

Джеффри Рихтер

ДЛЯ ПРОФЕССИОНАЛОВ

WINDOWS®

Создание эффективных Win32-приложений
с учетом специфики
64-разрядной версии Windows



ПИТЕР®

Microsoft Press

РУССКАЯ РЕДАКЦИЯ

*Кристин, трудно выразить словами, как много ты значишь
для меня. Твоя бьющая через край энергия всегда воодушев-
ляет меня. Твоя улыбка озаряет каждый мой день.
Видя тебя, хочется петь. Я люблю тебя (и Макса).
Счастья вам.*

*Моей матери, Арлин, сумевшей мужественно и бесстрашно
пережить самый трудный и мучительный период жизни.
Твоя любовь и поддержка сделали меня тем, кем я стал.
Где бы я ни был, ты всегда со мной.*

Jeffrey Richter

**Programming Applications
for Microsoft®
WINDOWS®**

FOURTH EDITION

Microsoft® Press

Джеффри Рихтер

ДЛЯ ПРОФЕССИОНАЛОВ

WINDOWS[®]

Создание эффективных Win32-приложений
с учетом специфики
64-разрядной версии Windows

ИЗДАНИЕ ЧЕТВЕРТОЕ

 РУССКАЯ РЕДАКЦИЯ

 **ПИТЕР[®]**

*Москва • Санкт-Петербург • Нижний Новгород • Воронеж
Новосибирск • Ростов-на-Дону • Екатеринбург • Самара
Киев • Харьков • Минск*

2008

УДК 004.43
ББК 32.973.26-018
Р49

Рихтер Дж.

Р49 Windows для профессионалов: создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows / Пер. с англ. — 4-е изд. — СПб.: Питер; М.: Издательство «Русская Редакция»; 2008. — 720 стр.: ил.

ISBN 5-272-00384-5 («Питер»)

ISBN 978-5-7502-0360-4 («Русская Редакция»)

Это издание — практически новая книга, посвященная программированию серьезных приложений на Microsoft Visual C++ в операционных системах Windows 2000 (32- и 64-разрядных версиях) и Windows 98 с использованием функций Windows API. Состоит из 27 глав, двух приложений. Гораздо глубже, чем в предыдущих изданиях, рассматриваются такие темы, как взаимодействие с операционной системой, библиотеки C/C++, программирование DLL и оптимизация кода, описываются новые механизмы и функции, появившиеся в Windows 2000, и приводится информация, специфическая для 64-разрядной Windows 2000. В этом издании автор, перейдя с языка C на C++, переработал все программы-примеры и представил ряд новых приложений, например ProcessInfo и LISWatch. Также появились совершенно новые материалы: выравнивание данных, привязка потоков к процессорам, кэш-линии процессоров, архитектура NUMA, перехват API-вызовов и др.

Книга предназначена профессиональным программистам, владеющим языком C/C++ и имеющим опыт разработки Windows-приложений. Прилагаемый компакт-диск содержит все программы из книги (исходный код и исполняемые файлы для процессоров x86, IA-64 и Alpha).

УДК 004.43
ББК 32.973.26-018

Подготовлено к печати по лицензионному договору с Microsoft Corporation, Редмонд, Вашингтон, США.

Intel является охраняемым товарным знаком корпорации Intel. Developer Studio, Microsoft, Microsoft Press, MS-DOS, Visual Basic, Visual C++, Visual Studio, Windows, Windows NT являются товарными знаками или охраняемыми товарными знаками корпорации Microsoft в США и/или других странах. Все другие товарные знаки являются собственностью соответствующих фирм.

Все названия компаний, организаций и продуктов, а также имена лиц, используемые в примерах, вымышлены и не имеют никакого отношения к реальным компаниям, организациям, продуктам и лицам.

ISBN 1-57231-996-8 (англ.)
ISBN 5-272-00384-5 («Питер»)
ISBN 978-5-7502-0360-4 («Русская Редакция»)

© Оригинальное издание на английском языке,
Джеффри Рихтер, 1999
© Перевод на русский язык, Microsoft Corporation, 2000
© Оформление и подготовка к изданию, издательство
«Русская Редакция», издательство «Питер», 2008

Оглавление

| | |
|----------------|------|
| Введение | XIII |
|----------------|------|

ЧАСТЬ I

| | |
|--|---|
| МАТЕРИАЛЫ ДЛЯ ОБЯЗАТЕЛЬНОГО ЧТЕНИЯ | 1 |
|--|---|

| | |
|---------------------------------------|---|
| ГЛАВА 1 Обработка ошибок | 2 |
|---------------------------------------|---|

| | |
|----------------------------------|---|
| Вы тоже можете это сделать | 6 |
| Программа-пример ErrorShow | 7 |

| | |
|------------------------------|----|
| ГЛАВА 2 Unicode | 11 |
|------------------------------|----|

| | |
|---|----|
| Наборы символов | 11 |
| Одно- и двухбайтовые наборы символов | 11 |
| Unicode: набор «широких» символов | 12 |
| Почему Unicode? | 13 |
| Windows 2000 и Unicode | 13 |
| Windows 98 и Unicode | 13 |
| Windows CE и Unicode | 14 |
| В чью пользу счет? | 14 |
| Unicode и COM | 15 |
| Как писать программу с использованием Unicode | 15 |
| Unicode и библиотека C | 15 |
| Типы данных, определенные в Windows для Unicode | 17 |
| Unicode- и ANSI-функции в Windows | 18 |
| Строковые функции Windows | 19 |
| Создание программ, способных использовать и ANSI, и Unicode | 20 |
| Ресурсы | 23 |
| Текстовые файлы | 23 |
| Перекодировка строк из Unicode в ANSI и обратно | 24 |

| | |
|-----------------------------------|----|
| ГЛАВА 3 Объекты ядра | 28 |
|-----------------------------------|----|

| | |
|---|----|
| Что такое объект ядра | 28 |
| Учет пользователей объектов ядра | 29 |
| Защита | 29 |
| Таблица описателей объектов ядра | 31 |
| Создание объекта ядра | 32 |
| Закрытие объекта ядра | 33 |
| Совместное использование объектов ядра несколькими процессами | 34 |
| Наследование описателя объекта | 35 |
| Именованные объекты | 38 |
| Дублирование описателей объектов | 42 |

ЧАСТЬ II

| | |
|-------------------------|----|
| НАЧИНАЕМ РАБОТАТЬ | 47 |
|-------------------------|----|

| | |
|-------------------------------|----|
| ГЛАВА 4 Процессы | 48 |
|-------------------------------|----|

| | |
|--------------------------------------|----|
| Ваше первое Windows-приложение | 49 |
| Описатель экземпляра процесса | 53 |

| | |
|---|------------|
| Описатель предыдущего экземпляра процесса | 54 |
| Командная строка процесса | 55 |
| Переменные окружения | 56 |
| Привязка к процессорам | 59 |
| Режим обработки ошибок | 59 |
| Текущие диск и каталог для процесса | 60 |
| Определение версии системы | 61 |
| Функция <i>CreateProcess</i> | 64 |
| Параметры <i>pszApplicationName</i> и <i>pszCommandLine</i> | 65 |
| Параметры <i>psaProcess</i> , <i>psaThread</i> и <i>blnInheritHandles</i> | 67 |
| Параметр <i>fdwCreate</i> | 69 |
| Параметр <i>pvEnvironment</i> | 71 |
| Параметр <i>pszCurDir</i> | 71 |
| Параметр <i>psiStartInfo</i> | 71 |
| Параметр <i>ppiProcInfo</i> | 75 |
| Завершение процесса | 77 |
| Возврат управления входной функцией первичного потока | 77 |
| Функция <i>ExitProcess</i> | 77 |
| Функция <i>TerminateProcess</i> | 79 |
| Когда все потоки процесса уходят | 79 |
| Что происходит при завершении процесса | 80 |
| Дочерние процессы | 80 |
| Запуск обособленных дочерних процессов | 82 |
| Перечисление процессов, выполняемых в системе | 82 |
| Программа-пример <i>ProcessInfo</i> | 83 |
| ГЛАВА 5 Задания | 98 |
| Определение ограничений, налагаемых на процессы в задании | 101 |
| Включение процесса в задание | 107 |
| Завершение всех процессов в задании | 108 |
| Получение статистической информации о задании | 109 |
| Уведомления заданий | 112 |
| Программа-пример <i>JobLab</i> | 114 |
| ГЛАВА 6 Базовые сведения о потоках | 130 |
| В каких случаях потоки создаются | 131 |
| И в каких случаях потоки не создаются | 132 |
| Ваша первая функция потока | 133 |
| Функция <i>CreateThread</i> | 134 |
| Параметр <i>psa</i> | 135 |
| Параметр <i>cbStack</i> | 135 |
| Параметры <i>pfnStartAddr</i> и <i>pvParam</i> | 136 |
| Параметр <i>fdwCreate</i> | 137 |
| Параметр <i>pdwThreadId</i> | 137 |
| Завершение потока | 137 |
| Возврат управления функцией потока | 138 |
| Функция <i>ExitThread</i> | 138 |
| Функция <i>TerminateThread</i> | 138 |
| Если завершается процесс | 139 |
| Что происходит при завершении потока | 139 |
| Кое-что о внутреннем устройстве потока | 140 |
| Некоторые соображения по библиотеке C/C++ | 142 |
| Ой, вместо <i>_beginthreadex</i> я по ошибке вызвал <i>CreateThread</i> | 150 |

| | |
|---|------------|
| Библиотечные функции, которые лучше не вызывать | 151 |
| Как узнать о себе | 152 |
| Преобразование псевдоописателя в настоящий описатель | 152 |
| ГЛАВА 7 Планирование потоков, приоритет и привязка к процессорам | 155 |
| Приостановка и возобновление потоков | 156 |
| Приостановка и возобновление процессов | 157 |
| Функция <i>Sleep</i> | 159 |
| Переключение потоков | 159 |
| Определение периодов выполнения потока | 160 |
| Структура CONTEXT | 162 |
| Приоритеты потоков | 167 |
| Абстрагирование приоритетов | 168 |
| Программирование приоритетов | 171 |
| Динамическое изменение уровня приоритета потока | 174 |
| Подстройка планировщика для активного процесса | 175 |
| Программа-пример Scheduling Lab | 176 |
| Привязка потоков к процессорам | 182 |
| ГЛАВА 8 Синхронизация потоков в пользовательском режиме | 187 |
| Атомарный доступ: семейство <i>Interlocked</i> -функций | 187 |
| Кэш-линии | 193 |
| Более сложные методы синхронизации потоков | 195 |
| Худшее, что можно сделать | 195 |
| Критические секции | 197 |
| Критические секции: важное дополнение | 200 |
| Критические секции и спин-блокировка | 202 |
| Критические секции и обработка ошибок | 203 |
| Несколько полезных приемов | 204 |
| ГЛАВА 9 Синхронизация потоков с использованием объектов ядра | 207 |
| <i>Wait</i> -функции | 209 |
| Побочные эффекты успешного ожидания | 211 |
| События | 213 |
| Программа-пример Handshake | 216 |
| Ожидаемые таймеры | 221 |
| Ожидаемые таймеры и APC-очередь | 224 |
| И еще кое-что о таймерах | 226 |
| Семафоры | 227 |
| Мьютексы | 229 |
| Мьютексы и критические секции | 231 |
| Программа-пример Queue | 232 |
| Сводная таблица объектов, используемых для синхронизации потоков | 239 |
| Другие функции, применяемые в синхронизации потоков | 240 |
| Асинхронный ввод-вывод на устройствах | 240 |
| Функция <i>WaitForInputIdle</i> | 241 |
| Функция <i>MsgWaitForMultipleObjects(Ex)</i> | 242 |
| Функция <i>WaitForDebugEvent</i> | 242 |
| Функция <i>SignalObjectAndWait</i> | 242 |
| ГЛАВА 10 Полезные средства для синхронизации потоков | 245 |
| Реализация критической секции: объект-оптекс | 245 |
| Программа-пример Optex | 247 |
| Создание инверсных семафоров и типов данных, безопасных в многопоточной среде | 256 |
| Программа-пример InterlockedType | 260 |

| | |
|---|------------|
| Синхронизация в сценарии «один писатель/группа читателей» | 267 |
| Программа-пример SWMRG | 269 |
| Реализация функции <i>WaitForMultipleExpressions</i> | 275 |
| Программа-пример <i>WaitForMultExp</i> | 277 |
| ГЛАВА 11 Пулы потоков | 289 |
| Сценарий 1: асинхронный вызов функций | 290 |
| Сценарий 2: вызов функций через определенные интервалы времени | 292 |
| Программа-пример <i>TimedMsgBox</i> | 296 |
| Сценарий 3: вызов функций при освобождении отдельных объектов ядра | 298 |
| Сценарий 4: вызов функций по завершении запросов на асинхронный ввод-вывод | 300 |
| ГЛАВА 12 Волокна | 303 |
| Работа с волокнами | 303 |
| Программа-пример <i>Counter</i> | 306 |
| Ч А С Т Ь I I I | |
| УПРАВЛЕНИЕ ПАМЯТЬЮ | 313 |
| ГЛАВА 13 Архитектура памяти в Windows | 314 |
| Виртуальное адресное пространство процесса | 314 |
| Как адресное пространство разбивается на разделы | 315 |
| Раздел для выявления нулевых указателей (Windows 2000 и Windows 98) | 316 |
| Раздел для совместимости с программами DOS и 16-разрядной Windows (только Windows 98) | 316 |
| Раздел для кода и данных пользовательского режима (Windows 2000 и Windows 98) .. | 316 |
| Закрытый раздел размером 64 Кб (только Windows 2000) | 318 |
| Раздел для общих MMF (только Windows 98) | 319 |
| Раздел для кода и данных режима ядра (Windows 2000 и Windows 98) | 319 |
| Регионы в адресном пространстве | 319 |
| Передача региону физической памяти | 320 |
| Физическая память и страничный файл | 320 |
| Физическая память в страничном файле не хранится | 323 |
| Атрибуты защиты | 324 |
| Защита типа «копирование при записи» | 325 |
| Специальные флаги атрибутов защиты | 326 |
| Подводя итоги | 326 |
| Блоки внутри регионов | 329 |
| Особенности адресного пространства в Windows 98 | 333 |
| Выравнивание данных | 337 |
| ГЛАВА 14 Исследование виртуальной памяти | 342 |
| Системная информация | 342 |
| Программа-пример <i>SysInfo</i> | 343 |
| Статус виртуальной памяти | 347 |
| Программа-пример <i>VMStat</i> | 348 |
| Определение состояния адресного пространства | 351 |
| Функция <i>VMQuery</i> | 353 |
| Программа-пример <i>VMMap</i> | 360 |
| ГЛАВА 15 Использование виртуальной памяти в приложениях | 368 |
| Резервирование региона в адресном пространстве | 368 |
| Передача памяти зарезервированному региону | 370 |
| Резервирование региона с одновременной передачей физической памяти | 370 |
| В какой момент региону передают физическую память | 371 |

| | |
|---|------------|
| Возврат физической памяти и освобождение региона | 373 |
| В какой момент физическую память возвращают системе | 374 |
| Программа-пример VMAlloc | 375 |
| Изменение атрибутов защиты | 382 |
| Сброс содержимого физической памяти | 383 |
| Программа-пример MemReset | 384 |
| Механизм Address Windowing Extensions (только Windows 2000) | 387 |
| Программа-пример AWE | 391 |
| ГЛАВА 16 Стек потока | 398 |
| Стек потока в Windows 98 | 401 |
| Функция из библиотеки C/C++ для контроля стека | 403 |
| Программа-пример Summation | 404 |
| ГЛАВА 17 Проецируемые в память файлы | 409 |
| Проецирование в память EXE- и DLL-файлов | 409 |
| Статические данные не разделяются несколькими экземплярами EXE или DLL | 411 |
| Статические данные разделяются несколькими экземплярами EXE или DLL | 413 |
| Программа-пример AppInst | 417 |
| Файлы данных, проецируемые в память | 420 |
| Метод 1: один файл, один буфер | 420 |
| Метод 2: два файла, один буфер | 421 |
| Метод 3: один файл, два буфера | 421 |
| Метод 4: один файл и никаких буферов | 422 |
| Использование проецируемых в память файлов | 422 |
| Этап 1: создание или открытие объекта ядра «файл» | 422 |
| Этап 2: создание объекта ядра «проекция файла» | 423 |
| Этап 3: проецирование файловых данных на адресное пространство процесса | 426 |
| Этап 4: отключение файла данных от адресного пространства процесса | 429 |
| Этапы 5 и 6: закрытие объектов «проекция файла» и «файл» | 430 |
| Программа-пример FileRev | 431 |
| Обработка больших файлов | 437 |
| Проецируемые файлы и когерентность | 438 |
| Базовый адрес файла, проецируемого в память | 439 |
| Особенности проецирования файлов на разных платформах | 441 |
| Совместный доступ процессов к данным через механизм проецирования | 443 |
| Файлы, проецируемые на физическую память из страничного файла | 443 |
| Программа-пример MMFSshare | 444 |
| Частичная передача физической памяти проецируемым файлам | 448 |
| Программа-пример MMFSparse | 450 |
| ГЛАВА 18 Динамически распределяемая память | 461 |
| Стандартная куча процесса | 461 |
| Дополнительные кучи в процессе | 462 |
| Защита компонентов | 462 |
| Более эффективное управление памятью | 463 |
| Локальный доступ | 464 |
| Исключение издержек, связанных с синхронизацией потоков | 464 |
| Быстрое освобождение всей памяти в куче | 465 |
| Создание дополнительной кучи | 465 |
| Выделение блока памяти из кучи | 466 |
| Изменение размера блока | 467 |
| Определение размера блока | 468 |
| Освобождение блока | 468 |

| | |
|---|-----|
| Уничтожение кучи | 468 |
| Использование куч в программах на C++ | 469 |
| Другие функции управления кучами | 472 |

ЧАСТЬ IV

| | |
|---|-----|
| ДИНАМИЧЕСКИ ПОДКЛЮЧАЕМЫЕ БИБЛИОТЕКИ | 475 |
|---|-----|

| | |
|-----------------------------------|------------|
| ГЛАВА 19 DLL: основы | 476 |
|-----------------------------------|------------|

| | |
|---|-----|
| DLL и адресное пространство процесса | 477 |
| Общая картина | 479 |
| Создание DLL-модуля | 481 |
| Что такое экспорт | 483 |
| Создание DLL для использования с другими средствами разработки (отличными от Visual C++) | 485 |
| Создание EXE-модуля | 486 |
| Что такое импорт | 487 |
| Выполнение EXE-модуля | 489 |

| | |
|--|------------|
| ГЛАВА 20 DLL: более сложные методы программирования | 492 |
|--|------------|

| | |
|--|-----|
| Явная загрузка DLL и связывание идентификаторов | 492 |
| Явная загрузка DLL | 492 |
| Явная выгрузка DLL | 494 |
| Явное подключение экспортируемого идентификатора | 496 |
| Функция входа/выхода | 497 |
| Уведомление DLL_PROCESS_ATTACH | 498 |
| Уведомление DLL_PROCESS_DETACH | 499 |
| Уведомление DLL_THREAD_ATTACH | 501 |
| Уведомление DLL_THREAD_DETACH | 502 |
| Как система упорядочивает вызовы <i>DllMain</i> | 503 |
| Функция <i>DllMain</i> и библиотека C/C++ | 505 |
| Отложенная загрузка DLL | 506 |
| Программа-пример DelayLoadApp | 510 |
| Переадресация вызовов функций | 516 |
| Известные DLL | 517 |
| Перенаправление DLL | 518 |
| Модификация базовых адресов модулей | 519 |
| Связывание модулей | 524 |

| | |
|---|------------|
| ГЛАВА 21 Локальная память потока | 527 |
|---|------------|

| | |
|--|-----|
| Динамическая локальная память потока | 528 |
| Использование динамической TLS | 530 |
| Статическая локальная память потока | 531 |

| | |
|--|------------|
| ГЛАВА 22 Внедрение DLL и перехват API-вызовов | 533 |
|--|------------|

| | |
|--|-----|
| Пример внедрения DLL | 533 |
| Внедрение DLL с использованием реестра | 535 |
| Внедрение DLL с помощью ловушек | 537 |
| Утилита для сохранения позиций элементов на рабочем столе | 538 |
| Внедрение DLL с помощью удаленных потоков | 549 |
| Программа-пример InjLib | 553 |
| Библиотека <i>ImgWalk.dll</i> | 558 |
| Внедрение троянской DLL | 561 |
| Внедрение DLL как отладчика | 561 |
| Внедрение кода в среде Windows 98 через проецируемый в память файл | 562 |
| Внедрение кода через функцию <i>CreateProcess</i> | 562 |

| | |
|--|-----|
| Перехват API-вызовов: пример | 563 |
| Перехват API-вызовов подменой кода | 563 |
| Перехват API-вызовов с использованием раздела импорта | 564 |
| Программа-пример LastMsgBoxInfo | 567 |
| ЧАСТЬ V | |
| СТРУКТУРНАЯ ОБРАБОТКА ИСКЛЮЧЕНИЙ | 583 |
| ГЛАВА 23 Обработчики завершения | 584 |
| Примеры использования обработчиков завершения | 585 |
| <i>Funcenstein1</i> | 585 |
| <i>Funcenstein2</i> | 586 |
| <i>Funcenstein3</i> | 587 |
| <i>Funcfurter1</i> | 588 |
| Проверьте себя: <i>FuncuDoodleDoo</i> | 589 |
| <i>Funcenstein4</i> | 590 |
| <i>Funcarama1</i> | 591 |
| <i>Funcarama2</i> | 592 |
| <i>Funcarama3</i> | 592 |
| <i>Funcarama4</i> : последний рубеж | 593 |
| И еще о блоке <i>finally</i> | 595 |
| <i>Funcfurter2</i> | 595 |
| Программа-пример SEHTerm | 596 |
| ГЛАВА 24 Фильтры и обработчики исключений | 599 |
| Примеры использования фильтров и обработчиков исключений | 599 |
| <i>Funcmeister1</i> | 600 |
| <i>Funcmeister2</i> | 600 |
| EXCEPTION_EXECUTE_HANDLER | 602 |
| Некоторые полезные примеры | 603 |
| Глобальная раскрутка | 605 |
| Остановка глобальной раскрутки | 608 |
| EXCEPTION_CONTINUE_EXECUTION | 609 |
| Будьте осторожны с EXCEPTION_CONTINUE_EXECUTION | 610 |
| EXCEPTION_CONTINUE_SEARCH | 611 |
| Функция <i>GetExceptionCode</i> | 612 |
| Функция <i>GetExceptionInformation</i> | 616 |
| Программные исключения | 620 |
| ГЛАВА 25 Необработанные исключения и исключения C++ | 623 |
| Отладка по запросу | 625 |
| Отключение вывода сообщений об исключениях | 626 |
| Принудительное завершение процесса | 626 |
| Создание оболочки вокруг функции потока | 627 |
| Создание оболочки вокруг всех функций потоков | 627 |
| Автоматический вызов отладчика | 628 |
| Явный вызов функции <i>UnhandledExceptionFilter</i> | 628 |
| Функция <i>UnhandledExceptionFilter</i> изнутри | 628 |
| Исключения и отладчик | 630 |
| Программа-пример Spreadsheet | 633 |
| Исключения C++ и структурные исключения | 642 |
| Перехват структурных исключений в C++ | 644 |

