeap* и повторить еще раз код для операторов *new* и *delete*. Компилятор увидит, что вы переопределили в производном классе операторы *new* и *delete*, и сформирует обращение именно к ним, а не к тем, которые содержатся в базовом классе.

Если вы не будете создавать отдельные кучи для каждого класса, то получите единственное преимущество: вам не придется выделять память под каждую кучу и соответствующие заголовки. Но кучи и заголовки не занимают значительных объемов памяти, так что даже это преимущество весьма сомнительно. Неплохо, конечно, если каждый класс, используя свою кучу, в то же время имеет доступ к куче базового класса. Но делать так стоит лишь после полной отладки приложения. И, кстати, проблему фрагментации куч это не снимает.

Другие функции управления кучами

Кроме уже упомянутых, в Windows есть еще несколько функций, предназначенных для управления кучами. В этом разделе я вкратце расскажу о них.

ToolHelp-функции (упомянутые в конце главы 4) дают возможность перечислять кучи процесса, а также выделенные внутри них блоки памяти. За более подробной информацией я отсылаю вас к документации Platform SDK: ищите разделы по функциям *Heap32First*, *Heap32Next*, *Heap32ListFirst* и *Heap32ListNext*.

В адресном пространстве процесса может быть несколько куч, и функция *GetProcessHeaps* позволяет получить их описатели «одним махом»:

```
DWORD GetProcessHeaps(
    DWORD dwNumHeaps,
    PHANDLE phHeaps);
```

Предварительно вы должны создать массив описателей, а затем вызвать функцию так, как показано ниже.

```
HANDLE hHeaps[25];
DWORD dwHeaps = GetProcessHeaps(25, hHeaps);
if (dwHeaps > 25) {
    // У процесса больше куч, чем мы ожидали
} else {
    // элементы от hHeaps[0] до hHeaps[dwHeaps - 1]
    // идентифицируют существующие кучи
}
```

Имейте в виду, что описатель стандартной кучи процесса тоже включается в этот массив описателей, возвращаемый функцией *GetProcessHeaps*. Целостность кучи позволяет проверить функция *HeapValidate*:

```
BOOL HeapValidate(
    HANDLE hHeap,
    DWORD fdwFlags,
    LPCVOID pvMem);
```

Обычно ее вызывают, передавая в *hHeap* описатель кучи, в *fdwFlags* — 0 (этот параметр допускает еще флаг HEAP_NO_SERIALIZE), а в *pvMem* — NULL. Функция просматривает все блоки в куче, чтобы убедиться в отсутствии поврежденных блоков. Чтобы она работала быстрее, в параметре *pvMem* можно передать адрес конкретного блока. Тогда функция проверит только этот блок.

Для объединения свободных блоков в куче, а также для возврата системе любых страниц памяти, на которых нет выделенных блоков, предназначена функция *HeapCompact*:

```
UINT HeapCompact( HANDLE hHeap, DWORD fdwFlags);
```

Обычно в параметре *fdwFhgs* передают 0, но можно передать и HEAP_NO_SERIALIZE.

Следующие две функции — *HeapLock* и *HeapUnbck* — используются парно:

```
B00L HeapLock(HANDLE hHeap);
B00L HeapUnlock(HANDLE hHeap);
```

Они предназначены для синхронизации потоков. После успешного вызова *HeapLock* поток, который вызывал эту функцию, становится владельцем указанной кучи. Если другой поток обращается к этой куче, указывая тот же описатель кучи, система приостанавливает его выполнение до тех пор, пока куча не будет разблокирована вызовом *HeapUnlock*.

Функции *HeapAlloc*, *HeapSize*, *HeapFree* и другие — все обращаются к *HeapLock* и *HeapUnlock*, чтобы обеспечить последовательный доступ к куче. Самостоятельно вызывать эти функции вам вряд ли понадобится.

Последняя функция, предназначенная для работы с кучами, — *HeapWalk*:

```
B00L HeapWalk(
    HANDLE hHeap,
    PPROCESS_HEAP_ENTRY pHeapEntry);
```

Она предназначена только для отладки и позволяет просматривать содержимое кучи. Обычно ее вызывают по несколько раз, передавая адрес структуры PROCESS_HEAP__ENTRY (Вы должны сами создать ее экземпляр и инициализировать):

```
typedef struct _PROCESS_HEAP_ENTRY {
    PVOID lpData;
    DWORD cbData;
    BYTE cbOverhead;
    BYTE iRegionIndex;
    WORD wFlags;
    union {
        struct {
            HANDLE hMem;
            DWORD dwReserved[ 3 ];
        } Block;
        struct {
            DWORD dwCommittedSize;
            DWORD dwUnCommittedSize;
            LPVOID lpFirstBlock;
            LPVOID lpLastBlock;
        } Region;
    };
} PROCESS_HEAP_ENTRY, * LPPROCESS_HEAP_ENTRY, *PPROCESS_HEAP_ENTRY;
```

Прежде чем перечислять блоки в куче, присвойте NULL элементу *lpData*, и это заставит функцию *HeapWalk* инициализировать все элементы структуры. Чтобы перейти к следующему блоку, вызовите *HeapWalk* еще раз, передав ей тот же описатель кучи и адрес той же структуры PROCESS_HEAP_ENTRY. Если *HeapWalk* вернет FALSE, значит, блоков в куче больше нет. Подробное описание элементов структуры PROCESS_HEAP_ENTRY см. в документации Platform SDK.

Обычно вызовы функции *HeapWalk* «обрамляют» вызовами *HeapLock* и *HeapUnlock*, чтобы посторонние потоки не портили картину, создавая или удаляя блоки в просматриваемой куче.

ЧАСТЬ IV

ДИНАМИЧЕСКИ ПОДКЛЮЧАЕМЫЕ БИБЛИОТЕКИ

Оглавление

ГЛАВА 19

DLL: основы

Динамически подключаемые библиотеки (dynamic-link libraries, DLL) — краеугольный камень операционной системы Windows, начиная с самой первой ее версии. В DLL содержатся все функции Windows API. Три самые важные DLL: Kernel32.dll (управление памятью, процессами и потоками), User32.dll (поддержка пользовательского интерфейса, в том числе функции, связанные с созданием окон и передачей сообщений) и GDI32.dll (графика и вывод текста).

В Windows есть и другие DLL, функции которых предназначены для более специализированных задач. Например, в AdvAPI32.dll содержатся функции для защиты объектов, работы с реестром и регистрации событий, в ComDlg32.dll — стандартные диалоговые окна (вроде File Open и File Save), а ComCtl32.dll поддерживает стандартные элементы управления.

В этой главе я расскажу, как создавать DLL-модули в ваших приложениях. Вот лишь некоторые из причин, по которым нужно применять DLL:

- **Расширение функциональности приложения.** DLL можно загружать в адресное пространство процесса динамически, что позволяет приложению, определив, какие действия от него требуются, подгружать нужный код. Поэтому одна компания, создав какое-то приложение, может предусмотреть расширение его функциональности за счет DLL от других компаний.
- **Более простое управление проектом.** Если в процессе разработки программного продукта отдельные его модули создаются разными группами, то при использовании DLL таким проектом управлять гораздо проще. Однако конечная версия приложения должна включать как можно меньше файлов. (Знал я одну компанию, которая поставляла свой продукт с сотней DLL. Их приложение запускалось ужасающе долго — перед началом работы ему приходилось открывать сотню файлов на диске.)
- **Экономия памяти.** Если одну и ту же DLL использует несколько приложений, в оперативной памяти может храниться только один ее экземп-

ляр, доступный этим приложениям. Пример — DLL-версия библиотеки C/C++. Ею пользуются многие приложения. Если всех их скомпоновать со статически подключаемой версией этой библиотеки, то код таких функций, как *sprintf*, *strcpy*, *malloc* и др., будет многократно дублироваться в памяти. Но если они компонуются с DLL-версией библиотеки C/C++, в памяти будет присутствовать лишь одна копия кода этих функций, что позволит гораздо эффективнее использовать оперативную память.

- **Разделение ресурсов.** DLL могут содержать такие ресурсы, как шаблоны диалоговых окон, строки, значки и битовые карты (растровые изображения). Эти ресурсы доступны любым программам.
- **Упрощение локализации.** DLL нередко применяются для локализации приложений. Например, приложение, содержащее только код без всяких компонентов пользовательского интерфейса, может загружать DLL с компонентами локализованного интерфейса.
- **Решение проблем, связанных с особенностями различных платформ.** В разных версиях Windows содержатся разные наборы функций. Зачастую разработчикам нужны новые функции, существующие в той версии системы, которой они пользуются. Если ваша версия Windows не поддерживает эти функции, вам не удастся запустить такое приложение: загрузчик попросту откажется его запускать. Но если эти функции будут находиться в отдельной DLL, вы загрузите программу даже в более ранних версиях Windows, хотя воспользоваться ими вы все равно не сможете.
- **Реализация специфических возможностей.** Определенная функциональность в Windows доступна только при использовании DLL. Например, отдельные виды ловушек (устанавливаемых вызовом *SetWindowsHookEx* и *SetWinEventHook*) можно задействовать при том условии, что функция уведомления ловушки размещена в DLL. Кроме того, расширение функциональности оболочки Windows возможно лишь за счет создания COM-объектов, существование которых допустимо только в DLL. Это же относится и к загружаемым веб-браузером ActiveX-элементам, позволяющим создавать веб-страницы с более богатой функциональностью.

DLL и адресное пространство процесса

Зачастую создать DLL проще, чем приложение, потому что она является лишь набором автономных функций, пригодных для использования любой программой, причем в DLL обычно нет кода, предназначенного для обработки циклов выборки сообщений или создания окон. DLL представляет собой набор модулей исходного кода, в каждом из которых содержится определенное число функций, вызываемых приложением (исполняемым файлом) или другими DLL. Файлы с исходным кодом компилируются и компонуются так же, как и при создании EXE-файла. Но, создавая DLL, вы должны указывать компоновщику ключ /DLL. Тогда компоновщик записывает в конеч-

ный файл информацию, по которой загрузчик операционной системы определяет, что данный файл — DLL, а не приложение.

Чтобы приложение (или другая DLL) могло вызывать функции, содержащиеся в DLL, образ ее файла нужно сначала спроецировать на адресное пространство вызывающего процесса. Это достигается либо за счет неявного связывания при загрузке, либо за счет явного — в период выполнения. Подробнее о неявном связывании мы поговорим чуть позже, а о явном — в главе 20.

Как только DLL спроецирована на адресное пространство вызывающего процесса, ее функции доступны всем потокам этого процесса. Фактически библиотеки при этом теряют почти всю индивидуальность: для потоков код и данные DLL — просто дополнительные код и данные, оказавшиеся в адресном пространстве процесса. Когда поток вызывает из DLL какую-то функцию, та считывает свои параметры из стека потока и размещает в этом стеке собственные локальные переменные. Кроме того, любые созданные кодом DLL объекты принадлежат вызывающему потоку или процессу — DLL ничем не владеет.

Например, если DLL-функция вызывает *VirtualAlloc*, резервируется регион в адресном пространстве того процесса, которому принадлежит поток, обратившийся к DLL-функции. Если DLL будет выгружена из адресного пространства процесса, зарезервированный регион не освободится, так как система не фиксирует того, что регион зарезервирован DLL-функцией. Считается, что он принадлежит процессу и поэтому освободится, только если поток этого процесса вызовет *VirtualFree* или завершится сам процесс.

Вы уже знаете, что глобальные и статические переменные EXE-файла не разделяются его параллельно выполняемыми экземплярами. В Windows 2000 — с помощью механизма копирования при записи, рассмотренного в главе 13. Глобальные и статические переменные DLL обрабатываются точно так же. Когда какой-то процесс проецирует образ DLL-файла на свое адресное пространство, система создает также экземпляры глобальных и статических переменных.

> **Примечание** Важно понимать, что единое адресное пространство состоит из одного исполняемого модуля и нескольких DLL-модулей. Одни из них могут быть скомпонованы со статически подключаемой библиотекой C/C++, другие — с DLL-версией той же библиотеки, а третьи (написанные не на C/C++) вообще ею не пользуются. Многие разработчики допускают ошибку, забывая, что в одном адресном пространстве может одновременно находиться несколько библиотек C/C++. Взгляните на этот код:

```
VOID EXEFunc() {
    PVOID pv = DLLFunc();
    // обращаемся к памяти, на которую указывает pv;
    // предполагаем,  что pv находится в C/C++-куче EXE-файла
    free(pv);
}
```

```
PVOID DLLFunc() {
    // выделяем блок в C/C++-куче DLL
    return(malloc(100));
}
```

Ну и что вы думаете? Будет ли этот код правильно работать? Освободит ли EXE-функция блок, выделенный DLL-функцией? Ответы на все вопросы одинаковы: может быть. Для точных ответов информации слишком мало. Если оба модуля (EXE и DLL) скомпонованы с DLL-версией библиотеки C/C++, код будет работать совершенно нормально. Но если хотя бы один из модулей связан со статической библиотекой C/C++, вызов *free* окажется неудачным. Я не раз видел, как разработчики обжигались на подобном коде.
На самом деле проблема решается очень просто: если в модуле есть функция, выделяющая память, в нем обязательно должна быть и противоположная функция, которая освобождает память. Давайте-ка перепишем предыдущий код так:

```
VOID EXEFunc() {
    PVOID pv = DLLFunc();
    // обращаемся к памяти, на которую указывает pv;
    // не делаем никаких предположений по поводу C/C++-кучи
    DLLFreeFunc(pv);
}

PVOID DLLFunc() {
    // выделяем блок в C/C++-куче DLL
    PVOID pv = malloc(100);
    return(pv);
}

BOOL DLLFreeFunc(PVOID pv) {
    // освобождаем блок,  выделенный в C/C++-куче DLL
    return(free(pv));
}
```

Этот код будет работать при любых обстоятельствах. Создавая свой модуль, не забывайте, что функции других модулей могут быть написаны на других языках, а значит, и ничего не знать о *malloc и free*. Не стройте свой код на подобных допущениях. Кстати, то же относится и к C++ операторам *new* и *delete*, реализованным с использованием *malloc и free*.

Общая картина

Попробуем разобраться в том, как работают DLL и как они используются вами и системой. Начнем с общей картины (рис. 19-1).

Для начала рассмотрим неявное связывание EXE- и DLL-модулей. *Неявное связывание* (implicit linking) — самый распространенный на сего-

дняшний день метод. (Windows поддерживает и явное связывание, но об этом — в главе 20.)

Сборка DLL-модуля	**Сборка EXE-модуля**
1) Заголовок с экспортируемыми прототипами, структурами и идентификаторами. 2) Файлы с исходным кодом на C/C++ с реализацией экспортируемых функций и переменных. 3) Компилятор генерирует .obj-файл для каждого из файлов из файлов с кодом на C/C++. 4) Компоновщик генерирует DLL из .obj-файла. 5) Компоновщик также генерирует .lib-файл, ссылки на импортированные функции и переменные, если найдена хотя бы одна экспортируемая функция или переменная.	6) Заголовок с экспортируемыми прототипами, структурами и идентификаторами. 7) Файлы с исходным кодом на C/C++, ссылающимся на импортированные функции и переменные. 8) Компилятор генерирует .obj-файл для каждого из файлов из файлов с кодом на C/C++. 9) Компоновщик комбинирует .obj-модули и разрешает ссылки на импортированные функции и переменные, используя .lib-файл. В результате получается .exe-файл (с таблицей импортированных функций — списком необходимых DLL и идентификаторов).

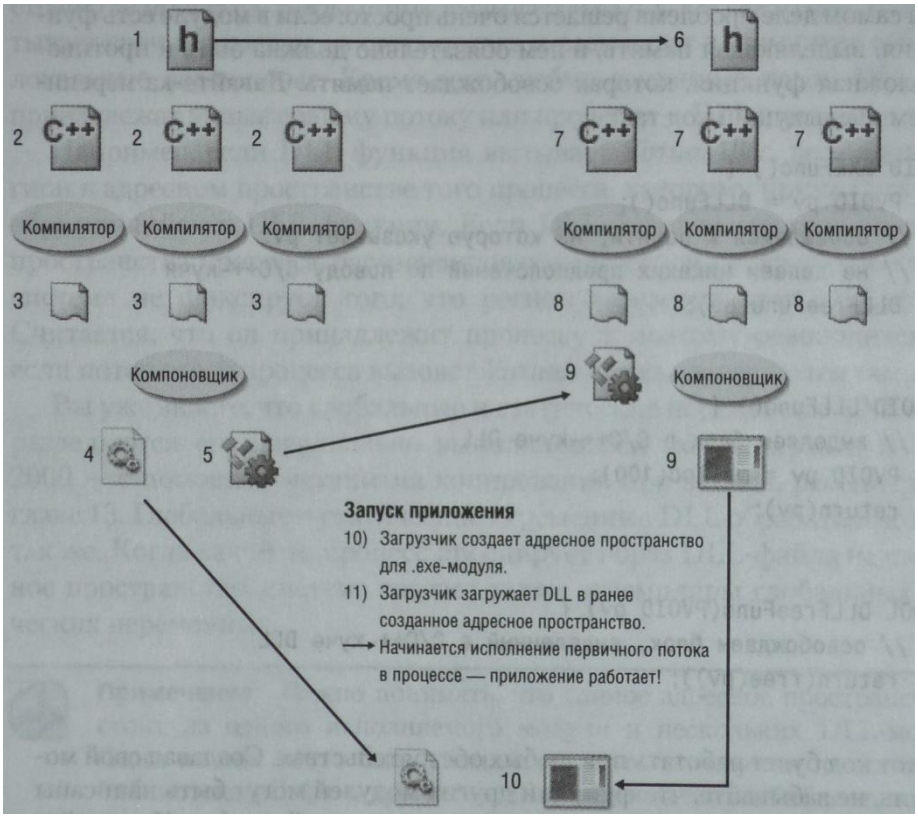


Рис. 19-1. *Так DLL создается и неявно связывается с приложением*

Как видно на рис. 19-1, когда некий модуль (например, EXE) обращается к функциям и переменным, находящимся в DLL, в этом процессе участвует несколько файлов и компонентов. Для упрощения будем считать, что исполняемый модуль (EXE) импортирует функции и переменные из DLL, а DLL-модули, наоборот, экспортируют их в исполняемый модуль. Но учтите, что DLL может (и это не редкость) импортировать функции и переменные из других DLL.

Собирая исполняемый модуль, который импортирует функции и переменные из DLL, вы должны сначала создать эту DLL. А для этого нужно следующее.

1. Прежде всего вы должны подготовить заголовочный файл с прототипами функции, структурами и идентификаторами, экспортируемыми из DLL. Этот файл включается в исходный код всех модулей вашей DLL. Как вы потом увидите, этот же файл понадобится и при сборке исполняемого модуля (или модулей), который использует функции и переменные из вашей DLL.
2. Вы пишете на C/C++ модуль (или модули) исходного кода с телами функций и определениями переменных, которые должны находиться в DLL. Так как эти модули исходного кода не нужны для сборки исполняемого модуля, они могут остаться коммерческой тайной компании-разработчика
3. Компилятор преобразует исходный код модулей DLL в OBJ-файлы (по одному на каждый модуль).
4. Компоновщик собирает все OBJ-модули в единый загрузочный DLL-модуль, в который в конечном итоге помещаются двоичный код и переменные (глобальные и статические), относящиеся к данной DLL. Этот файл потребуется при компиляции исполняемого модуля.
5. Если компоновщик обнаружит, что DLL экспортирует хотя бы одну переменную или функцию, то создаст и LIB-файл. Этот файл совсем крошечный, поскольку в нем нет ничего, кроме списка символьных имен функций и переменных, экспортируемых из DLL. Этот LIB-файл тоже понадобится при компиляции EXE-файла.

   Создав DLL, можно перейти к сборке исполняемого модуля.
6. Во все модули исходного кода, где есть ссылки на внешние функции, переменные, структуры данных или идентификаторы, надо включить заголовочный файл, предоставленный разработчиком DLL.
7. Вы пишете на C/C++ модуль (или модули) исходного кода с телами функций и определениями переменных, которые должны находиться в EXE-файле. Естественно, ничто не мешает вам ссылаться на функции и переменные, определенные в заголовочном файле DLL-модуля.
8. Компилятор преобразует исходный код модулей EXE в OBJ-файлы (по одному на каждый модуль).
9. Компоновщик собирает все OBJ-модули в единый загрузочный EXE-мо-дуль, в который в конечном итоге помешаются двоичный код и переменные (глобальные и статические), относящиеся к данному EXE. В нем так же создается раздел импорта, где перечисляются имена всех необходимых DLL-моделей (информацию о разделах см. в главе 17). Кроме того, для каждой DLL в этом разделе указывается, на какие символьные имена функций и переменных ссылается двоичный код исполняемого файла. Эти сведения потребуются загрузчику операционной системы, а как именно он ими пользуется — мы узнаем чуть позже.

Создав DLL- и EXE-модули, приложение можно запустить. При его запуске загрузчик операционной системы выполняет следующие операции.

10. Загрузчик операционной системы создает виртуальное адресное пространство для нового процесса и проецирует на него исполняемый модуль.
11. Далее загрузчик анализирует раздел импорта, находит все необходимые DLL-модули и тоже проецирует на адресное пространство процесса. Заметьте, что DLL может импортировать функции и переменные их другой DLL, а значит, у нее может быть собственный раздел импорта. Заканчивая подготовку процесса к работе, загрузчик просматривает раздел импорта каждого модуля и проецирует все требуемые DLL-модули на адресное пространство этого процесса. Как видите, на инициализацию процесса может уйти довольно длительное время. После отображения EXE- и всех DLL-модулей на адресное пространство процесса его первичный поток готов к выполнению, и приложение может начать работу. Далее мы подробно рассмотрим, как именно это происходит.

Создание DLL-модуля

Создавая DLL, вы создаете набор функций, которые могут быть вызваны из EXE-модуля (или другой DLL). DLL может экспортировать переменные, функции или C++-классы в другие модули. На самом деле я бы не советовал экспортировать переменные, потому что это снижает уровень абстрагирования вашего кода и усложняет его поддержку. Кроме того, C++-классы можно экспортировать, только если импортирующие их модули транслируются тем же компилятором. Так что избегайте экспорта C++-классов, если вы не уверены, что разработчики EXE-модулей будут пользоваться тем же компилятором.

При разработке DLL вы сначала создаете заголовочный файл, в котором содержатся экспортируемые из нее переменные (типы и имена) и функции (прототипы и имена). В этом же файле надо определить все идентификаторы и структуры данных, используемые экспортируемыми функциями и переменными. Заголовочный файл включается во все модули исходного кода вашей DLL. Более того, вы должны поставлять его вместе со своей DLL, чтобы другие разработчики могли включать его в свои модули исходного кода, которые импортируют ваши функции или переменные. Единый заголовочный файл, используемый при сборке DLL и любых исполняемых модулей, существенно облегчает поддержку приложения.

Вот пример единого заголовочного файла, включаемого в исходный код DLL- и EXE-модулей.

```
/************************************************************
Module: MyLlb.h
/************************************************************

#ifdef MYLIBAPI

// MYLIBAPI должен быть определен во всех модулях исходного кода DLL
// до включения этого файла

// здесь размещаются все экспортируемые функции и переменные

#else

// этот заголовочный файл включается в исходный код EXE-файла;
// указываем, что все функции и переменные импортируются
#define MYLIBAPI extern "C" __declspec(dllimport)

#endif

////////////////////////////////////////////////////////////

// здесь определяются все структуры данных и идентификаторы (символы)

////////////////////////////////////////////////////////////

// Здесь определяются экспортируемые переменные.
// Примечание:  избегайте экспорта переменных.
MYLIBAPI int g_nResult;

////////////////////////////////////////////////////////////

// здесь определяются прототипы экспортируемых функций
MYLIBAPI int Add(int nLeft,  int nRight);
////////////////// End of File ////////////////////////////
```

Этот заголовочный файл надо включать в самое начало исходных файлов вашей DLL следующим образом.

```
/************************************************************
Module: MyLibFilel.cpp
************************************************************/

// сюда включаются стандартные заголовочные файлы Windows и библиотеки C
#include <windows.h>

// этот файл исходного кода DLL экспортирует функции и переменные
#define MYLIBAPI extern "C" __declspec(dllexport)

// включаем экспортируемые структуры данных, идентификаторы, функции
// и переменные #include "MyLib.h"

////////////////////////////////////////////////////////////////////

// здесь размещается исходный код этой DLL
int g_nResult;

int Add(int nLeft,  int nRight) {
    g_nResult = nLeft + nRight;
    return(g_nResult);
}

////////////////////////////// End of File ////////////////////////////
```

При компиляции исходного файла DLL, показанного на предыдущем листинге, MYLIBAPI определяется как__*declspec(dllexport)* до включения заголовочного файла MyLib.h. Такой модификатор означает, что данная переменная, функция или C++-класс экспортируется из DLL. Заметьте, что идентификатор MYLIBAPI помещен в заголовочный файл до определения экспортируемой переменной или функции.

Также обратите внимание, что в файле MyLibFilel.cpp перед экспортируемой переменной или функцией не ставится идентификатор MYLIBAPI. Он здесь не нужен: проанализировав заголовочный файл, компилятор запоминает, какие переменные и функции являются экспортируемыми.

Идентификатор MYLIBAPI включает *extern.* Пользуйтесь этим модификатором только в коде на C++, но ни в коем случае не в коде на стандартном C. Обычно компиляторы C++ искажают (mangle) имена функций и переменных, что может приводить к серьезным ошибкам при компоновке. Представьте, что DLL написана на C++, а исполняемый код — на стандартном C. При сборке DLL имя функции будет искажено, но при сборке исполняемого модуля — нет. Пытаясь скомпоновать исполняемый модуль, компоновщик сообщит об ошибке: исполняемый модуль обращается к несуществующему идентификатору. Модификатор *extern* не дает компилятору иска-

жать имена переменных или функций, и они становятся доступными исполняемым модулям, написанным на C, C++ или любом другом языке программирования.

Теперь вы знаете, как используется заголовочный файл в исходных файлах DLL. А как насчет исходных файлов EXE-модуля? В них MYLIBAPI определять не надо: включая заголовочный файл, вы определяете этот идентификатор как *_declspec(dllimport),* и при компиляции исходного кода EXE-модуля компилятор поймет, что переменные и функции импортируются из DLL.

Просмотрев стандартные заголовочные файлы Windows (например, WinBase.h), вы обнаружите, что практически тот же подход исповедует и Майкрософт.

Что такое экспорт

В предыдущем разделе я упомянул о модификаторе *__declspec(dllexport).* Если он указан перед переменной, прототипом функции или C++-классом, компилятор Microsoft C/C++ встраивает в конечный OBJ-файл дополнительную информацию. Она понадобится компоновщику при сборке DLL из OBJ-файлов.

Обнаружив такую информацию, компоновщик создает LIB-файл со списком идентификаторов, экспортируемых из DLL. Этот LIB-файл нужен при сборке любого EXE-модуля, ссылающегося на такие идентификаторы. Компоновщик также вставляет в конечный DLL-файл таблицу экспортируемых идентификаторов — *раздел экспорта,* в котором содержится список (в алфавитном порядке) идентификаторов экспортируемых функций, переменных и классов. Туда же помещается *относительный виртуальный адрес* (relative virtual address, RVA) каждого идентификатора внутри DLL-модуля.

Воспользовавшись утилитой DumpBin.exe (с ключом *-exports)* из состава Microsoft Visual Studio, мы можем увидеть содержимое раздела экспорта в DLL-модуле. Вот лишь небольшой фрагмент такого раздела для Kernel32.dll:

```
C:\Windows\System32>DUMPBIN -exports Kernel32.DLL

Microsoft (R) COFF/PE Dumper Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file Kernel32.DLL

File Type: DLL

    Section contains the following exports for KERNEL32.dll

        00000000 characteristics
        4549AD66 time date stamp Thu Nov 02 09:33:42 2006
            0.00 version
```

```
            1 ordinal base
         1207 number of functions
         1207 number of names

ordinal  hint       RVA  name
      3     0            AcquireSRWLockExclusive (forwarded to
                         NTDLL.RtlAcquireSRWLockExclusive)
      4     1            AcquireSRWLockShared (forwarded to
                         NTDLL.RtlAcquireSRWLockShared)
      5     2  0002734D  ActivateActCtx = _ActivateActCtx@8
      6     3  000088E9  AddAtomA = _AddAtomA@4
      7     4  0001FD7D  AddAtomW = _AddAtomW@4
      8     5  000A30AF  AddConsoleAliasA = _AddConsoleAliasA@12
      9     6  000A306E  AddConsoleAliasW = _AddConsoleAliasW@12
     10     7  00087935  AddLocalAlternateComputerNameA =
                         _AddLocalAlternateComputerNameA@8
     11     8  0008784E  AddLocalAlternateComputerNameW =
                         _AddLocalAlternateComputerNameW@8
     12     9  00026159  AddRefActCtx = _AddRefActCtx@4
     13     A  00094456  AddSIDToBoundaryDescriptor =
                         _AddSIDToBoundaryDescriptor@8
    ...
   1205   4B4  0004328A  lstrlen  = _lstrlenA@4
   1206   4B5  0004328A  lstrlenA = _lstrlenA@4
   1207   4B6  00049D35  lstrlenW = _lstrlenW@4

Summary
        3000 .data
        A000 .reloc
        1000 .rsrc
       C9000 .text
```

Как видите, идентификаторы расположены по алфавиту; в графе RVA указывается смещение в образе DLL-файла, по которому можно найти экспортируемый идентификатор. Значения в графе ordinal предназначены для обратной совместимости с исходным кодом, написанным для 16-разрядной Windows, — применять их в современных приложениях не следует. Данные из графы hint используются системой и для нас интереса не представляют.

> **Примечание.** Многие разработчики — особенно те, у кого большой опыт программирования для 16-разрядной Windows, — привыкли экспортировать функции из DLL, присваивая им порядковые номера. Но Майкрософт не публикует такую информацию по системным DLL и требует связывать EXE- или DLL-файлы с Windows-функциями только

по именам. Используя порядковый номер, вы рискуете тем, что ваша программа не будет работать в других версиях Windows. Я поинтересовался, почему Майкрософт отказывается от порядковых номеров, и получил такой ответ: «Мы (Майкрософт) считаем, что PE-формат позволяет сочетать преимущества порядковых номеров (быстрый поиск) с гибкостью импорта по именам. Учтите и то, что в любой момент в API могут появиться новые функции. А с порядковыми номерами в большом проекте работать очень трудно — тем более, что такие проекты многократно пересматриваются».
Работая с собственными DLL-модулями и связывая их со своими EXE-файлами, порядковые номера использовать вполне можно. Майкрософт гарантирует, что этот метод будет работоспособен даже в будущих версиях операционной системы. Но лично я стараюсь избегать порядковых номеров и отныне применяю при связывании только имена.

Создание DLL для использования с другими средствами разработки (отличными от Visual C++)

Если вы используете Visual C++ для сборки как DLL, так и обращающегося к ней EXE-файла, то все сказанное ранее справедливо, и вы можете спокойно пропустить этот раздел. Но если вы создаете DLL на Visual C++, а EXE-файл — с помощью средств разработки от других поставщиков, вам не миновать дополнительной работы.

Я уже упоминал о том, как применять модификатор *extern* при «смешанном» программировании на C и C++. Кроме того, я говорил, что из-за искажения имен нужно применять один и тот же компилятор. Даже при программировании на стандартном C инструментальные средства от разных поставщиков создают проблемы. Дело в том, что компилятор Microsoft C, экспортируя C-функцию, искажает ее имя, даже если вы вообще не пользуетесь C++. Это происходит, только когда ваша функция экспортируется по соглашению *__stdcall.* (Увы, это самое популярное соглашение.) Тогда компилятор Microsoft искажает имя C-функции: впереди ставит знак подчеркивания, а к концу добавляет суффикс, состоящий из символа @ и числа байтов, передаваемых функции в качестве параметров. Например следующая функция экспортируется в таблицу экспорта DLL как *_MyFunc@8*:

```
__declspec(dllexport) LONG __stdcall MyFunc(int a, int b);
```

Если вы решите создать EXE-файл с помощью средств разработки от другого поставщика, то компоновщик попытается скомпилировать функцию *MyFunc,* которой нет в файле DLL, созданном компилятором Microsoft, и, естественно, произойдет ошибка.

Чтобы средствами Microsoft собрать DLL, способную работать с инструментарием от другого поставщика, нужно указать компилятору Microsoft экспортировать имя функции без искажений. Сделать это можно двумя спо-

собами. Первый — создать DEF-файл для вашего проекта и включить в него раздел EXPORTS так:

```
EXPORTS
    MyFunc
```

Компоновщик от Microsoft, анализируя этот DEF-файл, увидит, что экспортировать надо обе функции: *_MyFunc@8* и *MyFunc*. Поскольку их имена идентичны (не считая вышеописанных искажений), компоновщик на основе информации из DEF-файла экспортирует только функцию с именем *MyFunc*, а функцию *_MyFunc@8* не экспортирует вообще.

Может, вы подумали, что при сборке EXE-файла с такой DLL компоновщик от Microsoft, ожидая имя *_MyFunc@8*, не найдет вашу функцию? В таком случае вам будет приятно узнать, что компоновщик все сделает правильно и корректно скомпонует EXE-файл с функцией *MyFunc*.

Если вам не по душе DEF-файлы, можете экспортировать неискаженное имя функции еще одним способом. Добавьте в один из файлов исходного кода DLL такую строку:

```
#pragma comment(linker, "/export:MyFunc=_MyFunc@8")
```

Тогда компилятор потребует от компоновщика экспортировать функцию *MyFunc* с той же точкой входа, что и *_MyFunc@8*. Этот способ менее удобен, чем первый, так как здесь приходится самостоятельно вставлять дополнительную директиву с искаженным именем функции. И еще один минус этого способа в том, что из DLL экспортируется два идентификатора одной и той же функции: *MyFunc* и *_MyFunc@8*, тогда как при первом способе — только идентификатор *MyFunc*. По сути, второй способ не имеет особых преимуществ перед первым — он просто избавляет от DEF-файла.

Создание EXE-модуля

Вот пример исходного кода EXE-модуля, который импортирует идентификаторы, экспортируемые DLL, и ссылается на них в процессе выполнения.

```
/*****************************************************************
Module: MyExeFile1. cpp
*****************************************************************/

// сюда включаются стандартные заголовочные файлы Windows и библиотеки C
#include <windows.h>
#include <strsafe.h>
#include <stdlib.h>

// включаем экспортируемые структуры данных, идентификаторы, функции
// и переменные
#include "MyLib\MyLib.h"
```

```
//////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE,   HINSTANCE,   LPTSTR,  int) {

    int nleft = 10, nRight = 25;

    TCHAR sz[100];
    StringCchPrintf(sz, _countof(sz), TEXT("%d + %d = %d"),
       nLeft,  nRight, Add(nLeft,  nRight));
    MessageBox(NULL,  sz, TEXT("Calculation"),  MB_OK);

    StringCchPrintf(sz,  _countof(sz),
       TEXT(The result from the last Add is:  %d"), g_nResult);
    MessageBox(NULL,  sz,  TEXT("Last Result"), MB_OK);
    return(0);
}
////////////////////////////// End of File ///////////////////////////////
```

Создавая файлы исходного кода для EXE-модуля, вы должны включить в них заголовочный файл DLL, иначе импортируемые идентификаторы окажутся неопределенными, и компилятор выдаст массу предупреждений и сообщений об ошибках.

MYLIBAPI в исходных файлах EXE-модуля до заголовочного файла DLL не определяется. Поэтому при компиляции приведенного выше кода MYLIBAPI за счет заголовочного файла MyLib.h будет определен как *__declspec (dllimport)*. Встречая такой модификатор перед именем переменной, функции или C++-класса, компилятор понимает, что данный идентификатор импортируется из какого-то DLL-модуля. Из какого именно, ему не известно, да это его и не интересует. Компилятору нужно лишь убедиться в корректности обращения к импортируемым идентификаторам.

Далее компоновщик собирает все OBJ-модули в конечный EXE-модуль. Для этого он должен знать, в каких DLL содержатся импортируемые идентификаторы, на которые есть ссылки в коде. Информацию об этом он получает из передаваемого ему LIB-файла. Я уже говорил, что этот файл — просто список идентификаторов, экспортируемых DLL. Компоновщик должен удостовериться в существовании идентификатора, на который вы ссылаетесь в коде и узнать, в какой DLL он находится. Если компоновщик сможет разрешить все ссылки на внешние идентификаторы, на свет появится EXE-модуль.

Что такое импорт

В предыдущем разделе я упомянул о модификаторе *__declspec(dllimport)*. Импортируя идентификатор, необязательно прибегать к *__declspec(dllimport)* — можно использовать стандартное ключевое слово extern языка C. Но компилятор создаст чуть более эффективный код, если ему будет заранее известно,

что идентификатор, на который мы ссылаемся, импортируется из LIB-файла DLL-модуля. Вот почему я настоятельно рекомендую пользоваться ключевым словом *__declspec(dllimport)* для импортируемых функций и идентификаторов данных. Именно его подставляет за вас операционная система, когда вы вызываете любую из стандартных Windows-функций.

Разрешая ссылки на импортируемые идентификаторы, компоновщик создает в конечном EXE-модуле раздел *импорта* (imports section). В нем перечисляются DLL, необходимые этому модулю, и идентификаторы, на которые есть ссылки из всех используемых DLL.

Воспользовавшись утилитой DumpBin.exe (с ключом *-imports*), мы можем увидеть содержимое раздела импорта. Ниже показан фрагмент полученной с ее помощью таблицы импорта Calc.exe.

```
C:\Windows\System32>DUMPBIN -imports Calc.exe

Microsoft (R) COFF/PE Dumper Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file calc.exe

File Type:  EXECUTABLE IMAGE

    Section contains the following imports:

        SHELL32.dll
                     10010CC Import Address Table
                     1013208 Import Name Table
                    FFFFFFFF time date stamp
                    FFFFFFFF Index of first forwarder reference

        766EA0A5         110 ShellAboutW

        ADVAPI32.dll
                    1001000 Import Address Table
                    101313C Import Name Table
                   FFFFFFFF time date stamp
                   FFFFFFFF Index of first forwarder reference
             77CA8229  236 RegCreateKeyW
             77CC802D  278 RegSetValueExW
             77CD632E  268 RegQueryValueExW
             77CD64CC  22A RegCloseKey
...
    ntdll.dll
             1001250 Import Address Table
             101338C Import Name Table
FFFFFFFF time date stamp
```

```
FFFFFFFF Index of first forwarder reference

77F0850D       548 WinSqmAddToStream
KERNEL32.dll
    1001030 Import Address Table
    101316C Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference

77E01890   24F GetSystemTimeAsFileTime
77E47B0D   1AA GetCurrentProcessId
77E2AA46   170 GetCommandLineW
77E0918D   230 GetProfilelntW
…
Header contains the following bound import information:
Bound to SHELL32.dll [4549BDB4] Thu Nov 02 10:43:16 2006
Bound to ADVAPI32.dll [4549BCD2] Thu Nov 02 10:39:30 2006
Bound to 0LEAUT32.dll [4549BD95] Thu Nov 02 10:42:45 2006
Bound to ole32.dll [4549BD92] Thu Nov 02 10:42:42 2006
Bound to ntdll.dll [4549BDC9] Thu Nov 02 10:43:37 2006
Bound to KERNEL32.dll [4549BD80] Thu Nov 02 10:42:24 2006
Bound to GDI32.dll [4549BCD3] Thu Nov 02 10:39:31 2006
Bound to USER32.dll [4549BDE0] Thu Nov 02 10:44:00 2006
Bound to msvcrt.dll [4549BD61] Thu Nov 02 10:41:53 2006

Summary
    2000 .data
    2000 .reloc
    16000 .rsrc
13000 .text
```

Как видите, в разделе есть записи по каждой DLL необходимой Calc.exe: Shell32.dll, AdvAPI32.dll, OleAut32.dll, O1e32.dll, Ntdll.dll, Kernel32.dll, GDI32.dll, User32.dll и MSVCRXdll. Под именем DLL-модуля выводится список идентификаторов, импортируемых программой Calc.exe. Например, Calc.exe обращается к следующим функциям из Kernel32.dll: *GetCurrentProcessId*, *GetCommandLineW*, *GetProfileIntW* и др.

Число слева от импортируемого идентификатора называется «подсказкой» (hint) и для нас несущественно. Крайнее левое число в строке для идентификатора сообщает адрес, по которому он размещен в адресном пространстве процесса. Такой адрес показывается, только если было проведено связывание (binding) исполняемого модуля, но об этом — в главе 20.

Выполнение EXE-модуля

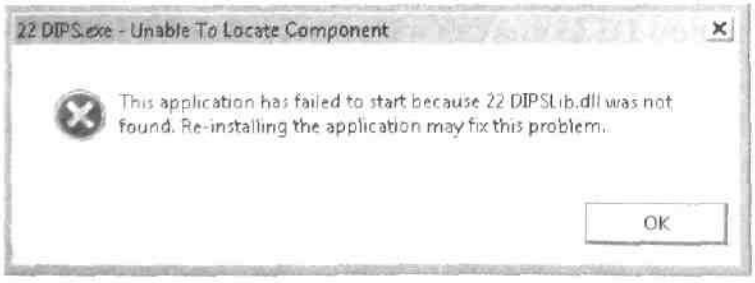
При запуске EXE-файла загрузчик операционной системы создает для его процесса виртуальное адресное пространство и проецирует на него исполняемый модуль. Далее загрузчик анализирует раздел импорта и пытается спроецировать все необходимые DLL на адресное пространство процесса. Поскольку в разделе импорта указано только имя DLL (без пути), загрузчику приходится самому искать ее на дисковых устройствах в компьютере пользователя. Поиск DLL осуществляется в следующей последовательности.

1. Каталог, содержащий EXE-файл.
2. Основной каталог Windows (по данным *GetWindowsDirectory*).
3. Системный каталог Windows (по данным *GetSystemDirectory*).
4. Текущий каталог процесса.
5. Каталоги, указанные в переменной окружения PATH.

Обратите внимание на то, что текущий каталог приложения просматривается после каталогов Windows. Это новшество, введенное в Windows XP SP2, должно помешать загрузке записанных злоумышленниками в текущей каталог приложения поддельных системных DLL вместо настоящих, расположенных в «законных» каталогах Windows. В электронной документации MSDN говорится, что DWORD-значение в разделе HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager позволяет изменять порядок просмотра каталогов, но лучше не трогайте его, чтобы вредоносные программы не смогли воспользоваться этой уязвимостью. Имейте в виду, что есть и другие факторы, влияющие на поиск DLL загрузчиком (подробнее см. в главе 20).

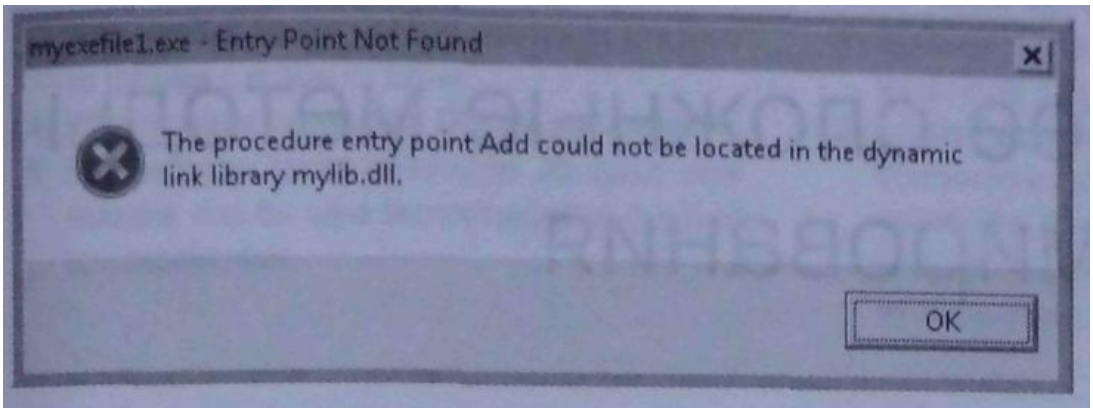
Учтите, что на процесс поиска библиотек могут повлиять и другие факторы. Проецируя DLL-модули на адресное пространство, загрузчик проверяет в каждом из них раздел импорта. Если у DLL есть раздел импорта (что обычно и бывает), загрузчик проецирует следующий DLL-модуль. При этом загрузчик ведет учет загружаемых DLL и проецирует их только один раз, даже если загрузки этих DLL требуют и другие модули.

Если найти файл DLL не удается, загрузчик выводит следующее сообщение.



Найдя и спроецировав на адресное пространство процесса все необходимые DLL-модули, загрузчик настраивает ссылки на импортируемые идеентификаторы. Для этого он вновь просматривает разделы импорта в каждом

модуле, проверяя наличие указанного идентификатора в соответствующей DLL. Не обнаружив его (что происходит крайне редко), загрузчик выводит следующее сообщение:



Если же идентификатор найден, загрузчик отыскивает его RVA и прибавляет к виртуальному адресу, по которому данная DLL размещена в адресном пространстве процесса, а затем сохраняет полученный виртуальный адрес в разделе импорта EXE-модуля. И с этого момента ссылка в коде на импортируемый идентификатор приводит к выборке его адреса из раздела импорта вызывающего модуля, открывая таким образом доступ к импортируемой переменной, функции или функции-члену C++-класса. Вот и все — динамические связи установлены, первичный поток процесса начал выполняться, и приложение наконец-то работает!

Естественно, загрузка всех этих DLL и настройка ссылок занимает какое-то время. Но, поскольку такие операции выполняются лишь при запуске процесса, на производительности приложения это не сказывается. Тем не менее, для многих программ подобная задержка при инициализации неприемлема. Чтобы сократить время загрузки приложения, вы должны модифицировать базовые адреса своих EXE- и DLL-модулей и провести их (модулей) связывание. Увы, лишь немногие разработчики знают, как это делается, хотя эти приемы очень важны. Если бы ими пользовались все компании-разработчики, система работала бы куда быстрее. Я даже считаю, что операционную систему нужно поставлять с утилитой, позволяющей автоматически выполнять эти операции. О модификации базовых адресов модулей и о связывании я расскажу в следующей главе.

Оглавление

ГЛАВА 20

DLL: более сложные методы программирования

В предыдущей главе мы говорили в основном о неявном связывании, поскольку это самый популярный метод. Представленной там информации вполне достаточно для создания большинства приложений. Однако DLL открывают нам гораздо больше возможностей, и в этой главе вас ждет целый «букет» новых методов, относящихся к программированию DLL. Во многих приложениях эти методы, скорее всего, не понадобятся, тем не менее они очень полезны, и познакомиться с ними стоит. Я бы посоветовал, как минимум, прочесть разделы «Модификация базовых адресов модулей» и «Связывание модулей»; подходы, изложенные в них, помогут существенно повысить быстродействие всей системы.

Явная загрузка DLL и связывание идентификаторов

Чтобы поток мог вызвать функцию из DLL-модуля, последний надо спроецировать на адресное пространство процесса, которому принадлежит этот поток. Делается это двумя способами. Первый состоит в том, что код вашего приложения просто ссылается на идентификаторы, содержащиеся в DLL, и тем самым заставляет загрузчик неявно загружать (и связывать) нужную DLL при запуске приложения.

Второй способ — явная загрузка и связывание требуемой DLL в период выполнения приложения. Иначе говоря, его поток явно загружает DLL в адресное пространство процесса, получает виртуальный адрес необходимой DLL-функции и вызывает ее по этому адресу. Изящество такого подхода в том, что все происходит в уже выполняемом приложении.

На рис. 20-1 показано, как приложение явно загружает DLL и связывается с ней.

Сборка DLL-модуля

1) Заголовок с экспортируемыми прототипами, структурами и идентификаторами.
2) Файлы с исходным кодом на C/C++ с реализацией экспортируемых функций и переменных.
3) Компилятор генерирует .obj-файл для каждого из файлов с кодом на C/C++.
4) Компоновщик генерирует DLL из .obj-файла.
5) Компоновщик также генерирует .lib-файл, если найдена хотя бы одна экспортируемая функция или переменная.

Сборка EXE-модуля

6) Заголовок с экспортируемыми прототипами, структурами и идентификаторами.
7) Файлы с исходным кодом на C/C++, ссылающимся на импортированные функции и переменные.
8) Компилятор генерирует .obj-файл для каждого из файлов с кодом на C/C++.
9) Компоновщик комбинирует .obj-модули и разрешает ссылки на импортированные функции и переменные, используя .lib-файл. В результате получается .exe-файл (с таблицей импортированных функций—списком необходимых DLL и идентификаторов).

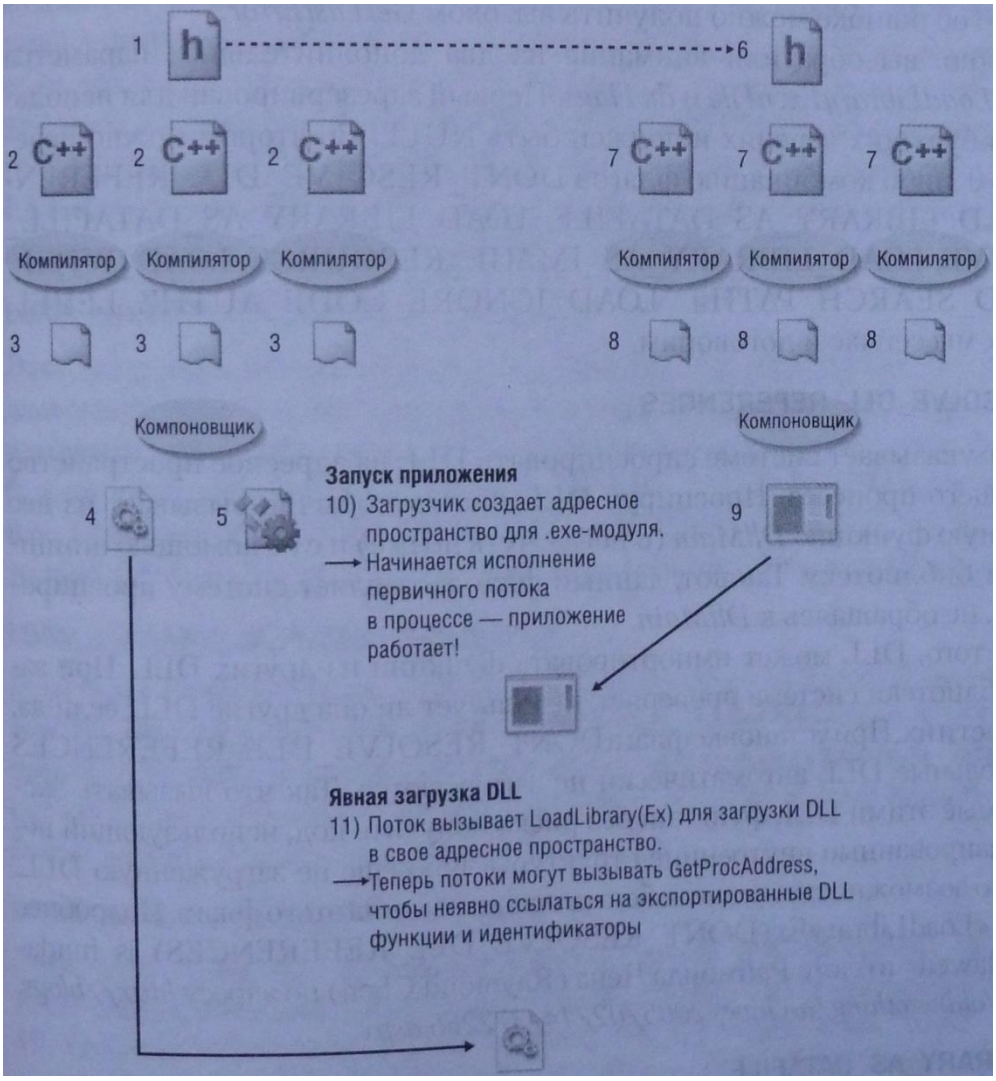


Рис. 20-1. *Так DLL создается и явно связывается с приложением*

Явная загрузка DLL

В любой момент поток может спроецировать DLL на адресное пространство процесса, вызвав одну из двух функций:

```
HMODULE LoadLibrary(PCTSTR pszDLLPathName);
```

```
HMOOULE LoadLibraryEx(
    PCTSTR pszDLLPathName,
    HANDLE hFile,
    DWORD dwFlags);
```

Обе функции ищут образ DLL-файла (в каталогах, список которых приведен в предыдущей главе) и пытаются спроецировать его на адресное пространство вызывающего процесса. Значение типа HINSTANCE, возвращаемое этими функциями, сообщает адрес виртуальной памяти, по которому спроецирован образ файла. Если спроецировать DLL на адресное пространство процесса не удалось, функции возвращают NULL. Дополнительную информацию об ошибке можно получить вызовом *GetLastError*.

Очевидно, вы обратили внимание на два дополнительных параметра функции *LoadLibraryEx*: *hFile* и *dwFlags*. Первый зарезервирован для использования в будущих версиях и должен быть NULL. Во втором можно передать либо 0, либо комбинацию флагов DONT_RESOLVE_DLL_REFERENCES, LOAD_LIBRARY_AS_DATAFILE, LOAD_LIBRARY_AS_DATAFILE_EXCLUSIVE, LOAD_LIBRARY_AS_IMAGE_RESOURCE, LOAD_WITH_ALTERED_SEARCH_PATH и LOAD_IGNORE_CODE_AUTHZ_LEVEL о которых мы сейчас и поговорим.

DONT_RESOLVE_DLL_REFERENCES

Этот флаг указывает системе спроецировать DLL на адресное пространство вызывающего процесса. Проецируя DLL. система обычно вызывает из нее специальную функцию *DllMain* (о ней — чуть позже) и с ее помощью инициализирует библиотеку. Так вот, данный флаг заставляет систему проецировать DLL, не обращаясь к *DllMain*.

Кроме того, DLL может импортировать функции из других DLL. При загрузке библиотеки система проверяет, использует ли она другие DLL; если да, то загружает и их. При установке флага DONT_RESOLVE_DLL_REFERENCES дополнительные DLL автоматически не загружаются. Так что вызывать экспортируемые этими DLL функции без риска получить код, использующий неинициализированные внутренние структуры либо еще не загруженную DLL Словом, по возможности лучше избегать использования этого флага. Подробнее см. пост «LoadLibraryEx (DONT_RESOLVE_DLL_REFERENCES) is fundamentally flawed» в блоге Рэймонда Чена (Raymond Chen) по адресу http://blogs.msdn.com/oldnewthing/archive/2005/02/14/372266.aspx.

LOAD_LIBRARY_AS_DATAFILE

Этот флаг очень похож на предыдущий. DLL проецируется на адресное пространство процесса так, будто это файл данных. При этом система не тратит дополнительное время на подготовку к выполнению какого-либо кода из данного файла. Например, когда DLL проецируется на адресное пространство, система считывает информацию из DLL-файла и на ее основе определяет, какие атрибуты защиты страниц следует присвоить разным частям фай-

ла. Если флаг LOAD_LIBRARY_AS_DATAFILE не указан, атрибуты защиты устанавливаются такими, чтобы кол из данного файла можно было выполнять. Например, если вызвать функцию *GetProcAddress* на DLL, загруженной с этим флагом, то *GetProcAddress* вернет NULL, а *GetLastError* — ERROR_MOD_NOT_FOUND.

Этот флаг может понадобиться по нескольким причинам. Во-первых, его стоит указать, если DLL содержит только ресурсы и никаких функций. Тогда DLL проецируется па адресное пространство процесса, после чего при вызове функций, загружающих ресурсы, можно использовать значение HMODULE, возвращенное функцией *LoadLibraryEx*. Во-вторых, он пригодится, если вам нужны ресурсы, содержащиеся в каком-нибудь EXE-файле. Обычно загрузка такого файла приводит к запуску нового процесса, но этого не произойдет, если его загрузить вызовом *LoadLibraryEx* в адресное пространство вашего процесса. Получив значение HMODULE/HINSTANCE для спроецированного EXE-файла, вы фактически получаете доступ к его ресурсам. Так как в EXE-файле нет *DllMain*, при вызове *LoadLibraryEx* для загрузки EXE-файла нужно указать флаг LOAD_LIBRARY_AS_DATAFILE.

LOAD_LIBRARY_AS_DATAFILE_EXCLUSIVE

Этот флаг всем похож на LOAD_LIBRARY_AS_DATAFILE, но открывает двоичный файл для монопольного доступа. Ни одно приложение не сможет изменить этот файл, пока ваша программа использует его. Этот флаг более безопасен по сравнению с LOAD_LIBRARY_AS_DATAFILE, поэтому лучше использовать LOAD_LIBRARY_AS_DATAFILE_EXCLUSIVE, если только другим программам не требуется возможность записи в ваш файл.

LOAD_LIBRARY_AS_IMAGE_RESOURCE

Этот флаг также похож на LOAD_LIBRARY_AS_DATAFILE, но отличается тем, что при загрузке DLL ее RVA (о RVA см. главу 19) изменяется операционной системой. Это позволяет непосредственно использовать *RVA*, не преобразуя их с учетом адреса, по которому DLL загружена в память. Это особенно удобно при анализе PE-разделов в DLL.

LOAD_WITH.ALTERED.SEARCH.PATH

Этот флаг изменяет алгоритм, используемый *LoadLibraryEx* при поиске DLL-файла. Обычно поиск осуществляется так, как я рассказывал в главе 19. Однако, если данный флаг установлен, функция ищет файл, в зависимости от значения переданного ей параметра *pszDLLPathName*, по одному из следующих алгоритмов.

1. Если pszDLLPathName не содержит символа \, поиск идет по алгоритму, описанному в главе 19.
2. Если в pszDLLPathName есть символ \, действия LoadLibraryEx зависят от того, какой путь ей передан: полный или относительный.

- Если функции передан полный путь либо сетевой путь (например, C:\Apps\Libraries\MyLibrary.dll или \\server\share\MyLibrary.dll), она пытается непосредственно загрузить заданный файл. Если этот файл не существует, LoadLibraryEx возвращает NULL, а GetLastError — ERROR_MOD_NOT_FOUND, на чем поиск и заканчивается.
- В противном случае путь поиска формируется конкатенацией значения pszDLLPathName с путями к перечисленным ниже папкам:
  - текущий каталог процесса;
  - системный каталог Windows;
  - каталог 16-разрядных системных компонентов, например Windows\System;
  - каталог Windows;
  - каталоги, указанные в переменной окружения PATH.

  Заметьте, что знаки «.» и «…» в значении параметра *pszDLLPathName* учитываются при построении относительного пути для каждого этапа поиска. Так, если передан параметр TEXT("..\\MyLibrary.dir), *LoadLibraryEx* будет искать MyLibrary.dll в следующих папках:
  - папки в текущем каталоге;
  - папки системного каталога Windows (т.е. каталога Windows);
  - каталог 16-разрядных компонентов Windows; Q корневой каталог тома;
  - родительские папки и каталоги, указанные в переменной окруже-ния PATH.

Поиск останавливается после загрузки искомой DLL

3. Если вы пишете программу, которая должна динамически загружать библиотеки из стандартных папок, вместо LoadLibraryEx с флагом LOAD_WITH_ALTERED_SEARCH_PATH вызывайте функцию SetDllDirectory, принимающую как параметр имя папки, в которой находится искомая библиотека. Эта функция использует LoadLibrary или LoadLibraryEx для поиска в следующих папках:
   - каталог приложения;
   - каталог, заданный с помощью SetDllDirectory;
   - системный каталог Windows (т.е. каталог «Windows»);
   - каталог 16-разрядных компонентов Windows;
   - корневой каталог тома;
   - родительские папки и каталоги, указанные в переменной окружения PATH.

Этот алгоритм позволяет хранить файлы приложения и общие DLL в стандартных каталогах, не опасаясь, что вместо них из каталога приложения будут загружены другие DLL с тем же именем, если в каталоге приложения есть ярлыки для таких DLL. Заметьте, что вызов *SetDllDirectory* с пустой

строкой — TEXT(“”) — удаляет текущий каталог из пути поиска. Если вместо этого передать NULL, будет восстановлен алгоритм поиска по умолчанию. И последнее: *GetDllDirectory* возвращает текущее значение данного конкретного каталога.

LOAD_IGNORE_CODE_AUTHZ_LEVEL

Этот флаг отключает защитный механизм WinSafer (известный также как *Software Restriction Policies* или *Safer*), появившийся в Windows XP. Этот механизм контролирует права доступа кода во время исполнения (подробнее см. по ссылке http://technet.microsoft.com/en-us/windowsvista/aa940985.aspx), Windows Vista этот механизм заменен User Account Control (UAC), о котором рассказывается в главе 4.

Явная выгрузка DLL

Если необходимость в DLL отпадает, ее можно выгрузить из адресного пространства процесса, вызвав функцию:

```
BOOL FreeLibrary(HMODULE hInstDll);
```

Вы должны передать в *FreeLibrary* значение типа HWSTANCE, которое идентифицирует выгружаемую DLL. Это значение вы получаете после вызова *LoadLibrary(Ex)*.

DLL можно выгрузить и с помощью другой функции:

```
V0ID FreeLibraryAndExitThread(
    HMODULE hInstDll,
    DWORD dwExitCode);
```

Она реализована в Kernel32.dll так:

```
VOID FreeLibraryAndExitThread(HMODULE hInstDll, DWORD dwExitCode) {
    FreeLibrary(hInstDll);
    ExitThread(dwExitCode);
}
```

На первый взгляд, в ней нет ничего особенного, и вы, наверное, удивляетесь, с чего это Майкрософт решила ее написать. Но представьте такой сценарий, вы пишете DLL, которая при первом отображении на адресное пространство процесса создает поток. Последний, закончив свою работу, отключает DLL от адресного пространства процесса и завершается, вызывая сначала *FreeLibrary*, а потом *ExitThread*.

Если поток станет сам вызывать *FreeLibrary* и *ExitThread*, возникнет очень серьезная проблема: *FreeLibrary* тут же отключит DLL от адресного пространства процесса. После возврата из *FreeLibrary* код, содержащий вызов *ExitThread*, окажется недоступен, и поток попытается выполнить неизвестно что. Это приведет к нарушению доступа и завершению всего процесса!

С другой стороны, если поток обратится к *FreeLibraryAndExitThread*, она вызовет *FreeLibrary*, и та сразу же отключит DLL. Но следующая исполняе-

мая инструкция находится в Kernel32.dll, а не в только что отключенной DLL. Значит, поток сможет продолжить выполнение и вызвать *ExitThread*, которая корректно завершит его, не возвращая управления.

На самом деле *LoadLibrary* и *LoadLibraryEx* лишь увеличивают счетчик числа пользователей указанной библиотеки, а *FreeLibrary* и *FreeLibraryAndExitThread* его уменьшают. Так, при первом вызове *LoadLibrary* для загрузки DLL система проецирует образ DLL-файла на адресное пространство вызывающего процесса и присваивает единицу счетчику числа пользователей этой DLL. Если поток того же процесса вызывает *LoadLibrary* для той же DLL еще раз, DLL больше не проецируется; система просто увеличивает счетчик числа ее пользователей — вот и все.

Чтобы выгрузить DLL из адресного пространства процесса, *FreeLibrary* придется теперь вызывать дважды: первый вызов уменьшит счетчик до 1, второй — до 0. Обнаружив, что счетчик числа пользователей DLL обнулен, система отключит ее. После этого попытка вызова какой-либо функции из данной DLL приведет к нарушению доступа, так как код по указанному адресу уже не отображается на адресное пространство процесса.

Система поддерживает в каждом процессе свой счетчик DLL, т. е. если поток процесса A вызывает приведенную ниже функцию, а затем тот же вызов делает поток в процессе B, то MyLib.dll проецируется на адресное пространство обоих процессов, а счетчики числа пользователей DLL в каждом из них приравниваются 1.

```
HMODULE hInstDll = LoadLibrary(TEXT("MyLib.dll"));
```

Если же поток процесса B вызовет далее:

```
FreeLibrary(hInstDll);
```

счетчик числа пользователей DLL в процессе B обнулится, что приведет к отключению DLL от адресного пространства процесса B. Но проекция DLL на адресное пространство процесса A не затрагивается, и счетчик числа пользователей DLL в нем остается прежним.

Чтобы определить, спроецирована ли DLL на адресное пространство процесса, поток может вызвать функцию *GetModuleHandle*:

```
HMODULE GetModuleHandle(PCTSTR pszModuleName);
```

Например, следующий код загружает MyLib.dll, только если она еще не спроецирована на адресное пространство процесса:

```
HMODULE hInstDll = GetModuleHandle(TEXT("MyLib"));    // подразумевается
                                                      // расширение .dll
if (hInstDll == NULL) {
   hInstDll = LoadLibrary(TEXT("MyLib"));  // подразумевается расширение .dll
}
```

Если передать NULL функции *GetModuleHandle*, она возвращает описатель исполняемого файла приложения.

Если у вас есть значение *HINSTANCE/HMODULE* для DLL, можно определить полное (вместе с путем) имя DLL или EXE с помощью *GetModuleFileName*:

```
DWORD GetModuleFileName(
    HMODULE hInstHodule,
    PTSTR pszPathName,
    DWORD cchPath);
```

Первый параметр этой функции — значение типа HINSTANCE нужной DLL (или EXE). Второй параметр *pszPathName*, задает адрес буфера, в который она запишет полное имя файла. Третий, и последний, параметр (*cchPath*) определяет размер буфера в символах. Если передать NULL в параметре *hInstModule*, функция *GetModuleFileName* возвращает имя исполняемого файла приложения в переменной *pszPathName*. Подробнее об этих функциях, псевдопеременной *__ImageBase* и *GetModuleHandleEx* см. в главе 4.

Одновременное использование *LoadLibrary* и *LoadLibraryEx* может привести к проецированию одной и той же DLL в разные области адресного пространства. Рассмотрим для примера следующий код:

```
HMODULE hDll1 = LoadLibrary(TEXT("MyLibrary.dll"));
HMODULE hDll2 = LoadLibraryEx(TEXT("MyLibrary.dll"), NULL,
    L0AD_LIBRARY_AS_IMAGE_RESOURCE);
HMODULE hD113 = LoadLibraryEx(TEXT("MyLibrary.dll"), NULL,
    LOAD_LIBRARY_AS_DATAFILE);
```

Как вы думаете, какое значение будут иметь переменные *hDtt1*, *hDU2* и *hDll3*? Ясно, что одинаковое, если загружается файл MyLibrary.dll. Сложнее ответить на этот вопрос, если поменять порядок вызовов следующим образом:

```
HMODULE hDll1 = LoadLibraryEx(TEXT("MyLibrary.dll"), NULL,
    LOAD_LIBRARY_AS_DATAFILE);
HMODULE hD112 = LoadLibraryEx(TEXT("MyLibrary.dll"), NULL,
    LOAD_LIBRARY_AS_IMAGE_RESOURCE); HMODULE hD113 = LoadLi-
    brary(TEXT("MyLibrary.dll"));
```

В этом случае значение переменных *hDll1*, *hDll2* и *hDll3* будет разным! При вызове *LoadLibraryEx* с флагами LOAD_LIBRARY_AS_DATAFILE, LOAD_LIBRARY_AS_DATAFILE_EXCLUSIVE, или LOAD_LIBRARY_AS_IMAGE_RESOURCE операционная система прежде всего проверяет, не загружена ли уже эта библиотека вызовами *LoadLibrary* и *LoadLibraryEx* без указанных флагов. Если это так, возвращается адрес, по которому эта библиотека была спроецирована ранее. Если же эта DLL еще не загружена, Windows проецирует ее в доступную область адресного пространства, но не считает ее полностью загруженной. Если сейчас вызвать *GetModuleFileName* на описателе такого модуля, эта функция вернет 0. Это удобный способ, позволяющий отличить описатели DLL, непригодные для динамических вызовов функций через *GetProcAddress* (см. следующий раздел).

Никогда не забывайте, что адреса спроецированного модуля, которые возвращают *LoadLibrary* и *LoadLibraryEx*, не тождественны даже в случае одного и того же файла DLL.

Явное подключение экспортируемого идентификатора

Поток получает адрес экспортируемого идентификатора из явно загруженной DLL вызовом *GetProcAddress*:

```
FARPROC GetProcAddress(
    HMODULE hInstDll,
    PCSTR pszSymbolName);
```

Параметр *hinstDll* — описатель, возвращенный *LoadLibrary(Ex)* или *GetModuleHandle* и относящийся к DLL, которая содержит нужный идентификатор. Параметр *pszSymbolName* разрешается указывать в двух формах. Во-первых, как адрес строки с нулевым символом в конце, содержащей имя интересующей вас функции:

```
FARPROC pfn = GetProcAddress(hInstDll, "SomeFuncInDll");
```

Заметьте: тип параметра *pszSymbolName* — PCSTR, а не PCTSTR. Это значит, что функция *GetProcAddress* принимает только ANSI-строки — ей нельзя передать Unicode-строку. А причина в том, что идентификаторы функций и переменных в разделе экспорта DLL всегда хранятся как ANSI-строки.

Вторая форма параметра *pszSymbolName* позволяет указывать порядковый номер нужной функции:

```
FARPROC pfn = GetProcAddress(hInstDll, MAKEINTRESOURCE(2));
```

Здесь подразумевается, что вам известен порядковый номер (2) искомого идентификатора, присвоенный ему автором данной DLL. И вновь повторю, что Майкрософт настоятельно не рекомендует пользоваться порядковыми номерами; поэтому вы редко встретите второй вариант вызова *GetProcAddress*.

При любом способе вы получаете адрес содержащегося в DLL идентификатора. Если идентификатор не найден, *GetProcAddress* возвращает NULL.

Учтите, что первый способ медленнее, так как системе приходится проводить поиск и сравнение строк. При втором способе, если вы передаете порядковый номер, не присвоенный ни одной из экспортируемых функций, *GetProcAddress* может вернуть значение, отличное от NULL. В итоге ваша программа, ничего не подозревая, получит неправильный адрес. Попытка вызова функции по этому адресу почти наверняка приведет к нарушению доступа. Я и сам — когда только начинал программировать под Windows и не очень четко понимал эти вещи — несколько раз попадал в эту ловушку. Так что будьте внимательны. (Вот вам, кстати, и еще одна причина, почему от использования порядковых номеров следует отказаться в пользу символьных имен — идентификаторов.)

Чтобы вызвать функцию с помощью указателя, полученного с помощью *GetProcAddress*, сначала необходимо привести его к правильного тину, соответствующему сигнатуре этой функции. Например *typedef void* (CALLBACK *PFN_DUMPMODULE)(HMODULE *hModule*) — сигнатура, соответствующая функции *void DynamicDumpModule(HMODULE hModule)*. Ниже покачано, как динамически вызывать эту функцию из экспортирующей ее DLL:

```
PFN_DUMPMODULE pfnDumpModule =
    (PFN_DUMPMODULE)GetProcAddress(hDll, "DumpModule");
if (pfnDumpModule != NULL) {
    pfnDumpModule(hDll);
}
```

Функция входа/выхода

В DLL может быть лишь одна функция входа/выхода. Система вызывает ее в некоторых ситуациях (о чем речь еще впереди) сугубо в информационных целях, и обычно она используется DLL для инициализации и очистки ресурсов в конкретных процессах или потоках. Если вашей DLL подобные уведомления не нужны, вы не обязаны реализовывать эту функцию. Пример — DLL, содержащая только ресурсы. Но если же уведомления необходимы, функция должна выглядеть так:

```
BOOL WINAPI DllMain(HINSTANCE hInstDll, DWORD fdwReason, PVOID fImpLoad) {

    switch (fdwReason) {
        case DLL_PROCESS_ATTACH:
            // DLL проецируется на адресное пространство процесса
            break;
        case DLL_THREAD_ATTACH:
            // создается поток
            break;
        case DLL_THREAD_DETACH:
            // поток корректно завершается
            break;
        case DLL_PROCESS_DETACH:
            // DLL отключается от адресного пространства процесса
            break;
    }
    return(TRUE);               // используется только для DLL_PROCESS_ATTACH
}
```

> **Примечание.** При вызове *DllMain* надо учитывать регистр букв. Многие случайно вызывают *DllMain*, и это вполне объяснимо: термин *DLL* обычно пишется заглавными буквами. Если вы назовете функцию входа/выхода не *DllMain*, а как-то иначе (пусть даже только один символ будет набран в другом регистре), компиляция и компоновка вашего кода пройдет без проблем, но система проигнорирует такую функцию входа/выхода, и ваша DLL никогда не будет инициализирована.

Параметр *hinstDll* содержит описатель экземпляра DLL. Как и *hinstExe* функции *(w)WinMain*, это значение — виртуальный адрес проекции файла DLL на адресное пространство процесса. Обычно последнее значение сохраняется в глобальной переменной, чтобы его можно было использовать и при вызовах функций, загружающих ресурсы (типа *DialogBox* или *LoadString*). Последний параметр, *fImpLoad*, отличен от 0, если DLL загружена неявно, и равен 0, если она загружена явно.

Параметр *fdwReason* сообщает о причине, по которой система вызвала эту функцию. Он принимает одно из четырех значений: DLL_PROCESS_ATTACH, DLL_PROCESS_DETACH, DLL_THREAD_ATTACH или DLL_THREAD_DETACH. Мы рассмотрим их в следующих разделах.

> **Примечание**. Не забывайте, что DLL инициализируют себя, используя ^5EI функции *DllMain*. К моменту выполнения вашей *DllMain* другие DLL в том же адресном пространстве могут не успеть выполнить свои функции *DllMain*, т. е. они окажутся неинициализированными. Поэтому вы должны избегать обращений из *DllMain* к функциям, импортируемым из других DLL. Кроме того, не вызывайте из *DllMain* функции *LoadLibrary(Ex)* и *FreeLibrary*, так как это может привести к взаимной блокировке.
>
> В документации Platform SDK утверждается, что *DllMain* должна выполнять лишь простые виды инициализации — настройку локальной памяти потока (см. главу 21), создание объектов ядра, открытие файлов и т. д. Избегайте обращений к функциям, связанным с User, Shell, ODBC, COM, RPC и сокетами (а также к функциям, которые их вызывают), потому что соответствующие DLL могут быть еще не инициализированы. Кроме того, подобные функции могут вызывать *LoadLibrary(Ex)* и тем самым приводить к взаимной блокировке.
>
> Аналогичные проблемы возможны и при создании глобальных или статических C++-объектов, поскольку их конструктор или деструктор вызывается в то же время, что и ваша *DllMain*.
>
> Подробнее об этом см. в статье «Best Practices for Creating DLLs» по ссылке http://wwwmicrosoft.com/whdc/driver/kernel/DLL_bestprac.mspx

Уведомление DLL_PROCESS_ATTACH

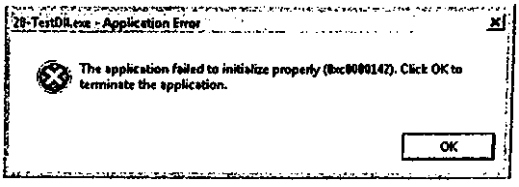
Система вызывает *DllMain* с этим значением параметра *fdwReason* сразу после того, как DLL спроецирована на адресное пространство процесса.

А это происходит, только когда образ DLL-файла проецируется в первый раз. Если затем поток вызовет *LoadLibrary(Ex)* для уже спроецированной DLL, система просто увеличит счетчик числа пользователей этой DLL; так что *DllMain* вызывается со значением DLL_PROCESS_ATTACH лишь раз.

Обрабатывая DLL_PROCESS_ATTACH, библиотскадолжна выполнить в процессе инициализацию, необходимую се функциям. Например, в DLL могут быть функции, которым нужна своя куча (создаваемая в адресном пространстве процесса). В этом случае *DllMain* могла бы создать такую кучу, вызвав *HeapCreate* при обработке уведомления DLL_PROCESS_ATTACH, а описатель созданной кучи сохранить в глобальной переменной, доступной функциям DLL.

При обработке уведомления DLL_PROCESS_ATTACH значение, возвращаемое функцией *DllMain*, указывает, корректно ли прошла инициализация DLL. Например, если вызов *HeapCreate* закончился благополучно, следует вернуть TRUE. А если кучу создать не удалось — FALSE. Для любых других значений *fdwReason* — DLL_PROCESS_DETACH, DLL_THREAD_ATTACH или DLL_THREAD_DETACH — значение, возвращаемое *DllMain*, системой игнорируется.

Конечно, где-то в системе должен быть поток, отвечающий за выполнение кода *DllMain*. При создании нового процесса система выделяет для него адресное пространство, куда проецируется EXE-файл и все необходимые ему DLL-модули. Далее создается первичный поток процесса, используемый системой для вызова *DllMain* из каждой DLL со значением DLL_PROCESS_ATTACH. Когда все спроецированные DLL ответят на это уведомление, система заставит первичный поток процесса выполнить стартовый код из библиотеки C/C++, а потом — входную функцию EXE-файла (*_tmain* или *_tWinMain*). Если *DllMain* хотя бы одной из DLL вернет FALSE, сообщая об ошибке при инициализации, система завершит процесс, удалив из его адресного пространства образы всех файлов; после этого пользователь увидит окно с сообщением о том, что процесс запустить не удалось. Ниже показано соответствующее окно для Windows Vista.



Теперь посмотрим, что происходит при явной загрузке DLL. Когда поток вызывает *LoadLibrary(Ex),* система отыскивает указанную DLL и проецирует ее на адресное пространство процесса. Затем вызывает *DllMain* со значением DLL_PROCESS_ATTACH, используя поток, вызвавший *LoadLibrary(Ex).* Как только *DllMain* обработает уведомление, произойдет возврат из

LoadLibrary(Ex), и поток продолжит работу в обычном режиме. Если же *DllMain* вернет FALSE (неудачная инициализация), система автоматически отключит образ файла DLL от адресного пространства процесса, а вызов *LoadLibrary(Ex)* даст NULL.

Уведомление DLL_PROCESS_DETACH

При отключении DLL от адресного пространства процесса вызывается ее функция *DllMain* со значением DLL_PROCESS_DETACH в параметре *fdwReason*. Обрабатывая это значение, DLL должна провести очистку в данном процессе. Например, вызвать *HeapDestroy*, чтобы разрушить кучу, созданную ею при обработке уведомления DLL_PROCESS_ATTACH. Обратите внимание: если функция *DllMain* вернула FALSE, получив уведомление DLL_PROCESS_ATTACH, то ее нельзя вызывать с уведомлением DLL_ PROCESS_DETACH. Если DLL отключается из-за завершения процесса, то за выполнение кода *DllMain* отвечает поток, вызвавший *ExitProcess* (обычно это первичный поток приложения). Когда ваша входная функция возвращает управление стартовому коду из библиотеки C/C++, тот явно вызывает *ExitProcess* и завершает процесс.

Если DLL отключается в результате вызова *FreeLibrary* или *FreeLibraryAndExitThread*, код *DllMain* выполняется потоком, вызвавшим одну из этих функций. В случае обращения к *FreeLibrary* управление не возвращается, пока *DllMain* не закончит обработку уведомления DLL_PROCESS_DETACH.

Учтите также, что DLL может помешать завершению процесса, если, например, ее *DllMain* входит в бесконечный цикл, получив уведомление DLL_ PROCESS_DETACH. Операционная система уничтожает процесс только после того, как все DLL-модули обработают уведомление DLL_PROCESS_DETACH.

> **Примечание.** Если процесс завершается в результате вызова TerminateProcess, система не вызывает DllMain со значением DLL_PROCESS_DETACH. А значит, ни одна DLL, спроецированная на адресное пространство процесса, не получит шанса на очистку до завершения процесса. Последствия могут быть плачевны — вплоть до потери данных. Вызывайте TerminateProcess только в самом крайнем случае!

На рис. 20-2 показаны операции, выполняемые при вызове *LoadLibrary*, а на рис. 20-3 — при вызове *FreeLibrary*.

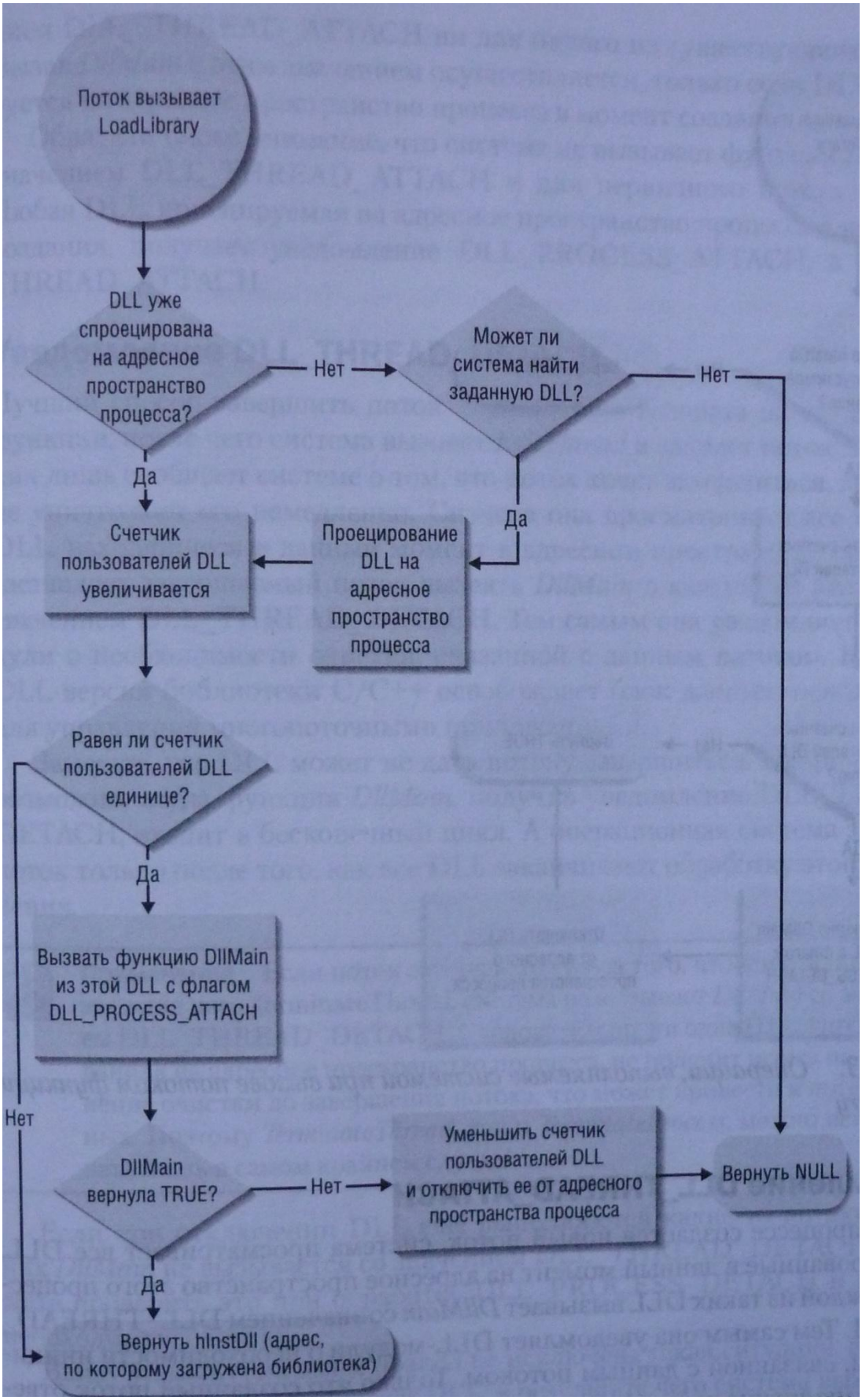


Рис. 20-2. *Операции, выполняемые системой при вызове потоком функции LoadLibrary*

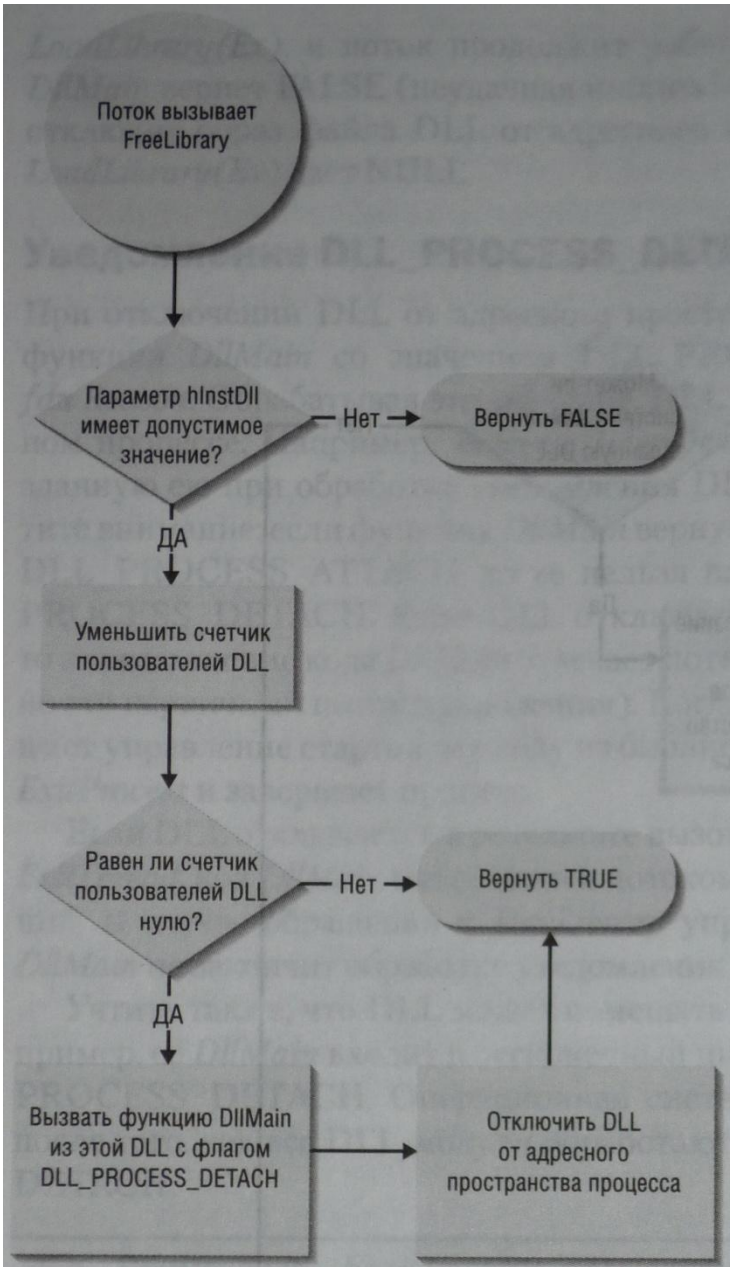


Рис. 20-3. *Операции, выполняемые системой при вызове потоком функции FreeLibrary*

Уведомление DLL_THREAD_ATTACH

Когда в процессе создается новый поток, система просматривает все DLL, спроецированные в данный момент на адресное пространство этого процесса, и в каждой из таких DLL вызывает *DllMain* со значением DLL_THREAD_ATTACH. Тем самым она уведомляет DLL-модули о необходимости инициализации, связанной с данным потоком. Только что созданный поток отвечает за выполнение кода в функциях *DllMain* всех DLL. Работа его собственной (стартовой) функции начинается лишь после того, как все DLL-модули обработают уведомление DLL_THREAD_ATTACH.

Если в момент проецирования DLL на адресное пространство процесса в нем выполняется несколько потоков, система *не* вызывает *DllMain* со значе-

нием DLL_YHREAD_ATTACH ни для одного из существующих потоков. Вызов *DllMain* с этим значением осуществляется, только если DLL проецируется на адресное пространство процесса в момент создания потока.

Обратите также внимание, что система не вызывает функции *DllMain* со значением DLL_THREAD_ATTACH и для первичного потока процесса. Любая DLL, проецируемая на адресное пространство процесса в момент его создания, получает уведомление DLL_PROCESS_ATTACH, а не DLL_THREAD_ATTACH.

Уведомление DLL_THREAD_DETACH

Лучший способ завершить поток — дождаться возврата из его стартовой функции, после чего система вызовет *ExitThread* и закроет поток. Эта функция лишь сообщает системе о том, что поток хочет завершиться, но система не уничтожает его немедленно. Сначала она просматривает все проекции DLL, находящиеся в данный момент в адресном пространстве процесса, и заставляет завершаемый поток вызвать *DllMain* в каждой из этих DLL со значением DLL_THREAD_DETACH. Тем самым она уведомляет DLL-модули о необходимости очистки, связанной с данным потоком. Например, DLL-версия библиотеки C/C++ освобождает блок данных, используемый для управления многопоточными приложениями.

Заметьте, что DLL может не дать потоку завершиться. Например, такое возможно, когда функция *DllMain*, получив уведомление DLL_THREAD_ DETACH, входит в бесконечный цикл. А операционная система закрывает поток только после того, как все DLL заканчивают обработку этого уведомления.

> **Примечание.** Если поток завершается из-за того, что другой поток вызвал для него TerminateThread, система не вызывает *DllMain* со значением DLL_THREAD_DETACH. Следовательно, ни одна DLL, спроецированная на адресное пространство процесса, не получит шанса на выполнение очистки до завершения потока, что может привести к потере данных. Поэтому *TerminateThread*, как и *TerminateProcess*, можно использовать лишь в самом крайнем случае!

Если при отключении DLL еще выполняются какие-то потоки, то для них *DllMain* не вызывается со значением DLL_THREAD_DETACH. Вы можете проверить это при обработке DLL_PROCESS_DETACH и провести необходимую очистку.

Ввиду упомянутых выше правил не исключена такая ситуация: поток вызывает *LoadLibrary* для загрузки DLL, в результате чего система вызывает из этой библиотеки *DllMain* со значением DLL_PROCESS_ATTACH. (В этом случае уведомление DLL_THREAD_ATTACH не посылается.) Затем лоток, загрузивший DLL, завершается, что приводит к новому вызову *DllMain* — на этот раз со значением DLL_THREAD_DETACH. Библиотека уведомля-

ется о завершении потока, хотя она не получала DLL_THREAD_ATTACH, уведомляющего о его подключении. Поэтому будьте крайне осторожны при выполнении любой очистки, связанной с конкретным потоком. К счастью, большинство программ пишется так, что *LoadLibrary* и *FreeLibrary* вызываются одним потоком.

Как система упорядочивает вызовы *DllMain*

Система упорядочивает вызовы функции *DllMain*. Чтобы понять, что я имею в виду, рассмотрим следующий сценарий. Процесс A имеет два потока: A и B. На его адресное пространство проецируется DLL-модуль SomeDLLdll. Оба потока собираются вызвать *CreateThread*, чтобы создать еще два потока: C и D.

Когда поток A вызывает для создания потока C функцию *CreateThread*, система обращается к *DllMain* из SomeDLL.dll со значением DLL_THREAD_ ATTACH. Пока поток C исполняет код *DllMain*, поток B вызывает *CreateThread* для создания потока D. Системе нужно вновь обратиться к *DllMain* со значением DLL_THREAD_ATTACH, и на этот раз код функции должен выполнять поток D. Но система упорядочивает вызовы *DllMain*, и поэтому приостановит выполнение потока D, пока поток C не завершит обработку кода *DllMain* и не выйдет из этой функции.

Закончив выполнение *DllMain*, поток C может начать выполнение своей функции потока. Теперь система возобновляет поток D и позволяет ему выполнить код *DllMain*, при возврате из которой он начнет обработку собственной функции потока.

Обычно никто и не задумывается над тем, что вызовы *DllMain* упорядочиваются. Но я завел об этом разговор потому, что один мой коллега как-то раз написал код, в котором была ошибка, связанная именно с упорядочиванием вызовов *DllMain*. Его код выглядел примерно так:

```
BOOL WINAPI DllMain(HINSTANCE hInstDll, DWORD fdwReason, PVOID fImpLoad) {

    HANDLE hThread;
    DWORD dwThreadId;

    switch (fdwReason) {
    case DLL_PROCESS_ATTACH:
        // DLL проецируется на адресное пространство процесса

        // создаем поток для выполнения какой-то работы
        hThread = CreateThread(NULL, 0, SomeFunction, NULL,
            0, &dwThreadId);

        // задерживаем наш поток до завершения нового потока

        WaitForSingleObject(hThread, INFINITE);
```

```
        // доступ к новому потоку больше не нужен
        CloseHandle(hThread);
        break;

    case DLL_THREAD_ATTACH:
        // создается еще один поток
        break;

    case DLL_THREAD_DETACH:
        // поток завершается корректно
        break;

    case DLL_PROCESS_DETACH:
        // DLL выгружается из адресного пространства процесса
        break;
    }
    return(TRUE);
}
```

Нашли «жучка»? Мы-то его искали несколько часов. Когда *DllMain* получает уведомление DLL_PROCESS_ATTACH, создается новый поток. Системе нужно вновь вызвать эту же *DllMain* со значением DLL_THREAD_ATTACH. Но выполнение нового потока приостанавливается — ведь поток, из-за которого в *DllMain* было отправлено уведомление DLL_PROCESS_ATTACH. свою работу еще не закончил. Проблема кроется в вызове *WaitForSingleObject*. Она приостанавливает выполнение текущего потока до тех пор, пока не завершится новый. Однако у нового потока нет ни единого шанса не только на завершение, но и на выполнение хоть какого-нибудь кода — он приостановлен в ожидании того, когда текущий поток выйдет из *DllMain*. Вот вам и взаимная блокировка — выполнение обоих потоков задержано навеки!

Впервые начав размышлять над этой проблемой, я обнаружил функцию *DisableThreadLibraryCalls:*

```
BOOL DisableThreadLibraryCalls(HMODULE hInstDll);
```

Вызывая ее, вы сообщаете системе, что уведомления DLL_THREAD_ATTACH и DLL_THREAD_DETACH не должны посылаться *DllMain* той библиотеки, которая указана в вызове. Мне показалось логичным, что взаимной блокировки не будет, если система не станет посылать DLL уведомления. Но, проверив свое решение (см. ниже), я убедился, что это не выход.

```
BOOL WINAPI DllMain(HINSTANCE hInstDll, DW0RD fdwReason, PVOID fImpLoad) {

    HANDLE hThread;
    DWORD dwThreadId;

    switch (fdwReason) {
    case DLL_PROCESS_ATTACH:
```

```
        // DLL проецируется на адресное пространство процесса
        // предотвращаем вызов DllMain при создании
        // или завершении потока
        DisableThreadLibraryCalls(hInstDll);

        // создаем лоток для выполнения какой-то работы
        hThread = CreateThread(NULL, 0, SomeFunction, NULL,
            0, &dwThreadId);

        // задерживаем наш поток до завершения нового потока
        WaitForSingleObject(hThread, INFINITE);

        // доступ к новому потоку больше не нужен
        CloseHandle(hThread);
        break;

    case DLL_THREAD_ATTACH:
        // создается еще один поток
        break;

    case DLL_THREAD_DETACH:
        // поток завершается корректно
        break;

    case DLL_PROCESS_DETACH:
        // DLL выгружается из адресного пространства процесса
        break;
    }
    return(TRUE);
}
```

Потом я понял, в чем дело. Создавая процесс, система создает и объект-мьютекс. У каждого процесса свой объект-мьютекс — он не разделяется между несколькими процессами. Его назначение — синхронизация всех потоков процесса при вызове ими функций *DllMain* из DLL, спроецированных на адресное пространство данного процесса.

Когда вызывается *CreateThread*, система создает сначала объект ядра «поток» и стек потока, затем обращается к *WaitForSingteObject*, передавая ей описатель объекта-мьютекса данного процесса. Как только поток захватит этот мьютекс, система заставит его вызвать *DllMain* из каждой DLL со значением DLL_THREAD_ATTACH. И лишь тогда система вызовет *ReleaseMutex*, чтобы освободить объект-мьютекс. Вот из-за того, что система работает именно так, дополнительный вызов *DisableThreadLibraryCalls* и не предотвращает взаимной блокировки потоков. Единственное, что я смог придумать, — переделать эту часть исходного кода так, чтобы ни одна *DllMain* не вызывала *WaitForSingleObject*.

Функция *DllMain* и библиотека C/C++

Рассматривая функцию *DllMain* в предыдущих разделах, я подразумевал, что для сборки DLL вы используете компилятор Microsoft Visual C++. Весьма вероятно, что при написании DLL вам понадобится поддержка со стороны стартового кода из библиотеки C/C++. Например, в DLL есть глобальная переменная — экземпляр какого-то C++-класса. Прежде чем DLL сможет безопасно ее использовать, для переменной нужно вызвать се конструктор, а это работа стартового кода.

При сборке DLL компоновщик встраивает в конечный файл адрес DLL-функции входа/выхода. Вы задаете этот адрес компоновщику ключом / ENTRY. Если у вас компоновщик Microsoft и вы указали ключ /DLL, то по умолчанию он считает, что функция входа/выхода называется *_DllMainCRTStartup*. Эта функция содержится в библиотеке C/C++ и при компоновке статически подключается к вашей DLL—даже если вы используете DLL-версию библиотеки C/C++.

Когда DLL проецируется на адресное пространство процесса, система на самом деле вызывает именно *_DllMainCRTStartup*, а не вашу функцию *DllMain*. Получив уведомление DLL_PROCESS_ATTACH, функция *_DllMainCRTStartup* инициализирует библиотеку C/C++ и конструирует все глобальные и статические C++-объекты. Закончив, _DllMainCRTStartup вызывает вашу *DllMain*. Перед направлением уведомлений *__DllMainCRTStartup* функция *_DllMainCRTStartup* обрабатывает все уведомления DLL_PROCESS_ATTACH согласно параметрам, заданным ключом /GS.

Как только DLL получает уведомление DLL_PROCESS_DETACH, система вновь вызывает *_DllMainCRTStartup*, которая теперь обращается к вашей функции *DllMain*, и, когда та вернет управление, *_DllMainCRTStartup* вызовет деструкторы для всех глобальных и статических C++-объектов. Получив уведомление DLL_THREAD_ATTACH или DLL_THREAD_DETACH, функция *_DllMainCRTStartup* не делает ничего особенного.

Я уже говорил, что реализовать в коде вашей DLL функцию *DllMain* не обязательно. Если у вас нет этой функции, библиотека C/C++ использует свою реализацию *DllMain*, которая выглядит примерно гак (если вы связываете DLL со статической библиотекой C/C++):

```
BOOL WINAPI DllMain(HINSTANCE hInstDll, DWORD fdwReason. PVOID fImpLoad) {

    if (fdwReason == DLL_PROCESS_ATTACH)
        DisableThreadLibraryCalls(hInstDll);
    return(TRUE);
}
```

При сборке DLL компоновщик, не найдя в ваших OBJ-файлах функцию *DllMain*, подключит *DllMain* из библиотеки C/C++. Если вы не предоставили свою версию функции *DllMain*, библиотека C/C++ вполне справедливо будет считать, что вас не интересуют уведомления DLL_TH READ_ATTACH

и DLL_THREAD_DETACH. Функция *DisableThreadLibraryCalls* вызывается для ускорения создания и разрушения потоков.

Отложенная загрузка DLL

Microsoft Visual C++ поддерживает отложенную загрузку DLL — новую, просто фантастическую функциональность, которая значительно упрощает работу с библиотеками. DLL отложенной загрузки (delay-load DLL) — это неявно связываемая DLL, которая не загружается до тех пор, пока ваш код не обратится к какому-нибудь экспортируемому из нее идентификатору. Такие DLL могут быть полезны в следующих ситуациях.

- Если ваше приложение использует несколько DLL, его инициализация может занимать длительное время, потому что загрузчику приходится проецировать их на адресное пространство процесса. Один из способов снять остроту этой проблемы — распределить загрузку DLL в ходе выполнения приложения. DLL отложенной загрузки позволяют легко решить эту задачу.
- Если приложение использует какую-то новую функцию и вы пытаетесь запустить его в более старой версии операционной системы, в которой нет такой функции, загрузчик сообщает об ошибке и не дает запустить приложение. Вам нужно как-то обойти этот механизм и уже в период выполнения, выяснив, что приложение работает в старой версии системы, не вызывать новую функцию. Например, ваша программа должна в Windows Vista использовать новые функции пула потоков, а в прежних версиях Windows — старые функции пула. При инициализации программа должна вызвать *GetVersionEx*, чтобы определить версию текущей операционной системы, и после этого обращаться к соответствующим функциям. Попытка запуска этой программы в Windows XP может привести к тому, что загрузчик сообщит об ошибке, поскольку в этой системе нет нужных функций. Так вот, и эта проблема легко решается за счет DLL отложенной загрузки.

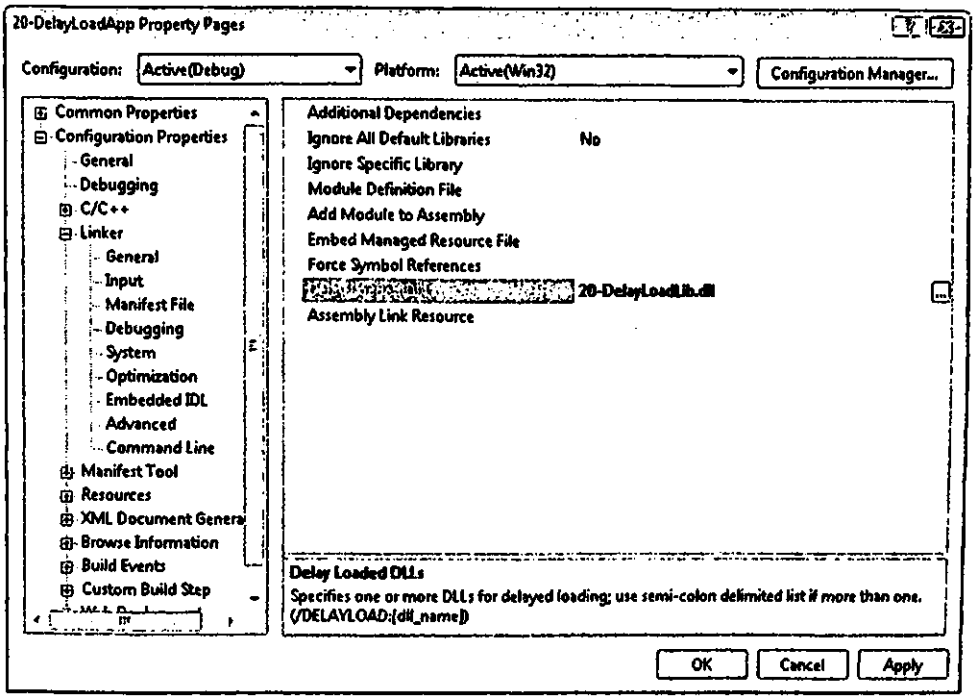
Я довольно долго экспериментировал с DLL отложенной загрузки в Visual C++ и должен сказать, что Microsoft прекрасно справилась со своей задачей. DLL отложенной загрузки открывают массу дополнительных возможностей и корректно работают во всех версиях Windows. Однако, обратите внимание вот на что:

- отложенная загрузка DLL, экспортирующих поля, невозможна;
- невозможна отложенная загрузка модуля Kernel32.dll, так он необходим для вызова LoadLibrary и GetProcAddress;
- нельзя вызывать функции отложенной загрузки в DllMain, иначе возможен крах процесса.
- Подробнее об этом см. в статье «Constraints of Delay Loading DLLs» по ссылке http://msdn2.microsoft.com/en-us/library/yx1x886y(VS.80).aspx.

Давайте начнем с простого: попробуем воспользоваться механизмом поддержки DLL отложенной загрузки. Для этого создайте, как обычно, свою DLL. Точно так же создайте и EXE-модуль, но потом вы должны поменять нару ключей компоновщика и повторить сборку исполняемого файла. Вот эти ключи:

- /Lib:DelayImp.lib
- /DelayLoad:MyDll.dll

> **Внимание!** Нельзя задавать ключи /DELAYLOAD и /DELAY для компоновщика с помощью директивы #pragma comment(linker, “”). Их задают только через свойства проекта параметр Delay Loaded DLLs в меню Configuration Properties | Linker | Input:



Первый ключ заставляет компоновщик внедрить в EXE-модуль специальную функцию, *__delayLoadHelper2*, а второй — выполнить следующие операции:

- удалить MyDll.dll из раздела импорта исполняемого модуля, чтобы при инициализации процесса загрузчик операционной системы не пытался неявно связывать эту библиотеку с EXE-модулем;
- встроить в EXE-файл новый раздел отложенного импорта (.didata) со списком функций, импортируемых из MyDll.dll;
- в привести вызовы функций из DLL отложенной загрузки к вызовам *__delayLoadHelper2*.

При выполнении приложения вызов функции из DLL отложенной загрузки (далее для краткости — DLL-функции) фактически переадресуется к *__delayLoadHelper2*. Последняя, просмотрев раздел отложенного импорта, знает, что нужно вызывать *LoadLibrary*, а затем *GetProcAddress*. Получив адрес DLL-функции, *__delayLoadHelper2* делает так, чтобы в дальнейшем эта DLL-функция вызывалась напрямую. Обратите внимание, что каждая функция в DLL настраивается индивидуально при первом ее вызове. Ключ /DelayLoad компоновщика указывается для каждой DLL, загрузку которой требуется отложить.

Вот собственно, и все. Как видите, ничего сложного здесь нет. Однако следует учесть некоторые тонкости. Загружая ваш EXE-файл, загрузчик операционной системы обычно пытается подключить требуемые DLL и при неудаче сообщает об ошибке. Но при инициализации процесса наличие DLL отложенной загрузки не проверяется. И если функция *__delayLoadHelper2* уже в период выполнения не найдет нужную DLL, она возбудит программное исключение. Вы можете перехватить его, используя SEH, и как-то обработать. Если же вы этого не сделаете, ваш процесс будет закрыт. (О структурной обработке исключений см. главы 23,24 и 25.)

Еще одна проблема может возникнуть, когда *__delayLoadHelper*, найдя вашу DLL, не обнаружит в ней вызываемую функцию (например, загрузчик нашел старую версию DLL). В этом случае *delayLoadHelver* также возбудит программное исключение, и все пойдет по уже описанной схеме. В программе-примере, которая представлена в следующем разделе, я покажу, как написать SEH-код, обрабатывающий подобные ошибки. В ней же вы увидите и массу другого кода, не имеющего никакого отношения к SEH и обработке ошибок. Он использует дополнительные возможности (о них — чуть позже), предоставляемые механизмом поддержки DLL отложенной загрузки. Если эта более «продвинутая» функциональность вас не интересует, просто удалите дополнительный код.

Разработчики Visual C++ определили два кода программных исключений: *VcppException(ERROR_SEVERTTY_ERROR, ERROR_MOD_NOT_FOUND)* и *VcppExceptiontfRROR_SEVERTTY_ERROR, ERROR_PROC_NOT_FOUND)*. Они уведомляют соответственно об отсутствии DLL и DLL-функции. Моя функция фильтра исключений *DelayLoadDllExceptionFilter* реагирует на оба кода. При возникновении любого другого исключения она, как и положено корректно написанному фильтру, возвращает EXCEPTION_CONTINUE_SEARCH. (Программа не должна «глотать» исключения, которые не умеет обрабатывать.) Однако, если генерируется один из приведенных выше кодов, функция *__delayLoadHelper2* предоставляет указатель на структуру *DelayLoadInfo*, содержащую некоторую дополнительную информацию. Она определена в заголовочном файле DelayImp.h, поставляемом с Visual C++.

```
typedef struct DelayLoadInfo {
    DWORD             cb;           // размер структуры
    PCImgDelayDescr   pidd;         // "сырые" данные (все, что пока не обработано)
    FARPROC*          ppfn;         // указатель на адрес функции, которую надо
    LPCSTR            szDll;        // загрузить имя DLL
    DelayLoadProc     dlp;          // имя или порядковый номер процедуры
    HMODULE           hmodCur;      // hInstance загруженной библиотеки
    FARPROC           pfnCur;       // функция, которая будет вызвана на самом деле
    DWORD             dwLastError;  // код ошибки
} DelayLoadInfo, * PDelayLoadInfo;
```

Экземпляр этой структуры данных создается и инициализируется функцией *__delayLoadHelper*, а ее элементы заполняются по мере выполнения задачи, связанной с динамической загрузкой DLL. Внутри вашего SEH-фильтра элемент *szDll* указывает на имя загружаемой DLL, а элемент *dlp* — на имя нужной DLL-функции. Поскольку искать функцию можно как по порядковому номеру, так и по имени, *dlp* представляет собой следующее.

```
typedef struct DelayLoadProc {
    BOOL fIroportByName;
    union {
        LPCSTR szProcName;
        DWORD dwOrdinal;
    };
} DelayLoadProc;
```

Если DLL загружается, но требуемой функции в ней нет, вы можете проверить элемент *hmodCur*, в котором содержится адрес проекции этой DLL, и элемент *dwLastError*, в который помещается код ошибки, вызвавшей исключение. Однако для фильтра исключения код ошибки, видимо, не понадобится, поскольку код исключения и так информирует о том, что произошло. Элемент *pfnCur* содержит адрес DLL-функции, и фильтр исключения устанавливает его в NULL, так как само исключение говорит о том, что *__delayLoadHelper2* не смогла найти этот адрес.

Что касается остальных элементов, то *cb* служит для определения версии системы, *pidd* указывает на раздел, встроенный в модуль и содержащий список DLL отложенной загрузки, а *ppfn* — это адрес, по которому вызывается функция, если она найдена в DLL. Последние два параметра используются внутри *__delayLoadHelper2* и рассчитаны на очень «продвинутое» применение — крайне маловероятно, что они вам когда-нибудь понадобятся.

Итак, самое главное о том, как использовать DLL отложенной загрузки, я рассказал. Но это лишь видимая часть айсберга — их возможности гораздо шире. В частности, вы можете еще и выгружать эти DLL. Допустим, что для распечатки документа вашему приложению нужна специальная DLL. Такая DLL — подходящий кандидат на отложенную загрузку, поскольку она требуется только на время печати документа. Когда пользователь выбирает ко-

манду Print, приложение обращается к соответствующей функции вашей DLL, и та автоматически загружается. Все отлично, но, напечатав документ, пользователь вряд ли станет сразу же печатать что-то еще, а значит, вы можете выгрузить свою DLL и освободить системные ресурсы. Потом, когда пользователь решит напечатать другой документ, DLL вновь будет загружена в адресное пространство вашего процесса.

Чтобы DLL отложенной загрузки можно было выгружать, вы должны сделать две вещи. Во-первых, при сборке исполняемого файла задать ключ /Delay: unload компоновщика. А во-вторых, немного изменить исходный код и поместить в точке выгрузки DLL вызов функции *__FUnloadDelayLoadedDLL2*:

```
BOOL __FUnloadDelayLoadedDLL2(PCSTR szDll);
```

Ключ /Delay:unload заставляет компоновщик создать в файле дополнительный раздел. В нем хранится информация, необходимая для сброса уже вызывавшихся DLL-функций, чтобы к ним снова можно было обратиться через *__delayLoadHelper*. Вызывая *__FUnloadDelayLoadedDLL2*, вы передаете имя выгружаемой DLL. После этого она просматривает раздел выгрузки (unload section) и сбрасывает адреса всех DLL-функций. И, наконец, *__FUnloadDelayLoadedDLL2* вызывает *FreeLibrary*, чтобы выгрузить эту DLL.

Обратите внимание на несколько важных моментов. Во-первых, ни при каких условиях не вызывайте сами *FreeLibrary* для выгрузки DLL, иначе сброса адреса DLL-функции не произойдет, и впоследствии любое обращение к ней приведет к нарушению доступа. Во-вторых, при вызове *__FUnloadDelayLoadedDLL2* в имени DLL нельзя указывать путь, а регистры всех букв должны быть точно такими же, как и при передаче компоновщику в ключе */DelayLoad*; в ином случае вызов *__FUnloadDelayLoadedDLL2* закончится неудачно. В-третьих, если вы вообще не собираетесь выгружать DLL отложенной загрузки, не задавайте ключ */Delay:unload* — тогда вы уменьшите размер своего исполняемого файла. И, наконец, если вы вызовете *__FUnloadDelayLoadedDLL2* из модуля, собранного без ключа */Delay:unload*, ничего страшного не случится: *__FUnloadDelayLoadedDLL2* проигнорирует вызов и просто вернет FALSE.

Другая особенность DLL отложенной загрузки в том, что вызываемые вами функции по умолчанию связываются с адресами памяти, по которым они, как считает система, будут находиться в адресном пространстве процесса. (О связывании мы поговорим чуть позже.) Поскольку связываемые разделы DLL отложенной загрузки увеличивают размер исполняемого файла, вы можете запретить их создание, указав ключ /Delay:nobind компоновщика. Однако связывание, как правило, предпочтительно, поэтому при сборке большинства приложений этот ключ использовать не следует.

И последняя особенность DLL отложенной загрузки. Она, кстати, наглядно демонстрирует характерное для Майкрософт внимание к деталям. Функция *__delayLoadHelper2* может вызывать предоставленные вами функции-ловушки (hook functions), и они будут получать уведомления о том, как

идет выполнение *__delayLoadHelper2*, а также уведомления об ошибках. Кроме того, они позволяют изменять порядок загрузки DLL и формирования виртуального адреса DLL-функций.

Чтобы получать уведомления или изменить поведение *__delauLoadHelper2*, нужно внести два изменения в спой исходный код. Во-первых, вы должны написать функцию-ловушку по образу и подобию *DliHook*, код которой показан на рис. 20-6. Моя функция *DliHook* не влияет на характер работы *__delayLoadHelper2*. Если вы хотите изменить поведение *__delayLoadHelper2*, начните с *DliHook* и модифицируйте ее код так, как вам требуется. Потом передайте ее адрес функции *__delayLoadHelper2*.

В статически подключаемой библиотеке DelayImp.lib определены две глобальные переменные типа *PfnDliHook*: *__pfnDliNotifyHook2* и *__pfnDliFailureHook2*:

```
typedef FARPR0C (WINAPI *PfnDliHook)(
    unsigned dliNotify,
    PDelayLoadInfo pdli);
```

Как видите, это тип данных, соответствующий функции, и он совпадает с прототипом моей *DliHook*. В DelayImp.lib эти две переменные инициализируются значением NULL, которое сообщает *__delayLoadHelper2*, что никаких функций-ловушек вызывать не требуется. Чтобы ваша функция-ловушка все же вызывалась, вы должны присвоить ее адрес одной из этих переменных. В своей программе я просто добавил на глобальном уровне две строки:

```
PfnDliHook _pfnDliNotifyHook2  = DliHook;
PfnDliHook _pfnDliFailureHook2 = DliHook;
```

Так что *__delayLoadHelper* фактически работает с двумя функциями обратного вызова: одна вызывается для уведомлений, другая—для сообщений об ошибках. Поскольку их прототипы идентичны, а первый параметр, *dliNotify*, сообщает о причине вызова функции, я всегда упрощаю себе жизнь, создавая одну функцию и настраивая на нее обе переменные.

> **Совет.** Утилита Dependency Walker (www.DependencyWalker.com) генерирует список зависимостей периода компоновки (статические, а также связанные с отложенной загрузкой DKK, а также отслеживает вызовы *LoadLibrary/GetProcAddress* в период выполнения.

Программа-пример DelayLoadApp

Эта программа (20-DelayLoadApp.exe) показывает, как использовать все преимущества DLL отложенной загрузки. Для демонстрации нам понадобится небольшой DLL-файл; он находится в каталоге 20-DelayLoadLib внутри архива, доступного на Web-сайте поддержки этой книги.

Так как программа загружает модуль «20 DelayLoadLib» с задержкой, загрузчик не проецирует его на адресное пространство процесса при запуске. Периодически вызывая функцию *IsModuleLoaded*, программа выводит окно, которое информирует, загружен ли модуль в адресное пространство процесса. При первом запуске модуль «20 DelayLoadLib» не загружается, о чем и сообщается в окне (рис. 20-4).

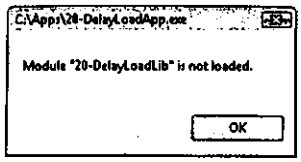


Рис. 20-4. *DelayLoadApp сообщает, что модуль «20 DelayLoadLib» незагружен*

Далее программа вызывает функцию, импортируемую из DLL, и это заставляет *__delayLoadHelper* автоматически загрузить нужную DLL. Когда функция вернет управление, программа выведет окно, показанное на рис. 20-5.

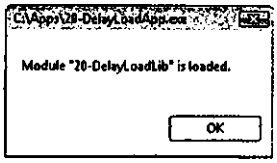


Рис. 20-5. *DelayLoadApp сообщает, что модуль «20 DelayLoadLib» загружен*

Когда пользователь закроет это окно, будет вызвана другая функция из той же DLL. В этом случае DLL не перезагружается в адресное пространство, но перед вызовом новой функции придется определять ее адрес.

Далее вызывается *__FUnloadDelayLoadedDLL2*, и модуль «20 DelayLoadLib» выгружается из памяти. После очередного вызова *IsModuleLoaded* на экране появляется окно, показанное на рис. 20-4. Наконец, вновь вызывается импортируемая функция, что приводит к повторной загрузке модуля «20 DelayLoadLib», а *IsModuleLoaded* открывает окно, как на рис. 20-5.

Если все нормально, то программа будет работать, как я только что рассказал. Однако, если перед запуском программы вы удалите модуль «20 DelayLoadLib» или если в этом модуле не окажется одной из импортируемых функций, будет возбуждено исключение. Из моего кода видно, как корректно выйти из такой ситуации.

Наконец, эта программа демонстрирует, как настроить функцию-ловушку из DLL отложенной загрузки. Моя схематическая функция *DliHook* не делает ничего интересного. Тем не менее она перехватывает различные уведомления и показывает их вам.

```
DelayLoadApp.cpp

/******************************************************************************
Module:  DelayLoadApp.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>
#include <StrSafe.h>


//////////////////////////////////////////////////////////////////////////////


#include <Delayimp.h>   // For error handling & advanced features
#include "..\20-DelayLoadLib\DelayLoadLib.h"    // My DLL function prototypes


//////////////////////////////////////////////////////////////////////////////


// Statically link __delayLoadHelper2/__FUnloadDelayLoadedDLL2
#pragma comment(lib, "Delayimp.lib")

// Note: it is not possible to use #pragma comment(linker, "")
//       for /DELAYLOAD and /DELAY


// The name of the Delay-Load module (only used by this sample app)
TCHAR g_szDelayLoadModuleName[] = TEXT("20-DelayLoadLib");


//////////////////////////////////////////////////////////////////////////////


// Forward function prototype
LONG WINAPI DelayLoadDllExceptionFilter(PEXCEPTION_POINTERS pep);


//////////////////////////////////////////////////////////////////////////////


void IsModuleLoaded(PCTSTR pszModuleName) {

   HMODULE hmod = GetModuleHandle(pszModuleName);
   char sz[100];
#ifdef UNICODE
   StringCchPrintfA(sz, _countof(sz), "Module \"%S\" is %Sloaded.",
      pszModuleName, (hmod == NULL) ? L"not " : L"");
#else
```

```
   StringCchPrintfA(sz, _countof(sz), "Module \"%s\" is %sloaded.",
      pszModuleName, (hmod == NULL) ? "not " : "");
#endif
   chMB(sz);
}


///////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Wrap all calls to delay-load DLL functions inside SEH
   __try {
      int x = 0;

      // If you're in the debugger, try the new Debug.Modules menu item to
      // see that the DLL is not loaded prior to executing the line below
      IsModuleLoaded(g_szDelayLoadModuleName);

      x = fnLib();  // Attempt to call delay-load function

      // Use Debug.Modules to see that the DLL is now loaded
      IsModuleLoaded(g_szDelayLoadModuleName);

      x = fnLib2(); // Attempt to call delay-load function

      // Unload the delay-loaded DLL
      // NOTE: Name must exactly match /DelayLoad:(DllName)
                                          PCSTR pszDll = "20-DelayLoadLib.dll";
      __FUnloadDelayLoadedDLL2(pszDll);

      // Use Debug.Modules to see that the DLL is now unloaded
      IsModuleLoaded(g_szDelayLoadModuleName);

      x = fnLib();  // Attempt to call delay-load function

      // Use Debug.Modules to see that the DLL is loaded again
      IsModuleLoaded(g_szDelayLoadModuleName);
   }
   __except (DelayLoadDllExceptionFilter(GetExceptionInformation())) {
      // Nothing to do in here, thread continues to run normally
   }
```

```
   // More code can go here...

   return(0);
}


//////////////////////////////////////////////////////////////////////////


LONG WINAPI DelayLoadDllExceptionFilter(PEXCEPTION_POINTERS pep) {

   // Assume we recognize this exception
   LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;

   // If this is a Delay-load problem, ExceptionInformation[0] points
   // to a DelayLoadInfo structure that has detailed error info
   PDelayLoadInfo pdli =
      PDelayLoadInfo(pep->ExceptionRecord->ExceptionInformation[0]);

   // Create a buffer where we construct error messages
   char sz[500] = { 0 };

   switch (pep->ExceptionRecord->ExceptionCode) {
   case VcppException(ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND):
      // The DLL module was not found at runtime
      StringCchPrintfA(sz, _countof(sz), "Dll not found: %s", pdli->szDll);
      break;

   case VcppException(ERROR_SEVERITY_ERROR, ERROR_PROC_NOT_FOUND):
      // The DLL module was found, but it doesn't contain the function
      if (pdli->dlp.fImportByName) {
         StringCchPrintfA(sz, _countof(sz), "Function %s was not found in %s",
            pdli->dlp.szProcName, pdli->szDll);
      } else {
         StringCchPrintfA(sz, _countof(sz), "Function ordinal %d was not found
in %s",
            pdli->dlp.dwOrdinal, pdli->szDll);
      }
      break;

   default:
      // We don't recognize this exception
      lDisposition = EXCEPTION_CONTINUE_SEARCH;
      break;
   }
```

```
   if (lDisposition == EXCEPTION_EXECUTE_HANDLER) {
      // We recognized this error and constructed a message, show it
      chMB(sz);
   }

   return(lDisposition);
}


///////////////////////////////////////////////////////////////////////////


// Skeleton DliHook function that does nothing interesting
FARPROC WINAPI DliHook(unsigned dliNotify, PDelayLoadInfo pdli) {

   FARPROC fp = NULL;   // Default return value

   // NOTE: The members of the DelayLoadInfo structure pointed
   // to by pdli shows the results of progress made so far.

   switch (dliNotify) {
   case dliStartProcessing:
      // Called when __delayLoadHelper2 attempts to find a DLL/function
      // Return 0 to have normal behavior or nonzero to override
      // everything (you will still get dliNoteEndProcessing)
      break;

   case dliNotePreLoadLibrary:
      // Called just before LoadLibrary
      // Return NULL to have __delayLoadHelper2 call LoadLibary
      // or you can call LoadLibrary yourself and return the HMODULE
      fp = (FARPROC) (HMODULE) NULL;
      break;

   case dliFailLoadLib:
      // Called if LoadLibrary fails
      // Again, you can call LoadLibary yourself here and return an HMODULE
      // If you return NULL, __delayLoadHelper2 raises the
      // ERROR_MOD_NOT_FOUND exception
      fp = (FARPROC) (HMODULE) NULL;
      break;

   case dliNotePreGetProcAddress:
      // Called just before GetProcAddress
      // Return NULL to have __delayLoadHelper2 call GetProcAddress,
      // or you can call GetProcAddress yourself and return the address
      fp = (FARPROC) NULL;
      break;
```

```
   case dliFailGetProc:
      // Called if GetProcAddress fails
      // You can call GetProcAddress yourself here and return an address
      // If you return NULL, __delayLoadHelper2 raises the
      // ERROR_PROC_NOT_FOUND exception
      fp = (FARPROC) NULL;
      break;

   case dliNoteEndProcessing:
      // A simple notification that __delayLoadHelper2 is done
      // You can examine the members of the DelayLoadInfo structure
      // pointed to by pdli and raise an exception if you desire
      break;
   }

   return(fp);
}


///////////////////////////////////////////////////////////////////////////


// Tell __delayLoadHelper2 to call my hook function
PfnDliHook __pfnDliNotifyHook2  = DliHook;
PfnDliHook __pfnDliFailureHook2 = DliHook;


/////////////////////////////////// End of File /////////////////////////////////
```

DelayLoadLib.cpp

```
/******************************************************************************
Module:  DelayLoadLib.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <Windowsx.h>
#include <tchar.h>


///////////////////////////////////////////////////////////////////////////


#define DELAYLOADLIBAPI extern "C" __declspec(dllexport)
#include "DelayLoadLib.h"


///////////////////////////////////////////////////////////////////////////
```

```
int fnLib() {

   return(321);
}


//////////////////////////////////////////////////////////////////////////


int fnLib2() {

   return(123);
}


//////////////////////////////// End of File ////////////////////////////////
```

```
DelayLoadLib.h

/******************************************************************************
Module:  DelayLoadLib.h
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#ifndef DELAYLOADLIBAPI
#define DELAYLOADLIBAPI extern "C" __declspec(dllimport)
#endif


//////////////////////////////////////////////////////////////////////////


DELAYLOADLIBAPI int fnLib();
DELAYLOADLIBAPI int fnLib2();


//////////////////////////////// End of File ////////////////////////////////
```

Переадресация вызовов функций

Запись о переадресации вызова функции (function forwarder) — это строка в разделе экспорта DLL, которая перенаправляет вызов к другой функции, находящейся в другой DLL. Например, запустив утилиту DumpBin из Visual C++ для Kernel32.dll в Windows Vista, вы среди прочей информации увидите и следующее.

```
C:\Windows\System32>DumpBin - Exports Kernel32.dll        (часть вывода опущена)
75    49     CloseThreadpoolIo (forwarded to NTDLL.TpReleaseIoCoropletion)
76    4A     CloseThreadpoolTimer (forwarded to NTDLL.TpReleaseTimer)
77    4B     CloseThreadpoolWait (forwarded to NTDLL.TpReleaseWait)
78    4C     CloseThreadpoolWork (forwarded to NTDLL.TpReleaseWork)
(остальное тоже опущено)
```

Здесь есть четыре переадресованные функции. Всякий раз, когда ваше приложение вызывает *CloseThreadpoolIo*, *CloseThreadpoolTimer*, *CloseThreadpoolWait*, or *CloseThreadpoolWork*, его EXE-модуль динамически связывается с Kernel32.dll. При запуске EXE-модуля загрузчик загружает Kernel32.dll и, обнаружив, что переадресуемые функции на самом деле находятся в NTDLL. dll, загружает и эту DLL. Обращаясь к *CloseThreadpoolIo*, программа фактически вызывает функцию *TpReleaseIoCompletion* из NTDLL.dll. А функции *CloseThreadpoolIo* вообще нет!

При вызове *CloseThreadpoolIo* (см. ниже) функция *GetProcAddress* просмотрит раздел экспорта Kernel32.dll и, выяснив, что *CloseThreadpoolIo* — переадресуемая функция, рекурсивно вызовет сама себя для поиска *RtlAllocateHeap* в разделе экспорта NTDLL.dll.

Вы тоже можете применять переадресацию вызовов функций в своих DLL. Самый простой способ — воспользоваться директивой *pragma*:

```
// переадресация к функции из DllWork
#pragma comment(linker, "/export:SomeFunc=DllWork.SomeOtherFunc")
```

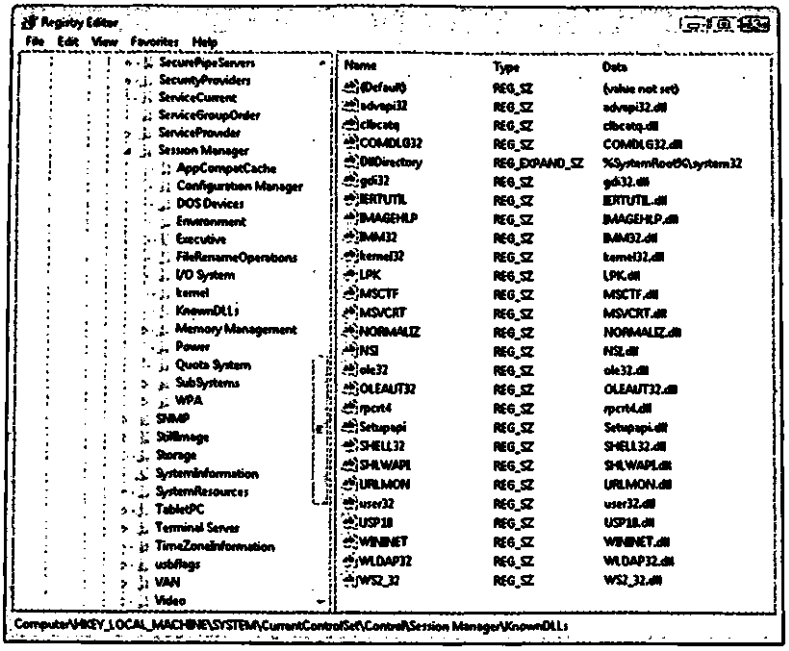
Эта директива сообщает компоновщику, что DLL должна экспортировать функцию *SomeFunc*, которая на самом деле реализована как функция *SomeOtherFunc* в модуле DllWork.dll. Такая запись нужна для каждой переадресуемой функции.

Известные DLL

Некоторые DLL, поставляемые с операционной системой, обрабатываются по-особому. Они называются *известными DLL* (known DLLs) и ведут себя точно так же, как и любые другие DLL с тем исключением, что система всегда ищет их в одном и том же каталоге. В реестре есть раздел:

```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
    Session Manager\KnownDLLs
```

Содержимое этого раздела может выглядеть примерно так, как показано ниже (при просмотре реестра с помощью утилиты RegEdit.exe). Как видите, здесь содержится набор параметров, имена которых совпадают с именами известных DLL Значения этих параметров представляют собой строки, идентичные именам параметров, но дополненные расширением .dll. (Впрочем, это не всегда так, и вы сами убедитесь в этом на следующем примере.) Когда вы вызываете *LoadLibrary* или *LoadLibraryEx*, каждая из них сначала проверяет, указано ли имя DLL вместе с расширением .dll. Если нет, поиск DLL ведется по обычным правилам.



Если же расширение .dll указано, функция его отбрасывает и ищет в разделе реестра *KnownDLLs* параметр, имя которого совпадает с именем DLL. Если его нет, вновь применяются обычные правила поиска. А если он есть, система считывает значение этого параметра и пытается загрузить заданную в нем DLL. При этом система ищет DLL в каталоге, на который указывает значение, связанное с параметром реестра *DllDirectory*. По умолчанию в Windows Vista параметру *DllDirectory* присваивается значение %SystemRoot%\System32.

А теперь допустим, что мы добавили в раздел реестра *KnownDLLs* такой параметр:

```
Имя параметра:      SomeLib
Значение параметра:  SomeOtherLib.dll
```

Когда мы вызовем следующую функцию, система будет искать файл по обычным правилам.

```
LoadLibrary(TEXT("SomeLib"));
```

Но если мы вызовем ее так, как показано ниже, система увидит, что в реестре есть параметр с идентичным именем (не забудьте: она отбрасывает расширение .dll).

```
LoadLibrary(TEXT("SomeLib.dir));
```

Таким образом, система попытается загрузить SomeOtherLib.dll вместо SomeLib.dll. При этом она будет сначала искать SomeOtherLib.dll в каталоге %SystemRoot%\System32. Если нужный файл в этом каталоге есть, будет загружен именно он. Нет — *LoadLibrary(Ex)* вернет NULL, а *GetLastError* — ERROR_FILE__NOT_FOUND (2).

Перенаправление DLL

Когда разрабатывались первые версии Windows, оперативная намять и дисковое пространство были крайне дефицитным ресурсом, так что Windows была рассчитана на предельно экономное их использование — с максимальным разделением между потребителями. В связи с этим Майкрософт рекомендовала размещать все модули, используемые многими приложениями (например, библиотеку C/C++ и DLL, относящиеся к MFC) в системном каталоге Windows, где их можно было легко найти.

Однако со временем это вылилось в серьезную проблему: программы установки приложений то и дело перезаписывали новые системные файлы старыми или не полностью совместимыми. Из-за этого уже установленные приложения переставали работать. Но сегодня жесткие диски стали очень емкими и недорогими, оперативная память тоже значительно подешевела. Поэтому Майкрософт сменила свою позицию на прямо противоположную: теперь она настоятельно рекомендует размещать все файлы приложения в своем каталоге и ничего не трогать в системном каталоге Windows. Тогда ваше приложение не нарушит работу других программ, и наоборот.

С той же целью Майкрософт ввела в Windows поддержку перенаправления DLL (DLL redirection). Она заставляет загрузчик операционной системы загружать модули сначала из каталога вашего приложения и, только если их там нет, искать в других каталогах.

Чтобы загрузчик всегда проверял сначала каталог приложения, нужно всего лишь поместить туда специальный файл. Его содержимое не имеет значения и игнорируется — важно только его имя: оно должно быть в виде AppName.local. Так, если исполняемый файл вашего приложения — SuperApp.exe, присвойте перенаправляющему файлу имя SuperApp.exe.local.

Функция *LoadLibrary(Ex)* проверяет наличие этого файла и, если он есть, загружает модуль из каталога приложения; в ином случае *LoadLibrary(Ex)* работает так же, как и раньше. Если нужный модуль отсутствует в каталоге приложения, *LoadLibrary(Ex)* действует как обычно. Учтите, что вместо .local-файла можно создать одноименный каталог, в котором Windows легко найдет их.

Перенаправление DLL исключительно полезно для работы с зарегистрированными COM-объектами. Оно позволяет приложению размещать DLL с COM-объектами в своем каталоге, и другие программы, регистрирующие тс же объекты, не будут мешать его нормальной работе.

Учтите, что по соображениям безопасности (во избежание загрузки поддельных системных DLL из каталога приложения вместо системных каталогов) в Windows Vista эта функция по умолчанию отключена. Чтобы включить ее снова, необходимо создать DWORD-значение DevOverrideEnable в разделе HKLM\Software\Microsoft\WindowsNT\CurrentVersion\Image File Execution Options и присвоить ему значение 1.

Примечание. Со времени выхода Windows XP и расцвета приложений для платформы Microsoft .NET автономные приложения и сборки, пригодные для одновременного развертывания (side-by-side assemblies), могут содержать не только управляемый, но и неуправляемый код (см. статью «Isolated Applications and Side-by-side Assemblies» at http://msdn2.microsoft.com/en-us/library/aa375193.aspx).

Модификация базовых адресов модулей

У каждого EXE и DLL-модуля есть *предпочтительный базовый адрес* (preferred base address) — идеальный адрес, по которому он должен проецироваться на адресное пространство процесса. Для EXE-модуля компоновщик выбирает в качестве такого адреса значение 0x00400000, а для DLL-модуля — 0x10000000. Выяснить этот адрес позволяет утилита DumpBin с ключом /Headers. Вот какую информацию сообщает DumpBin о самой себе:

```
C:\>DUMPBIN /headers dumpbin.exe

Microsoft (R) COFF/PE Dumper Version 8.00.50727.42
Copyright (C) Microsoft Corporation. All rights reserved.

Dump of file dumpbin.exe

PE signature found

File Type: EXECUTABLE IMAGE

FILE HEADER VALUES
             14C machine (i386)
               3 number of sections
        4333ABD8 time date stamp Fri Sep 23 09:16:40 2005
               0 file pointer to symbol table
               0 number of symbols
              E0 size of optional header
             123 characteristics
                   Relocations stripped
                   Executable
                   Application can handle large (>26B) addresses
                   32 bit word machine

OPTIONAL HEADER VALUES
              10B magic # (PE32)
             8.00 linker version
             1200 size of code
              800 size of initialized data
                0 size of uninitialized data
```

```
              170C entry point (0040170C)
              1000 base of code
              3000 base of data
            400000 image base (00400000 to 00404FFF) <- Module's preferred
              base address
              1000 section alignment
               200 file alignment
              5.00 operating system version
              8.00 image version
              4.00 subsystem version
                 0 Win32 version
              5000 size of image
               400 size of headers
             1306D checksum
                 3 subsystem (Windows CUI)
              8000 DLL characteristics
                    Terminal Server Aware
            100000 size of stack reserve
              2000 size of stack commit
            100000 size of heap reserve
              1000 size of heap commit
                 0 loader flags
                10 number of directories
...
```

При запуске исполняемого модуля загрузчик операционной системы создает виртуальное адресное пространство нового процесса и проецирует этот модуль по адресу 0x00400000, а DLL-модуль — по адресу 0x10000000. Почему так важен предпочтительный базовый адрес? Взгляните на следующий фрагмент кода:

```
int g_x;

void Func() {
    g.x = 5;                                // нас интересует эта строка
}
```

После обработки функции *Func* компилятором и компоновщиком полученный машинный код будет выглядеть приблизительно так:

```
M0V    [0x00414540], 5
```

Иначе говоря, компилятор и компоновщик «жестко зашили» в машинный код адрес переменной *g_x* в адресном пространстве процесса (0x00414540). Но, конечно, этот адрес корректен, только если исполняемый модуль будет загружен по базовому адресу 0x00400000.

А что получится, если тот же исходный код будет помещен в DLL? Тогда машинный код будет иметь такой вид:

```
MOV    [0x10014540], 5
```

Заметьте, что и на этот раз виртуальный адрес переменной *g_x* «жестко зашит» в машинный код. И опять же этот адрес будет правилен только при том условии, что DLL загрузится по своему базовому адресу.

А теперь представьте, что вы создали приложение с двумя DLL. По умолчанию компоновщик установит для EXE-модуля предпочтительный базовый адрес 0x00400000, а для обеих DLL — 0x10000000. Если вы затем попытаетесь запустить исполняемый файл, загрузчик создаст виртуальное адресное пространство и спроецирует EXE-модуль по адресу 0x00400000. Далее первая DLL будет спроецирована по адресу 0x10000000, но загрузить вторую DLL по предпочтительному базовому адресу не удастся — ее придется проецировать по какому-то другому адресу.

Переадресация (relocation) в EXE- или DLL-модуле — операция просто ужасающая, и вы должны сделать все, чтобы избежать ее. Почему? Допустим, загрузчик переместил вторую DLL по адресу 0x20000000. Тогда код, который присваивает переменной *g_x* значение 5, должен измениться на:

```
MOV    [0x20014540], 5
```

Но в образе файла код остался прежним:

```
MOV    [0x10014540], 5
```

Если будет выполнен именно этот код, он перезапишет какое-то 4-байтовое значение в первой DLL значением 5. Но, по идее, такого не должно случиться. Загрузчик исправит этот код. Дело в том, что, создавая модуль, компоновщик встраивает в конечный файл раздел переадресации (relocation section) со списком байтовых смещений. Эти смещения идентифицируют адреса памяти, используемые инструкциями машинного кода. Если загрузчику удастся спроецировать модуль по его предпочтительному базовому адресу, раздел переадресации не понадобится. Именно этого мы и хотим.

С другой стороны, если модуль не удастся спроецировать по базовому адресу, загрузчик обратится к разделу переадресации и последовательно обработает все его записи. Для каждой записи загрузчик обращается к странице памяти, где содержится машинная команда, которую надо модифицировать, получает используемый ею на данный момент адрес и добавляет к нему разницу между предпочтительным базовым адресом модуля и его фактическим адресом.

В предыдущем примере вторая DLL была спроецирована по адресу 0x20000000, тогда как ее предпочтительный базовый адрес — 0x10000000. Получаем разницу (0x10000000), добавляем ее к адресу в машинной команде и получаем:

```
MOV    [0x20014540], 5
```

Теперь и вторая DLL корректно ссылается на переменную *g_x*.

Невозможность загрузить модуль по предпочтительному базовому адресу создает две крупные проблемы.

- Загрузчику приходится обрабатывать все записи раздела переадресации и модифицировать уйму кода в модуле. Это сильнейшим образом сказывается на быстродействии и может резко увеличить время инициализации приложения.
- Из-за того что загрузчик модифицирует в оперативной памяти страницы с кодом модуля, системный механизм копирования при записи создает их копии в страничном файле.

Вторая проблема особенно неприятна, поскольку теперь страницы с кодом модуля больше нельзя выгружать из памяти и перезагружать из его файла на диске. Вместо этого страницы будут постоянно сбрасываться в страничный файл и подгружаться из него. Это тоже отрицательно скажется на производительности. Но и это еще не все. Поскольку все страницы с кодом модуля размещаются в страничном файле, в системе сокращается объем общей памяти, доступной другим процессам, а это ограничивает размер электронных таблиц, документов текстовых процессоров, чертежей CAD, растровых изображений и т. д.

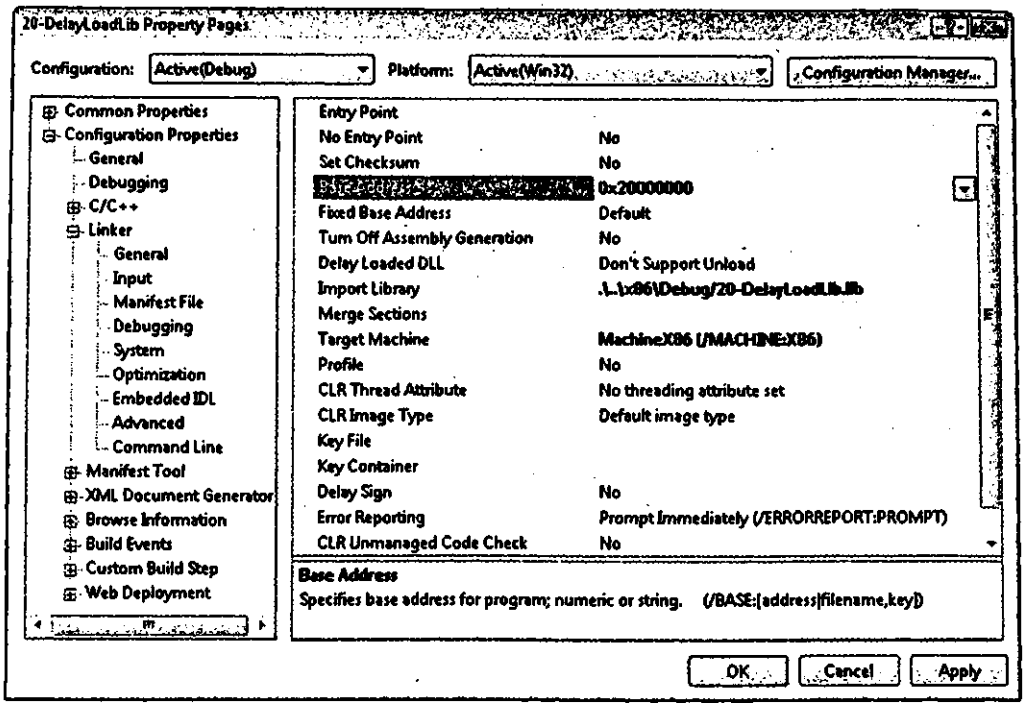
Кстати, вы можете создать EXE- или DLL-модуль без раздела переадресации, указав при сборке ключ /FIXED компоновщика. Тогда у модуля будет меньший размер, но загрузить его по другому базовому адресу, кроме предпочтительного, уже не удастся. Если загрузчику понадобится модифицировать адреса в модуле, в котором нет раздела переадресации, он уничтожит весь процесс, и пользователь увидит сообщение «Abnormal Process Termination> («аварийное завершение процесса»).

Для DLL, содержащей только ресурсы, это тоже проблема. Хотя в ней нет машинного кода, отсутствие раздела переадресации не позволит загрузить ее по базовому адресу, отличному от предпочтительного. Просто нелепо. Но, к счастью, компоновщик может встроить в заголовок модуля информацию о том, что в модуле нет раздела переадресации, так как он вообще не нужен. А загрузчик Windows 2000, обнаружив эту информацию, может загрузить DLL, которая содержит только ресурсы, без дополнительной нагрузки на страничный файл.

Для создания файла с немодифицируемыми адресами предназначен ключ /SUBSYSTEM:WINDOWS, 5.0 или /SUBSYSTEM:CONSOLE, 5.0; ключ /FIXED при этом не нужен. Если компоновщик определяет, что модификация адресов в модуле не понадобится, он опускает раздел переадресации и сбрасывает в заголовке специальный флаг IMAGEJFILE_RELOCS_ STRIPPED. Тогда Windows 2000 увидит, что данный модуль можно загружать по базовому адресу, отличному от предпочтительного, и что ему не требуется модификация адресов. Но все, о чем я только что рассказал, поддерживается начиная с Windows 2000 (вот почему в ключе /SUBSYSTEM указывается значение *5.0*).