ЧАСТЬ VI

| | |
|--|------------|
| ОПЕРАЦИИ С ОКНАМИ | 647 |
| ГЛАВА 26 Оконные сообщения | 648 |
| Очередь сообщений потока | 649 |
| Посылка асинхронных сообщений в очередь потока | 649 |
| Посылка синхронных сообщений окну | 651 |
| Пробуждение потока | 656 |
| Флаги состояния очереди | 657 |
| Алгоритм выборки сообщений из очереди потока | 658 |
| Пробуждение потока с использованием объектов ядра или флагов состояния очереди | 661 |
| Передача данных через сообщения | 664 |
| Программа-пример CopyData | 666 |
| Как Windows манипулирует с ANSI/Unicode-символами и строками | 669 |
| ГЛАВА 27 Модель аппаратного ввода и локальное состояние ввода | 671 |
| Поток необработанного ввода | 671 |
| Локальное состояние ввода | 673 |
| Ввод с клавиатуры и фокус | 673 |
| Управление курсором мыши | 677 |
| Подключение к очередям виртуального ввода и переменным локального состояния ввода | 678 |
| Программа-пример LISLab | 680 |
| Программа-пример LISWatch | 692 |
| ПРИЛОЖЕНИЕ А Среда разработки | 698 |
| Заголовочный файл CmnHdr.h | 698 |
| Раздел Windows Version Build Option | 698 |
| Раздел Unicode Build Option | 699 |
| Раздел Windows Definitions и диагностика уровня 4 | 699 |
| Вспомогательный макрос Pragma Message | 700 |
| Макросы chINRANGE и chDIMOF | 700 |
| Макрос chBEGINTHREADEX | 700 |
| Моя реализация <i>DebugBreak</i> для платформ <i>x86</i> | 702 |
| Определение кодов программных исключений | 702 |
| Макрос chMB | 702 |
| Макросы chASSERT и chVERIFY | 702 |
| Макрос chHANDLE_DLGMSG | 702 |
| Макрос chSETDLGICONS | 703 |
| Встраиваемые функции для проверки версии операционной системы | 703 |
| Проверка на поддержку Unicode | 703 |
| Принудительное указание компоновщику входной функции (<i>w</i>) <i>WinMain</i> | 703 |
| ПРИЛОЖЕНИЕ Б Распаковщики сообщений, макросы для дочерних элементов управления и API-макросы | 709 |
| Макросы — распаковщики сообщений | 710 |
| Макросы для дочерних элементов управления | 712 |
| API-макросы | 713 |
| Предметный указатель | 714 |

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.

Например, 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.

И вот тут мы подходим к тому, о чем же моя книга. А она — о строительных кирпичах 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.

Сегодняшние Windows-платформы

Сейчас Microsoft поставляет операционные системы Windows с тремя ядрами. Каждое ядро оптимизировано под свои виды вычислительных задач. Microsoft пытается переманить разработчиков программного обеспечения на Windows-платформы, утверждая, что интерфейс прикладного программирования (application programming interface, API) у каждой из них одинаков. Это означает лишь то, что, научившись писать Windows-приложения для одного ядра, Вы поймете, как сделать то же самое для остальных.

Поскольку я объясняю, как писать Windows-приложения на основе Windows API, то теоретически все, о чем Вы узнаете из моей книги, применимо ко всем трем ядрам. На самом деле они сильно отличаются друг от друга, и поэтому одни и те же функции соответствующих операционных систем реализованы по-разному. Скажем так: базовые концепции одинаковы, но детали могут различаться.

Начнем с того, что представляют собой эти три ядра Windows.

Ядро Windows 2000

Windows 2000 — это операционная система Microsoft класса «high-end». Список ее возможностей и особенностей займет не одну страницу. Вот лишь некоторые из них (в совершенно произвольном порядке).

- Windows 2000 рассчитана на рабочие станции и серверы, а также на применение в центрах обработки данных.
- Отказоустойчива — плохо написанные программы не могут привести к краху системы.
- Защищена — несанкционированный доступ к ресурсам (например, файлам или принтерам), управляемым этой системой, невозможен.
- Богатый набор средств и утилит для администрирования системы в масштабах организации.
- Ядро Windows 2000 написано в основном на C и C++, поэтому система легко портируется (переносится) на процессоры с другими архитектурами.
- Полностью поддерживает Unicode, что упрощает локализацию и работу с использованием различных языков.
- Имеет высокоэффективную подсистему управления памятью с чрезвычайно широкими возможностями.
- Поддерживает структурную обработку исключений (structured exception handling, SEH), облегчая восстановление после ошибок.
- Позволяет расширять функциональность за счет динамически подключаемых библиотек (DLL).
- Поддерживает многопоточность и мультипроцессорную обработку, обеспечивая высокую масштабируемость системы.
- Файловая система Windows 2000 дает возможность отслеживать, как пользователи манипулируют с данными на своих машинах.

Ядро Windows 98

Windows 98 — операционная система потребительского класса. Она обладает многими возможностями Windows 2000, но некоторые ключевые из них не поддерживает. Так, Windows 98 не отнесешь к числу отказоустойчивых (приложение вполне способно привести к краху системы), она менее защищена, работает только с одним процессором (что ограничивает ее масштабируемость) и поддерживает Unicode лишь частично.

Microsoft намерена ликвидировать ядро Windows 98, поскольку его доработка до уровня ядра Windows 2000 потребовала бы слишком много усилий. Да и кому нужно еще одно ядро Windows 2000? Так что Windows 2000 — это вроде бы надолго, а Windows 98 проживет года два-три, если не меньше.

Но почему вообще существует ядро Windows 98? Ответ очень прост: Windows 98 более дружелюбна к пользователю, чем Windows 2000. Потребители не любят регистрироваться на своих компьютерах, не хотят заниматься администрированием и т. д. Плюс ко всему в компьютерные игры они играют чаще, чем сотрудники корпораций в рабочее время (впрочем, это спорно). Многие старые игровые программы обращаются к оборудованию напрямую, что может приводить к зависанию компьютера. Windows 2000 — операционная система с отказоустойчивым ядром — такого не по-

звояет никому. Любая программа, которая пытается напрямую обратиться к оборудованию, немедленно завершается, не успев навредить ни себе, ни другим.

По этим причинам Windows 98 все еще с нами, и ее доля на рынке операционных систем весьма велика. Microsoft активно работает над тем, чтобы Windows 2000 стала дружелюбнее к пользователю, — очень скоро появится потребительская версия ее ядра. Поскольку ядра Windows 98 и Windows 2000 имеют сходные наборы функциональных возможностей и поскольку они наиболее популярны, я решил сосредоточиться в этой книге именно на них.

Готовя книгу, я старался обращать внимание на отличия реализаций Win32 API в Windows 98 и Windows 2000. Материалы такого рода я обводил рамками и, как показано ниже, помечал соответствующими значками — чтобы привлечь внимание читателей к каким-то деталям, характерным для той или иной платформы.

WINDOWS 98 Здесь рассказывается об особенностях реализации на платформе Windows 98.

WINDOWS 2000 А тут — об особенностях реализации на платформе Windows 2000.

Windows 95 я особо не рассматриваю, но все, что относится к Windows 98, применимо и к ней, так как ядра у них совершенно одинаковые.

Ядро Windows CE

Windows CE — самое новое ядро Windows от Microsoft. Оно рассчитано главным образом на карманные и автомобильные компьютеры, «интеллектуальные» терминалы, тостеры, микроволновые печи и торговые автоматы. Большинство таких устройств должно потреблять минимум электроэнергии, у них очень мало памяти, а дисков чаще всего просто нет. Из-за столь жестких ограничений Microsoft пришлось создать совершенно новое ядро операционной системы, намного менее требовательное к памяти, чем ядро Windows 98 или Windows 2000.

Как ни странно, Windows CE довольно мощная операционная система. Устройства, которыми она управляет, предназначены только для индивидуального использования, поэтому ее ядро не поддерживает администрирование, масштабирование и т. д. Тем не менее практически все концепции Win32 применимы и к данной платформе. Различия обычно проявляются там, где Windows CE накладывает ограничения на те или иные Win32-функции.

Завтрашние Windows-платформы (64-разрядная Windows 2000)

Будущее уже совсем близко. Когда я пишу эти строки, Microsoft напряженно трудится над переносом ядра Windows 2000 на 64-разрядную платформу. Предполагается, что эта истинно 64-разрядная операционная система получит название *64-bit Windows 2000* (64-разрядная Windows 2000). На первых порах она будет работать на процессорах Alpha (архитектура AXP64) от Compaq, а чуть позже и на новых процессорах Itanium (архитектура IA-64) от Intel.

Процессоры Alpha всегда были 64-разрядными. Так что, если у Вас есть машина с одним из этих процессоров, Вы просто установите 64-разрядную Windows 2000 и получите полноценную 64-разрядную программно-аппаратную платформу. Процес-

соры Intel серии Pentium (и более ранние) имеют 32-разрядную архитектуру (IA-32). Машины с такими процессорами не смогут работать с 64-разрядной Windows 2000. Intel сравнительно недавно закончил разработку новой 64-разрядной архитектуры процессоров и сейчас готовит к выпуску процессор Itanium (его кодовое название было Merced). Поставка машин на базе Itanium ожидается уже в 2000 году.

Меня очень интересует 64-разрядная Windows 2000, и я давно готовлюсь к ее появлению. Сейчас на Web-узле Microsoft можно найти много статей о 64-разрядной Windows 2000 и о том, какие изменения она принесет разработчикам программного обеспечения. С радостью сообщаю Вам следующее.

- Ядро 64-разрядной Windows 2000 получено в результате портирования ядра 32-разрядной версии. А значит, все, что Вы узнали о 32-разрядной Windows 2000, применимо и к 64-разрядной. В сущности, Microsoft так модифицировала исходный код 32-разрядной Windows, что из него можно получить как 32-, так и 64-разрядную систему. Таким образом, у них теперь одна база исходного кода, и любые новшества или исправления будут вноситься в обе системы одновременно.
- Поскольку эти ядра построены на одном коде и одинаковых концепциях, Windows API идентичен на обеих платформах. Следовательно, Ваши приложения потребуют лишь минимальной модификации.
- Если перенос 32-разрядных приложений так легок, то вскоре появится масса инструментальных средств (вроде Microsoft Developer Studio), поддерживающих разработку 64-разрядного программного обеспечения.
- Конечно, 64-разрядная Windows сможет выполнять и 32-разрядные приложения. Но, судя по обещаниям, истинно 64-разрядные приложения будут работать в ней гораздо быстрее.
- Вам не придется учиться заново. Вы обрадуетесь, узнав, что большинство типов данных осталось 32-разрядным. Это относится к целым типам, DWORD, LONG, BOOL и т. д. По сути, беспокоиться следует лишь об указателях и некоторых описателях, так как теперь они являются 64-разрядными.

Сведений о том, как подготовить исходный код к выполнению на 64-разрядной платформе, вполне хватает и на Web-узле Microsoft, 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, что позволило мне протестировать их на очень ранней версии 64-разрядной Windows 2000 для процессоров Alpha. 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 четвертом издании

Четвертое издание является практически новой книгой. Я решил разбить материал на большее количество глав для более четкой структуризации и изменил порядок его изложения. Надеюсь, так будет легче изучать его и усваивать. Например, глава по Unicode теперь находится в начале книги, поскольку с ним так или иначе связаны многие другие темы.

Более того, все темы рассматриваются гораздо глубже, чем в предыдущих изданиях. В частности, я подробнее, чем раньше, объясняю внутреннее устройство Windows,

чтобы Вы точно знали, что происходит за кулисами этой системы. Намного детальнее я рассказываю и о том, как взаимодействует с системой библиотека C/C++ (C/C++ run-time library) — особенно при создании и уничтожении процессов и потоков. Динамически подключаемым библиотекам я также уделяю больше внимания.

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

- **Новшества Windows 2000.** Книгу было бы нельзя считать действительно переработанной, не будь в ней отражены новшества Windows 2000: объект ядра «задание» (job kernel object), функции для создания пула потоков (thread pooling functions), изменения в механизме планирования потоков (thread scheduling), расширения Address Windowing, вспомогательные информационные функции (toolhelp functions), разреженные файлы (sparse files) 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.
- **Практичность программ-примеров.** Я заменил многие старые примеры новыми, более полезными в повседневной работе; они иллюстрируют решение не абстрактных, а реальных проблем программирования.
- **Применение C++.** По требованию читателей примеры теперь написаны на C++. В итоге они стали компактнее и легче для понимания.
- **Повторно используемый код.** Я старался писать по возможности универсальный и повторно используемый код. 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 VMMap.** Эта программа-пример из предыдущих изданий серьезно усовершенствована. Ее новая версия дает возможность исследовать адресное пространство любого процесса, выяснять полные имена (вместе с путями) любых файлов данных, спроецированных в адресное пространство процесса, копировать информацию из памяти в буфер обмена и (если Вы пожелаете) просматривать только регионы или блоки памяти внутри регионов.
- **Утилита ProcessInfo.** Это новая утилита. Она показывает, какие процессы выполняются в системе и какие DLL используются тем или иным модулем. Как только Вы выбираете конкретный процесс, ProcessInfo может запустить утилиту VMMap для просмотра всего адресного пространства этого процесса. ProcessInfo позволяет также узнать, какие модули загружены в системе и какие исполняемые файлы используют определенный модуль. Кроме того, Вы сможете увидеть, у каких модулей были изменены базовые адреса из-за неподходящих значений.
- **Утилита LISWatch.** Тоже новая утилита. Она отслеживает общесистемные и специфические для конкретного потока изменения в локальном состоянии ввода. Эта утилита поможет Вам разобраться в проблемах, связанных с перемещением фокуса ввода в пользовательском интерфейсе.
- **Информация по оптимизации кода.** В этом издании я даю гораздо больше информации о том, как повысить быстродействие кода и сделать его компактнее. В частности, я подробно рассказываю о выравнивании данных (data alignment), привязке к процессорам (processor affinity), кэш-линиях процессо-

ра (CPU cache lines), модификации базовых адресов (rebasing), связывании модулей (module binding), отложенной загрузке DLL (delay-loading DLLs) и др.

- **Существенно переработанный материал по синхронизации потоков.** Я полностью переписал и перестроил весь материал по синхронизации потоков. Теперь я сначала рассказываю о самых эффективных способах синхронизации, а наименее эффективные обсуждаю в конце. Попутно я добавил новую главу, посвященную набору инструментальных средств, которые помогают решать наиболее распространенные проблемы синхронизации потоков.
- **Детальная информация о форматах исполняемых файлов.** Форматы файлов EXE- и DLL-модулей рассматриваются намного подробнее. Я рассказываю о различных разделах этих модулей и некоторых специфических параметрах компоновщика, которые позволяют делать с модулями весьма интересные вещи.
- **Более подробные сведения о DLL.** Главы по DLL тоже полностью переписаны и перестроены. Первая из них отвечает на два основных вопроса: «Что такое DLL?» и «Как ее создать?». Остальные главы по DLL посвящены весьма продвинутому и отчасти новым темам — явному связыванию (explicit linking), отложенной загрузке, переадресации вызова функции (function forwarding), перенаправлению DLL (DLL redirection) (новая возможность, появившаяся в Windows 2000), модификации базового адреса модуля (module rebasing) и связыванию.
- **Перехват API-вызовов.** Да, это правда. За последние годы я получил столько почты с вопросами по перехвату API-вызовов (API hooking), что в конце концов решил включить эту тему в свою книгу. Я представлю Вам несколько C++-классов, которые сделают перехват API-вызовов в одном или всех модулях процесса тривиальной задачей. Вы сможете перехватывать даже вызовы *LoadLibrary* 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 увидите C++-класс — оболочку кода, управляющего виртуальной памятью за счет структурной обработки исключений (structured exception handling). Я также добавил сведения о соответствующих приемах отладки и о том, как обработка исключений в C++ соотносится со структурной обработкой исключений.
- **Обработка ошибок.** Это новая глава. В ней показывается, как правильно перехватывать ошибки при вызове API-функций. Здесь же представлены некоторые приемы отладки и ряд других сведений.
- **Windows Installer.** Чуть не забыл: программы-примеры (все они содержатся на прилагаемом компакт-диске) используют преимущества нового Windows Installer, встроенного в Windows 2000. Это позволит полностью контролировать состав устанавливаемого программного обеспечения и легко удалить больше не нужные его части через апплет Add/Remove Programs в Control Panel. Если Вы используете Windows 95/98 или Windows NT 4.0, программа Setup с моего компакт-диска сначала установит Windows Installer. Но, разумеется, Вы можете и сами скопировать с компакт-диска любые интересующие Вас файлы с исходным или исполняемым кодом.

В этой книге нет ошибок

Этот заголовок отражает лишь то, что я хотел бы сказать. Но все мы знаем: это полное вранье. Мои редакторы и я очень старались без ошибок донести до Вас новую, точную и глубокую информацию в простом для понимания виде. Увы, даже собрав самую фантастическую команду, никто не застрахован от проколов. Найдете какую-нибудь ошибку в этой книге, сообщите мне на <http://www.JeffreyRichter.com> — буду крайне признателен.

Содержимое компакт-диска и требования к системе

Компакт-диск, прилагаемый к книге, содержит исходный код и исполняемые файлы всех программ-примеров. Эти программы написаны и скомпилированы с использованием Microsoft Visual C++ 6.0. Большая их часть будет работать в Windows 95, Windows 98, Windows NT 4.0 и Windows 2000, но некоторые программы требуют такую функциональность, которая поддерживается только Windows NT 4.0 и Windows 2000. Если Вы захотите самостоятельно скомпилировать какие-то примеры, Вам понадобится Microsoft Visual C++ 6.0.

В корневом каталоге компакт-диска находится общий заголовочный файл (Cmnhdr.h) и около трех десятков каталогов, в которых хранятся соответствующие программы-примеры. В каталогах x86 и Alpha32 содержатся отладочные версии тех же программ — их можно запускать прямо с компакт-диска.

Вставив компакт-диск в привод CD-ROM, Вы увидите окно Welcome. Если оно не появится, перейдите в каталог Setup на компакт-диске и запустите файл PressCDx86.exe или PressCDAlpha32.exe (в зависимости от того, какой процессор в Вашем компьютере).

Техническая поддержка

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: *Programming Applications for Microsoft Windows*, 4th ed., editor

One Microsoft Way

Redmond, WA 98052-6399

mspinput@microsoft.com

Спасибо всем за помощь

Я не смог бы написать эту книгу без помощи и содействия многих людей. Вот кого хотелось бы поблагодарить особо.

Членов редакторской группы Microsoft Press: Джека Бьюдри (Jack Beaudry), Донни Камерон (Donnie Cameron), Айни Чэнга (Ina Chang), Карла Дилтца (Carl Diltz), Стефена Гьюти (Stephen Guty), Роберта Лайена (Robert Lyon), Ребекку Мак-Кэй (Rebecca McKay), Роба Нэнса (Rob Nance), Джослин Пол (Jocelyn Paul), Шона Пека (Shawn Peck), Джона Пиэрса (John Pierce), Барб Раньян (Barb Runyan), Бена Райена (Ben Ryan), Эрика Стру (Eric Stroo) и Уильяма Тила (William Teel).

Членов группы разработчиков Windows 2000: Асмуса Фрейтара (Asmus Freytag), Дэйва Харта (Dave Hart), Ли Харт (Lee Hart), Джеффа Хейвнса (Jeff Havens), Локеша Сриниваса Копполу (Lokesh Srinivas Koppolu), Он Ли (On Lee), Скотта Людвиг (Scott Ludwig), Люю Пераццоли (Lou Perazzoli), Марка Луковски (Mark Lucovsky), Лэнди Уэнга (Landy Wang) и Стива Вуда (Steve Wood).

Членов группы разработчиков Windows 95 и Windows 98: Брайена Смита (Brian Smith), Джона Томасона (Jon Thomason) и Майкла Тутуньи (Michael Toutonghi).

Членов группы разработчиков Visual C++: Джонатана Марка (Jonathan Mark), Чака Митчела (Chuck Mitchell), Стива Солсбери (Steve Salisbury) и Дэна Спэлдинга (Dan Spalding).

Членов группы разработчиков IA-64 из корпорации Intel: Джэфа Мюррея (Geoff Murray), Хуана Родригеса (Juan Rodriguez), Джейсона Уэксмана (Jason Waxman), Койчи Ямаду (Koichi Yamada), Кита Йедлина (Keith Yedlin) и Уилфреда Ю (Wilfred Yu).

Членов группы разработчиков AXP64 из корпорации Compaq: Тома ван Баака (Tom Van Baak), Билла Бакстера (Bill Baxter), Джима Лэйна (Lim Lane), Рича Питерсона (Rich Peterson), Энни По (Annie Poh) и Джозефа Сиримарко (Joseph Sirimarco).

Членов группы разработчиков InstallShield's Installer: Боба Бэйкера (Bob Baker), Кевина Фута (Kevin Foote) и Тайлера Робинсона (Tyler Robinson).

Участников всевозможных вечеринок: Джеффа Куперстайна (Jeff Cooperstein) и Стефани (Stephanie), Кита Плиса (Keith Pleas) и Сюзан (Susan), Сюзан Рейми (Susan Ramee) и Сэнджив Сурати (Sanjeev Surati), Скотта Людвиг (Scott Ludwig) и Вал Хорвач (Val Horvath) с их сыном Николасом (Nicholas), Даррен (Darren) и Шаула Массену (Shaula Massena), Дэвида Соломона (David Solomon), Джеффа Просиза (Jeff Prosis), Джима Харкинса (Jim Harkins), Тони Спику (Tony Spika).

Членов Братства Рихтеров: Рона (Ron), Марию (Maria), Джоуи (Joey) и Брэнди (Brandi).

Основателей семьи Рихтеров: Арлин (Arlene) и Силвэна (Sylvan).

Члена фракции косматых: Макса (Max).

Члена группы поддержки: Кристин Трейс (Kristin Trace).

Ч А С Т Ь I

МАТЕРИАЛЫ ДЛЯ ОБЯЗАТЕЛЬНОГО ЧТЕНИЯ



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.

| 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 или 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. |
| LONG или DWORD | Это значение — «крепкий орешек». Функции, которые сообщают 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. |

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

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 Microsoft, 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 NO_ERROR 0L // dderror
#define SEC_E_OK ((HRESULT)0x00000000L)

//
// 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
```

см. след. стр.

```
//
// 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 21 000 строк!

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



GetLastError возвращает последнюю ошибку, возникшую в потоке. Если этот поток вызывает другую функцию Windows и все проходит успешно, код последней ошибки не перезаписывается и не используется как индикатор благополучного вызова функции. Лишь несколько функций Windows нарушают это правило и все же изменяют код последней ошибки. Однако в документации Platform SDK утверждается обратное: якобы после успешного выполнения API-функции обычно изменяют код последней ошибки.

WINDOWS 98

Многие функции Windows 98 на самом деле реализованы в 16-разрядном коде, унаследованном от операционной системы Windows 3.1. В нем не было механизма, сообщающего об ошибках через некую функцию наподобие *GetLastError*, и Microsoft не стала «исправлять» 16-разрядный код в Windows 98 для поддержки обработки ошибок. На практике это означает, что многие Win32-функции в Windows 98 не устанавливают код последней ошибки после неудачного завершения, а просто возвращают значение, которое свидетельствует об ошибке. Поэтому Вам не удастся определить причину ошибки.

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 Microsoft 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*. О других функциях см. Platform SDK.

На мой взгляд, особенно полезно отслеживать код последней ошибки в процессе отладки. Кстати, отладчик в Microsoft Visual Studio 6.0 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.

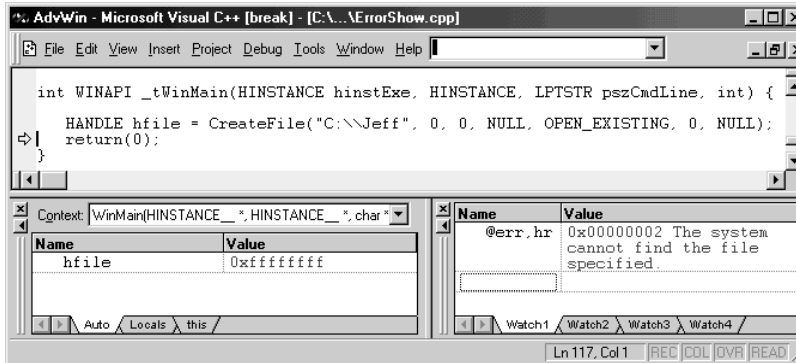
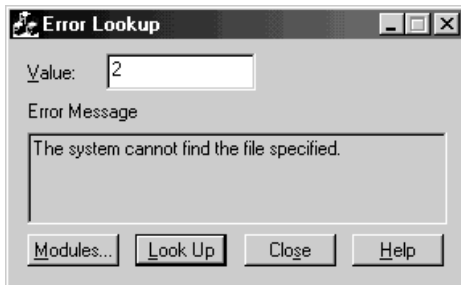


Рис. 1-1. Используя «@err.hr» в окне Watch среды Visual Studio 6.0, Вы можете просматривать код последней ошибки в текущем потоке

C 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 со 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 кон-

це главы, демонстрирует, как вызывать эту функцию для получения текстового описания ошибки по ее коду, определенному Microsoft.

Время от времени меня кто-нибудь да спрашивает, составит ли Microsoft 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 все 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 Microsoft 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

О'кэй, я показал, как функции Windows сообщают об ошибках. Microsoft 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);
```

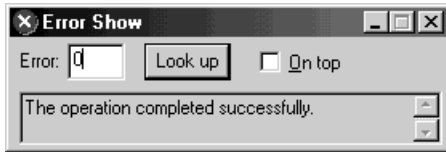
и передаете ей нужное 32-битное число. Я стараюсь использовать коды, уже определенные в WinError.h, — при условии, что они подходят под те ошибки, о которых могут сообщать мои функции. Если Вы считаете, что ни один из кодов в WinError.h не годится для ошибки, возможной в Вашей функции, определите свой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

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

Подробнее об этих полях я рассказываю в главе 24. На данный момент единственное важное для Вас поле — бит 29. Microsoft 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 Microsoft, — ни сейчас, ни в будущем.

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. 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_ERRORCODE, NULL, FALSE);

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

// получаем текстовое описание ошибки
BOOL fOk = FormatMessage(
    FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
    NULL, dwError, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
    (LPTSTR) &hlocal, 0, NULL);

:

if (hlocal != NULL) {
    SetDlgItemText(hwnd, IDC_ERRORTXT, (PCTSTR) LocalLock(hlocal));
    LocalFree(hlocal);
} else {
    SetDlgItemText(hwnd, IDC_ERRORTXT, TEXT("Error number not found."));
}
```

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 описание.

Вызывая *FormatMessage*, я передаю флаг *FORMAT_MESSAGE_FROM_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*. Третий параметр указывает код интересующей нас ошибки, а четвертый — язык, на котором мы хотим увидеть ее описание.

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. 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.



ErrorShow.cpp

```

/*****
Модуль: ErrorShow.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h" /* см. приложение A */
#include <Windowsx.h>
#include <tchar.h>
#include "Resource.h"

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

#define ESM_POKECODEANDLOOKUP (WM_USER + 100)
const TCHAR g_szAppName[] = TEXT("Error Show");

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

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

    chSETDLGICONS(hwnd, IDI_ERRORSHOW);

    // не принимаем коды ошибок, состоящие более чем из 5 цифр
    Edit_LimitText(GetDlgItem(hwnd, IDC_ERRORCODE), 5);

    // проверяем, не передан ли код ошибки через командную строку
    SendMessage(hwnd, ESM_POKECODEANDLOOKUP, lParam, 0);
    return(TRUE);
}

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

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {

        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case IDC_ALWAYSONTOP:
            SetWindowPos(hwnd, IsDlgButtonChecked(hwnd, IDC_ALWAYSONTOP)
                ? HWND_TOPMOST : HWND_NOTOPMOST, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
            break;
    }
}

```

Рис. 1-2. Программа-пример ErrorShow

Рис. 1-2. продолжение

```

case IDC_ERRORCODE:
    EnableWindow(GetDlgItem(hwnd, IDOK), Edit_GetTextLength(hwndCtl) > 0);
    break;

case IDOK:
    // получаем код ошибки
    DWORD dwError = GetDlgItemInt(hwnd, IDC_ERRORCODE, NULL, FALSE);

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

    // получаем текстовое описание ошибки
    BOOL fOk = FormatMessage(
        FORMAT_MESSAGE_FROM_SYSTEM | FORMAT_MESSAGE_ALLOCATE_BUFFER,
        NULL, dwError, MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US),
        (PTSTR) &hlocal, 0, NULL);

    if (!fOk) {
        // не связана ли ошибка с сетью?
        HMODULE hDll = LoadLibraryEx(TEXT("netmsg.dll"), NULL,
            DONT_RESOLVE_DLL_REFERENCES);

        if (hDll != NULL) {
            FormatMessage(
                FORMAT_MESSAGE_FROM_HMODULE | FORMAT_MESSAGE_FROM_SYSTEM
                | FORMAT_MESSAGE_ALLOCATE_BUFFER, hDll, dwError,
                MAKELANGID(LANG_ENGLISH, SUBLANG_ENGLISH_US), (PTSTR) &hlocal, 0, NULL);
            FreeLibrary(hDll);
        }
    }

    if (hlocal != NULL) {
        SetDlgItemText(hwnd, IDC_ERRORTXT, (PCTSTR) LocalLock(hlocal));
        LocalFree(hlocal);
    } else {
        SetDlgItemText(hwnd, IDC_ERRORTXT, TEXT("Error number not found."));
    }
    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);

    case ESM_POKECODEANDLOOKUP:
        SetDlgItemInt(hwnd, IDC_ERRORCODE, (UINT) wParam, FALSE);
        FORWARD_WM_COMMAND(hwnd, IDOK, GetDlgItem(hwnd, IDOK), BN_CLICKED,

```

см. след. стр.

Рис. 1-2. *продолжение*

```

        PostMessage);
        SetForegroundWindow(hwnd);
        break;
    }

    return(FALSE);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    HWND hwnd = FindWindow(TEXT("#32770"), TEXT("Error Show"));
    if (IsWindow(hwnd)) {
        // экземпляр уже выполняется, активизируем его и посылаем ему новый номер
        SendMessage(hwnd, ESM_POKECODEANDLOOKUP, _ttoi(pszCmdLine), 0);
    } else {
        DialogBoxParam(hinstExe, MAKEINTRESOURCE(IDD_ERRORSHOW),
            NULL, Dlg_Proc, _ttoi(pszCmdLine));
    }
    return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

Unicode

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.

Я решил переместить эту главу в начало книги, потому что вопрос о поддержке Unicode стал одним из основных при разработке любого приложения. Проблемы, связанные с Unicode, обсуждаются почти в каждой главе; все программы-примеры в моей книге «готовы к Unicode». Тот, кто пишет программы для Microsoft Windows 2000 или Microsoft Windows CE, просто обязан использовать Unicode, и точка. Но если Вы разрабатываете приложения для Microsoft Windows 98, у Вас еще есть выбор. В этой главе мы поговорим и о применении Unicode в Windows 98.

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).

Одно- и двухбайтовые наборы символов

В двухбайтовом наборе символ представляется либо одним, либо двумя байтами. Так, для японской каны, 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.

Простой вызов функции *strlen* не дает количества символов в строке — она возвращает только число байтов. В ANSI-библиотеке C нет функций, работающих с двухбайтовыми наборами символов. Но в аналогичную библиотеку Visual C++ включено множество функций (типа *mbslen*), способных оперировать со строками мультбайтовых (как одно-, так и двухбайтовых) символов.

Для работы с DBCS-строками в Windows предусмотрен целый набор вспомогательных функций:

| Функция | Описание |
|---|---|
| <i>PTSTR CharNext (PCTSTR pszCurrentChar);</i> | Возвращает адрес следующего символа в строке |
| <i>PTSTR CharPrev (PCTSTR pszStart, PCTSTR pszCurrentChar);</i> | Возвращает адрес предыдущего символа в строке |
| <i>BOOL IsDBCSLeadByte (BYTE bTestChar);</i> | Возвращает TRUE, если данный байт — первый в DBCS-символе |

Функции *CharNext* и *CharPrev* позволяют «перемещаться» по двухбайтовой строке одновременно на 1 символ вперед или назад, а *IsDBCSLeadByte* возвращает TRUE, если переданный ей байт — первый в двухбайтовом символе.

Хотя эти функции несколько облегчают работу с DBCS-строками, необходимость в ином подходе очевидна. Перейдем к Unicode.

Unicode: набор «широких» символов

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**).

Строки в Unicode просты и логичны. Все символы в них представлены 16-битными значениями (по 2 байта на каждый). В них нет особых байтов, указывающих, чем является следующий байт — частью того же символа или новым символом. Это значит, что прохождение по строке реализуется простым увеличением или уменьшением значения указателя. Функции *CharNext*, *CharPrev* и *IsDBCSLeadByte* больше не нужны.

Так как каждый символ — 16-битное число, Unicode позволяет кодировать 65 536 символов, что более чем достаточно для работы с любым языком. Разительное отличие от 256 знаков, доступных в однобайтовом наборе!

В настоящее время кодовые позиции¹ определены для арабского, китайского, греческого, еврейского, латинского (английского) алфавитов, а также для кириллицы (русского), японской каны, корейского хангыль и некоторых других алфавитов. Кроме того, в набор символов включено большое количество знаков препинания, математических и технических символов, стрелок, диакритических и других знаков. Все вместе они занимают около 35 000 кодовых позиций, оставляя простор для будущих расширений.

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

¹ Кодовая позиция (code point) — позиция знака в наборе символов.

| 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 | Еврейский |
| 0250–02AF | Стандартные фонетические | 0600–06FF | Арабский |
| 02B0–02FF | Модифицированные литеры | 0900–097F | Деванагари |

Около 29 000 кодовых позиций пока не заняты, но зарезервированы на будущее. Примерно 6 000 позиций оставлено специально для программистов (на их усмотрение).

Почему Unicode?

Разрабатывая приложение, Вы определенно должны использовать преимущества Unicode. Даже если Вы пока не собираетесь локализовать программный продукт, разработка с прицелом на Unicode упростит эту задачу в будущем. Unicode также позволяет:

- легко обмениваться данными на разных языках;
- распространять единственный двоичный EXE- или DLL-файл, поддерживающий все языки;
- увеличить эффективность приложений (об этом мы поговорим чуть позже).

Windows 2000 и Unicode

Windows 2000 — операционная система, целиком и полностью построенная на Unicode. Все базовые функции для создания окон, вывода текста, операций со строками и т. д. ожидают передачи Unicode-строк. Если какой-то функции Windows передается ANSI-строка, она сначала преобразуется в Unicode и лишь потом передается операционной системе. Если Вы ждете результата функции в виде ANSI-строки, операционная система преобразует строку — перед возвратом в приложение — из Unicode в ANSI. Все эти преобразования протекают скрытно от Вас, но, конечно, на них тратятся и лишнее время, и лишняя память.

Например, функция *CreateWindowEx*, вызываемая с ANSI-строками для имени класса и заголовка окна, должна, выделив дополнительные блоки памяти (в стандартной куче Вашего процесса), преобразовать эти строки в Unicode и, сохранив результат в выделенных блоках памяти, вызвать Unicode-версию *CreateWindowEx*.

Для функций, заполняющих строками выделенные буферы, системе — прежде чем программа сможет их обрабатывать — нужно преобразовать строки из Unicode в ANSI. Из-за этого Ваше приложение потребует больше памяти и будет работать медленнее. Поэтому гораздо эффективнее разрабатывать программу, с самого начала ориентируясь на Unicode.

Windows 98 и Unicode

Windows 98 — не совсем новая операционная система. У нее «16-разрядное наследство», которое не было рассчитано на Unicode. Введение поддержки Unicode в Windows 98 было бы слишком трудоемкой задачей, и при разработке этой операционной системы от нее отказались. По этой причине вся внутренняя обработка строк в Windows 98, как и у ее предшественниц, построена на применении ANSI.

И все же Windows 98 допускает работу с приложениями, обрабатывающими символы и строки в Unicode, хотя вызов функций Windows при этом заметно усложняется. Например, если Вы, обращаясь к *CreateWindowEx*, передаете ей ANSI-строки, вызов проходит очень быстро — не требуется ни выделения буферов, ни преобразования строк. Но для вызова *CreateWindowEx* с Unicode-строками Вам придется самому выделять буферы, явно вызывать функции, преобразующие строки из Unicode в ANSI, обращаться к *CreateWindowEx*, снова вызывать функции, преобразующие строки — на этот раз из ANSI в Unicode, и освобождать временные буферы. Так что в Windows 98 работать с Unicode не столь удобно, как в Windows 2000. Подробнее о преобразованиях строк в Windows 98 я расскажу в конце главы.

Хотя большинство Unicode-функций в Windows 98 ничего не делает, некоторые все же реализованы. Вот они:

- | | |
|--------------------------|-------------------------|
| ■ EnumResourceLanguagesW | ■ GetTextExtentPoint32W |
| ■ EnumResourceNamesW | ■ GetTextExtentPointW |
| ■ EnumResourceTypesW | ■ lstrlenW |
| ■ ExtTextOutW | ■ MessageBoxExW |
| ■ FindResourceW | ■ MessageBoxW |
| ■ FindResourceExW | ■ TextOutW |
| ■ GetCharWidthW | ■ WideCharToMultiByte |
| ■ GetCommandLineW | ■ MultiByteToWideChar |

К сожалению, многие из этих функций в Windows 98 работают из рук вон плохо. Одни не поддерживают определенные шрифты, другие повреждают область динамически распределяемой памяти (кучу), третьи нарушают работу принтерных драйверов и т. д. С этими функциями Вам придется здорово потрудиться при отладке программы. И даже это еще не значит, что Вы сможете устранить все проблемы.

Windows CE и Unicode

Операционная система Windows CE создана для небольших вычислительных устройств — бездисковых и с малым объемом памяти. Вы вполне могли бы подумать, что Microsoft, раз уж эту систему нужно было сделать предельно компактной, в качестве «родного» набора символов выберет ANSI. Но Microsoft поступила дальновиднее. Зная, что вычислительные устройства с Windows CE будут продаваться по всему миру, там решили сократить затраты на разработку программ, упростив их локализацию. Поэтому Windows CE полностью поддерживает Unicode.

Чтобы не увеличивать ядро Windows CE, Microsoft вообще отказалась от поддержки ANSI-функций Windows. Так что, если Вы пишете для Windows CE, то просто обязаны разбираться в Unicode и использовать его во всех частях своей программы.

В чью пользу счет?

Для тех, кто ведет счет в борьбе Unicode против ANSI, я решил сделать краткий обзор «История Unicode в Microsoft»:

- Windows 2000 поддерживает Unicode и ANSI — Вы можете использовать любой стандарт;
- Windows 98 поддерживает только ANSI — Вы обязаны программировать в расчете на ANSI;

- Windows CE поддерживает только Unicode — Вы обязаны программировать в расчете на Unicode.

Несмотря на то что Microsoft пытается облегчить написание программ, способных работать на всех трех платформах, различия между Unicode и ANSI все равно создают проблемы, и я сам не раз с ними сталкивался. Не поймите меня неправильно, но Microsoft твердо поддерживает Unicode, поэтому я настоятельно рекомендую переходить именно на этот стандарт. Только имейте в виду, что Вас ждут трудности, на преодоление которых потребуется время. Я бы посоветовал применять Unicode и, если Вы работаете в Windows 98, преобразовывать строки в ANSI лишь там, где без этого не обойтись.

Увы, есть еще одна маленькая проблема, о которой Вы должны знать, — COM.

Unicode и COM

Когда Microsoft переносила COM из 16-разрядной Windows на платформу Win32, руководство этой компании решило, что все методы COM-интерфейсов, работающие со строками, должны принимать их только в Unicode. Это было удачное решение, так как COM обычно используется для того, чтобы компоненты могли общаться друг с другом, а Unicode позволяет легко локализовать строки.

Если Вы разрабатываете программу для Windows 2000 или Windows CE и при этом используете COM, то выбора у Вас просто нет. Применяя Unicode во всех частях программы, Вам будет гораздо проще обращаться и к операционной системе, и к COM-объектам.

Если Вы пишете для Windows 98 и тоже используете COM, то попадаете в затруднительное положение. COM требует строк в Unicode, а большинство функций операционной системы — строк в ANSI. Это просто кошмар! Я работал над несколькими такими проектами, и мне приходилось писать прорву кода только для того, чтобы гонять строки из одного формата в другой.

Как писать программу с использованием Unicode

Microsoft разработала Windows API так, чтобы как можно меньше влиять на Ваш код. В самом деле, у Вас появилась возможность создать единственный файл с исходным кодом, компилируемый как с применением Unicode, так и без него, — достаточно определить два макроса (UNICODE и _UNICODE), которые отвечают за нужные изменения.

Unicode и библиотека C

Для использования Unicode-строк были введены некоторые новые типы данных. Стандартный заголовочный файл `String.h` модифицирован: в нем определен `wchar_t` — тип данных для Unicode-символа:

```
typedef unsigned short wchar_t;
```

Если Вы хотите, скажем, создать буфер для хранения Unicode-строки длиной до 99 символов с нулевым символом в конце, поставьте оператор:

```
wchar_t szBuffer[100];
```

Он создает массив из ста 16-битных значений. Конечно, стандартные функции библиотеки C для работы со строками вроде *strcpy*, *strchr* и *strcat* оперируют только с ANSI-строками — они не способны корректно обрабатывать Unicode-строки. Поэто-

му в ANSI C имеется дополнительный набор функций. На рис. 2-1 приведен список строчковых функций ANSI C и эквивалентных им Unicode-функций.