Теперь вы понимаете, насколько важен предпочтительный базовый адрес. Загружая несколько модулей в одно адресное пространство, для каждого из них приходится выбирать свои базовые адреса. Диалоговое окно Project

Settings в среде Microsoft Visual Studio значительно упрощает решение этой задачи. Вам нужно лишь открыть вкладку Link, в списке Category указать Output, а в поле Base Address ввести предпочтительный адрес. Например, на следующей иллюстрации для DLL установлен базовый адрес 0x20000000.



Кстати, всегда загружайте DLL, начиная со старших адресов; это позволяет уменьшить фрагментацию адресного пространства.

> **Примечание.** Предпочтительные базовые адреса должны быть кратны гранулярности выделения памяти (64 Кб на всех современных платформах). В будущем эта цифра может измениться. Подробнее о гранулярности выделения памяти см. главу 13.

Все это просто замечательно, но что делать, если понадобится загрузить кучу модулей в одно адресное пространство? Было бы неплохо «одним махом» задать правильные базовые адреса для всех модулей. К счастью, такой способ есть.

В Visual Studio есть утилита Rebase.exe. Запустив ее без ключей в командной строке, вы получите информацию о том, как ею пользоваться. Вот что представляет собой эта функция:

```
usage: REBASE [switches]
              [-R image-root [-G filename] [-0 filename] [-N filename]]
              image-names...
```

```
One of -b and -i switches are mandatory,

[-a] Does nothing
[-b InitialBase] specify initial base address
[-c coffbase_filename] generate coffbase.txt
     -C includes filename extensions, -c does not
[-d] top down rebase
[-e .SizeAdjustment] specify extra size to allow for image growth
[-f] Strip relocs after rebasing the image
[-i coffbase_filename] get base addresses from coffbase_filename
[-l logFilePath] write image bases to log file.
[-p] Does nothing
[-q] minimal output
[-s] just sum image range
[-u symbol_dir] Update debug info in .DBQ along this path
[-v] verbose output
[-x symbol_dir] Same as -u
[-z] allow system file rebasing
[-?] display this message
[-R image_root] set image root for use by -G, -0f -N
[-G filename] group images together in address space
[-0 filename] overlay images in address space
[-N filename] leave images at their original address
   -G, -0, -N, may occur multiple times.   File "filename"
   contains a list of files (relative to "image-root")

'image-names' can be either a file (foo.dll) or files (*.dll)
              or a file that lists other files (dfiles.txt).
              If you want to rebase to a fixed address (ala OFE)
              use the eefiles.txt format where files.txt contains
              address/size combos in addition to the filename
```

Она описана в документации Platform SDK, и я не буду ее здесь детально рассматривать. Добавлю лишь, что в ней нет ничего сверхъестественного: она просто вызывает функцию *ReBaseImage* из ImageHlp API для каждого указанного файла.

```
BOOL ReBaseImage(
    PCSTR CurrentImageName,           // полное имя обрабатываемого файла
    PCSTR SymbolPath,                 // символьный путь к файлу (необходим для
                                      // корректности отладочной информации)
    BOOL bRebase,                     // TRUE = выполнить реальную модификацию
                                      // адреса; FALSE = имитировать такую
    BOOL bRebaseSysFileOk,            // модификацию FALSE = не модифицировать
                                      // адреса системных файлов
    BOOL bGoingDown,                  // TRUE = модифицировать адрес модуля,
```

```
                                    // продвигаясь в сторону уменьшения адресов
    ULONG CheckImageSize,           // ограничение на размер получаемого в итоге
                                    // модуля (0 8 размер не ограничен)
    ULONG* pOldImageSize,           // исходный размер модуля
    ULONG* pOldImageBase,           // исходный базовый адрес модуля
    ULONG* pNewImageSize,           // новый размер модуля
    ULONG* pNewImageBase,           // новый базовый адрес модуля
    ULONG TimeStamp);               // новая временная метка модуля
```

Когда вы запускаете утилиту Rebase, указывая ей несколько файлов, она выполняет следующие операции.

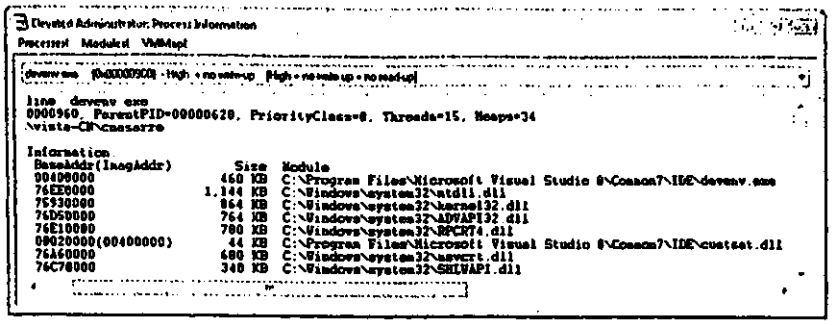
1. Моделирует создание адресного пространства процесса.
2. Открывает все модули, которые загружались бы в это адресное пространство, и получает предпочтительный базовый адрес и размер каждого модуля.
3. Моделирует переадресацию модулей в адресном пространстве, добиваясь того, чтобы модули не перекрывались.
4. В каждом модуле анализирует раздел переадресации и соответственно изменяет код в файле модуля на диске.
5. Записывает новый базовый адрес в заголовок файла.

Rebase — отличная утилита, и я настоятельно рекомендую вам пользоваться ею. Вы должны запускать ее ближе к концу цикла сборки, когда уже созданы все модули приложения. Кроме того, применяя утилиту Rebase, можно проигнорировать настройку базового адреса в диалоговом окне Project Settings. Она автоматически изменит базовый адрес 0x10000000 для DLL, задаваемый компоновщиком по умолчанию.

Но ни при каких обстоятельствах не модифицируйте базовые адреса системных модулей. Их адреса уже оптимизированы Microsoft, так что при загрузке в одно адресное пространство системные модули не перекрываются.

Я, кстати, добавил специальный инструмент в свою программу ProcessInfo.exe (см. главу 4). Он показывает список всех модулей, находящихся в адресном пространстве процесса. В колонке BaseAddr сообщается виртуальный адрес, по которому загружен модуль. Справа от BaseAdd расположена колонка ImagAddr. Обычно она пуста, указывая, что соответствующий модуль загружен по его предпочтительному базовому адресу. Так и должно быть для всех модулей. Однако, если в этой колонке присутствует адрес в скобках, значит, модуль загружен не по предпочтительному базовому адресу, и в колонке ImagAddr показывается базовый адрес, взятый из заголовка его файла на диске.

Ниже приведена информация о процессе devenv.exe, предоставленная моей программой ProcessInfo. Обратите внимание, что часть модулей загружена по предпочтительным базовым адресам, а часть — нет. Для последних сообщается один и тот же базовый адрес, 0x00400000; значит, автор этих DLL не подумал о проблемах модификации базовых адресов — пусть ему будет стыдно.



Связывание модулей

Модификация базовых адресов действительно очень важна и позволяет существенно повысить производительность всей системы. Но вы можете сделать еще больше. Допустим, вы должным образом модифицировали базовые адреса всех модулей своего приложения. Вспомните из главы 19, как загрузчик определяет адреса импортируемых идентификаторов: он записывает виртуальные адреса идентификаторов в раздел импорта EXE-модуля. Это позволяет, ссылаясь на импортируемые идентификаторы, адресоваться к нужным участкам в памяти.

Давайте поразмыслим. Сохраняя виртуальные адреса импортируемых идентификаторов в разделе импорта EXE-модуля, загрузчик записывает их на те страницы памяти, где содержится этот раздел. Здесь включается в работу механизм копирования при записи, и их копии попадают в страничный файл. И у нас опять та же проблема, что и при модификации базовых адресов: отдельные части проекции модуля периодически сбрасываются в страничный файл и вновь подгружаются из него. Кроме того, загрузчику приходится преобразовывать адреса всех импортируемых идентификаторов (для каждого модуля), на что может понадобиться немалое время.

Для ускорения инициализации и сокращения объема памяти, занимаемого вашим приложением, можно применить связывание модулей (module binding). Суть этой операции в том, что в раздел импорта модуля помещаются виртуальные адреса всех импортируемых идентификаторов. Естественно, она имеет смысл, только если проводится до загрузки модуля.

В Visual Studio есть еще одна утилита, Bind.exe. Информацию о том, как ею пользоваться, вы получите, запустив Bind.exe без ключей в командной строке. Вот что представляет собой эта функция:

```
usage: BIND [switches] image-names...
            [-?] display this message
            [-c] no caching of import dlls
            [-o] disable new import descriptors
            [-p dll search path]
```

```
[-s Symbol directory] update any associated .OBG file
[-u] update the image
[-v] verbose output
[-x image name] exclude this image from binding
[-y] allow binding on images located above 2G
```

Она описана в документации Platform SDK, и я не буду ее здесь детально рассматривать. Добавлю лишь, что в ней, как и в утилите Rebase, тоже нет ничего сверхъестественного: она просто вызывает функцию *BindImageEx* для каждого указанного файла.

```
BOOL BindlmageEx(
    DWORD dwFlags,                      // управляющие флаги
    PCSTR pszImageName,                 // полное имя обрабатываемого файла
    PCSTR pszDllPath,                   // путь для поиска образов файлов
    PCSTR pszSymbolPath,                // путь для поиска отладочной информации
    PIMAGEHLP_STATUS_ROUTINE pfnStatusRoutine);   // функция обратного вызова
```

Последний параметр, *StatusRoutine*, — адрес функции обратного вызова, к которой периодически обращается *BindImageEx*, позволяя отслеживать процесс связывания. Прототип функции обратного вызова должен выглядеть так:

```
BOOL WINAPI StatusRoutine(
    IMAGEHLP_STATUS_REASON Reason,      // причина неудачи
    PCSTR pszImageName,                 // полное имя обрабатываемого файла
    PCSTR pszDllName,                   // полное имя OLL
    ULONG_PTR VA,                       // вычисленный виртуальный адрес
    ULONG_PTR Parameter);               // дополнительные сведения (зависят от
                                        значения Reason)
```

Когда вы запускаете утилиту Bind, указывая ей нужный файл, она выполняет следующие операции.

1. Открывает раздел импорта указанного файла.
2. Открывает каждую DLL, указанную в разделе импорта, и просматривает ее заголовок, чтобы определить предпочтительный базовый адрес.
3. Отыскивает все импортируемые идентификаторы в разделе экспорта DLL.
4. Получает RVA (относительный виртуальный адрес) идентификатора, суммирует его с предпочтительным базовым адресом модуля и записывает полученное значение в раздел импорта обрабатываемого файла.
5. Вносит в раздел импорта модуля некоторую дополнительную информацию, включая имена всех DLL, с которыми связывается файл, и их временные метки.

В главе 19 мы исследовали раздел импорта Calc.exe с помощью утилиты DumpBin. В конце выведенного ею текста можно заметить информацию о связывании, добавленную при операции по п. 5. Вот эти строки:

```
Header contains the following bound import information:
    Bound to SHELL32.dll [4549BDB4] Thu Nov 02 10:43:16 2006
    Bound to ADVAPI32.dll [4549BCD2] Thu Nov 02 10:39:30 2006
    Bound to 0LEAUT32.dll [4549BD95] Thu Nov 02 10:42:45 2006
    Bound to ole32.dll [4549BD92] Thu Nov 02 10:42:42 2006
    Bound to ntdll.dll [4549BDC9] Thu Nov 02 10:43:37 2006
    Bound to KERNEL32.dll [4549BD80] Thu Nov 02 10:42:24 2006
    Bound to 6DI32.dll [4549BCD3] Thu Nov 02 10:39:31 2006
    Bound to USER32.dll [4549BDE0] Thu Nov 02 10:44:00 2006
    Bound to msvcrt.dll [4549BD61] Thu Nov 02 10:41:53 2006
```

Здесь видно, с какими модулями связан файл Calc.exe, а номер в квадратных скобках идентифицирует время создания каждого DLL-модуля. Это 32-разрядное значение расшифровывается и отображается за квадратными скобками в более привычном нам виде.

Утилита Bind использует два важных правила.

- При инициализации процесса все необходимые DLL действительно загружаются по своим предпочтительным базовым адресам. Вы можете соблюсти это правило, применив утилиту Rebase.
- Адреса идентификаторов в разделе экспорта остаются неизменными со времени последнего связывания. Загрузчик проверяет это, сравнивая временную метку каждой DLL со значением, сохраненным при операции по п. 5.

Конечно, если загрузчик обнаружит, что нарушено хотя бы одно из правил, он решит, что Bind не справилась со своей задачей, и самостоятельно модифицирует раздел импорта исполняемого модуля (по обычной процедуре). Но если загрузчик увидит, что модуль связан, нужные DLL загружены по предпочтительным базовым адресам и временные метки корректны, он фактически ничего делать не будет, и приложение сможет немедленно начать свою работу!

Кроме того, приложение не потребует лишнего места в страничном файле. И очень жаль, что многие коммерческие приложения поставляются без должной модификации базовых адресов и связывания.

Теперь вы знаете, что все модули приложения нужно связывать. Но вот вопрос: когда? Если вы свяжете модули в своей системе, вы привяжете их к системным DLL, установленным на вашем компьютере, а у пользователя могут быть установлены другие версии DLL. Поскольку вам заранее не известно, в какой операционной системе (Windows XP, Windows Server 2003 или Windows Vista) будет запускаться ваше приложение и какие сервисные пакеты в ней установлены, связывание нужно проводить в процессе установки приложения.

Естественно, если пользователь применяет конфигурацию с альтернативной загрузкой Windows XP и Windows Vista, то для одной из операционных систем модули будут связаны неправильно. Тот же эффект даст и обновление операционной системы установкой в ней сервисного пакета. Эту проблему ни вам, ни тем более пользователю решить не удастся. Майкрософт следовало бы поставлять с операционной системой утилиту, которая автоматически проводила бы повторное связывание всех модулей после обновления системы. Но, увы, такой утилиты нет.

Оглавление

ГЛАВА 21

Локальная память потока

Иногда данные удобно связывать с экземпляром какого-либо объекта. Например, чтобы сопоставить какие-то дополнительные данные с окном, применяют функции *SetWindowWord* и *SetWindowLong*. Локальная память потока (thread-local storage, TLS) позволяет связать данные и с определенным потоком (скажем, сопоставить с ним время его создания), а по завершении этого потока вычислить время его жизни.

TLS также используется в библиотеке C/C++. Но эту библиотеку разработали задолго до появления многопоточных приложений, и большая часть содержащихся в ней функций рассчитана на однопоточные программы. Наглядный пример — функция *_tcstok_s*. При первом вызове она получает адрес строки и запоминает его в собственной статической переменной. Когда при следующих вызовах *_tcstok_s* вы передаете ей NULL, она оперирует с адресом, записанным в своей переменной.

В многопоточной среде вероятна такая ситуация: один поток вызывает *_tcstok_s*, и, не успел он вызвать ее повторно, как к ней уже обращается другой. Тогда второй поток заставит функцию занести в статическую переменную новый адрес, неизвестный первому. И в дальнейшем первый поток, вызывая *_tcstok_s*, будет использовать строку, принадлежащую второму. Вот вам и «жучок», найти который очень трудно.

Чтобы устранить эту проблему, в библиотеке C/C++ теперь применяется механизм локальной памяти потока: за каждым потоком закрепляется свой строковый указатель, зарезервированный для *_tcstok_s*. Аналогичный механизм действует и для других библиотечных функций, в том числе *asctime* и *gmtime*.

Локальная память потока может быть той соломинкой, за которую придется ухватиться, если ваша программа интенсивно использует глобальные или статические переменные. К счастью, сейчас наметилась тенденция отхода от применения таких переменных и перехода к автоматическим (размещаемым в стеке) переменным и передаче данных через параметры функций.

И правильно: ведь расположенные в стеке переменные всегда связаны только с конкретным потоком.

Стандартная библиотека C существует уже долгие годы — это и хорошо, и плохо. Ее переделывали под многие компиляторы, и ни один из них без нее не стоил бы ломаного гроша. Программисты пользовались и будут пользоваться ею, а значит, прототипы и поведение функций вроде *_tcstok_s* останутся прежними. Но если бы эту библиотеку взялись перерабатывать сегодня, ее построили бы с учетом многопоточности и уж точно не стали бы применять глобальные и статические переменные.

В своих программах я стараюсь избегать глобальных переменных. Если же вы используете глобальные и статические переменные, советую проанализировать каждую из них и подумать, нельзя ли заменить ее переменной, размещаемой в стеке. Усилия окупятся сторицей, когда вы решите создать в программе дополнительные потоки; впрочем, и однопоточное приложение лишь выиграет от этого.

Хотя два вида TLS-памяти, рассматриваемые в этой главе, применимы как в приложениях, так и в DLL, они все же полезнее при разработке DLL, поскольку именно в этом случае вам не известна структура программы, с которой они будут связаны. Если же вы пишете приложение, то обычно знаете, сколько потоков оно создаст и для чего. Поэтому здесь еще можно как-то вывернуться. Но разработчик DLL ничего этого не знает. Чтобы помочь ему, и был создан механизм локальной памяти потока. Однако сведения, изложенные в этой главе, пригодятся и разработчику приложений.

Динамическая локальная память потока

Приложение работает с динамической локальной памятью потока, оперируя набором из четырех функций. Правда, чаще с ними работают DLL-, а не EXE-модули. На рис. 21-1 показаны внутренние структуры данных, используемые для управления TLS в Windows.

Каждый флаг выполняемого в системе процесса может находиться в состоянии FREE или INUSE, указывая, свободна или занята данная область локальной памяти потока (TLS-область). Майкрософт гарантирует доступность по крайней мере TLS_MINIMUM_A^ILABLE битовых флагов. Идентификатор TLS_MINIMUM_AVAILABLE определен в файле WinNT.h как 64. Но в Windows 2000 этот флаговый массив вмещает свыше 1000 элементов! Этого более чем достаточно для любого приложения.

Чтобы воспользоваться динамической TLS, вызовите сначала функцию TlsAlloc:

```
DWORD TlsAlloc();
```

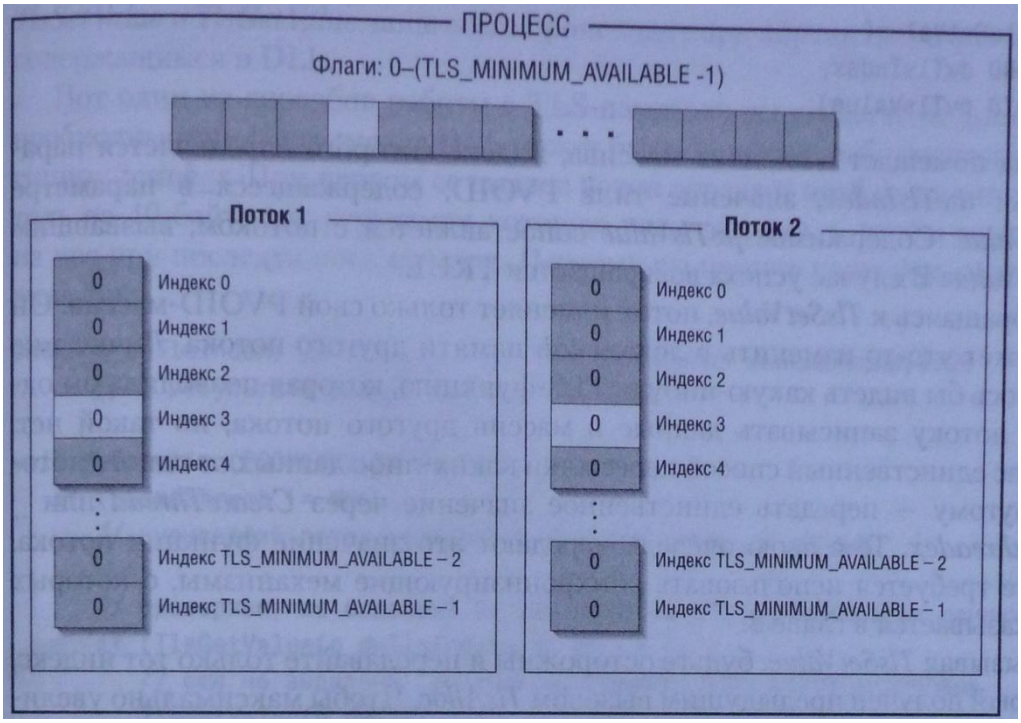


Рис. 21-1. *Внутренние структуры данных, предназначенные для управления локальной памятью потока*

Она заставляет систему сканировать битовые флаги в текущем процессе и искать флаг FREE. Отыскав, система меняет его на DWSE, а *TlsAlloc* возвращает индекс флага в битовом массиве. DLL (или приложение) обычно сохраняет этот индекс в глобальной переменной. Не найдя в списке флаг FREE, *TlsAlloc* возвращает код TLS_OUT_OF_NDEXES (определенный в файле WinBase.h как OxFFFFFFFF).

Когда *TlsAlloc* вызывается впервые, система узнает, что первый флаг — FREE, и немедленно меняет его на INUSE, а *TlsAlloc* возвращает 0. Вот 99 процентов того, что делает *TlsAlloc*. Об оставшемся одном проценте мы поговорим позже.

Создавая поток, система создает и массив из TLS_MINIMUM_AVAILABLE элементов — значений типа PVOID; она инициализирует его нулями и сопоставляет с потоком. Таким массивом (элементы которого могут принимать любые значения) располагает каждый лоток (рис. 21-1).

Прежде чем сохранить что-то в PVOID-массиве потока, выясните, какой индекс в нем доступен, — этой цели и служит предварительный вызов *TlsAlloc*. Фактически она резервирует какой-то элемент этого массива. Скажем, если возвращено значение 3, то в вашем распоряжении третий элемент PVOID-массива в каждом потоке данного процесса — не только в выполняемых сейчас, но и в тех, которые могут быть созданы в будущем.

Чтобы занести в массив потока значение, вызовите функцию *TlsSetValue*:

```
BOOL TlsSetValue(
    DWORD dwTlsIndex,
    PVOID pvTlsValue);
```

Она помещает в элемент массива, индекс которого определяется параметром *dwTbIndex*, значение типа PVOID, содержащееся в параметре *pvTbValue*. Содержимое *pvTlsValue* сопоставляется с потоком, вызвавшим *TbSetValue*. В случае успеха возвращается TRUE.

Обращаясь к *TbSetValue*, поток изменяет только свой PVOID-массив. Он не может что-то изменить в локальной памяти другого потока. Лично мне хотелось бы видеть какую-нибудь TLS-функцию, которая позволила бы одному потоку записывать данные в массив другого потока, но такой нет. Сейчас единственный способ пересылки каких-либо данных от одного потока другому — передать единственное значение через *CreateThread* или *_ beginthreadex*. Те в свою очередь передают это значение функции потока. Иначе требуется использовать синхронизирующие механизмы, о которых рассказывается в главе 8.

Вызывая *TkSetValue*, будьте осторожны и передавайте только тот индекс, который получен предыдущим вызовом *TlsAlloc*. Чтобы максимально увеличить быстродействие этих функций, Майкрософт отказалась от контроля ошибок. Если вы передадите индекс, не зарезервированный ранее *TlsAlloc*, система все равно запишет в соответствующий элемент массива значение, и тогда ждите неприятностей.

Для чтения значений из массива потока служит функция *TlsGetVahie*:

```
PVOID TlsGetValue(DWORD dwTlsIndex);
```

Она возвращает значение, сопоставленное с TLS-областью под индексом *dwThIndex*. Как и *TlsSetValue*, функция *TlsGetValue* обращается только к массиву, который принадлежит вызывающему потоку. Она тоже не контролиру-ет допустимость передаваемого индекса.

Когда необходимость в TLS-области у всех потоков в процессе отпадет, вызовите *TlsFree*:

```
BOOL TlsFree(DWORD dwTlsIndex);
```

Эта функция просто сообщит системе, что данная область больше не нужна. Флаг INUSE, управляемый массивом битовых флагов процесса, установится как FREE, и в будущем, когда поток еще раз вызовет *TlsAlloc*, этот участок памяти окажется вновь доступен. *TlsFree* возвращает TRUE, если вызов успешен. Попытка освобождения невыделенной TLS-области даст ошибку.

Использование динамической TLS

Обычно, когда в DLL применяется механизм TLS-памяти, вызов *DllMain* со значением DLL_PROCESS_ATTACH заставляет DLL обратиться к *TlsAlloc*, а вызов *DllMain* со значением DLL_PROCESS_DETACH — к *TlsFree*. Вызовы

TlsSetValue и *TlsGetValue* чаще всего происходят при обращении к функциям, содержащимся u DLL.

Вот один из способов работы с TLS-памятью: вы создаете се только по необходимости. Например, в DLL может быть функция, работающая аналогично *_t_stok_s*. При первом се вызове поток передаст этой функции указатель на 40-байтовую структуру, которую надо сохранить, чтобы ссылаться на нее при последующих вызовах. Поэтому вы пишете свою функцию, скажем, так:

```
DWORD g_dwTlsIndex; // считаем, что эта переменная инициализируется
    // в результате вызова функции TlsAlloc
...
void MyFunction(PSOMESTRUCT pSomeStruct) {
    if (pSomeStruct != NULL) {
        // вызывающий поток передает в функцию какие-то данные

        // проверяем, не выделена ли уже область для хранения этих данных
        if (TlsGetValue(g_dwTlsIndex) == NULL) {
            // еще не выделена; функция вызывается этим потоком впервые
            TlsSetValue(g_dwTlsIndex,
                HeapAlloc(GetProcessHeap(), 0, sizeof(*pSomeStruct));
        }

        // память уже выделена; сохраняем только что переданные значения
        memcpy(TlsGetValue(g_dwTlsIndex), pSomeStruct,
            sizeof(*pSomeStruct));

    } else {
        // вызывающий код уже передал функции данные;
        // теперь что-то делаем с ними

        // получаем адрес записанных данных
        pSomeStruct = (PSOMESTRUCT) TisOetValue(g.dwTlsIndex);

        // на эти данные указывает pSomeStruct; используем ее
        …
    }
}
```

Если поток приложения никогда не вызовет *MyFunction*, то и блок памяти никогда не будет выделен.

Если вам показалось, что 64 TLS-области — слишком много, напомню: приложение может динамически подключать несколько DLL. Одна DLL займет, допустим, 10 TLS-иидексов, вторая — 5 и т. д. Так что это вовсе не много — напротив, стремитесь к тому, чтобы DLL использовала минимальное число TLS-индексов. И для этого лучше всего применять метод, показанный на примере функции *MyFunction*. Конечно, я могу сохранить 40-бай-

товую структуру в 10 TLS-индексах, но тогда не только будет попусту расходоваться TLS-массив, но и затруднится работа с данными. Гораздо эффективнее выделить отдельный блок памяти для данных, сохранив указатель на него в одном TLS-индексе, — именно так и делается в *MyFunction*. Как я уже упомянул, в Windows количество TLS-областей увеличено и превышает 64. Майкрософт пошла на это из-за того, что многие разработчики слишком бесцеремонно использовали TLS-области и их не хватало другим DLL.

Теперь вернемся к тому единственному проценту, о котором я обещал рассказать, рассматривая *ThAlloc*. Взгляните на фрагмент кода:

```
DWORD dwTlsIndex;
PVOID pvSomeValue;
…
dwTlsIndex = TlsAlloc();
TlsSetValue(dwTlsIndex, (PV0ID) 12345);
TlsFree(dwTlsIndex);

// допустим, значение dwTlsIndex, возвращенное после этого вызова TlsAlloc,
// идентично индексу, полученному при предыдущем вызове TlsAlloc
dwTlsIndex = TlsAlloc();

pvSomeValue = TlsGetValue(dwTlsIndex);
```

Как вы думаете, что содержится в *pvSomeValue* после выполнения этого кода? 12345? Нет — нуль. Прежде чем вернуть управление, *TlsAlloc* «проходит» по всем потокам в процессе и заносит 0 по только что выделенному индексу в массив каждого потока. И прекрасно! Ведь не исключено, что приложение вызовет *LoadLibrary*, чтобы загрузить DLL, а последняя — *TlsAlloc*, чтобы зарезервировать какой-то индекс. Далее поток может обратиться к *FreeLibrary* и удалить DLL. Последняя должна освободить выделенный ей индекс, вызвав *TlsFree*, но кто знает, какие значения код DLL занес в тот или иной TLS-массив? В следующее мгновение поток вновь вызывает *LoadLibrary* и загружает другую DLL, которая тоже обращается к *TlsAlloc* и получает тот же индекс, что и предыдущая DLL. И если бы *TlsAlloc* не делала того, о чем я упомянул в самом начале, поток мог бы получить старое значение элемента, и программа стала бы работать некорректно.

Допустим, DLL, загруженная второй, решила проверить, выделена ли какому-то потоку локальная память, и вызвала *TlsGetValue*, как в предыдущем фрагменте кода. Если бы *TlsAlloc* не очищала соответствующий элемент в массиве каждого потока, то в этих элементах оставались бы старые данные от первой DLL. И тогда было бы вот что. Поток обращается к *MyFunction*, а та — в полной уверенности, что блок памяти уже выделен, — вызывает *memcpy* и таким образом копирует новые данные в ту область, которая, как ей кажется, и является выделенным блоком. Результат мог бы быть катастрофическим. К счастью, *TlsAlloc* инициализирует элементы массива, и такое просто немыслимо.

Статическая локальная память потока

Статическая локальная память потока основана па той же концепции, что и динамическая, — она предназначена для того, чтобы с потоком можно было сопоставить те или иные данные. Однако статическую TLS использовать гораздо проще, так как при этом не нужно обращаться к каким-либо функциям.

Возьмем такой пример: вы хотите сопоставлять стартовое время с каждым потоком, создаваемым программой. В этом случае нужно лить объявить переменную для хранения стартового времени:

```
_declspec(thread) DWORD gt_dwStartTime = 0;
```

Префикс *__declspec(thread)* — модификатор, поддерживаемый компилятором Microsoft Visual C++. Он сообщает компилятору, что соответствующую переменную следует поместить в отдельный раздел EXE- или DLL-файла. Переменная, указываемая за *__declspec(thread),* должна быть либо глобальной, либо статической внутри (или вне) функции. Локальную переменную с модификатором *__declspec(thread)* объявить нельзя. Но это не должно вас беспокоить, ведь локальные переменные и так связаны с конкретным потоком. Кстати, глобальные TLS-переменные я помечаю префиксом *gt_*, а статические — *st_*.

Обрабатывая программу, компилятор выносит все TLS-переменные в отдельный раздел, и вы вряд ли удивитесь, что этому разделу присваивается имя .tls. Компоновщик объединяет эти разделы из разных объектных модулей и создает в итоге один большой раздел .tls, помещаемый в конечный EXE- или DLL-файл.

Работа статической TLS строится на тесном взаимодействии с операционной системой. Загружая приложение в память, система отыскивает в EXE-файле раздел .tls и динамически выделяет блок памяти для хранения всех статических TLS-переменных. Всякий раз, когда ваша программа ссылается на одну из таких переменных, ссылка переадресуется к участку, расположенному в выделенном блоке памяти. В итоге компилятору приходится генерировать дополнительный код для ссылок на статические TLS-переменные, что увеличивает размер приложения и замедляет скорость его работы. В частности, на процессорах *x86* каждая ссылка на статическую TLS-переменную заставляет генерировать три дополнительные машинные команды.

Если в процессе создается другой поток, система выделяет еще один блок памяти для хранения статических переменных нового потока. Только что созданный поток имеет доступ лишь к своим статическим TLS-переменным, и не может обратиться к TLS-переменным любого другого потока.

Вот так в общих чертах и работает статическая TLS-память. Теперь посмотрим, что происходит при участии DLL. Ведь скорее всего ваша программма, использующая статические TLS-переменные, связывается с какой-нибудь DLL, в которой тоже применяются переменные этого типа. Загружая такую программу, система сначала определяет объем ее раздела .tls, а затем

добавляет эту величину к сумме размеров всех разделов .tls, содержащихся в DLL, которые связаны с вашей программой. При создании потоков система автоматически выделяет блок памяти, достаточно большой, чтобы в нем уместились все TLS-переменные, необходимые как приложению, так и неявно связываемым с ней DLL. Все так хорошо, что даже не верится!

И не верьте! Подумайте, что будет, если приложение вызовет *LoadLibrary* и подключит DLL, тоже содержащую статические TLS-переменные. Системе придется проверить потоки, уже существующие в процессе, и увеличить их блоки TLS-памяти, чтобы подогнать эти блоки под дополнительные требования, предъявляемые новой DLL. Ну а если вы вызовете *FreeLibrary* для выгрузки DLL со статическими TLS-переменными, системе придется ужать блоки памяти, сопоставленные с потоками в данном процессе. К счастью, Windows Vista прекрасно со всем этим справляется.

Оглавление

ГЛАВА 22

Внедрение DLL и перехват API-вызовов

В среде Windows каждый процесс получает свое адресное пространство. Указатели, используемые вами для ссылки на определенные участки памяти, — это адреса в адресном пространстве вашего процесса, и в нем *нельзя* создать указатель, ссылающийся на память, принадлежапгую другому процессу. Так, если в вашей программе есть «жучок», из-за которого происходит запись по случайному адресу, он не разрушит содержимое памяти, отведенной другим процессам.

Раздельные адресные пространства очень выгодны и разработчикам, и пользователям. Первым важно, что Windows перехватывает обращения к памяти по случайным адресам, вторым — что операционная система более устойчива и сбой одного приложения не приведет к краху другого или самой системы. Но, конечно, за надежность приходится платить: написать программу, способную взаимодействовать с другими программами или манипулировать другими процессами, теперь гораздо сложнее.

Вот ситуации, в которых требуется прорыв за границы процессов и доступ к адресному пространству другого процесса:

- создание подкласса окна, порожденного другим процессом;
- получение информации для отладки (например, чтобы определить, какие DLL используются другим процессом);
- установка ловушек (hooks) в других процессах.

В этой главе я расскажу о нескольких механизмах, позволяющих внедрить (inject) какую-либо DLL в адресное пространство другого процесса, ваш код, попав в чужое адресное пространство, может устроить в нем настоящий хаос, поэтому хорошенько взвесьте, так ли вам необходимо это внедрение.

Пример внедрения DLL

Допустим, вы хотите создать подкласс от экземпляра окна, порожденного другим процессом. Это, как вы помните, позволит изменять поведение окна. Все, что от вас для этого требуется, — вызвать функцию *SetWindowLongPtr*, чтобы заменить адрес оконной процедуры в блоке памяти, принадлежащем окну, новым — указывающим на вашу функцию WndProc. В документации Platform SDK утверждается, что приложение не может создать подкласс окна другого Процесса. Это не совсем верно. Проблема создания подкласса окна из другого процесса на самом деле сводится к преодолению границ адресного пространства.

Вызывая *SetWindowLongPtr* для создания подкласса окна (как показано ниже), вы говорите системе, что все сообщения окну, на которое указывает *hwnd*, следует направлять не обычной оконной процедуре, а функции *MySubclassProc*.

```
SetWindowLongPtr(hWnd, GWLP_WNDPROC, MySubclassProc);
```

Иными словами, когда системе надо передать сообщение процедуре *WndProc* указанного окна, она находит ее адрес и вызывает напрямую. В нашем примере система видит, что с *окном сопоставлен* адрес функции *MySubclassProc*, и поэтому вызывает именно ее, а не исходную оконную процедуру.

Проблема с созданием подкласса окна, принадлежащего другому процессу, состоит в том, что процедура подкласса находится в чужом адресном пространстве. Упрощенная схема приема сообщений оконной процедурой представлена на рис. 22-1. Процесс A создает окно. На адресное пространство этого процесса проецируется файл User32.dll. Эта проекция User32.dll отвечает за прием и диспетчеризацию сообщений (синхронных и асинхронных), направляемых любому из окон, созданных потоками процесса A. Обнаружив какое-то сообщение, она определяет адрес процедуры *WndProc* окна и вызывает ее, передавая описатель окна, сообщение и параметры *wParam* и *lParam*. Когда *WndProc* обработает сообщение, User32.dll вернется в начало цикла и будет ждать следующее оконное сообщение.

Теперь допустим, что процесс B хочет создать подкласс окна, порожденного одним из потоков процесса A. Сначала код процесса B должен определить описатель этого окна, что можно сделать самыми разными способами. В примере на рис. 22-1 поток процесса B просто вызывает *FindWindow*, затем — *Set WindowLongPtr*, пытаясь изменить адрес процедуры *WndProc* окна. Обратите внимание: *пытаясь*. Этот вызов не даст ничего, кроме NULL. Функция *SetWindowLongPtr* просто проверяет, не хочет ли процесс изменить адрес *WndProc* окна, созданного другим процессом, и, если да, игнорирует вызов.

А если бы функция *SetWindowLongPtr* могла изменить адрес *WndProc*? Система тогда связала бы адрес процедуры *MySubclassProc* с указанным окном. Затем при посылке сообщения этому окну код User32 в процессе A из-

лек бы данное сообщение, получил адрес *MySubclassProc* и попытался бы вызвать процедуру по этому адресу. Но это привело бы к крупным неприятностям, так как *MySubclassProc* находится в адресном пространстве процесса B, а активен — процесс A. Очевидно, если бы User32 обратился по данному адресу, то на самом деле он обратился бы к какому-то участку памяти в адресном пространстве процесса A, что, естественно, привело бы к нарушению доступа к памяти.

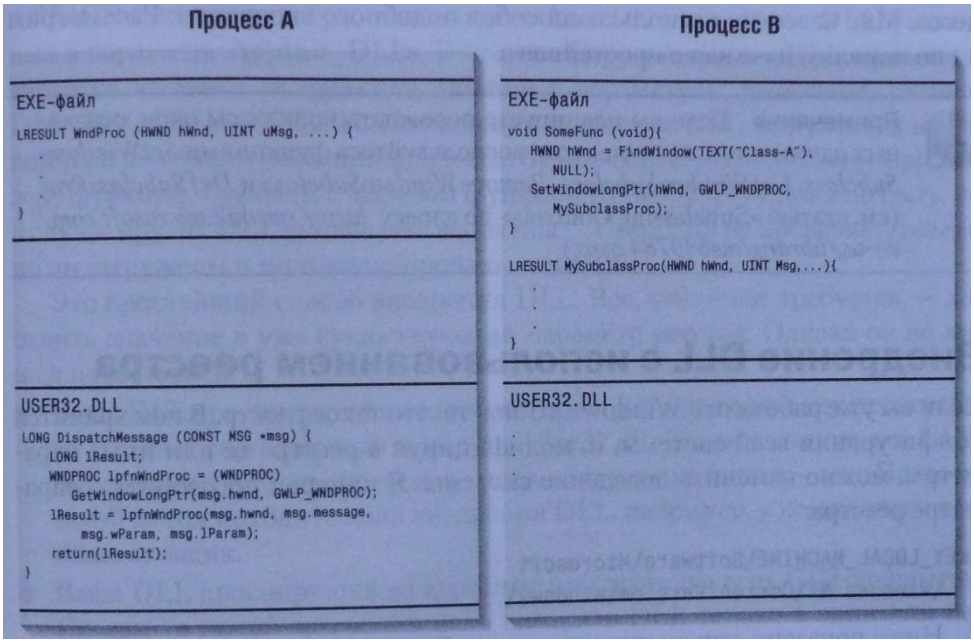


Рис. 22-1. *Поток процесса B пытается создать подкласс окна, сформированного потоком процесса A*

Чтобы избежать этого, было бы неплохо сообщить системе, что MySubclassProc находится в адресном пространстве процесса B, и тогда она переключила бы контекст перед вызовом процедуры подкласса. Увы, по ряду причин такая функциональность в системе не реализована.

- Подклассы окон, созданных потоками других процессов, порождаются весьма редко. Большинство приложений делает это лишь применительно к собственным окнам, и архитектура памяти в Windows этому не препятствует.
- Переключение активных процессов отнимает слишком много процессорного времени.
- Код MySubclassProc должен был бы выполняться потоком процесса B, но каким именно — новым или одним из существующих?
- Как User32.dll узнает, с каким процессом связан адрес оконной процедуры?

Поскольку удачных решений этих проблем нет, Майкрософт предпочла запретить функции SetWindowLongPtr замену процедуры окна, созданного другим процессом.

Тем не менее порождение подкласса окна, созданного чужим процессом, возможно: нужно просто пойти другим путем. Ведь на самом деле проблема не столько в создании подкласса, сколько в закрытости адресного пространства процесса. Если бы вы могли как-то поместить код своей оконной процедуры в адресное пространство процесса A, это позволило бы вызвать *SetWindowLongPtr* и передать ей адрес *MySubclassProc* в процессе A. Я называю такой прием внедрением (injecting) DLL в адресное пространство процесса. Мне известно несколько способов подобного внедрения. Рассмотрим их по порядку, начиная с простейшего.

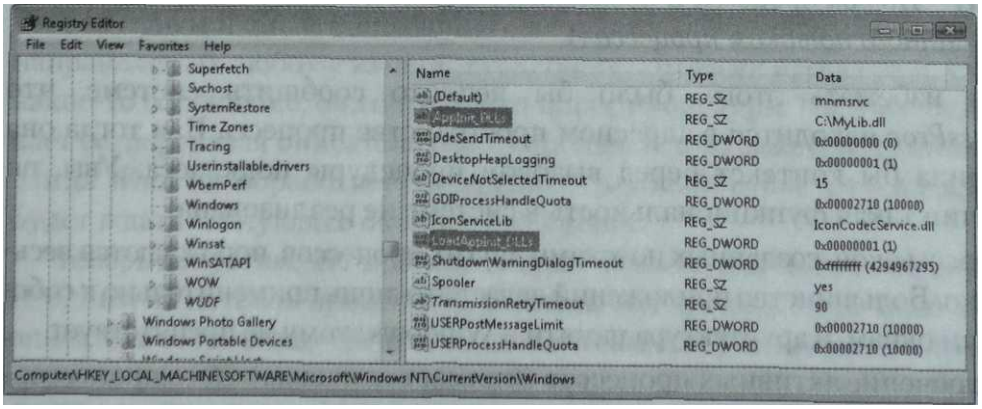
> **Примечание.** Если вы планируете порождать подклассы окон, созданных одним и тем же процессом, воспользуйтесь функциями *SetWindowSubclass*, *GetWindowSubclass*, *RemoveWindowSubclass* и *DefSubclassProc* (см. статью «Subclassing Controls» по адресу http://msdn2.microsoft.com/en-us/library/ms649784.aspx).

Внедрение DLL с использованием реестра

Если вы уже работали с Windows, то знаете, что такое реестр. В нем хранится конфигурация всей системы, и, модифицируя в реестре те или иные параметры, можно изменить поведение системы. Я намерен поговорить о параметре реестра:

```
HKEY_LOCAL_MACHINE\Software\Microsoft
    \Windows NT\CurrentVersion\Windows\
```

Ниже показано, как он выгляди в окне Registry Editor:



Список параметров в разделе реестра, где находится *AppInit_DLLs*, можно просмотреть с помощью программы Registry Editor (Редактор реестра). Значением параметра AppInitJDLLs может быть как имя одной DLL (с указанием пути доступа), так и имена нескольких DLL, разделенных пробелами или запятыми. Поскольку пробел используется здесь в качестве разделителя, в именах файлов не должно быть пробелов. Система считывает, путь

только первой DLL в списке — пути остальных DLL игнорируются, поэтому лучше размещать свои DLL в системном каталоге Windows, чтобы не указывать пути. Как видите, я указал в параметре *AppInit_DLLs* только одну DLL и задал путь к ней: C:\MyLib.dll.

При следующей перезагрузке компьютера Windows сохранит значение этого параметра. Далее, когда User32.dll будет спроецирован на адресное пространство процесса, этот модуль получит уведомление DLL_PROCESS_ ATTACH и после его обработки вызовет *LoadLibrary* для всех DLL, указанных в параметре AppInit_DLLs. В момент загрузки каждая DLL инициализируется вызовом ее функции *DllMain* с параметром *fwdReason*, равным DLL_PROCESS_ATTACH. Поскольку внедряемая DLL загружается на такой ранней стадии создания процесса, будьте особенно осторожны при вызове функций. Проблем с вызовом функций Kernel32.dll не должно быть, но в случае других DLL они вполне вероятны — User32.dll не проверяет, успешно ли загружены и инициализированы эти DLL.

Это простейший способ внедрения DLL. Все, что от вас требуется, — добавить значение в уже существующий параметр реестра. Однако он не лишен недостатков.

- Ваша DLL проецируется на адресные пространства только тех процессов, на которые спроецирован и модуль User32.dll. Его используют все GUI-приложения, но большинство программ *консольного типа* — нет. Поэтому такой метод не годится для внедрения DLL, например, в компилятор или компоновщик.
- Ваша DLL проецируется на адресные пространства всех GUI-процессов. Но вам-то почти наверняка надо внедрить DLL только в один или несколько определенных процессов. Чем больше процессов попадет «под тень» такой DLL, тем выше вероятность аварийной ситуации. Ведь теперь ваш код выполняется потоками этих процессов, и, если он зациклится или некорректно обратится к памяти, вы повлияете на поведение и устойчивость соответствующих процессов. Поэтому лучше внедрять свою DLL в как можно меньшее число процессов.
- Ваша DLL проецируется на адресное пространство каждого GUI-процесса в течение всей его жизни. Тут есть некоторое сходство с предыдущей проблемой. Желательно не только внедрять DLL в минимальное число процессов, но и проецировать ее на эти процессы как можно меньшее время. Допустим, вы хотите создать подкласс главного окна Word Pad в тот момент, когда пользователь запускает ваше приложение. Естественно, пока пользователь не откроет ваше приложение, внедрять DLL в адресное пространство Word Pad не требуется. Когда пользователь закроет ваше приложение, целесообразно отменить переопределение оконной процедуры Word Pad. И в этом случае DLL тоже незачем «держать» в адресном пространстве Word Pad. Так что лучшее решение — внедрять DLL только на то время, в течение которого она действительно нужна конкретной программе.

Внедрение DLL с помощью ловушек

Внедрение DLL в адресное пространство процесса возможно и с применением ловушек. Чтобы они работали так же, как и в 16-разрядной Windows, Майкрософт пришлось создать механизм, позволяющий внедрять DLL в адресное пространство другого процесса. Рассмотрим его на примере.

Процесс А (вроде утилиты Spy++) устанавливает ловушку WH_ GETMESSAGE и наблюдает за сообщениями, которые обрабатываются окнами в системе, Ловушка устанавливается вызовом *SetWindowsHookEx:*

```
HHOOK hHook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, hInstDll, 0);
```

Аргумент WH_GETMESSAGE определяет тип ловушки, а параметр *GetMsgProc* — адрес функции (в адресном пространстве вашего процесса), которую система должна вызывать всякий раз, когда окно собирается обработать сообщение. Параметр *hinstDll* идентифицирует DLL, содержащую функцию *GetMsgProc.* В Windows значение *hinstDll* для DLL фактически задает адрес в виртуальной памяти, по которому DLL спроецирована на адресное пространство процесса. И, наконец, последний аргумент, 0, указывает поток, для которого предназначена ловушка. Поток может вызвать *SetWindowsHookEx* и передать ей идентификатор другого потока в системе. Передавая 0, мы сообщаем системе, что ставим ловушку для всех существующих в ней GUT-потоков.

Теперь посмотрим, как все это действует:

1. Поток процесса В собирается направить сообщение какому-либо окну.
2. Система проверяет, не установлена ли для данного потока ловушка WH_GETMESSAGE.
3. Затем выясняет, спроецирована ли DLL, содержащая функцию *GetMsgProc,* на адресное пространство процесса В.
4. Если указанная DLL еще не спроецирована, система отображает ее на адресное пространство процесса В и увеличивает счетчик блокировок (lock count) проекции DLL в процессе В на 1.
5. Система проверяет, не совпадают ли значения *hinstDll* этой DLL, относящиеся к процессам А и В. Если *hinstDll* в обоих процессах одинаковы, то и адрес *GetMsgProc* в этих процессах тоже одинаков. Тогда система может просто вызвать *GetMsgProc* в адресном пространстве процесса В. Если же *hinstDll* различны, система определяет адрес функции *GetMsgProc* в адресном пространстве процесса В по формуле:

   ```
   GetMsgProc B =hInstDll B + (GetMsgProc A - hInstDll A)
   ```

   Вычитая *hinstDll* А из *GetMsgProc A,* вы получаете смещение (в байтах) адреса функции *GetMsgProc.* Добавляя это смещение к *hinstDll B,* вы получаете адрес *GetMsgPtvc,*. соответствующий проекции DLL в адресном пространстве процесса В.

6. Счетчик блокировок проекции DLL в процессе B увеличивается на 1.
7. Вызывается *GetMsgProc* в адресном пространстве процесса B.
8. После возврата из *GetMsgProc* счетчик блокировок проекции DLL в адресном пространстве процесса B уменьшается на 1.

Кстати, когда система внедряет или проецирует DLL, содержащую функцию фильтра ловушки, проецируется вся DLL, а не только эта функция. А значит, потокам, выполняемым в контексте процесса B, теперь доступны все функции такой DLL.

Итак, чтобы создать подкласс окна, сформированного потоком другого процесса, можно сначала установить ловушку WH_GETMESSAGE для этого потока, а затем — когда будет вызвана функция *GetMsgProc* — обратиться к *SetWindowLongPtr* и создать подкласс. Разумеется, процедура подкласса должна быть в той же DLL, что и *GetMsgProc*.

В отличие от внедрения DLL с помощью реестра этот способ позволяет в любой момент отключить DLL от адресного пространства процесса, для чего достаточно вызвать:

```
BOOL UnhookWindowsHookEx(HHOOK hHook);
```

Когда поток обращается к этой функции, система просматривает внутренний список процессов, в которые ей пришлось внедрить данную DLL, и уменьшает счетчик ее блокировок на 1. Как только этот счетчик обнуляется, DLL автоматически выгружается. Вспомните: система увеличивает его непосредственно перед вызовом *GetMsgProc* (см. выше п. 6). Это позволяет избежать нарушения доступа к памяти. Если бы счетчик не увеличивался, то другой поток мог бы вызвать *UnhookWindowsHookEx* в тот момент1, когда поток процесса B пытается выполнить код *GetMsgProc*.

Все это означает, что нельзя создать подкласс окна и тут же убрать ловушку — она должна действовать в течение всей жизни подкласса.

Утилита для сохранения позиций элементов на рабочем столе

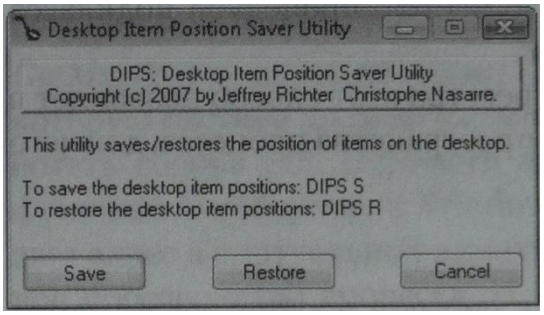
Эта утилита (DIPS.exe), использует ловушки окон для внедрения DLL в адресное пространство Explorer.exe. Файлы исходного кода и ресурсов этой программы и DLL находятся в каталогах 22-DIPS и 22-DIPSLib внутри архива, доступного на веб-сайте поддержки этой книги.

Компьютер я использую в основном для работы, и, на мой взгляд, самое оптимальное в этом случае разрешение экрана — 1400 x 1050. Иногда я демонстрирую презентации через видеопроектор, рассчитанный на более низкое разрешение. Когда у меня появляется настроение поиграть, приходится открывать апплет Display в Control Panel и устанавливать разрешение, необходимое проектору, а закончив игру, вновь возвращаться в Display и восстанавливать разрешение 1400 x 1050.

Возможность изменять экранное разрешение «на лету» — очень удобная функция Windows. Единственное, что мне не по душе, — при смене экран-

ного разрешения не сохраняются позиции ярлыков на рабочем столе. У меня на рабочем столе масса ярлыков для быстрого доступа к часто используемым программам и файлам. Стоит мне сменить разрешение, размеры рабочего стола изменяются, и ярлыки перестраиваются так, что уже ничего не найдешь. А когда я восстанавливаю прежнее разрешение, ярлыки так и остаются вперемешку. Чтобы навести порядок, приходится вручную перемешать каждый ярлык на свое место — очень интересное занятие!

В общем, мне это так осточертело, что я придумал утилиту, сохраняющую позиции элементов на экране (Desktop Item Position Saver, DIPS). DIPS состоит из крошечного исполняемого файла и компактной DLL. После запуска исполняемого файла появляется следующее окно.



В этом окне поясняется, как работать с утилитой. Когда она запускается с ключом 5 в командной строке, в реестре создается подраздел:

```
HKEY_CURRENT_USER\Software\Wintellect\Desktop Item Position Saver
```

куда добавляется по одному параметру на каждый ярлык, расположенный на вашем рабочем столе. Значение каждого параметра — позиция соответствующего ярлыка. Утилиту DIPS следует запускать с ключом S перед установкой более низкого экранного разрешения. Всласть наигравшись и восстановив нормальное разрешение, вновь запустите DIPS — на этот раз с ключом *R.* Тогда DIPS откроет соответствующий подраздел реестра и восстановит для каждого объекта рабочего стола его исходную позицию.

На первый взгляд утилита DIPS тривиальна и очень проста в реализации. Вроде бы только и надо, что получить описатель элемента управления ListView рабочего стола, заставить его (послав соответствующие сообщения) перечислить все ярлыки и определить их координаты, а потом сохранить полученные данные в реестре. Но попробуйте, и вы убедитесь, что все не так просто. Проблема в том, что большинство оконных сообщений для стандартных элементов управления (например, LVMGETITEM и LVM_ GETITEMPOSITION) не может преодолеть границы процессов. Почему?

Сообщение LVM_GETITEM требует, чтобы вы передали в параметре *lParam* адрес структуры LV_ITEM. Поскольку ее адрес имеет смысл лишь в адресном пространстве процесса — отправителя сообщения, процесс-приемник не может безопасно использовать его. Поэтому, чтобы DIPS работала так, как было обещано, в Explorer.exe надо внедрить код, посылающий сообщения LVM_GETITEM и LVMGETITEMPO SITI ON элементу управления ListView рабочего стола.

Примечание. В отличие от новых стандартных элементов управления встроенные (кнопки, поля, метки, списки, комбинированные списки и т. д.) позволяют передавать оконные сообщения через границы процессов. Например, окну списка, созданному каким-нибудь потоком другого процесса, можно послать сообщение LB_GETTEXT, чей параметр *lParam* указывает на строковый буфер в адресном пространстве процесса-отправителя. Это срабатывает, потому что операционная система специально проверяет, не отправлено ли сообщение LB_GETTEXT, и, если да, сама создает проецируемый в память файл и копирует строковые данные из адресного пространства одного процесса в адресное пространство другого.

Почему Майкрософт решила по-разному обрабатывать встроенные и новые элементы управления? Дело в том, что в 16-разрядной Windows, в которой все приложения выполняются в едином адресном пространстве, любая программа могла послать сообщение LB_GETTEXT окну, созданному другой программой. Чтобы упростить перенос таких приложений в Win32, Майкрософт и пошла на эти ухищрения. А поскольку в 16-разрядной Windows нет новых элементов управления, то проблемы их переноса тоже нет, и Майкрософт ничего подобного для них делать не стала.

Сразу после запуска DIPS получает описатель окна элемента управления List View рабочего стола:

```
// окно ListView рабочего стола - "внук" окна ProgMan
hWndLV = GetFirstChild(
    GetFirstChild(FindWindow(TEXT("ProgMan"), NULL)));
```

Этот код сначала ищет окно класса ProgMan. Даже несмотря на то что никакой Program Manager не запускается, новая оболочка по-прежнему создает окно этого класса — для совместимости с приложениями, рассчитанными на старые версии Windows. У окна ProgMan единственное дочернее окно класса SHELLDLL_DetView, у которого тоже одно дочернее окно — класса SysListView32. Оно-то и служит элементом управления List View рабочего стола. (Кстати, всю эту информацию я *выудил* благодаря Spy++.)

Получив описатель окна List View, я определяю идентификатор создавшего его потока, для чего вызываю *GetWindowThreadProcessId*. Этот идентификатор я передаю функции *SetDIPSHook*, реализованной в DIPSLib.cpp. Последняя функция устанавливает ловушку WH_GETMESSAGE для данного потока и вызывает:

```
PostThreadMessage(dwThreadId, WM_NULL, 0, 0);
```

чтобы разбудить поток Windows Explorer. Поскольку для него установлена ловушка WH_GETMESSAGE, операционная система автоматически внедряет мою DIPSLib.dll в адресное пространство Explorer и вызывает мою функцию *GetMsgProc*. Та сначала проверяет, впервые ли она вызвана, и, если да, создает скрытое окно с заголовком «Richter DIPS». Возьмите на заметку,

что это окно создается потоком, принадлежащим Explorer. Пока окно создается, поток DIPS.exe возвращается из функции SetDIPSHook и вызывает:

```
GetMessage(&msg, NULL, 0, 0);
```

Этот вызов «усыпляет» поток до появления в очереди какого-нибудь сообщения. Хотя DIPS.exe сам не создает ни одного окна, у него все же есть очередь сообщений, и они помещаются туда исключительно в результате вызовов *PostThreadMessage*. Взгляните на код *GetMsgProc* в DIPSLib.cpp: сразу после обращения к *CreateDialog* стоит вызов *PostThreadMessage*, который вновь пробуждает поток DIPS.exe. Идентификатор потока сохраняется в разделяемой переменной внутри функции *SetDIPSHook*.

Очередь сообщений я использую для синхронизации потоков. В э4ом нет ничего противозаконного, и иногда гораздо проще синхронизировать потоки именно так, не прибегая к объектам ядра — мьютексам, семафорам, событиям и т. д. (В Windows очень богатый API; пользуйтесь этим.)

Когда поток DIPS.exe пробуждается, он узнает, что серверное диалоговое окно уже создано, и обращается к *FindWindow*, чтобы получить его описатель. С этого момента для организации взаимодействия между клиентом (утилитой DIPS) и сервером (скрытым диалоговым окном) можно использовать механизм оконных сообщений. Поскольку это диалоговое окно создано потоком, выполняемым в контексте процесса Explorer, нас мало что ограничивает в действиях с Explorer.

Чтобы сообщить своему диалоговому окну сохранить или восстановить позиции ярлыков на экране, достаточно послать сообщение:

```
// сообщаем окну DIPS, с каким окном ListView работать
// и что делать: сохранять или восстанавливать позиции ярлыков
SendMessage(hWndDIPS, WM_APP, (WPARAM) hWndLV, bSave);
```

Процедура диалогового окна проверяет сообщение WM_APP. Когда она принимает это сообщение, параметр *wParam* содержит описатель нужного элемента управления ListView, а *lParam* — булево значение, определяющее, сохранять текущие позиции ярлыков в реестре или восстанавливать.

Так как здесь используется *SendMessage*, а не *PostMessage*, управление не передается до завершения операции. Если хотите, определите дополнительные сообщения для процедуры диалогового окна — это расширит возможности программы в управлении Explorer. Закончив, я завершаю работу сервера, для чего посылаю ему сообщение WM_CLOSE, которое говорит диалоговому окну о необходимости самоуничтожения.

Наконец, перед своим завершением DIPS вновь вызывает SetDIPSHook, но на этот раз в качестве идентификатора потока передается 0. Получив нулевое значение, функция снимает ловушку WH_GETMESSAGE. А когда ловушка удаляется, операционная система автоматически выгружает DIPSLib.dll из адресного пространства процесса Explorer, и это означает, что теперь процедура диалогового окна больше не принадлежит данному адрес-

ному пространству. Поэтому важно уничтожить диалоговое окно заранее — до снятия ловушки. Иначе очередное сообщение, направленное диалоговому окну, вызовет нарушение доступа. И тогда Explorer будет аварийно завершен операционной системой — с внедрением DLL шутки плохи!

DIPSLib.cpp

```
/******************************************************************************
Module:  DIPS.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"
#include "..\22-DIPSLib\DIPSLib.h"


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_DIPS);
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   switch (id) {
      case IDC_SAVE:
      case IDC_RESTORE:
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
```

```
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   // Convert command-line character to uppercase.
   CharUpperBuff(pszCmdLine, 1);
   TCHAR cWhatToDo = pszCmdLine[0];

   if ((cWhatToDo != TEXT('S')) && (cWhatToDo != TEXT('R'))) {

      // An invalid command-line argument; prompt the user.
      cWhatToDo = 0;
   }

   if (cWhatToDo == 0) {
      // No command-line argument was used to tell us what to
      // do; show usage dialog box and prompt the user.
      switch (DialogBox(hInstExe, MAKEINTRESOURCE(IDD_DIPS), NULL, Dlg_Proc)) {
         case IDC_SAVE:
            cWhatToDo = TEXT('S');
            break;

         case IDC_RESTORE:
            cWhatToDo = TEXT('R');
            break;
      }
   }

   if (cWhatToDo == 0) {
      // The user doesn't want to do anything.
      return(0);
   }

   // The Desktop ListView window is the grandchild of the ProgMan window.
   HWND hWndLV = GetFirstChild(GetFirstChild(
      FindWindow(TEXT("ProgMan"), NULL)));
   chASSERT(IsWindow(hWndLV));

   // Set hook that injects our DLL into the Explorer's address space. After
   // setting the hook, the DIPS hidden modeless dialog box is created. We
   // send messages to this window to tell it what we want it to do.
   chVERIFY(SetDIPSHook(GetWindowThreadProcessId(hWndLV, NULL)));
```

```
   // Wait for the DIPS server window to be created.
   MSG msg;
   GetMessage(&msg, NULL, 0, 0);

   // Find the handle of the hidden dialog box window.
   HWND hWndDIPS = FindWindow(NULL, TEXT("Wintellect DIPS"));

   // Make sure that the window was created.
   chASSERT(IsWindow(hWndDIPS));

   // Tell the DIPS window which ListView window to manipulate
   // and whether the items should be saved or restored.
   BOOL bSave = (cWhatToDo == TEXT('S'));
   SendMessage(hWndDIPS, WM_APP, (WPARAM) hWndLV, bSave);

   // Tell the DIPS window to destroy itself. Use SendMessage
   // instead of PostMessage so that we know the window is
   // destroyed before the hook is removed.
   SendMessage(hWndDIPS, WM_CLOSE, 0, 0);

   // Make sure that the window was destroyed.
   chASSERT(!IsWindow(hWndDIPS));

   // Unhook the DLL, removing the DIPS dialog box procedure
   // from the Explorer's address space.
   SetDIPSHook(0);

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

DIPSLib.cpp

```
/******************************************************************************
Module:  DIPSLib.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <WindowsX.h>
#include <CommCtrl.h>

#define DIPSLIBAPI __declspec(dllexport)
#include "DIPSLib.h"
#include "Resource.h"
```

```
//////////////////////////////////////////////////////////////////////////////


#ifdef _DEBUG
// This function forces the debugger to be invoked
void ForceDebugBreak() {
   __try { DebugBreak(); }
   __except(UnhandledExceptionFilter(GetExceptionInformation())) { }
}
#else
#define ForceDebugBreak()
#endif


//////////////////////////////////////////////////////////////////////////////


// Forward references
LRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam);

INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam);


//////////////////////////////////////////////////////////////////////////////


// Instruct the compiler to put the g_hHook data variable in
// its own data section called Shared. We then instruct the
// linker that we want to share the data in this section
// with all instances of this application.
#pragma data_seg("Shared")
HHOOK g_hHook = NULL;
DWORD g_dwThreadIdDIPS = 0;
#pragma data_seg()

// Instruct the linker to make the Shared section
// readable, writable, and shared.
#pragma comment(linker, "/section:Shared,rws")


//////////////////////////////////////////////////////////////////////////////


// Nonshared variables
HINSTANCE g_hInstDll = NULL;


//////////////////////////////////////////////////////////////////////////////


BOOL WINAPI DllMain(HINSTANCE hInstDll, DWORD fdwReason, PVOID fImpLoad) {

   switch (fdwReason) {

      case DLL_PROCESS_ATTACH:
```

```
         // DLL is attaching to the address space of the current process.
         g_hInstDll = hInstDll;
         break;

      case DLL_THREAD_ATTACH:
         // A new thread is being created in the current process.
         break;

      case DLL_THREAD_DETACH:
         // A thread is exiting cleanly.
         break;

      case DLL_PROCESS_DETACH:
         // The calling process is detaching the DLL from its address space.
         break;
   }
   return(TRUE);
}


////////////////////////////////////////////////////////////////////////////


BOOL WINAPI SetDIPSHook(DWORD dwThreadId) {

   BOOL bOk = FALSE;

   if (dwThreadId != 0) {
      // Make sure that the hook is not already installed.
      chASSERT(g_hHook == NULL);

      // Save our thread ID in a shared variable so that our GetMsgProc
      // function can post a message back to the thread when the server
      // window has been created.
      g_dwThreadIdDIPS = GetCurrentThreadId();

      // Install the hook on the specified thread
      g_hHook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, g_hInstDll,
         dwThreadId);

      bOk = (g_hHook != NULL);
      if (bOk) {
         // The hook was installed successfully; force a benign message to
         // the thread's queue so that the hook function gets called.
         bOk = PostThreadMessage(dwThreadId, WM_NULL, 0, 0);
      }
   } else {
```

```
      // Make sure that a hook has been installed.
      chASSERT(g_hHook != NULL);
      bOk = UnhookWindowsHookEx(g_hHook);
      g_hHook = NULL;
   }

   return(bOk);
}


///////////////////////////////////////////////////////////////////////////


LRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam) {

   static BOOL bFirstTime = TRUE;

   if (bFirstTime) {
      // The DLL just got injected.
      bFirstTime = FALSE;

      // Uncomment the line below to invoke the debugger
      // on the process that just got the injected DLL.
      // ForceDebugBreak();

      // Create the DIPS Server window to handle the client request.
      CreateDialog(g_hInstDll, MAKEINTRESOURCE(IDD_DIPS), NULL, Dlg_Proc);

      // Tell the DIPS application that the server is up
      // and ready to handle requests.
      PostThreadMessage(g_dwThreadIdDIPS, WM_NULL, 0, 0);
   }

   return(CallNextHookEx(g_hHook, nCode, wParam, lParam));
}


///////////////////////////////////////////////////////////////////////////


void Dlg_OnClose(HWND hWnd) {

   DestroyWindow(hWnd);
}


///////////////////////////////////////////////////////////////////////////


static const TCHAR g_szRegSubKey[] =
   TEXT("Software\\Wintellect\\Desktop Item Position Saver");


///////////////////////////////////////////////////////////////////////////
```

```
void SaveListViewItemPositions(HWND hWndLV) {

   int nMaxItems = ListView_GetItemCount(hWndLV);

   // When saving new positions, delete the old position
   // information that is currently in the registry.
   LONG l = RegDeleteKey(HKEY_CURRENT_USER, g_szRegSubKey);

   // Create the registry key to hold the info
   HKEY hkey;
   l = RegCreateKeyEx(HKEY_CURRENT_USER, g_szRegSubKey, 0, NULL,
      REG_OPTION_NON_VOLATILE, KEY_SET_VALUE, NULL, &hkey, NULL);
   chASSERT(l == ERROR_SUCCESS);

   for (int nItem = 0; nItem < nMaxItems; nItem++) {

      // Get the name and position of a ListView item.
      TCHAR szName[MAX_PATH];
      ListView_GetItemText(hWndLV, nItem, 0, szName, _countof(szName));

      POINT pt;
      ListView_GetItemPosition(hWndLV, nItem, &pt);

      // Save the name and position in the registry.
      l = RegSetValueEx(hkey, szName, 0, REG_BINARY, (PBYTE) &pt, sizeof(pt));
      chASSERT(l == ERROR_SUCCESS);
   }
   RegCloseKey(hkey);
}


////////////////////////////////////////////////////////////////////////////


void RestoreListViewItemPositions(HWND hWndLV) {

   HKEY hkey;
   LONG l = RegOpenKeyEx(HKEY_CURRENT_USER, g_szRegSubKey,
      0, KEY_QUERY_VALUE, &hkey);
   if (l == ERROR_SUCCESS) {

      // If the ListView has AutoArrange on, temporarily turn it off.
      DWORD dwStyle = GetWindowStyle(hWndLV);
      if (dwStyle & LVS_AUTOARRANGE)
         SetWindowLong(hWndLV, GWL_STYLE, dwStyle & ~LVS_AUTOARRANGE);

      l = NO_ERROR;
      for (int nIndex = 0; l != ERROR_NO_MORE_ITEMS; nIndex++) {
```

```
         TCHAR szName[MAX_PATH];
         DWORD cbValueName = _countof(szName);

         POINT pt;
         DWORD cbData = sizeof(pt), nItem;

         // Read a value name and position from the registry.
         DWORD dwType;
         l = RegEnumValue(hkey, nIndex, szName, &cbValueName,
            NULL, &dwType, (PBYTE) &pt, &cbData);

         if (l == ERROR_NO_MORE_ITEMS)
            continue;

         if ((dwType == REG_BINARY) && (cbData == sizeof(pt))) {
            // The value is something that we recognize; try to find
            // an item in the ListView control that matches the name.
            LV_FINDINFO lvfi;
            lvfi.flags = LVFI_STRING;
            lvfi.psz = szName;
            nItem = ListView_FindItem(hWndLV, -1, &lvfi);
            if (nItem != -1) {
               // We found a match; change the item's position.
               ListView_SetItemPosition(hWndLV, nItem, pt.x, pt.y);
            }
         }
      }
      // Turn AutoArrange back on if it was originally on.
      SetWindowLong(hWndLV, GWL_STYLE, dwStyle);
      RegCloseKey(hkey);
   }
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_CLOSE, Dlg_OnClose);

      case WM_APP:
         // Uncomment the line below to invoke the debugger
         // on the process that just got the injected DLL.
         // ForceDebugBreak();
```

```
         if (lParam)
            SaveListViewItemPositions((HWND) wParam);
         else
            RestoreListViewItemPositions((HWND) wParam);
         break;
   }

   return(FALSE);
}



////////////////////////////// End of File //////////////////////////////////
```

Внедрение DLL с помощью удаленных потоков

Третий способ внедрения DLL — самый гибкий. В нем используются многие особенности Windows: процессы, потоки, синхронизация потоков, управление виртуальной памятью, поддержка DLL и Unicode. (Если вы плаваете в каких-то из этих тем, прочтите сначала соответствующие главы книги.) Большинство Windows-функций позволяет процессу управлять лишь самим собой, исключая тем самым риск повреждения одного процесса другим. Однако есть и такие функции, которые дают возможность управлять чужим процессом. Изначально многие из них были рассчитаны на применение в отладчиках и других инструментальных средствах. Но ничто не мешает использовать их и в обычном приложении.

Внедрение DLL этим способом предполагает вызов функции *LoadLibrary* потоком целевого процесса для загрузки нужной DLL. Так как управление потоками чужого процесса сильно затруднено, вы должны создать в нем свой поток. К счастью, Windows-функция *CreateRemoteThread* делает эту задачу несложной:

```
HANDLE CreateRemoteThread(
   HANDLE hProcess,
   PSECURITY_ATTRIBUTES psa,
   DWORD dwStackSize,
   PTHREAD_START_ROUTINE pfnStartAddr,
   PVOID pvParam,
   DWORD fdwCreate,
   PDWORD pdwThreadld);
```

Она идентична *Create Thread,* но имеет дополнительный параметр *hProcess*, идентифицирующий процесс, которому будет принадлежать новый поток. Параметр *pfnStartAddr* определяет адрес функции потока. Этот адрес, разумеется, относится к удаленному процессу — функция потока не может находиться в адресном пространстве вашего процесса.

Теперь вы знаете, как создать поток в другом процессе. Но как заставить этот поток загрузить нашу DLL? Ответ прост: нужно, чтобы он вызвал функцию *LoadLibrary*:

```
HMODULE LoadLibrary(PCTSTR pszLibFile);
```

Заглянув в заголовочный файл WinBase.h, вы увидите, что для *LoadLibrary* там есть такие строки:

```
HMODULE WINAPI LoadLibraryA(LPCSTR  lpLibFileName);
HMODULE WINAPI LoadLibraryW(LPCWSTR lpLibFileName);
#ifdef UNICODE
#define LoadLibrary  LoadLibraryW
#else
#define LoadLibrary  LoadLibraryA
#endif // !UNICODE
```

В действительности существует две функции *LoadLibrary*: *LoadLibraryA* и *LoadLibraryW*. Они различаются только типом передаваемого параметра. Если имя файла библиотеки хранится как ANSI-строка, вызывайте *LoadLibraryA*; если же имя файла представлено Unicode-строкой — *LoadLibraryW*. Самой функции *LoadLibrary* нет. В большинстве программ макрос *LoadLibrary* раскрывается в *LoadLibraryA*.

К счастью, прототипы *LoadLibrary* и функции потока идентичны. Вот как выглядит прототип функции потока:

```
DWORD WINAPI ThreadFunc(PVOID pvParam);
```

Хорошо, не идентичны, но очень похожи друг на друга. Обе функции принимают единственный параметр и возвращают некое значение. Кроме того, обе используют одни и те же правила вызова — WINAPI. Это крайне удачное стечение обстоятельств, потому что нам как раз и нужно создать новый поток, адрес функции которого является адресом *LoadLibraryA* или *LoadLibraryW*. По сути, требуется выполнить примерно такую строку кода:

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   LoadLibraryW, L"C:\\MyLib.dll", 0, NULL);
```

Или, если вы предпочитаете Unicode:

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   LoadLibraryA, "C:\\MyLib.dll", 0, NULL);
```

Новый поток в удаленном процессе немедленно вызывает *LoadLibraryA* (или *LoadLibraryW*), передавая ей адрес полного имени DLL. Все просто. Однако вас ждут две проблемы.

Первая в том, что нельзя вот так запросто, как я показал выше, передать *LoadLibraryA* или *LoadLibraryW* в четвертом параметре функции *CreateRemoteThread*. Причина этого весьма неочевидна. При сборке программы в конечный двоичный файл помещается раздел импорта (описанный в главе

19). Этот раздел состоит из серии шлюзов к импортируемым функциям. Так что, когда ваш код вызывает функцию вроде *LoadLibraryA,* в разделе импорта модуля генерируется вызов соответствующего шлюза. А уже от шлюза происходит переход к реальной функции.

Следовательно, прямая ссылка на *LoadLibraryW* в вызове *CreateRemote-Thread* преобразуется в обращение к шлюзу *LoadLibraryW* в разделе импорта вашего модуля. Передача адреса шлюза в качестве стартового адреса удаленного потока заставит этот поток выполнить неизвестно что. И скорее всего это закончится нарушением доступа. Чтобы напрямую вызывать *LoadLibraryW*, минуя шлюз, вы должны выяснить ее точный адрес в памяти с помощью *GetProcAddress*.

Вызов *Create Remote Thread* предполагает, что Kernel32.dll спроецирована в локальном процессе на ту же область памяти, что и в удаленном. Kernel32.dll используется всеми приложениями, и, как показывает опыт, система проецирует эту DLL в каждом процессе по одному и тому же адресу. Так что *CreateRemoteThread* надо вызвать так:

```
// получаем истинный адрес LoadLibraryW в Kernel32.dll
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
   GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");

HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   pfnThreadRtn, L"C:\\MyLib.dll", 0, NULL);
```

Или, если вы предпочитаете Unicode:

```
// получаем истинный адрес LoadLibraryA в Kernel32.dll
PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
   GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryA");

HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
   pfnThreadRtn, "C:\\MyLib.dll", 0, NULL);
```

Отлично, одну проблему мы решили. Но я говорил, что их две. Вторая связана со строкой, в которой содержится полное имя файла DLL. Строка «C:\\MyLib.dll» находится в адресном пространстве вызывающего процесса. Ее адрес передается только что созданному потоку, который в свою очередь передает его в *LoadLibraryW*. Но, когда *LoadLibraryW* будет проводить разыменование (dereferencing) этого адреса, она не найдет по нему строку с полным именем файла DLL и скорее всего вызовет нарушение доступа в потоке удаленного процесса; пользователь увидит сообщение о необрабатываемом исключении, и удаленный процесс будет закрыт. Все верно: вы благополучно угробили чужой процесс, сохранив свой в целости и сохранности!

Эта проблема решается размещением строки **с** полным именем файла DLL в адресном пространстве удаленного процесса Впоследствии, вызывая *CreateRemoteThread,* мы передадим ее адрес (в удаленном процессе). На этот случай в Windows предусмотрена функция *VirtualAllocEx,* которая позволяет процессу выделять память в чужом адресном пространстве:

```
PVOID VirtualAllocEx(
    HANDLE hProcess,
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD flAllocationType,
    DWORD flProtect);
```

А освободить эту память можно с помощью функции *VirtualFreeEx*.