```
char * strcat(char *, const char *);
wchar_t * wcscat(wchar_t *, const wchar_t *);

char * strchr(const char *, int);
wchar_t * wcschr(const wchar_t *, wchar_t);

int strcmp(const char *, const char *);
int wcscmp(const wchar_t *, const wchar_t *);

char * strcpy(char *, const char *);
wchar_t * wcsncpy(wchar_t *, const wchar_t *);

size_t strlen(const char *);
size_t wcslen(const wchar_t *);
```

Рис. 2-1. Строчковые функции ANSI C и их Unicode-аналоги

Обратите внимание, что имена всех новых функций начинаются с *wcs* — это аббревиатура *wide character set* (набор широких символов). Таким образом, имена Unicode-функций образуются простой заменой префикса *str* соответствующих ANSI-функций на *wcs*.



Один очень важный момент, о котором многие забывают, заключается в том, что библиотека C, предоставляемая Microsoft, отвечает стандарту ANSI. А он требует, чтобы библиотека C поддерживала символы и строки в Unicode. Это значит, что Вы можете вызывать функции C для работы с Unicode-символами и строками даже при работе в Windows 98. Иными словами, функции *wcscat*, *wcslen*, *wcstok* и т. д. прекрасно работают и в Windows 98; беспокоиться нужно за функции операционной системы.

Код, содержащий явные вызовы *str*- или *wcs*-функций, просто так компилировать с использованием и ANSI, и Unicode нельзя. Чтобы реализовать возможность компиляции «двойного назначения», замените в своей программе заголовочный файл *String.h* на *TChar.h*. Он помогает создавать универсальный исходный код, способный задеятьствовать как ANSI, так и Unicode, — и это единственное, для чего нужен файл *TChar.h*. Он состоит из макросов, заменяющих явные вызовы *str*- или *wcs*-функций. Если при компиляции исходного кода Вы определяете *_UNICODE*, макросы ссылаются на *wcs*-функции, а в его отсутствие — на *str*-функции.

Например, в *TChar.h* есть макрос *_tcsncpy*. Если Вы включили этот заголовочный файл, но *_UNICODE* не определен, *_tcsncpy* раскрывается в ANSI-функцию *strcpy*, а если *_UNICODE* определен — в Unicode-функцию *wcsncpy*. В файле *TChar.h* определены универсальные макросы для всех стандартных строчковых функций C. При использовании этих макросов вместо конкретных имен ANSI- или Unicode-функций Ваш код можно будет компилировать в расчете как на Unicode, так и на ANSI.

Но, к сожалению, это еще не все. В файле *TChar.h* есть дополнительные макросы.

Чтобы объявить символьный массив «универсального назначения» (ANSI/Unicode), применяется тип данных *TCHAR*. Если *_UNICODE* определен, *TCHAR* объявляется так:

```
typedef wchar_t TCHAR;
```

А если *_UNICODE* не определен, то:

```
typedef char TCHAR;
```

Используя этот тип данных, можно объявить строку символов как:

```
TCHAR szString[100];
```

Можно также определять указатели на строки:

```
TCHAR *szError = "Error";
```

Правда, в этом операторе есть одна проблема. По умолчанию компилятор Microsoft C++ транслирует строки как состоящие из символов ANSI, а не Unicode. В итоге этот оператор нормально компилируется, если `_UNICODE` не определен, но в ином случае дает ошибку. Чтобы компилятор сгенерировал Unicode-, а не ANSI-строку, оператор надо переписать так:

```
TCHAR *szError = L"Error";
```

Заглавная буква *L* перед строковым литералом указывает компилятору, что его надо обрабатывать как Unicode-строку. Тогда, размещая строку в области данных программы, компилятор вставит между всеми символами нулевые байты. Но возникает другая проблема — программа компилируется, только если `_UNICODE` определен. Следовательно, нужен макрос, способный избирательно ставить *L* перед строковым литералом. Эту работу выполняет макрос `_TEXT`, также содержащийся в `Tchar.h`. Если `_UNICODE` определен, `_TEXT` определяется как:

```
#define _TEXT(x)    L ## x
```

В ином случае `_TEXT` определяется следующим образом:

```
#define _TEXT(x)    x
```

Используя этот макрос, перепишем злополучный оператор так, чтобы его можно было корректно компилировать независимо от того, определен `_UNICODE` или нет:

```
TCHAR *szError = _TEXT("Error");
```

Макрос `_TEXT` применяется и для символьных литералов. Например, чтобы проверить, является ли первый символ строки заглавной буквой *J*:

```
if (szError[0] == _TEXT('J')) {
    // первый символ - J
    :
} else {
    // первый символ - не J
    :
}
```

Типы данных, определенные в Windows для Unicode

В заголовочных файлах Windows определены следующие типы данных.

| Тип данных | Описание |
|------------|--|
| WCHAR | Unicode-символ |
| PWSTR | Указатель на Unicode-строку |
| PCWSTR | Указатель на строковую константу в Unicode |

Эти типы данных относятся исключительно к символам и строкам в кодировке Unicode. В заголовочных файлах Windows определены также универсальные (ANSI/

Unicode) типы данных PTSTR и PCTSTR, указывающие — в зависимости от того, определен ли при компиляции макрос UNICODE, — на ANSI- или на Unicode-строку.

Кстати, на этот раз имя макроса UNICODE не предваряется знаком подчеркивания. Дело в том, что макрос _UNICODE используется в заголовочных файлах библиотеки C, а макрос UNICODE — в заголовочных файлах Windows. Для компиляции модулей исходного кода обычно приходится определять оба макроса.

Unicode- и ANSI-функции в Windows

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,
    PCWSTR pWindowName,
    DWORD dwStyle,
    int X,
    int Y,
    int nWidth,
    int nHeight,
    HWND hWndParent,
    HMENU hMenu,
    HINSTANCE hInstance,
    PVOID pParam);
```

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

CreateWindowExW — это Unicode-версия. Буква *W* в 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 // !UNICODE
```

Какая именно версия *CreateWindowEx* будет вызвана, зависит от того, определен ли UNICODE в период компиляции. Перенос 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 2000 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.

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.)

В Windows 98 основную работу выполняет *CreateWindowExA*. В этой операционной системе предусмотрены точки входа для всех функций Windows, принимающих Unicode-строки, но функции не транслируют их в ANSI, а просто сообщают об ошибке. Последующий вызов *GetLastError* дает `ERROR_CALL_NOT_IMPLEMENTED`. Должным образом действуют только ANSI-версии функций. Ваше приложение не будет работать в Windows 98, если в скомпилированном коде присутствуют вызовы «широкосимвольных» функций.

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.

Строковые функции Windows

Windows предлагает внушительный набор функций, работающих со строками. Они похожи на строковые функции из библиотеки C, например на *strcpy* и *wcsncpy*. Однако функции Windows являются частью операционной системы, и многие ее компоненты используют именно их, а не аналоги из библиотеки C. Я советую отдать предпочтение функциям операционной системы. Это немного повысит быстродействие Вашего приложения. Дело в том, что к ним часто обращаются такие тяжеловесные процессы, как оболочка операционной системы (*Explorer.exe*), и скорее всего эти функции будут загружены в память еще до запуска Вашего приложения.

Данные функции доступны в Windows 2000 и Windows 98. Но Вы сможете вызывать их и в более ранних версиях Windows, если установите Internet Explorer версии 4.0 или выше.

По классической схеме именования функций в операционных системах их имена состоят из символов нижнего и верхнего регистра и выглядят так: *StrCat*, *StrChr*, *StrCmp*, *StrCpy* и т. д. Для использования этих функций включите в программу заголовочный файл *ShlWApi.h*. Кроме того, как я уже говорил, каждая строковая функция существует в двух версиях — для ANSI и для Unicode (например, *StrCatA* и *StrCatW*). Поскольку это функции операционной системы, их имена автоматически преобразуются в нужную форму, если в исходном тексте Вашей программы перед ее сборкой определен UNICODE.

Создание программ, способных использовать и ANSI, и Unicode

Неплохая мысль — заранее подготовить свое приложение к Unicode, даже если Вы пока не планируете работать с этой кодировкой. Вот главное, что для этого нужно:

- привыкайте к тому, что текстовые строки — это массивы символов, а не массивы байтов или значений типа *char*;
- используйте универсальные типы данных (вроде TCHAR или PTSTR) для текстовых символов и строк;
- используйте явные типы данных (вроде BYTE или PBYTE) для байтов, указателей на байты и буферов данных;
- применяйте макрос _TEXT для определения символьных и строковых литералов;
- предусмотрите возможность глобальных замен (например, PSTR на PTSTR);
- модифицируйте логику строковой арифметики. Например, функции обычно принимают размер буфера в символах, а не в байтах. Это значит, что вместо *sizeof(szBuffer)* Вы должны передавать *(sizeof(szBuffer) / sizeof(TCHAR))*. Но блок памяти для строки известной длины выделяется в байтах, а не символах, т. е. вместо *malloc(nCharacters)* нужно использовать *malloc(nCharacters * sizeof(TCHAR))*. Из всего, что я перечислил, это запомнить труднее всего — если Вы ошибетесь, компилятор не выдаст никаких предупреждений.

Разрабатывая программы-примеры для первого издания книги, я сначала написал их так, что они компилировались только с использованием ANSI. Но, дойдя до этой главы (она была тогда в конце), понял, что Unicode лучше, и решил написать примеры, которые показывали бы, как легко создавать программы, компилируемые с применением и Unicode, и ANSI. В конце концов я преобразовал все программы-примеры так, чтобы их можно было компилировать в расчете на любой из этих стандартов.

Конверсия всех программ заняла примерно 4 часа — неплохо, особенно если учесть, что у меня совсем не было опыта в этом деле.

В Windows есть набор функций для работы с Unicode-строками. Эти функции перечислены ниже.

| Функция | Описание |
|-----------------|---|
| <i>lstrcat</i> | Выполняет конкатенацию строк |
| <i>lstrcmp</i> | Сравнивает две строки с учетом регистра букв |
| <i>lstrcmpi</i> | Сравнивает две строки без учета регистра букв |
| <i>lstrcpy</i> | Копирует строку в другой участок памяти |
| <i>lstrlen</i> | Возвращает длину строки в символах |

Они реализованы как макросы, вызывающие либо Unicode-, либо ANSI-версию функции в зависимости от того, определен ли UNICODE при компиляции исходного модуля. Например, если UNICODE не определен, *lstrcat* раскрывается в *lstrcatA*, определен — в *lstrcatW*.

Строковые функции *lstrcmp* и *lstrcmpi* ведут себя не так, как их аналоги из библиотеки C (*strcmp*, *strcmpi*, *wscmp* и *wscmpi*), которые просто сравнивают кодовые позиции в символах строк. Игнорируя фактические символы, они сравнивают числовое значение каждого символа первой строки с числовым значением символа второй

строки. Но *lstrcmp* и *lstrcmpi* реализованы через вызовы Windows-функции *CompareString*:

```
int CompareString(
    LCID lcid,
    DWORD fdwStyle,
    PCWSTR pString1,
    int cch1,
    PCWSTR pString2,
    int cch2);
```

Она сравнивает две Unicode-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 C.

Когда любая из функций семейства *lstrcmp* вызывает *CompareString*, в первом параметре передается результат вызова Windows-функции *GetThreadLocale*:

```
LCID GetThreadLocale();
```

Она возвращает уже упомянутый идентификатор, который назначается потоку в момент его создания.

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

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

Вызывая *CompareString*, функция *lstrcmp* передает в параметре *fdwStyle* нуль, а *lstrcmpi* — флаг NORM_IGNORECASE. Остальные четыре параметра определяют две строки и их длину. Если *cch1* равен –1, функция считает, что строка *pString1* завершается нулевым символом, и автоматически вычисляет ее длину. То же относится и к параметрам *cch2* и *pString2*.

Многие функции C-библиотеки с Unicode-строками толком не работают. Так, *tolower* и *toupper* неправильно преобразуют регистр букв со знаками ударения. Поэтому для Unicode-строк лучше использовать соответствующие Windows-функции. К тому же они корректно работают и с ANSI-строками.

Первые две функции:

```
PTSTR CharLower(PTSTR pszString);
```

```
PTSTR CharUpper(PTSTR pszString);
```

преобразуют либо отдельный символ, либо целую строку с нулевым символом в конце. Чтобы преобразовать всю строку, просто передайте ее адрес. Но, преобразуя отдельный символ, Вы должны передать его так:

```
TCHAR cLowerCaseChr = CharLower((PTSTR) szString[0]);
```

Приведение типа отдельного символа к PTSTR вызывает обнуление старших 16 битов передаваемого значения, а в его младшие 16 битов помещается сам символ. Обнаружив, что старшие 16 битов этого значения равны 0, функция «поймет», что Вы хотите преобразовать не строку, а отдельный символ. Возвращаемое 32-битное значение содержит результат преобразования в младших 16 битах.

Следующие две функции аналогичны двум предыдущим за исключением того, что они преобразуют символы, содержащиеся в буфере (который не требуется завершать нулевым символом):

```
DWORD CharLowerBuff(
    PTSTR pszString,
    DWORD cchString);
```

```
DWORD CharUpperBuff(
    PTSTR pszString,
    DWORD cchString);
```

Прочие функции библиотеки C (например, *isalpha*, *islower* и *isupper*) возвращают значение, которое сообщает, является ли данный символ буквой, а также строчная она или прописная. В Windows API тоже есть подобные функции, но они учитывают и язык, выбранный пользователем в Control Panel:

```
BOOL IsCharAlpha(TCHAR ch);
BOOL IsCharAlphaNumeric(TCHAR ch);
BOOL IsCharLower(TCHAR ch);
BOOL IsCharUpper(TCHAR ch);
```

И последняя группа функций из библиотеки C, о которых я хотел рассказать, — *printf*. Если при компиляции `_UNICODE` определен, они ожидают передачи всех символьных и строковых параметров в Unicode; в ином случае — в ANSI.

Microsoft ввела в семейство функций *printf* своей C-библиотеки дополнительные типы полей, часть из которых не поддерживается в ANSI C. Они позволяют легко сравнивать и смешивать символы и строки с разной кодировкой. Также расширена функция *wsprintf* операционной системы. Вот несколько примеров (обратите внимание на использование буквы *s* в верхнем и нижнем регистре):

```
char  szA[100];  // строковый буфер в ANSI
WCHAR szW[100];  // строковый буфер в Unicode

// обычный вызов sprintf: все строки в ANSI
sprintf(szA, "%s", "ANSI Str");

// преобразуем строку из Unicode в ANSI
sprintf(szA, "%S", L"Unicode Str");

// обычный вызов swprintf: все строки в Unicode
swprintf(szW, L"%s", L"Unicode Str");

// преобразуем строку из ANSI в Unicode
swprintf(szW, L"%S", "ANSI Str");
```


Ресурсы

Компилятор ресурсов генерирует двоичное представление всех ресурсов, используемых Вашей программой. Строки в ресурсах (таблицы строк, шаблоны диалоговых окон, меню и др.) всегда записываются в Unicode. Если в программе не определяется макрос UNICODE, Windows 98 и Windows 2000 сами проводят нужные преобразования. Например, если при компиляции исходного модуля UNICODE не определен, вызов *LoadString* на самом деле приводит к вызову *LoadStringA*, которая читает строку из ресурсов и преобразует ее в ANSI. Затем Вашей программе возвращается ANSI-представление строки.

Текстовые файлы

Текстовых файлов в кодировке Unicode пока очень мало. Ни в одном текстовом файле, поставляемом с операционными системами или другими программными продуктами Microsoft, не используется Unicode. Думаю, однако, что эта тенденция изменится в будущем — пусть даже в отдаленном. Например, программа Notepad в Windows 2000 позволяет создавать или открывать как Unicode-, так и ANSI-файлы. Посмотрите на ее диалоговое окно Save As (рис. 2-2) и обратите внимание на предлагаемые форматы текстовых файлов.

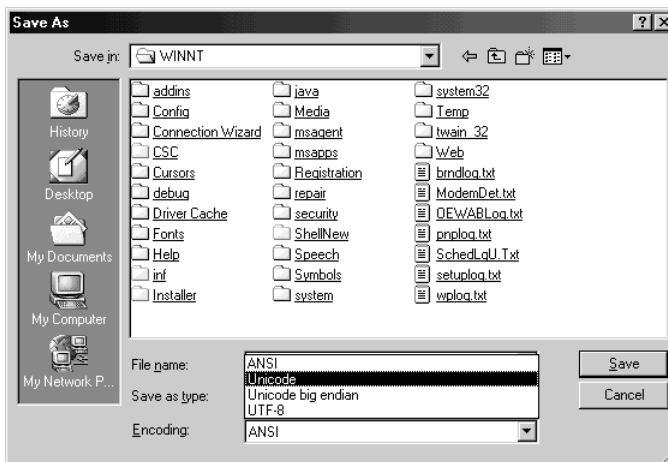


Рис. 2-2. Диалоговое окно *Save As* программы Notepad в Windows 2000

Многим приложениям, которые открывают и обрабатывают текстовые файлы (например, компиляторам), было бы удобнее, если после открытия файла можно было бы определить, содержит он символы в ANSI или в Unicode. В этом может помочь функция *IsTextUnicode*:

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

Проблема с текстовыми файлами в том, что не существует четких и строгих правил относительно их содержимого. Это крайне затрудняет определение того, содержит файл символы в ANSI или в Unicode. Поэтому *IsTextUnicode* применяет набор статистических и детерминистских методов для того, чтобы сделать взвешенное предположение о содержимом буфера. Поскольку тут больше алхимии, чем точной науки, нет гарантий, что Вы не получите неверные результаты от *IsTextUnicode*.

Первый ее параметр, *pvBuffer*, указывает на буфер, подлежащий проверке. При этом используется указатель типа *void*, поскольку неизвестно, в какой кодировке данный массив символов.

Параметр *cb* определяет число байтов в буфере, на который указывает *pvBuffer*. Так как содержимое буфера не известно, *cb* — счетчик именно байтов, а не символов. Заметьте: вовсе не обязательно задавать всю длину буфера. Но чем больше байтов проанализирует функция, тем больше шансов получить правильный результат.

Параметр *pResult* — это адрес целочисленной переменной, которую надо инициализировать перед вызовом функции. Ее значение сообщает, какие тесты должна провести *IsTextUnicode*. Если *pResult* равен NULL, функция *IsTextUnicode* делает все проверки. (Подробнее об этом см. документацию Platform SDK.)

Функция возвращает TRUE, если считает, что буфер содержит текст в Unicode, и FALSE — в ином случае. Да-да, она возвращает именно булево значение, хотя в прототипе указано DWORD. Если через целочисленную переменную, на которую указывает *pResult*, были запрошены лишь определенные тесты, функция (перед возвратом управления) устанавливает ее биты в соответствии с результатами этих тестов.

WINDOWS 98 В Windows 98 функция *IsTextUnicode* по сути не реализована и просто возвращает FALSE; последующий вызов *GetLastError* дает код ошибки ERROR_CALL_NOT_IMPLEMENTED.

Применение функции *IsTextUnicode* иллюстрирует программа-пример FileRev (см. главу 17).

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 cchMultiByte,
    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. 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. Обычно конверсия мультибайтовой строки в ее Unicode-эквивалент проходит так:

1. Вызывают *MultiByteToWideChar*, передавая NULL в параметре *pWideCharStr* и 0 в параметре *cchWideChar*.
2. Выделяют блок памяти, достаточный для сохранения преобразованной строки. Его размер получают из предыдущего вызова *MultiByteToWideChar*.

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 cchMultiByte,
    PCSTR pDefaultChar,
    PBOOL 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 *pWideCharStr* 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.

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.

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) {

    // получаем указатель на последний символ в строке
    PWSTR pEndOfStr = pWideCharStr + wcslen(pWideCharStr) - 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
        // последнего символа
        *pEndOfStr = cCharT;

        // продвигаемся на 1 символ вправо
        pWideCharStr++;

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

    // строка обращена; сообщаем об успешном завершении
    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) {
    PWSTR pWideCharStr;
    int nLenOfWideCharStr;
    BOOL fOk = FALSE;

    // вычисляем количество символов, необходимых
    // для хранения широкосимвольной версии строки
    nLenOfWideCharStr = MultiByteToWideChar(CP_ACP, 0,
        pMultiByteStr, -1, NULL, 0);

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

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

```

```
// преобразуем мультибайтовую строку в широкосимвольную
MultiByteToWideChar(CP_ACP, 0, pMultiByteStr, -1,
    pWideCharStr, nLenOfWideCharStr);

// вызываем широкосимвольную версию этой функции
// для выполнения настоящей работы
fOk = StringReverseW(pWideCharStr);

if (fOk) {
    // преобразуем широкосимвольную строку обратно в мультибайтовую
    WideCharToMultiByte(CP_ACP, 0, pWideCharStr, -1,
        pMultiByteStr, 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);
BOOL StringReverseA (PSTR pMultiByteStr);

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

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 рутинной. 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). Эти объекты создаются Windows-функциями. Например, *CreateFileMapping* заставляет систему сформировать объект «проекция файла». 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 большая 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 Microsoft ввела намеренно, чтобы ни одна программа не нарушила целостность структур объектов ядра. Это же ограничение позволяет Microsoft 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 с 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. 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 Microsoft сделала так, чтобы значения описателей зависели от конкретного процесса. Поэтому, если Вы передаете 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; создавая клиентское приложение, Вы можете игнорировать это свойство объектов ядра.

WINDOWS 98 В Windows 98 дескрипторы защиты отсутствуют, так как она не предназначена для выполнения серверных приложений. Тем не менее Вы должны знать о тонкостях, связанных с защитой, и реализовать соответствующие механизмы, чтобы Ваше приложение корректно работало и в Windows 2000.

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 dwMaximumSizeLow,
    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,
    PAGE_READWRITE, 0, 1024, "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, "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, проверяет тип защиты объекта. Если меня, как зарегистрировавшегося пользователя, допускают к существующему 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.

WINDOWS 98

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 Windows 98, в Windows 2000 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 Windows 98 создавались без учета специфики Windows 2000. Поскольку Windows 98 не защищает свой реестр, разработчики часто вызывали *RegOpenKeyEx* со значением *KEY_ALL_ACCESS*. Так проще и не надо ломать голову над тем, какой уровень доступа требуется на самом деле. Но проблема в том, что раздел реестра может быть доступен для чтения и заблокирован для записи. В Windows 2000 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 перенести многие приложения Windows 98 в Windows 2000.

Кроме объектов ядра Ваша программа может использовать объекты других типов — меню, окна, курсоры 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);
```

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 и к тому же эта таблица по-разному реализуется в Windows 2000, Windows 98 и Windows CE. 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.

| 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??????? |
| ... | ... | ... | ... |

Таблица 3-1. Структура таблицы описателей, принадлежащей процессу

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, 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,
    DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD dwCreationFlags,
    PDWORD pdwThreadId);
```

```
HANDLE CreateFile(
    PCTSTR pszFileName,
    DWORD dwDesiredAccess,
    DWORD dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD dwCreationDistribution,
    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. Вот поэтому при отладке своего приложения и просмотре фактического значения описателя объекта ядра Вы и видите такие малые величины: 1, 2 и т. д. Но помните, что физическое содержимое описателей не задокументировано и может быть изменено. Кстати, в Windows 2000 это значение определяет, по сути, не индекс, а скорее байтовое смещение нужной записи от начала таблицы описателей.