```
BOOL VirtualFreeEx(
    HANDLE hProcess,
    PVOID pvAddress,
    SIZE_T dwSize,
    DWORD dwFreeType);
```

Обе функции аналогичны своим версиям без суффикса *Ex* в конце (о них я рассказывал в главе 15). Единственная разница между ними в том, что эти две функции требуют передачи в первом параметре описателя удаленного процесса.

Выделив память, мы должны каким-то образом скопировать строку из локального адресного пространства в удаленное. Для этого в Windows есть две функции:

```
BOOL ReadProcessMemory(
    HANDLE hProcess,
    LPCVOID pvAddressRemote,
    PVOID pvBufferLocal,
    SIZE_T dwSize,
    SIZE_T* pdwNumBytesRead);

BOOL WriteProcessMemory(
    HANDLE hProcess,
    PVOID pvAddressRemote,
    LPCVOID pvBufferLocal,
    SIZE_T dwSize,
    SIZE_T* pdwNumBytesWritten);
```

Параметр *hProcess* идентифицирует удаленный процесс, *pvAddressRemote* и *pvBufferLocal* определяют адреса в адресных пространствах удаленного и локального процесса, а *dwSize* — число передаваемых байтов. По адресу, на который указывает параметр *pdwNumBytesRead* или *pdwNumBytesWritten*, возвращается число фактически считанных или записанных байтов.

Теперь, когда вы понимаете, что я пытаюсь сделать, давайте суммируем все сказанное и запишем это в виде последовательности операций, которые вам надо будет выполнить.

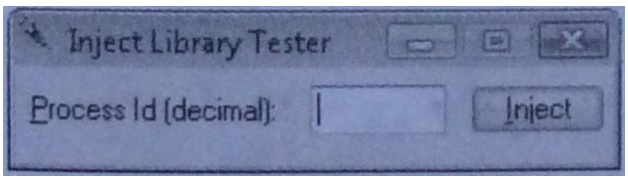
1. Выделите блок памяти в адресном пространстве удаленного процесса через *VirtualAllocEx*.
2. Вызвав *WriteProcessMemory*, скопируйте строку с полным именем файла DLL в блок памяти, выделенный в п.1.

3. Используя *GetProcAddress*, получите истинный адрес функции *LoadLibraryA* или *LoadLibraryW* внутри Kernel32.dll.
4. Вызвав *CreateRemoteThread*, создайте поток в удаленном процессе, который вызовет соответствующую функцию *LoadLibrary*, передав ей адрес блока памяти, выделенного в п. 1.

   На этом этапе DLL внедрена в удаленный процесс, а ее функция ***DllMain*** получила уведомление DLL_PROCESS_ATTACH и может приступить к выполнению нужного кода. Когда *DllMain* вернет управление, удаленный поток выйдет из *LoadLibrary* и вернется в функцию *BaseThreadStart* (см. главу 6), которая в свою очередь вызовет *ExitThread* и завершит этот поток. Теперь в удаленном процессе имеется блок памяти, выделенный в п. 1, и DLL, все еще «сидящая» в его адресном пространстве. Для очистки после завершения удаленного потока потребуется несколько дополнительных операций.
5. Вызовом *VirtualFreeEx* освободите блок памяти, выделенный в п. 1.
6. С помощью *GetProcAddress* определите истинный адрес функции *FreeLibrary* внутри Kernel32.dll.
7. Используя *Create RemoteThread*, создайте в удаленном процессе поток, который вызовет *FreeLibrary* с передачей HMODULE внедренной DLL. Вот, собственно, и все.

Программа-пример InjLib

Эта программа, (22-InjLib.exe), внедряет DLL с помощью функции *CreateRemoteThread*. Файлы исходного кода и ресурсов этой программы и DLL находятся в каталогах 22-InjLib и 22-ImgWalk внутри архива, доступного на веб-сайте поддержки этой книги. После запуска InjLib на экране появляется диалоговое окно для ввода идентификатора выполняемого процесса, в который будет внедрена DLL.



Вы можете выяснить этот идентификатор через Task Manager. Получив его, программа попытается открыть описатель этого процесса, вызвав *OpenProcess* и запросив соответствующие права доступа.

```
hProcess = OpenProcess(
    PROCESS_CREATE_THREAD |    // для CreateRemoteTnread
    PR0CESS_VM_0PERATI0N  |    // для VirtualAllocEx/VirtualFreeEx
    PROCESS_VM_WRITE,          // для WriteProcessMemory
    FALSE, dwProcessId);
```

Если *OpenProcess* вернет NULL, значит, программа выполняется в контексте защиты, в котором открытие описателя этого процесса не разрешено.

Некоторые процессы вроде WinLogon, SvcHost и Csrss выполняются по локальной системной учетной записи, которую зарегистрированный пользователь не имеет права изменять. Описатель такого процесса можно открыть, *только* если вы получили полномочия на отладку этих процессов. Программа ProcessInfo из главы 4 демонстрирует, как это делается.

При успешном выполнении *OpenProcess* записывает в буфер полное имя внедряемой DLL. Далее программа вызывает *InjectLib* и передает ей описатель удаленного процесса. И, наконец, после возврата из *InjectLib* программа выводит окно, где сообщает, успешно ли внедрена DLL, а потом закрывает описатель процесса.

Наверное, вы заметили, что я специально проверяю, не равен ли идентификатор процесса нулю. Если да, то вместо идентификатора удаленного процесса я передаю идентификатор процесса самой InjLib.exe, получаемый вызовом *GetCurrentProcessId*. Тогда при вызове *InjectLib* библиотека внедряется в адресное пространство процесса InjLib. Я сделал это для упрощения отладки. Сами понимаете, при возникновении ошибки иногда трудно определить, в каком процессе она находится: локальном или удаленном. Поначалу я отлаживал код с помощью двух отладчиков: один наблюдал за InjLib, другой — за удаленным процессом. Это оказалось страшно неудобно. Потом меня осенило, что InjLib способна внедрить DLL и в себя, т. е. в адресное пространство вызывающего процесса. И это сразу упростило отладку;

Просмотрев начало исходного кода модуля, вы увидите, что *InjectLib* — на самом деле макрос, заменяемый на *InjectLibA* или *InjectLibW* в зависимости от того, как компилируется исходный код. В исходном коде достаточно комментариев, и я добавлю лишь одно. Функциям *InjectLibA* весьма компактна. Она просто преобразует полное имя DLL из ANSI в Unicode и вызывает *InjectLibW*, которая и делает всю работу. Тут я придерживаюсь того подхода, который я рекомендовал в главе 2.

InjLib.cpp

```
/******************************************************************************
Module:  InjLib.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <stdio.h>
#include <tchar.h>
#include <malloc.h>        // For alloca
#include <TlHelp32.h>
#include "Resource.h"
#include <StrSafe.h>
```

```
//////////////////////////////////////////////////////////////////////////////


#ifdef UNICODE
   #define InjectLib InjectLibW
   #define EjectLib  EjectLibW
#else
   #define InjectLib InjectLibA
   #define EjectLib  EjectLibA
#endif   // !UNICODE


//////////////////////////////////////////////////////////////////////////////


BOOL WINAPI InjectLibW(DWORD dwProcessId, PCWSTR pszLibFile) {

   BOOL bOk = FALSE; // Assume that the function fails
   HANDLE hProcess = NULL, hThread = NULL;
   PWSTR pszLibFileRemote = NULL;

   __try {
      // Get a handle for the target process.
      hProcess = OpenProcess(
         PROCESS_QUERY_INFORMATION |   // Required by Alpha
         PROCESS_CREATE_THREAD     |   // For CreateRemoteThread
         PROCESS_VM_OPERATION      |   // For VirtualAllocEx/VirtualFreeEx
         PROCESS_VM_WRITE,             // For WriteProcessMemory
         FALSE, dwProcessId);
      if (hProcess == NULL) __leave;

      // Calculate the number of bytes needed for the DLL's pathname
      int cch = 1 + lstrlenW(pszLibFile);
      int cb  = cch * sizeof(wchar_t);

      // Allocate space in the remote process for the pathname
      pszLibFileRemote = (PWSTR)
         VirtualAllocEx(hProcess, NULL, cb, MEM_COMMIT, PAGE_READWRITE);
      if (pszLibFileRemote == NULL) __leave;

      // Copy the DLL's pathname to the remote process' address space
      if (!WriteProcessMemory(hProcess, pszLibFileRemote,
         (PVOID) pszLibFile, cb, NULL)) __leave;

      // Get the real address of LoadLibraryW in Kernel32.dll
      PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
         GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");
      if (pfnThreadRtn == NULL) __leave;
```

```
      // Create a remote thread that calls LoadLibraryW(DLLPathname)
      hThread = CreateRemoteThread(hProcess, NULL, 0,
         pfnThreadRtn, pszLibFileRemote, 0, NULL);
      if (hThread == NULL) __leave;

      // Wait for the remote thread to terminate
      WaitForSingleObject(hThread, INFINITE);

      bOk = TRUE; // Everything executed successfully
   }
   __finally { // Now, we can clean everything up

      // Free the remote memory that contained the DLL's pathname
      if (pszLibFileRemote != NULL)
         VirtualFreeEx(hProcess, pszLibFileRemote, 0, MEM_RELEASE);

      if (hThread  != NULL)
         CloseHandle(hThread);

      if (hProcess != NULL)
         CloseHandle(hProcess);
   }

   return(bOk);
}


//////////////////////////////////////////////////////////////////////////////


BOOL WINAPI InjectLibA(DWORD dwProcessId, PCSTR pszLibFile) {

   // Allocate a (stack) buffer for the Unicode version of the pathname
   SIZE_T cchSize = lstrlenA(pszLibFile) + 1;
   PWSTR pszLibFileW = (PWSTR)
      _alloca(cchSize * sizeof(wchar_t));

   // Convert the ANSI pathname to its Unicode equivalent
   StringCchPrintfW(pszLibFileW, cchSize, L"%S", pszLibFile);

   // Call the Unicode version of the function to actually do the work.
   return(InjectLibW(dwProcessId, pszLibFileW));
}


//////////////////////////////////////////////////////////////////////////////


BOOL WINAPI EjectLibW(DWORD dwProcessId, PCWSTR pszLibFile) {
```

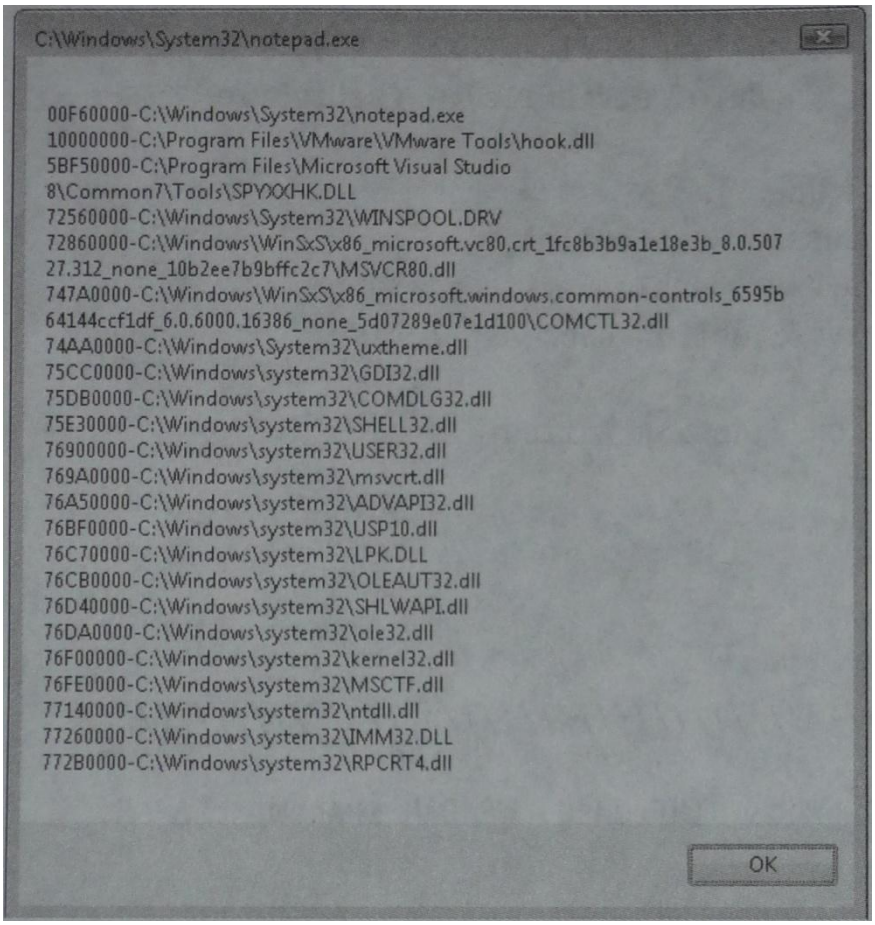
```
BOOL bOk = FALSE; // Assume that the function fails
HANDLE hthSnapshot = NULL;
HANDLE hProcess = NULL, hThread = NULL;

__try {
   // Grab a new snapshot of the process
   hthSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, dwProcessId);
   if (hthSnapshot == INVALID_HANDLE_VALUE) __leave;

   // Get the HMODULE of the desired library
   MODULEENTRY32W me = { sizeof(me) };
   BOOL bFound = FALSE;
   BOOL bMoreMods = Module32FirstW(hthSnapshot, &me);
   for (; bMoreMods; bMoreMods = Module32NextW(hthSnapshot, &me)) {
      bFound = (_wcsicmp(me.szModule,  pszLibFile) == 0) ||
               (_wcsicmp(me.szExePath, pszLibFile) == 0);
      if (bFound) break;
   }
   if (!bFound) __leave;

   // Get a handle for the target process.
   hProcess = OpenProcess(
      PROCESS_QUERY_INFORMATION |
      PROCESS_CREATE_THREAD     |
      PROCESS_VM_OPERATION,  // For CreateRemoteThread
      FALSE, dwProcessId);
   if (hProcess == NULL) __leave;

   // Get the real address of FreeLibrary in Kernel32.dll
   PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
      GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "FreeLibrary");
   if (pfnThreadRtn == NULL) __leave;

   // Create a remote thread that calls FreeLibrary()
   hThread = CreateRemoteThread(hProcess, NULL, 0,
      pfnThreadRtn, me.modBaseAddr, 0, NULL);
   if (hThread == NULL) __leave;

   // Wait for the remote thread to terminate
   WaitForSingleObject(hThread, INFINITE);

   bOk = TRUE; // Everything executed successfully
}
__finally { // Now we can clean everything up

   if (hthSnapshot != NULL)
```

```
         CloseHandle(hthSnapshot);

      if (hThread     != NULL)
         CloseHandle(hThread);

      if (hProcess    != NULL)
         CloseHandle(hProcess);
   }

   return(bOk);
}


//////////////////////////////////////////////////////////////////////////////


BOOL WINAPI EjectLibA(DWORD dwProcessId, PCSTR pszLibFile) {

   // Allocate a (stack) buffer for the Unicode version of the pathname
   SIZE_T cchSize = lstrlenA(pszLibFile) + 1;
   PWSTR pszLibFileW = (PWSTR)
      _alloca(cchSize * sizeof(wchar_t));

   // Convert the ANSI pathname to its Unicode equivalent
      StringCchPrintfW(pszLibFileW, cchSize, L"%S", pszLibFile);

   // Call the Unicode version of the function to actually do the work.
   return(EjectLibW(dwProcessId, pszLibFileW));
}


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_INJLIB);
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;
```

```
      case IDC_INJECT:
         DWORD dwProcessId = GetDlgItemInt(hWnd, IDC_PROCESSID, NULL, FALSE);
         if (dwProcessId == 0) {
            // A process ID of 0 causes everything to take place in the
            // local process; this makes things easier for debugging.
            dwProcessId = GetCurrentProcessId();
         }

         TCHAR szLibFile[MAX_PATH];
         GetModuleFileName(NULL, szLibFile, _countof(szLibFile));
         PTSTR pFilename = _tcsrchr(szLibFile, TEXT('\\')) + 1;
         _tcscpy_s(pFilename, _countof(szLibFile) - (pFilename - szLibFile),
             TEXT("22-ImgWalk.DLL"));
         if (InjectLib(dwProcessId, szLibFile)) {
            chVERIFY(EjectLib(dwProcessId, szLibFile));
            chMB("DLL Injection/Ejection successful.");
         } else {
            chMB("DLL Injection/Ejection failed.");
         }
         break;
   }
}


//////////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////////


int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_INJLIB), NULL, Dlg_Proc);
   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

Библиотека ImgWalk.dll

ImgWalk.dll — это DLL, которая, будучи внедрена в адресное пространство процесса, выдает список всех DLL, используемых этим процессом. Файлы исходного кода и ресурсов этой DLL находятся в каталоге 22-ImgWalk внутри архива, доступного на веб-сайте поддержки этой книги. Если, например, сначала запустить Notepad, а потом InjLib, передав ей идентификатор процесса Notepad, то InjLib внедрит ImgWalk.dll в адресное пространство Notepad. Попав туда, ImgWalk определит, образы каких файлов (EXE и DLL) используются процессом Notepad, и покажет результаты в следующем окне.



Модуль ImgWalk сканирует адресное пространство процесса и ищет спроецированные файлы, вызывая в цикле функцию VirtualQuery, которая заполняет структуру MEMORYBASIC INFORMATION. На каждой итерации цикла ImgWalk проверяет, нет ли строки с полным именем файла, которую можно было бы добавить в список, выводимый на экран.

```
/******************************************************************************
Module:  ImgWalk.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <tchar.h>
```

```
//////////////////////////////////////////////////////////////////////////////


BOOL WINAPI DllMain(HINSTANCE hInstDll, DWORD fdwReason, PVOID fImpLoad) {

   if (fdwReason == DLL_PROCESS_ATTACH) {
      char szBuf[MAX_PATH * 100] = { 0 };

      PBYTE pb = NULL;
      MEMORY_BASIC_INFORMATION mbi;
      while (VirtualQuery(pb, &mbi, sizeof(mbi)) == sizeof(mbi)) {

         int nLen;
         char szModName[MAX_PATH];

         if (mbi.State == MEM_FREE)
            mbi.AllocationBase = mbi.BaseAddress;

         if ((mbi.AllocationBase == hInstDll) ||
             (mbi.AllocationBase != mbi.BaseAddress) ||
             (mbi.AllocationBase == NULL)) {
            // Do not add the module name to the list
            // if any of the following is true:
            // 1. If this region contains this DLL
            // 2. If this block is NOT the beginning of a region
            // 3. If the address is NULL
            nLen = 0;
         } else {
            nLen = GetModuleFileNameA((HINSTANCE) mbi.AllocationBase,
               szModName, _countof(szModName));
         }

         if (nLen > 0) {
            wsprintfA(strchr(szBuf, 0), "\n%p-%s",
               mbi.AllocationBase, szModName);
         }

         pb += mbi.RegionSize;
      }

      // NOTE: Normally, you should not display a message box in DllMain
      // due to the loader lock described in Chapter 20. However, to keep
      // this sample application simple, I am violating this rule.
      chMB(&szBuf[1]);
   }
```

```
   return(TRUE);
}


//////////////////////////////// End of File //////////////////////////////////
```

Сначала я проверяю, не совпадает ли базовый адрес региона с базовым адресом внедренной DLL. Если да, я обнуляю *nLen*, чтобы не показывать в окне имя внедренной DLL. Нет — пытаюсь получить имя модуля, загруженного по базовому адресу данного региона. Если значение *nLen* больше 0, система распознает, что указанный адрес идентифицирует загруженный модуль, и помещает в буфер *szModName* полное имя (вместе с путем) этого модуля. Затем я присоединяю HINSTANCE данного модуля (базовый адрес) и его полное имя к строке *szBuf*, которая в конечном счете и появится в окне. Когда цикл заканчивается, DLL открывает на экране окно со списком.

Внедрение троянской DLL

Другой способ внедрения состоит в замене DLL, загружаемой процессом, на другую DLL. Например, зная, что процессу нужна Xyz.dll, вы можете создать свою DLL и присвоить ей то же имя. Конечно, перед этим вы должны переименовать исходную Xyz.dll.

В своей Xyz.dll вам придется экспортировать те же идентификаторы, что и в исходной Xyz.dll. Это несложно, если задействовать механизм переадресации функций (см. главу 20); однако его лучше не применять, иначе вы окажетесь в зависимости от конкретной версии DLL. Если вы замените, скажем, системную DLL, а Майкрософт потом добавит в нее новые функции, в вашей версии той же DLL их не будет. А значит, не удастся загрузить приложения, использующие эти новые функции.

Если вы хотите применить этот метод только для одного приложения, то можете присвоить своей DLL уникальное имя и записать его в раздел импорта исполняемого модуля приложения. Дело в том, что раздел импорта содержит имена всех DLL, нужных EXE-модулю. Вы можете «покопаться» в этом разделе и изменить его так, чтобы загрузчик операционной системы загружал вашу DLL. Этот прием совсем неплох, но требует глубоких знаний о формате EXE- и DLL-файлов.

Внедрение DLL как отладчика

Отладчик может выполнять особые операции над отлаживаемым процессом. Когда отлаживаемый процесс загружен и его адресное пространство создано, но первичный поток еще не выполняется, система автоматически уведомляет об этом отладчик. В этот момент отладчик может внедрить в него нужный код (используя, например, *WriteProcessMemory*), а затем заставить его первичный поток выполнить внедренный код.

Этот метод требует манипуляций со структурой CONTEXT потока отлаживаемого процесса, а значит, ваш код будет зависим от типа процессора, и его придется модифицировать при переносе на другую процессорную платформу. Кроме того, вам почти наверняка придется вручную корректировать машинный код, который должен быть выполнен отлаживаемым процессом. Не забудьте и о жесткой связи между отладчиком и отлаживаемой программой: как только отладчик закрывается, Windows немедленно закрывает и отлаживаемую программу. Можно изменить это, вызвав функуию *DebugSetProc essKillOnExitc* параметром FALSE, а функция *DebugActiveProcessStop* позволяет остановить отладку процесса, не закрывая отлаживаемый процесс.

Внедрение кода через функцию *CreateProcess*

Если ваш процесс порождает дочерний, в который надо внедрить какой-то код, то задача значительно упрощается. Родительский процесс может создать новый процесс и сразу же приостановить его. Это позволит изменить состояние дочернего процесса до начала его выполнения. В то же время родительский процесс получает описатель первичного потока дочернего процесса. Зная его, вы можете модифицировать код, который будет выполняться этим потоком. Тем самым вы решите проблему, упомянутую в предыдущем разделе: в данном случае нетрудно установить регистр указателя команд, принадлежащий потоку, на код в проекции файла.

Вот один из способов контроля того, какой код выполняется первичным потоком дочернего процесса:

1. Создайте дочерний процесс в приостановленном состоянии.
2. Получите стартовый адрес его первичного потока, считав его из заголовка исполняемого модуля.
3. Сохраните где-нибудь машинные команды, находящиеся по этому адресу памяти.
4. Введите на их место свои команды. Этот код должен вызывать *LoadLibrary* для загрузки DLL.
5. Разрешите выполнение первичного потока дочернего процесса.
6. Восстановите ранее сохраненные команды по стартовому адресу первичного потока.
7. Пусть процесс продолжает выполнение со стартового адреса так, будто ничего и не было.

Этапы 6 и 7 довольно трудны, но реализовать их можно — такое уже делалось.

У этого метода масса преимуществ. Во-первых, мы получаем адресное пространство до выполнения приложения. Во-вторых, данный метод применим как в Windows 98, так и в Windows 2000. В третьих, мы можем без проблем отлаживать приложение с внедренной DLL, не пользуясь отладчиком. Наконец, он работает как в консольных, так и в GUI-приложениях.

Однако у него есть и недостатки. Внедрение DLL возможно, только если это делается из родительского процесса. И, конечно, этот метод создает зависимость программы от конкретного процессора; при ее переносе на другую процессорную платформу потребуются определенные изменения в коде.

Перехват API-вызовов: пример

Внедрение DLL в адресное пространство процесса — замечательный способ узнать, что происходит в этом процессе. Однако простое внедрение DLL не дает достаточной информации. Зачастую надо точно знать, как потоки определенного процесса вызывают различные функции, а иногда и изменять поведение той или иной Windows-функции.

Мне известна одна компания, которая выпустила DLL для своего приложения, работающего с базой данных. Эта DLL должна была расширить возможности основного продукта. При закрытии приложения DLL получала уведомление DLL_PROCESS_DETACH и только после этого проводила очистку ресурсов. При этом DLL должна была вызывать функции из других DLL для закрытия сокетов, файлов и других ресурсов, но к тому моменту другие DLL тоже получали уведомление DLL_PROCESS_DETACH, так что корректно завершить работу никак не удавалось.

Для решения этой проблемы компания наняла меня, и я предложил поставить ловушку на функцию *ExitProcess*. Как вам известно, вызов *ExitProcess* заставляет систему посылать библиотекам уведомление DLL_PROCESS_DETACH. Перехватывая вызов *ExitProcess*, мы гарантируем своевременное уведомление внедренной DLL о вызове этой функции. Причем уведомление приходит до того, как аналогичные уведомления посылаются другим DLL. В этот момент внедренная DLL узнаёт о завершении процесса и успевает провести корректную очистку. Далее вызывается функция *ExitProcess*, что приводит к рассылке уведомлений DLL_PROCESS_DETACH остальным DLL, и они корректно завершаются. Это же уведомление получает и внедренная DLL, но ничего особенного она не делает, так как уже выполнила свою задачу.

В этом примере внедрение DLL происходило как бы само по себе: приложение было рассчитано на загрузку именно этой DLL. Оказываясь в адресном пространстве процесса, DLL должна была просканировать EXE-модуль и все загружаемые DLL-модули, найти все обращения к *ExitProcess* и заменить их вызовами функции, находящейся во внедренной DLL. (Эта задача не так сложна, как кажется.) Подставная функция (функция ловушки), закончив свою работу, вызывала настоящую функцию *ExitProcess* из Kernel32.dll.

Данный пример иллюстрирует типичное применение перехвата API-вызовов, который позволил решить насущную проблему при минимуме дополнительного кода.

Перехват API-вызовов подменой кода

Перехват API-вызовов далеко не новый метод — разработчики пользуются им уже многие годы. Когда сталкиваешься с проблемой, аналогичной той, о которой я только что рассказал, то первое, что приходит в голову, — установить ловушку, подменив часть исходного кода, вот как это делается.

1. Найдите адрес функции, вызов которой вы хотите перехватывать (например, *ExitProcess* в Kernel32.dll).
2. Сохраните несколько первых байтов этой функции в другом участке памяти.
3. На их место вставьте машинную команду JUMP для перехода по адресу подставной функции. Естественно, сигнатура вашей функции должна быть такой же, как и исходной, т. е. все параметры, возвращаемое значение и правила вызова должны совпадать.
4. Теперь, когда поток вызовет перехватываемую функцию, команда JUMP перенаправит его к вашей функции. На этом этапе вы можете выполнить любой нужный код.
5. Снимите ловушку, восстановив ранее сохраненные (в п. 2) байты.
6. Если теперь вызвать перехватываемую функцию (таковой больше не являющуюся), она будет работать так, как работала до установки ловушки.
7. После того как она вернет управление, вы можете выполнить операции 2 и 3 и тем самым вновь поставить *ловушку* на эту функцию.

Этот метод был очень популярен среди программистов, создававших приложения для 16-разрядной Windows, и отлично работал в этой системе. В современных системах у этого метода возникло несколько серьезных недостатков, и я настоятельно не рекомендую его применять. Во-первых, он создает зависимость от конкретного процессора из-за команды JUMP, и, кроме того, приходится вручную писать машинные коды. Во-вторых, в системе с вытесняющей многозадачностью данный метод вообще не годится. На замену кода в начале функции уходит какое-то время, а в этот момент перехватываемая функция может понадобиться другому потоку. Результаты могут быть просто катастрофическими! Так что этот метод работает только в ситуациях, когда потоки вызывают функции строго поочередно.

Перехват API-вызовов с использованием раздела импорта

Данный способ API-перехвата решает обе упомянутые мной проблемы. Он прост и довольно надежен. Но для его понимания нужно иметь представление о том, как осуществляется динамическое связывание. В частности, вы должны разбираться в структуре раздела импорта модуля. В главе 19 я достаточно подробно объяснил, как создается этот раздел и что в нем находится. Читая последующий материал, вы всегда можете вернуться к этой главе.

Как вам уже известно, в разделе импорта содержится список DLL, необходимых модулю для нормальной работы. Кроме того, в нем перечислены все идентификаторы, которые модуль импортирует из каждой DLL. Вызывая импортируемую функцию, поток получает ее адрес фактически из раздела импорта.

Поэтому, чтобы перехватить определенную функцию, надо лишь изменить ее адрес в разделе импорта. Все! И никакой зависимости от процессорной платформы. А поскольку вы ничего не меняете в коде функции, то и о синхронизации потоков можно не беспокоиться.

Вот функция, которая делает эту сказку былью. Она ищет в разделе импорта модуля ссылку на идентификатор по определенному адресу и, найдя ее, подменяет адрес соответствующего идентификатора.

```
void CAPIHook::ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
    PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) {

    // в этом модуле нет раздела импорта ULONG
    ulSize;

    // Explorer сгенерировал исключение, обратившись при просмотре
    // содержимого палки к imagehlp.dll. Похоже, один из модулей был
    // выгружен. Также возможны проблемы из-за многопоточности: Toolhelp может
    // дать неточный список модулей, если во время их перечисления будет вызвана
    // функция FreeLibrary.
    PIMAGE_IMPORT_DESCRIPTOR pImportDesc = NULL;
    __try {
        pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR) ImageDirectoryEntryToData(
            hmodCaller, TRUE, IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);
    }
    __except (InvalidReadExceptionFilter(GetExceptionInformation())) {
        // здесь делать нечего, поток продолжает нормальную работу,
        // получив NULL в pImportDesc
    }

    if (pImportDesc == NULL)
        return;                                 // в этом модуле нет раздела импорта

    // находим дескриптор раздела импорта со ссылками
    // на функции DLL (вызываемого модуля)
    for (; pImportDesc->Name; pImportDesc++) {
        PSTR pszModName = (PSTR) ((PBYTE) hmodCaller + pImportDesc->Name);
        if (lstrcmpiA(pszModName, pszCalleeModName) == 0) {
            // получаем таблицу адресов импорта (IAT) для функций DLL
            PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
                ((PBYTE) hmodCaller + pImportDesc->FirstThunk);

            // заменяем адреса исходных функций адресами своих функций
            for (; pThunk->u1.Function; pThunk++) {

                // получаем адрес адреса функции
                PROC* ppfn = (PROC*) &pThunk->u1.Function;
```

```
                // та ли это функция, которая нас интересует?
                BOOL bFound = (*ppfn == ofnCurrent);
                if (bFound) {
                    if (!WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
                        sizeof(pfnNew), NULL) && (ERROR_NOACCESS ==
                        GetLastError())) {

                        DWORD dwOldProtect;
                        if (VirtualProtect(ppfn, sizeof(pfnNew), PAGE_WRITECOPY,
                            &dwOldProtect)) {

                            WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
                                sizeof(pfnNew), NULL);
                            VirtualProtect(ppfn, sizeof(pfnNew), dwOldProtect,
                                &dwOldProtect);
                        }
                    }
                    return;                    // получилось; выходим
                }
            }
        }   // Система анализирует все разделы импорта, пока не будет
            // найдена и настроена подходящая запись
    }
}
```

Чтобы понять, как вызывать эту функцию, представьте, что у нас есть модуль с именем DataBase.exe. Он вызывает *ExitProcess* из Kernel32.dll, но мы хотим, чтобы он обращался к *MyExitProcess* в нашем модуле DBExtend.dll. Для этого надо вызвать *ReplacelATEntryInOneMod* следующим образом.

```
PROC pfnOrig = GetProcAddress(GetModuleHandle("Kernel32"),
    "ExitProcess");
HMODULE hmodCaller = GetModuleHandle("Database.exe");

ReplaceIATEntryInOneMod(
    "Kernel32.dll", // модуль, содержащий ANSI-функцию
    pfnOrig,        // адрес исходной функции в вызываемой DLL
    MyExitProcess,  // адрес заменяющей функции
    hmodCaller);    // описатель модуля, из которого надо вызывать новую функцию
```

Первое, что делает *ReplaceIATEntryInOneMod*, — находит в модуле *hmodCaller раздел* импорта. Для этого она вызывает *ImageDirectoryEntryToData* и передает ей IMAGE_DIRECTORY_ENTRY_IMPORT. Если последняя функция возвращает NULL, значит, в модуле DataBase.exe такого раздела нет, и на этом все заканчивается. Вызов *ImageDirectoryEntryToData* защищен блоком *__try/__except* (см. главу 24), перехватывающим неожиданные исключения, которые могут сгенерировать функции из ImageHlp.dll. Это необходимо, поскольку *ReplaceIATEntryInOneMod* может быть вызвана с не-

действительным описателем модуля, что приведет к возникновению исключения с кодом OxC0000005. В частности, такое возможно в контексте Explorer, который очень быстро динамически загружает и выгружает DLL в потоках, отличных от потоков, вызывающих *ReplacelATEntrylnOneMod*.

Если же в DataBase.exe раздел импорта присутствует, то *ImageDirectoryEntryToData* возвращает его адрес как указатель типа PIMAGE_IMPORT_DESCRIPTOR. Тогда мы должны искать в разделе импорта DLL, содержащую требуемую импортируемую функцию. В данном примере мы ищем идентификаторы, импортируемые из Kernel32.dll (имя которой указывается в первом параметре *ReplacelATEntrylnOneMod*). В цикле *for* сканируются имена DLL. Заметьте, что в разделах импорта все строки имеют формат ANSI (Unicode не применяется). Вот почему я вызываю функцию *lstrcmpiA*, а не макрос *lstrcmpi*.

Если программа не найдет никаких ссылок на идентификаторы в Кегпе132.dll, то и в этом случае функция просто вернет управление и ничего делать не станет. А если такие ссылки есть, мы получим адрес массива структур IMAGE_THUNK_DATA, в котором содержится информация об импортируемых идентификаторах. Далее в списке из Kernel32.dll ведется поиск идентификатора с адресом, совпадающим с искомым. В данном Случае мы ищем адрес, соответствующий адресу функции *ExitProcess*. Если поиск ссылок на идентификаторы в Kernel32.dll в цикле завершится безрезультатно, функция возвращает управление. Если в секции импорта модуля все же обнаружены ссылки на Kernel32.dll, мы получаем массив структур IMAGE_THUNK_DATA, содержащих информацию об импортированных функциях. Заметьте, что некоторые компиляторы, такие как компилятор Borland Delphi, генерируют несколько разделов импорта для одного модуля, поэтому мы не останавливаем поиск на первом найденном разделе. Далее в каждом из подходящих разделов импорта мы перебираем идентификаторы, импортированные из Kernel32.dll в поисках адреса, соответствующего текущему адресу функции. В нашем примере ведется поиск адреса, соответствующего адресу функции ExitProcess.

Если такого адреса нет, значит, данный модуль не импортирует нужный идентификатор, и *ReplacelATEntryInOneMod* просто возвращает управление. Но если адрес обнаруживается, мы вызываем *WriteProcessMemory*, чтобы заменить его на адрес подставной функции. Я применяю *WriteProcessMemory*, а не *InterlockedExchangePointer*, потому что она изменяет байты, не обращая внимания на тип защиты страницы памяти, в которой эти байты находятся. Так, если страница имеет атрибут защиты PAGE_READONLY, вызов *InterlockedExchangePointer* приведет к нарушению доступа, а *WriteProcessMemory* сама модифицирует атрибуты защиты и без проблем выполнит свою задачу.

С этого момента любой поток, выполняющий код в модуле DataBase.exe, при обращении к *ExitProcess* будет вызывать нашу функцию. А из нее мы сможем легко получить адрес исходной функции *ExitProcess* в Kernel32.dll и при необходимости вызвать ее.

Обратите внимание, что *ReplaceIATEntryInOneMod* подменяет вызовы функций только в одном модуле. Если в его адресном пространстве присутствует другая DLL, использующая *ExitProcess*, она будет вызывать именно *ExitProcess* из Kernel32.dll.

Если вы хотите перехватывать обращения к *ExitProcess* из всех модулей, вам придется вызывать *ReplacelATEntrylnOneMod* для каждого модуля в адресном пространстве процесса. Я, кстати, написал еще одну функцию, *ReplaceIATEntryInAllMods*. С помощью Toolhelp-функций она перечисляет все модули, загруженные в адресное пространство процесса, и для каждого из них вызывает *ReplaceIATEntryInOneMod*, передавая в качестве последнего параметра описатель соответствующего модуля.

Но и в этом случае могут быть проблемы. Например, что получится, если после вызова *ReplaceIATEntryInAllMods* какой-нибудь поток вызовет *LoadLibrary* для загрузки новой DLL? Если в только что загруженной DLL имеются вызовы *ExitProcess*, она будет обращаться не к вашей функции, а к исходной. Для решения этой проблемы вы должны перехватывать функции *LoadLibraryA, LoadLibraryW, LoadLibraryExA* и *LoadLibraryExW* и вызывать *ReplaceIATEntryInOneMod* для каждого загружаемого модуля. Но и этого мало. Представьте себе, что у только что загруженного модуля есть зависимости периода компоновки с другими DLL, которые также могут вызвать *ExitProcess*. При вызове *LoadLibrary**-функции Windows сначала загружает такие статически связанные DLL, не позволяя обновить адрес *ExitProcess* в таблице импортируемых адресов (Import Address Table, IAT). Решение здесь простое: вместо вызова *ReplaceIATEntryInOneMod* на каждой явно загружаемой DLL мы вызываем функцию *ReplaceIATEntryInAllMods*, которая учитывает неявно загружаемые модули.

И, наконец, есть еще одна проблема, связанная с *GetProcAddress*. Допустим, поток выполняет такой код:

```
typedef int (WINAPI *PFNEXITPROCESS)(UINT uExitCode);
PENEXITPROCESS pfnExitProcess = (PFNEXITPROCESS) GetProcAddress(
   GetModuleHandle("Kernel32"), "ExitProcess");
pfnExitProcess(0);
```

Этот код сообщает системе, что надо получить истинный адрес *ExitProcess* в Kernel32.dll, а затем сделать вызов по этому адресу. Данный код будет выполнен в обход вашей подставной функции. Проблема решается перехватом обращений к *GetProcAddress*. При ее вызове Вы должны возвращать адрес своей функции.

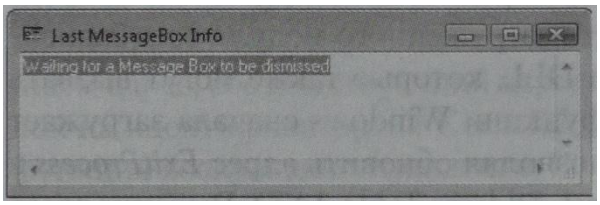
В следующем разделе я покажу, как на практике реализовать перехват API-вызовов и решить все проблемы, связанные с использованием *LoadLibrary* и *GetProcAddress*.

> **Примечание.** О создании более полного протокола для двухсторонних коммуникаций между программой и процессом, вызовы которого она перехватывает, с использованием специальных потоков и спроецированных в память файлов см. в статье «Detect and Plug GDI Leaks in Your Code with Two Powerful Tools for Windows XP» MSDN Magazine (http://msdn.microsoft.com/msdnmag/issues/03/01/GDILeaks).

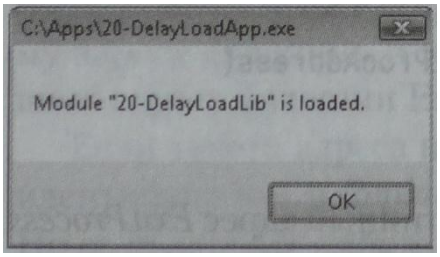
Программа-пример LastMsgBoxInfo

Эта программа, (22-LastMsgBoxInfo.exe), демонстрирует перехват API-вызовов. Она перехватывает все обращения к функции *MessageBox* из User32.dll. Для этого программа внедряет DLL с использованием ловушек. Файлы исходного кода и ресурсов этой программы и DLL находятся в каталогах 22-LastMsgBoxInfo и 22-LastMsgBoxInfoLib внутри архива, доступного на сайте поддержки этой книги.

После запуска LastMsgBoxInfo открывает диалоговое окно, показанное ниже.



В этот момент программа находится в состоянии ожидания. Запустите какое-нибудь приложение и заставьте его открыть окно с тем или иным сообщением. Тестируя свою программу, я использовал программу-пример 20-DelayLoadApp.exe (см. главу 20). Это простая C++-программа выводит различные сообщения при отложенной загрузке DLL.



Как видите, LastMsgBoxInfo позволяет наблюдать за вызовами функции *MessageBox* из других процессов. Однако несложно заметить, что программа не «видит» первого вызова *MessageBox*. Причина этого проста: ловушка Windows, используемая для внедрения седящего кода, срабатывает на первый вызов *MessageBox*, то есть слишком поздно.

Код, отвечающий за вывод диалогового окна LastMsgBoxInfo и управление им весьма прост. Трудности начинаются при настройке перехвата API-вызовов. Чтобы упростить эту задачу, я создал C++-класс CAPIHook, определенный в заголовочном файле APIHook.h и реализованный в файле APIHook.cpp. Пользоваться им очень легко, так как в нем лишь несколько

открытых функций-членов: конструктор, деструктор и метод, возвращающий адрес исходной функции, на которую вы ставите ловушку.

Для перехвата вызова какой-либо функции вы просто создаете экземпляр этого класса:

```
CAPIHook g_MessageBoxA("User32.dll", "MessageBoxA",
    (PROC) Hook_HessageBoxA, TRUE);

CAPIHook g_MessageBoxW("User32.dll", "MessageBoxW",
    (PROC) Hook_MessageBoxW, TRUE);
```

Мне приходится ставить ловушки на две функции: *MessageBoxA* и *MessageBoxW*. Обе эти функции находятся в User32.dll. Я хочу, чтобы при обращении к *MessageBoxA* вызывалась *Hook_MessageBoxA*, а при вызове *MessageBoxW* — *Hook_MessageBoxW*.

Конструктор класса CAPIHook просто запоминает, какую API-функцию нужно перехватывать, и вызывает *ReplacelATEntrylnAllMods*, которая, собственно, и выполняет эту задачу.

Следующая открытая функция-член — деструктор. Когда объект CAPIHook выходит за пределы области видимости, деструктор вызывает *ReplaceIATEntryInAllMods* для восстановления исходного адреса идентификатора во всех модулях, т. е. для снятия ловушки.

Третий открытый член класса возвращает адрес исходной функции. Эта функция обычно вызывается из подставной функции для обращения к перехватываемой функции. Вот как выглядит код функции *Hook_MessageBoxA:*

```
int WINAPI Hook_HessageBoxA(HWND hWnd, PCSTR pszText,
    PCSTR pszCaption, UINT uType) {

    int nResult = ((PFNMESSAGEBOXA)(PROC) g_MessageBoxA)
        (hWnd, pszText, pszCaption, uType);
    SendLastHsgBoxInfo(FALSE, (PVOID) pszCaption, (PVOID) pszText, nResult);
    return(nResult);
}
```

Этот код ссылается на глобальный объект *g_MessageBoxA* класса CAPIHook. Приведение его к типу PROC заставляет функцию-член вернуть адрес исходной функции *MessageBoxA* в User32.dll.

Всю работу по установке и снятию ловушек этот C++-класс берет на себя. До конца просмотрев файл CAPIHook.cpp, вы заметите, что мой C++-класс автоматически создает экземпляры объектов CAPIHook для перехвата вызовов *LoadLibraryA*, *LoadLibraryW*, *LoadLibraryExA*, *LoadLibraryExW* и *GetProcAddress*. Так что он сам справляется с проблемами, о которых я рассказывал в предыдущем разделе.

Обратите внимание, что модуль, экспортирующий функцию, которую вы перехватываете, должен быть загружен в конструкторе CAPIHook, иначе будет невозможно получить адрес исходной функции: *GetModuleHandleA*

вернет NULL, и вызов *GetProcAddress* закончится неудачей. Это ограничение очень существенно, поскольку он не позволяет корректно обрабатывать ситуации с отложенной загрузкой. Суть оптимизации отложенной загрузки в том, что загрузка такого модуля на самом деле происходит *после* вызова экспортируемой им функции.

Возможное решение — перехват *LoadLibrary**-функций для обнаружения модулей, экспортирующих функции, которые не перехваются и не изменяются. Обнаружив такой модуль, можно сделать одно из двух:

1. Снова получить таблицу импорта уже загруженного модуля, поскольку теперь вызовом *GetProcAddress* можно получить указатель на исходную функцию, которую нужно перехватить. Заметьте, что имя этой функции должно храниться в поле класса и устанавливаться во время исполнения конструктора.
2. Напрямую обновить адрес перехватываемой функции в таблице EAT экспортирующего модуля (см. код функции *ReplaceEATEntrylnOneMod*). В результате все новые модули, вызывающие перехватываемую функцию, будут вызывать и наш обработчик.

Но что, если модуль, экспортирующий перехватываемую функцию, будет выгружен вызовом FreeLibrary, а затем снова загружен? Эта задача не является предметом этой главы, но в этом примере содержится все необходимое для ее решения.

> **Примечание.** На сайте Microsoft Research опубликовано описание API перехвата под названием Detours (см. http://research.microsoft.com/sn/detours/).

```
LastMsgBoxInfo.cpp

/******************************************************************************
Module:  LastMsgBoxInfo.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"
#include "..\22-LastMsgBoxInfoLib\LastMsgBoxInfoLib.h"


//////////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_LASTMSGBOXINFO);
   SetDlgItemText(hWnd, IDC_INFO,
      TEXT("Waiting for a Message Box to be dismissed"));
```

```
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////


void Dlg_OnSize(HWND hWnd, UINT state, int cx, int cy) {

   SetWindowPos(GetDlgItem(hWnd, IDC_INFO), NULL,
      0, 0, cx, cy, SWP_NOZORDER);
}


//////////////////////////////////////////////////////////////////////////


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   switch (id) {
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;
   }
}


//////////////////////////////////////////////////////////////////////////


BOOL Dlg_OnCopyData(HWND hWnd, HWND hWndFrom, PCOPYDATASTRUCT pcds) {

   // Some hooked process sent us some message box info, display it
   SetDlgItemTextW(hWnd, IDC_INFO, (PCWSTR) pcds->lpData);
   return(TRUE);
}


//////////////////////////////////////////////////////////////////////////


INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_SIZE,       Dlg_OnSize);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
      chHANDLE_DLGMSG(hWnd, WM_COPYDATA,   Dlg_OnCopyData);
   }
   return(FALSE);
}


//////////////////////////////////////////////////////////////////////////
```

```
int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR pszCmdLine, int) {

   DWORD dwThreadId = 0;
   LastMsgBoxInfo_HookAllApps(TRUE, dwThreadId);
   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_LASTMSGBOXINFO), NULL, Dlg_Proc);
   LastMsgBoxInfo_HookAllApps(FALSE, 0);
   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

LastMsgBoxInfoLib.cpp

```
/******************************************************************************
Module:  LastMsgBoxInfoLib.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"
#include <WindowsX.h>
#include <tchar.h>
#include <stdio.h>
#include "APIHook.h"

#define LASTMSGBOXINFOLIBAPI extern "C" __declspec(dllexport)
#include "LastMsgBoxInfoLib.h"
#include <StrSafe.h>


//////////////////////////////////////////////////////////////////////////////


// Prototypes for the hooked functions
typedef int (WINAPI *PFNMESSAGEBOXA)(HWND hWnd, PCSTR pszText,
   PCSTR pszCaption, UINT uType);

typedef int (WINAPI *PFNMESSAGEBOXW)(HWND hWnd, PCWSTR pszText,
   PCWSTR pszCaption, UINT uType);


// We need to reference these variables before we create them.
extern CAPIHook g_MessageBoxA;
extern CAPIHook g_MessageBoxW;


//////////////////////////////////////////////////////////////////////////////


// This function sends the MessageBox info to our main dialog box
void SendLastMsgBoxInfo(BOOL bUnicode,
   PVOID pvCaption, PVOID pvText, int nResult) {
```

```
   // Get the pathname of the process displaying the message box
   wchar_t szProcessPathname[MAX_PATH];
   GetModuleFileNameW(NULL, szProcessPathname, MAX_PATH);

   // Convert the return value into a human-readable string
   PCWSTR pszResult = L"(Unknown)";
   switch (nResult) {
      case IDOK:       pszResult = L"Ok";        break;
      case IDCANCEL:   pszResult = L"Cancel";    break;
      case IDABORT:    pszResult = L"Abort";     break;
      case IDRETRY:    pszResult = L"Retry";     break;
      case IDIGNORE:   pszResult = L"Ignore";    break;
      case IDYES:      pszResult = L"Yes";       break;
      case IDNO:       pszResult = L"No";        break;
      case IDCLOSE:    pszResult = L"Close";     break;
      case IDHELP:     pszResult = L"Help";      break;
      case IDTRYAGAIN: pszResult = L"Try Again"; break;
      case IDCONTINUE: pszResult = L"Continue";  break;
   }

   // Construct the string to send to the main dialog box
   wchar_t sz[2048];
   StringCchPrintfW(sz, _countof(sz), bUnicode
      ? L"Process: (%d) %s\r\nCaption: %s\r\nMessage: %s\r\nResult: %s"
      : L"Process: (%d) %s\r\nCaption: %S\r\nMessage: %S\r\nResult: %s",
      GetCurrentProcessId(), szProcessPathname,
      pvCaption, pvText, pszResult);

   // Send the string to the main dialog box
   COPYDATASTRUCT cds = { 0, ((DWORD)wcslen(sz) + 1) * sizeof(wchar_t), sz };
   FORWARD_WM_COPYDATA(FindWindow(NULL, TEXT("Last MessageBox Info")),
      NULL, &cds, SendMessage);
}


//////////////////////////////////////////////////////////////////////////////


// This is the MessageBoxW replacement function
int WINAPI Hook_MessageBoxW(HWND hWnd, PCWSTR pszText, LPCWSTR pszCaption,
   UINT uType) {

   // Call the original MessageBoxW function
   int nResult = ((PFNMESSAGEBOXW)(PROC) g_MessageBoxW)
      (hWnd, pszText, pszCaption, uType);

   // Send the information to the main dialog box
   SendLastMsgBoxInfo(TRUE, (PVOID) pszCaption, (PVOID) pszText, nResult);
```

```
   // Return the result back to the caller
   return(nResult);
}


//////////////////////////////////////////////////////////////////////////


// This is the MessageBoxA replacement function
int WINAPI Hook_MessageBoxA(HWND hWnd, PCSTR pszText, PCSTR pszCaption,
   UINT uType) {

   // Call the original MessageBoxA function
   int nResult = ((PFNMESSAGEBOXA)(PROC) g_MessageBoxA)
      (hWnd, pszText, pszCaption, uType);

   // Send the information to the main dialog box
   SendLastMsgBoxInfo(FALSE, (PVOID) pszCaption, (PVOID) pszText, nResult);

   // Return the result back to the caller
   return(nResult);
}


//////////////////////////////////////////////////////////////////////////


// Hook the MessageBoxA and MessageBoxW functions
CAPIHook g_MessageBoxA("User32.dll", "MessageBoxA",
   (PROC) Hook_MessageBoxA);

CAPIHook g_MessageBoxW("User32.dll", "MessageBoxW",
   (PROC) Hook_MessageBoxW);

HHOOK g_hhook = NULL;


//////////////////////////////////////////////////////////////////////////


static LRESULT WINAPI GetMsgProc(int code, WPARAM wParam, LPARAM lParam) {
   return(CallNextHookEx(g_hhook, code, wParam, lParam));
}


//////////////////////////////////////////////////////////////////////////


// Returns the HMODULE that contains the specified memory address
static HMODULE ModuleFromAddress(PVOID pv) {

   MEMORY_BASIC_INFORMATION mbi;
   return((VirtualQuery(pv, &mbi, sizeof(mbi)) != 0)
      ? (HMODULE) mbi.AllocationBase : NULL);
```

```
}


//////////////////////////////////////////////////////////////////////////


BOOL WINAPI LastMsgBoxInfo_HookAllApps(BOOL bInstall, DWORD dwThreadId) {

   BOOL bOk;

   if (bInstall) {

      chASSERT(g_hhook == NULL); // Illegal to install twice in a row

      // Install the Windows' hook
      g_hhook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc,
         ModuleFromAddress(LastMsgBoxInfo_HookAllApps), dwThreadId);

      bOk = (g_hhook != NULL);
   } else {

      chASSERT(g_hhook != NULL); // Can't uninstall if not installed
      bOk = UnhookWindowsHookEx(g_hhook);
      g_hhook = NULL;
   }

   return(bOk);
}


//////////////////////////////// End of File //////////////////////////////////
```

LastMsgBoxInfoLib.h

```
/******************************************************************************
Module:  LastMsgBoxInfoLib.h
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#ifndef LASTMSGBOXINFOLIBAPI
#define LASTMSGBOXINFOLIBAPI extern "C" __declspec(dllimport)
#endif


//////////////////////////////////////////////////////////////////////////////


LASTMSGBOXINFOLIBAPI BOOL WINAPI LastMsgBoxInfo_HookAllApps(BOOL bInstall,
   DWORD dwThreadId);


//////////////////////////////// End of File //////////////////////////////////
```

APIHook.cpp

```
/*****************************************************************************
Module:  APIHook.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
*****************************************************************************/


#include "..\CommonFiles\CmnHdr.h"
#include <ImageHlp.h>
#pragma comment(lib, "ImageHlp")

#include "APIHook.h"
#include "..\CommonFiles\Toolhelp.h"
#include <StrSafe.h>


//////////////////////////////////////////////////////////////////////////////
//


// The head of the linked-list of CAPIHook objects
CAPIHook* CAPIHook::sm_pHead = NULL;

// By default, the module containing the CAPIHook() is not hooked
BOOL CAPIHook::ExcludeAPIHookMod = TRUE;


//////////////////////////////////////////////////////////////////////////////


CAPIHook::CAPIHook(PSTR pszCalleeModName, PSTR pszFuncName, PROC pfnHook) {

   // Note: the function can be hooked only if the exporting module
   //       is already loaded. A solution could be to store the function
   //       name as a member; then, in the hooked LoadLibrary* handlers, parse
   //       the list of CAPIHook instances, check if pszCalleeModName
   //       is the name of the loaded module to hook its export table and
   //       re-hook the import tables of all loaded modules.

   m_pNext  = sm_pHead;    // The next node was at the head
   sm_pHead = this;        // This node is now at the head

   // Save information about this hooked function
   m_pszCalleeModName   = pszCalleeModName;
   m_pszFuncName        = pszFuncName;
   m_pfnHook            = pfnHook;
   m_pfnOrig            =
      GetProcAddressRaw(GetModuleHandleA(pszCalleeModName), m_pszFuncName);
```

```
   // If function does not exit,... bye bye
   // This happens when the module is not already loaded
   if (m_pfnOrig == NULL)
   {
      wchar_t szPathname[MAX_PATH];
      GetModuleFileNameW(NULL, szPathname, _countof(szPathname));
      wchar_t sz[1024];
      StringCchPrintfW(sz, _countof(sz),
         TEXT("[%4u - %s] impossible to find %S\r\n"),
         GetCurrentProcessId(), szPathname, pszFuncName);
      OutputDebugString(sz);
      return;
   }

#ifdef _DEBUG
   // This section was used for debugging sessions when Explorer died as
   // a folder content was requested
   //
   //static BOOL s_bFirstTime = TRUE;
   //if (s_bFirstTime)
   //{
   //   s_bFirstTime = FALSE;

   //   wchar_t szPathname[MAX_PATH];
   //   GetModuleFileNameW(NULL, szPathname, _countof(szPathname));
   //   wchar_t* pszExeFile = wcsrchr(szPathname, L'\\') + 1;
   //   OutputDebugStringW(L"Injected in ");
   //   OutputDebugStringW(pszExeFile);
   //   if (_wcsicmp(pszExeFile, L"Explorer.EXE") == 0)
   //   {
   //      DebugBreak();
   //   }
   //   OutputDebugStringW(L"\n   --> ");
   //   StringCchPrintfW(szPathname, _countof(szPathname), L"%S", pszFuncName);
   //   OutputDebugStringW(szPathname);
   //   OutputDebugStringW(L"\n");
   //}
#endif

   // Hook this function in all currently loaded modules
   ReplaceIATEntryInAllMods(m_pszCalleeModName, m_pfnOrig, m_pfnHook);
}


//////////////////////////////////////////////////////////////////////////////
```

```
CAPIHook::~CAPIHook() {

   // Unhook this function from all modules
   ReplaceIATEntryInAllMods(m_pszCalleeModName, m_pfnHook, m_pfnOrig);

   // Remove this object from the linked list
   CAPIHook* p = sm_pHead;
   if (p == this) {     // Removing the head node
      sm_pHead = p->m_pNext;
   } else {

      BOOL bFound = FALSE;

      // Walk list from head and fix pointers
      for (; !bFound && (p->m_pNext != NULL); p = p->m_pNext) {
         if (p->m_pNext == this) {
            // Make the node that points to us point to our next node
            p->m_pNext = p->m_pNext->m_pNext;
            bFound = TRUE;
         }
      }
   }
}


////////////////////////////////////////////////////////////////////////////


// NOTE: This function must NOT be inlined
FARPROC CAPIHook::GetProcAddressRaw(HMODULE hmod, PCSTR pszProcName) {

   return(::GetProcAddress(hmod, pszProcName));
}


////////////////////////////////////////////////////////////////////////////


// Returns the HMODULE that contains the specified memory address
static HMODULE ModuleFromAddress(PVOID pv) {

   MEMORY_BASIC_INFORMATION mbi;
   return((VirtualQuery(pv, &mbi, sizeof(mbi)) != 0)
      ? (HMODULE) mbi.AllocationBase : NULL);
}


////////////////////////////////////////////////////////////////////////////


void CAPIHook::ReplaceIATEntryInAllMods(PCSTR pszCalleeModName,
   PROC pfnCurrent, PROC pfnNew) {
```

```
   HMODULE hmodThisMod = ExcludeAPIHookMod
      ? ModuleFromAddress(ReplaceIATEntryInAllMods) : NULL;

   // Get the list of modules in this process
   CToolhelp th(TH32CS_SNAPMODULE, GetCurrentProcessId());

   MODULEENTRY32 me = { sizeof(me) };
   for (BOOL bOk = th.ModuleFirst(&me); bOk; bOk = th.ModuleNext(&me)) {

      // NOTE: We don't hook functions in our own module
      if (me.hModule != hmodThisMod) {

         // Hook this function in this module
         ReplaceIATEntryInOneMod(
            pszCalleeModName, pfnCurrent, pfnNew, me.hModule);
      }
   }
}


//////////////////////////////////////////////////////////////////////////


// Handle unexpected exceptions if the module is unloaded
LONG WINAPI InvalidReadExceptionFilter(PEXCEPTION_POINTERS pep) {

   // handle all unexpected exceptions because we simply don't patch
   // any module in that case
   LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;

   // Note: pep->ExceptionRecord->ExceptionCode has 0xc0000005 as a value

   return(lDisposition);
}


void CAPIHook::ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
   PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) {

   // Get the address of the module's import section
   ULONG ulSize;

   // An exception was triggered by Explorer (when browsing the content of
   // a folder) into imagehlp.dll. It looks like one module was unloaded...
   // Maybe some threading problem: the list of modules from Toolhelp might
   // not be accurate if FreeLibrary is called during the enumeration.
   PIMAGE_IMPORT_DESCRIPTOR pImportDesc = NULL;
   __try {
      pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR) ImageDirectoryEntryToData(
```

```
         hmodCaller, TRUE, IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);
   }
   __except (InvalidReadExceptionFilter(GetExceptionInformation())) {
      // Nothing to do in here, thread continues to run normally
      // with NULL for pImportDesc
   }

   if (pImportDesc == NULL)
      return;  // This module has no import section or is no longer loaded


   // Find the import descriptor containing references to callee's functions
   for (; pImportDesc->Name; pImportDesc++) {
      PSTR pszModName = (PSTR) ((PBYTE) hmodCaller + pImportDesc->Name);
      if (lstrcmpiA(pszModName, pszCalleeModName) == 0) {

         // Get caller's import address table (IAT) for the callee's functions
         PIMAGE_THUNK_DATA pThunk = (PIMAGE_THUNK_DATA)
            ((PBYTE) hmodCaller + pImportDesc->FirstThunk);

         // Replace current function address with new function address
         for (; pThunk->u1.Function; pThunk++) {

            // Get the address of the function address
            PROC* ppfn = (PROC*) &pThunk->u1.Function;

            // Is this the function we're looking for?
            BOOL bFound = (*ppfn == pfnCurrent);
            if (bFound) {
               if (!WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
                     sizeof(pfnNew), NULL) && (ERROR_NOACCESS == GetLastEr-
ror())) {
                  DWORD dwOldProtect;
                  if (VirtualProtect(ppfn, sizeof(pfnNew), PAGE_WRITECOPY,
                     &dwOldProtect)) {

                     WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
                        sizeof(pfnNew), NULL);
                     VirtualProtect(ppfn, sizeof(pfnNew), dwOldProtect,
                        &dwOldProtect);
                  }
               }
               return;  // We did it, get out
            }
         }
```

```
      }  // Each import section is parsed until the right entry is found and
patched
   }
}


//////////////////////////////////////////////////////////////////////////


void CAPIHook::ReplaceEATEntryInOneMod(HMODULE hmod, PCSTR pszFunctionName,
   PROC pfnNew) {

   // Get the address of the module's export section
   ULONG ulSize;

   PIMAGE_EXPORT_DIRECTORY pExportDir = NULL;
   __try {
      pExportDir = (PIMAGE_EXPORT_DIRECTORY) ImageDirectoryEntryToData(
         hmod, TRUE, IMAGE_DIRECTORY_ENTRY_EXPORT, &ulSize);
   }
   __except (InvalidReadExceptionFilter(GetExceptionInformation())) {
      // Nothing to do in here, thread continues to run normally
      // with NULL for pExportDir
   }

   if (pExportDir == NULL)
      return;  // This module has no export section or is unloaded

   PDWORD pdwNamesRvas = (PDWORD) ((PBYTE) hmod + pExportDir->AddressOfNames);
   PWORD pdwNameOrdinals = (PWORD)
      ((PBYTE) hmod + pExportDir->AddressOfNameOrdinals);
   PDWORD pdwFunctionAddresses = (PDWORD)
      ((PBYTE) hmod + pExportDir->AddressOfFunctions);

   // Walk the array of this module's function names
   for (DWORD n = 0; n < pExportDir->NumberOfNames; n++) {
      // Get the function name
      PSTR pszFuncName = (PSTR) ((PBYTE) hmod + pdwNamesRvas[n]);

      // If not the specified function, try the next function
      if (lstrcmpiA(pszFuncName, pszFunctionName) != 0) continue;

      // We found the specified function
      // --> Get this function's ordinal value
      WORD ordinal = pdwNameOrdinals[n];
```

```
      // Get the address of this function's address
      PROC* ppfn = (PROC*) &pdwFunctionAddresses[ordinal];

      // Turn the new address into an RVA
      pfnNew = (PROC) ((PBYTE) pfnNew - (PBYTE) hmod);

      // Replace current function address with new function address
      if (!WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
         sizeof(pfnNew), NULL) && (ERROR_NOACCESS == GetLastError())) {
         DWORD dwOldProtect;
         if (VirtualProtect(ppfn, sizeof(pfnNew), PAGE_WRITECOPY,
            &dwOldProtect)) {

            WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
               sizeof(pfnNew), NULL);
            VirtualProtect(ppfn, sizeof(pfnNew), dwOldProtect, &dwOldProtect);
         }
      }
      break;  // We did it, get out
   }
}


//////////////////////////////////////////////////////////////////////////
// Hook LoadLibrary functions and GetProcAddress so that hooked functions
// are handled correctly if these functions are called.