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). Например, если *CreateFile* не сможет открыть указанный файл, она вернет именно INVALID_HANDLE_VALUE. Будьте очень осторожны при проверке значения, возвращаемого функцией, которая создает объект ядра. Так, для *CreateMutex* проверка на INVALID_HANDLE_VALUE бессмысленна:

```
HANDLE hMutex = 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)
}
```

Заккрытие объекта ядра

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

```
BOOL CloseHandle(HANDLE hobj);
```

Эта функция сначала проверяет таблицу описателей, принадлежащую вызывающему процессу, чтобы убедиться, идентифицирует ли переданный ей индекс (описа-

тель) 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.

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.

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 выполняется. 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.

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 Microsoft 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 с ними, 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 возвращаемый описатель наследуемым

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

:
```

Этот код инициализирует структуру SECURITY_ATTRIBUTES, указывая, что объект следует создать с защитой по умолчанию (в Windows 98 это игнорируется) и что возвращаемый описатель должен быть наследуемым.

WINDOWS 98 Хотя Windows 98 не полностью поддерживает защиту, она все же поддерживает наследование и поэтому корректно обрабатывает элемент *bInheritHandle*.

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, передаете 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.

| 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 |

Таблица 3-2. Таблица описателей с двумя действительными записями

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*.

```
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);
```

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 с 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. 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), «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.

| 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 |

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

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 *sscanf*), 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 *WaitForInputIdle*, 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.

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,
    DWORD dwMask,
    DWORD 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*:

```
BOOL GetHandleInformation(
    HANDLE hObj,
    PDWORD 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. Именованные допускают многие (но не все) объекты ядра. Например, следующие функции создают именованные объекты ядра:

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

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

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

```
HANDLE CreateWaitableTimer(
    PSECURITY_ATTRIBUTES psa,
    BOOL bManualReset,
    PCTSTR pszName);
```

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

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

Последний параметр, *pszName*, у всех этих функций одинаков. Передавая в нем NULL, Вы создасте 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, Microsoft ничего не сообщает о правилах именования объектов ядра. Например, создавая объект с именем JeffObj, 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, "JeffObj");
HANDLE hSem = CreateSemaphore(NULL, 1, 1, "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.

Теперь, когда Вы научились именовать объекты, рассмотрим, как разделять их между процессами по именам. Допустим, после запуска процесса А вызывается функция:

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

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

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

```
HANDLE hMutexProcessB = CreateMutex(NULL, FALSE, "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 В, 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.

Однако, хотя процесс В 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 — А и В. 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.



Разделяя объекты ядра по именам, помните об одной крайне важной вещи. Вызывая *CreateMutex*, процесс В 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, "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 *Create*-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, Microsoft 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 *main* или *WinMain* 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 WinMain(HINSTANCE hinstExe, HINSTANCE, PSTR pszCmdLine, int nCmdShow) {
    HANDLE h = CreateMutex(NULL, FALSE,
        "{FA531CC1-0497-11d3-A180-00105A276C3E}");
    if (GetLastError() == ERROR_ALREADY_EXISTS) {
        // экземпляр этого приложения уже выполняется
        return(0);
    }
    // запущен первый экземпляр данного приложения
    :

    // перед выходом закрываем объект
    CloseHandle(h);
    return(0);
}
```

Пространства имен Terminal Server

Terminal Server несколько меняет описанный выше сценарий. На машине с Terminal Server 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.

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, "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, "Local\\MyName");
```

Microsoft рассматривает префиксы `Global` и `Local` как зарезервированные ключевые слова, которые не должны встречаться в самих именах объектов. К числу таких слов Microsoft относит и `Session`, хотя на сегодняшний день оно не связано ни с какой функциональностью. Также обратите внимание на две вещи: все эти ключевые слова чувствительны к регистру букв и игнорируются, если компьютер работает без Terminal Server.

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* 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*.

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».

| 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 3-4. Таблица описателей в процессе C

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».

| 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 |

Таблица 3-5. Таблица описателей в процессе S

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.

| 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 |

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

Если процесс C теперь вызовет *DuplicateHandle* так:

```
DuplicateHandle(1, 2, 2, &hObj, 0, TRUE, DUPLICATE_SAME_ACCESS);
```

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

| 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 |

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

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 *blnInheritHandle* функции *DuplicateHandle* мы передали *TRUE*.

Очевидно, Вы никогда не станете передавать в *DuplicateHandle* жестко зашитые значения, как это сделал я, просто демонстрируя работу функции. В реальных программах значения описателей хранятся в переменных и, конечно же, именно эти переменные передаются функциям.

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 Т, что тот имеет теперь доступ к новому объекту; для этого нужно воспользоваться одной из форм межпроцессной связи и передать в процесс Т значение описателя в переменной *hObj*. Ясное дело, в данном случае не годится ни командная строка, ни изменение переменных окружения процесса Т, 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 увидите, 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 Т, используйте *DuplicateHandle* так:

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

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

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

HANDLE hObjProcessT;    // неинициализированный описатель,
                        // связанный с процессом Т

// предоставляем процессу Т доступ к объекту-мьютексу
DuplicateHandle(GetCurrentProcess(), hObjProcessS, hProcessT,
               &hObjProcessT, 0, FALSE, DUPLICATE_SAME_ACCESS);

// используем какую-нибудь форму межпроцессной связи, чтобы передать
// значение описателя из hObjProcessS в процесс Т
:

// связь с процессом Т больше не нужна
CloseHandle(hProcessT);

:

// если процессу S не нужен объект-мьютекс, он должен закрыть его
CloseHandle(hObjProcessS);
```

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 Т 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(hObjProcessT);
```

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 процесс S закроет неизвестно какой объект, и что будет потом — остается только гадать.

Теперь о другом способе применения *DuplicateHandle*. 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 WinMain(HINSTANCE hinstExe, HINSTANCE,
    PSTR pszCmdLine, 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 hFileMapRO;
    DuplicateHandle(GetCurrentProcess(), hFileMapRW, GetCurrentProcess(),
        &hFileMapRO, FILE_MAP_READ, FALSE, 0);

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

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

    // проекция файла нам по-прежнему полностью доступна через hFileMapRW
    :

    // если проекция файла больше не нужна основному коду, закрываем ее
    CloseHandle(hFileMapRW);
}
```

Ч А С Т Ь II

НАЧИНАЕМ РАБОТАТЬ



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. О'кэй, 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.

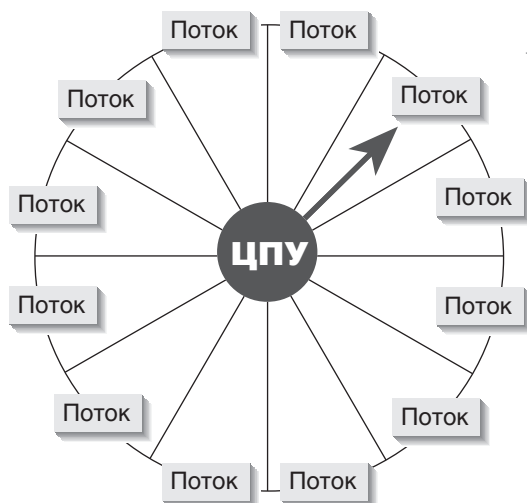


Рис. 4-1. Операционная система выделяет потокам кванты времени по принципу карусели

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. 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 времени (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. Если в машине установлено более одного процессора, алгоритм работы операционной системы значительно усложняется (в этом случае система стремится сбалансировать нагрузку между процессорами).

При создании процесса первый (точнее, первичный) поток создается системой автоматически. Далее этот поток может породить другие потоки, те в свою очередь — новые и т. д.

WINDOWS 2000

Windows 2000 в полной мере использует возможности машин с несколькими процессорами. Например, эту книгу я писал, сидя за машиной с двумя процессорами. Windows 2000 способна закрепить каждый поток за отдельным процессором, и тогда два потока исполняются действительно одновременно. Ядро Windows 2000 полностью поддерживает распределение процессорного времени между потоками и управление ими на таких системах. Вам не придется делать ничего особенного в своем коде, чтобы задействовать преимущества многопроцессорной машины.

WINDOWS 98

Windows 98 работает только с одним процессором. Даже если у компьютера несколько процессоров, под управлением Windows 98 действует лишь один из них — остальные простаивают.

Ваше первое Windows-приложение

Windows поддерживает два типа приложений: основанные на графическом интерфейсе (graphical user interface, GUI) и консольные (console user interface, CUI). 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 консольные 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 приложений 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 про-

граммы: 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 «дружелюбнее» 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 CUI-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 CUI. 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 четыре такие функции:

```
int WINAPI WinMain(
    HINSTANCE hinstExe,
    HINSTANCE,
    PSTR pszCmdLine,
    int nCmdShow);
```

```
int WINAPI wWinMain(
    HINSTANCE hinstExe,
    HINSTANCE,
    PWSTR pszCmdLine,
    int nCmdShow);
```

```
int __cdecl main(
    int argc,
    char *argv[],
    char *envp[]);
```

```
int __cdecl wmain(
    int argc,
    wchar_t *argv[],
    wchar_t *envp[]);
```

На самом деле входная функция операционной системой не вызывается. Вместо этого происходит обращение к стартовой функции из библиотеки C/C++. 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.

| Тип приложения | Входная функция | Стартовая функция, встраиваемая в Ваш исполняемый файл |
|---|-----------------|--|
| GUI-приложение, работающее с ANSI-символами и строками | <i>WinMain</i> | <i>WinMainCRTStartup</i> |
| GUI-приложение, работающее с Unicode-символами и строками | <i>wWinMain</i> | <i>wWinMainCRTStartup</i> |
| CUI-приложение, работающее с ANSI-символами и строками | <i>main</i> | <i>mainCRTStartup</i> |
| CUI-приложение, работающее с Unicode-символами и строками | <i>wmain</i> | <i>wmainCRTStartup</i> |

Нужную стартовую функцию в библиотеке C/C++ выбирает компоновщик при сборке исполняемого файла. Если указан ключ `/SUBSYSTEM:WINDOWS`, компоновщик ищет в Вашем коде функцию *WinMain* или *wWinMain*. Если ни одной из них нет, он сообщает об ошибке «unresolved external symbol» («неразрешенный внешний символ»); в ином случае — 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 и 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 `/SUBSYSTEM:WINDOWS` на `/SUBSYSTEM:CONSOLE`. 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.

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 CRt0.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 4-1;
- 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 *wWinMain*, то она вызывается так:

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

А если Вы предпочли *WinMain*, то:

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

И, наконец, то же самое для функций *wmain* и *main*:

```
int nMainRetVal = wmain(__argc, __wargv, _wenvron);
int nMainRetVal = main(__argc, __argv, _environ);
```

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;
- вызывает Windows-функцию *ExitProcess*, передавая ей значение *nMainRetVal*. Это заставляет операционную систему уничтожить Ваш процесс и установить код его завершения.

| Имя переменной | Тип | Описание |
|------------------|---------------------|---|
| <i>_osver</i> | <i>unsigned int</i> | Версия сборки операционной системы. Например, у Windows 2000 Beta 3 этот номер был 2031, соответственно <i>_osver</i> равна 2031. |
| <i>_winmajor</i> | <i>unsigned int</i> | Основной номер версии Windows в шестнадцатеричной форме. Для Windows 2000 это значение равно 5. |

Таблица 4-1. Глобальные переменные из библиотеки C/C++, доступные Вашим программам

продолжение

| Имя переменной | Тип | Описание |
|---|---|---|
| <code>_winminor</code> | <code>unsigned int</code> | Дополнительный номер версии Windows в шестнадцатеричной форме. Для Windows 2000 это значение равно 0. |
| <code>_winver</code> | <code>unsigned int</code> | Вычисляется как <code>(_winmajor << 8) + _winminor</code> . |
| <code>__argc</code> | <code>unsigned int</code> | Количество аргументов, переданных в командной строке. |
| <code>__argv</code> <code>__wargv</code> | <code>char **</code> <code>wchar_t **</code> | Массив размером <code>__argc</code> с указателями на ANSI- или Unicode-строки. Каждый элемент массива указывает на один из аргументов командной строки. |
| <code>_environ</code> <code>_wenviron</code> | <code>char **</code> <code>wchar_t **</code> | Массив указателей на ANSI- или Unicode-строки. Каждый элемент массива указывает на строку — переменную окружения. |
| <code>_pgmptr</code> <code>_wpgmptr</code> | <code>char **</code> <code>wchar_t **</code> | Полный путь и имя (в ANSI или Unicode) запускаемой программы. |

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* — *hinstExe*. 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 hinst,
    PCTSTR pszIcon);
```

Первый параметр в *LoadIcon* указывает, в каком файле (EXE или DLL) содержится интересующий Вас ресурс. Многие приложения сохраняют параметр *hinstExe* 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 hinstModule,
    PTSTR pszPath,
    DWORD cchPath);
```



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 *hinstExe* 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 *hinstExe* 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 Microsoft), можно изменить базовый адрес, по которому будет загружаться приложение.

При попытке загрузить исполняемый файл в Windows 98 по базовому адресу ниже 0x00400000 загрузчик переместит его на другой адрес. Это увеличит время загрузки приложения, но оно по крайней мере будет выполнено. Если Вы разрабатываете программы и для Windows 98, и для Windows 2000, сделайте так, чтобы приложение загружалось по базовому адресу не ниже 0x00400000.

Функция *GetModuleHandle*:

```
HMODULE 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.

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 *hinstExePrev* 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 WinMain(
    HINSTANCE hinstExe,
    HINSTANCE,
    PSTR pszCmdLine,
    int nCmdShow);
```

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

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/C++ считывает командную строку процесса, пропускает имя исполняемого файла и заносит в параметр *pszCmdLine* функции (*w*)*WinMain* указатель на оставшуюся часть командной строки.

Параметр *pszCmdLine* всегда указывает на ANSI-строку. Но, заменив *WinMain* на *wWinMain*, Вы получите доступ к Unicode-версии командной строки для своего процесса.

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*:

```
PCTSTR GetCommandLine();
```

Она возвращает указатель на буфер, содержащий полную командную строку, включая полное имя (вместе с путем) исполняемого файла.

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*). Функция *CommandLineToArgvW* 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 pNumArgs;
PWSTR *ppArgv = CommandLineToArgvW(GetCommandLineW(), &pNumArgs);
```

см. след. стр.

```
// 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. Строки в блоке переменных окружения должны быть отсортированы в алфавитном порядке по именам переменных.

Знак равенства разделяет имя переменной и ее значение, так что его нельзя использовать как символ в имени переменной. Важную роль играют и пробелы. Например, объявив две переменные:

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

и сравнив значения переменных *XYZ* и *ABC*, 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 перед знаком равенства)
XYZ=Work
```

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

Конец блока переменных окружения помечается дополнительным нулевым символом.

WINDOWS 98 Чтобы создать исходный набор переменных окружения для Windows 98, надо модифицировать файл Autoexec.bat, поместив в него группу строк SET в виде:

```
SET VarName=VarValue
```

При перезагрузке система учтет новое содержимое файла Autoexec.bat, и тогда любые заданные Вами переменные окружения станут доступны всем процессам, иницируемым в сеансе работы с Windows 98.

WINDOWS
2000

При регистрации пользователя на входе в Windows 2000 система создает процесс-оболочку, связывая с ним группу строк — переменных окружения. Система получает начальные значения этих строк, анализируя два раздела в реестре. В первом:

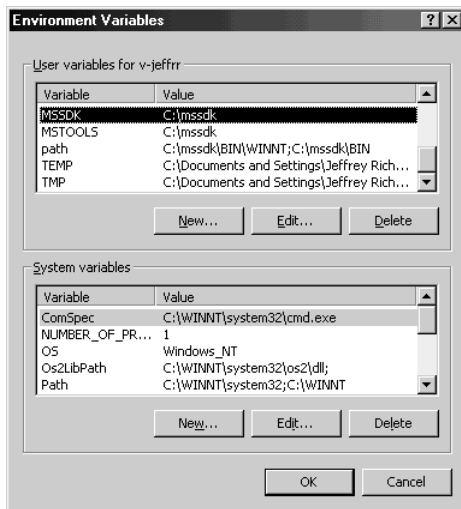
```
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\
  SessionManager\Environment
```

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 апплет System из Control Panel. В этом апплете 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 с 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) TEXT("Environment"));
```

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 зай-

мемся 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.

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

```
%USERPROFILE%\My 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:\Documents and Settings\Administrator
```

После подстановки переменной в строку реестра получим:

```
C:\Documents and Settings\Administrator\My Documents
```

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

```
DWORD ExpandEnvironmentStrings(
    PCTSTR pszSrc,
    PTSTR pszDst,
    DWORD nSize);
```

Параметр *pszSrc* принимает адрес строки, содержащей подставляемые части, а параметр *pszDst* — адрес буфера, в который записывается развернутая строка. Параметр *nSize* определяет максимальный размер буфера в символах.

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

```
DWORD 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. Как я уже говорил, строки в блоке переменных нужно отсортировать в алфавитном порядке по именам переменных (тогда *GetEnvironmentVariable* быстрее находит нужные переменные), а *SetEnvironmentVariable* как раз и следит за порядком расположения переменных.

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, комбинируемых побитовой операцией OR:

| 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 |
| 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);
```

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.



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 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 функции защиты, заложенные в Windows API, но в полной мере эти функции реализованы лишь в Windows 2000.

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, Microsoft 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(POSVERSIONINFO pVersionInformation);
```

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

```
typedef struct {
    DWORD dwOSVersionInfoSize;
    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 — новинка 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 байтослов (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 4-2.

| Элемент | Описание |
|----------------------------|--|
| <i>dwOSVersionInfoSize</i> | Размер структуры; перед обращением к функции <i>GetVersionEx</i> должен быть заполнен вызовом <i>sizeof(OSVERSIONINFO)</i> или <i>sizeof(OSVERSIONINFOEX)</i> |
| <i>dwMajorVersion</i> | Основной номер версии операционной системы |
| <i>dwMinorVersion</i> | Дополнительный номер версии операционной системы |
| <i>dwBuildNumber</i> | Версия сборки данной системы |
| <i>dwPlatformId</i> | Идентификатор платформы, поддерживаемой данной системой; его возможные значения: VER_PLATFORM_WIN32s (Win32s), VER_PLATFORM_WIN32_WINDOWS (Windows 95/98), VER_PLATFORM_WIN32_NT (Windows NT или Windows 2000), VER_PLATFORM_WIN32_CE (Windows CE) |
| <i>szCSDVersion</i> | Этот элемент содержит текст — дополнительную информацию об установленной операционной системе |
| <i>wServicePackMajor</i> | Основной номер версии последнего установленного пакета исправлений (service pack) |
| <i>wServicePackMinor</i> | Дополнительный номер версии последнего установленного пакета исправлений |

Таблица 4-2. Элементы структуры OSVERSIONINFOEX

продолжение

| Элемент | Описание |
|---------------------|---|
| <i>wSuiteMask</i> | 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 |
| <i>wProductType</i> | Сообщает, какой именно вариант операционной системы установлен; его возможные значения: VER_NT_WORKSTATION, VER_NT_SERVER, VER_NT_DOMAIN_CONTROLLER |
| <i>wReserved</i> | Зарезервирован на будущее |

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);
```

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_CONDITION(
    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, идентичны передаваемым в параметре *dwTypeMask* функции *VerifyVersionInfo*.

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 2000:

```
// готовим структуру OSVERSIONINFOEX, сообщая, что нам нужна Windows 2000
OSVERSIONINFOEX osver = { 0 };
osver.dwOSVersionInfoSize = sizeof(osver);
osver.dwMajorVersion = 5;
osver.dwMinorVersion = 0;
osver.dwPlatformId = VER_PLATFORM_WIN32_NT;

// формируем маску условий
DWORDLONG dwlConditionMask = 0;    // всегда инициализируйте этот элемент так
VER_SET_CONDITION(dwlConditionMask, VER_MAJORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_MINORVERSION, VER_EQUAL);
VER_SET_CONDITION(dwlConditionMask, VER_PLATFORMID, VER_EQUAL);

// проверяем версию
if (VerifyVersionInfo(&osver, VER_MAJORVERSION | VER_MINORVERSION | VER_PLATFORMID,
    dwlConditionMask)) {
    // хост-система точно соответствует Windows 2000
} else {
    // хост-система не является Windows 2000
}
```

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.



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*.



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++ 6.0 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.)

см. след. стр.

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 /GF.) В общем, лучшее, что можете сделать Вы, — использовать ключ /GF или создать временный буфер. А еще лучше, если Microsoft 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.
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:\\WINNT\\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 2000.

Параметры *psaProcess*, *psaThread* и *blInheritHandles*

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 на рис. 4-2 демонстрирует, как наследуются описатели объектов ядра. Будем считать, что процесс А порождает процесс В и заносит в параметр *psaProcess* адрес структуры SECURITY_ATTRIBUTES, в которой элемент *blInheritHandle* установлен как TRUE. Одновременно параметр *psaThread* указывает на другую структуру SECURITY_ATTRIBUTES, в которой значение элемента *blInheritHandle* — 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 *blInheritHandles*. 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 *blInheritHandles*. Кроме того, если процесс А вызывает *CreateProcess*, передавая через параметр *blInheritHandles* значение FALSE, процесс С не наследует никаких описателей, используемых в данный момент процессом А.

Inherit.c

```

/*****
Модуль: Inherit.c
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include <Windows.h>

int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,
    PSTR 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 В из процесса А;
    // 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 В
    lstrcpy(szPath, TEXT("ProcessB"));
    CreateProcess(NULL, szPath, &saProcess, &saThread,
        FALSE, 0, NULL, NULL, &si, &piProcessB);

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

    // готовимся создать процесс С из процесса А;
    // так как в saProcess и saThread передаются NULL, описатели
    // объектов "процесс" и "первичный поток" процесса С считаются
    // ненаследуемыми по умолчанию

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

    // поскольку в параметре bInheritHandles передается TRUE,
    // процесс С унаследует описатель, идентифицирующий объект
    // "процесс" процесса В, но НЕ описатель, идентифицирующий объект

```

Рис. 4-2. Пример, иллюстрирующий наследование описателей объектов ядра

Рис. 4-2. продолжение

```
// "первичный поток" того же процесса
lstrcpy(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. 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.
- 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 `DETACHED_PROCESS` 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.
- 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 выполняется 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 Windows 2000. 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 отдельных VDM, имеют и отдельные 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 2000. 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 одной VDM.
- 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).