CAPIHook CAPIHook::sm_LoadLibraryA  ("Kernel32.dll", "LoadLibraryA",
   (PROC) CAPIHook::LoadLibraryA);

CAPIHook CAPIHook::sm_LoadLibraryW  ("Kernel32.dll", "LoadLibraryW",
   (PROC) CAPIHook::LoadLibraryW);

CAPIHook CAPIHook::sm_LoadLibraryExA("Kernel32.dll", "LoadLibraryExA",
   (PROC) CAPIHook::LoadLibraryExA);

CAPIHook CAPIHook::sm_LoadLibraryExW("Kernel32.dll", "LoadLibraryExW",
   (PROC) CAPIHook::LoadLibraryExW);

CAPIHook CAPIHook::sm_GetProcAddress("Kernel32.dll", "GetProcAddress",
   (PROC) CAPIHook::GetProcAddress);


//////////////////////////////////////////////////////////////////////////


void CAPIHook::FixupNewlyLoadedModule(HMODULE hmod, DWORD dwFlags) {
```

```
   // If a new module is loaded, hook the hooked functions
   if ((hmod != NULL) &&   // Do not hook our own module
       (hmod != ModuleFromAddress(FixupNewlyLoadedModule)) &&
       ((dwFlags & LOAD_LIBRARY_AS_DATAFILE) == 0) &&
       ((dwFlags & LOAD_LIBRARY_AS_DATAFILE_EXCLUSIVE) == 0) &&
       ((dwFlags & LOAD_LIBRARY_AS_IMAGE_RESOURCE) == 0)
       ) {

      for (CAPIHook* p = sm_pHead; p != NULL; p = p->m_pNext) {
         if (p->m_pfnOrig != NULL) {
            ReplaceIATEntryInAllMods(p->m_pszCalleeModName,
               p->m_pfnOrig, p->m_pfnHook);
         } else {
#ifdef _DEBUG
            // We should never end up here
            wchar_t szPathname[MAX_PATH];
            GetModuleFileNameW(NULL, szPathname, _countof(szPathname));
            wchar_t sz[1024];
            StringCchPrintfW(sz, _countof(sz),
               TEXT("[%4u - %s] impossible to find %S\r\n"),
               GetCurrentProcessId(), szPathname, p->m_pszCalleeModName);
            OutputDebugString(sz);
#endif
         }
      }
   }
}


//////////////////////////////////////////////////////////////////////////


HMODULE WINAPI CAPIHook::LoadLibraryA(PCSTR pszModulePath) {

   HMODULE hmod = ::LoadLibraryA(pszModulePath);
   FixupNewlyLoadedModule(hmod, 0);
   return(hmod);
}


//////////////////////////////////////////////////////////////////////////


HMODULE WINAPI CAPIHook::LoadLibraryW(PCWSTR pszModulePath) {

   HMODULE hmod = ::LoadLibraryW(pszModulePath);
   FixupNewlyLoadedModule(hmod, 0);
   return(hmod);
}
```

```
//////////////////////////////////////////////////////////////////////////////


HMODULE WINAPI CAPIHook::LoadLibraryExA(PCSTR pszModulePath,
   HANDLE hFile, DWORD dwFlags) {

   HMODULE hmod = ::LoadLibraryExA(pszModulePath, hFile, dwFlags);
   FixupNewlyLoadedModule(hmod, dwFlags);
   return(hmod);
}


//////////////////////////////////////////////////////////////////////////////


HMODULE WINAPI CAPIHook::LoadLibraryExW(PCWSTR pszModulePath,
   HANDLE hFile, DWORD dwFlags) {

   HMODULE hmod = ::LoadLibraryExW(pszModulePath, hFile, dwFlags);
   FixupNewlyLoadedModule(hmod, dwFlags);
   return(hmod);
}


//////////////////////////////////////////////////////////////////////////////


FARPROC WINAPI CAPIHook::GetProcAddress(HMODULE hmod, PCSTR pszProcName) {

   // Get the true address of the function
   FARPROC pfn = GetProcAddressRaw(hmod, pszProcName);

   // Is it one of the functions that we want hooked?
   CAPIHook* p = sm_pHead;
   for (; (pfn != NULL) && (p != NULL); p = p->m_pNext) {

      if (pfn == p->m_pfnOrig) {

         // The address to return matches an address we want to hook
         // Return the hook function address instead
         pfn = p->m_pfnHook;
         break;
      }
   }

   return(pfn);
}


////////////////////////////////// End of File /////////////////////////////////
```

APIHook.h

```
/*****************************************************************************
Module:  APIHook.h
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
*****************************************************************************/


#pragma once


//////////////////////////////////////////////////////////////////////////////


class CAPIHook {
public:
   // Hook a function in all modules
   CAPIHook(PSTR pszCalleeModName, PSTR pszFuncName, PROC pfnHook);

   // Unhook a function from all modules
   ~CAPIHook();

   // Returns the original address of the hooked function
   operator PROC() { return(m_pfnOrig); }

   // Hook module w/CAPIHook implementation?
   // I have to make it static because I need to use it
   // in ReplaceIATEntryInAllMods
   static BOOL ExcludeAPIHookMod;


public:
   // Calls the real GetProcAddress
   static FARPROC WINAPI GetProcAddressRaw(HMODULE hmod, PCSTR pszProcName);

private:
   static PVOID sm_pvMaxAppAddr; // Maximum private memory address
   static CAPIHook* sm_pHead;    // Address of first object
   CAPIHook* m_pNext;            // Address of next  object

   PCSTR m_pszCalleeModName;     // Module containing the function (ANSI)
   PCSTR m_pszFuncName;          // Function name in callee (ANSI)
   PROC  m_pfnOrig;              // Original function address in callee
   PROC  m_pfnHook;              // Hook function address

private:
   // Replaces a symbol's address in a module's import section
   static void WINAPI ReplaceIATEntryInAllMods(PCSTR pszCalleeModName,
      PROC pfnOrig, PROC pfnHook);
```

```
   // Replaces a symbol's address in all modules' import sections
   static void WINAPI ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
      PROC pfnOrig, PROC pfnHook, HMODULE hmodCaller);

   // Replaces a symbol's address in a module's export sections
   static void ReplaceEATEntryInOneMod(HMODULE hmod, PCSTR pszFunctionName,
PROC pfnNew);

private:
   // Used when a DLL is newly loaded after hooking a function
   static void    WINAPI FixupNewlyLoadedModule(HMODULE hmod, DWORD dwFlags);

   // Used to trap when DLLs are newly loaded
   static HMODULE WINAPI LoadLibraryA(PCSTR pszModulePath);
   static HMODULE WINAPI LoadLibraryW(PCWSTR pszModulePath);
   static HMODULE WINAPI LoadLibraryExA(PCSTR pszModulePath,
      HANDLE hFile, DWORD dwFlags);
   static HMODULE WINAPI LoadLibraryExW(PCWSTR pszModulePath,
      HANDLE hFile, DWORD dwFlags);

   // Returns address of replacement function if hooked function is requested
   static FARPROC WINAPI GetProcAddress(HMODULE hmod, PCSTR pszProcName);

private:
   // Instantiates hooks on these functions
   static CAPIHook sm_LoadLibraryA;
   static CAPIHook sm_LoadLibraryW;
   static CAPIHook sm_LoadLibraryExA;
   static CAPIHook sm_LoadLibraryExW;
   static CAPIHook sm_GetProcAddress;
};


//////////////////////////////// End of File //////////////////////////////////
```

ЧАСТЬ V

СТРУКТУРНАЯ ОБРАБОТКА ИСКЛЮЧЕНИЙ

Оглавление

ГЛАВА 23

Обработчики завершения

Закроем глаза и помечтаем, какие бы программы мы писали, если бы сбои в них были невозможны! Представляете: памяти навалом, неверных указателей никто не передает, нужные файлы всегда на месте. Не программирование, а праздник, да? А код программ? Насколько он стал бы проще и понятнее! Без всех этих *if* и *goto*.

И если вы давно мечтали о такой среде программирования, вы сразу же оцените *структурную обработку исключений* (structured exception handling, SEH). Преимущество SEH в том, что при написании кода можно сосредоточиться на решении своей задачи. Если при выполнении программы возникнут неприятности, система сама обнаружит их и сообщит вам.

Хотя полностью игнорировать ошибки в программе при использовании SEH нельзя, она все же позволяет отделить основную работу от рутинной обработки ошибок, к которой можно вернуться позже.

Главное, почему Майкрософт ввела в Windows поддержку SEH, было ее стремление упростить разработку операционной системы и повысить ее надежность. А нам SEH поможет сделать надежнее наши программы.

Основная нагрузка по поддержке SEH ложится на компилятор, а не на операционную систему. Он генерирует специальный код на входах и выходах блоков исключений (exception blocks), создает таблицы вспомогательных структур данных для поддержки SEH и предоставляет функции обратного вызова, к которым система могла бы обращаться для прохода по блокам исключений. Компилятор отвечает и за формирование стековых фреймов (stack frames) и другой внутренней информации, используемой операционной системой. Добавить поддержку SEH в компилятор — задача не из легких, поэтому не удивляйтесь, когда увидите, что разные поставщики по-разному реализуют SEH в своих компиляторах. К счастью, на детали реализации можно не обращать внимания, а просто задействовать возможности компилятора в поддержке SEH.

Различия в реализации SEH разными компиляторами могли бы затруднить описание конкретных примеров использования SEH. Но большинство

поставщиков компиляторов придерживается синтаксиса, рекомендованного Майкрософт. Синтаксис и ключевые слова в моих примерах могут отличаться от применяемых в других компиляторах, по основные концепции SEH везде одинаковы. В этой главе я использую синтаксис компилятора Microsoft Visual C++.

> **Примечание**. Не путайте SEH с обработкой исключений в C++, которая представляет собой еще одну форму обработки исключений, построенную на применении ключевых слов языка C++ *catch* и *throw*. При этом Microsoft Visual C++ использует преимущества поддержки SEH, уже обеспеченной компилятором и операционными системами Windows.

SEH предоставляет две основные возможности: обработку завершения (termination handling) и обработку исключений (exception handling). В этой главе мы рассмотрим обработку завершения.

Обработчик завершения гарантирует, что блок кода (собственно обработчик) будет выполнен независимо от того, как происходит выход из другого блока кода — защищенного участка программы. Синтаксис обработчика завершения при работе с компилятором Microsoft Visual C++ выглядит так:

```
__try {
    // защищенный блок
    ...
}
__finally {
    // обработчик завершения
    ...
}
```

Ключевые слова *__try* и *__finally* обозначают два блока обработчика завершения. В предыдущем фрагменте кода совместные действия операционной системы и компилятора гарантируют, что код блока *finally* обработчика завершения будет выполнен независимо от того, как произойдет выход из защищенного блока. И неважно, разместите вы в защищенном блоке операторы *return*, *goto* или даже *longjump* — обработчик завершения все равно будет вызван. Далее я покажу вам несколько примеров использования обработчиков завершения.

Примеры использования обработчиков завершения

Поскольку при использовании SEH компилятор и операционная система вместе контролируют выполнение вашего кода, то лучший, на мой взгляд, способ продемонстрировать работу SEH — изучать исходные тексты программ и рассматривать порядок выполнения операторов в каждом из примеров.

Поэтому в следующих разделах приведен ряд фрагментов исходного кода, а связанный с каждым из фрагментов текст поясняет, как компилятор и операционная система изменяют порядок выполнения кода.

Funcenstein1

Чтобы оценить последствия применения обработчиков завершения, рассмотрим более конкретный пример:

```
DWORD Funcenstein1() {
   DWORD dwTemp;

   // 1. Что-то делаем здесь
   ...
   __try {
      // 2. Запрашиваем разрешение на доступ
      // к защищенным данным, а затем используем их
      WaitForSingleObject(g_hSem, INFINITE);

      g_dwProtectedData = 5;
      dwTemp = g_dwProtectedData;
   }
   __finally {
      // 3. Даем и другим попользоваться защищенными данными
      ReleaseSemaphore(g_hSem, 1, NULL);
   }

   // 4. Продолжаем что-то делать
   return(dwTerap);
}
```

Пронумерованные комментарии подсказывают, в каком порядке будет выполняться этот код. Использование в *Funcenstein1* блоков *try-finally* на самом деле мало что дает. Код ждет освобождения семафора, изменяет содержимое защищенных данных, сохраняет новое значение в локальной переменной *dwTemp,* освобождает семафор и возвращает новое значение тому, кто вызвал эту функцию.

Funcenstein2

Теперь чуть-чуть изменим код функции и посмотрим, что получится:

```
DWORD Funcenstein2() {
   DWORD dwTerop;

   // 1. Что-то делаем здесь
   ...
   __try {
      // 2. Запрашиваем разрешение на доступ
```

```
        // к защищенным данным, а затем используем их
        WaitForSingleObject(g_hSem, INFINITE);

        g_dwProtectedData = 5;
        dwTemp = g_dwProtectedData;

        // возвращаем новое значение
        return(dwTemp);
    }
    __finally {
        // 3. Даем и другим попользоваться защищенными данными
        ReleaseSemaphore(g_hSem, 1, NULL);
    }

    // продолжаем что-то делать - в данной версии
    // этот участок кода никогда не выполняется
    dwTemp = 9;
    return(dwTemp);
}
```

В конец блока *try* в функции *Funcenstein2* добавлен оператор *return*. Он сообщает компилятору, что вы хотите выйти из функции и вернуть значение переменной *dwTemp* (в данный момент равное 5). Но, если будет выполнен *return*, текущий поток никогда не освободит семафор, и другие потоки не получат шанса занять этот семафор. Такой порядок выполнения грозит вылиться в действительно серьезную проблему: ведь потоки, ожидающие семафора, могут оказаться не в состоянии возобновить свое выполнение.

Применив обработчик завершения, мы не допустили преждевременного выполнения оператора *return*. Когда *return* пытается реализовать выход из блока *try*, компилятор проверяет, чтобы сначала был выполнен код в блоке *finally*, — причем до того, как оператору *return* в блоке *try* будет позволено реализовать выход из функции. Вызов *ReleaseSemaphore* в обработчике завершения (в функции *Funcenstein2*) гарантирует освобождение семафора — поток не сможет случайно сохранить права на семафор и тем самым лишить процессорного времени все ожидающие этот семафор потоки.

После выполнения блока *finally* функция фактически завершает работу. Любой код за блоком *finally* не выполняется, поскольку возврат из функции происходит внутри блока *try*. Так что функция возвращает 5 и никогда — 9.

Каким же образом компилятор гарантирует выполнение блока *finally* до выхода из блока *try*? Дело вот в чем. Просматривая исходный текст, компилятор видит, что вы вставили *return* внутрь блока *try*. Тогда он генерирует код, который сохраняет возвращаемое значение (в нашем примере 5) в созданной им же временной переменной. Затем создает код для выполнения инструкций, содержащихся внутри блока *finally*, — это называется *локальной раскруткой* (local unwind). Точнее, локальная раскрутка происходит, когда система выполняет блок *finally* из-за преждевременного выхода из блока *try*. Значение

временной переменной, сгенерированной компилятором, возвращается из функции после выполнения инструкций в блоке *finally*.

Как видите, чтобы все это вытянуть, компилятору приходится генерировать дополнительный код, а системе — выполнять дополнительную работу. На разных типах процессоров поддержка обработчиков завершения реализуется по-разному. Например, процессору Alpha понадобится несколько сотен или даже тысяч машинных команд, чтобы перехватить преждевременный возврат из *try* и вызвать код блока *finally*. Поэтому лучше не писать код, вызывающий преждевременный выход из блока *try* обработчика завершения, — это может отрицательно сказаться на быстродействии программы. Чуть позже мы обсудим ключевое слово *__leave*, которое помогает избежать написания кода, приводящего к локальной раскрутке.

Обработка исключений предназначена для перехвата тех исключений, которые происходят не слишком часто (в нашем случае — преждевременного возврата). Если же какое-то исключение — чуть ли не норма, гораздо эффективнее проверять его явно, не полагаясь на SEH.

Заметьте: когда поток управления выходит из блока *try* естественным образом (как в *Funcenstein1*), издержки от вызова блока *finally* минимальны. При использовании компилятора Microsoft на процессорах *x86* для входа в *finally* при нормальном выходе из *try* исполняется всего одна машинная команда — вряд ли вы заметите ее влияние на быстродействие своей программы. Но издержки резко возрастут, если компилятору придется генерировать дополнительный код, а операционной системе — выполнять дополнительную работу, как в *Funcenstein2*.

Funcenstein3

Снова изменим код функции:

```
DWORD Funcenstein3() {
   DWORD dwTemp;

   // 1. Что-то делаем здесь
   ...
   __try {
      // 2. Запрашиваем разрешение на доступ
      // к защищенным данным, а затем используем их
      WaitForSingleObject(g_hSen, INFINITE);

      g.dwProtectedData = 5;
      dwTemp = g.dwProtectedData;

      // пытаемся перескочить через блок finally
      goto ReturnValue;
   }
```

```
    __finally {
        // 3. Даем и другим попользоваться защищенными данными
        ReleaseSemaphore(g_hSem, 1, NULL);
    }

    dwTemp = 9;
    // 4. Продолжаем что-то делать
    ReturnValue:
    return(dwTemp);
}
```

Обнаружив в блоке *try* функции *Funcenstein3* оператор *goto*, компилятор генерирует код для локальной раскрутки, чтобы сначала выполнялся блок *finally*. Но на этот раз после *finally* исполняется код, расположенный за меткой *ReturnValue*, так как возврат из функции не происходит ни в блоке *try*, ни в блоке *finally*. В итоге функция возвращает 5. И опять, поскольку вы прервали естественный ход потока управления из *try* в *finally*, быстродействие программы — в зависимости от типа процессора — может снизиться весьма значительно.

Funcfurter1

А сейчас разберем другой сценарий, в котором обработка завершения действительно полезна:

```
DWORD Funcfurter1() {
    DWORD dwTemp;

    // 1. Что-то делаем здесь
    ...
    __try {
        // 2. Запрашиваем разрешение на доступ
        // к защищенным данным, а затем используем их
        WaitForSingleObject(g_hSem, INFINITE);

        dwTemp = Funcinator(g_dwProtectedData);
    }
    __finally {
        // 3. Даем и другим попользоваться защищенными данными
        ReleaseSemaphore(g_hSem, 1, NULL);
    }

    // 4. Продолжаем что-то делать
    return(dwTemp);
}
```

Допустим, в функции *Funcinator*, вызванной из блока *try*, — «жучок», из-за которого возникает нарушение доступа к памяти. Без SEH пользователь

в очередной раз увидел бы самое известное диалоговое окно Application has stopped working (см. главу 25). Стоит его закрыть — завершится и приложение. Если бы процесс завершился (из-за неправильного доступа к памяти), семафор остался бы занят — соответственно и ожидающие его потоки не получили бы процессорное время. Но вызов *ReleaseSemaphore* в блоке *finally* гарантирует освобождение семафора, даже если нарушение доступа к памяти происходит в какой-то другой функции.

Однако, начиная с Windows Vista, разработчики должны явно защищать свои блоки *try/finally*, гарантируя исполнение блока *finally* в случае исключения, подробнее см. ниже, а также в следующей главе.

В прежних вызовах Windows исполнение блока *finally* гарантировалось не для всех типов исключений. Так, если в Windows XP возникало исключение из-за переполнения стека в блоке *try*, вероятность вызова *finally* была небольшой, поскольку коду службы WER, функционирующему внутри сбойного процесса, просто не хватит места в стеке, и процесс «молча» завершится. Аналогично, не гарантируется исполнение обработчика завершения в случае исключения, сгенерированного при повреждении SEH-цепочки. То же верно для исключения, возникшего при работе фильтра исключений. Чтобы избежать этого, придерживайтесь простого правила: код в блоках *catch* и *finally* должен выполнять минимум действий, иначе процесс просто завершится и никакие блоки *finally* больше не будут исполнены. По этой причине служба WER в Windows Vista работает в отдельном процессе (см. главу 25).

Раз обработчик завершения — такое мощное средство, способное перехватывать завершение программы из-за неправильного доступа к памяти, можно смело рассчитывать и на то, что оно также перехватит комбинации *setjump/longjump* и элементарные операторы типа *break* и *continue*.

Проверьте себя: *FuncaDoodleDoo*

Посмотрим, отгадаете ли вы, что именно возвращает следующая функция:

```
DWORD FuncaDoodleDoo() {
   DWORD dwTemp = 0;

   while (dwTemp < 10) {

      __try {
         if (dwTemp == 2)
            continue;
         if (dwTemp == 3)
            break;
      }
      __finally {
         dwTemp++;
      }
```

```
        dwTemp++;
    }

    dwTemp += 10;
    return(dwTemp);
}
```

Проанализируем эту функцию шаг за шагом. Сначала *dwTemp* приравнивается 0. Код в блоке *try* выполняется, но ни одно из условий в операторах *if* не дает TRUE, и поток управления естественным образом переходит в блок *finally*, где *dwTemp* увеличивается до 1. Затем инструкция после блока *finally* снова увеличивает значение *dwTemp*, приравнивая его 2.

На следующей итерации цикла *dwTemp* равно 2, поэтому выполняется оператор *continue* в блоке *try*. Без обработчика завершения, вызывающего принудительное выполнение блока *finally* перед выходом из *try*, управление было бы передано непосредственно в начало цикла *while*, значение *dwTemp* больше бы не менялось — и мы в бесконечном цикле! В присутствии же обработчика завершения система обнаруживает, что оператор *continue* приводит к преждевременному выходу из *try*, и передает управление блоку *finally*. Значение *dwTemp* в нем увеличивается до 3, но код за этим блоком не выполняется, так как управление снова передается оператору *continue*, и мы вновь в начале цикла.

Теперь обрабатываем третий проход цикла. На этот раз значение выражения в первом *if* равно FALSE, а во втором — TRUE. Система снова перехватывает нашу попытку прервать выполнение блока *try* и обращается к коду *finally*. Значение *dwTemp* увеличивается до 4. Так как выполнен оператор *break*, выполнение возобновляется после тела цикла. Поэтому код, расположенный за блоком *finally* (но в теле цикла), не выполняется. Код, расположенный за телом цикла, добавляет 10 к значению *dwTemp*, что дает в итоге 14, — это и есть результат вызова функции. Даже не стану убеждать вас никогда не писать такой код, как в *FuncaDoodleDoo*. Я-то включил *continue* и *break* в середину кода, только чтобы продемонстрировать поведение обработчика завершения.

Хотя обработчик завершения справляется с большинством ситуаций, в которых выход из блока *try* был бы преждевременным, он не может вызвать выполнение блока *finally* при завершении потока или процесса. Вызов *ExitThread* или *ExitProcess* сразу завершит поток или процесс — без выполнения какого-либо кода в блоке *finally*. То же самое будет, если ваш поток или процесс погибнут из-за того, что некая программа вызвала *TenninateThread* или *TerminateProcess*. Некоторые функции библиотеки C (вроде *abort*), в свою очередь вызывающие *ExitProcess*, тоже исключают выполнение блока *finally*. Раз вы нс можете запретить другой программе завершение какого-либо из своих потоков или процессов, так хоть сами не делайте преждевременных вызовов *ExitThread* и *ExitProcess*.

Funcenstein4

Рассмотрим еще один сценарий обработки завершения.

```
DWORD Funcenstein4() {
    DWORD dwTemp;
    // 1. Что-то делаем здесь
    ...
    __try {
        // 2. Запрашиваем разрешение на доступ
        // к защищенным данным, а затем используем их
        WaitForSingleObject(g_hSem, INFINITE);

        g_dwProtectedData = 5;
        dwTemp = g_dwProtectedData;

        // возвращаем новое значение
        return(dwTemp);
    }
    __finally {
        // 3. Даем и другим попользоваться защищенными данными
        ReleaseSemaphore(g_hSem, 1, NULL);
        return(103);
    }

    // продолжаем что-то делать - этот код
    // никогда не выполняется
    dwTemp = 9;
    return(dwTemp);
}
```

Блок *try* в *Funcenstein4* пытается вернуть значение переменной *dwTemp* (5) функции, вызвавшей *Funcenstein4*. Как мы уже отметили при обсуждении *Funcenstein2*, попытка преждевременного возврата из блока *try* приводит к генерации кода, который записывает возвращаемое значение во временную переменную, созданную компилятором. Затем выполняется код в блоке *finally*. Кстати, в этом варианте *Funcenstein2* я добавил в блок *finally* оператор *return*. Вопрос: что вернет *Funcenstein4* — 5 или 103? Ответ: 103, так как оператор *return* в блоке *finally* приведет к записи значения 103 в ту же временную переменную, в которую занесено значение 5. По завершении блока *finally* текущее значение временной переменной (103) возвращается функции, вызвавшей *Funcenstein4*.

Итак, обработчики завершения, весьма эффективные при преждевременном выходе из блока *try*, могут дать нежелательные результаты именно потому, что предотвращают досрочный выход из блока *try*. Лучше всего избегать любых операторов, способных вызвать преждевременный выход из блока *try* обработчика завершения. А в идеале — удалить все операторы *return*, *continue*,

break, *goto* (и им подобные) как из блоков *try*, так и из блоков *finally*. Тогда компилятор сгенерирует код и более компактный (перехватывать преждевременные выходы из блоков *try* не понадобится), и более быстрый (на локальную раскрутку потребуется меньше машинных команд). Да и читать ваш код будет гораздо легче.

Funcarama1

Мы уже далеко продвинулись в рассмотрении базового синтаксиса и семантики обработчиков завершения. Теперь поговорим о том, как обработчики завершения упрощают более сложные задачи программирования. Взгляните на функцию, в которой не используются преимущества обработки завершения:

```
BOOL Funcarama1() {
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PV0ID pvBuf = NULL;
    DWORD dwNumBytesRead;
    BOOL bOk;

    hFile = CreateFile(TEXT("SOMEDATA.DAT"), GENERIC_READ,
        FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
    if (hFile == INVALID.HANOLE.VALUE) {
        return(FALSE);
    }

    pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
    if (pvBuf == NULL) {
        CloseHandle(hFile);
        return(FALSE);
    }

    bOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!bOk || (dwNumBytesRead == 0)) {
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
        CloseHandle(hFile);
        return(FALSE);
    )

    // что-то делаем с данными
    ...
    // очистка всех ресурсов
    VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    CloseHandle(hFile);
    return(TRUE);
}
```

Проверки ошибок в функции *Funcarama1* затрудняют чтение ее текста, что усложняет ее понимание, сопровождение и модификацию.

Funcarama2

Конечно, можно переписать *Funcarama1* так, чтобы она стала яснее:

```
BOOL Funcararoa2() {
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PVOID pvBuf = NULL;
    DWORD dwNumBytesRead;
    BOOL bOk, bSuccess = FALSE;

    hFile = CreateFile(TEXT("SOHEDATA.DAT"), GENERIC_READ, FILE_SHARE_READ,
        NULL, OPEN_EXISTING, 0, NULL);

    if (hFile != INVALID_HANDLE_VALUE) {
        pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PAGE_READWRITE);
        if (pvBuf != NULL) {
            bOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
            if (bOk && (dwNumBytesRead != 0)) {
                // что-то делаем с данными
                ...
                bSuccess = TRUE;
            }
            VirtualFree(pvBuf, MEM_RELEASE | NEM_DECOMMIT);
        }
        CloseHandle(hFile);
    }
    return(bSuccess);
}
```

Funcarama2 легче для понимания, но по-прежнему трудна для модификации и сопровождения. Кроме того, приходится делать слишком много отступов по мере добавления новых условных операторов; после такой переделки вы того и гляди начнете писать код на правом краю экрана и переносить операторы на другую строку через каждые пять символов!

Funcarama3

Перепишем-ка еще раз первый вариант (*Funcarama1*), задействовав преимущества обработки завершения:

```
DWORD Funcarama3() {

    // Внимание: инициализируйте все переменные, предполагая худшее
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PV0ID pvBuf = NULL;

    __try {
        DWORD dwNumBytesRead;
        B00L bOk;
```

```
        hFile = CreateFile(TEXT("SOMEDATA. DAT"), GENERIC_READ,
            FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
        if (hFile == INVALID_HANDLE_VALUE) {
            return(FALSE);
        }

        pvBuf = VirtualAlloc(NULL, 1024, MEM_COMMIT, PA6E_READWRITE);
        if (pvBuf == NULL) {
            return(FALSE);
        }

        bOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
        if (!bOk || (dwNuffiBytesRead != 1024)) {
            return(FALSE);
        }

        // что-то делаем с данными
        ...
    }

    __finally {
        // очистка всех ресурсов
        if (pvBuf != NULL)
            VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
        if (hFile != INVALID_HANDLE_VALUE)
            CloseHandle(hFile);
    }
    // продолжаем что-то делать
    return(TRUE);
}
```

Главное достоинство *Funcarama3* в том, что весь код, отвечающий за очистку, собран в одном месте — в блоке *finally*. Если понадобится включить что-то в эту функцию, то для очистки мы просто добавим одну-единственную строку в блок *finally* — возвращаться к каждому месту возможного возникновения ошибки и вставлять в него строку для очистки не нужно.

Funcarama4: последний рубеж

Настоящая проблема в *Funcarama3* — расплата за изящество. Я уже говорил: избегайте по возможности операторов *return* внутри блока *try*.

Чтобы облегчить последнюю задачу, Майкрософт ввела еще одно ключевое слово в свой компилятор C++: *leave*. Вот новая версия (*Funcarama4*), построенная на применении нового ключевого слова:

```
DWORD Funcarama4() {
```

```
    // Внимание: инициализируйте все переменные, предполагая худшее
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PVOID pvBuf = NULL;

    // предполагаем, что выполнение функции будет неудачным
    BOOL bFunctionOk = FALSE;

    __try {
        DWORD dwNumBytesRead;
        BOOL bOk;

        hFile = CreateFile(TEXT("SOHEDATA.DAT"), GENERIC_READ,
            FILE_SHARE_READ, NULL, OPEN_EXISTING, 0, NULL);
        if (hFile == INVALID_HANDLE_VALUE) {
            __leave;
        }

        pvBuf = VirtualAlloc(NULL. 1024, MEM_COMMIT, PAGE_READWRITE);

        if (pvBuf == NULL) {
            __leave;
        }

        bOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
        if (!bOk || (dwNumBytesRead » 0)) {
            __leave;
        }

        // что-то делаем с данными
        ...
        // функция выполнена успешно
        bFunctionOk = TRUE;
    }
    __finally {
        // очистка всех ресурсов
        if (pvBuf != NULL)
            VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
        if (hFile != INVALID_HANDLE_VALUE)
            CloseHandle(hFile);
    }
    // продолжаем что-то делать
    return(bFunctionOk);
}
```

Ключевое слово *__leave* в блоке *try* вызывает переход в конец этого блока. Можете рассматривать это как переход на закрывающую фигурную скобку блока try. И никаких неприятностей это не сулит, потому что выход из блока

try и вход в блок *finally* происходит естественным образом. Правда, нужно ввести дополнительную булеву переменную *fFunctionOk*, сообщающую о завершении функции: удачно оно или нет. Но это дает минимальные издержки.

Разрабатывая функции, использующие обработчики завершения именно так, инициализируйте все описатели ресурсов недопустимыми значениями перед входом в блок *try*. Тогда в блоке *finally* вы проверите, какие ресурсы выделены успешно, и узнаете тем самым, какие из них следует потом освободить. Другой распространенный метод отслеживания ресурсов, подлежащих освобождению, — установка флага при успешном выделении ресурса. Код *finally* проверяет состояние флага и таким образом определяет, надо ли освобождать ресурс.

И еще о блоке finally

Пока нам с вами удалось четко выделить только два сценария, которые приводят к выполнению блока *finally*:

- нормальная передача управления от блока *try* блоку *finally*;
- локальная раскрутка — преждевременный выход из блока *try* (из-за операторов *goto*, *longjump*, *continue*, *break*, *return* и т. д.), вызывающий принудительную передачу управления блоку *finally*.

Третий сценарий — *глобальная раскрутка* (global unwind) — протекает не столь выраженно. Вспомним *Funcfurter1*. Ее блок *try* содержал вызов функции *Funcinator*. При неверном доступе к памяти в *Funcinator* глобальная раскрутка приводила к выполнению блока *finally* в *Funcfurter1*. Но подробнее о глобальной раскрутке мы поговорим в следующих главах.

Выполнение кода в блоке *finally* всегда начинается в результате возникновения одной из этих трех ситуаций. Чтобы определить, какая из них вызвала выполнение блока *finally*, вызовите встраиваемую функцию[1] *AbnormalTermination*:

```
BOOL AbnormalTermination();
```

> **Примечание.** *Встраиваемые функции* (intrinsic function) — это особые функции, распознаваемые компилятором. Обнаружив вызов такой функции, компилятор, вместо того, чтобы, как обычно, генерировать код для вызова, встраивает код этой функции. Примером встраиваемой функции может быть *memcpy* (если для компилятора задан ключ /Oi). Обнаружив вызов *memcpy*, компилятор встраивает эту функцию прямо в код *memcpy* вместо, собственно, ее вызова. Как правило, это повышает быстродействие, но увеличивает размер кода.
>
> Встраиваемая функция *AbnormalTermination* отличается от *memcpy* тем, что существует только во встроенном состоянии, в библиотеке C/C++ этой функции нет.

Ее можно вызвать только из блока *finally*; она возвращает булево значение, которое сообщает, был ли преждевременный выход из блока *try*, связан-

[1]

ного с данным блоком *finally*. Иначе говоря, если управление естественным образом передано из *try* в *finally*, *AbnormalTermination* возвращает FALSE. А если выход был преждевременным — обычно либо из-за локальной раскрутки, вызванной оператором *goto*, *return*, *break* или *continue*, либо из-за глобальной раскрутки, вызванной нарушением доступа к памяти, — то вызов *AbnormalTermination* дает TRUE. Но, когда она возвращает TRUE, различить, вызвано выполнение блока *finally* глобальной или локальной раскруткой, нельзя. Впрочем, это не проблема, так как вы должны избегать кода, приводящего к локальной раскрутке.

Funcfurter2

Следующий фрагмент демонстрирует использование встраиваемой функции *AbnormalTermination*:

```
DWORD Funcfurter2() {
   DWORD dwTemp;

   // 1. Что-то делаем здесь
   ...
   __try {
      // 2. Запрашиваем разрешение на доступ
      // к защищенным данным, а затем используем их
      WaitForSingleObject(g_hSem, INFINITE);

      dwTemp = Funcinator(g_dwProtectedData);
   }
   __finally {
      // 3. Даем и другим попользоваться защищенными данными
      ReleaseSemaphore(g_hSem, 1, NULL);

      if (!AbnormalTermination()) {
         // в блоке try не было ошибок - управление
         // передано в блок finally естественным образом
         ...
      } else {
         // что-то вызвало исключение, и, так как в блоке try
         // нет кода, который мог бы вызвать преждевременный
         // выход, блок finally выполняется из-за глобальной
         // раскрутки

         // если бы в блоке try был оператор goto, мы бы
         // не узнали, как попали сюда
         ...
      }
   }
```

```
    // 4. Продолжаем что-то делать
    return(dwTemp);
}
```

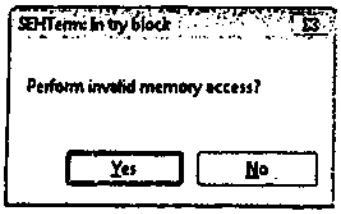
Теперь вы знаете, как создавать обработчики завершения. Вскоре вы увидите, что они могут быть еще полезнее и важнее, — когда мы дойдем до фильтров и обработчиков исключений (в следующей главе). А пока давайте суммируем причины, по которым следует применять обработчики завершения.

- Упрощается обработка ошибок — очистка гарантируется и проводится в одном месте.
- Улучшается восприятие текста программ.
- Облегчается сопровождение кода.
- Удается добиться минимальных издержек по скорости и размеру кода — при условии правильного применения обработчиков.

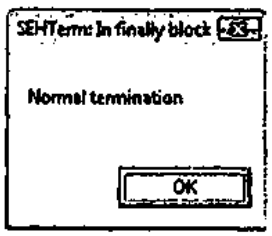
Программа-пример SEHTerm

Эта программа (23-SEHTerm.exe), демонстрирует обработчики завершения. Файлы исходного кода и ресурсов этой программы находятся в каталоге 23-SEHTerm внутри архива, доступного на сайте поддержки этой книги.

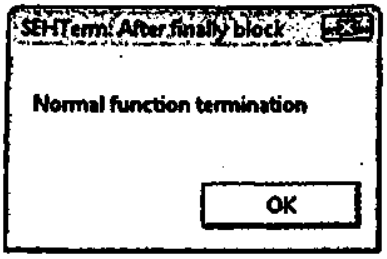
После запуска SEHTerm ее первичный поток входит в блок *try*. Из него открывается следующее окно.



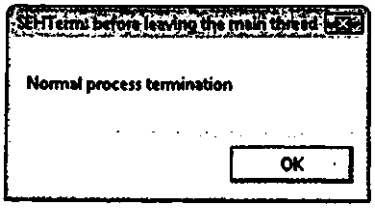
В этом окне предлагается обратиться к памяти по недопустимому адресу. (Большинство приложений не столь тактично — они обращаются по недопустимым адресам, никого не спрашивая.) Давайте обсудим, что случится, если вы щелкнете кнопку Yes. В этом случае поток попытается записать значение 5 по нулевому адресу памяти. Запись по нулевому адресу всегда вызывает исключение, связанное с нарушением доступа. А когда поток возбуждает такое исключение, Windows выводит окно, показанное ниже.



Заметьте, что это сообщение говорит об успешном исполнении кода блоке *try*. Если закрыть сообщение, поток покинет блок *finally* и покажет следующее сообщение:



Перед завершением главного потока выводится последнее сообщение об отсутствии необработанных исключений:



После того как вы закроете и это окно, процесс нормально завершится, поскольку функция *_tWinMain* вернет управление. Заметьте, что данное окно не появляется при аварийном завершении процесса.

А сейчас снова запустим эту программу. Но на этот раз попробуйте щелкнуть кнопку Yes, чтобы обратиться к памяти по недопустимому адресу (попытаться записать 5 по адресу NULL). В результате возникает необработанное нарушение доступа и в Windows XP открывается следующее окно.

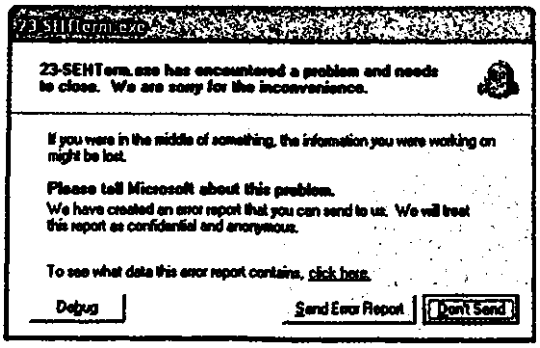


Рис. 23-1. *Сообщение о необработанном исключении в Windows XP*

В Windows Vista соответствующее окно выглядит так:

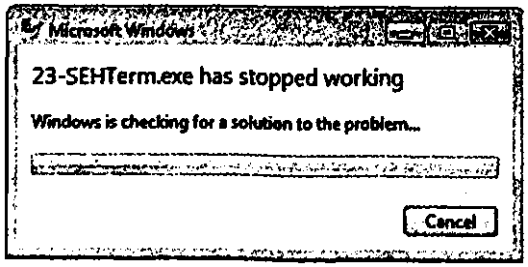


Рис. 23-2. *Первое сообщение о необработанном исключении в Windows Vista*

Если закрыть его щелчком кнопки Cancel, приложение по-тихому завершится. В противном случае выводится окно следующего пила:

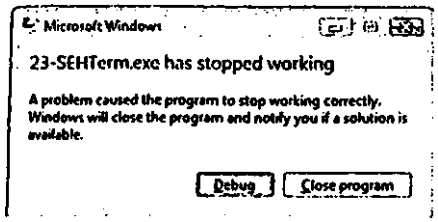
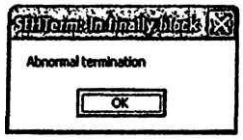


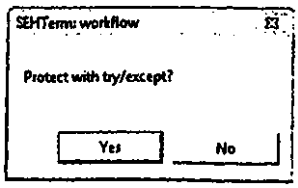
Рис. 23-3. *Второе сообщение о необработанном исключении в Windows Vista*

Щелчок кнопки Debug система выполняет действия, описанные в главе 25, и запускает отладчик, который подключается к сбойному процессу. Если вместо этого щелкнуть Close Program (в Windows Vista), Send Error Report либо Don't Send (в Windows XP), процесс завершится. Однако в коде программы-примера есть блок finally, поэтому он должен быть исполнен перед завершением процесса. При этом в Windows XP вывод1ггся следующее сообщение:

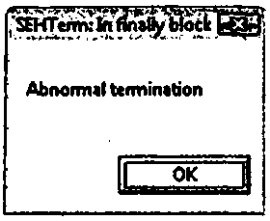


Блок *finally* исполняется, поскольку код в блоке try завершился аварийно. Если закрыть это окно, процесс действительно завершится. Однако это происходит только в версиях Windows, предшествующих Vista, в которых блок *finally* исполняется только в случае глобальной раскрутки. Как сказано в главе 6, точка входа потока защищена блоком *try/except*. В этом случае вызывается функция фильтра исключений в *__except()*, которая должна вернуть EXCEPTION_EXECUTE_ HANDLER. В Windows Vista механизмы необработанных исключений были в значительной степени переделаны с целью повышения надежности (см. главу 25). Сразу бросается в глаза один из недостатков этих нововведений: возвращается фильтр исключений, являющийся также его оболочкой, возвращает EXCEP-TION_CONTINUE_SEARCH, поэтому процесс тут же завершается без шансов на исполнение блока *finally*.

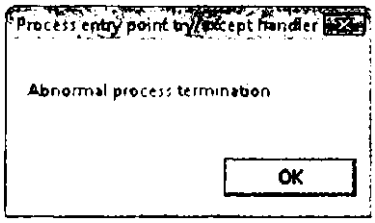
Код SEHTerm.exe проверяет, в какой системе он работает. Если это Windows Vista, следующее окно позволяет указать, следует ли защитить сбойную функцию блоком *try/except*.



Если щелкнуть кнопку Yes, блок *try/finally* будет защищен фильтром исключений, который всегда возвращает EXCEPTION_EXECUTE_HANDLER. При этом в случае возникновения исключения непременно начинается глобальная раскрутка, и при исполнении блока *finally* открывается окно следующего вида:



Перед завершением главного потока приложения с кодом -1 блок *except* выводит следующее сообщение:



Если же щелкнуть кнопку No, при возникновении исключения приложение завершится без исполнения блока *finally* (если вы не выберете запуск отладки).

```
SEHTerm.cpp

/******************************************************************************
Module:  SEHTerm.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#include "..\CommonFiles\CmnHdr.h"     /* See Appendix A. */
#include <windows.h>
#include <tchar.h>


//////////////////////////////////////////////////////////////////////////////


BOOL IsWindowsVista() {

   // Code from Chapter 4
   // Prepare the OSVERSIONINFOEX structure to indicate Windows Vista.
   OSVERSIONINFOEX osver = { 0 };
   osver.dwOSVersionInfoSize = sizeof(osver);
   osver.dwMajorVersion = 6;
   osver.dwMinorVersion = 0;
   osver.dwPlatformId = VER_PLATFORM_WIN32_NT;
```

```
   // Prepare the condition mask.
   DWORDLONG dwlConditionMask = 0;          // You MUST initialize this to 0.
   VER_SET_CONDITION(dwlConditionMask, VER_MAJORVERSION, VER_EQUAL);
   VER_SET_CONDITION(dwlConditionMask, VER_MINORVERSION, VER_EQUAL);
   VER_SET_CONDITION(dwlConditionMask, VER_PLATFORMID, VER_EQUAL);

   // Perform the version test.
   if (VerifyVersionInfo(&osver, VER_MAJORVERSION  | VER_MINORVERSION |
      VER_PLATFORMID, dwlConditionMask)) {
      // The host system is Windows Vista exactly.
      return(TRUE);
   } else {
      // The host system is NOT Windows Vista.
      return(FALSE);
   }
}


void TriggerException() {

   __try {
      int n = MessageBox(NULL, TEXT("Perform invalid memory access?"),
         TEXT("SEHTerm: In try block"), MB_YESNO);

      if (n == IDYES) {
         * (PBYTE) NULL = 5;  // This causes an access violation
      }
   }
   __finally {
      PCTSTR psz = AbnormalTermination()
         ? TEXT("Abnormal termination") : TEXT("Normal termination");
      MessageBox(NULL, psz, TEXT("SEHTerm: In finally block"), MB_OK);
   }

   MessageBox(NULL, TEXT("Normal function termination"),
      TEXT("SEHTerm: After finally block"), MB_OK);
}


int WINAPI _tWinMain(HINSTANCE, HINSTANCE, PTSTR, int) {

   // In Windows Vista, a global unwind occurs if an except filter
   // returns EXCEPTION_EXECUTE_HANDLER. If an unhandled exception
   // occurs, the process is simply terminated and the finally blocks
   // are not exectuted.
   if (IsWindowsVista()) {
```

```
      DWORD n = MessageBox(NULL, TEXT("Protect with try/except?"),
         TEXT("SEHTerm: workflow"), MB_YESNO);

      if (n == IDYES) {
         __try {
            TriggerException();
         }
         __except (EXCEPTION_EXECUTE_HANDLER) {
            // But the system dialog will not appear.
            // So, popup a message box.
            MessageBox(NULL, TEXT("Abnormal process termination"),
               TEXT("Process entry point try/except handler"), MB_OK);

            // Exit with a dedicated error code
            return(-1);
         }
      } else {
         TriggerException();
      }
   } else {
      TriggerException();
   }

   MessageBox(NULL, TEXT("Normal process termination"),
      TEXT("SEHTerm: before leaving the main thread"), MB_OK);

   return(0);
}


//////////////////////////////// End of File //////////////////////////////////
```

Оглавление

ГЛАВА 24

Фильтры и обработчики исключений

Исключение — это событие, которого вы не ожидали. В хорошо написанной программе не предполагается попыток обращения по неверному адресу или деления на нуль. И все же такие ошибки случаются. За перехват попыток обращения по неверному адресу и деления на нуль отвечает центральный процессор, возбуждающий исключения в ответ на эти ошибки. Исключение, возбужденное процессором, называется *аппаратным* (hardware exception). Далее мы увидим, что операционная система и прикладные программы способны возбуждать собственные исключения — *программные* (software exceptions).

При возникновении аппаратного или программного исключения операционная система дает вашему приложению шанс определить его тип и самостоятельно обработать. Синтаксис обработчика исключений таков:

```
_try {
    // защищенный блок
    …
}
__except (exception filter) {
    // обработчик исключений
    …
}
```

Обратите внимание на ключевое слово *__cxcept*. За блоком *try* всегда должен следовать либо блок *finnally*, либо блок *except*. Для данного блока *try* нельзя указать одновременно и блок *finally*, и блок *except*; к тому же за *try* не может следовать несколько блоков *finally* или *except*. Однако *try-finally* можно вложить в *try-except*, и наоборот.

Примеры использования фильтров и обработчиков исключений

В отличие от обработчиков завершения (рассмотренных в предыдущей главе), фильтры и обработчики исключений выполняются непосредственно операционной системой — нагрузка на компилятор при этом минимальна. В следующих разделах я расскажу, как обычно выполняются блоки *try-except*, как и когда операционная система проверяет фильтры исключений и в каких случаях она выполняет код обработчиков исключений.

Funcmeister1

Вот более конкретный пример блока *try-except*:

```
DWORD Funcmeister1() {
    DWORD dwTemp;

    // 1. Что-то делаем здесь
    …
    __try {
        // 2. Выполняем какую-то операцию
        dwTemp = 0;
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        // обрабатываем исключение; этот код никогда не выполняется
        …
    }

    // 3. Продолжаем что-то делать
    return(dwTemp);
}
```

В блоке *try* функции *Funcmeister1* мы просто присваиваем 0 переменной *dwTemp*. Такая операция не приведет к исключению, и поэтому код в блоке *except* никогда не выполняется. Обратите внимание на такую особенность: конструкция *try-finally* ведет себя иначе. После того как переменной *dwTemp* присваивается 0, следующим исполняемым оператором оказывается *return*.

Хотя ставить операторы *return*, *goto*, *continue* и *break* в блоке *try* обработчика завершения настоятельно не рекомендуется, их применение в этом блоке не приводит к снижению быстродействия кода или к увеличению его размера. Использование этих операторов в блоке *try*, связанном с блоком *except*, не вызовет таких неприятностей, как локальная раскрутка.

Funcmeister2

Попробуем модифицировать нашу функцию и посмотрим, что будет:

```
DWORD Funcmeister2() {
    DWORD dwTemp = 0;

    // 1. Что-то делаем здесь
    …
    _try {
        // 2. Выполняем какую-то операцию
        dwTemp = 5 / dwTemp;                // генерирует исключение
        dwTemp += 10;                       // никогда не выполняется
    }
    __except ( /* 3. Проверяем фильтр. */ EXCEPTION_EXECUTE_HANDLER) {
        // 4. Обрабатываем исключение.

        MessageBeep(0);
        …
    }

    // 5. Продолжаем что-то делать
    eturn(dwTemp);
}
```

Инструкция внутри блока *try* функции *Funcmeister2* пытается поделить 5 на 0. Перехватив это событие, процессор возбуждает аппаратное исключение. Тогда операционная система ищет начало блока *except* и проверяет выражение, указанное в качестве фильтра исключений; оно должно дать один из трех идентификаторов, определенных в заголовочном Windows-файле Excpt.h.

Табл. 24-1. Значения, возвращаемые фильтром исключений

Идентификатор	Значение
EXCEPTION_EXECUTE_HANDLER	1
EXCEPTION_CONTINUE_SEARCH	0
EXCEPTION_CONTINUE_EXECUTION	-1

Далее мы обсудим, как эти идентификаторы изменяют выполнение потока. Читая следующие разделы, посматривайте на блок-схему на рис. 24-1, которая иллюстрирует операции, выполняемые системой после генерации; исключения.

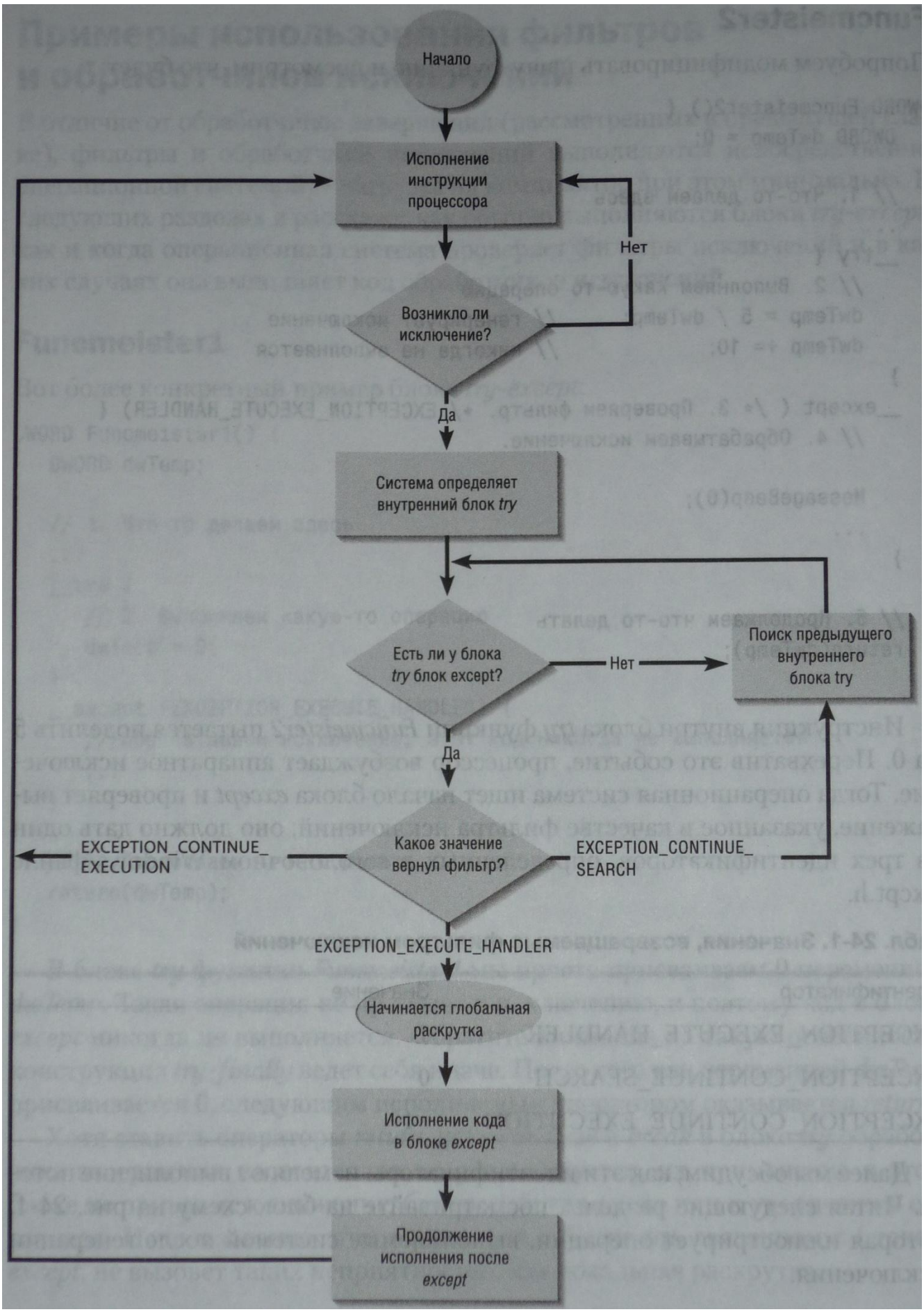


Рис. 24-1. *Так система обрабатывает исключения.*

EXCEPTION_EXECUTE_HANDLER

Фильтр исключений в *Funcmeister2* определен как EXCEPTION_EXECUTE_HANDLER. Это значение сообщает системе в основном вот что: «Я вижу это исключение; так и знал, что оно где-нибудь произойдет; у меня есть код для его обработки, и я хочу его сейчас выполнить». В этот момент система проводит глобальную раскрутку (о ней — немного позже), а затем управление передается коду внутри блока *except* (коду обработчика исключений). После его выполнения система считает исключение обработанным и разрешает программе продолжить работу. Этот механизм позволяет Windows-приложениям перехватывать ошибки, обрабатывать их и продолжать выполнение — пользователь даже не узнает, что была какая-то ошибка.

Но вот откуда возобновится выполнение? Поразмыслив, можно представить несколько вариантов.

Первый вариант. Выполнение возобновляется сразу за строкой, возбудившей исключение. Тогда в *Funcmeister2* выполнение продолжилось бы с инструкции, которая прибавляет к *dwTemp* число 10. Вроде логично, но на деле в большинстве программ нельзя продолжить корректное выполнение, если одна из предыдущих инструкций вызвала ошибку.

В нашем случае нормальное выполнение можно продолжить, но *Funcmeister2* в этом смысле не типична. Ваш код скорее всего структурирован так, что инструкции, следуклцие за той, где произошло исключение, ожидают от нее корректное значение. Например, у вас может быть функция, выделяющая блок памяти; тогда для операций с ним, несомненно, предусмотрена целая серия инструкций. Если блок памяти вышелить не удастся, все они потерпят неудачу, и программа повторно вызовет исключение.

Вот еще пример того, почему выполнение нельзя продолжить сразу после команды, возбудившей исключение. Заменим оператор языка С, дающий исключение в *Funcmeister2,* строкой:

```
malloc(5 / dwTemp);
```

Компилятор сгенерирует для нее машинные команды, которые выполняют деление, результат помещают в стек и вызывают *malloc.* Если попытка деления привела к ошибке, дальнейшее (корректное) выполнение кода невозможно. Система должна поместить что-то в стек, иначе он будет разрушен.

К счастью, Майкрософт не дает нам шанса возобновить выполнение со строки, расположенной вслед за возбудившей исключение. Это спасает нас от только что описанных потенциальных проблем.

Второй вариант. Выполнение возобновляется с той же команды, которая возбудила исключение. Этот вариант довольно интересен. Допустим, в блоке *except* присутствует оператор:

```
dwTemp = 2;
```

Тогда вы вполне могли бы возобновить выполнение с возбудившей исключение команды. На этот раз вы поделили бы 5 на 2, и программа спокой-

но продолжила бы свою работу. Иначе говоря, вы что-то меняете и заставляете систему повторить выполнение команды, возбудившей исключение. Но, применяя такой прием, нужно иметь в виду некоторые тонкости (о них — чуть позже).

Третий, и последний, вариант — приложение возобновляет выполнение с инструкции, следующей за блоком *except*. Именно так и происходит, когда фильтр исключений определен как EXCEPTION_EXECUTE_HANDLER. По окончании выполнения кода в блоке *except* управление передается на первую строку за этим блоком.

Некоторые полезные примеры

Допустим, вы хотите создать отказоустойчивое приложение, которое должно работать 24 часа в сутки и 7 дней в неделю. В наше время, когда программное обеспечение настолько усложнилось и подвержено влиянию множества непредсказуемых факторов, мне кажется, что без SEH просто нельзя создать действительно надежное приложение. Возьмем элементарный пример: функцию *strcpy* из библиотеки C:

```
char* strcpy(
    char* strDestination,
    const char* strSource);
```

Крошечная, давно известная и очень простая функция, да? Разве она может вызвать завершение процесса? Ну, если в каком-нибудь из параметров будет передан NULL (или любой другой недопустимый адрес), *strcpy* приведет к нарушению доступа, и весь процесс будет закрыт.

Создание абсолютно надежной функции *strcpy* возможно только при использовании SEH:

```
char* RobustStrCpy(char* strDestination, const char* strSource) {

    __try {
        strcpy(strDestination, strSource);
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        // здесь ничего не делаем
    }

    return(strDestination);
}
```

Все, что делает эта функция, — помещает вызов *strcpy* в SEH-фрейм. Если вызов *strcpy* проходит успешно, *RobustStrCpy* просто возвращает управление. Если же *strcpy* генерирует нарушение доступа, фильтр исключений возвращает значение EXCEPTION_EXECUTE_HANDLER, которое заставляет поток выполнить код обработчика. В функции *RobustStrCpy* обработчик не делает

ровным счетом ничего, и опять *RobustStrCpy* просто возвращает управление. Но она никогда не приведет к аварийному завершению процесса!

Поскольку вы не знаете, как реализована функция *strcpy*, вы также не имеете представления, какие исключения могут возникнуть в ходе ее исполнения. В описании функции упомянут лишь случай, когда параметр *strDestination* равен NULL или недопустим. А что, если адрес правильный, но буфер для хранения *strSource*, но размер буфера слишком мал? Если блок памяти, на который указывает *strDestination*, входит в блок большего размера, его содержимое может быть повреждено функцией strcpy. При этом также возможно нарушение доступа. Однако во время обработки исключения работа поврежденного процесса продолжается, что впоследствии может привести к краху, причины которого будет трудно установить, либо к возникновению «дыры» в защите. Мораль проста: обрабатывайте только те исключения, последствия которых вы в состоянии устранить, но не забывайте и про другие меры защиты от повреждения данных и брешей в защите (подробнее о безопасных строковых функциях см. в главе 2).

Рассмотрим другой пример. Вот функция, которая сообщает число отделенных пробелами лексем в строке:

```
int RobustHowManyToken(const char* str) {

    int nHowManyTokens = -1;  // значение, равное -1, сообщает о неудаче
    char* strTemp = NULL;     // предполагаем худшее

    __try {
        // создаем временный буфер
        strTemp = (char*) malloc(strlen(str) + 1);

        // копируем исходную строку во временный буфер
        strcpy(strTemp, str);

        // получаем первую лексему
        char* pszToken = strtok(strTemp, " ");

        // перечисляем все лексемы
        for (; pszToken != NULL; pszToken = strtok(NULL, " "))
            nHowManyTokens++;

        nHowManyTokens++;                    // добавляем 1, так как мы начали с -1
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        // здесь ничего не делаем
    }

    // удаляем временный буфер (гарантированная операция)
```

```
    free(strTemp);

    return(nHowManyTokens);
}
```

Эта функция создает временный буфер и копирует в него строку. Затем, вызывая библиотечную функцию *strtok,* она разбирает строку на отдельные лексемы. Временный буфер необходим из-за того, что *strtok* модифицирует анализируемую строку.

Благодаря SEH эта обманчиво простая функция справляется с любыми неожиданностями. Давайте посмотрим, как она работает в некоторых ситуациях.

Во-первых, если ей передается NULL (или любой другой недопустимый адрес), переменная *nHowManyTokens* сохраняет исходное значение -1. Вызов *strien* внутри блока *try* приводит к нарушению доступа. Тогда управление передается фильтру исключений, а от него — блоку *except,* который ничего не делает. После блока *except* вызывается *free,* чтобы удалить временный буфер в памяти. Однако он не был создан, и в данной ситуации мы вызываем/гее с передачей ей NULL Стандарт ANSI C допускает вызов *free* с передачей NULL, в каковом случае эта функция просто возвращает управление, так что ошибки здесь нет. В итоге *RobustHowManyToken* возвращает значение -1, сообщая о неудаче, и аварийного завершения процесса не происходит.

Во-вторых, если функция получает корректный адрес, но вызов *malloc* (внутри блока *try*) заканчивается неудачно и дает NULL, то обращение к *strcpy* опять приводит к нарушению доступа. Вновь активизируется фильтр исключений, выполняется блок *except* (который ничего не делает), вызывается *free* с передачей NULL (из-за чего она тоже ничего не делает), и *RobustHowManyToken* возвращает -1, сообщая о неудаче. Аварийного завершения процесса не происходит.

Наконец, допустим, что функции передан корректный адрес и вызов *malloc* прошел успешно. Тогда преуспеет и остальной код, а в переменную *nHowManyTokens* будет записано число лексем в строке. В этом случае выражение в фильтре исключений (в конце блока *try*) не оценивается, код в блоке *except* не выполняется, временный буфер нормально удаляется, и *nHowManyTokens* сообщает количество лексем в строке.

Функция *RobustHowManyToken* демонстрирует, как обеспечить гарантированную очистку ресурса, не прибегая к *try-finally*. Также гарантируется выполнение любого кода, расположенного за обработчиком исключения (если, конечно, функция не возвращает управление из блока *try,* но таких вещей вы должны избегать).

А теперь рассмотрим последний, особенно полезный пример использования SEH. Вот функция, которая дублирует блок памяти:

```
PBYTE RobustHemDup(PBYTE pbSrc, 8ize_t cb) {

    pbDup = NULL;                          // заранее предполагаем неудачу
```

```
    _try {

        // создаем буфер для дублированного блока памяти
        pbDup = (PBYTE) malloc(cb);

        memcpy(pbDup, pbSrc, cb);
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        free(pbDup);
        pbDup = NULL;
    }

    return(pbDup);
}
```

Эта функция создает буфер в памяти и копирует в него байты из исходного блока. Затем она возвращает адрес этого дубликата (или NULL, если вызов закончился неудачно). Предполагается, что буфер освобождается вызывающей функцией — когда необходимость в нем отпадает. Это первый пример, где в блоке *except* понадобится какой-то код. Давайте проанализируем работу этой функции в различных ситуациях.

- Если в параметре *pbSrc* передается некорректный адрес или если вызов *malloc* завершается неудачно (и дает NULL), *memcpy* возбуждает нарушение доступа. А это приводит к выполнению фильтра, который передает управление блоку *except*. Код в блоке *except* освобождает буфер памяти и устанавливает *pbDup* в NULL, чтобы вызвавший эту функцию поток узнал о ее неудачном завершении. (Не забудьте, что стандарт ANSI C допускает передачу NULL функции *free.)*
- Если в параметре *pbSrc* передается корректный адрес и вызов *malloc* проходит успешно, функция возвращает адрес только что созданного блока памяти.

Глобальная раскрутка

Когда фильтр исключений возвращает EXCEPTION_EXECUTE_HANDLER, системе приходится проводить глобальную раскрутку. Она приводит к продолжению обработки всех незавершенных блоков *try-finally*, выполнение которых началось вслед за блоком *try-except*, обрабатывающим данное исключение. Блок-схема на рис. 24-2 поясняет, как система осуществляет глобальную раскрутку. Посматривайте на эту схему, когда будете читать мои пояснения к следующему примеру.

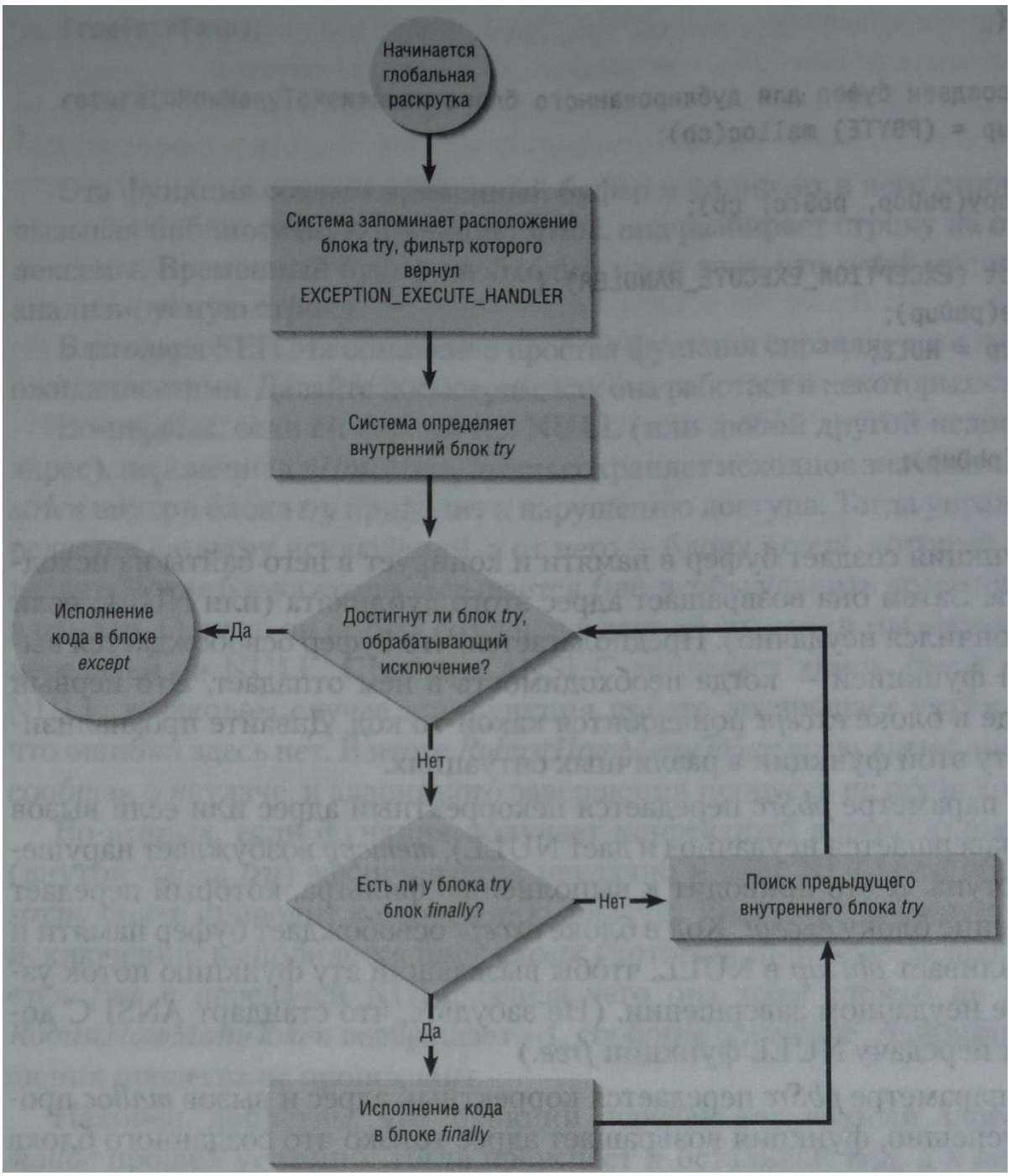


Рис. 24-2. *Так система проводит глобальную раскрутку*

```
void FuncOStimpy1() {

    // 1. Что-то делаем здесь
    …
    _try {
        // 2. Вызываем другую функцию
        FuncORen1();

        // этот код никогда не выполняется
    }
    __except ( /* 6. Проверяем фильтр исключений. */ EXCEPTION_EXECUTE_HANDLER) {
        // 8. После раскрутки выполняется этот обработчик
```

```
        MessageBox(…);
    }

    // 9. Исключение обработано - продолжаем выполнение
    …
}
void FuncORen1() {
    DWORD dwTemp = 0;

    // 3. Что-то делаем здесь
    __try {
        // 4. Запрашиваем разрешение на доступ к защищенным данным
        WaitForSingleObJect(g_hSem, INFINITE);

        // 5. Изменяем данные, и здесь генерируется исключение
        g_dwProtectedData = 5 / dwTemp;
    }
    __finally {
        // 7. Происходит глобальная раскрутка, так как
        // фильтр возвращает EXCEPTION_EXECUTE_HANDLER

        // Даем и другим попользоваться защищенными данными
        ReleaseSemaphore(g_hSem, 1, NULL);
    }

    // сюда мы никогда не попадем
    …
}
```

FuncOStimpy1 и *FuncORen1* иллюстрируют самые запутанные аспекты структурной обработки исключений. Номера в начале комментариев показывают порядок выполнения, в котором сходу не разберешься, но возьмемся за руки и пойдем вместе.

FuncOStimpy1 начинает выполнение со входа в свой блок *try* и вызова *FuncORen1*. Последняя тоже начинает со входа в свой блок *try* и ждет освобождения семафора. Завладев им, она пытается изменить значение глобальной переменной *g_dwProtectedData*. Деление на нуль возбуждает исключение. Система, перехватив управление, ищет блок *try*, которому соответствует блок *except*. Поскольку блоку *try* функции *FuncORen1* соответствует блок *finally*, система продолжает поиск и находит блок *try* в *FuncOStimpy1*, которому как раз и соответствует блок *except*.

Тогда система проверяет значение фильтра исключений в блоке *except* функции *FuncOStimpy1*. Обнаружив, что оно — EXCEPTION_EXECUTE_HANDLER, система начинает глобальную раскрутку с блока *finally* в функции *FuncORen1*. Заметьте: раскрутка происходит до выполнения кода из

блока *except* в *FuncOStimpy1*. Осуществляя глобальную раскрутку, система возвращается к последнему незавершенному блоку *try* и ищет теперь блоки *try*, которым соответствуют блоки *finally*. В нашем случае блок *finnally* находится в функции *FuncORen1*. Мощь SEH по-настоящему проявляется, когда система выполняет код *finally* в *FuncORen1*. Из-за его выполнения семафор освобождается, и поэтому другой поток получает возможность продолжить работу. Если бы вызов *ReleaseSemaphore* в блоке *finally* отсутствовал, семафор никогда бы не освободился.

Завершив выполнение блока *finally*, система ищет другие незавершенные блоки *finally*. В нашем примере таких нет. Дойдя до блока *except*, обрабатывающего исключение, система прекращает восходящий проход по цепочке блоков. В этой точке глобальная раскрутка завершается, и система может выполнить код в блоке *except*.

Вот так и работает структурная обработка исключений. Вообще-то, SHE — штука весьма трудная для понимания: выполнение вашего кода вмешивается операционная система. Код больше не выполняется последовательно, сверху вниз; система устанавливает свой порядок — сложный, но все же предсказуемый. Поэтому, следуя блок-схемам на рис. 24-1 и 24-2, вы сможете уверенно применять SEH.

Чтобы лучше разобраться в порядке выполнения кода, посмотрим на происходящее под другим углом зрения. Возвращая EXCEPTION_EXECUTE_HANDLER, фильтр сообщает операционной системе, что регистр указателя команд данного потока должен быть установлен на код внутри блока *except*. Однако этот регистр указывал на код внутри блока *try* функции *FuncORen1*. А из главы 23 вы должны помнить, что всякий раз, когда поток выходит из блока *try*, соответствующего блоку *finally*, обязательно выполняется код в этом блоке *finally*. Глобальная раскрутка как раз и является тем механизмом, который гарантирует соблюдение этого правила при любом исключении.

> **Внимание!** Одно из новшеств Windows Vista состоит в следующем. Если в вашем коде нет внешнего блока *try/except* (EXCEPTION_EXECUTE_HANDLER) и в одном из внутренних блоков *try/finally* возникает исключение, процесс просто закрывается без глобальной раскрутки, код из внутренних блоков *finally* при этом не исполняется. В прежних версиях Windows перед остановкой процесса выполнялась глобальная раскрутка, что давало шансы на исполнение блоков *finally*. Подробнее о необработанных исключениях см. в следующей главе.

Остановка глобальной раскрутки

Глобальную раскрутку, осуществляемую системой, можно остановить, если в блок *finally* включить оператор *return*. Взгляните:

```
void FuncMonkey() {
    __try {
        FuncFish();
    }
    __except (EXCEPTION_EXECUTE_HANDLER) {
        MessageBeep(0);
    }
    MessageBox(…);
}

void FuncFish() {
    FuncPheasant();
    MessageBox(…);
}

void FuncPheasant() {

    __try {
        strcpy(NULL, NULL);
    }

    __finally {
        return;
    }
}
```

При вызове *strcpy* в блоке *try* функции *FuncPheasant* из-за нарушения доступа к памяти генерируется исключение. Как только это происходит, система начинает просматривать код, пытаясь найти фильтр, способный обработать данное исключение. Обнаружив, что фильтр в *FuncMonkey* готов обработать его, система приступает к глобальной раскрутке. Она начинается с выполнения кода в блоке *finally* функции *FuncPheasant.* Но этот блок содержит оператор *return.* Он заставляет систему прекратить раскрутку, и *FuncPheasant* фактически завершается возвратом в *FuncFish,* которая выводит сообщение на экран. Затем *FuncFish* возвращает управление *FuncMonkey,* и та вызывает *MessageBox.*

Заметьте: код блока *except* в *FuncMonkey* никогда не вызовет *MessageBeep.* Оператор *return* в блоке *finally* функции *FuncPheasant* заставит систему вообще прекратить раскрутку, и поэтому выполнение продолжится так, будто ничего не произошло.

Избегайте операторов *return* в блоках *finally.* Чтобы помочь в выявлении таких ситуаций, компилятор C++ генерирует предупреждение C4532:

```
'return' : jump out of _finally block has undefined behavior during
termination handling.
```

EXCEPTION_CONTINUE_EXECUTION

Давайте приглядимся к тому, как фильтр исключений получает один из трех идентификаторов, определенных в файле Excpt.h. В *Funcmeister2* идентификатор EXCEPTION_EXECUTE_HANDLER «зашит> (простоты ради) в код самого фильтра, но вы могли бы вызывать там функцию, которая определяла бы нужный идентификатор. Взгляните:

```
TCHAR g_szBuffer[100];

voidFunclinRoosevelt1() {
    int x = 0;
    TCHAR *pchBuffer = NULL;

    __try {
        *pchBuffer = TEXT('J');
        x = 5 / x;
    }
    __except (OilFilter1(&pchBuffer)) {
        MessageBox(NULL, TEXT("An exception occurred"), NULL, MB_0K);
    }
    MessageBox(NULL, TEXT("Function completed"), NULL, MB_0K);
}