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.

| 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. (Подробнее на эту тему см. главу 7.)



Классы приоритета BELOW_NORMAL_PRIORITY_CLASS и ABOVE_NORMAL_PRIORITY_CLASS введены лишь в Windows 2000; они не поддерживаются в Windows NT 4.0, Windows 95 или Windows 98.

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 *GetEnvironmentStrings*:

```
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:

```
typedef struct _STARTUPINFO {
    DWORD cb;
    PSTR lpReserved;
    PSTR lpDesktop;
    PSTR lpTitle;
    DWORD dwX;
    DWORD dwY;
```

см. след. стр.


```
    DWORD dwXSize;
    DWORD dwYSize;
    DWORD dwXCountChars;
    DWORD dwYCountChars;
    DWORD dwFillAttribute;
    DWORD dwFlags;
    WORD wShowWindow;
    WORD cbReserved2;
    PBYTE lpReserved2;
    HANDLE hStdInput;
    HANDLE hStdOutput;
    HANDLE hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

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 4-3. 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.

| Элемент | Окно или консоль | Описание |
|-------------------|------------------|---|
| <i>cb</i> | То и другое | Содержит количество байтов, занимаемых структурой <code>STARTUPINFO</code> . Служит для контроля версий — на тот случай, если Microsoft расширит эту структуру в будущем. Программа должна инициализировать <i>cb</i> как <i>sizeof(STARTUPINFO)</i> . |
| <i>lpReserved</i> | То и другое | Зарезервирован. Инициализируйте как <code>NULL</code> . |
| <i>lpDesktop</i> | 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 <i>lpDesktop</i> равен <code>NULL</code> (что чаще всего и бывает), процесс связывается с текущим рабочим столом. |
| <i>lpTitle</i> | Консоль | Определяет заголовок консольного окна. Если <i>lpTitle</i> — <code>NULL</code> , в заголовок выводится имя исполняемого файла. |

Таблица 4-3. Элементы структуры `STARTUPINFO`

продолжение

| Элемент | Окно или консоль | Описание |
|---|------------------|--|
| <i>dwX</i> <i>dwY</i> | То и другое | Указывают <i>x</i> - и <i>y</i> -координаты (в пикселах) окна приложения. Эти координаты используются, только если дочерний процесс создает свое первое перекрываемое окно с идентификатором <i>CW_USEDEFAULT</i> в параметре <i>x</i> функции <i>CreateWindow</i> . В приложениях, создающих консольные окна, данные элементы определяют верхний левый угол консольного окна. |
| <i>dwXSize</i> <i>dwYSize</i> | 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 <i>CW_USEDEFAULT</i> в параметре <i>nWidth</i> функции <i>CreateWindow</i> . В приложениях, создающих консольные окна, данные элементы определяют ширину и высоту консольного окна. |
| <i>dwXCountChars</i> <i>dwYCountChars</i> | Консоль | Определяют ширину и высоту (в символах) консольных окон дочернего процесса. |
| <i>dwFillAttribute</i> | Консоль | Задаёт цвет текста и фона в консольных окнах дочернего процесса. |
| <i>dwFlags</i> | То и другое | См. ниже и следующую таблицу. |
| <i>wShowWindow</i> | Окно | Определяет, как именно должно выглядеть первое перекрываемое окно дочернего процесса, если приложение при первом вызове функции <i>ShowWindow</i> передает в параметре <i>nCmdShow</i> идентификатор <i>SW_SHOWDEFAULT</i> . В этот элемент можно записать любой из идентификаторов типа <i>SW_*</i> , обычно используемых при вызове <i>ShowWindow</i> . |
| <i>cbReserved2</i> | То и другое | Зарезервирован. Инициализируйте как 0. |
| <i>lpReserved2</i> | То и другое | Зарезервирован. Инициализируйте как NULL. |
| <i>hStdInput</i> <i>hStdOutput</i> <i>hStdError</i> | Консоль | Определяют описатели буферов для консольного ввода-вывода. По умолчанию <i>hStdInput</i> идентифицирует буфер клавиатуры, а <i>hStdOutput</i> и <i>hStdError</i> — 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.

| Флаг | Описание |
|--------------------------------|--|
| <i>STARTF_USESIZE</i> | Заставляет использовать элементы <i>dwXSize</i> и <i>dwYSize</i> |
| <i>STARTF_USESHOWWINDOW</i> | Заставляет использовать элемент <i>wShowWindow</i> |
| <i>STARTF_USEPOSITION</i> | Заставляет использовать элементы <i>dwX</i> и <i>dwY</i> |
| <i>STARTF_USECOUNTCHARS</i> | Заставляет использовать элементы <i>dwXCountChars</i> и <i>dwYCountChars</i> |
| <i>STARTF_USEFILLATTRIBUTE</i> | Заставляет использовать элемент <i>dwFillAttribute</i> |
| <i>STARTF_USESTDHANDLES</i> | Заставляет использовать элементы <i>hStdInput</i> , <i>hStdOutput</i> и <i>hStdError</i> |

см. след. стр.

| Флаг | Описание |
|-----------------------|--|
| STARTF_RUN_FULLSCREEN | Приводит к тому, что консольное приложение на компьютере с процессором типа x86 запускается в полноэкранном режиме |

Два дополнительных флага — `STARTF_FORCEONFEEDBACK` и `STARTF_FORCEOFFFEEDBACK` — позволяют контролировать форму курсора 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 курсора мыши:



Курсор 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 курсора. Когда функция *CreateProcess* вызывается с этим флагом, курсор преобразуется в «песочные часы». Если спустя две секунды от нового процесса не поступает GUI-вызов, она восстанавливает исходную форму курсора.

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 курсор, 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 курсор на стандартный и прекратит мониторинг нового процесса.

В заключение раздела — несколько слов об элементе *wShowWindow* структуры `STARTUPINFO`. 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.

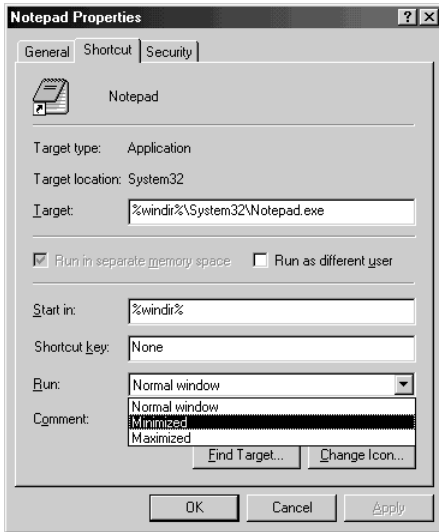


Рис. 4-3. Окно свойств для ярлыка Notepad

Наконец, чтобы получить копию структуры `STARTUPINFO`, инициализированной родительским процессом, приложение может вызвать:

```
VOID GetStartupInfo(PSTARTUPINFO pStartupInfo);
```

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



Хотя в документации на Windows об этом четко не сказано, перед вызовом *GetStartupInfo* нужно инициализировать элемент *cb* структуры `STARTUPINFO`:

```
STARTUPINFO si = { sizeof(si) };
GetStartupInfo(&si);
:
```

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.



Не забывайте 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, 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.

Подчеркну еще один чрезвычайно важный момент: система способна повторно использовать идентификаторы процессов и потоков. Например, при создании процесса система формирует объект «процесс», присваивая ему идентификатор со значением, допустим, 122. 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 — 122.

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.

Иногда программе придется 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 взаимодей-

ствовать 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 *CloseHandle* 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 завершился 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;
- 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("Destructor\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 его выполнении Вы увидите:

```
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 рассказывал, относится не только к объектам, но и ко многим другим вещам, которые библиотека C/C++ делает для Вашего процесса.



Явные вызовы *ExitProcess* и *ExitThread* — распространенная ошибка, которая мешает правильной очистке ресурсов. В случае *ExitThread* процесс продолжает работать, но при этом весьма вероятно 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* идентифицирует описатель завершаемого процесса, а в параметре *fuExitCode* возвращается код завершения процесса.

Пользуйтесь *TerminateProcess* лишь в том случае, когда иным способом завершить процесс не удастся. Процесс не получает абсолютно никаких уведомлений 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.



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 выполняется 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 заверщенного 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);
```

Код завершения возвращается как значение типа `DWORD`, на которое указывает *pdwExitCode*. 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.

Дочерние процессы

При разработке приложения часто бывает нужно, чтобы какую-то операцию выполнял другой блок кода. Поэтому — хочешь, не хочешь — приходится 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 позво-

лит 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_INFORMATION pi;
DWORD dwExitCode;

// порождаем дочерний процесс
BOOL fSuccess = CreateProcess(..., &pi);
if (fSuccess) {

    // закрывайте дескриптор потока, как только необходимость 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, &dwExitCode);

    // закрывайте дескриптор процесса, как только необходимость 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 замечанием. 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, связанные с новым процессом и его первичным потоком. Приведенный ниже фрагмент кода демонстрирует, как, создав процесс, сделать его обособленным:

```
PROCESS_INFORMATION pi;

BOOL fSuccess = CreateProcess(..., &pi);
if (fSuccess) {
    // разрешаем системе уничтожить объекты ядра "процесс" и "поток"
    // сразу после завершения дочернего процесса
    CloseHandle(pi.hThread);
    CloseHandle(pi.hProcess);
}
```

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 корне-

вой 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, Microsoft 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*.

Microsoft, 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 Windows 2000!

Программа-пример ProcessInfo

Эта программа, «04 ProcessInfo.exe» (см. листинг на рис. 4-6), демонстрирует, как создать очень полезную утилиту на основе ToolHelp-функций. Файлы исходного кода и ресурсов программы находятся в каталоге 04-ProcessInfo на компакт-диске, прилагаемом к книге. После запуска ProcessInfo открывается окно, показанное на рис. 4-4.

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 спроеци-

рован 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.

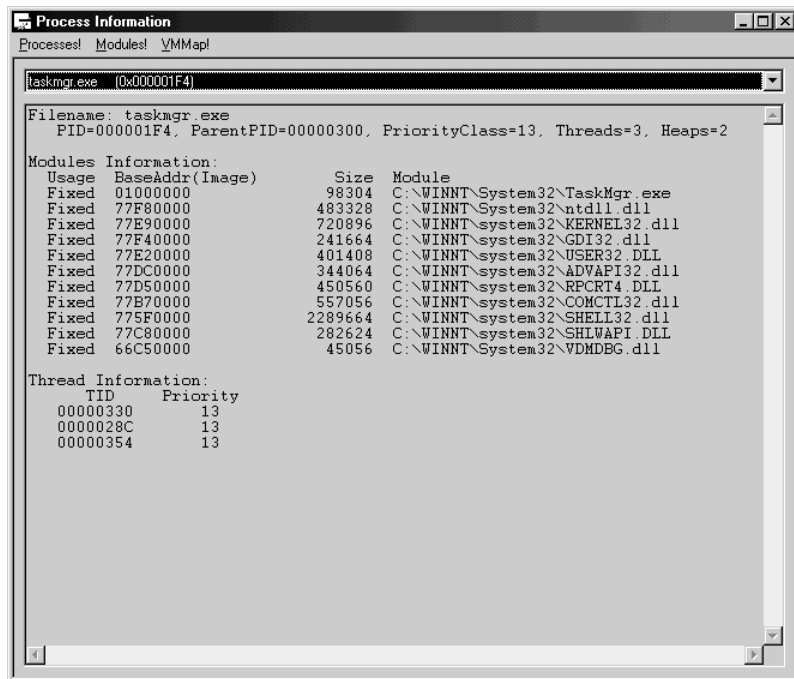


Рис. 4-4. ProcessInfo в действии

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).

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 ProcessInfo.cpp:

```
PVOID GetModulePreferredBaseAddr(
    DWORD dwProcessId,
    PVOID pvModuleRemote);
```

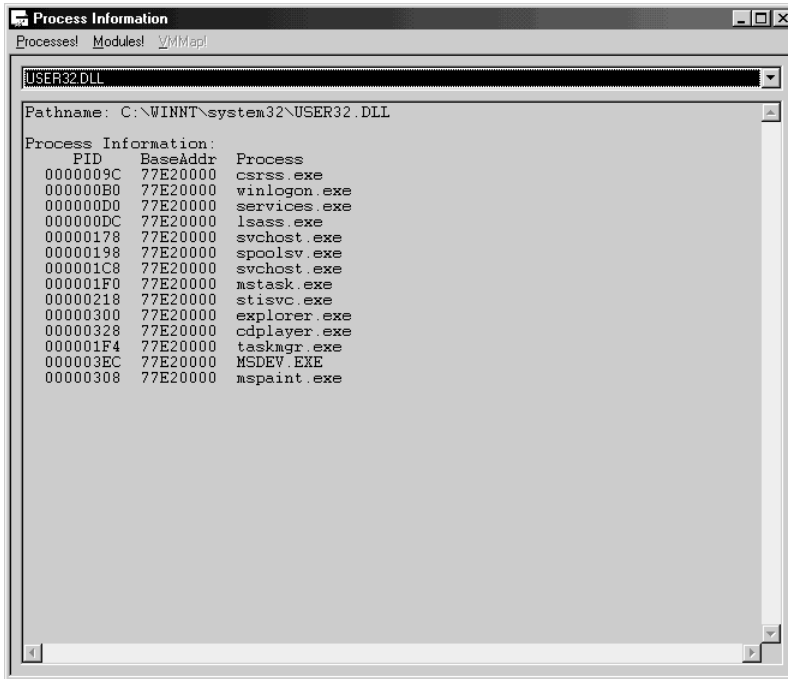


Рис. 4-5. *ProcessInfo* перечисляет все процессы, в адресные пространства которых загружен модуль *User32.dll*

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 главе 20 (в разделе «Модификация базовых адресов модулей»).



ProcessInfo.cpp

```

/*****
Модуль: ProcessInfo.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tlhelp32.h>
#include <tchar.h>
#include <stdarg.h>
#include <stdio.h>
#include "Toolhelp.h"
#include "Resource.h"

```

Рис. 4-6. *Программа-пример ProcessInfo*

см. след. стр.

Рис. 4-6. *продолжение*

```

////////////////////////////////////
// добавляем строку в текстовое поле
void AddText(HWND hwnd, PCTSTR pszFormat, ...) {

    va_list argList;
    va_start(argList, pszFormat);

    TCHAR sz[20 * 1024];
    Edit_GetText(hwnd, sz, chDIMOF(sz));
    _vstprintf(_tcschr(sz, 0), pszFormat, argList);
    Edit_SetText(hwnd, sz);
    va_end(argList);
}

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

VOID Dlg_PopulateProcessList(HWND hwnd) {

    HWND hwndList = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
    SetWindowRedraw(hwndList, FALSE);
    ComboBox_ResetContent(hwndList);

    CToolhelp thProcesses(TH32CS_SNAPPROCESS);
    PROCESSENTRY32 pe = { sizeof(pe) };
    BOOL fOk = thProcesses.ProcessFirst(&pe);
    for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {
        TCHAR sz[1024];

        // помещаем в список имя процесса (без пути) и идентификатор
        PCTSTR pszExeFile = _tcsrchr(pe.szExeFile, TEXT('\\'));
        if (pszExeFile == NULL) pszExeFile = pe.szExeFile;
        else pszExeFile++; // пропускаем наклонную черту ("слэш")
        wsprintf(sz, TEXT("%s (0x%08X)"), pszExeFile, pe.th32ProcessID);
        int n = ComboBox_AddString(hwndList, sz);

        // сопоставляем идентификатор процесса с добавленным элементом
        ComboBox_SetItemData(hwndList, n, pe.th32ProcessID);
    }
    ComboBox_SetCurSel(hwndList, 0); // выбираем первый элемент

    // имитируем выбор пользователем первого элемента,
    // чтобы в текстовом поле появилось что-нибудь интересное
    FORWARD_WM_COMMAND(hwnd, IDC_PROCESSMODULELIST,
        hwndList, CBN_SELCHANGE, SendMessage);

    SetWindowRedraw(hwndList, TRUE);
    InvalidateRect(hwndList, NULL, FALSE);
}

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

```

Рис. 4-6. продолжение

```

VOID Dlg_PopulateModuleList(HWND hwnd) {

    HWND hwndModuleHelp = GetDlgItem(hwnd, IDC_MODULEHELP);
    ListBox_ResetContent(hwndModuleHelp);
    CToolhelp thProcesses(TH32CS_SNAPPROCESS);
    PROCESSENTRY32 pe = { sizeof(pe) };
    BOOL fOk = thProcesses.ProcessFirst(&pe);
    for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {

        CToolhelp thModules(TH32CS_SNAPMODULE, pe.th32ProcessID);
        MODULEENTRY32 me = { sizeof(me) };
        BOOL fOk = thModules.ModuleFirst(&me);
        for (; fOk; fOk = thModules.ModuleNext(&me)) {
            int n = ListBox_FindStringExact(hwndModuleHelp, -1, me.szExePath);
            if (n == LB_ERR) {
                // этот модуль еще не был добавлен
                ListBox_AddString(hwndModuleHelp, me.szExePath);
            }
        }
    }

    HWND hwndList = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
    SetWindowRedraw(hwndList, FALSE);
    ComboBox_ResetContent(hwndList);
    int nNumModules = ListBox_GetCount(hwndModuleHelp);
    for (int i = 0; i < nNumModules; i++) {
        TCHAR sz[1024];
        ListBox_GetText(hwndModuleHelp, i, sz);
        // помещаем в список имя модуля (без пути)
        int nIndex = ComboBox_AddString(hwndList, _tcsrchr(sz, TEXT('\\')) + 1);
        // сопоставляем индекс полного пути с добавленным элементом
        ComboBox_SetItemData(hwndList, nIndex, i);
    }

    ComboBox_SetCurSel(hwndList, 0); // выбираем первый элемент

    // имитируем выбор пользователем первого элемента,
    // чтобы в текстовом поле появилось что-нибудь интересное
    FORWARD_WM_COMMAND(hwnd, IDC_PROCESSMODULELIST,
        hwndList, CBN_SELCHANGE, SendMessage);

    SetWindowRedraw(hwndList, TRUE);
    InvalidateRect(hwndList, NULL, FALSE);
}

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

PVOID GetModulePreferredBaseAddr(DWORD dwProcessId, PVOID pvModuleRemote) {

    PVOID pvModulePreferredBaseAddr = NULL;

```

см. след. стр.