LONG OilFilter1(TCHAR **ppchBuffer) {
    if (*ppchBuffer == NULL) {
        *ppchBuffer = g_szBuffer;
        return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(EXCEPTION_EXECUTE_HANDLER);
}
```

В первый раз проблема возникает, когда мы пытаемся поместить *J* в буфер, на который указывает *pchBuffer*. К сожалению, мы не определили *pchBuffer* как указатель на наш глобальный буфер *g_szBuffer* — вместо этого он указывает на NULL. Процессор генерирует исключение и вычисляет выражение в фильтре исключений в блоке *except*, связанном с блоком *try*, в котором и произошло исключение. В блоке *except* адрес переменной *pchBuffer* передается функции *OilFilter1*.

Получая управление, *OilFilter1* проверяет, не равен ли *pchBuffer* значению NULL, и, если да, устанавливает его так, чтобы он указывал на глобальный буфер *g_szBuffer*. Тогда фильтр возвращает EXCEPTION_CONTINUE_ EXECUTION. Обнаружив такое значение выражения в фильтре, система возвращается к инструкции, вызвавшей исключение, и пытается выполнить ее снова. На этот раз все проходит успешно, и будет записана в первый байт буфера *g_szBuffer*.

Когда выполнение кода продолжится, мы опять столкнемся с проблемой в блоке *try* ~- теперь это деление на нуль. И вновь система вычислит выражение фильтра исключений. На этот раз *pchBuffer* не равен NULL, и поэтому *OilFilter1* вернет EXCEPTION_EXECUTE_HANDLER, что подскажет системе выполнить код в блоке *except,* и на экране появится окно с сообщением об исключении.

Как видите, внутри фильтра исключений можно проделать массу всякой работы. Разумеется, в итоге фильтр должен вернуть один из трех идентификаторов, но он может выполнять и другие нужные вам действия. Учтите, однако, что процесс, в котором возникло исключение, может быть нестабильным, поэтому код внутри фильтра должен быть как можно более простым. Например, в случае повреждения кучи фильтр, пытающийся выполнить множество операций, может привести к зависанию либо «тихому» завершению процесса.

Будьте осторожны с EXCEPTION_CONTINUE_EXECUTION

Будет ли удачной попытка исправить ситуацию в только что рассмотренной функции и заставить систему продолжить выполнение программы, зависит от типа процессора, от того, как компилятор генерирует машинные команды при трансляции операторов C/C++, и от параметров, заданных компилятору. Компилятор мог сгенерировать две машинные команды для оператора:

```
*pchBuffer = TEXT('J');
```

которые выглядят так:

```
MOV EAX, DWORD PTR[pchBuffer]          // адрес помещается в регистр EAX
MOV WORD PTR[EAX], 'J'                 // символ J записывается по адресу
                                       // из регистра EAX
```

Последняя команда и возбудила бы исключение. Фильтр исключений, перехватив его, исправил бы значение *pchBuffer* и указал бы системе повторить эту команду. Но проблема в том, что содержимое регистра не изменится так, чтобы отразить новое значение *pchBuffer,* и поэтому повторение команды снова приведет к исключению. Вот и бесконечный цикл!

Выполнение программы благополучно возобновится, если компилятор оптимизирует код, но может прерваться, если компилятор код не оптимизирует. Обнаружить такой «жучок» очень трудно, и — чтобы определить, откуда он взялся в программе, — придется анализировать ассемблерный текст, сгенерированный для исходного кода. Вывод: будьте крайне осторожны, возвращая EXCEPTION_CONTINUE_EXECUTION из фильтра исключений.

EXCEPTION_CONTINUE_EXECUTION всегда срабатывает лишь в одной ситуации: при передаче памяти зарезервированному региону. О том, как зарезервировать большую область адресного пространства, а потом передавать ей память лишь по мере необходимости, я рассказывал в главе 15.

Соответствующий алгоритм демонстрировала программа-пример VMAlloc. На основе механизма SEH то же самое можно было бы реализовать гораздо эффективнее (и не пришлось бы все время вызывать функцию *VirtualAlloc).*

В главе 16 мы говорили о стеках потоков. В частности, я показал, как система резервирует для стека потока регион адресного пространства размером 1 Мб и как она автоматически передает ему новую память по мере разрастания стека. С этой целью система создает SEH-фрейм. Когда поток пытается задействовать несуществующую часть стека, генерируется исключение. Системный фильтр определяет, что исключение возникло из-за попытки обращения к адресному пространству, зарезервированному под стек, вызывает функцию *VirtualAlloc* для передачи дополнительной памяти стеку потока и возвращает EXCEPTION_CONTINUE_EXECUTION. После этого машинная команда, пытавшаяся обратиться к несуществующей части стека, благополучно выполняется, и поток продолжает свою работу.

Механизмы использования виртуальной памяти в сочетании со структурной обработкой исключений позволяют создавать невероятно «шустрые» приложения. Программа-пример Spreadsheet в следующей главе продемонстрирует, как на основе SEH эффективно реализовать управление памятью в электронной таблице. Этот код выполняется чрезвычайно быстро.

EXCEPTION_CONTINUE_SEARCH

Приведенные до сих пор примеры были ну просто детскими. Чтобы немного встряхнуться, добавим вызов функции:

```
TCHAR g_szBuffer[100];

void FunclInRoosevelt2() {
    TCHAR *pchBuffer = NULL;

    __try {
        FuncAtude2(pchBuffer);
    }
    __except (OilFilter2(&pchBuffer)) {
        MessageBox(…);
    }
}

void FuncAtude2(TCHAR *sz) {
    *sz = TEXT('\0');
}

LONG OilFilter2 (TCHAR **ppchBuffer) {
    if (*ppchBuffer == NULL) {
```

```
        *ppchBuffer = g_szBuffer;
        return(EXCERTION_CONTINUE_EXECUTION);
    }
    return(EXCEPTION_EXECUTE_HANDLER);
}
```

При выполнении *FunclinRoosevelt2* вызывается *FuncAtude2,* которой передается NULL. Последняя приводит к исключению. Как и раньше, система проверяет выражение в фильтре исключений, связанном с последним исполняемым блоком *try.* В нашем примере это блок *try* в *FunclinRoosevelt2,* поэтому для оценки выражения в фильтре исключений система вызывает *OilFilter2* (хотя исключение возникло в *FuncAtude2).*

Замесим ситуацию еще круче, добавив другой блок *try-except.*

```
TCHAR g_szBuffer[100];

void FunclinRoosevelt3() {

    TCHAR *pchBuffer = NULL;

    __try {
        FuncAtude3(pchBuffer);
    }
    __except (OilFilter3(&pchBuffer)) {
        MessageBox(…);
    }
}

void FuncAtude3(TCHAR *sz) {
    _try {
        *sz = TEXT('\0');
    }
    __except (EXCEPTION_CONTINUE_SEARCH) {
        // этот код никогда не выполняется
        …
    }
}

LONG OilFilter3(TCHAR **ppchBuffer) {
    if (*ppchBuffer == NULL) {
        *ppchBuffer = g_szBuffer;
        return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(EXCEPTION_EXECUTE_HANDLER);
}
```

Теперь, когда *FuncAtude3* пытается занести 0 по адресу NULL, по-прежнему возбуждается исключение, но в работу вступает фильтр исключений из

FuncAtude3. Значение этого очень простого фильтра — EXCEPTION_CONTINUE_SEARCH. Данный идентификатор указывает системе перейти к предыдущему блоку *try,* которому соответствует блок *except,* и обработать его фильтр.

Так как фильтр в *FuncAtude3* дает EXCEPTION_CONTINUE_SEARCH, система переходит к предыдущему блоку *try(B* функции *FunclinRoosevelt3)* и вычисляет его фильтр *OUMlter3.* Обнаружив, что значение *pchBuffer* равно NULL, *OilFilter3* меняет его так, чтобы оно указывало на глобальный буфер, и сообщает системе возобновить выполнение с инструкции, вызвавшей исключение. Это позволяет выполнить код в блоке *try* функции *FuncAtude3,* но, увы, локальная переменная *sz* в этой функции не изменена, и возникает новое исключение. Опять бесконечный цикл! *OilFilter3* увидит, что *pchBuffer* не равен NULL и вернет EXCEPTION_EXECUTE_HANDLER, позволив системе возобновить исполнение с блока *except* Таким образом, код в блоке except функции *FunclinRoosevelt3* будет исполнен.

Заметьте, я сказал, что система переходит к последнему исполнявшемуся блоку try, которому соответствует блок *except,* и проверяет его фильтр. Это значит, что система пропускает при просмотре цепочки блоков любые блоки *try,* которым соответствуют блоки *finally* (а не *except).* Причина этого очевидна: в блоках *finally* нет фильтров исключений, а потому и проверять в них нечего. Если бы в последнем примере *FuncAtude3* содержала вместо *except* блок *finally,* система начала бы проверять фильтры исключений с *OilFilter3* в *FunclinRoosevelt3.*

Дополнительную информацию об EXCEPTION_CONTINUE_SEARCH см. в главе 25.

Функция *GetExceptionCode*

Часто фильтр исключений должен проанализировать ситуацию, прежде чем определить, какое значение ему вернуть. Например, ваш обработчик может знать, что делать при делении на нуль, но не знать, как обработать нарушение доступа к памяти. Именно поэтому фильтр отвечает за анализ ситуации и возврат соответствующего значения.

Этот фрагмент иллюстрирует метод, позволяющий определять тип исключения:

```
_try {
    x = 0;
    У = 4 / x;      // переменная у используется, поэтому она осталась в коде
    …
}

__except((GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZER0) ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    // обработка деления на нуль
}
```

Встраиваемая функция *GetExceptionCode* возвращает идентификатор типа исключения:

```
DWORD GetExceptionCode();
```

Ниже приведен список всех предопределенных идентификаторов исключений с пояснением их смысла (информация взята из документации Platform SDK). Эти идентификаторы содержатся в заголовочном файле WmBase.h. Я сгруппировал исключения по категориям.

Исключений, связанные с памятью

- **EXCEPTION_ACCESS_VIOLATION.** Поток пытался считать или записать по виртуальному адресу, не имея на то необходимых прав. Это самое распространенное исключение.
- **EXCEPTION_DATATYPE_MISALIGNMENT.** Поток пытался считать или записать невыровненные данные на оборудовании, которое не поддерживает автоматическое выравнивание. Например, 16-битные значения должны быть выровнены по двухбайтовым границам, 32-битные — по четырехбайтовым и т. д.
- **EXCEPTION_ARRAY_BOUNDS_EXCEEDED.** Поток пытался обратиться к элементу массива, индекс которого выходит за границы массива; при этом оборудование должно поддерживать такой тип контроля.
- **EXCEPTION_IN_PAGE_ERROR.** Ошибку страницы нельзя обработать, так как файловая система или драйвер устройства сообщили об ошибке чтения.
- **EXCEPTION_GUARD_PAGE.** Поток пытался обратиться к странице памяти с атрибутом защиты PAGE_GUARD. Страница становится доступной, и генерируется данное исключение.
- **EXCEPTION_STACK_OVERFLOW.** Стек, отведенный потоку, исчерпан.
- **EXCEPTION_ILLEGAL_INSTRUCTION.** Поток выполнил недопустимую инструкцию. Это исключение определяется архитектурой процессора; можно ли перехватить выполнение неверной инструкции, зависит от типа процессора.
- **EXCEPTION_PRIV_INSTRUCTION.** Поток пытался выполнить инструкцию, недопустимую в данном режиме работы процессора.

Исключения, связанные с обработкой самих исключений

- **EXCEPTION_INVALID_DISPOSITION.** Фильтр исключений вернул значение, отличное от EXCEPTION_EXECUTE_HANDLER, EXCEPTION_CONTINUE_SEARCH или EXCEPTION_CONTINUE_EXECUTION.
- **EXCEPTION_NONCONTINUABLE_EXCEPTION.** Фильтр исключений вернул EXCEPTION_CONTINUE_EXECUTION в ответ на невозобновляемое исключение (noncontinuable exception).

Исключения, связанные с отладкой

- **EXCEPTION_BREAKPOINT**. Встретилась точка прерывания (останова).
- **EXCEPTION_SINGLE_STEP**. Трассировочная ловушка или другой механизм пошагового исполнения команд подал сигнал о выполнении одной команды.
- **EXCEPTION_INVALID_HANDLE**. В функцию передан недопустимый описатель.

Исключения, связанные с операциями над целыми числами

- **EXCEPTION_INT_DIVIDE_BY_ZERO**. Поток пытался поделить число целого типа на делитель того же типа, равный 0.
- **EXCEPTION_FLOAT_OVERFLOW**. Операция над целыми числами вызвала перенос старшего разряда результата.

Исключения, связанные с операциями над вещественными числами

- **EXCEPTION_FLT_DENORMAL_OPERAND** Один из операндов в операции над числами с плавающей точкой (вещественного типа) не нормализован. Ненормализованными являются значения, слишком малые для стандартного представления числа с плавающей точкой.
- **EXCEPTION_FLT_DIVIDE_BY_ZERO**. Поток пытался поделить число вещественного типа на делитель того же типа, равный 0.
- **EXCEPTION_FLT_INEXACT_RESULT**. Результат операции над числами с плавающей точкой нельзя точно представить в виде десятичной дроби.
- **EXCEPTION_FLT_INVALID_OPERATION**. Любое другое исключение, относящееся к операциям над числами с плавающей точкой и не включенное в этот список.
- **EXCEPTION_FLT_OVERFLOW**. Порядок результата операции над числами с плавающей точкой превышает максимальную величину для указанного типа данных
- **EXCEPTION_FLT_STACK_CHECK.** Переполнение стека или выход за его нижнюю границу в результате выполнения операции над числами с плавающей точкой.
- **EXCEPTION_FLT_UNDERFLOW.** Порядок результата операции над числами с плавающей точкой меньше минимальной величины для указанного типа данных.

Встраиваемую функцию *GetExceptionCode* можно вызвать только из фильтра исключений (между скобками, которые следуют за *except)* или из *обработчика* исключений. Скажем, такой код вполне допустим:

```
__try {
    y  = 0;
```

```
    x = 4 / y;
}

__except (
    ((GetExceptionCode() (EXCEPTION_ACCESS_VIOLATION) ||
    (GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO)) ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {

    switch (GetExceptionCode()) {
        case EXCEPTION_ACCESS_VIOLATION:
            // обработка нарушения доступа к памяти
            …
            break;

        case EXCEPTION_INT_DIVIDE_BY_ZERO:
            // обработка деления целого числа на нуль
            …
            break;
    }
}
```

Однако *GetExceptionCode* нельзя вызывать из функции фильтра исключений. Компилятор помогает вылавливать такие ошибки и обязательно сообщит о таковой, если вы попытаетесь скомпилировать, например, следующий код:

```
__try {
    y = 0;
    x = 4 / y;
}

__except (CoffeeFilter()) {
    // обработка исключения
    …
}

LONG CoffeeFilter (void) {
    // ошибка при компиляции: недопустимый вызов GetExceptionCode
    return((GetExceptionCode() == EXCEPTION_ACCESS_VIOLATION) ?
        EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}
```

Нужного эффекта можно добиться, переписав код так:

```
_try {
    y = 0;
    x = 4 / y;
}
```

```
__except (CoffeeFilter(GetExceptionCode())) {

    // обработка исключения
    …
}

LONG CoffeeFilter (DWORD dwExceptionCode) {
    return((dwExceptionCode == EXCEPTION_ACCESS_VIOLATION) ?
        EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH);
}
```

Коды исключений формируются по тем же правилам, что и коды ошибок, определенные в файле WinError.h. Каждое значение типа DWORD разбивается на поля, как показано в таблице 24-2.

Табл. 24-2. Поля кода ошибки

Биты	31-30	29	28	27-16	15
Содержимое:	Код степени «тяжести» (severity)	Кем определен — Майкрософт или пользователем	Зарезервирован	Код подсистемы (facility code)	Код исключения
Значение	0 = успех 1 = информация	0 = Майкрософт 1 = пользователь	Должен быть 0	Определяется Майкрософт	Определяется Майкрософт
или	2 = предупреждение 3 = ошибка		(см. таблицу ниже)		пользователем

На сегодняшний день определены такие коды подсистемы.

Табл. 24-3. Коды подсистемы

Код подсистемы	Значение	Код подсистемы	Значение
FACILITY_RPC	1	FACILITY_HTTP	25
FACIUTY_DISPATCH	2	FACILITY_USERMODE_ COMMONLOG	26
FACILITY_STORAGE	3	FACILITY_USERMODE_FILTER_MANAGER	31
FACILITY_ITF	4	FACILITY_BACKGROUNDCOPY	32
FACILITY_WIN32	7	FACILITY_CONFIGURATION	33
FACILITY_WINDOWS	8	FACILITY_STATE_MANAGEMENT	34
FACILITY_SECURITY	9	FACILITY METADIRECTORY	35

Табл. 24-3. *(окончание)*

Код подсистемы	Значение	Код подсистемы	Значение
FACILITY_CONTROL	10	FACILITY_WINDOWSUPDATE	36
FACILITY_CERT	11	FACILITY_DIRECTORYSERVICE	37
FACILITY_INTERNET	12	FACILITY_GRAPHICS	38
FACILITY_MEDIA_SERVER	13	FACILITY_SHELL	39
FACILITY_MSMQ	14	FACILITY_TPM_SERVICES	40
FACILITY_SETUPAPI	15	FACILITY_TPM_SOFTWARE	41
FACILITY_SCARD	16	FACILITY_PLA	48
FACILITY_COMPLUS	17	FACILITY_FVE	49
FACILITY_AAF	18	FACILITY_FWP	50
FACILITY_URT	19	FACILITY_WINRM	51
FACILITY_ACS	20	FACILITY_NDIS	52
FACILITY_DPLAY	21	FACILITY_USERMODE_HYPERVISOR	53
FACILITY_UMI	22	FACILITY_CMI	54
FACILITY_SXS	23	FACILITY_WINDOWS_DEFENDER	80

Разберем на части, например, код исключения EXCEPTION_ACCESS_VIOLATION. Если вы посмотрите его значение в файле WmBase.h, то увидите, что оно равно 0xC0000005:

```
   C    0    0    0    0    0    0    5    (в шестнадцатеричном виде)
1100 0000 0000 0000 0000 0000 0000 0101    (в двоичном виде)
```

Биты 30 и 31 установлены в 1, указывая, что нарушение доступа является ошибкой (поток не может продолжить выполнение). Бит 29 равен 0, а это значит, что данный код определен Майкрософт. Бит 28 равен 0, так как зарезервирован на *будущее.* Биты 16-27 равны 0, сообщая код подсистемы FACILITY_NULL (нарушение доступа может произойти в любой подсистеме операционной системы, а не в какой-то одной). Биты 0-15 дают значение 5, которое означает лишь то, что Майкрософт присвоила исключению, связанному с нарушением доступа, код 5.

Функция *GetExceptionInformation*

Когда возникает исключение, операционная система заталкивает в стек соответствующего потока структуры EXCEPTION_RECORD, CONTEXT и EXCEPTION_POINTERS.

EXCEPTION_RECORD содержит информацию об исключении, независимую от типа процессора, а CONTEXT — машинно-зависимую информацию об этом исключении. В структуре EXCEPTION_POWTERS всего два элемента — указатели на помещенные в стек структуры EXCEPTION_ RECORD и CONTEXT:

```
typedef Struct _EXCEPTION_P0INTERS {
    PEXCEPTION_RECORD ExceptionRecord;
    PCONTEXT ContextRecord;
} EXCEPTION_P0INTERS, *PEXCEPTION_POINTERS;
```

Чтобы получить эту информацию и использовать ее в программе, вызовите *GetExceptionInformation:*

```
PEXCEPTION_POINTERS GetExceptionInformation();

PEXCEPTION_POINTERS GetExceptionInfornation();
```

Эта встраиваемая функция возвращает указатель на структуру EXCEPTION_POINTERS.

Самое важное в *GetExceptionInformation* то, что ее можно вызывать только в фильтре исключений и больше. Нигде, потому что структуры CONTEXT, EXCEPTION_RECORD и EXCEPTION_POINTERS существуют лишь во время обработки фильтра исключений. Когда управление переходит к обработчику исключений, эти данные в стеке разрушаются.

Если вам нужно получить доступ к информации об исключении из обработчика — сохраните структуру EXCEPTION_RECORD и/или CONTEXT (на которые указывают элементы структуры EXCEPTION_POINTERS) в объявленных вами переменных. Вот пример сохранения этих структур:

```
void FuncSkunk() {
    // объявляем переменные, которые мы сможем потом использовать
    // для сохранения информации об исключении (если оно произойдет)
    EXCEPTION_RECORD SavedExceptRec;
    CONTEXT SavedContext;
    …
    __try {
        …
    }

    __except (
        SavedExceptRec =
            *(GetExceptionInformation())->ExceptionRecord,
        SavedContext =
            *(GetExceptionInformation())->ContextRecord,
        EXCEPTION_EXECUTE_HANDLER) {
        // мы можем теперь использовать переменные SavedExceptRec
        // и SavedContext в блоке обработчика исключений
```

```
        switch (SavedExceptRec.ExceptionCode) {
            …
        }
    }
}
```

В фильтре исключений применяется оператор-запятая (,) — мало кто из программистов знает о нем. Он указывает компилятору, что выражения, отделенные запятыми, следует выполнять слева направо. После вычисления всех выражений возвращается результат последнего из них — крайнего справа.

В *FuncSkunk* сначала вычисляется выражение слева, что приводит к сохранению находящейся в стеке структуры EXCEPTION_RECORD в локальной переменной *SavedExceptRec*. Результат этого выражения является значением *SavedExceptRec*. Но он отбрасывается, и вычисляется выражение, расположенное правее. Это приводит к сохранению размещенной в стеке структуры CONTEXT в локальной переменной *SavedContext*. И снова результат — значение *SavedContext* — отбрасывается, и вычисляется третье выражение. Оно равно EXCEPTION_EXECUTE_HANDLER — это и будет результатом всего выражения в скобках.

Так как фильтр возвращает EXCEPTION_EXECUTE_HANDLER, выполняется код в блоке *except*. К этому моменту переменные *SavedExceptRec* и *SavedContext* уже инициализированы, и их можно использовать в данном блоке. Важно, чтобы переменные *SavedExceptRec* и *SavedContext* были объявлены вне блока *try*.

Вероятно, вы уже догадались, что элемент *ExceptionRecord* структуры EXCEPTION_POINTERS указывает на структуру EXCEPTION_RECORD:

```
typedef struct _EXCEPTION_RECORD {
    DWORD ExceptionCode;
    DWORD ExceptionFlags;
    struct _EXCEPTION_RECORD *ExceptionRecord;
    PVOID ExceptionAddress;
    DWORD NumberParameters;
    ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD;
```

Структура EXCEPTION_RECORD содержит подробную машинно-независимую информацию ***о*** последнем исключении. Вот что представляют собой ее элементы.

- *ExceptionCode* — код исключения. Это информация, возвращаемая функцией *GetExceptionCode*.
- *ExceptionMags* — флаги исключения. На данный момент определено только два значения: 0 (возобновляемое исключение) и EXCEPTION_NONCONTINUABLE (невозобновляемое исключение). Любая попытка возоб-

новить работу программы после невозобновляемого исключения генерирует исключение EXCEPTION_NONCONTINUABLE_EXCEPTION.

- *ExceptionRecord* — указатель на структуру EXCEPTION_RECORD, содержащую информацию о другом необработанном исключении. При обработке одного исключения может возникнуть другое. Например, код внутри фильтра исключений может попытаться выполнить деление на нуль. Когда возникает серия вложенных исключений, записи с информацией о них могут образовывать связанный список. Исключение будет вложенным, если оно генерируется при обработке фильтра. В отсутствие необработанных исключений *ExceptionRecord* равен NULL.
- *ExceptionAddress* — адрес машинной команды, при выполнении которой произошло исключение.
- *NumberParameters* — количество параметров, связанных с исключением (0-15). Это число заполненных элементов в Массиве *ExceptionInformation*. Почти для всех исключений значение этого элемента равно 0.
- *ExceptionInformation* — массив дополнительных аргументов, описывающих исключение. Почти для всех исключений элементы этого массива не определены.

Последние два элемента структуры EXCEPTION_RECORD сообщают фильтру дополнительную информацию об исключении. Сейчас такую информацию дает только один тип исключений: EXCEPTION_ACCESS_VIOLATION. Все остальные дают нулевое значение в элементе *NumberParameters*. Проверив его, вы узнаете, надо ли просматривать массив *ExceptionInformation*.

При исключении EXCEPTION_ACCESS_VIOLATION элемент *ExceptionInformation[0]* содержит флаг, указывающий тип операции, которая вызвала нарушение доступа. Если его значение равно 0, поток пытался читать недоступные ему данные; 1 — записывать данные по недоступному ему адресу. Элемент *ExceptionInformation[1]* определяет адрес недоступных данных. Когда защитный механизм Data Execution Prevention (DEP) обнаруживает попытку исполнения кода, хранящегося в странице, для которой исполнение запрещено, генерируется исключение и в *ExceptionInformation[0] запи*сывается 2 для IA-64 и 8 в противном случае.

Эта структура позволяет писать фильтры исключений, сообщающие значительный объем информации о работе программы. Можно создать, например, такой фильтр:

```
__try {
    …
}
__except(ExpFltr(GetExceptionInformation())) {
    …
}
```

```
LONG ExpFltr (LPEXCEPTION_POINTERS pep) {
    TCHAR szBuf[300], *p;
    PEXCEPTION_RECORD pER = pep->ExceptionRecord;
    DWORD dwExceptionCode = pER->ExceptionCode;

    StringCchPrintf(szBuf, _countof(szBuf), TEXT("Code = %x, Address = %p"),
        dwExceptionCode, pER->ExceptionAddress);

    // находим конец строки
    p = _tcschr(szBuf, TEXT('0'));

    // я использовал оператор switch на тот случай, если Майкрософт
    // в будущем добавит информацию для других исключений
    switch (dwExceptionCode) {
        case EXCEPTION_ACCESS_VIOLATION:
            StringCchPrintf(p, _countof(szBuf),
                TEXT("\n Attempt to %s data at address %p"),
                pER->ExceptionInformation[0] ? TEXT("write") : TEXT("read"),
                pER->ExceptionInformation[1]);
            break;

        default:
            break;
    }

    MessageBox(NULL, szBuf, TEXT("Exception"), MB_OK | MB_ICONEXCLAMATION);

    return(EXCEPTION_CONTINUE_SEARCH);
}
```

Элемент *ContextRecord* структуры EXCEPTION_POTNTERS указывает на структуру CONTEXT (см. главу 7), содержимое которой зависит от типа процессора.

С помощью этой структуры, в основном содержащей по одному элементу для каждого регистра процессора, можно получить дополнительную информацию о возникшем исключении. Увы, это потребует написания машинно-зависимого кода, способного распознавать тип процессора и использовать подходящую для него структуру CONTEXT. При этом вам придется включить в код набор директив #ifdef для разных типов процессоров. Структуры CONTEXT для различных процессоров, поддерживаемых Windows, определены в заголовочном файле WinNT.h.

Программные исключения

До сих пор мы рассматривали обработку аппаратных исключений, когда *процессор* перехватывает *некое* событие и возбуждает исключение. Но вы можете и сами генерировать исключения. Это еще один способ для функции

сообщить о неудаче вызвавшему ее коду Традиционно функции, которые могут закончиться неудачно, возвращают некое особое значение — признак ошибки. При этом предполагается, что код, вызвавший функцию, проверяет, не вернула ли она это особое значение, и, если да, выполняет какие-то альтернативные операции. Как правило, вызывающая функция проводит в таких случаях соответствующую очистку и в свою очередь тоже возвращает код ошибки. Подобная передача кодов ошибок по цепочке вызовов резко усложняет написание и сопровождение кода.

Альтернативный подход заключается в том, что при неудачном вызове функции возбуждают исключения. Тогда написание и сопровождение кода становится гораздо проще, а программы работают намного быстрее. Последнее связано с тем, что та часть кода, которая отвечает за контроль ошибок, вступает в действие лишь при сбоях, т. е. в исключительных ситуациях.

К сожалению, большинство разработчиков не привыкло пользоваться исключениями для обработки ошибок. На то есть две причины. Во-первых, многие просто не знакомы с SEH. Если один разработчик создаст функцию, которая генерирует исключение, а другой не сумеет написать SEH-фрейм для перехвата этого исключения, его приложение при неудачном вызове функции будет завершено операционной системой.

Вторая причина, по которой разработчики избегают пользоваться SEH, — невозможность его переноса на другие операционные системы. Ведь компании нередко выпускают программные продукты, рассчитанные на несколько операционных систем, и, естественно, предпочитают работать с одной базой исходного кода для каждого продукта. А структурная обработка исключений — это технология, специфичная для Windows.

Если вы все же решились на уведомление об ошибках через исключения, я аплодирую этому решению и пишу этот раздел специально для вас. Давайте для начала посмотрим на семейство Heap-функций (*HeapCreate, HeapAlloc* и т. д.) Наверное, вы помните из главы 18, что они предлагают разработчику возможность выбора. Обычно, когда их вызовы заканчиваются неудачно, они возвращают NULL, сообщая об ошибке. Но вы можете передать флаг HEAP_GENERATE_EXCEPTIONS, и тогда при неудачном вызове Heap-функция не станет возвращать NULL; вместо этого она возбудит программное исключение STATUS_NO_MEMORY, перехватываемое с помощью SEH-фрейма.

Чтобы использовать это исключение, напишите код блока *try* так, будто выделение памяти всегда будет успешным; затем — в случае ошибки при выполнении данной операции — вы сможете либо обработать исключение в блоке *except,* либо заставить функцию провести очистку, дополнив блок *try* блоком *finally.* Очень удобно!

Программные исключения перехватываются точно так же, как и аппаратные. Иначе говоря, все, что я рассказывал об аппаратных исключениях, в полной мере относится и к программным исключениям.

В этом разделе основное внимание мы уделим тому, как возбуждать программные исключения в функциях при неудачных вызовах. В сущности, вы

можете реализовать свои функции по аналогии с Heap-функциями: пусть вызывающий их код передает специальный флаг, который сообщает функциям способ уведомления об ошибках.

Возбудить программное исключение несложно — достаточно вызвать функцию *RaiseException:*

```
VOID RaiseException(
    DWORD dwExceptionCode,
    DWORD dwExceptionFlags,
    DW0RD nNumberOfArguments,
    CONST ULONG_PTR *pArguments);
```

Ее первый параметр, *dwExceptionCode*, — значение, которое идентифицирует генерируемое исключение. *HeapAlloc* передает в нем STATUS_NO_MEMORY. Если вы определяете собственные идентификаторы исключений, придерживайтесь формата, применяемого для стандартных кодов ошибок в Windows (файл WinError.h). Не забудьте, что каждый такой код представляет собой значение типа DWORD; его поля описаны в таблице 24-1. Определяя собственные коды исключений, заполните все пять его полей:

- биты 31 и 30 должны содержать код степени «тяжести»;
- бит 29 устанавливается в 1 (0 зарезервирован для исключений, определяемых Майкрософт, вроде STATUS_NO_MEMORY для *HeapAlloc*);
- бит 28 должен быть равен 0;
- биты 27-16 должны указывать один из кодов подсистемы, предопределенных Майкрософт;
- биты 15-0 могут содержать произвольное значение, идентифицирующее ту часть вашего приложения, которая возбуждает исключение.

Второй параметр функции *RaiseException* — *dwExceptionFlags* — должен быть либо 0, либо EXCEPTION_NONCONTINUABLE. В принципе этот флаг указывает, может ли фильтр исключений вернуть EXCEPTION_CONTINUE_EXECUTION в ответ на данное исключение. Если вы передаете в этом параметре нулевое значение, фильтр может вернуть EXCEPTION_CONTINUE_EXECUTION. В нормальной ситуации это заставило бы поток снова выполнить машинную команду, вызвавшую программное исключение. Однако Майкрософт пошла на некоторые ухищрения, и поток возобновляет выполнение с оператора, следующего за вызовом *RaiseException.*

Но, передав функции *RaiseException* флаг EXCEPTION_NONCONTINUABLE, вы сообщаете системе, что возобновить выполнение после данного исключения нельзя. Операционная система использует этот флаг, сигнализируя о критических (фатальных) ошибках. Например, *HeapAlloc* устанавливает этот флаг при возбуждении программного исключения STATUS_NO_MEMORY, чтобы указать системе: выполнение продолжить нельзя. Ведь если вся память занята, выделить в ней новый блок и продолжить выполнение программы не удастся.

Если возбуждается исключение EXCEPTION_NONCONTINUABLE, а фильтр все же возвращает EXCEPTION_CONTINUE_EXECUTION, система генерирует новое исключение EXCEPTION_NONCONTINUABLE_EXCEPTION.

При обработке программой одного исключения вполне вероятно возбуждение нового исключения. И смысл в этом есть. Раз уж мы остановились на этом месте, замечу, что нарушение доступа к памяти возможно и в блоке *finally*, и в фильтре исключений, и в обработчике исключений. Когда происходит нечто подобное, система создает список исключений. Помните функцию *GetExceptionInformation*? Она возвращает адрес структуры EXCEPTION_POINTERS. Ее элемент *ExceptionRecord* указывает на структуру EXCEPTION_RECORD, которая в свою очередь тоже содержит элемент *ExceptionRecord*. Он указывает на другую структуру EXCEPTION_RECORD, где содержится информация о предыдущем исключении.

Обычно система единовременно обрабатывает *только одно исключение,* и элемент *ExceptionRecord* равен NULL. Но если исключение возбуждается при обработке другого исключения, то в первую структуру EXCEPTION_RECORD помещается информация о последнем исключении, а ее элемент *ExceptionRecord* указывает на аналогичную структуру с аналогичными данными о предыдущем исключении. Если есть и другие необработанные исключения, можно продолжить просмотр этого связанного списка структур EXCEPTION_RECORD, чтобы определить, как обработать конкретное исключение.

Третий и четвертый параметры *(nNumberOfArguments* и *pArguments)* функции *RaiseException* позволяют передать дополнительные данные о генерируемом исключении. Обычно это не нужно, и в *pArguments* передается NULL; тогда *RaiseException* игнорирует параметр *nNumberOfArguments*. А если вы передаете дополнительные аргументы, *nNumberOfArguments* должен содержать число элементов в массиве типа ULONG_PTR, на который указывает *pArguments*. Значение *nNumberOfArguments* не может быть больше EXCEPTION_MAXIMUM_PARAMETERS (в файле WinNT.h этот идентификатор определен равным 15).

При обработке исключения написанный вами фильтр — чтобы узнать значения *nNumberOfArguments* и *pArguments* — может ссылаться на элементы *NumberParameters* и *ExceptionInformation* структуры EXCEPTION_RECORD.

Собственные программные исключения генерируют в приложениях по целому ряду причин. Например, чтобы посылать информационные сообщения в системный журнал событий. Как только какая-нибудь функция в вашей программе столкнется с той или иной проблемой, вы можете вызвать *RaiseException;* при этом обработчик исключений следует разместить выше по дереву вызовов, тогда — в зависимости от типа исключения — он будет либо заносить его в журнал событий, либо сообщать о нем пользователю. Вполне допустимо возбуждать программные исключения и для уведомления о внутренних фатальных ошибках в приложении.

Оглавление

ГЛАВА 25

Необработанные исключения, векторная обработка исключений и исключения C++

В предыдущей главе мы обсудили, что происходит, когда фильтр возвращает значение EXCEPTION_CONTINUE_SEARCH. Оно заставляет систему искать дополнительные фильтры исключений, продвигаясь вверх по дереву вызовов. А что будет, если все фильтры вернут EXCEPTION_CONTINUE_SEARCH? Тогда мы получим *необработанное исключение* (unhandled exception).

Windows поддерживает функцию *SetUnhandledExceptionFilter*, которая дает последний шанс на обработку исключения, прежде чем Windows объявит его необработанным:

```
PTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
    PTOP_LEVEL_EXCEPTION_FILTER pTopLevelExceptionFilter);
```

Как правило, эта функция вызывается во время инициализации процесса. В результате при возникновении необработанного исключения в любом из потоков вашего процесса будет вызвана функция фильтра верхнего уровня, заданная параметром *SetUnhandledExceptionFilter*. Прототип функции фильтра выглядит следующим образом:

```
LONG WINAPI TopLevelUnhandledExceptionFilter(PEXCEPTION_POINTERS pException Info);
```

Функция фильтра может выполнять любые нужные вам действия, но возвращать она должна один из трех идентификаторов EXCEPTION_*, перечисленных в табл. 25-1. Учите, что состояние процесса при вызове этой функции может быть повреждено из-за переполнения стека, а также синхронизирующих объектов или блоков памяти в куче, оставшихся занятыми.

Поэтому в функции фильтра следует выполнять минимальное число действий, также нельзя использовать динамическое выделение памяти, поскольку куча может быть повреждена. При установке нового фильтра для необработанных исключений *SetUnhandledExceptionFilter* возвращает адрес ранее установленного фильтра. Заметьте, что в случае приложений, использующих библиотеку C/C++, перед исполнением входной функции программы библиотека устанавливает собственный глобальный фильтр для необработанных исключений — *__CxxUnhandledExceptionFilter*. Эта функция просто проверяет, является ли возникшее исключение исключением C++ (подробнее об этом см. ниже в этой главе), и, если это так, исполняет функцию *abort*, которая, в свою очередь, вызывает *UnhandledExceptionFilter* из Kernel32.dll. Прежние версии библиотеки C/C++ просто прерывают исполнение процесса. Функция *_set_abort_behavior* служит для настройки отчетов об ошибках, которые выводит *abort*. Если возникшее исключение не опознано как C++-исключение, фильтр возвращает идентификатор EXCEPTION_CONTINUE_SEARCH, который говорит операционной системе, что ей придется самой позаботиться об этом исключении.

Табл. 25-1. Значения, возвращаемые фильтром исключений верхнего уровня

Идентификатор	Действия системы
EXCEPTION_EXECUTE_HANDLER	Исполнение процесса прерывается без уведомления пользователя. При этом происходит глобальная раскрутка, а значит, код в блоке finally будет исполнен
EXCEPTION_CONTINUE_EXECUTION	Исполнение возобновляется с команды, генерировавшей исключение. Можно модифицировать сведения об исключении, на которые указывает параметр PEXCEPTION_POINTERS. Если ошибка не устранена и исключение возникает повторно, процесс может попасть в бесконечный цикл
EXCEPTION_CONTINUE_SEARCH	Исключение действительно становится необработанным, подробнее об этой ситуации насказывается в следующем разделе

Таким образом, вызвав *SetUnhandledExceptionFilter*, чтобы установить собственный фильтр, вы получите адрес функции *__CxxUnhandledExceptionFilter*, что и показывает IntelliSense при отладке кода в Visual Studio. В противном случае фильтром исключений по умолчанию становится функция *UnhandledExceptionFilter*.

> **Примечание.** Чтобы сбросить фильтр исключений, достаточно вызвать *SetUnhandledExceptionFilter* и передать ей NULL — глобальным фильтром необработанных исключений станет *UnhandledExceptionFilter*.

Если ваш фильтр возвращает EXCEPTION_CONTINUE_SEARCH, так и хочется вызвать ранее установленный фильтр, адрес которого возвращает

функция *SetUnhandledExceptionFilter*. Однако так поступать не стоит, поскольку какие-нибудь сторонние программы вполне могли установить собственные фильтры исключений. Если эти фильтры располагаются в динамически загружаемых модулях, к моменту возникновения исключения они могут быть уже выгружены. Есть и еще одна причина, по которой не следует вызывать ранее установленный фильтр, о ней рассказывается ниже на стр. 708.

Как вы помните из главы 6, выполнение потока начинается с функции *BaseProcessStart* или *BaseThreadStart* в Kernel32.dll. Единственная разница между этими функциями в том, что первая используется для запуска первичного потока процесса, а вторая — для запуска остальных потоков процесса.

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
    __try {
        ExitThread((pfnStartAddr)(pvParam));
    }
    __except (UnhandledExceptionFilter(GetExceptionlnformation())) {
        ExitProcess(GetExceptionCode())
    }
    // Примечание: сюда мы никогда не попадем
}
```

Обратите внимание, что обе функции содержат SEH-фрейм: поток запускается из блока *try*. Если поток возбудит исключение, в ответ на которое все ваши фильтры вернут EXCEPTION_CONTINUE_SEARCH, будет вызвана особая функция фильтра, предоставляемая операционной системой:

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionlnfo);
```

Подобно любому фильтру исключений, эта функция возвращает один из трех EXCEPTION_*-идентификаторов, но действия системы при их получении несколько отличаются (см. табл. 25-2).

Табл. 25-2. Значения, возвращаемые функцией UnhandledExceptionFilter

Идентификатор	Действия системы
EXCEPTION_EXECUTE_HANDLER	Идет глобальная раскрутка и выполняется код каждого из необработанных блоков finally. В случае необработанного исключения обработчик BaseThreadStart вызывает ExitProcess и «молча» завершается. Заметьте, что код исключения становится кодом завершения процесса
EXCEPTION_CONTINUE_EXECUTION	Исполнение продолжается с команды, вызвавшей исключение. Подробнее см. в разделе о функции UnhandledExceptionFilter ниже

Табл. 25-2. *(окончание)*

Идентификатор	Действия системы
EXCEPTION_CONTINUE_ SEARCH	К сбойному процессу будет подключен специальный отладчик либо отладчик по умолчанию. В обоих случаях система уведомляет отладчик об исключении, в результате отладчик продолжает исполнять код с команды, в которой возникло исключение. Подробнее об этом рассказывается в разделе, посвященном отладке по запросу. Если отладчик не подключается, Windows считает, что возникло необработанное исключение пользовательского режима.

> **Примечание.** В случае вложенных исключений, т.е. когда функция фильтра сама провоцирует исключение, *UnhandledExceptionFilter* возвращает идентификатор EXCEPTION_NESTED_CALL (так происходит в Windows Vista, в прежних версиях Windows *UnhandledExceptionFilter* вовсе не возвращала управление в этом случае и процесс попросту завершался).

Однако до возврата этих значений исполняется изрядное количество кода, и теперь настало время разобраться в том, как работает *UnhandledExceptionFilter*.

Как работает функция *UnhandledExceptionFilter*

Функция *UnhandledExceptionFilter* обрабатывает исключения в пять этапов. Ниже мы разберем их все по порядку. Выполнив эти действия, *UnhandledExceptionFilter* передает управление службе *Windows Error Reporting* (WER), как описано в следующем разделе.

Этап 1. Разрешение доступа к ресурсу для записи и продолжение исполнения

Если нарушение доступа возникло при попытке потока выполнить запись, *UnhandledExceptionFilter* проверяет, не пытается ли поток модифицировать .exe- или .dll-модуль. По умолчанию ресурсы являются (и совершенно обоснованно) неизменяемыми, поэтому попытка их модификации приводит к нарушению доступа. Однако 16-разрядные версии Windows разрешают модификацию ресурсов, и для преемственной совместимости в 32- и 64-разрядных версиях также необходим доступ для записи к ресурсам. Чтобы разрешить его, *UnhandledExceptionFilter* вызовом *VirtualProtect* изменяет атрибут защиты страниц, хранящих ресурс, на PAGEREADWRITE, и возвращает EXCEPTION_CONTINUE_EXECUTION, чтобы система повторила исполнение команды, вызвавшей сбой.

Этап 2. Уведомление отладчика о необработанном исключении

Далее *UnhandledExceptionFilter* проверяет, не работает ли приложение под контролем отладчика. Если это так, *UnhandledExceptionFilter* возвращает EXCEP-

TION_CONTINUE_SEARCH. Поскольку исключение до сих пор не обработано, Windows уведомляет подключенный к процессу отладчик. Отладчик получает значение поля *ExceptionInformation* созданной для исключения структуры EXCEPTION_RECORD. Затем отладчик использует эту информацию для поиска команды, вызвавшей исключение, и уведомления пользователя о характере исключения. Чтобы выяснить, не работает ли ваша программа под контролем отладчика, вызовите функцию *IsDebuggerPresent*.

Этап 3. Уведомление глобального фильтра исключений

Если вызовом *SetUnhandledExceptionFilter* была установлена функция глобального фильтра исключений, *UnhandledExceptionFilter* вызывает ее. Если фильтр возвращает EXCEPTION_EXECUTE_HANDLER или EXCEPTION_CONTINUE_ EXECUTION, функция *UnhandledExceptionFilter* передает это значение системе. Если же фильтр необработанных исключений возвращает EXCEPTION_CONTINUE_SEARCH, выполняется четвертый этап (см. ниже). Но постойте-ка, я только что говорил, что глобальный фильтр исключений C/C++, *__CxxUnhandkdExceptionFilter*, явно вызывает *UnhandledExceptionFilter*! Подобные серии рекурсивных вызовов быстро приведут к переполнению стека и возникновению нового исключения, которое скроет исходное исключение. Это еще одна причина, по которой не следует вызывать ранее установленный глобальный фильтр исключений. Чтобы предотвратить рекурсию, *__CxxUnhandledExceptionFilter* вызывает *SetUnhandledExceptionFilter* (NULL) непосредственно перед *UnhandledExceptionFilter*.

Если программа использует библиотеку C/C++, то библиотека заключает вызов входной функции в блок *try/except*, и устанавливает фильтр исключений, вызывающий библиотечную C/C++- функцию *_XcpFilter*. Сама *_XcpFilter* вызывает *UnhandledExceptionFilter*, а та — функцию глобального фильтра исключений, если таковая задана. Таким образом, если установлен глобальный фильтр, он будет вызван в случае возникновения необработанного исключения, замеченного *_XcpFilter*. Если ваш фильтр вернет EXCEPTION_CONTINUE_SEARCH, то необработанное исключение (на данном этапе оно действительно является необработанным) передается фильтру except функции *BaseThreadStart*, а тот вызывает *UnhandledExceptionFilter*. В результате заданный вами глобальный фильтр вызывается повторно.

Этап 4. Повторное уведомление отладчика о необработанном исключении

На предыдущем этапе глобальный фильтр необработанных исключений может вызвать отладчик, чтобы он подключился к процессу, где работает поток, в котором возникло необработанное исключение. Если и теперь фильтр необработанных исключений вернет EXCEPTION_CONTINUE_SEARCH, отладчик вновь получит уведомление (см. этап 2).

Этап 5. «Тихое» завершение процесса

Если один из потоков процесса вызвал функцию *SetErrorMode* и передал ей флаг SEM_NOGPFAULTERRORBOX, *UnhandledExeptionFilter* вернет EXCEP-

TION_EXECUTE_HANDLER. В случае необработанного исключения возврат этого значения инициирует глобальную раскрутку, в ходе которой выполняется код из блока *finally*, сразу после этого процесс «тихо» завершается. Аналогичным образом, если процесс включен в задание (см. главу 5), для которого установлен флаг JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION, функция *UnhandledExceptionFilter* вернет EXCEPTION_EXECUTE_HANDLER с теми же последствиями.

Все эти операции *UnhandledExceptionFilter* выполняет «тихо», незаметно для пользователя пытаясь устранить вызвавший исключение сбой, уведомить подключенный к процессу отладчик (если таковой имеется), либо, при необходимости, просто завершить приложение. Если же обработать исключение не удается, возвращается значение EXCEPTION_CONTINUE_SEARCH и управление переходит к ядру, которое уведомляет пользователя о том, что в приложении возник какой-то сбой. Но перед разбором действий ядра Windows при завершении *UnhandledExceptionFilter* и маршрутизации исключений в системе (см. следующий раздел) давайте посмотрим, какие окна выводит система в различных сценариях обработки исключений.

На рис. 25-1 показано, что происходит в Windows XP при передаче исключения функций *UnhandledExceptionFilter*.

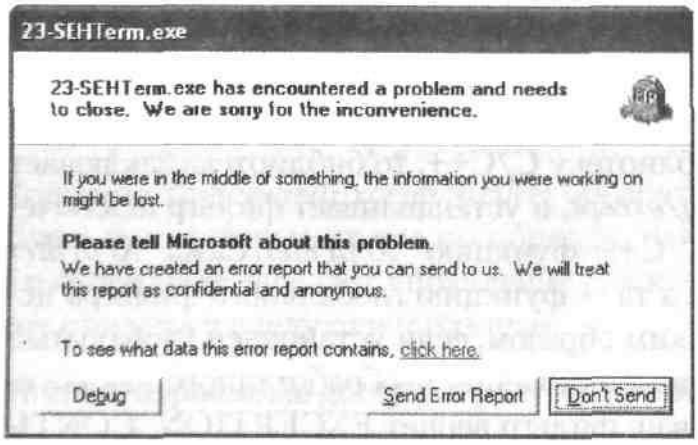


Рис. 25-1. *Сообщение о необработанном исключении в Windows XP*

В подобной ситуации Windows Vista поочередно выводит два окна, показанных на рис. 25-2 и 25-3.

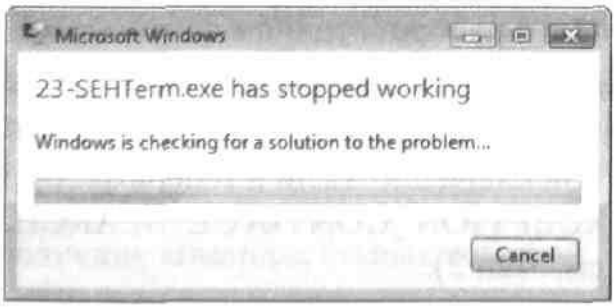


Рис. 25-2. *Первое сообщение о необработанном исключении в Windows Vista*

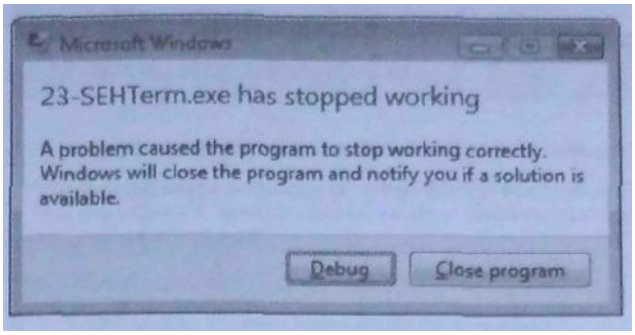


Рис. 25-3. *Второе сообщение о необработанном исключении в Windows Vista*

Взаимодействие *UnhandledExceptionFilter* с WER

На рис. 25-4 показана схема обработки необработанных исключений в Windows с использованием инфраструктуры Windows Error Reporting; первые два этапа этого процесса мы обсудили в предыдущем разделе. В Windows Vista функция *UnhandledExceptionFilter*, в отличие от предыдущих версий Windows, не посылает отчет об ошибке на сервер Майкрософт, а выполняет действия, описанные в предыдущем разделе, просто возвращая EXCEPTION_CONTINUE_SEARCH (см. описание 3 этапа работы *UnhandledExceptionFilter* выше). В этом случае ядро определяет, что поток пользовательского режима не смог обработать исключение (этап 4). В итоге уведомление об исключении направляется специальной службе *WerSvc*.

Передача этого уведомления осуществляется с использованием недокументированного механизма ALPC (Advanced Local Procedure Call), блокирующего исполнение потока до завершения обработки уведомления службой WerSvc (этап 5). Вызовом *CreateProcess* эта служба порождает процесс WerFault.exe (этап 6) и ожидает завершения нового процесса. Создание и отправку отчета (этап 7) выполняет приложение WerFault.exe. Диалоговые окна, в которых пользователь может выбрать между завершением приложения и подключением отладчика также выводятся в контексте приложения WerFault.exe (этап 8). Если пользователь решает закрыть программы, WerFault.exe вызывает *TerminateProcess*, в результате исполнение программы прерывается без дополнительных уведомлений (этап 9). Как видите, все громоздкие операции выполняются «за пределами» сбойного приложения, что обеспечивает стабильность системы и возможность создания отчетов.

Соответствующие окна настраиваются с помощью соответствующих разделов реестра, описанных в статье, доступной по этой ссылке: http://msdn2.microsoft.com/en-us/library/bb513638.aspx. Если параметр DontShowUI в разделе HKEY_CURRENT_USER\Software\Microsoft\Windows\Windows Error Reporting установлен в 1, никакие окна не открываются, система «молча» генерирует отчет и отправляет его на один из серверов Майкрософт. Чтобы позволить пользователю отменить отправку отчета, измените значение DWORD-параметра *DefaultConsent* в разделе *Consent*. Однако лучше открыть Control Panel и выбрать аплет WER Console в категории Problem Reports And Solutions. После этого щелкните ссылку Change Settings — откроется окно, показанное на рис. 25-5.

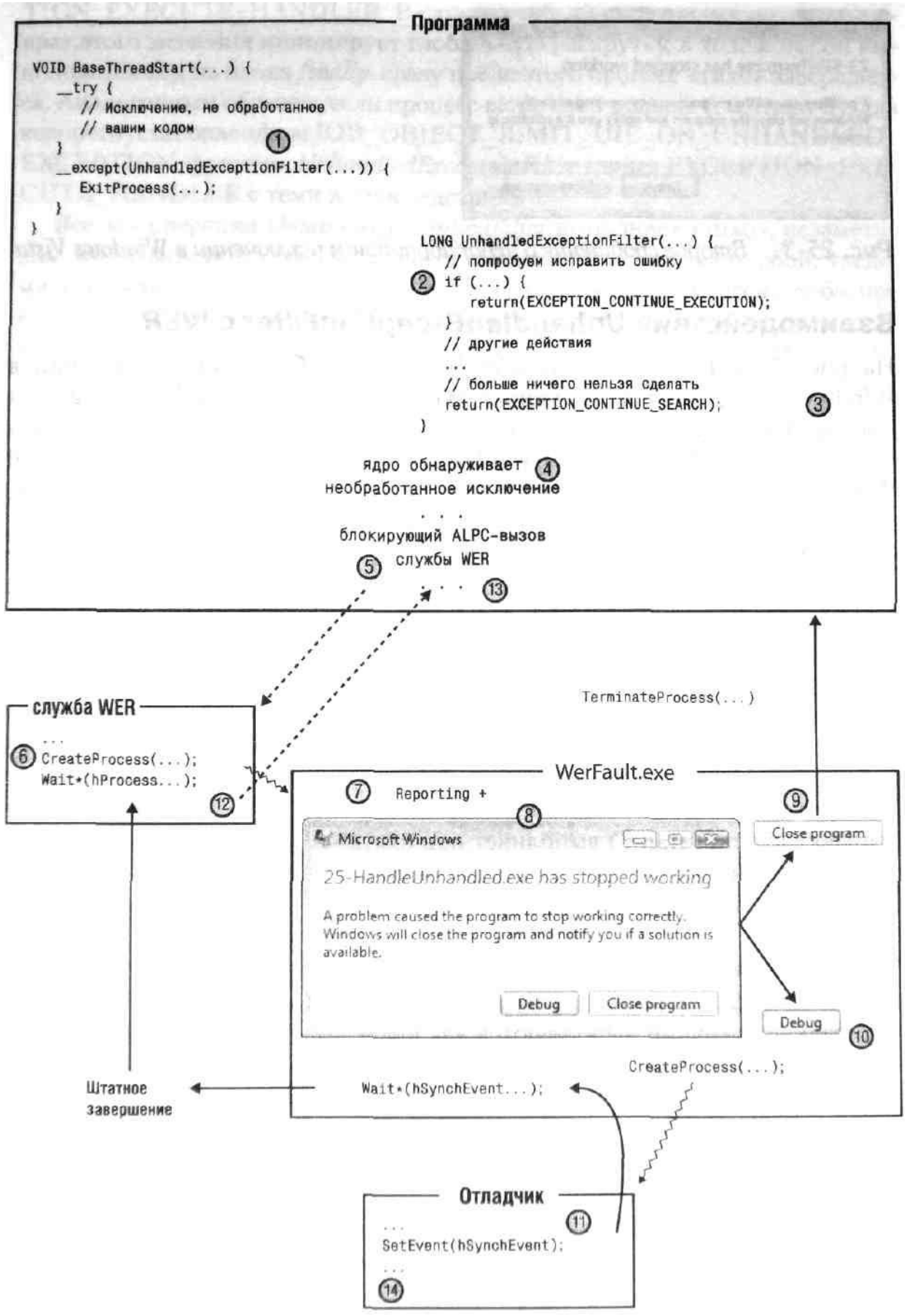


Рис. 25-4. *Обработка необработанных исключений в Windows с использованием инфраструктуры Windows Error Reporting*

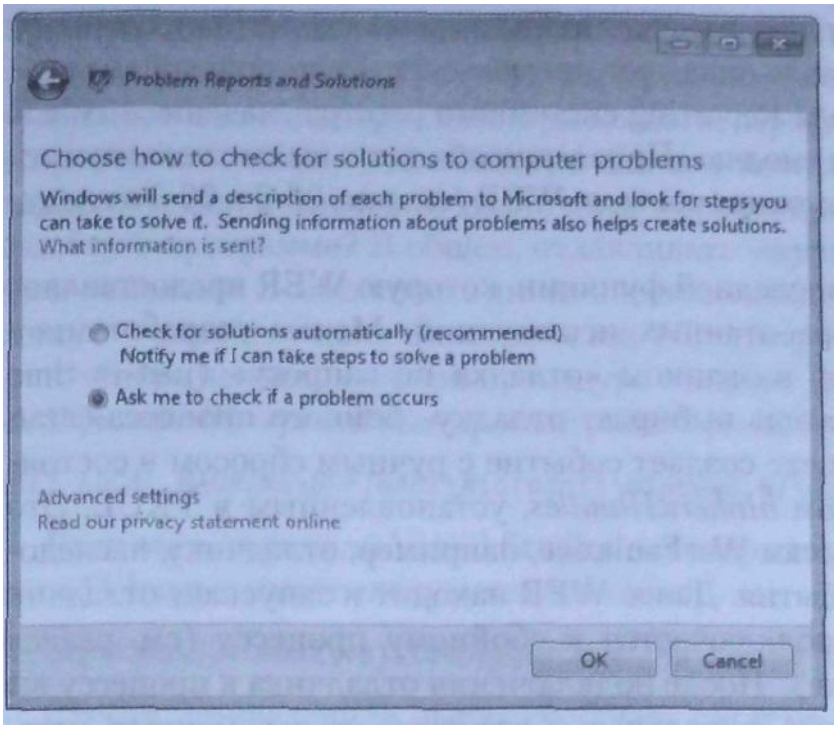


Рис. 25-5. *Включение окна, в котором пользователь может указать, следует ли отправлять в Майкрософт отчет об ошибке*

Если включить параметр Ask Me To Check If A Problem Occurs, WER откроет одно окно, показанное на рис. 25-6, вместо обычных двух (см. рис. 25-2 и 25-3).

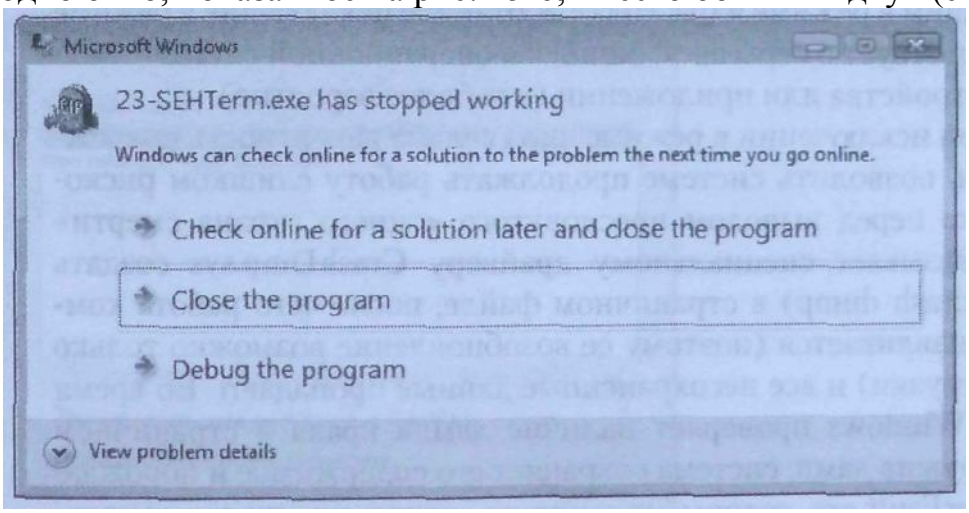


Рис. 25-6. *Пользователь может отменить автоматическую отправку отчета об ошибке в Майкрософт*

На «рабочих» компьютерах включать этот параметр не рекомендуется, так вам необходимо знать о возникающих ошибках. Однако при отладке отключение этого окна поможет вам сэкономить массу времени, поскольку вам не придется ждать завершения генерации и отправки отчета. Экономия будет особенно ощутимой, если компьютер не подключен к сети, поскольку в этом случае окно WER открывается только после тайм-аута WerSvc.

Появление окон WER также нежелательно во время автоматизированного тестирования, так при выводе окна тест прерывается. Если присвоить параметру *ForceQueue* из раздела Reporting системного реестра значение 1, WER будет генерировать отчеты «молча». По завершении теста вы сможете изучить ошибки и их описание с помощью консоли WER (см. рис. 26-2 и 26-3 в следующей главе).

А теперь поговорим о последней функции, которую WER предоставляет при возникновении необработанных исключений. Мечты разработчиков воплотились в механизме, названном «отладка по запросу» (just-in-time debugging). Если пользователь выбирает отладку сбойного процесса (этап 10), приложение WerFault.exe создает событие с ручным сбросом в состоянии «занято» и параметром *bInheritHandles*, установленным в TRUE. Это позволит дочерним процессам WerFault.exe, например, отладчику, наследователь описатель этого события. Далее WER находит и запускает отладчик по умолчанию, который подключается к сбойному процессу (см. раздел «Отладка по запросу» ниже). После подключения отладчика к процессу вы сможете изучать глобальные, локальные и статические переменные, устанавливать точки прерывания, анализировать дерево вызовов, перезапускать процесс и выполнять любые другие действия, обычные для отладки.

> **Примечание.** Предметом этой книги является разработка программ для пользовательского режима. Однако некоторых читателей может заинтересовать, что происходит, если исключение провоцируется потоком, работающим в режиме ядра. Необработанное исключение в режиме ядра свидетельствует о серьезной ошибке в операционной системе либо в драйвере устройства или приложении (что более вероятно). Поскольку при исключении в режиме ядра велика вероятность повреждения памяти, позволить системе продолжать работу слишком рискованно. Однако перед выводом пресловутого «синего экрана смерти» система приказывает специальному драйверу CrashDmp.sys создать дамп краха (crash dump) в страничном файле, после чего работа компьютера останавливается (поэтому ее возобновление возможно только после перезагрузки) и все несохраненные данные пропадают. Во время перезапуска Windows проверяет наличие дампа краха в страничном файле. Обнаружив дамп, система сохраняет его содержимое и порождает процесс WerFault.exe, который генерирует отчет и, при желании пользователя, отправляет его на сервер Майкрософт. Эти действия также позволяют вывести список сбоев Windows в окне консоли WER.

Отладка по запросу

Windows позволяет подключать отладчик к любому процессу в любой момент времени — эта функциональность называется отладкой по запросу (just-in-time debugging). Основная польза отладки по запросу в том, что она позволяет обрабатывать сбои в приложении по мере их возникновения.

В других операционных системах отладка обычно требует перезапуска приложения с отладчиком. То есть, вам придется закрыть процесс, запустить отладчик и вызвать через него приложение. Проблема здесь в том, что для устранения ошибки ее необходимо сначала воспроизвести. Но как узнать, какие сочетания значений различных переменных вызывают именно эту ошибку в программе? В общем, отлавливать «жучком» таким способом намного сложнее. Возможность динамического подключения отладчика к процессу — одна из лучших возможностей Windows.

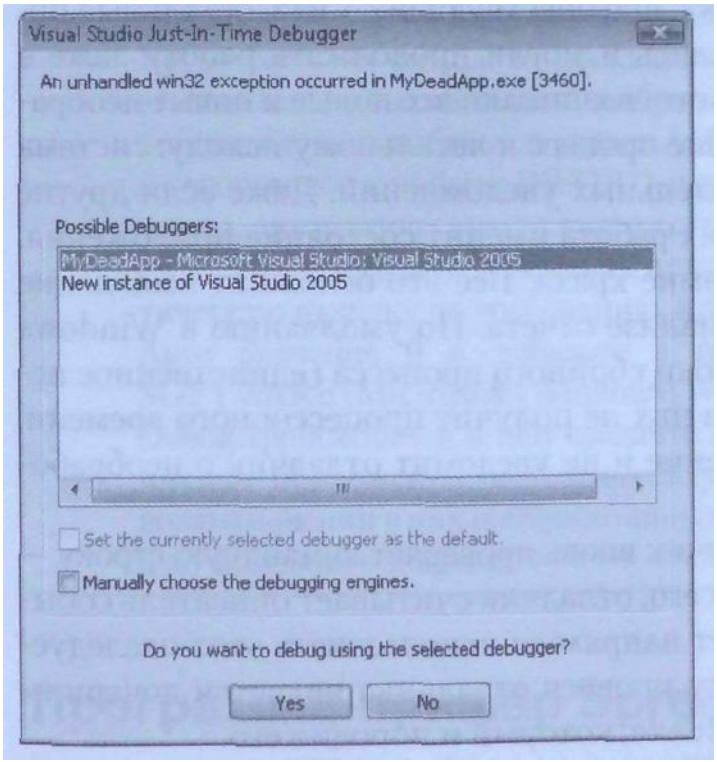
А теперь немного о самой процедуре отладки. Для активизации отладчика *UnhandledExceptionFilter* просматривает раздел реестра:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Micro30ft\Window8 NT\CurrentVersion\AeDebug
```

Если вы установили Visual Studio, то содержащийся в этом разделе параметр Debugger имеет следующее значение:

```
"C:\Windows\system32\vsjitdebugger.exe" -p %ld -e %ld
```

Строка, приведенная выше, сообщает системе, какой отладчик надо запустить (в данном случае — vsjitdebugger.exe). На самом деле, vsjitdebugger.exe не отладчик, а программа, которая позволяет выбрать настоящий отладчик в этом окне:



Естественно, вы можете изменить это значение, указав другой отладчик WerFault.exe передает отладчику два параметра в командной строке. Первый — это идентификатор процесса, который нужно отладить, а второй -- наследуемое событие со сбросом вручную, которое создается службой WerSvc в занятом состоянии. Отладчик должен распознавать ключи *-p* и *-e* как идентификатор процесса и описатель события.

После конкатенации идентификатора процесса и описателя события WerFault.exe запускает отладчик вызовом *CreateProcess* с параметром *bInheritHandles*, установленным в TRUE, при этом процесс отладчика наследует описатель объекта-события. После этого процесс отладчика начинает работу и проверяет переданные ему через командную строку аргументы. Если задан ключ *-p*, отладчик подключается к процессу, заданному этим ключом, вызывая *DebugActiveProcess*:

```
BOOL DebugActiveProcess(DWORD dwProcessID);
```

После этого система начинает уведомлять отладчик о состоянии отлаживаемого процесса, сообщая, например, сколько в нем потоков и какие DLL спроецированы на его адресное пространство. На сбор этих данных отладчику нужно какое-то время, в течение которого сбойный процесс должен находиться в режиме ожидания. Поток, обнаруживший необработанное исключение (этап 4), все еще ждет ALPC-уведомления от WerSvc (шаг 5). ALPC при этом блокирован на вызове *WaitForSingleObjectEx* с описателем процесса WerFault.exe, который ждет завершения своей работы. Заметьте, что здесь вместо *WaitForSingleObject* используется функция *WaitForSingleObjectEx*, которая переводит поток в состояние тревожного ожидания. Это позволяет обработать все APC-вызовы, поставленные в очередь потока.

Заметьте, что в версиях Windows, предшествующих Vista, остальные потоки процесса не приостанавливались и могли продолжать работу даже в поврежденном контексте. В результате возникали все новые и новые необработанные исключения, приводившие процесс к летальному исходу: система просто «убивала» его без дополнительных уведомлений. Даже если другие потоки этого процесса не рухнут, их работа именит состояние приложения, которое будет зафиксировано в дампе краха. Все это осложнит выявление первопричины исключения при анализе отчета. По умолчанию в Windows Vista приостанавливаются все потоки сбойного процесса (единственное исключение — службы) и ни один из них не получит процессорного времени, пока WER не возобновит исполнение и не уведомит отладчик о необработанном исключении.

Закончив инициализацию, отладчик вновь проверяет командную строку — на этот раз он ищет ключ *-e*. Найдя его, отладчик считывает описатель события и вызывает *SetEvent*. Он может напрямую использовать этот наследуемый описатель события, поскольку процесс отладчика является дочерним по отношению к процессу, WerFault.exe, который и породил его.

По переходу события в свободное состояние (этап 11) WerFault.exe определяет, что к сбойному процессу подключился отладчик, готовый к приему уведомления об исключении. После этого WerFault.exe завершается, WerSvc обнаруживает это и позволяет продолжить работу ALPC (этап 12). Далее пробуждается поток отлаживаемого процесса, и отладчик получает информацию о необработанном исключении (этап 13). Эти операции дают тот же результат, что действия, выполняемые на третьем этапе работы функции *Un-*

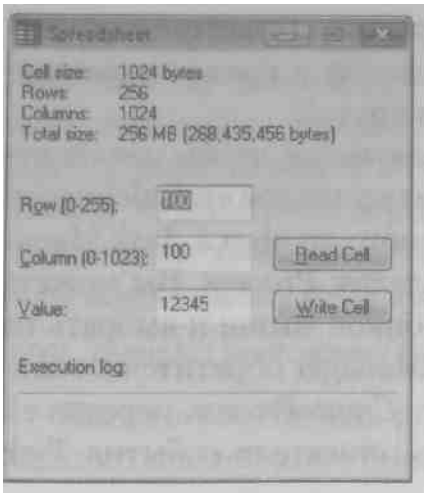
handledExceptionFilter. Получив сведения об исключении, отладчик загружает соответствующий файл исходного кода и переходит к команде, вызвавшей исключение (этап 14). Вот это действительно круто!

Кстати, совсем не обязательно дожидаться исключения, чтобы начать отладку. Отладчик можно подключить в любой момент командой «vsjitdebugger.exe *-p PID*», где PID — идентификатор отлаживаемого процесса. Task Manager еще больше упрощает эту задачу. Открыв вкладку Process, Вы можете щелкнуть строку с нужным процессом правой кнопкой мыши и выбрать из контекстного меню команду Debug. В ответ Task Manager обратится к только что рассмотренному разделу реестра и вызовет *CreateProcess*, передав ей идентификатор выбранного процесса. Но вместо описателя события Task Manager передаст 0.

> **Совет.** В том же разделе HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug находится REG_SZ-параметр Auto. Его значение указывает WER, следует ли дать пользователю выбор между завершением и отладкой сбойного процесса. Если Auto = 1, WER не дает пользователю никакого выбора и сразу же вызывает отладчик. Этот параметр рекомендуется включать на компьютере, на котором ведется разработка, чтобы не ждать вывода диалоговых окон WER, поскольку отлаживать приложение, в котором возникают необработанные исключения, вам все равно придется.
>
> Кроме того, нет смысла отлаживать приложения-контейнеры, такие как svchost.exe, при крахе какой-нибудь из работающих в нем служб. В подобных случаях добавьте в раздел AeDebug подраздел AutoExclusionList, в котором должен быть DWORD-параметр названный по имени приложения, автоматическую отладку которого следует запретить, и значением 1. Чтобы выбрать, для каких приложений следует включить автоматическую отладку по требованию, а для каких — нет, оставьте параметру Auto значение 0 и добавьте в раздел HKEY_CURRENT_USER\SOFTWARE\Microsoft\Windows\Windows Error Reporting подраздел DebugApplications и в нем создайте DWORD-параметры, названные по именам приложений, которые следует автоматически отлаживать при возникновении в них необработанных исключений; этим параметрам следует присвоить значение 1.