Рис. 4-6. *продолжение*

```

IMAGE_DOS_HEADER idh;
IMAGE_NT_HEADERS inth;

// считываем DOS-заголовок удаленного модуля
Toolhelp32ReadProcessMemory(dwProcessId,
    pvModuleRemote, &idh, sizeof(idh), NULL);

// проверяем DOS-заголовок его образа
if (idh.e_magic == IMAGE_DOS_SIGNATURE) {
    // считываем NT-заголовок удаленного модуля
    Toolhelp32ReadProcessMemory(dwProcessId,
        (PBYTE) pvModuleRemote + idh.e_lfanew, &inth, sizeof(inth), NULL);

    // проверяем NT-заголовок его образа
    if (inth.Signature == IMAGE_NT_SIGNATURE) {
        // NT-заголовок корректен,
        // получаем предпочтительный базовый адрес для данного образа
        pvModulePreferredBaseAddr = (PVOID) inth.OptionalHeader.ImageBase;
    }
}
return(pvModulePreferredBaseAddr);
}

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

VOID ShowProcessInfo(HWND hwnd, DWORD dwProcessID) {

    SetWindowText(hwnd, TEXT("")); // очищаем поле вывода

    CToolhelp th(TH32CS_SNAPALL, dwProcessID);

    // показываем подробную информацию о процессе
    PROCESSENTRY32 pe = { sizeof(pe) };
    BOOL fOk = th.ProcessFirst(&pe);
    for (; fOk; fOk = th.ProcessNext(&pe)) {
        if (pe.th32ProcessID == dwProcessID) {
            AddText(hwnd, TEXT("Filename: %s\r\n"), pe.szExeFile);
            AddText(hwnd, TEXT("    PID=%08X, ParentPID=%08X, ")
                TEXT("PriorityClass=%d, Threads=%d, Heaps=%d\r\n"),
                pe.th32ProcessID, pe.th32ParentProcessID,
                pe.pcPriClassBase, pe.cntThreads,
                th.HowManyHeaps());
            break; // продолжать цикл больше не нужно
        }
    }

    // показываем модули в процессе;
    // подсчитываем количество символов для вывода адреса
    const int cchAddress = sizeof(PVOID) * 2;
    AddText(hwnd, TEXT("\r\nModules Information:\r\n")
        TEXT("  Usage %-*s(%-*s) %8s Module\r\n"),
        cchAddress, TEXT("BaseAddr"),

```

Рис. 4-6. продолжение

```

        cchAddress, TEXT("ImagAddr"), TEXT("Size"));

MODULEENTRY32 me = { sizeof(me) };
fOk = th.ModuleFirst(&me);

for (; fOk; fOk = th.ModuleNext(&me)) {
    if (me.ProcCntUsage == 65535) {
        // модуль загружен неявно, и его нельзя выгрузить
        AddText(hwnd, TEXT(" Fixed"));
    } else {
        AddText(hwnd, TEXT(" %5d"), me.ProcCntUsage);
    }
    PVOID pvPreferredBaseAddr =
        GetModulePreferredBaseAddr(pe.th32ProcessID, me.modBaseAddr);
    if (me.modBaseAddr == pvPreferredBaseAddr) {
        AddText(hwnd, TEXT(" %p %s %8u %s\r\n"),
            me.modBaseAddr, cchAddress, TEXT(""),
            me.modBaseSize, me.szExePath);
    } else {
        AddText(hwnd, TEXT(" %p(%p) %8u %s\r\n"),
            me.modBaseAddr, pvPreferredBaseAddr, me.modBaseSize, me.szExePath);
    }
}

// показываем потоки в процессе
AddText(hwnd, TEXT("\r\nThread Information:\r\n"));
TEXT(" TID Priority\r\n"));
THREADENTRY32 te = { sizeof(te) };
fOk = th.ThreadFirst(&te);
for (; fOk; fOk = th.ThreadNext(&te)) {
    if (te.th32OwnerProcessID == dwProcessID) {
        int nPriority = te.tpBasePri + te.tpDeltaPri;
        if ((te.tpBasePri < 16) && (nPriority > 15)) nPriority = 15;
        if ((te.tpBasePri > 15) && (nPriority > 31)) nPriority = 31;
        if ((te.tpBasePri < 16) && (nPriority < 1)) nPriority = 1;
        if ((te.tpBasePri > 15) && (nPriority < 16)) nPriority = 16;
        AddText(hwnd, TEXT(" %08X %2d\r\n"),
            te.th32ThreadID, nPriority);
    }
}
}

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

VOID ShowModuleInfo(HWND hwnd, LPCTSTR pszModulePath) {

    SetWindowText(hwnd, TEXT("")); // очищаем поле вывода

    CToolhelp thProcesses(TH32CS_SNAPPROCESS);
    PROCESSENTRY32 pe = { sizeof(pe) };
    BOOL fOk = thProcesses.ProcessFirst(&pe);

```

см. след. стр.

Рис. 4-6. *продолжение*

```

AddText(hwnd, TEXT("Pathname: %s\r\n\r\n"), pszModulePath);
AddText(hwnd, TEXT("Process Information:\r\n"));
AddText(hwnd, TEXT("    PID    BaseAddr  Process\r\n"));
for (; fOk; fOk = thProcesses.ProcessNext(&pe)) {
    CToolhelp thModules(TH32CS_SNAPMODULE, pe.th32ProcessID);
    MODULEENTRY32 me = { sizeof(me) };
    BOOL fOk = thModules.ModuleFirst(&me);
    for (; fOk; fOk = thModules.ModuleNext(&me)) {
        if (_tcscmp(me.szExePath, pszModulePath) == 0) {
            AddText(hwnd, TEXT("  %08X  %p  %s\r\n"),
                pe.th32ProcessID, me.modBaseAddr, pe.szExeFile);
        }
    }
}
}

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

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

    chSETDLGICONS(hwnd, IDI_PROCESSINFO);

    // убираем окно списка модулей
    ShowWindow(GetDlgItem(hwnd, IDC_MODULEHELP), SW_HIDE);

    // пусть в окне результатов используется фиксированный шрифт
    SetWindowFont(GetDlgItem(hwnd, IDC_RESULTS),
        GetStockFont(ANSI_FIXED_FONT), FALSE);

    // по умолчанию показываем список выполняемых процессов
    Dlg_PopulateProcessList(hwnd);

    return(TRUE);
}

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

BOOL Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {

    RECT rc;
    int n = LOWORD(GetDialogBaseUnits());

    HWND hwndCtl = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
    GetClientRect(hwndCtl, &rc);
    SetWindowPos(hwndCtl, NULL, n, n, cx - n - n, rc.bottom, SWP_NOZORDER);

    hwndCtl = GetDlgItem(hwnd, IDC_RESULTS);
    SetWindowPos(hwndCtl, NULL, n, n + rc.bottom + n,
        cx - n - n, cy - (n + rc.bottom + n) - n, SWP_NOZORDER);

    return(0);
}

```

Рис. 4-6. продолжение

```

////////////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    static BOOL s_fProcesses = TRUE;

    switch (id) {
        case IDCANCEL:
            EndDialog(hwnd, id);
            break;

        case ID_PROCESSES:
            s_fProcesses = TRUE;
            EnableMenuItem(GetMenu(hwnd), ID_VMMAP, MF_BYCOMMAND | MF_ENABLED);
            DrawMenuBar(hwnd);
            Dlg_PopulateProcessList(hwnd);
            break;

        case ID_MODULES:
            EnableMenuItem(GetMenu(hwnd), ID_VMMAP, MF_BYCOMMAND | MF_GRAYED);
            DrawMenuBar(hwnd);
            s_fProcesses = FALSE;
            Dlg_PopulateModuleList(hwnd);
            break;

        case IDC_PROCESSMODULELIST:
            if (codeNotify == CBN_SELCHANGE) {
                DWORD dw = ComboBox_GetCurSel(hwndCtl);
                if (s_fProcesses) {
                    dw = (DWORD) ComboBox_GetItemData(hwndCtl, dw); // ID процесса
                    ShowProcessInfo(GetDlgItem(hwnd, IDC_RESULTS), dw);
                } else {
                    // индекс в окне вспомогательного списка
                    dw = (DWORD) ComboBox_GetItemData(hwndCtl, dw);
                    TCHAR szModulePath[1024];
                    ListBox_GetText(GetDlgItem(hwnd, IDC_MODULEHELP), dw, szModulePath);
                    ShowModuleInfo(GetDlgItem(hwnd, IDC_RESULTS), szModulePath);
                }
            }
            break;

        case ID_VMMAP:
            STARTUPINFO si = { sizeof(si) };
            PROCESS_INFORMATION pi;
            TCHAR szCmdLine[1024];
            HWND hwndCB = GetDlgItem(hwnd, IDC_PROCESSMODULELIST);
            DWORD dwProcessId = (DWORD)
                ComboBox_GetItemData(hwndCB, ComboBox_GetCurSel(hwndCB));
            wsprintf(szCmdLine, TEXT("\"%14 VMMMap\" %d"), dwProcessId);
            BOOL fOk = CreateProcess(NULL, szCmdLine, NULL, NULL,
                FALSE, 0, NULL, NULL, &si, &pi);
    }
}

```

см. след. стр.

Рис. 4-6. *продолжение*

```

if (fOk) {
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
} else {
    chMB("Failed to execute VMMAP.EXE.");
}
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_SIZE, Dlg_OnSize);
        chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }
    return(FALSE);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    CToolhelp::EnableDebugPrivilege(TRUE);
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_PROCESSINFO), NULL, Dlg_Proc);
    CToolhelp::EnableDebugPrivilege(FALSE);
    return(0);
}

//////////////////////////////////// Конец файла //////////////////////////////////////

```

Toolhelp.h

```

/*****
Модуль: Toolhelp.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h" /* см. приложение A */
#include <tlhelp32.h>
#include <tchar.h>

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

class CToolhelp {
private:
    HANDLE m_hSnapshot;

```


Рис. 4-6. продолжение

```

public:
    CToolhelp(DWORD dwFlags = 0, DWORD dwProcessID = 0);
    ~CToolhelp();

    BOOL CreateSnapshot(DWORD dwFlags, DWORD dwProcessID = 0);

    BOOL ProcessFirst(PPROCESSENTRY32 ppe) const;
    BOOL ProcessNext(PPROCESSENTRY32 ppe) const;
    BOOL ProcessFind(DWORD dwProcessId, PPROCESSENTRY32 ppe) const;

    BOOL ModuleFirst(PMODULEENTRY32 pme) const;
    BOOL ModuleNext(PMODULEENTRY32 pme) const;
    BOOL ModuleFind(PVOID pvBaseAddr, PMODULEENTRY32 pme) const;
    BOOL ModuleFind(PTSTR pszModName, PMODULEENTRY32 pme) const;

    BOOL ThreadFirst(PTHREADENTRY32 pte) const;
    BOOL ThreadNext(PTHREADENTRY32 pte) const;

    BOOL HeapListFirst(PHEAPLIST32 phl) const;
    BOOL HeapListNext(PHEAPLIST32 phl) const;
    int  HowManyHeaps() const;

    // Примечание: функции, оперирующие с блоками памяти в куче, не ссылаются
    // на "снимок", а просто каждый раз с самого начала просматривают кучу
    // процесса. Если исследуемый процесс изменит свою кучу, когда любая
    // из этих функций будет перечислять блоки в его куче, возможно вхождение
    // в бесконечный цикл.
    BOOL HeapFirst(PHEAPENTRY32 phe, DWORD dwProcessID,
        UINT_PTR dwHeapID) const;
    BOOL HeapNext(PHEAPENTRY32 phe) const;
    int  HowManyBlocksInHeap(DWORD dwProcessID, DWORD dwHeapId) const;
    BOOL IsAHeap(HANDLE hProcess, PVOID pvBlock, PDWORD pdwFlags) const;

public:
    static BOOL EnableDebugPrivilege(BOOL fEnable = TRUE);
    static BOOL ReadProcessMemory(DWORD dwProcessID, LPCVOID pvBaseAddress,
        PVOID pvBuffer, DWORD cbRead, PDWORD pdwNumberOfBytesRead = NULL);
};

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

inline CToolhelp::CToolhelp(DWORD dwFlags, DWORD dwProcessID) {

    m_hSnapshot = INVALID_HANDLE_VALUE;
    CreateSnapshot(dwFlags, dwProcessID);
}

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

inline CToolhelp::~~CToolhelp() {

```

см. след. стр.

Рис. 4-6. *продолжение*

```

    if (m_hSnapshot != INVALID_HANDLE_VALUE)
        CloseHandle(m_hSnapshot);
}

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

inline CToolhelp::CreateSnapshot(DWORD dwFlags, DWORD dwProcessID) {

    if (m_hSnapshot != INVALID_HANDLE_VALUE)
        CloseHandle(m_hSnapshot);

    if (dwFlags == 0) {
        m_hSnapshot = INVALID_HANDLE_VALUE;
    } else {
        m_hSnapshot = CreateToolhelp32Snapshot(dwFlags, dwProcessID);
    }
    return(m_hSnapshot != INVALID_HANDLE_VALUE);
}

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

inline BOOL CToolhelp::EnableDebugPrivilege(BOOL fEnable) {

    // передавая приложению полномочия отладчика, мы разрешаем ему
    // видеть информацию о сервисных приложениях
    BOOL fOk = FALSE; // предполагаем худшее
    HANDLE hToken;

    // пытаемся открыть маркер доступа (access token) для этого процесса
    if (OpenProcessToken(GetCurrentProcess(), TOKEN_ADJUST_PRIVILEGES,
        &hToken)) {

        TOKEN_PRIVILEGES tp;
        tp.PrivilegeCount = 1;
        LookupPrivilegeValue(NULL, SE_DEBUG_NAME, &tp.Privileges[0].Luid);
        tp.Privileges[0].Attributes = fEnable ? SE_PRIVILEGE_ENABLED : 0;
        AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
        fOk = (GetLastError() == ERROR_SUCCESS);
        CloseHandle(hToken);
    }
    return(fOk);
}

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

inline BOOL CToolhelp::ReadProcessMemory(DWORD dwProcessID,
    LPCVOID pvBaseAddress, PVOID pvBuffer, DWORD cbRead,
    PDWORD pdwNumberOfBytesRead) {

    return(Toolhelp32ReadProcessMemory(dwProcessID, pvBaseAddress, pvBuffer,
        cbRead, pdwNumberOfBytesRead));
}

```

Рис. 4-6. продолжение

```

////////////////////////////////////
inline BOOL CToolhelp::ProcessFirst(PPROCESSENTRY32 ppe) const {

    BOOL fOk = Process32First(m_hSnapshot, ppe);
    if (fOk && (ppe->th32ProcessID == 0))
        fOk = ProcessNext(ppe); // удаляем "[System Process]" (PID = 0)
    return(fOk);
}

inline BOOL CToolhelp::ProcessNext(PPROCESSENTRY32 ppe) const {

    BOOL fOk = Process32Next(m_hSnapshot, ppe);
    if (fOk && (ppe->th32ProcessID == 0))
        fOk = ProcessNext(ppe); // удаляем "[System Process]" (PID = 0)
    return(fOk);
}

inline BOOL CToolhelp::ProcessFind(DWORD dwProcessId, PPROCESSENTRY32 ppe) const {

    BOOL fFound = FALSE;
    for (BOOL fOk = ProcessFirst(ppe); fOk; fOk = ProcessNext(ppe)) {
        fFound = (ppe->th32ProcessID == dwProcessId);
        if (fFound) break;
    }
    return(fFound);
}

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

inline BOOL CToolhelp::ModuleFirst(PMODULEENTRY32 pme) const {

    return(Module32First(m_hSnapshot, pme));
}

inline BOOL CToolhelp::ModuleNext(PMODULEENTRY32 pme) const {

    return(Module32Next(m_hSnapshot, pme));
}

inline BOOL CToolhelp::ModuleFind(PVOID pvBaseAddr, PMODULEENTRY32 pme) const {

    BOOL fFound = FALSE;
    for (BOOL fOk = ModuleFirst(pme); fOk; fOk = ModuleNext(pme)) {
        fFound = (pme->modBaseAddr == pvBaseAddr);
        if (fFound) break;
    }
    return(fFound);
}

```

см. след. стр.

Рис. 4-6. *продолжение*

```

inline BOOL CToolhelp::ModuleFind(PTSTR pszModName, PMODULEENTRY32 pme) const {
    BOOL fFound = FALSE;
    for (BOOL fOk = ModuleFirst(pme); fOk; fOk = ModuleNext(pme)) {
        fFound = (lstrcmpi(pme->szModule, pszModName) == 0) ||
            (lstrcmpi(pme->szExePath, pszModName) == 0);
        if (fFound) break;
    }
    return(fFound);
}

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

inline BOOL CToolhelp::ThreadFirst(PTHREADENTRY32 pte) const {

    return(Thread32First(m_hSnapshot, pte));
}

inline BOOL CToolhelp::ThreadNext(PTHREADENTRY32 pte) const {

    return(Thread32Next(m_hSnapshot, pte));
}

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

inline int CToolhelp::HowManyHeaps() const {

    int nHowManyHeaps = 0;
    HEAPLIST32 hl = { sizeof(hl) };
    for (BOOL fOk = HeapListFirst(&hl); fOk; fOk = HeapListNext(&hl))
        nHowManyHeaps++;
    return(nHowManyHeaps);
}

inline int CToolhelp::HowManyBlocksInHeap(DWORD dwProcessID,
    DWORD dwHeapID) const {

    int nHowManyBlocksInHeap = 0;
    HEAPENTRY32 he = { sizeof(he) };
    BOOL fOk = HeapFirst(&he, dwProcessID, dwHeapID);
    for (; fOk; fOk = HeapNext(&he))
        nHowManyBlocksInHeap++;
    return(nHowManyBlocksInHeap);
}

inline BOOL CToolhelp::HeapListFirst(PHEAPLIST32 phl) const {

    return(Heap32ListFirst(m_hSnapshot, phl));
}

inline BOOL CToolhelp::HeapListNext(PHEAPLIST32 phl) const {

```

Рис. 4-6. *продолжение*

```

        return(Heap32ListNext(m_hSnapshot, ph1));
    }

inline BOOL CToolhelp::HeapFirst(PHEAPENTRY32 phe, DWORD dwProcessID,
    UINT_PTR dwHeapID) const {

    return(Heap32First(phe, dwProcessID, dwHeapID));
}

inline BOOL CToolhelp::HeapNext(PHEAPENTRY32 phe) const {

    return(Heap32Next(phe));
}

inline BOOL CToolhelp::IsAHeap(HANDLE hProcess, PVOID pvBlock,
    PDWORD pdwFlags) const {

    HEAPLIST32 hl = { sizeof(hl) };
    for (BOOL fOkHL = HeapListFirst(&hl); fOkHL; fOkHL = HeapListNext(&hl)) {
        HEAPENTRY32 he = { sizeof(he) };
        BOOL fOkHE = HeapFirst(&he, hl.th32ProcessID, hl.th32HeapID);
        for (; fOkHE; fOkHE = HeapNext(&he)) {
            MEMORY_BASIC_INFORMATION mbi;
            VirtualQueryEx(hProcess, (PVOID) he.dwAddress, &mbi, sizeof(mbi));
            if (chINRANGE(mbi.AllocationBase, pvBlock,
                (PBYTE) mbi.AllocationBase + mbi.RegionSize)) {

                *pdwFlags = hl.dwFlags;
                return(TRUE);
            }
        }
    }
    return(FALSE);
}

////////////////////////////////////// Конец файла ////////////////////////////////////////

```

Задания

Группу процессов зачастую нужно рассматривать как единую сущность. Например, когда Вы командуете Microsoft Developer Studio собрать проект, он порождает процесс Cl.exe, а тот в свою очередь может создать другие процессы (скажем, для дополнительных проходов компилятора). Но, если Вы пожелаете прервать сборку, Developer 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.

В Windows 2000 введен новый объект ядра — 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* (рис. 5-1). Она включает процесс в задание, которое ограничивает возможность выполнения определенных операций.

WINDOWS 98 Windows 98 не поддерживает задания.

```
void StartRestrictedProcess() {
    // создаем объект ядра "задание"
    HANDLE hjob = CreateJobObject(NULL, NULL);

    // вводим ограничения для процессов в задании

    // сначала определяем некоторые базовые ограничения
    JOBOBJECT_BASIC_LIMIT_INFORMATION jobli = { 0 };
```

Рис. 5-1. Функция *StartRestrictedProcess*

Рис. 5-1. *продолжение*

```

// процесс всегда выполняется с классом приоритета idle
jobli.PriorityClass = IDLE_PRIORITY_CLASS;

// задание не может использовать более одной секунды процессорного времени
jobli.PerJobUserTimeLimit.QuadPart = 10000000; // 1 секунда, выраженная в
// 100-наносекундных интервалах

// два ограничения, которые я налагаю на задание (процесс)
jobli.LimitFlags = JOB_OBJECT_LIMIT_PRIORITY_CLASS
| JOB_OBJECT_LIMIT_JOB_TIME;
SetInformationJobObject(hjob, JobObjectBasicLimitInformation, &jobli,
    sizeof(jobli));

// теперь вводим некоторые ограничения по пользовательскому интерфейсу
JOB_OBJECT_BASIC_UI_RESTRICTIONS jobuir;
jobuir.UIRestrictionsClass = JOB_OBJECT_UILIMIT_NONE; // "замысловатый" нуль

// процесс не имеет права останавливать систему
jobuir.UIRestrictionsClass |= JOB_OBJECT_UILIMIT_EXITWINDOWS;

// процесс не имеет права обращаться к USER-объектам в системе
// (например, к другим окнам)
jobuir.UIRestrictionsClass |= JOB_OBJECT_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;
CreateProcess(NULL, "CMD", NULL, NULL, FALSE,
    CREATE_SUSPENDED, 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;

```

см. след. стр.

Рис. 5-1. *продолжение*

```

DWORD dw = WaitForMultipleObjects(2, h, FALSE, INFINITE);
switch (dw - WAIT_OBJECT_0) {
    case 0:
        // процесс завершился...
        break;
    case 1:
        // лимит процессорного времени исчерпан...
        break;
}
// проводим очистку
CloseHandle(pi.hProcess);
CloseHandle(hjob);
}

```

А теперь я объясню, как работает *StartRestrictedProcess*. Сначала 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 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, GetCurrentProcess());

// 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 cbJobObjectInformationLength);
```

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.

| Вид ограничений | Значение второго параметра | Структура, указываемая в третьем параметре |
|---|--|--|
| Базовые ограничения | <i>JobObjectBasicLimitInformation</i> | JOBOBJECT_BASIC_LIMIT_INFORMATION |
| Расширенные базовые ограничения | <i>JobObjectExtendedLimitInformation</i> | JOBOBJECT_EXTENDED_LIMIT_INFORMATION |
| Базовые ограничения по пользовательскому интерфейсу | <i>JobObjectBasicUIRestrictions</i> | JOBOBJECT_BASIC_UI_RESTRICTIONS |
| Ограничения, связанные с защитой | <i>JobObjectSecurityLimitInformation</i> | 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 *SetInformationJobObject*. Данная структура выглядит так:

```
typedef struct _JOBOBJECT_BASIC_LIMIT_INFORMATION {
    LARGE_INTEGER PerProcessUserTimeLimit;
    LARGE_INTEGER PerJobUserTimeLimit;
    DWORD        LimitFlags;
    DWORD        MinimumWorkingSetSize;
```

см. след. стр.