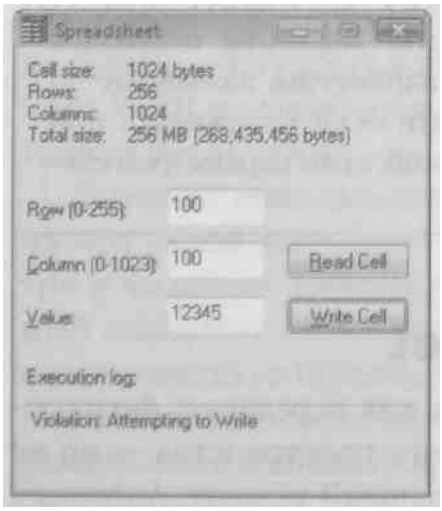
Программа-пример Spreadsheet

Эта программа (25-Spreadsheet.exe), демонстрирует, как передавать физическую память зарезервированному региону адресного пространства — но не всему региону, а только его областям, нужным в данный момент. Алгоритм опирается на структурную обработку исключений. Файлы исходного кода и ресурсов этой программы находятся в каталоге 25-Spreadsheet на компакт-диске, прилагаемом к книге. После запуска Spreadsheet на экране появляется диалоговое окно, показанное ниже.



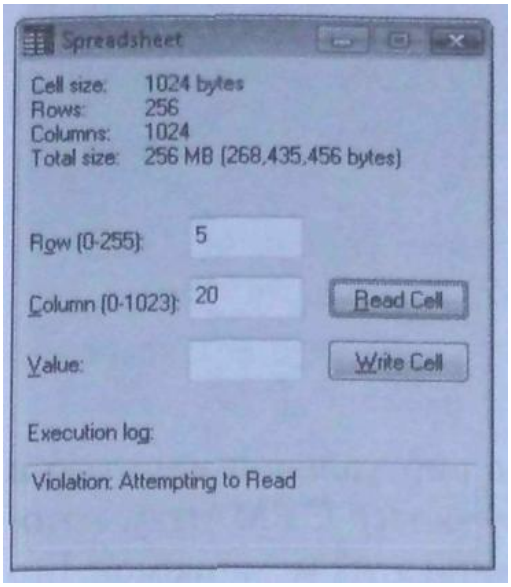
Программа Spreadsheet резервирует регион для двухмерной таблицы, содержащей 256 строк и 1024 колонки, с размером ячеек по 1024 байта. Если бы программа заранее передавала физическую память под всю таблицу, то ей понадобилось бы 268 435 456 байтов, или 256 Мб. Поэтому для экономии драгоценных ресурсов программа резервирует в своем адресном пространстве регион размером 256 Мб, не передавая ему физическую память.

Допустим, пользователь хочет поместить значение 12345 в ячейку на пересечении строки 100 и колонки 100 (как на предыдущей иллюстрации). Как только он щелкнет кнопку Write Cell, программа попытается записать это значение в указанную ячейку таблицы. Естественно, это вызовет нарушение доступа. Но, так как я использую в программе SEH, мой фильтр исключений, распознав попытку записи, выведет в нижней части диалогового окна сообщение «Violation: Attempting to Write», передаст память под нужную ячейку и заставит процессор повторить выполнение команды, возбудившей исключение. Теперь значение будет сохранено в ячейке таблицы, поскольку этой ячейке передана физическая память.

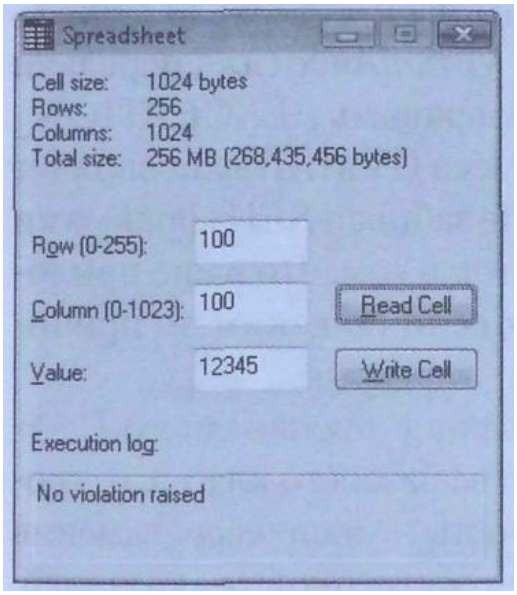


Проделаем еще один эксперимент. Попробуем считать значение из ячейки на пересечении строки 5 и колонки 20. Этим мы вновь вызовем нарушение доступа. На этот раз фильтр исключений не передаст память, а выведет

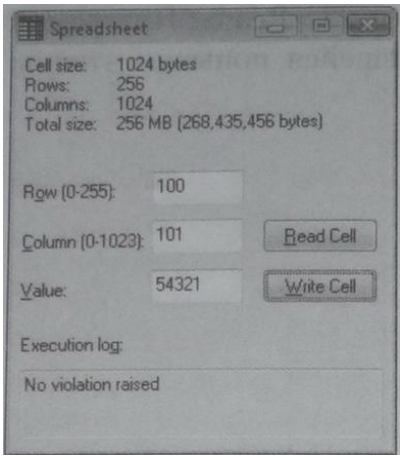
в диалоговом окне сообщение «Violation: Attempting to Read». Программа корректно возобновит свою работу после неудавшейся попытки чтения, очистив поле Value диалогового окна.



Третий эксперимент: попробуем считать значение из ячейки на пересечении строки 100 и колонки 100. Так как этой ячейке передана физическая память, никаких исключений не возбуждается, и фильтр не выполняется (что положительно сказывается на быстродействии программы). Диалоговое окно будет выглядеть следующим образом.



Ну и последний эксперимент: запишем значение 54321 в ячейку на пересечении строки 100 и колонки 101. Эта операция пройдет успешно, без исключений, потому что данная ячейка находится на той же странице памяти, что и ячейка (100, 100). В подтверждение этого вы увидите сообщение «No Violation raised» в нижней части диалогового окна.



В своих проектах я довольно часто пользуюсь виртуальной памятью и SEH. Как-то раз я решил создать шаблонный C++-класс CVMArray, который инкапсулирует все, что нужно для использования этих механизмов. Его исходный код содержится в файле VMArray.h (он является частью программы-примера Spreadsheet). Вы можете работать с классом CVMArray двумя способами. Во-первых, просто создать экземпляр этого класса, передав конструктору максимальное число элементов массива. Класс автоматически устанавливает действующий на уровне всего процесса фильтр необработанных исключений, чтобы любое обращение из любого потока к адресу в виртуальном массиве памяти заставляло фильтр вызывать VirtualAlloc (для передачи физической памяти новому элементу) и возвращать EXCEPTION_ CONTINUEEXECUTION. Такое применение класса CVMArray позволяет работать с разреженной памятью (sparse storage), не забивая SEH-фреймами исходный код программы. Единственный недостаток в том, что ваше приложение не сможет возобновить корректную работу, если по каким-то причинам передать память не удастся.

Второй способ использования CVMArray — создание производного C++-класса. Производный класс даст вам все преимущества базового класса, и, кроме того, вы сможете расширить его функциональность — например, заменив исходную виртуальную функцию *OnAccessViolation* собственной реализацией, более аккуратно обрабатывающей нехватку памяти. Программа Spreadsheet как раз и демонстрирует этот способ использования класса CVMArray.

Spreadsheet.cpp

```
/******************************************************************************
Module:  Spreadsheet.cpp
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/


#include "..\CommonFiles\CmnHdr.h"     /* см. приложение A */
#include <windowsx.h>
```

```
#include <tchar.h>
#include "Resource.h"
#include "VMArray.h"
#include <StrSafe.h>

//////////////////////////////////////////////////////////////////////

HWND g_hWnd;  // глобальный описатель окна, применяемого для SEH-отчетов
const int g_nNumRows = 256;
const int g_nNumCols = 1024;

// определяем структуру ячеек таблицы
typedef struct {
   DWORD dwValue;
   BYTE  bDummy[1020];
} CELL, *PCELL;

// объявляем тип данных для всей таблицы
typedef CELL SPREADSHEET[g_nNumRows][g_nNumCols];
typedef SPREADSHEET *PSPREADSHEET;

//////////////////////////////////////////////////////////////////////////////

// таблица является двухмерным массивом переменных типа CELL
class CVMSpreadsheet : public CVMArray<CELL> {
public:
   CVMSpreadsheet() : CVMArray<CELL>(g_nNumRows * g_nNumCols) {}

private:
   LONG OnAccessViolation(PVOID pvAddrTouched, BOOL bAttemptedRead,
      PEXCEPTION_POINTERS pep, BOOL bRetryUntilSuccessful);
};

//////////////////////////////////////////////////////////////////////////////

LONG CVMSpreadsheet::OnAccessViolation(PVOID pvAddrTouched, BOOL bAttempte-
    dRead,
   PEXCEPTION_POINTERS pep, BOOL bRetryUntilSuccessful) {

   TCHAR sz[200];
   StringCchPrintf(sz, _countof(sz), TEXT("Violation: Attempting to %s"),
      bAttemptedRead ? TEXT("Read") : TEXT("Write"));
   SetDlgItemText(g_hWnd, IDC_LOG, sz);
```

```
   LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;
   if (!bAttemptedRead) {
     // возвращаем значение, определяемое базовым классом
      lDisposition = CVMArray<CELL>::OnAccessViolation(pvAddrTouched,
         bAttemptedRead, pep, bRetryUntilSuccessful);
   }

   return(lDisposition);
}

//////////////////////////////////////////////////////////////////////////////

// это глобальный объект CVMSpreadsheet
static CVMSpreadsheet g_ssObject;

// создаем глобальный указатель на весь регион, зарезервированный под
// таблицу

SPREADSHEET& g_ss = * (PSPREADSHEET) (PCELL) g_ssObject;

//////////////////////////////////////////////////////////////////////////////

BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_SPREADSHEET);

   g_hWnd = hWnd; // сохраняем для SEH-отчетов

   // инициализируем элементы управления в диалоговом окне значениями
   // по умолчанию

   Edit_LimitText(GetDlgItem(hWnd, IDC_ROW),    3);
   Edit_LimitText(GetDlgItem(hWnd, IDC_COLUMN), 4);
   Edit_LimitText(GetDlgItem(hWnd, IDC_VALUE),  7);
   SetDlgItemInt(hWnd, IDC_ROW,    100,   FALSE);
   SetDlgItemInt(hWnd, IDC_COLUMN, 100,   FALSE);
   SetDlgItemInt(hWnd, IDC_VALUE,  12345, FALSE);
   return(TRUE);
}

//////////////////////////////////////////////////////////////////////////////

void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   int nRow, nCol;
```

```
   switch (id) {
      case IDCANCEL:
         EndDialog(hWnd, id);
         break;

      case IDC_ROW:
         // пользователь изменил строку, обновляем данные в окне
         nRow = GetDlgItemInt(hWnd, IDC_ROW, NULL, FALSE);
         EnableWindow(GetDlgItem(hWnd, IDC_READCELL),
            chINRANGE(0, nRow, g_nNumRows - 1));
         EnableWindow(GetDlgItem(hWnd, IDC_WRITECELL),
            chINRANGE(0, nRow, g_nNumRows - 1));
         break;

      case IDC_COLUMN:
         // пользователь изменил колонку, обновляем данные в окне
         nCol = GetDlgItemInt(hWnd, IDC_COLUMN, NULL, FALSE);
         EnableWindow(GetDlgItem(hWnd, IDC_READCELL),
            chINRANGE(0, nCol, g_nNumCols - 1));
         EnableWindow(GetDlgItem(hWnd, IDC_WRITECELL),
            chINRANGE(0, nCol, g_nNumCols - 1));
         break;

      case IDC_READCELL:
         // пытаемся считать значение ячейки, выбранной пользователем
         SetDlgItemText(g_hWnd, IDC_LOG, TEXT("No violation raised"));
         nRow = GetDlgItemInt(hWnd, IDC_ROW, NULL, FALSE);
         nCol = GetDlgItemInt(hWnd, IDC_COLUMN, NULL, FALSE);
         __try {
            SetDlgItemInt(hWnd, IDC_VALUE, g_ss[nRow][nCol].dwValue, FALSE);
         }
         __except (
            g_ssObject.ExceptionFilter(GetExceptionInformation(), FALSE)) {

            // ячейке не передана физическая память, и ее значение не определе-
   но
            SetDlgItemText(hWnd, IDC_VALUE, TEXT(""));
         }
         break;

      case IDC_WRITECELL:
         // пытаемся считать значение ячейки, выбранной пользователем
         SetDlgItemText(g_hWnd, IDC_LOG, TEXT("No violation raised"));
```

```
         nRow = GetDlgItemInt(hWnd, IDC_ROW, NULL, FALSE);
         nCol = GetDlgItemInt(hWnd, IDC_COLUMN, NULL, FALSE);

         // если ячейке не передана физическая память,
         // возбуждается исключение, и память передается ей автоматически
         g_ss[nRow][nCol].dwValue =
            GetDlgItemInt(hWnd, IDC_VALUE, NULL, FALSE);
         break;
   }
}

//////////////////////////////////////////////////////////////////////

INT_PTR WINAPI Dlg_Proc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam) {

   switch (uMsg) {
      chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
      chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}

//////////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hInstExe, HINSTANCE, PTSTR, int) {

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_SPREADSHEET), NULL, Dlg_Proc);
   return(0);
}

//////////////////////////////// End of File /////////////////////////////
```

VMArray.h

```
/******************************************************************************
Module:  VMArray.h
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
******************************************************************************/

#pragma once

//////////////////////////////////////////////////////////////////////////////
```

```
// Примечание: этот C++-класс небезопасен в многопоточной среде.
// Несколько потоков не может одновременно создавать/уничтожать
// объекты этого класса. Однако несколько потоков может одновременно
// обращаться к уже созданным объектам класса CVMArray (для доступа
// к одному объекту вам придется самостоятельно синхронизировать потоки).

//////////////////////////////////////////////////////////////////////////

template <class TYPE>
class CVMArray {
public:
    // резервирует разреженную матрицу
    CVMArray(DWORD dwReserveElements);

    // освобождает разреженную матрицу
    virtual ~CVMArray();

    // обеспечивает доступ к элементу массива
    operator TYPE*()             { return(m_pArray); }
    operator const TYPE*() const { return(m_pArray); }

    // может вызываться для более "тонкой" обработки,
    // если передать память не удалось
    LONG ExceptionFilter(PEXCEPTION_POINTERS pep,
      BOOL bRetryUntilSuccessful = FALSE);

protected:
    // замещается для более "тонкой" обработки исключения,
    // связанного с нарушением доступа
    virtual LONG OnAccessViolation(PVOID pvAddrTouched, BOOL bAttemptedRead,
      PEXCEPTION_POINTERS pep, BOOL bRetryUntilSuccessful);

private:
   static CVMArray* sm_pHead;    // адрес первого объекта
   CVMArray* m_pNext;            // адрес следующего объекта

   TYPE* m_pArray;               // указатель на регион,
                                 // зарезервированный под массив
   DWORD m_cbReserve;            // Размер зарезервированного региона (в бай-
    тах)

private:
   // адрес предыдущего фильтра необработанных исключений
   static PTOP_LEVEL_EXCEPTION_FILTER sm_pfnUnhandledExceptionFilterPrev;
```

```
   // наш глобальный фильтр необработанных исключений для экземпляров этого
    класса
   static LONG WINAPI UnhandledExceptionFilter(PEXCEPTION_POINTERS pep);
};

////////////////////////////////////////////////////////////////////////////

// заголовок связанного списка объектов
template <class TYPE>
CVMArray<TYPE>* CVMArray<TYPE>::sm_pHead = NULL;

// адрес предыдущего фильтра необработанных исключений
template <class TYPE>
PTOP_LEVEL_EXCEPTION_FILTER CVMArray<TYPE>::sm_pfnUnhandledExceptionFilterPrev;

////////////////////////////////////////////////////////////////////////////

template <class TYPE>
CVMArray<TYPE>::CVMArray(DWORD dwReserveElements) {

   if (sm_pHead == NULL) {
      // устанавливаем наш глобальный фильтр необработанных исключений
      // при создании первого экземпляра класса
      sm_pfnUnhandledExceptionFilterPrev =
         SetUnhandledExceptionFilter(UnhandledExceptionFilter);
   }

   m_pNext = sm_pHead; // следующий узел был вверху списка
   sm_pHead = this;    // сейчас вверху списка находится этот узел

   m_cbReserve = sizeof(TYPE) * dwReserveElements;

   // резервируем регион для всего массива
   m_pArray = (TYPE*) VirtualAlloc(NULL, m_cbReserve,
      MEM_RESERVE | MEM_TOP_DOWN, PAGE_READWRITE);
   chASSERT(m_pArray != NULL);
}

////////////////////////////////////////////////////////////////////////////

template <class TYPE>
CVMArray<TYPE>::~CVMArray() {

   // освобождаем регион массива (возвращаем всю переданную ему память)
   VirtualFree(m_pArray, 0, MEM_RELEASE);
```

```
   // удаляем этот объект из связанного списка
   CVMArray* p = sm_pHead;
   if (p == this) {     // удаляем верхний узел
      sm_pHead = p->m_pNext;
   } else {

      BOOL bFound = FALSE;

      // проходим по списку сверху и модифицируем указатели

      for (; !bFound && (p->m_pNext != NULL); p = p->m_pNext) {
         if (p->m_pNext == this) {
            // узел, указывающий на нас, должен указывать на следующий узел
            p->m_pNext = p->m_pNext->m_pNext;
            break;
         }
      }
      chASSERT(bFound);
   }
}

////////////////////////////////////////////////////////////////////////

// предлагаемый по умолчанию механизм обработки нарушений доступа,
// возникающих при попытках передачи физической памяти
template <class TYPE>
LONG CVMArray<TYPE>::OnAccessViolation(PVOID pvAddrTouched,
   BOOL bAttemptedRead, PEXCEPTION_POINTERS pep, BOOL bRetryUntilSuccessful) {

   BOOL bCommittedStorage = FALSE;  // считаем, что передать память не удалось

   do {
      // пытаемся передать физическую память
      bCommittedStorage = (NULL != VirtualAlloc(pvAddrTouched,
         sizeof(TYPE), MEM_COMMIT, PAGE_READWRITE));

      // если передать память не удается, предлагаем пользователю освободить
      // память
      if (!bCommittedStorage && bRetryUntilSuccessful) {
         MessageBox(NULL,
            TEXT("Please close some other applications and Press OK."),
            TEXT("Insufficient Memory Available"), MB_ICONWARNING | MB_OK);
      }
   } while (!bCommittedStorage && bRetryUntilSuccessful);
```

```
   // если память передана, пытаемся возобновить выполнение программы;
   // в ином случае активизируем обработчик
   return(bCommittedStorage
      ? EXCEPTION_CONTINUE_EXECUTION : EXCEPTION_EXECUTE_HANDLER);
}
////////////////////////////////////////////////////////////////////////////////

// фильтр, связываемый с отдельным объектом класса CVMArray
template <class TYPE>
LONG CVMArray<TYPE>::ExceptionFilter(PEXCEPTION_POINTERS pep,
   BOOL bRetryUntilSuccessful) {

   // по умолчанию пытаемся использовать другой фильтр (самый
   // безопасный путь)
   LONG lDisposition = EXCEPTION_CONTINUE_SEARCH;

   // мы обрабатываем только нарушение доступа
   if (pep->ExceptionRecord->ExceptionCode != EXCEPTION_ACCESS_VIOLATION)
      return(lDisposition);

   // получаем адрес и информацию о попытке чтения/записи
   PVOID pvAddrTouched = (PVOID) pep->ExceptionRecord->ExceptionInformation[1];
   BOOL bAttemptedRead = (pep->ExceptionRecord->ExceptionInformation[0] == 0);

   // попадает ли полученный адрес в регион,
   // зарезервированный для этого VMArray?
   if ((m_pArray <= pvAddrTouched) &&
     (pvAddrTouched < ((PBYTE) m_pArray + m_cbReserve))) {

     // была попытка записи в наш массив, пытаемся исправить ситуацию
     lDisposition = OnAccessViolation(pvAddrTouched, bAttemptedRead,
         pep, bRetryUntilSuccessful);
   }

   return(lDisposition);
}

////////////////////////////////////////////////////////////////////////////////

// фильтр, связываемый со всеми объектами класса CVMArray
template <class TYPE>
LONG WINAPI CVMArray<TYPE>::UnhandledExceptionFilter(PEXCEPTION_POINTERS pep) {
```

```
   // по умолчанию пытаемся использовать другой фильтр (самый безопасный путь)
   LONG lDisposition = EXCEPTION_CONTINUE_SEARCH;

   // мы обрабатываем только нарушение доступа
   if (pep->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION) {

      // проходим все узлы связанного описка
      for (CVMArray* p = sm_pHead; p != NULL; p = p->m_pNext) {

         // Интересуемся, может ли данный узел обработать исключение.
         // Примечание: исключение НАДО обработать,  иначе процесс будет за-
   крыт!
         lDisposition = p->ExceptionFilter(pep, TRUE);

         // если подходящий узел найден и он обработал исключение,
         // выходим из цикла
         if (lDisposition != EXCEPTION_CONTINUE_SEARCH)
            break;
      }
   }

   // если ни один узел не смог устранить проблему,
   // пытаемся использовать предыдущий фильтр исключений
   if (lDisposition == EXCEPTION_CONTINUE_SEARCH)
      lDisposition = sm_pfnUnhandledExceptionFilterPrev(pep);

   return(lDisposition);
}

//////////////////////////////// End of File //////////////////////////////////
```

Векторная обработка исключений и обработчики возобновления

Работа механизма SEH, о котором шла речь в главах 23 и 24, основана на *фреймах*. То есть, каждый раз, когда поток входит в блок (или фрейм) *try*, к связанному списку фреймов добавляется узел. Если возникает исключение, система просматривает занесенные в связанный список фреймы. Просмотр начинается с фрейма, в который поток вошел последним и заканчивается первым фреймом. При этом система ищет обработчики, указанные в блоках *catch*, связанных с фреймами *try*. Обнаружив обработчик в блоке *catch*, система снова просматривает связанный список, исполняя код в блоках *finally*. После завершения раскрутки либо после нормального (без исключений) завершения кода в блоке *try* фреймы удаляются из связанного списка.

Windows также поддерживает механизм векторной обработки исключений (vectored exception handling, VEH), работающий совместно с SEH. Вместо конструкций языка программирования в этом механизме используются функции, которые регистрируются и вызываются каждый раз, когда возникает исключение, либо при обнаружении исключений, «просочившихся» сквозь стандартные механизмы SEH.

Функция *AddVectoredExceptionHandler* отвечает за регистрацию обработчика исключения (exception handler), который добавляется во внутренний список функций, вызываемых при возникновении исключения в любом из потоков процесса:

```
PVOID AddVectoredExceptionHandler (
    ULONG bFirstlnTheList,
    PVECTORED_EXCEPTION_HANDLER pfnHandler);
```

Параметр *pfnHandler* — это указатель на векторный обработчик исключения. Функция, которая играет эту роль, должна соответствовать следующей сигнатуре:

```
LONG WINAPI ExceptionHandler(struct _EXCEPTION_POINTERS* pExceptionlnfo);
```

Если параметр *bFirstInTheList* равен 0, то функция, заданная параметром *pfnHandler*, добавляется к концу внутреннего списка. В противном случае *bFirstInTheList* заданная функция добавляется в начало внутреннего списка. Когда возникает исключение, функции из списка VEH вызываются поочередно (но до вызова любых фильтров SEH). Если вызов устранил проблему, он должен вернуть EXCEPTION_CONTINUE_EXECUTION, чтобы система возобновила исполнение, начав с повтора команды, исходно вызвавшей исключение; в этом случае SEH-фильтры так и не вступают в игру. Если же вызванный обработчик не справился с ошибкой, он должен вернуть EXCEPTION_CONTINUE_SEARCH, чтобы другие обработчики, внесенные в список, получили возможность обработать это исключение. Если все VEH-обработчики вернули EXCEPTION_CONTINUE_SEARCH, начинается обработка исключения с помощью SEH-фильтров. Заметьте, что VEH-фильтрам запрещено возвращать EXCEPTION_EXECUTE_HANDLER.

Ранее установленный VEH-обработчик исключений можно удалить из внутреннего списка вызовом следующей функции:

```
ULONG RemoveVectoredExceptionHandler (PVOID pHandler);
```

Параметр *pHandler* задает описатель ранее установленной функции-обработчика, возвращаемый функцией *AddVectoredExceptionHandler*.

> **Примечание.** Рекомендую также изучить статью Мэтта Петрека (Matt Pietrek), посвященную реализации перехвата API-вызовов внутри процессов с применением точек прерывания («Under the Hood: New Vectored Exception Handling in Windows XP», см. http://msdn.microsoft.com/msdn/mag/issues/01/09/hood/). Этот метод отличается от показанного мной в главе 22.

VEH-обработчики не только обрабатывают исключения прежде, чем это сделают SEH, но и позволяют уведомить разработчика о возникновении необработанных исключений. Для этого необходимо зарегистрировать обработчик возобновления (continue handler) вызовом следующей функции:

```
PVOID AddVectoredContinueHandler (
    ULONG bFirstlnThelist,
    PVECTORED_EXCEPTION_HANDLER pfnHandler);
```

Если параметр *bErstInTheList* равен 0, переданная в параметре *pfnHandler* функция добавляется к концу внутреннего списка обработчиков возобновления; в противном случае эта функция добавляется в начал списка. Когда возникает необработанное исключение, функции-обработчики возобновления из списка вызываются поочередно. В частности, это происходите, когда глобальный фильтр необработанных исключений, установленный вызовом *SetUnhandl edExceptionFilter*, возвращает EXCEPTION_CONTINUE_SEARCH. Функция-обработчик возобновления может, вернув EXCEPTION_CONTINUE_EXECUTION, отменить исполнение остальных функций в списке обработчиков возобновления и заставить систему повторно выполнить команду, вызвавшую исключение. Обработчик исключения также может вернуть значение EXCEPTION_CONTINUE_SEARCH, и тогда остальные функции-обработчики будут вызваны.

Чтобы удалить из списка ранее установленную функцию-обработчик возобновления, вызовите следующую функцию:

```
ULONG RemoveVectoredContinueHandler (PVOID pHandler);
```

Параметр *pHandler* задает описатель ранее установленной функции-обработчика, возвращенный функцией *AddVectoredContinueHandler*.

Несложно догадаться, что обработчики возобновления применяются, главным образом, для трассировки и диагностики.

Исключения C++ и структурные исключения

Разработчики часто спрашивают меня, что лучше использовать: SEH или исключения C++. Ответ на этот вопрос вы найдете здесь.

Для начала позвольте напомнить, что SEH механизм операционной системы, доступный в любом языке программирования, а исключения C++ поддерживаются только в C++. Создавая приложение на C++, вы должны использовать средства именно этого языка, а не SEH. Причина в том, что исключения C++ - часть самого языка и его компилятор автоматически создает код, который вызывает деструкторы объектов и тем самым обеспечивает корректную очистку ресурсов.

Однако вы должны иметь в виду, что компилятор Microsoft Visual C++ реализует обработку исключений C++ на основе SEH операционной системы. Например, когда вы создаете C++-блок *try*, компилятор генерирует SEH-блок *__try*. C++-блок *catch* становится SEH-фильтром исключений, а

код блока *catch* — кодом SEH-блока *__except*. По сути, обрабатывая C++-оператор *throw*, компилятор генерирует вызов Windows-функции *RaiseException*, и значение переменной, указанной в *throw*, передается этой функции как дополнительный аргумент.

Сказанное мной поясняет фрагмент кода, показанный ниже. Функция слева использует средства обработки исключений C++, а функция справа демонстрирует, как компилятор C++ создает соответствующие им SEH-эквиваленты.

```
void ChunkyFunky() {
    try {
        // тело блока try
        …
        throw 5;


    }
    catch (int x) {


        // тело блока catch
        …
    }
    …
}
```

```
void ChunkyFunky() {
    __try {
        // тело блока try
        …
        RaiseException(Code=0xE06D7363,
            Flag=EXCEPTION_NONCONTINUABLE,
            Args=5);
    }
    __except ((ArgType == Integer) ?
        EXCEPTION_EXECUTE_HANDLER :
        EXCEPTION_C0NTINUE_SEARCH) {
        // тело блока catch
        …
    }
    …
}
```

Обратите внимание на ряд интересных особенностей этого кода. Во-первых, *RaiseException* вызывается с кодом исключения 0xE06D7363. Это код программного исключения, выбранный разработчиками Visual C++ на случай генерации (throwing) исключений C++. Вы можете сами в этом убедиться, преобразовав этот код в AsCII-символы: в результате получится «.msc».

Заметьте также, что при генерации исключения C++ всегда используется флаг EXCEPTION_NONCONTINUABLE. Исключения C++ не разрешают возобновлять выполнение программы, и возврат EXCEPTION_CONTINUE_EXECUTION фильтром, диагностирующим исключения C++, был бы ошибкой. Если вы посмотрите на фильтр *__except* в функции справа, то увидите, что он возвращает только EXCEPTION_EXECUTE_HANDLER или EXCEPTION_CONTINUE_SEARCH.

Остальные аргументы *RaiseException* используются механизмом, который фактически генерирует исключение. Точный механизм того, как данные из переменной передаются *RaiseException*, не задокументирован, но догадаться о его реализации в компиляторе не так уж трудно.

Последнее, на что я хочу обратить ваше внимание, — фильтр *__except*. Он служит для сравнения типа данных переменной *throw* с типом переменной, используемой C++-оператором *catch*. Если эти переменные одного типа, фильтр возвращает EXCEPTION_EXECUTE_HANDLER, заставляя систе-

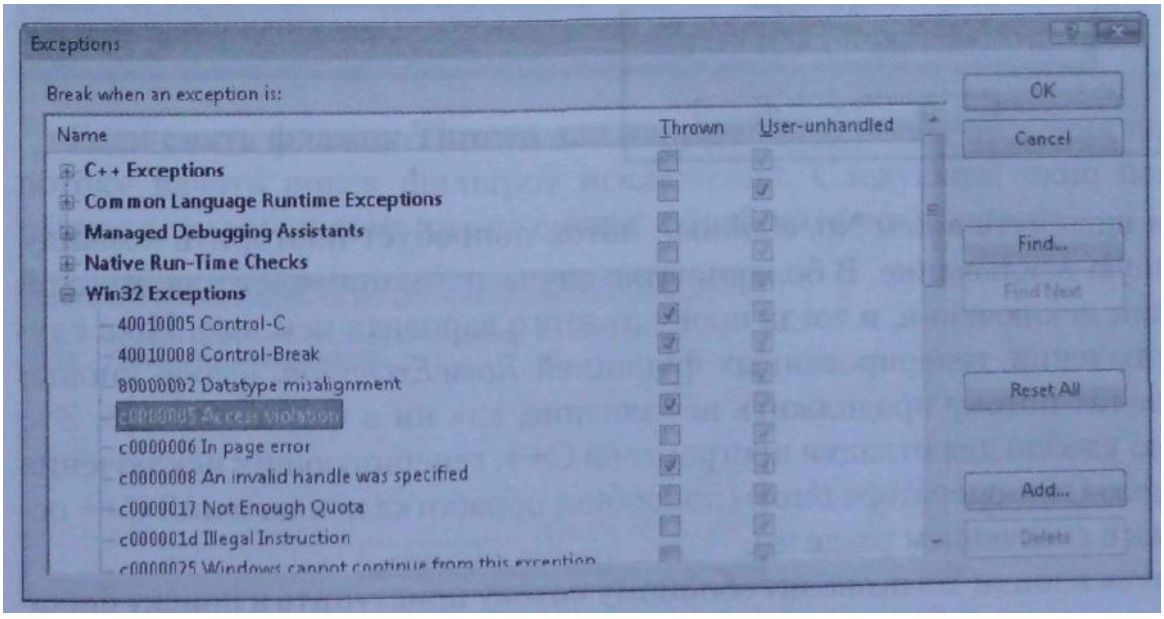
му исполнить код в блоке *catch*. В противном случае фильтр возвращает EXCEPTION_CONTINUE_SEARCH, и система проверяет фильтры *catch*, расположенные выше в дереве вызовов.

> **Примечание.** Так как исключения C++ реализуются через SEH, оба эти механизма можно использовать в одной программе. Например, я предпочитаю передавать физическую память при исключениях, вызываемых нарушениями доступа. Хотя C++ вообще не поддерживает этот тип обработки исключений (с возобновлением выполнения), он позволяет применять SEH в тех местах программы, где это нужно, и ваш фильтр __except может возвращать EXCEPTION_CONTINUE_EXECUTION. Ну а в остальных частях исходного кода, где возобновление выполнения после обработки исключения не требуется, я пользуюсь механизмом обработки исключений, предлагаемым C++.

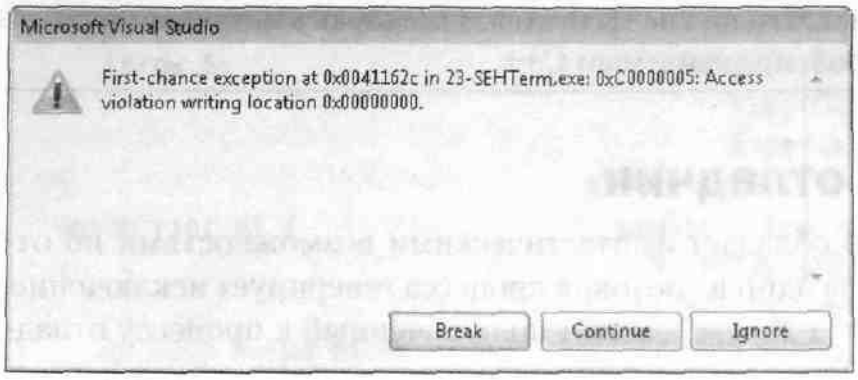
Исключения и отладчик

Отладчик Visual Studio обладает фантастическими возможностями по отладке исключений. Когда один из потоков процесса генерирует исключение, операционная система тут же уведомляет подключенный к процессу отладчик (если он имеется). Такое уведомление называется *уведомлением первого шанса* (first-chance notification). Обычно отладчик, реагируя на это уведомление, заставляет поток начать поиск фильтра исключения. Если все фильтры исключений вернут EXCEPTION_CONTINUE_SEARCH, операционная система направляет отладчику другое уведомление — т.н. *уведомление последнего шанса* (last-chance notification). Эти два типа уведомлений дают разработчикам больше контроля над отладкой и обработкой исключений.

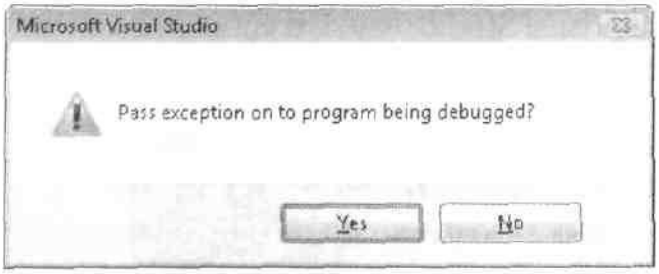
При настройке отдельных решений в Visual Studio окно Exceptions (см. ниже) позволяет настроить реакцию отладчика на уведомления первого шанса



Видно, что исключения в этом окне рассортированы по категориям, включая Win32 Exceptions, в которой находятся все системные исключения. Для каждого исключения отображается 32-разрядный код, текстовое описание и действие отладчика при получении уведомления первого шанса (флажок Thrown) и уведомления последнего шанса (флажок User-Unhandled). Учтите, что последний параметр применим только к исключениям CLR. В показанном выше окне я выбрал исключение, возникающее при нарушении доступа, и задал остановку как действие, которое отладчик выполняет при генерации этого исключении. Теперь, как только в одном из потоков отлаживаемого процесса возникнет нарушение доступа, отладчик получит уведомление первого шанса и покажет сообщение следующего вида:



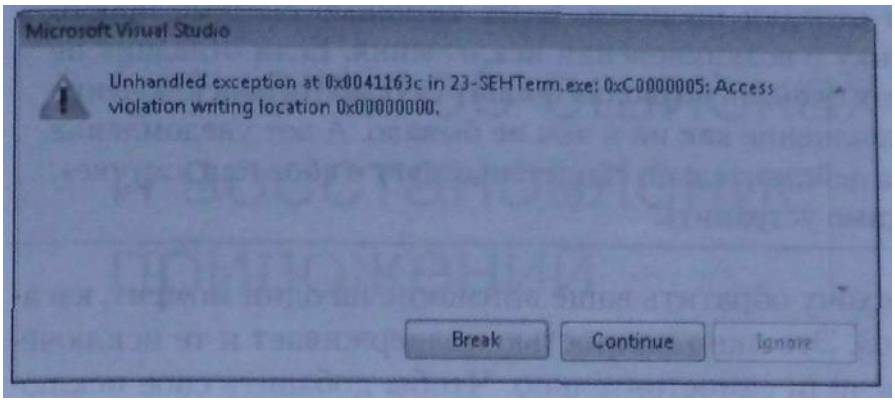
У потока еще не было шанса найти фильтр исключений. Теперь я могу расставить точки прерывания в коде, проверить значения переменных и стек вызовов потока. Ни один из фильтров еще не вызван: исключение только что возникло. Если сейчас начать в отладчике пошаговое исполнение, откроется следующее окно:



Если щелкнуть в нем No, сбойный поток попробует повторить команду, вызвавшую исключение. В большинстве случаев это приведет к повторной генерации исключения, и тогда проку от этого варианта немного. Но в случае исключений, генерированных функцией *RaiseException*, выбор кнопки No позволит потоку продолжить исполнение, как ни в чем не бывало. Это особенно удобно для отладки программ на C++, генерирующих исключения с помощью C++-оператора *throw* (подробнее обработка исключений C++ освещалась в следующем разделе).

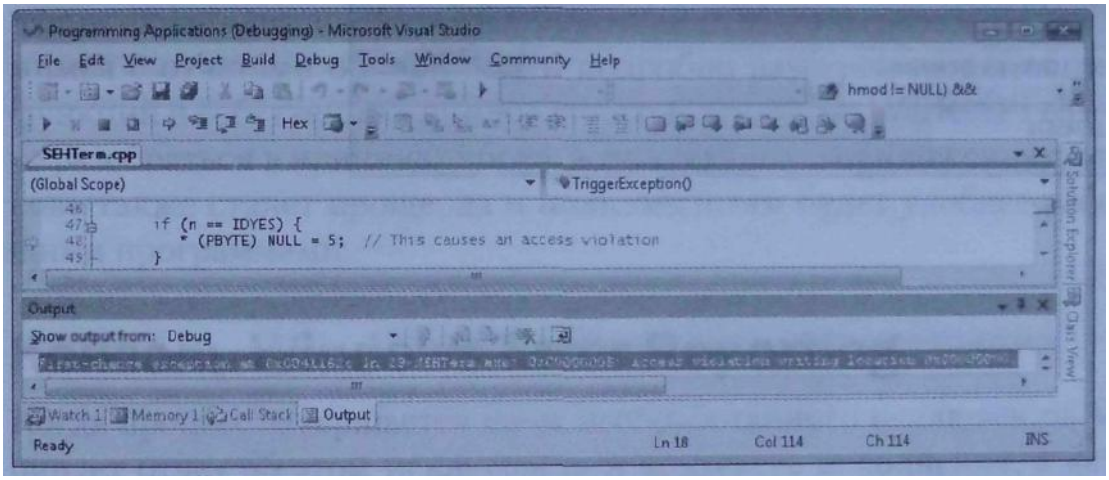
Щелчок кнопки Yes позволит сбойному потоку приступить к поиску фильтров исключений. Если найденный фильтр исключений возвращает EXCEPTION_EXECUTE_HANDLER или EXCEPTION_CONTINUE_XECUTION,

все хорошо, и поток продолжает исполнение. Если же все фильтры вернут EXCEPTION_CONTINUE_SEARCH, отладчик получит уведомление последнего шанса и откроет такое окно:

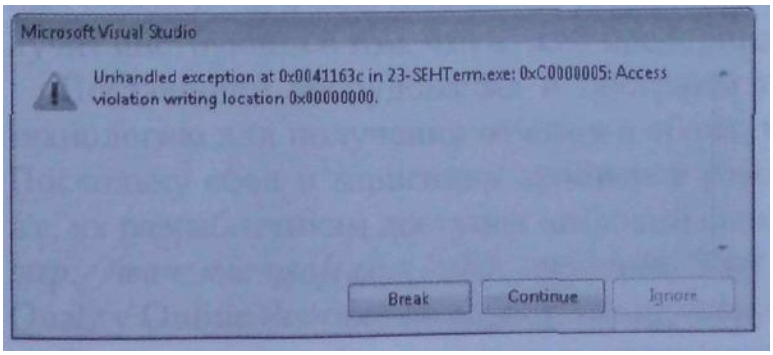


Теперь остается лишь два варианта: отладка приложения либо его завершения.

Описанные выше события происходят, если настроить отладчик для остановки исполнения при возникновении исключения (что я и сделал, пометив флажок Thrown). Однако для большинства исключений этот флажок по умолчанию снят. В этом случае отладчик, получив уведомление первого шанса, просто показывает уведомление в своем окне Output:

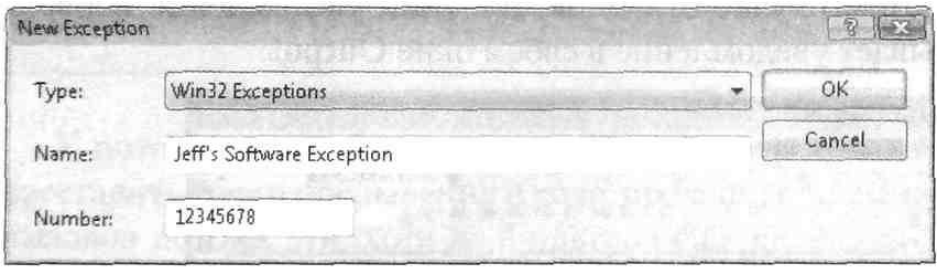


Если снять флажок Thrown для нарушения доступа, отладчик позволит потоку начать поиск фильтров исключений. Следующее окно появится, только если ни один из них не сможет обработать исключение:



> **Примечание.** Важно помнить, что уведомления первого шанса сами по себе не означают, что в код системы или приложений вкралась серьезная ошибка. На самом деле, эти уведомления могут направляться только во время отладки процесса. С их помощью система просто сообщает отладчику о возникновении исключения. Если отладчик не открывает никаких окон, он позволит фильтру обработать исключение и продолжить исполнение, как ни в чем не бывало. А вот уведомление последнего шанса действительно свидетельствует о сбое или «жучке», который необходимо устранить.

В завершение главы хочу обратить ваше внимание на один момент, касающийся окна Exceptions. Это окно полностью поддерживает и те исключения, которые вы определили самостоятельно. Чтобы добавить свое исключение в список, достаточно щелкнуть кнопку Add — откроется окно New Exception. В нем следует выбрать тип исключения (Win32 Exceptions) и ввести имя вашего исключения, затем число, которое станет его кодом, и щелкнуть ОК — исключение появится в списке. На рисунке ниже показано добавленное мной программное исключение:



Оглавление

ГЛАВА 26

Отчеты об ошибках и восстановление приложений

В предыдущей главе мы обсудили, как механизмы VEH, SEH и служба Windows Error Reporting (WER) взаимодействуют для получения и регистрации информации о сбоях в приложениях. В этой главе мы ближе познакомимся с отчетами об ошибках и подробно разберем использование WER-функций в собственных приложениях. С помощью WER API вам будет легче разобраться в причинах сбоев, возникающих в приложениях, отлов «жучков» также станет проще, да и пользователям будет удобнее работать с вантами программами.

Консоль Windows Error Reporting

Когда процесс завершается из-за необработанного исключения, WER генерирует отчет об этом исключении и контексте исполнения, в котором оно возникло.

С разрешения пользователя этот отчет по защищенному каналу передается на серверы Майкрософт, где содержимое отчета сравнивается с базой известных сбоев. Если в базе есть метод устранения данного сбоя, пользователь получит инструкции о том, что нужно предпринять, чтобы продолжить работу.

Поставщики оборудования и программ также могут использовать эту технологию для получения отчетов о сбоях, возникающих в их продукции. Поскольку сбои и зависания драйверов режима ядра обрабатываются так же, их разработчикам доступен широкий спектр решений (см. веб-страницу http://www.microsoft.com/whdc/maintain/StartWER.mspx и сайт Windows Quality Online Services по адресу https://winqual.microsoft.com).

Если пользователь отказался от отправки отчета в Майкрософт, сгенерированный отчет сохраняется на машине пользователя. С помощью консоли

WER пользователь может просматривать и анализировать сведения о сбоях, возникших на его компьютере.

На рис. 26-1 показан аплет Problem Reports And Solutions, доступный на Панели управления (%SystemRoot%\system32\wercon.exe).

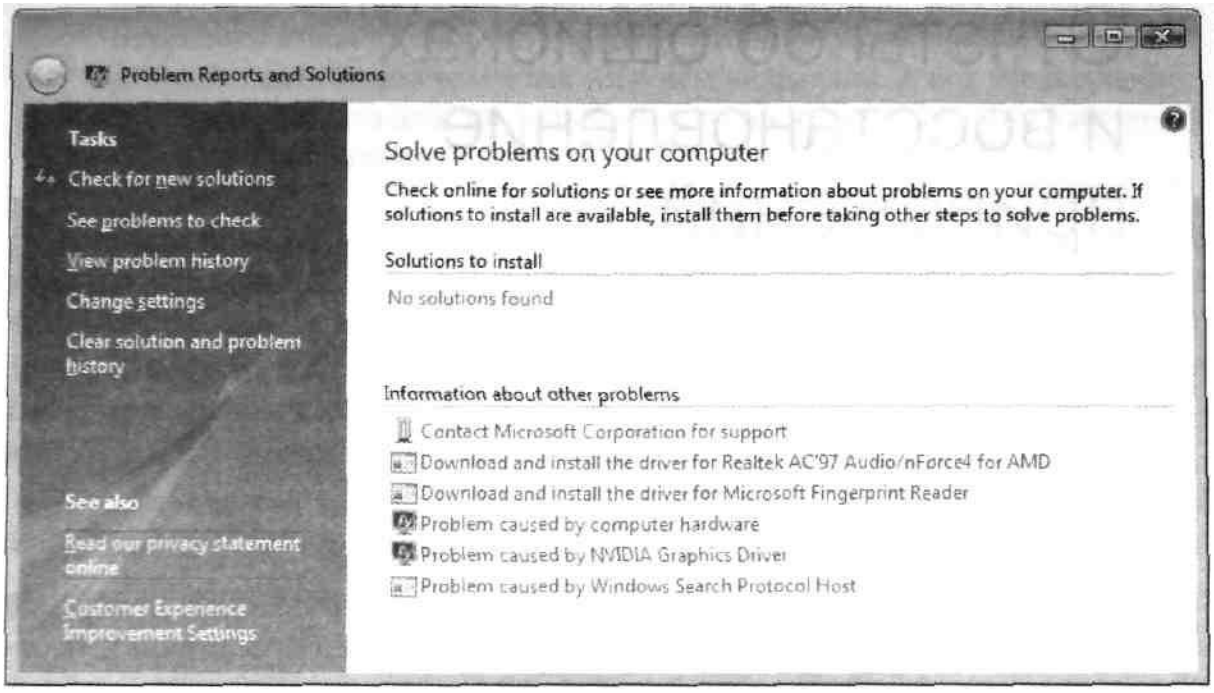


Рис. 26-1. *Консоль WER доступна через Control Panel*

Щелчок ссылки View Problem History слева выводит в окне консоли WER список всех крахов и зависаний процессов (рис. 26-2), а также других сбоев, включая ненайденные драйверы и крахи системы.

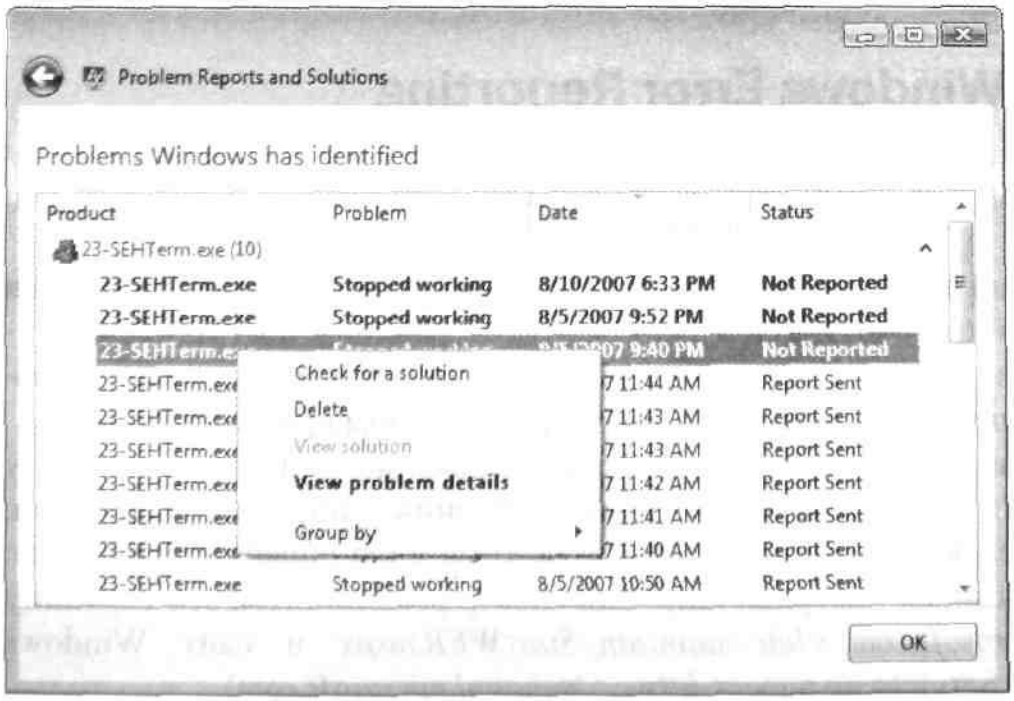


Рис. 26-2. *Список крахов приложений в консоли WER (упорядоченный по категории Product)*

Заметьте, что в столбце Status показано, по каким сбоям отчеты в Майкрософт уже отправлены. Сбои, отчеты по которым еще не отправлены, выделены полужирным шрифтом. Если щелкнуть запись о сбое правой кнопкой, можно проверить наличие решения для этой проблемы, удалить отчет либо вывести подробное описание сбоя. Команда View Problem Details (либо двойной щелчок записи о сбое) открывает окно, показанное на рис. 26-3.

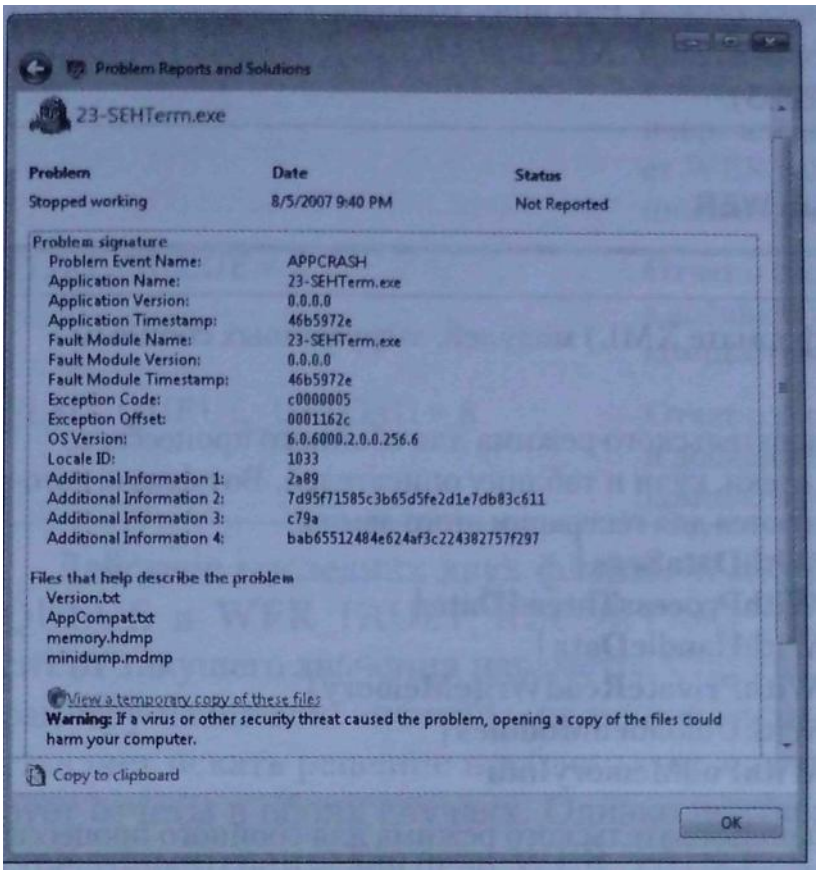


Рис. 26-3. *Отчет о сбое в окне консоли WER*

В окне итогов отображается сводная информация о сбое, взятая, главным образом, из поля *ExceptionInformation* структуры EXCEPTION_RECORD, передаваемой функции *UnhandledExceptionFilter* как Exception Code (код нарушения доступа — c0000005, см. рис. 26-3). Смысл этой информации для пользователей неясен, интерпретировать ее смогут только разработчики. Однако и этих сведений недостаточно, чтобы понять, что же все-таки случилось с программой. Но не беспокойтесь: ссылка View A Temporary Copy Of These Files открывает доступ к файлам, сгенерированным WER при вызове *UnhandledExcepionFilter* (см. табл. 26-1). По умолчанию эти файлы доступны, пока отчет о сбое не будет отправлен в Майкрософт. Ниже я покажу, как заставить WER сохранять эти файлы и после отправки отчета.

Самая интересная часть отчета — файл Memory.hdmp. С его помощью можно попытаться отладить программу в вашем любимом отладчике даже после ее краха. При этом отладчик может найти ту самую команду, исполнение которой вызвало исключение!

Примечание. В будущих версиях Windows имя файла дампа может измениться. Так, в него может быть добавлено имя сбойного приложения, расширение же, скорее всего, останется прежним — .hdmp или .mdmp. Например, вместо Memory.hdmp и MiniDump.mdmp будут генерироваться файлы MyApp.exe.hdmp и MyApp.exe.mdmp. Все, что нужно знать о мини-дампах, вы найдете в книге «Отладка приложений для *Microsoft.NET and Microsoft Windows»* (Джон Роббинс, Русская Редакция, 2004) *«Debugging Applications for Microsoft .NET and Microsoft Windows»* (John Robbins, Microsoft Press, 2003).

Табл. 26-1. Файлы, генерируемые WER

Имя	Описание
AppCompat.txt	Список (в формате XML) модулей, загруженных сбойным процессом
Memory.hdmp	Дамп пользовательского режима для сбойного процесса, содержит его стеки, кучи и таблицу описателей. Вот флаги, которые используются для генерации этого дампа: **MiniDumpWithDataSegs |** **MiniDumpWithProcessThreadData |** **MiniDumpWithHandleData |** **MiniDumpWithPrivateReadWriteMemory |** **MiniDumpWithUnloadedModules |** **MiniDumpWithFullMemorylnfo**
MiniDump.mdmp	Мини-дамп пользовательского режима для сбойного процесса. Вот флаги, которые используются для генерации этого дампа: **MiniDumpWithDataSegs |** **MiniDumpWithUnloadedModules |** **MiniDumpWithProcessThreadData**
Version.txt	Описание экземпляра Windows **Windows NT Version 6.0 Build: 6000** **Product (0x6): Windows Vista (TM) Business Edition: Business** **BuildString: 6000.16386.x86fre.vista_rtm.061101-2205** **Flavor: Multiprocessor Free** **Architecture: X86** **LCID: 1033**

Программная генерация отчетов об ошибках в Windows

Следующая функционирования, экспортированная kernel32.dll и объявленная в werapi.h, позволяет настраивать некоторые параметры вашего процесса (см. табл. 26-2).

```
HREZULT WerSetFlags (DWORD dwFlags)
```

Табл. 26-2. Параметры WerSetFlags

Параметр WER_FAULT_REPORTING_*	Описание
FLAG NOHEAP = 1	Запрещает включать в отчет содержимое кучи, это удобно, если требуется ограничить размер отчета
FLAG_DISABLE_THREAD_ SUSPENSION = 4	По умолчанию WER приостанавливает все потоки процесса, взаимодействующего с пользователем, чтобы они не усугубили повреждения данных. Данный флаг запрещает WER делать это, и потому является потенциально опасным
FLAG_QUEUE = 2	Отчет о сбое не отправляется в Майкрософт, а добавляется в очередь на локальном компьютере
FLAG_QUEUE_UPLOAD = 8	Отчет о сбое отправляется в Майкрософт и добавляется в очередь на локальном компьютере

Действие последних двух флагов, WER_FAULT_REPORTING_FLAG_QUEUE и WER_FAULT_REPORTING_FLAG_QUEUE_UPLOAD, зависит от текущего значения параметра Consent (см. рис. 25-5). Если этому параметру назначено значение, отличное от значения по умолчанию (оно заставляет искать решение проблемы на сервере Майкрософт), WER генерирует отчеты в обоих случаях. Однако подтверждение отправки запрашивается, только если задан флаг WER_FAULT_REPORTING_FLAG_QUEUE_UPLOAD. Если же задан флаг WER_FAULT_REPORTING_FLAG_QUEUE, отчет не отправляется. Приложение не может заставить систему отправить отчет без разрешения пользователя (или администратора) компьютера, поскольку приоритет этих параметров выше, чем у функций WER.

> **Примечание.** Сгенерированный отчет добавляется в очередь на локальном компьютере. Если параметр *Consent* разрешает, отчет затем отправляется в Майкрософт, но сведения о нем сохраняются, поэтому запись о сбое отображается консолью WER. Если же значение *Consent* запрещает автоматическую отправку отчетов и поиск способа устранения сбоя, WER спрашивает у пользователя, что делать дальше. Если пользователь запретит отправку отчета, его данные сохраняются в локальной очереди и отображаются в консоли WER.

Узнать текущие параметры процесса позволяет следующая функция:

```
HRESULT WerGetFlags(HANDLE hProcess,  PDWORD pdwFlags);
```

Первый параметр, *hProcess*, — это описатель интересующего вас процесса (с правом доступа PROCESS_VM_READ). Получить этот описатель можно вызовом *GetCurrentProcess*.

Внимание! Если не вызвать *WerSetFlags* перед функцией *WerGetFlags*, последняя вернет WERE_NOT_FOUND.

Отключение генерации и отправки отчетов

Приложение может запретить службе WER генерировать и отправлять отчеты в случае сбоев. Это, безусловно, очень удобно при разработке и тестировании приложений. Чтобы отключить создание и отправку отчетов, вызовите следующую функцию:

```
HRESULT WerAddExcludedApplication(PCWSTR pwzExeName,  BOOL bAllUsers);
```

Параметр *pwzExeName* задает имя вашего .exe-файла (или полный путь к нему), включая расширение. Параметр *bAllUser* определяет, следует ли отключить создание отчетов при сбоях этой программы для всех пользователей компьютера либо только для пользователя, зарегистрированного в системе в данный момент. Если в этом параметре передается TRUE, приложение должно быть запущено под учетной записью администратора с соответствующими привилегиями, иначе функция вернет E_ACCESSDENIED (см. раздел «Работа администратора с пользовательскими полномочиями»). При возникновении необработанного исключения в программе, для которой отключена генерация отчетов, WER не генерирует отчет, но все равно запускает WerFault.exe, чтобы дать пользователю выбрать между отладкой и завершением приложения (рис 26-4).

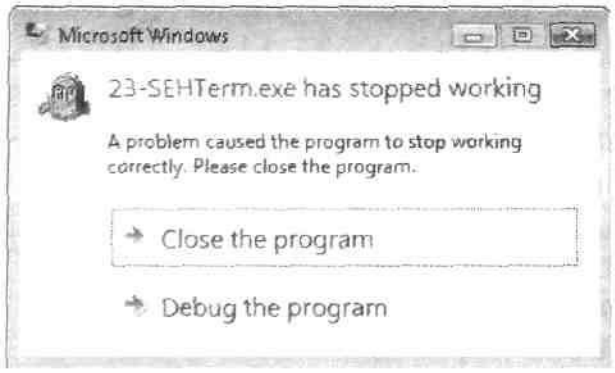


Рис. 26-4. *Возможности, которые остаются у пользователя после отключения генерации отчетов*

Чтобы вновь включить генерацию отчетов для некоторого приложения, вызовите функцию *WerRemoveExcludedApplication:*

```
HRESULT WerRemoveExcludedApplication(PCWSTR pwzExeName,  BOOL bAHUsers);
```

Примечание. Обе эти функции экспортированы wer.dll и объявлены в werapi.h.

Настройка генерируемых для процесса отчетов о сбоях

Иногда требуется, чтобы приложения генерировали нестандартные отчеты о сбоях путем вызова различных WER-функций. Это удобно:

- при написании собственных фильтров необработанных исключений;
- когда приложение должно генерировать отчеты, даже если необработанных исключений не было;
- если необходимо добавить в отчет дополнительную информацию.

Проще всего настроить отчет, указав дополнительные блоки данных и файлы, которые нужно включать в отчеты, генерируемые для вашего процесса. Чтобы добавить к отчету произвольный блок данных, достаточно вызвать следующую функцию:

```
HRESULT WerRegisterMemoryBlock(PVOID pvAddress,  DWORD dwSize);
```

Адрес нужного блока передается в параметре *pvAddress*, а его размер — в параметре *dwSize*. Теперь при генерации отчета заданный блок будет сохранен в мини-дампе, благодаря чему его можно будет изучить при «посмертной» отладке процесса. Заметьте, что функцию *WerRegisterMemoryBlock* можно вызывать несколько раз, чтобы сохранить в мини-дампе несколько блоков данных.

Чтобы добавить к отчету произвольный файл, вызовите следующую функцию:

```
HRESULT WerRegisterFile(
    PCWSTR pwzFilename,
    WER_REGISTER_FILE_TYPE regFileType,
    DWORD dwFlags);
```

Путь к файлу передается в параметре *pwzFilename*. Если вместо пути передано только имя файла, WER будет искать файл в рабочем каталоге. Параметр принимает одно из значений, перечисленных в таблице 26-3.

Табл. 26-3. Типы файлов, добавляемых к отчету о сбое

Значение regFileType	Добавляемый файл
WerRegFileTypeUserDocument = 1	В файле могут содержаться конфиденциальные пользовательские данные. По умолчанию этот файл не отправляется в Майкрософт, но в дальнейшем планируется открыть разработчикам доступ к этим файлам через сайт Windows Quality
WerRegFileTypeOther = 2	Любой другой файл

Параметр *dwFlags* — результат поразрядного сложения значений, описанных в табл. 26-4.

Табл. 26-4. Флаги, связанные с добавляемыми файлами

Флаг	Описание
DELETE_WHEN_DONE = 1	После отправки отчета файл следует удалить
ANONYMOUS_DATA = 2	Файл не содержит персональной информации, по которой можно определить пользователя. Если этот флаг не задан, при первом запросе сервера Майкрософт на передачу этого файла система спрашивает разрешение у пользователя. Если пользователь соглашается, параметру Consent в системном реестре присевается значение 3. После этого все файлы, помеченные этим флагом, будут отправляться без запроса подтверждения у пользователя

Зарегистрированные таким образом файлы будут сохранены в отчете. Заметьте, что функцию *WerRegisterFile* можно вызывать многократно, чтобы добавить к отчету несколько файлов.

> **Примечание.** Число элементов отчета WER, регистрируемых вызовами *WerRegisterMemoryBlock* и *WerRegisterFile*, не может быть больше, чем задано параметром WER_MAX_REGISTERED_ENTRIES (в настоящее время — 512). Возвращаемое в случае этой ошибки значение HRESULT можно сопоставить с кодом ошибки Win32 следующим образом: if (HRESULT_ CODE(hr) == ERROR_INSUFFICIENT_BUFFER).

В завершение замечу, что для удаления из отчетов блоков данных и файлов служат следующие функции:

```
HRESULT WerUnregisterMemoryBlock(PVOID pvAddress);
HRESULT WerUnregisterFile(PCWSTR pwzFilePath);
```

Создание и настройка отчетов о сбоях

В этом разделе я расскажу, как создавать нестандартные отчеты о сбоях в собственных приложениях. Соответствующие функции позволяют создавать отчеты не только о необработанных исключениях, но и о проблемах, никак с ними не связанных. Кроме того, после генерации отчета приложения не обязано завершаться, и может спокойно работать дальше. Так что стоит подумать об использовании службы Windows Error Reporting вместо добавления непонятных сообщений в журнал событий Windows. Однако инфраструктура WER ограничивает число и размеры отчетов посредством параметров системного реестра, расположенных в разделе HKEY_ CURRENT_USER\Software\Microsoft\Windows\Windows Error Reporting (табл. 26-5).

Табл. 26-6. Параметры реестра, управляющие хранением данных WER

Параметр	Описание
MaxArchiveCount	Максимальное число файлов в архиве (допустимые значения — 1-5000; значение по умолчанию — 1000)
MaxQueueCount	Максимальное число отчетов, которое можно поставить в очередь на локальном компьютере до отправки их в Майкрософт (допустимые значения — 1-500; значение по умолчанию — 50)

Примечание. Результаты трассировки отчетов, отправленных в Microsoft (за исключением вложенных в отчет файлов), архивируется в каталоге AppData\Local\Microsoft\Windows\WER\ReportArchive профиля текущего пользователя. Очередь отчетов, которые еще не отправлены, хранится в папке AppData\Local\Microsoft\Windows\WER\ReportQueue текущего пользователя. К сожалению, API, который консоль WER использует для работы с этими отчетами, не задокументирован, поэтому вы не сможете перебрать отчеты в своем коде. Будем надеяться, что в будущих версиях Windows это изменится.

Создание, настройка и передача отчетов о сбоях службе WER осуществляется путем вызова следующих функций в указанной последовательности:

1. Вызов *WerReportCreate* создает новый отчет о сбое.
2. Функция *WerReportSetParameter* служит для настройки параметров отчета.
3. Вызов *WerReportAddDump* добавляет к отчету мини-дамп.
4. Функция *WerReportAddFile* добавляет к отчету произвольные файлы (например, пользовательские документы).
5. Функция *WerReportSetUIOption* модифицирует текст, который отображается пользователю при вызове *WerReportSubmit.*
6. Вызов *WerReportSubmit* передает отчет WER. В зависимости от заданных флагов, Windows может поставить запрос в очередь, либо запросить у пользователя подтверждение и отправить отчет в Майкрософт.
7. Вызов *WerReportCloseHandle* закрывает отчет.

А теперь разберем эту процедуры подробнее. При чтении следующих разделов вам, возможно, пригодится описание функции *GenerateWerReport,* применяемой в приложении-примере Customized WER (см. ниже).

После запуска этой программы откройте консоль WER и щелкните ссылку View Problem History, Чтобы получить список отчетов (рис. 26-5).

В записи о сбоях, вызванных программой 23-SEHTerm.exe, имя продукта соответствует имени исполняемого файла программы. В отличие от нее, отчет о сбое 25-HandleUnhandled.exe выводится в секции Wintellect Applications Suite. В столбце Problem также выводится информация о сбое, но более подробная, чем комментарий по умолчанию «Stopped working».

Если двойным щелчком отрыть подробности отчета, будет видно, что его содержимое также изменено (рис. 26-6) с помощью *Wer**-функций (сравните рис. 26-6 и 26-3).

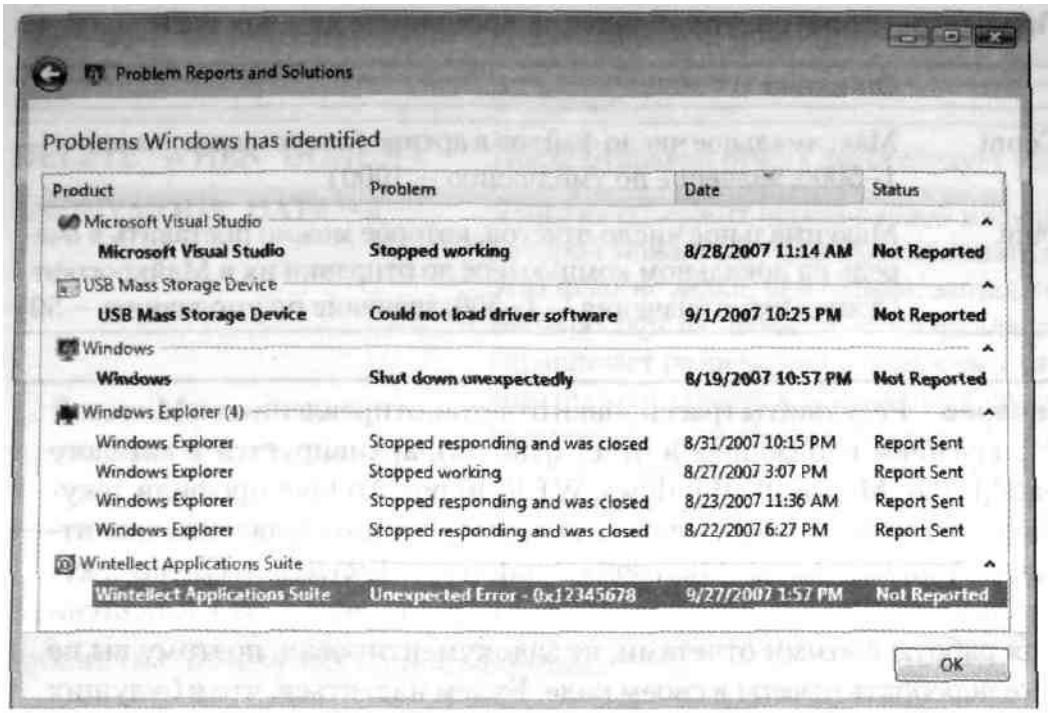


Рис. 26-5. *Список отчетов в консоли WER, отсортированный по названию продукта*

Заголовок отчета и название сбоя — те же, что в сводке, отображаемой в списке консоли WER. Новое поле Description выводит краткое описание сбоя, а значение Problem Event Name теперь содержит больше полезной информации (а не только строку APPCRASH, как прежде).

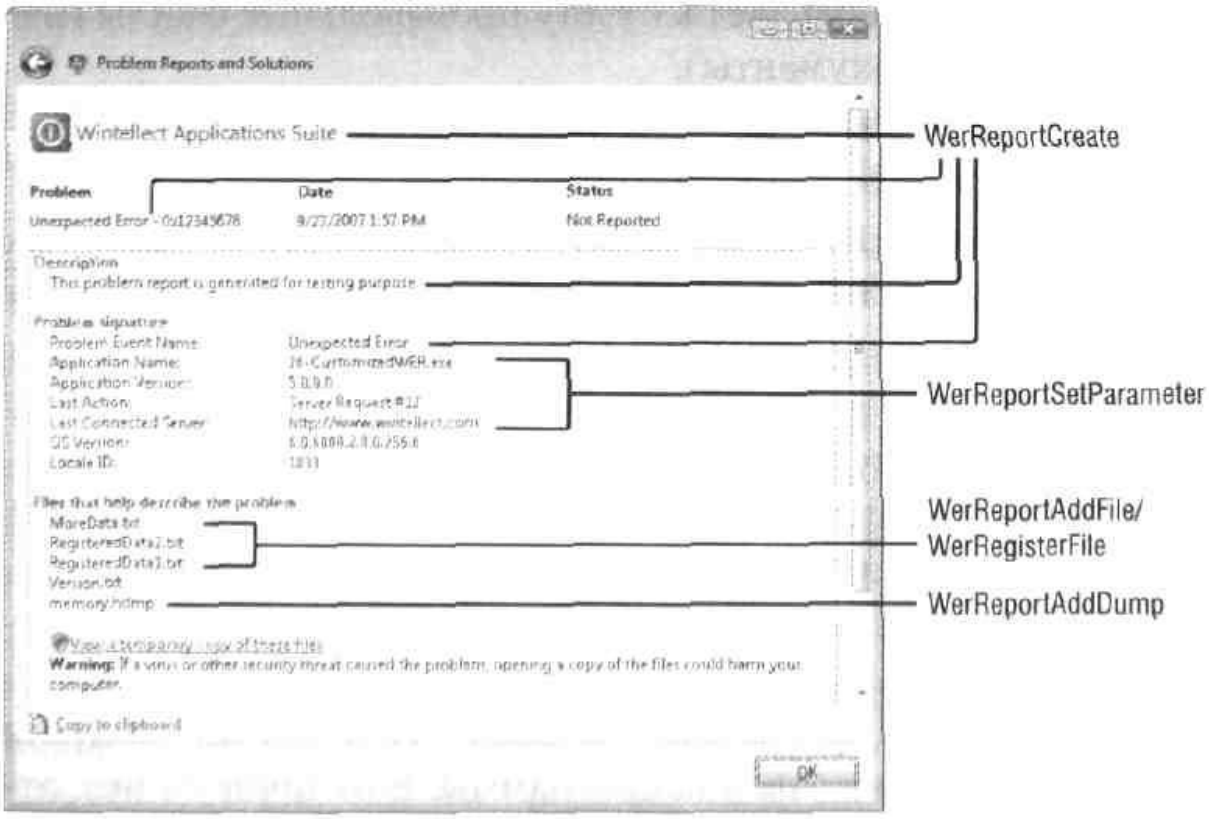


Рис. 26-6. *Нестандартный отчет в консоли WER*

Создание собственного отчета о сбое: функция *WerReportCreate*

Чтобы создать собственный отчет, вызовите функцию *WerReportCreate* и передайте ей сведения об отчете, который требуется создать:

```
HRESULT WerReportCreate(
    PCWSTR pwzEventType,
    WER_REPORT_TYPE repType,
    PWER_REPORT_INFORMATION pReportInformation,
     HREPORT *phReport);
```

Параметр *pwzEventType* — это Unicode-строка, которая представляет собой первый элемент Problem Signature. Заметьте: чтобы ваш отчет был доступен для просмотра на сайте Windows Quality Web (http://WinQual.Microsoft.com), соответствующий тип события должен быть зарегистрирован в Майкрософт (подробнее см. в описании *WerReportCreate* на сайте MSDN по ссылке http://msdn2.microsoft.com/en-us/library/bb513625.aspx).

> **Внимание!** Все *Wer**-функции принимают только Unicode-строки, поэтому в их именах нет суффиксов ***A*** и ***W***.

Параметр *repType* принимает значения, перечисленные в табл. 26-6.

Табл. 26-6. Значения параметра *repType*

Значение	Описание
WerReportNonCritical = 0	Отчет «молча» добавляется в очередь и отправляется в Майкрософт согласно значению параметра Consent
WerReportCritical = 1	Отчет добавляется в локальную очередь с уведомлением пользователя и, при необходимости, завершением приложения
WerReportApplicationCrash = 2	Идентичен *WerReportCritical*, но в уведомлении пользователя выводится понятное имя приложения, а не его исполняемого файла
WerReportApplicationHang = 3	Идентичен *WerReportApplicationCrash*, но используется при зависаниях и взаимных блокировках

Параметр *pReportInformation* указывает на структуру WER_REPORT_INFORMATION, содержащую Unicode-поля, перечисленные в табл. 26-7.

Табл. 26-7. Поля структуры WER_REPORT_INFORMATION

Поле	Описание
wzApplicationName	Продукт отображается в столбце Problem History консоли WER и рядом со значком приложения в описании сбоя
wzApplicationPath	Содержимое не отображается на локальной системе, но доступно на сайте Windows Quality
wzDescription	Содержимое отображается в поле Problem Description
wzFriendlyEventName	Содержимое отображается в поле Event Name секции Problem Signature отчета

Как и все *Wer**-функции, *WerReportCreate* возвращает HRESULT и, если вызов завершился успешно, — описатель отчета (в параметр *phReport*).

Настройка параметров отчета: функция *WerReportSetParameter*

После поля Problem Event Name, но перед OS Version и Locale ID в секции отчета Problem Signature содержится набор пар «ключ-значение», которые задают с помощью следующих функций:

```
HRESULT WerReportSetParameter(
    HREPORT hReport,
    DWORD dwParamID,
    PCWSTR pwzName,
    PCWSTR pwzValue);
```

В параметре *hReport* передается описатель, ранее полученный вызовом *WerReportCreate*. Параметр *dwParamID* задает пару «ключ-значение», которую необходимо определить. Эта функция позволяет определять до 10 пар, которые задаются объявленными в werapi.h макросами WER_P0—WER_P9 (со значениями 0-9). В параметрах *pwzName к pwzValue* вы можете передать нужные вам Unicode-строки.

Заметьте, что при передаче целого числа, которое меньше 0 или больше 9, *WerReportSetParameter* возвращает E_INYALIDARG. Также нельзя пропускать идентификаторы, то есть, устанавливая WERP2, необходимо также установить WERP1 и WER_P0. Порядок установки не важен, но если вы пропустите идентификатор, вызов *WerReportSubmit* закончится неудачей и возвратом HRESULT с значением 0x8008FF05.

По умолчанию, для стандартного отчета WER устанавливает параметры, перечисленные в табл. 26-8.

Табл. 26-8. Параметры по умолчанию отчета о сбое

Номер	Описание
1	Имя сбойного приложения
2	Версия сбойного приложения
3	Временная отметка сборки двоичного файла
4	Имя сбойного модуля
5	Версия сбойного модуля
6	Временная отметка сборки двоичного файла
7	Код возникшего исключения
8	Смещение в модуле, по которому возникло исключение. Рассчитывается путем вычитания из EIP-указателя (или его эквивалента для компьютеров с архитектурой, отличной от x86) сбоя адреса, по которому загружен сбойный модуль

Добавление к отчету мини-дампа: функция *WerReportAddDump*

Добавить к отчету о сбое мини-дамп можно вызовом функции *WerReportAddDump:*

```
HRESULT WerReportAddDump(
    HREPORT hReport,
    HANDLE hProcess,
    HANDLE hThread,
    WER_DUMP_TYPE dumpType,
    PWER_EXCEPTION_INFORMATION pel,
    PWER_DUMP_CUSTOM_OPTIONS pDumpCustomOptions,
    DWORD dwFlags);
```

Параметр *hReport* задает отчет, к которому следует добавить мини-дамп. Параметр *hProcess* задает описатель процесса, дамп которого следует добавить (для этого описателя должны быть назначены права доступа STANDARD_RIGHTS_READ и PROCESS_QUERY_INFORMATION). Обычно этой функции передают описатель, полученный вызовом *GetCurrentProcess* — такие описатели получают все права, имеющиеся у процессов.

Параметр *hThread* задает поток процесса, заданного параметром *hProcess;* функция *WerReportAddDump* проходит по стеку вызовов этого потока. При «посмертной» отладке этот стек используется отладчиком для поиска команды, вызвавшей исключение. Помимо сохраненного стека вызовов, *WerReportAddDump* нужна дополнительная информация, которую вы должны ей передать через параметр *pExceptionParam:*

```
WER_EXCEPTION INFORMATION wei;
weiTbClientPointers = FALSE;              // мы находимся в процессе, для которого
wei.pExceptionPointers = pExceptionlnfo;  // указатель pExceptionInfo действителен
```

В этом примере кода *pExceptionInfo* задает сведения об исключении, возвращаемые функцией *GetExceptionInformation;* ту же самую информацию, как правило, передают фильтру исключений. Далее *&wei* передается параметру *pei* функции *WerReportAddDump*. Тип дампа определяется параметрами *dumpType* и *pDumpCustomOptions* (подробнее см. в соответствующих статьях MSDN).

В параметре *dwFlags* передают 0 или WER_DUMP_NOHEAP_ONQUEUE. Обычно в мини-дамп включают содержимое кучи, но флаг WER_DUMP_NOHEAP_ONQUEUE приказывает *WerReportAddDump* не делать этого. Этот флаг удобен, когда требуется сэкономить место на диске за счет исключения из отчета содержимого кучи.

Добавление к отчету произвольных файлов: функция *WerReportAddFile*

В разделе о настройке отчетов, я рассказал, как добавить произвольные файлы во все отчеты о сбоях заданного процесса. Однако вызовом функции *WerReportAddFile* вы можете добавить к отчету изрядное число файлов (даже больше, чем 512):

```
HRESULT WerReportAddFile(
    HREPORT hReport,
    PCWSTR pwzFilename,
    WER_FILE_TYPE addFileType,
    DWORD dwFileFlags);
```

В параметре *hReport* передают описатель отчета, к которому следует добавить файл, заданный параметром *pwzFilename*. Значения параметра *addFileType* перечислены в табл. 26-9.

Табл. 26-9. Типы добавляемых к отчету файлов

Тип	Описание
WerFileTypeMicrodump = 1	Пользовательский микро-дамп
WerFileTypeMinidump = 2	Пользовательский мини-дамп
WerFileTypeHeapdump = 3	Пользовательский дамп кучи
WerFileTypeUserDocument = 4	Файл, в котором могут быть конфиденциальные пользовательские данные. По умолчанию такие файлы не отправляются в Майкрософт, но в будущем планируется открыть к ним доступ для разработчиков через сайт Windows Quality
WerFileTypeOther = 5	Любой другой файл

Не путайте флаги *WerFileTypeMicrodump, WerFileTypeMinidump* и *WerFileTypeHeapdump*. При отправке отчета на сервер Майкрософт последний вступает в довольно сложный диалог с пользовательским компьютером для согласования списка отправляемых файлов. В некоторых случаях сервер запрашивает файлы дампов. Тогда локальное хранилище использует эти фла-

ги для отбора пользовательских дампов, генерированных приложением без использования функции *WerReportAddDump*. Дополнительные сведения о коммуникационном протоколе, по которому осуществляется отправка отчетов о сбоях, см. на сайте Windows Quality. Параметр *dwFileFlags* выполняет те же функции, что и одноименный параметр *WerRegisterFile* (см. табл. 26-4).

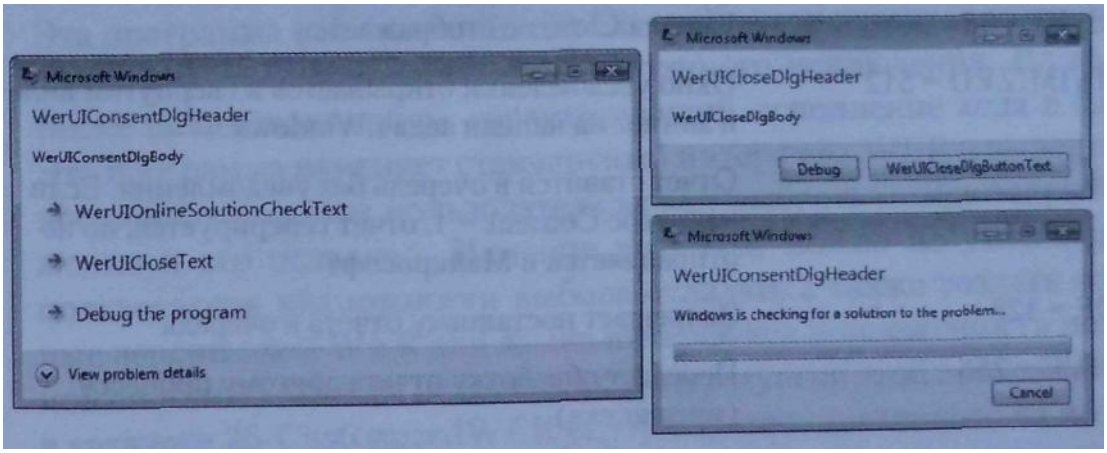
Модификация текста элементов интерфейса: функция WerReportSetUIOption

Чтобы настроить текст, отображаемый в окне запроса разрешения на отправку отчета WER, вызовите следующую функцию:

```
HRESULT WerReportSetUIOption(
    HREPORT hReport,
    WER_REPORT_UI repUITypeID,
    PCWSTR pwzValue);
```

Параметр *hReport* задает отчет, для которого необходимо модифицировать текст пользовательского интерфейса. Параметр *repUITypeID* определяет элемент, текст которого требуется изменить, а параметр *pwzValue* — текст (в формате Unicode), который необходимо вывести в этом элементе интерфейса.

Функцию *WerReportSetUIOption* следует вызывать для каждого из элементов интерфейса, который вы хотите изменить. Заметьте, что текст некоторых надписей и кнопок изменить невозможно. Примеры изменения текстовых полей показаны на экранных снимках ниже. Дополнительные сведения см. в описании функции *WerReportSetUIOption* в документации Platform SDK.



Передача отчета о сбое: функция *WerReportSubmit*

Теперь можно отправить отчет о сбое, вызвав функцию:

```
HRESULT WerReportSubmit(
    HREPORT hReport,
    WER_CONSENT consent,
    DWORD dwFlags,
    PWER_SUBMIT_RESULT pSubmitResult);
```

Параметр *hReport* задает описатель отчета, который требуется передать. Параметр *consent* принимает одно из следующих значений: *WerConsentNotAsked*, *WerConsentApproved* или *WerConsentDenied*. Как сказано выше, будет ли отчет отправлен в Майкрософт, зависит от значения параметра Consent в системном реестре, но окна, которые открывает WER при передаче отчета, определяются одноименным параметром функции *WerReportSubmit*. *E*сли он равен *WerConsentDenied*, отчет не отправляется, а если *WerConsentApproved* — открываются стандартные окна с уведомлениями о генерации отправке отчетов (рис. 25-2 и 25-3). Если параметру *consent* присваивается значение *WerConsentNotAsked*, а одноименный параметр реестра равен 1 (то есть, система всегда должна запрашивать у пользователя разрешение, см. рис. 25-5), открывается окно, показанное на рис. 25-6. При этом пользователь должен сам решить, следует ли отправлять отчет о сбое в Майкрософт для поиска способа его устранения либо выбрать отладку или завершение приложения. Параметр *dwFlags* — это битовая маска, описанная в табл. 26-10.

Табл. 26-10. Параметры передачи отчета о сбое

Параметр	Описание
HONOR_RECOVERY = 1	Разрешает восстановление после критических сбоев (см. ниже)
HONOR_RESTART = 2	Разрешает перезапуск приложения после критических сбоев (см. ниже)
SHOW_DEBUG = 8	Если этот флаг не установлен, кнопка Debug не отображается
NO_CLOSE_UI = 64	Кнопка Close не отображается
START_MINIMIZED = 512	Окно уведомления открывается в свернутом виде и мигает на панели задач Windows
QUEUE = 4	Отчет ставится в очередь без уведомления. Если в реестре Consent = 1, отчет генерируется, но не отправляется в Майкрософт
NO_QUEUE = 128	Запрещает постановку отчета в очередь
NO_ARCHIVE = 256	Передает обработку отчета другому процессу (wermgr.exe)
OUTOFPROCESS_ASYNC = 1024	Передает обработку отчета другому процессу (wermgr.exe), после чего *WerReportSubmit* немедленно возвращает управление, не дожидаясь завершения обработки
ADD REGISTERED DATA = 16	Добавляет к отчету зарегистрированные данные. Заметьте, что если это флаг установлен вместе с флагом, передающим обработку отчета другому процессу, добавленные с помощью функции *WerRegisterFile* файлы попадут в отчет дважды. Эта ошибка будет устранена в будущих версиях Windows

Об успешном или неудачном завершении вызова *WerReportSubmit* сообщает возвращаемое ей значение HRESULT. Однако, собственно, результат возвращается в параметре *pSubmitResult* как ссылка на переменную WER_SUBMIT_RESULT (подробнее об этом см. в документации MSDN). Параметр *pSubmitResult* бесполезен, только когда в *dwFlags* установлен флаг WER_SUBMIT_OUTOFPROCESS_ASYNC. В этом случае *WerReportSubmit* возвращает управление до завершения обработки запросов. Ясно, что использовать этот флаг следует с осторожностью и ни в коем случае не применять его в контексте необработанного исключения: процесс будет закрыт прежде, чем завершится создание и отправка отчета. Этот флаг предназначен для генерации отчетов в случаях, когда в программе нет необработанных исключений, и ее исполнение нежелательно задерживать до завершения создания отчета. Флаг WER_SUBMIT_OUTOFPROCESS_ASYNC обычно применяется для процессов, отслеживающих возникновение сбоев в других процессах. Например, Service Control Manager (SCM) в Windows использует этот флаг для уведомления о зависании процессов служб.

Закрытие отчета о сбое: функция WerReportCloseHandle

После отправки отчета не забудьте передать его описатель функции *WerReportCloseHandle*, чтобы освободить связанные с ним внутренние структуры данных:

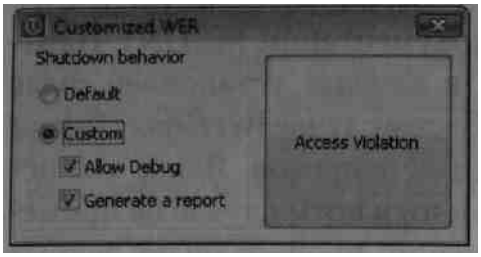
```
HRESULT WerReportCloseHandle(HREPORT hReportHandle);
```

Программа-пример Customized WER

Эта программа (26-CustomizedWER.exe) демонстрирует создание отчетов о сбоях при возникновении необработанного исключения. На ее примере я также показываю прием, гарантирующий исполнение кода в блоке *finally*. Эта программа изменяет стандартный интерфейс WER и выводит собственное окно, в котором пользователь может выбрать между закрытием приложения и его отладкой. Изменив этот код, можно по умолчанию лишить пользователя возможности выбора отладки, а также создать локализованные версии окон WER для языка интерфейса приложения (а не операционной системы). Файлы исходного кода и ресурсов этой программы находятся в каталоге 26-CustomizedWER внутри архива, доступного на веб-сайте поддержки этой книги.

> **Примечание.** Если ваше приложение должно работать в версиях Windows, предшествующих Vista, используйте функцию ReportFault. У этой функции намного меньше возможностей, чем у вышеописанных *Wer**-функций. В приложениях для Windows Vista использовать ReportFault не рекомендуется.

После запуска программы Customized WER открывается следующее окно:



По щелчку кнопки Access Violation вызывается следующая функция:

```
void TriggerException() {
    // провоцируем исключение,  с которым связан блок finally,
    // код в котором исполняется только при глобальной раскрутке
    __try {
        TCHAR* p = NULL;
        *p = TEXT('a');
    }
    __finally {
        MessageBox(NULL, TEXT("Finally block is executed"),  NULL,  MB_OK);
    }
}
```

Далее исполняется функция *CustomUnhandledExceptionFilter* — фильтр исключений блока *try/except*, защищающего входную функцию приложения:

```
int APIENTRY _tWinMain(HINSTANCE hInstExe,   HINSTANCE,   LPTSTR,   int)  {

    int iReturn = 0;

    // отключаем автоматическую отладку по запросу,
    // которая могла быть включена ранее вызовом
    // CustomUnhandledExceptionFilter
    EnableAutomaticJITDebug(FALSE);

    // Защищаем код собственным фильтром исключений
    __try {
        DialogBox(hInstExe, MAKEINTRESOURCE(IDD_MAINDLG), NULL, Dlg_Proc);
    }
    __except(CustomUnhandledExceptionFilter(GetExceptionInformation())) {
        MessageBox(NULL, TEXT("Bye bye"), NULL, MB_OK);
        Exit Process( GetExceptionCode());
    }

    return(iReturn);
}
```

Работа фильтра исключений управляется через главное окно приложения:

```
static BOOL s_bFirstTime = TRUE;

LONG WINAPI CustomUnhandledExceptionFilter(
    struct _EXCEPTION_POINTERS* pExceptionlnfo) {

    // После подключения отладчика и завершения отладки
    // исполнение возобновляется с этого места.
    // Если это так,  приложение «молча» завершается
    if (s_bFirstTime)
        s_bFirstTime = FALSE;
    else
        ExitProcess(pExceptionInfo->ExceptionRecord->ExceptionCode);

    // проверяем параметр завершения
    if (!s.bCustom)
        // позволяем Windows самой обработать исключение
        return(UnhandledExceptionFilter(pExceptionInfo));

    // по умолчанию разрешаем глобальную раскрутку
    LONG lReturn = EXCEPTION_EXECUTE_HANDLER;

    // позволим пользователю выбрать между отладкой и закрытием приложений,
    // если отладка по запросу не отключена в главном окне
    int iChoice = IDCANCEL;
    if (s_bAllowJITDebug) {
        iChoice = MessageBox(NULL,
            TEXT("Click RETRY if you want to debug\nClick CANCEL to quit"),
            TEXT("The application must stop"), MB_RETRYCANCEL | MB_ICONHAND);
    }

    if (iChoice == IDRETRY) {
        // включаем автоматической отладки для этого приложения
        EnableAutomaticJITDebug(TRUE);

        // просим Windows подключать по запросу отладчик по умолчанию
        lReturn = EXCEPTION_CONTINUE_SEARCH;
    } else {
        // приложение будет завершено
        lReturn = EXCEPTION_EXECUTE_HANDLER;

        // но сначала проверим, не требуется ли сначала создать отчет о сбое
        if (s_bGenerateReport)
            GenerateWerReport(pExceptionInfo);
    }

    return(lReturn);
}
```

Если выбран переключатель Default, возвращается вывод функции *UnhandledExceptionFilter* (обработчика исключений по умолчанию в Windows). В этом случае открываются обычные окна и создается отчет о сбое, как описано в предыдущих разделах.

Если же установлен переключатель Custom, работой фильтра *CustomUnhandledExceptionFilter* управляют дополнительные параметры. Если выбрать Allow Debug, откроется такое диалоговое окно:

Если пользователь щелкнет Retry, нужно попробовать обхитрить Windows, чтобы начать отладку по запросу, поскольку возврата EXCEPTION_CONTINUESEARCH недостаточно: он просто означает, что исключение не обработано, и WER снова открывает стандартные окна, чтобы пользователь выбрал, что делать. Роль функции *EnableAutomaticJITDebug* заключается в том, чтобы сообщить WER, что выбранное подключение отладчика уже выполнено.

```
void EnableAutomaticJITDebug(BOOL bAutomaticDebug) {

   // при необходимости создаем раздел реестра
   const LPCTSTR szKeyName = TEXT("Software\\Microsoft\\Windows\\" +
      TEXT(Windows Error ReportingWDebugApplications");
   HKEY hKey = NULL;
   DWORD dwDisposition = 0;
   LSTATUS lResult = ERR0R_SUCCESS;
   lResult = RegCreateKeyEx(HKEY_CURRENT_USER,  szKeyName,  0,  NULL,
      REG_0PTI0N_NON_V0LATILE, KEY_WRITE, NULL,  &Mey, &dwDisposition);
   if (lResult != ERROR_SUCCESS) {
      MessageBox(NULL, TEXT("RegCreateKeyEx failed"),
         TEXT("EnableAutomaticJITDebug"),  MB_OK | MB_CONHAND);
      return;
   }

   // присваиваем правильное значение параметру реестра
   DWORD dwValue = bAutomaticDebug ? 1 : 0;
   TCHAR szFullpathName[MAX_PATH];
   GetModuleFileName(NULL, szFullpathName, _countof(szFullpathName));
   LPTSTR pszExeName = _tcsrchr(szFullpathName, TEXT('W'));
   if (pszExeName != NULL) {
      // пропускаем '\'
      pszExeName++;
```

```
        // устанавливаем значение
        lResult = RegSetValueEx(hKey, pszExeName, 0, REG_DWORD,
            (const BYTE*)&dwValue, sizeof(dwValue));
        if (lResult != ERROR_SUCCESS) {
            MessageBox(NULL, TEXT("RegSetValueExfailed"),
               TEXTCEnableAutomaticJITDebug"), MB_OK | MB_ICOMHAND);
            return;
        }
    }
}
```

Ничего сложного в этом коде нет, он использует параметры реестра, описанные в разделе «Отладка по запросу» предыдущей главы. Заметьте, что при запуске приложения функция *EnableAutomaticJITDebug* вызывается с параметром FALSE для сброса параметра реестра в 0.

Если в главном окне программы выбран параметр Allow Debug, процесс завершается без уведомлений, как при щелчке кнопки Cancel: возвращается значение EXCEPTION_EXECUTE_HANDLER и глобальный блок *except* вызывает *ExitProcess*. Однако перед возвратом EXCEPTION_EXECUTE_HANDLER программа проверяет, не выбран ли параметр Generate A Report. Если это так, генерируется нестандартный отчет WER с помощью функции *GenerateWerReport*, в свою очередь вызывающей *WerReport**-функции, описанное в этой главе выше:

```
LONG GenerateWerReport(struct _EXCEPTION_POINTERS* pExceptionInfo) {

   // Default return value
   LONG lResult = EXCEPTION_CONTINUE_SEARCH;

   // Avoid stack problem because wri is a big structure
   static WER_REPORT_INFORMATION wri = { sizeof(wri) };

   // Set the report details
   StringCchCopyW(wri.wzFriendlyEventName, _countof(wri.wzFriendlyEventName),
       L"Unexpected Error - 0x12345678");
   StringCchCopyW(wri.wzApplicationName, _countof(wri.wzApplicationName),
      L"Wintellect Applications Suite");
   GetModuleFileNameW(NULL, (WCHAR*)&(wri.wzApplicationPath),
      _countof(wri.wzApplicationPath));
   StringCchCopyW(wri.wzDescription, _countof(wri.wzDescription),
      L"This problem report is generated for testing purpose");

   HREPORT hReport = NULL;

   // Create a report and set additional information
   __try {                          // V-- instead of the default APPCRASH_EVENT
      HRESULT hr = WerReportCreate(L"Unexpected Error",
         WerReportApplicationCrash, &wri, &hReport);
```

```
if (FAILED(hr)) {
   MessageBox(NULL, TEXT("WerReportCreate failed"),
      TEXT("GenerateWerReport"), MB_OK | MB_ICONHAND);
   return(EXCEPTION_CONTINUE_SEARCH);
}
if (hReport == NULL) {
   MessageBox(NULL, TEXT("WerReportCreate failed"),
      TEXT("GenerateWerReport"), MB_OK | MB_ICONHAND);
   return(EXCEPTION_CONTINUE_SEARCH);
}

// Set more details important to help fix the problem
WerReportSetParameter(hReport, WER_P0,
   L"Application Name", L"26-CustomizedWER.exe");
WerReportSetParameter(hReport, WER_P1,
   L"Application Version", L"5.0.0.0");
WerReportSetParameter(hReport, WER_P2,
   L"Last Action", L"Server Request #12");
WerReportSetParameter(hReport, WER_P3,
   L"Last Connected Server", L"http://www.wintellect.com");

// Add a dump file corresponding to the exception information
WER_EXCEPTION_INFORMATION wei;
wei.bClientPointers = FALSE;              // We are in the process where
wei.pExceptionPointers = pExceptionInfo;  // pExceptionInfo is valid
hr = WerReportAddDump(
   hReport, GetCurrentProcess(), GetCurrentThread(),
   WerDumpTypeHeapDump, &wei, NULL, 0);
if (FAILED(hr)) {
   MessageBox(NULL, TEXT("WerReportAddDump failed"),
      TEXT("GenerateWerReport"), MB_OK | MB_ICONHAND);
   return(EXCEPTION_CONTINUE_SEARCH);
}

// Let memory blocks be visible from a mini-dump
s_moreInfo1.dwCode = 0x1;
s_moreInfo1.dwValue = 0xDEADBEEF;
s_moreInfo2.dwCode = 0x2;
s_moreInfo2.dwValue = 0x0BADBEEF;
hr = WerRegisterMemoryBlock(&s_moreInfo1, sizeof(s_moreInfo1));
if (hr != S_OK) { // Don't want S_FALSE
   MessageBox(NULL, TEXT("First WerRegisterMemoryBlock failed"),
      TEXT("GenerateWerReport"), MB_OK | MB_ICONHAND);
   return(EXCEPTION_CONTINUE_SEARCH);
}
hr = WerRegisterMemoryBlock(&s_moreInfo2, sizeof(s_moreInfo2));
```

```
      if (hr != S_OK) { // Don't want S_FALSE
         MessageBox(NULL, TEXT("Second WerRegisterMemoryBlock failed"),
            TEXT("GenerateWerReport"), MB_OK | MB_ICONHAND);
         return(EXCEPTION_CONTINUE_SEARCH);
      }

      // Add more files to this particular report
      wchar_t wszFilename[] = L"MoreData.txt";
      char textData[] = "Contains more information about the execution \r\n\
context when the problem occurred. The goal is to \r\n\
help figure out the root cause of the issue.";
      // Note that error checking is removed for readability
      HANDLE hFile = CreateFileW(wszFilename, GENERIC_WRITE, 0, NULL,
         CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
      DWORD dwByteWritten = 0;
      WriteFile(hFile, (BYTE*)textData, sizeof(textData), &dwByteWritten,
         NULL);
      CloseHandle(hFile);
      hr = WerReportAddFile(hReport, wszFilename, WerFileTypeOther,
         WER_FILE_ANONYMOUS_DATA);
      if (FAILED(hr)) {
         MessageBox(NULL, TEXT("WerReportAddFile failed"),
            TEXT("GenerateWerReport"), MB_OK | MB_ICONHAND);
         return(EXCEPTION_CONTINUE_SEARCH);
      }

      // It is also possible to use WerRegisterFile
      char textRegisteredData[] = "Contains more information about the \
execution \r\ncontext when the problem occurred. The goal is to \r\n\
help figure out the root cause of the issue.";
      // Note that error checking is removed for readability
      hFile = CreateFileW(L"RegisteredData1.txt", GENERIC_WRITE, 0, NULL,
         CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
      dwByteWritten = 0;
      WriteFile(hFile, (BYTE*)textRegisteredData, sizeof(textRegisteredData),
         &dwByteWritten, NULL);
      CloseHandle(hFile);
      hr = WerRegisterFile(L"RegisteredData1.txt", WerRegFileTypeOther,
         WER_FILE_ANONYMOUS_DATA);
      if (FAILED(hr)) {
         MessageBox(NULL, TEXT("First WerRegisterFile failed"),
            TEXT("GenerateWerReport"), MB_OK | MB_ICONHAND);
         return(EXCEPTION_CONTINUE_SEARCH);
      }
      hFile = CreateFileW(L"RegisteredData2.txt", GENERIC_WRITE, 0, NULL,
         CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
```

```
      dwByteWritten = 0;
      WriteFile(hFile, (BYTE*)textRegisteredData, sizeof(textRegisteredData),
         &dwByteWritten, NULL);
      CloseHandle(hFile);
      hr = WerRegisterFile(L"RegisteredData2.txt", WerRegFileTypeOther,
         WER_FILE_DELETE_WHEN_DONE);   // File is deleted after WerReportSubmit
      if (FAILED(hr)) {
         MessageBox(NULL, TEXT("Second WerRegisterFile failed"),
            TEXT("GenerateWerReport"), MB_OK | MB_ICONHAND);
         return(EXCEPTION_CONTINUE_SEARCH);
      }

      // Submit the report
      WER_SUBMIT_RESULT wsr;
      DWORD submitOptions =
         WER_SUBMIT_QUEUE |
         WER_SUBMIT_OUTOFPROCESS |
         WER_SUBMIT_NO_CLOSE_UI;  // Don't show any UI
      hr = WerReportSubmit(hReport, WerConsentApproved, submitOptions, &wsr);
      if (FAILED(hr)) {
         MessageBox(NULL, TEXT("WerReportSubmit failed"),
            TEXT("GenerateWerReport"), MB_OK | MB_ICONHAND);
         return(EXCEPTION_CONTINUE_SEARCH);
      }

      // The submission was successful, but we might need to check the result
      switch(wsr)
      {
         case WerReportQueued:
         case WerReportUploaded: // To exit the process
            lResult = EXCEPTION_EXECUTE_HANDLER;
            break;

         case WerReportDebug: // To end up in the debugger
            lResult = EXCEPTION_CONTINUE_SEARCH;
            break;

         default: // Let the OS handle the exception
            lResult = EXCEPTION_CONTINUE_SEARCH;
            break;
      }

      // In our case, we always exit the process after the report generation
      lResult = EXCEPTION_EXECUTE_HANDLER;
   }
   __finally {
```

```
      // Don't forget to close the report handle
      if (hReport != NULL) {
         WerReportCloseHandle(hReport);
         hReport = NULL;
      }
   }

   return(lResult);
}
```

Автоматический перезапуск и восстановление приложений

При возникновении в приложении критического сбоя, WER может автоматически закрыть, а затем перезапустить его. Поддержка этой функции — непременный атрибут стандартных программ Windows Vista (Explorer, Internet Explorer, RegEdit, игр и пр.). Более того, WER позволяет сохранить важные данные перед завершением приложения.

Автоматический перезапуск приложения

Приложение, поддерживающее автоматический перезапуск, должно зарегистрироваться у WER вызовом следующей функции:

```
HRESULT RegisterApplicationRestart(
    PCWSTR pwzCommandline,
    DWORD dwFlags);
```

Ее параметр *pwzCommandLine* — это командная строка (в формате Unicode), которую WER должна использовать для перезапуска приложения. Если ваша программа не использует специальных аргументов командной строки, уведомляющих ее о перезапуске, передавайте в этом параметре NULL. Если передать 0 в параметре *dwFlags,* приложение будет перезапускаться каждый раз, когда WER обнаружит в нем критический сбой. Тонкую настройку условий, требующих перезапуска приложения, можно выполнить с помощью комбинации значений, перечисленных в табл. 26-11.

Табл. 26-11. Флаги для настройки перезапуска приложений

Флаг	Описание
RESTART_NO_CRASH = 1	Запрещает перезапуск приложения после его краха
RESTART_NO_HANG = 2	Запрещает перезапуск приложения после его зависания
RESTART_NO_PATCH = 4	Запрещает перезапуск приложения после установки обновлений
RESTART_NO_REBOOT = 8	Запрещает перезапуск приложения после перезагрузки системы вследствие установки обновления

Последние два флага выглядят странными в контексте обработки исключений, но поддержка перезапуска приложений входит в более общий API под названием Restart Manager (подробнее о нем см. в статье MSDN по ссылке http://msdn2.microsoft.com/en-us/library/aa373651.aspx).

После вызова функции *RegisterApplicationRestart* при возникновении в процесс критического сбоя, который обрабатывает WER, приложение перезапускается и открывается окно, показанное на рис. 26-7.

Рис. 26-7. *Уведомление пользователя о перезапуске приложения*

Чтобы предотвратить многократный перезапуск сбойного приложения, WER, прежде чем перезапустить процесс, проверяет, проработал ли он до перезапуска хотя бы 60 секунд.

> **Примечание.** Чтобы запретить WER перезапуск приложения, вызовите следующую функцию:
>
> ```
> HRESULT UnregisterApplicationRestart();
> ```

Поддержка восстановления приложений

Процесс может зарегистрировать функцию обратного вызова, которую WER может вызвать при аварийном завершении этого процесса. Эта функция может сохранять нужные данные или сведения о состоянии. Для регистрации функции обратного вызова используют следующую функцию:

```
HRESULT RegisterApplicatiortRecoveryCallback(
    APPLICATION_RECOVERY_CALLBACK pfnRecoveryCallback,
    PVOID pvParameter,
    DWORD dwPinglnterval,
    DWORD dwFlags);                          // зарезервирован;  должен быть равен 0
```

Параметр *pfnRecoveryCallback* должен указывать на функцию со следующей сигнатурой:

```
DWORD WINAPI ApplicationRecoveryCallback(PVOID pvParameter);
```

WER осуществляет обратный вызов с параметром *pvParameter*, который вы передали при вызове *RegisterApplicationRecoveryCallback*. При этом WER открывает окно, показанное на рис. 26-8.

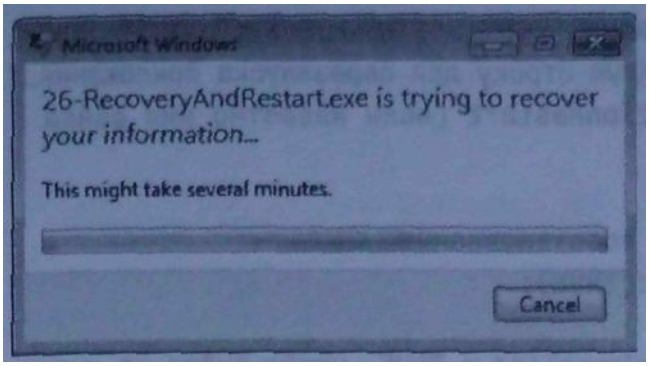


Рис. 26-8. *Уведомление пользователя о подготовке к восстановлению приложения*

Функция *pfnRecoveryCallback* должна уведомлять WER о своем исполнении, вызывая функцию *ApplicationRecoverylnProgress* по истечении интервала, заданного параметром *dwPinglnterval* (в мс). Если своевременный вызов *ApplicationRecoverylnProgress* не последует, WER закрывает процесс. Функция *ApplicationRecoverylnProgress* принимает указатель на BOOL-значение, уведомляющее о щелчке кнопки Cancel пользователем (см. рис. 26-8). Завершаясь, выполняющая восстановление функция должна вызвать *ApplicationRecoveryFinished*, чтобы уведомить WER об успехе или неудаче восстановления.

Вот пример функции обратного вызова, выполняющей восстановление приложения:

```
DWORD WINAPI ApplicationRecoveryCallback(PVOID pvParameter) {

    DWORD dwReturn = 0;

    BOOL bCancelled = FALSE;
    while (!bCancelled) {

        // индикатор хода операции
        ApplicationRecoveryInProgress(&bCancelled);

        // проверяем, не отменил ли пользователь операцию
        if (bCancelled) {
            // пользователь щелкнул кнопку Cancel

            // уведомляем о неудаче восстановления
            ApplicationRecoveryFlnished(FALSE);
        } else {
            // сохраняем порции данных, необходимые для восстановления

            if (MoreInformationToSave()) {
                // запись порции данных занимает меньше времени, чем задержка,
                // установленная параметром dwPingInterval функции
                // RegisterApplicationRecoveryCallback
```

```
            } else { // все данные сохранены
                // Еще можно обновить командную строку для перезапуска приложения,
                // вызывав RegisterApplicationRestart (если известно имя файла
                // для восстановления).

                // уведомляем о завершении восстановления
                ApplicationRecoveryFinished(TRUE);

                // закончив, устанавливаем bCancelled в TRUE, чтобы выйти из цикла
                bCancelled = TRUE;
            }
        }
    }

    return(dwReturn);
}
```

Помните, что во время исполнения обратного вызова процесс может быть уже поврежден, поэтому для таких функций обратного вызова действую те же ограничения, что и на фильтры исключений.

Оглавление

ПРИЛОЖЕНИЕ А

Среда разработки

При сборке программ-примеров вам придется иметь дело с ключами компилятора и компоновщика. Чтобы не засорять ими тексты программ, я выделил их почти все в один заголовочный файл CmnHdr.h, включаемый в исходные файлы.

К сожалению, некоторые параметры разместить там нельзя, и поэтому пришлось вносить кое-какие изменения в свойства проекта каждой программы. Во всех проектах я открывал диалоговое окно Properties и изменял следующие настройки Configuration Properties:

- на вкладке General в поле Output Directory я указывал конкретный каталог, чтобы в него помещались все EXE- и DLL-файлы;
- на вкладке C/C++, Code Generation в списке Run-Time Library я выбирал Multi-Threaded DLL;
- на вкладке C/C++ в списке Detect 64-Bit Portability я устанавливал значение Yes (/Wp64).

Вот и все. Лишь эти три параметра требовали модификации вручную, для остальных параметров меня устраивали значения, предлагаемые по умолчанию. Кстати, эти изменения я вносил как в Debug-, так и в Release-версию каждого проекта (т. е. в его отладочную и конечную версию). Остальные настройки компилятора и компоновщика мне удалось разместить в исходном коде, и они вступают в силу, как только вы включаете в свой проект любой из моих модулей исходного кода.

Заголовочный файл CmnHdr.h

Все программы-примеры в этой книге включают файл CmnHdr.h перед остальными заголовочными файлами. Я написал его, чтобы хоть чуть-чуть облегчить себе жизнь. Он содержит макросы, директивы компоновщика и прочий код, общий для всех программ. Иногда, чтобы что-то попробовать, мне нужно было всего лишь модифицировать этот файл и собрать все программы-примеры заново. Файл CmnHdr.h находится в корневом каталоге архива, доступного на сайте поддержке этой книги.

Далее я расскажу обо всех разделах заголовочного файла CmnHdr.li и объясню, для чего предназначен каждый из них, а также как его изменить и зачем.

Раздел Windows Version Build Option

Поскольку часть моих программ обращается к новым функциям, появившимся только в Windows Vista, в этом разделе определяются идентификаторы _WIN32_WINNT и WINVER:

```
// = 0x0600 for VISTA level from sdkddkver.h
#define _WIN32_WINNT _WIN32_WINNT_L0NGH0RN
#deflne WINVER       _WIN32_WINNT_L0NGH0RN
```

Мне пришлось это сделать, так как прототипы новых функций в заголовочных файлах Windows задаются так:

```
#if (_WIN_WINNT >= 0x0600)
…

HANDLE
WINAPI
CreateMutexExW(
    LPSECURITY_ATTRIBUTES lpMutexAttributes,
    LPCWSTR lpName,
    DWORD dwFlags,
    DWORD dwDesiredAccess
    );
…

#endif /* _WIN32_WINNT >= 0x0600 */
```

Пока вы сами не определите _WIN32_WINNT (до включения файла Windows.h), прототипы новых функций не будут объявлены, и компилятор, встретив попытки вызова этих функций, сообщит об ошибках. Майкрософт специально защитила эти функции идентификатором _WIN32_WINNT, чтобы разработчики, ориентирующиеся на разные версии Windows, случайно не использовали функции, существующие пока только в Windows Vista.

Раздел Unicode Build Option

Если вы хотите создать Unicode-версию программы для процессоров x86, раскомментируйте всего одну строку, в которой определяется макрос UNICODE, и перекомпилируйте программу. Определяя макрос UNICODE в файле CmnHdr.h, я упрощаю себе управление сборкой программ-примеров. Подробнее о Unicode см. главу 2.

Раздел Windows Definitions и диагностика уровня 4

Разрабатывая программы, я всегда стараюсь, чтобы компилятор мог транслировать их, не выдавая предупреждений и сообщений об ошибках. Кроме того, я предпочитаю устанавливать при компиляции наивысший уровень диагностики. В этом случае компилятор делает за меня максимум работы и проверяет в коде даже самые незначительные мелочи. Для компиляторов Microsoft C/C++ это означает, что я компилировал все примеры с использованием диагностики уровня 4.

Увы, отдел операционных систем в Майкрософт не разделяет моих сантиментов насчет компиляции с применением диагностики уровня 4. В итоге, когда я установил такой уровень, компилятор выдал предупреждения по множеству строк из заголовочных файлов самой Windows. К счастью, они не свидетельствуют о каких-то проблемах в коде — большинство из них генерируется из-за нетипичного употребления конструкций языка C, в которых используются расширения, реализованные практически всеми поставщиками Windows-совместимых компиляторов.

Поэтому в данном разделе CmnHdr.h я указываю уровень диагностики 3, включаю стандартный заголовочный файл Windows.h, а потом задаю уровень диагностики 4. На этом уровне компилятор часто выдает предупреждения по таким ерундовым поводам, что приходится вставлять директиву *#pragma warning для* подавления некоторых предупреждений.

Вспомогательный макрос Pragma Message

Мне всегда хочется, чтобы какая-то часть программы, которую я еще пишу, сразу же начала работать, а доводку обычно откладываю на потом. Раньше, когда мне надо было напомнить себе, дескать, этот участок кода еще потребует внимания, я включал в код строки типа:

```
#pragma message("Fix this later")
```

Встречая такую строку, компилятор выдавал мне сообщение, напоминающее, что в этом месте работа еще не закончена. Увы, толку от него не очень много. Было бы куда полезнее, если бы он сообщал имя исходного файла и номер строки, где стоит эта директива. Тогда я не только узнавал бы, что какая-то работа не доделана, но и мог бы тут же найти нужный участок кода.

А для этого не обойтись без набора макросов. В итоге я создал макрос chMSG, вызываемый так:

```
#pragma chMSG(Fix this later)
```

Дойдя до соответствующей строки, компилятор выдает что-то вроде:

```
C:\CD\CominonFiles\CmnHdr.h(82): Fix this later
```

Теперь, работая в Microsoft Visual Studio, можно дважды щелкнуть это сообщение в окне вывода, и тогда открывается соответствующий файл, а курсор ввода переходит на нужную строку.

И еще одна удобная мелочь — макрос chMSG не требует заключать текстовую строку в кавычки.

Макрос chINRANGE

Это весьма полезный макрос, которым я часто пользуюсь в своих программах. Он проверяет, укладывается ли какое-то значение в заданный диапазон.

Макрос chBEGINTHREADEX

Все многопоточные программы из этой книги используют вместо Windows-функции *CreateThread* функцию *_beginthreadex* из библиотеки Microsoft C/ C++. Это связано с тем, что *_beginthreadex* подготавливает новый поток так, чтобы он мог вызывать библиотечные функции, а при его завершении гарантирует уничтожение информации, используемой библиотекой в данном потоке (подробнее см. главу 6). К сожалению, прототип этой функции имеет вид:

```
unsigned long __cdecl _beginthreadex(
    void *,
    unsigned,
    unsigned (__stdcall *)(void *),
    void *,
    unsigned,
    unsigned *);
```

Хотя значения параметров *_beginthreadex* идентичны значениям параметров функции *CreateThread,* их типы не совпадают. Вот прототип *CreateThread:*

```
typedef DWORD (WINAPI *PTHREAD_START_ROUTINE)(PVOID pvParam);

HANDLE CreateThread(
    PSECURITY_ATTRIBUTES psa,
    SIZE_T cbStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD dwCreateFlags,
    PDWORD pdwThreadId);
```

Типы данных Windows-функции и ее библиотечного аналога так отличаются потому, что группа разработчиков библиотеки Microsoft C/C++ не пожелала зависеть от группы разработчиков операционных систем. Похвальное решение, но оно затруднило использование функции *_beginthreadex*.

В том, как Майкрософт объявила прототип *_beginthreadex,* есть целых две проблемы. Во-первых, некоторые типы данных, используемые этой функцией, не совпадают с базовыми типами, примененными в *CreateThread.* Например, в Windows API тип данных DWORD определен как:

```
typedef unsigned long DWORD;
```

К этому типу данных относятся параметры *cbStack* и *fdwCreate* функции *CreateThread.* Но в прототипе *_begjtnthreadex* эти параметры объявлены как *unsigned,* что на деле означает *unsigned int.* Компилятор считает типы *unsigned int* и *unsigned long* разными и выдает предупреждение. Поскольку *_beginthreadex* не является частью стандартной библиотеки C/C++ и существует лишь как альтернатива Windows-функции *CreateThread, я* полагаю, что Майкрософт следовало бы объявить прототип *_beginthreadex* так, чтобы предупреждения не генерировались:

```
unsigned long __cdecl _beginthreadex(
    void *psa,
    unsigned long cbStackSize,
    unsigned (__stdcall *) (void *pvParam),
    void *pvParam,
    unsigned long dwCreateFlags,
    unsigned long *pdwThreadId);
```

Вторая проблема — продолжение первой. Функция *_beginthreadex* возвращает значение типа *unsigned long* — описатель только что созданного потока. Обычно программа сохраняет это значение в переменной типа HANDLE:

```
HANDLE hThread = _beginthreadex(…);
```

Эта строка заставит компилятор выдать другое предупреждение. Чтобы избежать этого предупреждения, надо переписать строку с явным преобразованием типов:

```
HANDLE hThread = (HANDLE) _beginthreadex(…);
```

Это, конечно, неудобно. Чтобы компилятор ко мне не приставал, я определил в файле CmnHdr.h макрос chBEGINTHREADEX, который сам делает эти преобразования:

```
typedef unsigned (_stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStackSize, pfnStartAddr,  \
    pvParam, dwCreateFlags, pdwThreadId)                 \
        ((HANDLE)_beginthreadex(                         \
        (void *)        (psa),                           \
        (unsigned)      (cbStackSize),                   \
        (PTHREAD_START) (pfnStartAddr),                  \
        (void *)        (pvParam),                       \
```

```
(unsigned)      (dwCreateFlags),                \
(unsigned *)    (pdwThreadId)))
```

Моя реализация DebugBreak для платформы x86

Иногда мне нужно прервать код в какой-то точке, даже если процесс не выполняется под управлением отладчика. Для этого поток должен вызвать функцию *DebugBreak*, которая содержится в Kernel32.dll. Она позволяет подключить к процессу отладчик. Как только отладчик подключается к процессу, регистр указателя команд позиционируется на машинную команду, вызвавшую прерывание (останов). Эта команда находится в функции *DebugBreak* из Kernel32.dll, и, чтобы увидеть свой исходный код, я должен выйти из *DebugBreak*, используя режим пошагового выполнения кода.

На процессорной платформе *x86* выполнение кода прерывается инструкцией «int 3». Поэтому я переопределил *DebugBreak* для платформы я86, включив эту ассемблерную инструкцию и сделав функцию подставляемой в код. При выполнении моей *DebugBreak* перехода в Kernel32.dll не происходит, прерывание возникает непосредственно в моем коде, и регистр указателя команд позиционируется на следующий оператор языка C/C++. Так удобнее.

Определение кодов программных исключений

Работая с программными исключениями, вы должны определить собственные 32-битные коды исключений. Эти коды имеют специфический формат, о котором я рассказывал в главе 24. Чтобы упростить определение кодов программных исключений, я использую макрос MAKESOFTWAREEXCEPTTON.

Макрос chMB

Он просто выводит окно, в заголовке которого показывает полное (вместе с путем) имя исполняемого файла вызывающего процесса.

Макросы chASSERT и chVERIFY

Чтобы упростить выявление ошибок при разработке программ, я часто вкраплял в код макросы chASSERT. Этот макрос проверяет, истинно ли выражение x, и, если нет, выводит окно с именем файла, строкой и выражением, вычисление которого закончилось неудачно. В готовой программе макрос chASSERT раскрывается в пустое определение. Макрос chVERIFY практически идентичен предыдущему за исключением того, что выражения проверяются не только при отладке, но и в готовой программе.

Макрос chHANDLE_DLGMSG

Используя распаковщики сообщений (message crackers) при работе с диалоговыми окнами, нельзя обращаться к макросу HANDLE_MSG из заголовочного файла WindowsX.h. Дело в том, что он не возвращает TRUE или FALSE в зависимости от того, обработано сообщение процедурой диалогового окна или нет. Чтобы устранить эту проблему, я и написал макрос chHANDLE_DLGMSG.

Макрос chSETDLGICONS

Поскольку во многих программах-примерах диалоговое окно выступает как главное, его значок надо изменять вручную, чтобы он корректно показывался и на панели задач, и в окне переключения задач, и в заголовке самого окна программы. Макрос chSETDLGICONS, вызываемый всякий раз, когда диалоговое окно получает сообщение WM INITDIALOG, обеспечивает корректную обработку значков.

Принудительное указание компоновщику входной функции *(w)WinMain*

Некоторые читатели предыдущих изданий этой книги, включавшие мои модули исходного кода в новые проекты, получали при их сборке сообщения об ошибках от компоновщика. Проблема была в том, что они создавали проект Win32 Console Application, заставляя компоновщик искать входную функцию *(w)main*. Но, поскольку все мои программы-примеры являются GUI-приложениями, в их исходном коде используется входная функция *_tWinMain* — вот почему компоновщик сообщал об ошибках.

Мой стандартный ответ этим читателям был таков: удалите текущий проект, создайте в проект Win32 Application (в названии шаблона проектов этого типа не должно быть слова «Console») и добавьте в него мои файлы с исходным кодом. Тогда компоновщик будет искать входную функцию *(w)WinMain*, которая и присутствует в моем коде, — никаких ошибок при сборке не возникнет.

В конце концов, чтобы сократить поток почты по этому поводу, я добавил в файл CmnHdr.h директиву *pragma*, которая заставляет компоновщик искать входную функцию *(w)WinMain*, даже если вы создали в Visual Studio проект Win32 Console Application.

Чем отличаются эти типы проектов Visual C++, как компоновщик выбирает нужный вид входных функций и как изменить стандартное поведение компоновщика, я подробно объяснил в главе 4.

Поддержка тем оформления Windows XP с помощью директивы pragma

Windows, начиная с Windows XP, поддерживает наборы настроек внешнего вида — темы оформления, определяющие облик большинства элементов пользовательского интерфейса системы и приложений. Однако по умолчанию приложения не поддерживают темы. Простой способ наделить приложение поддержкой тем — добавить к нему XML-манифест, требующий связывания к корректной версии библиотеки ComCtl32.dll, которая придаст элементам интерфейса Windows правильный вид. Компоновщик Microsoft C++ поддерживает ключ *manifestdependency*, который задают в CmnHdr.h, используя директиву *pragma* с соответствующими параметрами (подробнее о поддержке тем оформления см. в статье «Using Windows XP Visual Styles» по ссылке (http://msdn2.microsoft.com/en-us/library/ms997646.aspx).

```
CmnHdr. H

/******************************************************************************
Module:  CmnHdr.h
Notices: Copyright (c) 2008 Jeffrey Richter & Christophe Nasarre
Purpose: Common header file containing handy macros and definitions
         used throughout all the applications in the book.
         See Appendix A.
******************************************************************************/


#pragma once   // Include this header file once per compilation unit


////////////////////////// Windows Version Build Option /////////////////////////


// = 0x0600 for VISTA level from sdkddkver.h
#define _WIN32_WINNT _WIN32_WINNT_LONGHORN
#define WINVER       _WIN32_WINNT_LONGHORN


///////////////////////////// Unicode Build Option //////////////////////////////


// Always compiler using Unicode.
#ifndef UNICODE
   #define UNICODE
#endif
```

```
// When using Unicode Windows functions, use Unicode C-Runtime functions too.
#ifdef UNICODE
   #ifndef _UNICODE
      #define _UNICODE
   #endif
#endif


////////////////////////// Include Windows Definitions ////////////////////////

#pragma warning(push, 3)
#include <Windows.h>
#pragma warning(pop)
#pragma warning(push, 4)
#include <CommCtrl.h>
#include <process.h>       // For _beginthreadex


////////////// Verify that the proper header files are being used //////////////


#ifndef FILE_SKIP_COMPLETION_PORT_ON_SUCCESS
#pragma message("You are not using the latest Platform SDK header/library ")
#pragma message("files. This may prevent the project from building correctly.")
#endif


////////////// Allow code to compile cleanly at warning level 4 ///////////////


/* nonstandard extension 'single line comment' was used */
#pragma warning(disable:4001)

// unreferenced formal parameter
#pragma warning(disable:4100)

// Note: Creating precompiled header
#pragma warning(disable:4699)

// function not inlined
#pragma warning(disable:4710)

// unreferenced inline function has been removed
#pragma warning(disable:4514)

// assignment operator could not be generated
#pragma warning(disable:4512)

// conversion from 'LONGLONG' to 'ULONGLONG', signed/unsigned mismatch
#pragma warning(disable:4245)
```

```
// 'type cast' : conversion from 'LONG' to 'HINSTANCE' of greater size
#pragma warning(disable:4312)

// 'argument' : conversion from 'LPARAM' to 'LONG', possible loss of data
#pragma warning(disable:4244)

// 'wsprintf': name was marked as #pragma deprecated
#pragma warning(disable:4995)

// unary minus operator applied to unsigned type, result still unsigned
#pragma warning(disable:4146)

// 'argument' : conversion from 'size_t' to 'int', possible loss of data
#pragma warning(disable:4267)

// nonstandard extension used : nameless struct/union
#pragma warning(disable:4201)

/////////////////////////// Pragma message helper macro ////////////////////////


/*
When the compiler sees a line like this:
   #pragma chMSG(Fix this later)

it outputs a line like this:

  c:\CD\CmnHdr.h(82):Fix this later

You can easily jump directly to this line and examine the surrounding code.
*/

#define chSTR2(x) #x
#define chSTR(x)  chSTR2(x)
#define chMSG(desc) message(__FILE__ "(" chSTR(__LINE__) "):" #desc)


////////////////////////////// chINRANGE Macro //////////////////////////////


// This macro returns TRUE if a number is between two others
#define chINRANGE(low, Num, High) (((low) <= (Num)) && ((Num) <= (High)))



////////////////////////////// chSIZEOFSTRING Macro //////////////////////////////


// This macro evaluates to the number of bytes needed by a string.
#define chSIZEOFSTRING(psz)   ((lstrlen(psz) + 1) * sizeof(TCHAR))



////////////////////// chROUNDDOWN & chROUNDUP inline functions //////////////////


// This inline function rounds a value down to the nearest multiple
template <class TV, class TM>
```

```
inline TV chROUNDDOWN(TV Value, TM Multiple) {
   return((Value / Multiple) * Multiple);
}


// This inline function rounds a value down to the nearest multiple
template <class TV, class TM>
inline TV chROUNDUP(TV Value, TM Multiple) {
   return(chROUNDDOWN(Value, Multiple) +
      (((Value % Multiple) > 0) ? Multiple : 0));
}



////////////////////////////// chBEGINTHREADEX Macro ////////////////////////////


// This macro function calls the C runtime's _beginthreadex function.
// The C runtime library doesn't want to have any reliance on Windows' data
// types such as HANDLE. This means that a Windows programmer needs to cast
// values when using _beginthreadex. Since this is terribly inconvenient,
// I created this macro to perform the casting.
typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStackSize, pfnStartAddr, \
   pvParam, dwCreateFlags, pdwThreadId)                 \
      ((HANDLE)_beginthreadex(                          \
         (void *)        (psa),                         \
         (unsigned)      (cbStackSize),                 \
         (PTHREAD_START) (pfnStartAddr),                \
         (void *)        (pvParam),                     \
         (unsigned)      (dwCreateFlags),               \
         (unsigned *)    (pdwThreadId)))


////////////////// DebugBreak Improvement for x86 platforms /////////////////


#ifdef _X86_
   #define DebugBreak()    _asm { int 3 }
#endif


//////////////////////////// Software Exception Macro ////////////////////////


// Useful macro for creating your own software exception codes
#define MAKESOFTWAREEXCEPTION(Severity, Facility, Exception) \
   ((DWORD) ( \
   /* Severity code    */  (Severity       ) |     \
   /* MS(0) or Cust(1) */  (1          << 29) |     \
   /* Reserved(0)      */  (0          << 28) |     \
   /* Facility code    */  (Facility   << 16) |     \
   /* Exception code   */  (Exception <<  0)))
```

```
//////////////////////////// Quick MessageBox Macro ////////////////////////////


inline void chMB(PCSTR szMsg) {
   char szTitle[MAX_PATH];
   GetModuleFileNameA(NULL, szTitle, _countof(szTitle));
   MessageBoxA(GetActiveWindow(), szMsg, szTitle, MB_OK);
}


//////////////////////////// Assert/Verify Macros ////////////////////////////


inline void chFAIL(PSTR szMsg) {
   chMB(szMsg);
   DebugBreak();
}


// Put up an assertion failure message box.
inline void chASSERTFAIL(LPCSTR file, int line, PCSTR expr) {
   char sz[2*MAX_PATH];
   wsprintfA(sz, "File %s, line %d : %s", file, line, expr);
   chFAIL(sz);
}


// Put up a message box if an assertion fails in a debug build.
#ifdef _DEBUG
   #define chASSERT(x) if (!(x)) chASSERTFAIL(__FILE__, __LINE__, #x)
#else
   #define chASSERT(x)
#endif


// Assert in debug builds, but don't remove the code in retail builds.
#ifdef _DEBUG
   #define chVERIFY(x) chASSERT(x)
#else
   #define chVERIFY(x) (x)
#endif


//////////////////////////// chHANDLE_DLGMSG Macro ////////////////////////////


// The normal HANDLE_MSG macro in WindowsX.h does not work properly for dialog
// boxes because DlgProc returns a BOOL instead of an LRESULT (like
// WndProcs). This chHANDLE_DLGMSG macro corrects the problem:
#define chHANDLE_DLGMSG(hWnd, message, fn)                 \
   case (message): return (SetDlgMsgResult(hWnd, uMsg,     \
      HANDLE_##message((hWnd), (wParam), (lParam), (fn))))
```

```
////////////////////////// Dialog Box Icon Setting Macro ////////////////////////


// Sets the dialog box icons
inline void chSETDLGICONS(HWND hWnd, int idi) {
   SendMessage(hWnd, WM_SETICON, ICON_BIG,  (LPARAM)
      LoadIcon((HINSTANCE) GetWindowLongPtr(hWnd, GWLP_HINSTANCE),
         MAKEINTRESOURCE(idi)));
   SendMessage(hWnd, WM_SETICON, ICON_SMALL, (LPARAM)
      LoadIcon((HINSTANCE) GetWindowLongPtr(hWnd, GWLP_HINSTANCE),
      MAKEINTRESOURCE(idi)));
}


///////////////////////////// Common Linker Settings ////////////////////////////


#pragma comment(linker, "/nodefaultlib:oldnames.lib")

// Needed for supporting XP/Vista styles.
#if defined(_M_IA64)
#pragma comment(linker, "/manifestdependency:\"type='win32'
    name='Microsoft.Windows.Common-Controls' version='6.0.0.0' processorArchi-
    tecture='IA64' publicKeyToken='6595b64144ccf1df' language='*'\"")
#endif
#if defined(_M_X64)
#pragma comment(linker, "/manifestdependency:\"type='win32'
    name='Microsoft.Windows.Common-Controls' version='6.0.6000.0' processorAr-
    chitecture='amd64' publicKeyToken='6595b64144ccf1df' language='*'\"")
#endif
#if defined(M_IX86)
#pragma comment(linker, "/manifestdependency:\"type='win32'
    name='Microsoft.Windows.Common-Controls' version='6.0.0.0' processorArchi-
    tecture='x86' publicKeyToken='6595b64144ccf1df' language='*'\"")
#endif


//////////////////////////////////// End of File ////////////////////////////////
```

Оглавление

ПРИЛОЖЕНИЕ Б

Распаковщики сообщений, макросы для дочерних элементов управления и API-макросы

Используя в программах-примерах язык C/C++ совместно с распаковщиками сообщений, я стремился показать всем эти малоизвестные, но очень полезные макросы.

Распаковщики содержатся в файле WindowsX.h, поставляемом с Microsoft Visual C++. Этот файл обычно включается сразу после Windows.h, и он — не что иное, как набор директив *#define*, определяющих макросы трех групп: распаковщики сообщений (message crackers), макросы для дочерних элементов управления (child control macros) и API-макросы (API macros). Эти макросы дают разработчикам ряд дополнительных возможностей.

- Сокращают число явных преобразований типов в коде программы, а также возникающих при этом ошибок. Большой объем таких преобразований — одна из насущных проблем в программировании для Windows на C/C++. Редко когда вызов Windows-функции не требует преобразования типов. В то же время явных преобразований типов следует избегать — они препятствуют выявлению возможных ошибок при компиляции. Явно преобразуя типы, вы говорите компилятору: «Здесь передается неправильный тип, но это нормально. Мне лучше знать, что надо, а что не надо». Однако при большом количестве таких преобразований ошибиться очень легко. А потому позвольте компилятору максимально помочь Вам в выявлении ошибок.
- Делают код более читабельным.
- Упрощают перенос программ между 32- и 64-разрядными версиями Windows.

- Они просты и понятны — в конце концов это всего лишь макросы.
- Их легко включить в уже написанную программу и использовать для написания нового кода, расширяющего ее функциональность. Переделка существующего кода не потребуется.
- Их можно применять в коде, написанном как на C, так и на C++, хотя в них нет нужды при работе с библиотекой классов C++.
- Если вам понадобится что-то, чего макросы дать не могут, вы сами напишете то, что нужно, следуя модели, используемой в заголовочном файле.
- Работая с этими макросами, вам не придется обращаться к справочной информации или запоминать туманные конструкции Windows. Например, многие функции в Windows принимают параметр типа LONG, в младшем слове которого содержится одна величина, а в старшем — совсем другая. Перед вызовом функции из двух этих значений надо собрать одно (типа LONG). Обычно это делается с помощью макроса MAKELONG, определенного в файле WinDef.h. Сколько раз я ошибался при его использовании — не сосчитать, а ведь в результате функция получала неверный параметр. И здесь тоже помогут макросы из WindowsX.h.

Макросы — распаковщики сообщений

Распаковщики сообщений упрощают написание оконных процедур. Последние обычно представляют собой один огромный оператор *switch*. Мне случалось видеть в оконных процедурах операторы *switch*, содержавшие свыше 500 строк кода. Все мы знаем, что это образец плохого стиля, но… продолжаем писать именно так. Я и сам этим грешу. А распаковщики заставляют разбивать оператор ***switch*** на небольшие функции — по одной на оконное сообщение. Это значительно улучшает внутреннюю структуру кода.

Другая проблема с оконными процедурами в том, что у каждого сообщения есть параметры *wParam* и *lParam*, смысл которых зависит от типа сообщения. Бывает, что в сообщении WM_COMMAND параметр *wParam* содержит два разных значения. Старшее его слово — код уведомления, младшее — идентификатор элемента управления. Или наоборот? Вечно я это путаю. Но если вы используете распаковщики, Вам не придется запоминать эту информацию или искать ее в справочниках. Такие макросы названы распаковщиками потому, что они распаковывают параметры заданного сообщения. Чтобы обработать WMCOMMAND, Вы просто пишете функцию, которая выглядит примерно так:

```
void Cls_OnCommand(HWND hWnd, int id, HWND hWndCtl,
    UINT codeNotify) {

    switch (id) {

        case ID_SOMELISTBOX:
            If (codeNotify != LBN_SELCHANGE)
            break;
```

```
            // обрабатываем LBM_SELGHANGE
            break;
        case ID_SOMEBUTTON:
            break;
    …
```

Смотрите, как просто! Распаковщики берут параметры *wParam* и *lParam*, расщепляют их на отдельные компоненты и вызывают вашу функцию.

Чтобы использовать распаковщики, нужно внести кое-какие изменения в оператор *switch* оконной процедуры:

```
LRESULT WndProc (HWND hWnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam) {

    switch (uMsg) {
        HANDLE_MSG(hWnd, WM_COMMAND, Cls_OnCommand);
        HANDLE_MSG(hWnd, WM_PAINT,   Cls_OnPaint);
        HANDLE_MSG(hWnd, WM_DESTROY, Cls_OnDestroy);
        default:
            return(DefWindowProc(hWnd, uMsg, wParam, lParam));
    }
}
```

В файле WindowsX.h макрос HANDLEMSG определен так:

```
#define HANDLE_MSG(hwnd, message, fn) \
    case (message): \
        return HANDLE_##message((hwnd), (wParam), (lParam), (fn))
```

Для сообщения WMCOMMAND эта строка после обработки препроцессором выглядит как:

```
case (WM_COMMAND):
    return HANDLE_WM_COMMAND((hwnd), (wParam), (lParam), (Cls_OnCommand));
```

Макросы HANDLE_WM_ определены тоже в WindowsX.h. Это и есть настоящие распаковщики сообщений. Они распаковывают содержимое параметров *wParam* и *lParam*, выполняют нужные преобразования типов и вызывают соответствующую функцию — обработчик сообщения (вроде показанной ранее функции ClsOnCommand). Вот макрос HANDLE_WM_COMMAND:

```
#define HANDLE_WM_COMMAND(hwnd, wParam, lParam, fn) \
    ( (fn) ((hwnd), (int) (L0W0RD(wParam)), (HWND)(lParam),
    (UINT) HIWORD(wParam)), OL)
```

Результат раскрытия препроцессором этого макроса — вызов функции *Cls_OnCommand*, которой передаются (после соответствующих преобразований типов) распакованные части параметров *wParam* и *lParam*.

Чтобы использовать распаковщик для обработки сообщения, откройте файл WindowsX.h и найдите сообщение, которое вы собираетесь обрабатывать. Например, если вас интересует сообщение WM_COMMAND, ищите фрагмент файла, содержащий строки:

```
/* void Cls_OnCommand(HWND hWnd, int Id, HWND hWndCtl,
    UINT codeNotify); */
#define HANDLE_WM_COMMAND(hwnd, wParam, lParam, fn) \
    ((fn)((hwnd), (int)(LOWORD(wParam)), (HWND)dParam), \
    (UINT)HIWORD(wParam)), OL)
#deflne FORWARD_WM_COMMAND(hwnd, id, hwndCtl, codeNotify, fn) \
    (void)(fn)((hwnd), WM_COMMAND, \
    MAKEWPARAM((UINT)(id),(UINT)(codeNotify)), \
    (LPARAM)(HWND)(hwndCtl))
```

Первая строка—комментарий, показывающий прототип функции, которую вы должны написать. Следующая строка — уже рассмотренный нами макрос HANDLE_WM_, а последняя содержит предописатель сообщения (message forwarder). Допустим, при обработке сообщения WMCOMMAND вы хотите вызвать оконную процедуру, используемую по умолчанию. Это выглядело бы так.

```
void Cls_OnCommand (HWND hWnd, int id, HWND hWndCtl,
    UINT codeNotify) {
    // выполняем обычную обработку

    // обработка по умолчанию
    FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify,
        DefWindowProc);
}
```

Макросы FORWARD_WM_ принимают распакованные параметры сообщения и воссоздают их эквиваленты *wParam* и *lParam*, после чего вызывают указанную вами функцию. В приведенном примере это *DefWindowProc*, но так же легко можно использовать *SendMessage* или *PostMessage*. Фактически, чтобы отправить (синхронно или асинхронно) сообщение какому-то окну в системе, можно применить макрос FORWARD_WM_ — это поможет скомбинировать отдельные параметры сообщения.

Макросы для дочерних элементов управления

Эти макросы упрощают посылку сообщений дочерним элементам управления. Они очень похожи на макросы FORWARD_WM_. Имя каждого из них начинается с типа управляющего элемента, которому передается сообщение, затем идут знак подчеркивания и имя сообщения. Например, чтобы послать сообщение LB_ GETCOUNT окну списка, задействуйте такой макрос из файла WindowsX.h:

```
#define ListBox_GetCount(hwndCtl) \
    ((int)(DWORD)SNDMSG((hwndCtl), LB_GETCOUNT, 0, 0L))
```

Позвольте сделать несколько замечаний по этому макросу. Во-первых, у него только один параметр *(hwndCtl)* — описатель окна списка. Так как сообщение LB_GETCOUNT игнорирует параметры wParam и lParam, вам вообще нет нужды беспокоиться о них. Макрос, как вы уже видели, автоматичес-

ки передаст нули. Во-вторых, тип значения, возвращаемого *SendMessage*, преобразуется в *int*, в связи с чем не нужно самому преобразовывать тип.

Что мне не нравится в этих макросах, так это необходимость передавать им описатель окна элемента управления. Чаще всего элементы управления, которым вы посылаете сообщения, являются дочерними окнами диалогового окна. Из-за этого придется все время вызывать *GetDlgItem* примерно так:

```
int n = ListBox_GetCount(GetDlgItem(hDlg, ID_LISTBOX));
```

Этот код выполняется ничуть не медленнее, чем код с использованием *SendDlgItemMessage*, но объем программы увеличивается из-за дополнительных вызовов *GetDlgItem*. Если надо послать несколько сообщений одному и тому же элементу управления, вы, возможно, захотите обратиться к *GetDlgItem* лишь раз, сохранить описатель дочернего окна, а затем вызвать все необходимые макросы:

```
HWND hWndCtl = GetDlgItem(hDlg, ID_LISTBOX);
int n = ListBox_GetCount(hWndCtl);
ListBox_AddString(hWndCtl, TEXT("Another string"));
…
```

Если вы пишете код именно так, приложение заработает быстрее, поскольку не будет повторных вызовов *GetDlgItem*. Эта функция вообще-то выполняется довольно медленно, если в диалоговом окне много элементов управления, а искомый элемент находится где-то в конце Z-цепочки.

API-макросы

Они упрощают выполнение некоторых распространенных операций — например, создание нового шрифта, выбор его в контекст устройства и сохранение описателя исходного шрифта. Необходимый для этого код выглядит как-то вроде:

```
HFONT hFontOrig = (HFONT) SelectObject(hDC, (HGDIOBJ) hFontNew);
```

Этот оператор требует два преобразования типов, чтобы избежать предупреждений при компиляции. Как раз для этого и предназначен один из макросов, определенных в файле WindowsX.h:

```
#define SelectFont(hdc, hfont) \
    ((HFONT) SelectObject( (hdc), (HGDIOBJ) (HFONT) ((hfont)))
```

При его использовании строка кода в программе станет такой:

```
HFONT hFontOrig = SelectFont(hDC, hFontNew);
```

Этот код читать гораздо легче, и он меньше подвержен ошибкам.

В файле WindowsX.h есть еще несколько API-макросов, помогающих в реализации распространенных операций. Ознакомьтесь с ними самостоятельно и чаще их используйте.