```

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 5-1.

| Элементы | Описание | Примечание |
|---|--|---|
| <i>PerProcessUser-TimeLimit</i> | Максимальное время в пользовательском режиме, выделяемое каждому процессу (в порциях по 100 нс) | Система автоматически завершает любой процесс, который пытается использовать больше отведенного времени. Это ограничение вводится флагом <code>JOB_OBJECT_LIMIT_PROCESS_TIME</code> в <i>LimitFlags</i> . |
| <i>PerJobUser-TimeLimit</i> | 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 <code>JOB_OBJECT_LIMIT_JOB_TIME</code> в <i>LimitFlags</i> . |
| <i>LimitFlags</i> | Виды ограничений для задания | См. раздел после таблицы. |
| <i>MinimumWorkingSetSize</i> и <i>MaximumWorkingSetSize</i> | 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 <i>MaximumWorkingSetSize</i> , Вы введете жесткое ограничение. Когда размер рабочего набора какого-либо процесса достигнет заданного предела, процесс начнет сбрасывать свои страницы на диск. Вызовы функции <i>SetProcessWorkingSetSize</i> этим процессом будут игнорироваться, если только он не обращается к ней для того, чтобы очистить свой рабочий набор. Это ограничение вводится флагом <code>JOB_OBJECT_LIMIT_WORKINGSET</code> в <i>LimitFlags</i> . |
| <i>ActiveProcessLimit</i> | Максимальное количество процессов, одновременно выполняемых в задании | Любая попытка обойти такое ограничение приведет к завершению нового процесса с ошибкой «not enough quota» («превышение квоты»). Это ограничение вводится флагом <code>JOB_OBJECT_LIMIT_ACTIVE_PROCESS</code> в <i>LimitFlags</i> . |
| <i>Affinity</i> | Подмножество процессоров, на которых можно выполнять процессы этого задания | Для индивидуальных процессов это ограничение можно еще больше детализировать. Вводится флагом <code>JOB_OBJECT_LIMIT_AFFINITY</code> в <i>LimitFlags</i> . |

Таблица 5-1. Элементы структуры `JOBOBJECT_BASIC_LIMIT_INFORMATION`

продолжение

| Элементы | Описание | Примечание |
|------------------------|--|--|
| <i>PriorityClass</i> | Класс приоритета для всех процессов в задании | Вызванная процессом функция <i>SetPriorityClass</i> сообщает об успехе даже в том случае, если на самом деле она не выполнила свою задачу, а <i>GetPriorityClass</i> возвращает класс приоритета, каковой и пытался установить процесс, хотя в реальности его класс может быть совсем другим. Кроме того, <i>SetThreadPriority</i> не может поднять приоритет потоков выше <i>normal</i> , но позволяет понижать его. Это ограничение вводится флагом <i>JOB_OBJECT_LIMIT_PRIORITY_CLASS</i> в <i>LimitFlags</i> . |
| <i>SchedulingClass</i> | 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 <i>JOB_OBJECT_LIMIT_SCHEDULING_CLASS</i> в <i>LimitFlags</i> . |

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_JOB_TIME*. 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 *JOB_OBJECT_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.

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. Кроме того, учтите, что все сказанное здесь относится только к Windows 2000. В будущих версиях Windows планировщик потоков предполагается существенно изменить, чтобы операционная система могла более гибко планировать потоки в заданиях и процессах.

И последнее ограничение, которое заслуживает отдельного упоминания, связано с флагом `JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_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, впуская расходуя системные ресурсы.

Помимо базовых ограничений, Вы можете устанавливать расширенные, для чего применяется структура `JOB_OBJECT_EXTENDED_LIMIT_INFORMATION`:

```
typedef struct _JOB_OBJECT_EXTENDED_LIMIT_INFORMATION {
    JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitInformation;
    IO_COUNTERS IoInfo;
    SIZE_T ProcessMemoryLimit;
    SIZE_T JobMemoryLimit;
    SIZE_T PeakProcessMemoryUsed;
    SIZE_T PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION, *PJOB_OBJECT_EXTENDED_LIMIT_INFORMATION;
```

Как видите, она включает структуру `JOB_OBJECT_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 `JOB_OBJECT_BASIC_UI_RESTRICTIONS` выглядит так:

```
typedef struct _JOB_OBJECT_BASIC_UI_RESTRICTIONS {
    DWORD UIRestrictionsClass;
} JOB_OBJECT_BASIC_UI_RESTRICTIONS, *PJOB_OBJECT_BASIC_UI_RESTRICTIONS;
```

В этой структуре всего один элемент, *UIRestrictionsClass*, который содержит набор битовых флагов, кратко описанных в таблице 5-2.

| Флаг | Описание |
|--|---|
| <code>JOB_OBJECT_UILIMIT_EXITWINDOWS</code> | Запрещает выдачу команд из процессов на выход из системы, завершение ее работы, перезагрузку или выключение компьютера через функцию <i>ExitWindowsEx</i> |
| <code>JOB_OBJECT_UILIMIT_READCLIPBOARD</code> | Запрещает процессам чтение из буфера обмена |
| <code>JOB_OBJECT_UILIMIT_WRITECLIPBOARD</code> | Запрещает процессам стирание буфера обмена |
| <code>JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS</code> | Запрещает процессам изменение системных параметров через <i>SystemParametersInfo</i> |
| <code>JOB_OBJECT_UILIMIT_DISPLAYSETTINGS</code> | Запрещает процессам изменение параметров экрана через <i>ChangeDisplaySettings</i> |
| <code>JOB_OBJECT_UILIMIT_GLOBALATOMS</code> | Предоставляет заданию отдельную глобальную таблицу атомарного доступа (global atom table) и разрешает его процессам пользоваться только этой таблицей |
| <code>JOB_OBJECT_UILIMIT_DESKTOP</code> | Запрещает процессам создание новых рабочих столов или переключение между ними через функции <i>CreateDesktop</i> или <i>SwitchDesktop</i> |
| <code>JOB_OBJECT_UILIMIT_HANDLES</code> | Запрещает процессам в задании использовать USER-объекты (например, <code>HWND</code>), созданные внешними по отношению к этому заданию процессами |

Таблица 5-2. Битовые флаги базовых ограничений по пользовательскому интерфейсу для объекта-задания

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. 5-2 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 3) 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.)

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.

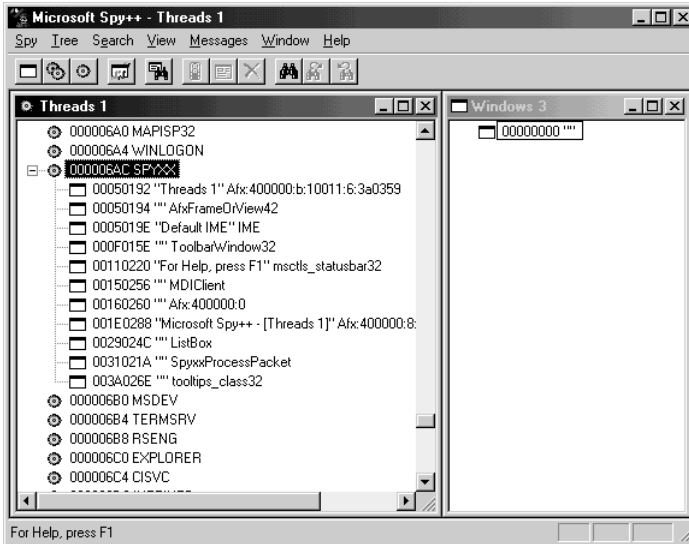


Рис. 5-2. Microsoft Spy++ работает в задании, которому ограничен доступ к описателям USER-объектов

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 fGrant);
```

Параметр *hUserObj* идентифицирует конкретный USER-объект, доступ к 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 работает, если ее вызывает 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 `JOB_OBJECT_SECURITY_LIMIT_INFORMATION` выглядит так:


```
typedef struct _JOB_OBJECT_SECURITY_LIMIT_INFORMATION {
    DWORD SecurityLimitFlags;
    HANDLE JobToken;
    PTOKEN_GROUPS SidsToDisable;
    PTOKEN_PRIVILEGES PrivilegesToDelete;
    PTOKEN_GROUPS RestrictedSids;
} JOB_OBJECT_SECURITY_LIMIT_INFORMATION, *PJOB_OBJECT_SECURITY_LIMIT_INFORMATION;
```

Ее элементы описаны в следующей таблице.

| Элемент | Описание |
|---------------------------|--|
| <i>SecurityLimitFlags</i> | 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 |
| <i>JobToken</i> | Маркер доступа, связываемый со всеми процессами в задании |
| <i>SidsToDisable</i> | Указывает, по каким SID не разрешается доступ |
| <i>PrivilegesToDelete</i> | Определяет привилегии, которые снимаются с маркера доступа |
| <i>RestrictedSids</i> | Задаёт 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 cbJobObjectInformationLength,
    PDWORD pdwReturnLength);
```

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.



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'кэй, 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 *LimitFlags* 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.) 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 `JOB_OBJECT_BASIC_ACCOUNTING_INFORMATION`:

```
typedef struct _JOB_OBJECT_BASIC_ACCOUNTING_INFORMATION {
    LARGE_INTEGER TotalUserTime;
    LARGE_INTEGER TotalKernelTime;
    LARGE_INTEGER ThisPeriodTotalUserTime;
    LARGE_INTEGER ThisPeriodTotalKernelTime;
    DWORD TotalPageFaultCount;
    DWORD TotalProcesses;
    DWORD ActiveProcesses;
    DWORD TotalTerminatedProcesses;
} JOB_OBJECT_BASIC_ACCOUNTING_INFORMATION, *PJOB_OBJECT_BASIC_ACCOUNTING_INFORMATION;
```

Элементы этой структуры кратко описаны в таблице 5-3.

| Элемент | Описание |
|----------------------------------|---|
| <i>TotalUserTime</i> | Процессорное время, израсходованное процессами задания в пользовательском режиме |
| <i>TotalKernelTime</i> | Процессорное время, израсходованное процессами задания в режиме ядра |
| <i>ThisPeriodTotalUserTime</i> | То же, что <i>TotalUserTime</i> , но обнуляется, когда базовые ограничения изменяются вызовом <i>SetInformationJobObject</i> , а флаг <code>JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME</code> не используется |
| <i>ThisPeriodTotalKernelTime</i> | То же, что <i>ThisPeriodTotalUserTime</i> , но относится к процессорному времени, израсходованному в режиме ядра |
| <i>TotalPageFaultCount</i> | Общее количество ошибок страниц, вызванных процессами задания |
| <i>TotalProcesses</i> | Общее число процессов, когда-либо выполнявшихся в этом задании |
| <i>ActiveProcesses</i> | Текущее количество процессов в задании |
| <i>TotalTerminatedProcesses</i> | Количество процессов, завершенных из-за превышения ими отведенного лимита процессорного времени |

Таблица 5-3. Элементы структуры `JOB_OBJECT_BASIC_ACCOUNTING_INFORMATION`

Вы можете извлечь те же сведения вместе с учетной информацией по вводу-выводу, передав *JobObjectBasicAndIoAccountingInformation* во втором параметре и адрес структуры `JOB_OBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION`:

```
typedef struct JOB_OBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
    JOB_OBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
    IO_COUNTERS IoInfo;
} JOB_OBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION;
```

Как видите, она просто возвращает `JOB_OBJECT_BASIC_ACCOUNTING_INFORMATION` и `IO_COUNTERS`. Последняя структура показана на следующей странице.

```
typedef struct _IO_COUNTERS {
    ULONGLONG ReadOperationCount;
    ULONGLONG WriteOperationCount;
    ULONGLONG OtherOperationCount;
    ULONGLONG ReadTransferCount;
    ULONGLONG WriteTransferCount;
    ULONGLONG OtherTransferCount;
} IO_COUNTERS;
```

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 pIoCounters);
```

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)
    // для хранения идентификаторов и структуры
    DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
        (MAX_PROCESS_IDS - 1) * sizeof(DWORD);

    // выделяем этот блок памяти
    PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil = _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 (int x = 0; x < pjobpil->NumberOfProcessIdsInList; x++) {
```

```

    // используем pjobpil->ProcessIdList[x]...
}

// так как для выделения памяти мы вызывали _alloca,
// освободить 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). Рис. 5-3 иллюстрирует некоторые из доступных в системе счетчиков заданий (job object counters), а рис. 5-4 — счетчики, относящиеся к отдельным параметрам заданий (job object details counters). Заметьте, что в задании Jeff содержится четыре процесса: calc, cmd, notepad и wordpad.

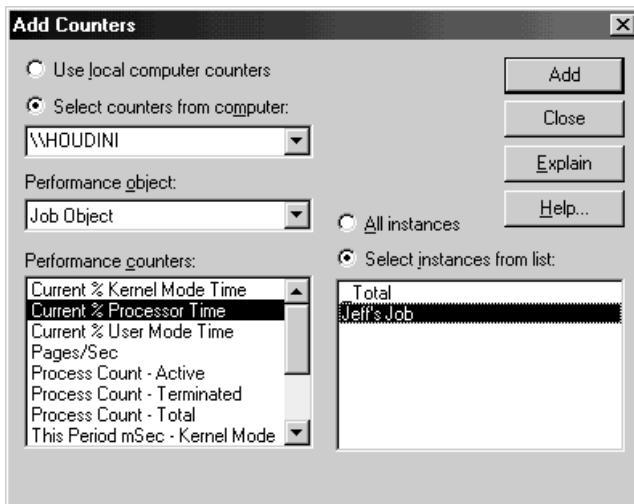


Рис. 5-3. MMC Performance Monitor: счетчики задания

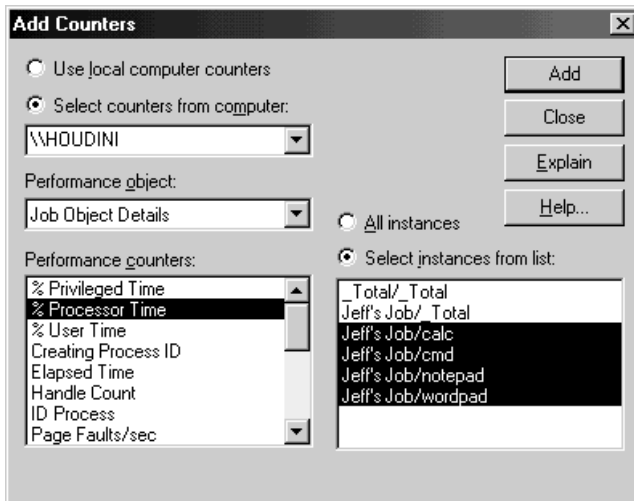


Рис. 5-4. MMC Performance Monitor: счетчики, относящиеся к отдельным параметрам задания

Извлечь сведения из этих счетчиков Вы сможете только для тех заданий, которым были присвоены имена при вызове *CreateJobObject*. По этой причине, наверное, лучше всегда именовать задания, даже если Вы и не собираетесь ссылаться на них по именам из других процессов.

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 глубже.

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 *SetInformationJobObject* 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 Microsoft предпочла сделать по-другому: объект-задание переходит в свободное состояние после того, как исчерпает выделенное ему время. Поскольку большинство заданий начинается 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:

```

JOB_OBJECT_ASSOCIATE_COMPLETION_PORT joacp;
joacp.CompletionKey = 1;    // любое значение, уникально идентифицирующее
                             // это задание
joacp.CompletionPort = hIOCP; // описатель порта завершения, принимающего
                             // уведомления
SetInformationJobObject(hJob, JobObjectAssociateCompletionPortInformation,
    &joacp, sizeof(joacp));

```

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 hIOCP,
    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 (таблица 5-4). В зависимости от конкретного события в **pOverlapped* может возвращаться идентификатор процесса.

| 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_JOB_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 <i>TerminateJobObject</i> |

Таблица 5-4. Уведомления о событиях задания, посылаемые системой связанному с этим заданием порту завершения

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 JOBOBJECT_END_OF_JOB_TIME_INFORMATION
// и инициализируем ее единственный элемент
```

см. след. стр.


```
JOB_OBJECT_END_OF_JOB_TIME_INFORMATION joeojti;
joeojti.EndOfJobTimeAction = JOB_OBJECT_POST_AT_END_OF_JOB;
```

```
// сообщаем заданию, что ему нужно делать по истечении его времени
SetInformationJobObject(hJob, JobObjectEndOfJobTimeInformation,
    &joeojti, sizeof(joeojti));
```

Вы можете указать и другое значение, `JOB_OBJECT_TERMINATE_AT_END_OF_JOB`, но оно задается по умолчанию, еще при создании задания.

Программа-пример JobLab

Эта программа, «05 JobLab.exe» (см. листинг на рис. 5-6), 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. 5-5.

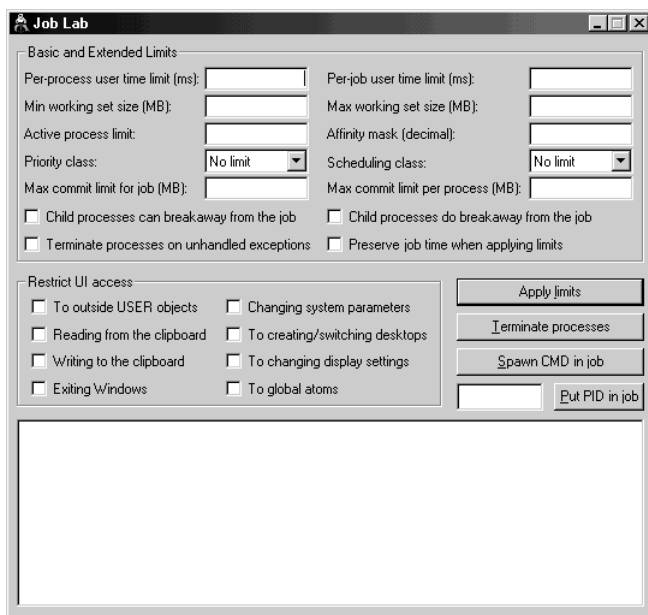


Рис. 5-5. Программа-пример JobLab

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 получаете возможность изменять ограничения, не сбрасывая значения элементов *ThisPeriod-*

TotalUserTime и *ThisPeriodTotalKernelTime* при запросе базовой учетной информации. Этот флажок становится недоступен при наложении ограничений на процессорное время для отдельных заданий.

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.

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*.



JobLab.cpp

```

/*****
Модуль: JobLab.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"
#include <windowsx.h>
#include <process.h>    // для доступа к _beginthreadex
#include <tchar.h>
#include <stdio.h>
#include "Resource.h"
#include "Job.h"

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

CJob  g_job;           // задание
HWND  g_hwnd;          // описатель диалогового окна (доступен всем потокам)
HANDLE g_hIOCP;         // порт завершения, принимающий уведомления от задания
HANDLE g_hThreadIOCP;   // поток порта завершения

```

Рис. 5-6. Программа-пример *JobLab*

см. след. стр.

Рис. 5-6. *продолжение*

```
// ключи завершения, относящиеся к порту завершения
#define COMPKEY_TERMINATE ((UINT_PTR) 0)
#define COMPKEY_STATUS ((UINT_PTR) 1)
#define COMPKEY_JOBOBJECT ((UINT_PTR) 2)

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

DWORD WINAPI JobNotify(PVOID) {
    TCHAR sz[2000];
    BOOL fDone = FALSE;

    while (!fDone) {
        DWORD dwBytesXferred;
        ULONG_PTR CompKey;
        LPOVERLAPPED po;
        GetQueuedCompletionStatus(g_hIOCP,
            &dwBytesXferred, &CompKey, &po, INFINITE);

        // приложение закрывается, выходим из этого потока
        fDone = (CompKey == COMPKEY_TERMINATE);

        HWND hwndLB = FindWindow(NULL, TEXT("Job Lab"));
        hwndLB = GetDlgItem(hwndLB, IDC_STATUS);

        if (CompKey == COMPKEY_JOBOBJECT) {
            lstrcpy(sz, TEXT("--> Notification: "));
            LPTSTR psz = sz + lstrlen(sz);
            switch (dwBytesXferred) {
                case JOB_OBJECT_MSG_END_OF_JOB_TIME:
                    wsprintf(psz, TEXT("Job time limit reached"));
                    break;

                case JOB_OBJECT_MSG_END_OF_PROCESS_TIME:
                    wsprintf(psz, TEXT("Job process (Id=%d) time limit reached"), po);
                    break;

                case JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT:
                    wsprintf(psz, TEXT("Too many active processes in job"));
                    break;

                case JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO:
                    wsprintf(psz, TEXT("Job contains no active processes"));
                    break;

                case JOB_OBJECT_MSG_NEW_PROCESS:
                    wsprintf(psz, TEXT("New process (Id=%d) in job"), po);
                    break;

                case JOB_OBJECT_MSG_EXIT_PROCESS:
                    wsprintf(psz, TEXT("Process (Id=%d) terminated"), po);
                    break;
            }
        }
    }
}
```

Рис. 5-6. продолжение

```

        case JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS:
            wsprintf(psz, TEXT("Process (Id=%d) terminated abnormally"), po);
            break;

        case JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT:
            wsprintf(psz, TEXT("Process (Id=%d) exceeded memory limit"), po);
            break;

        case JOB_OBJECT_MSG_JOB_MEMORY_LIMIT:
            wsprintf(psz,
                TEXT("Process (Id=%d) exceeded job memory limit"), po);
            break;

        default:
            wsprintf(psz, TEXT("Unknown notification: %d"), dwBytesXferred);
            break;
    }
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, psz));
    CompKey = 1; // принудительное обновление списка
                // при получении уведомления

if (CompKey == COMPKEY_STATUS) {
    static int s_nStatusCount = 0;
    _stprintf(sz, TEXT("--> Status Update (%u)"), s_nStatusCount++);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // выводим базовую учетную информацию
    JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION jobai;
    g_job.QueryBasicAccountingInfo(&jobai);

    _stprintf(sz, TEXT("Total Time: User=%I64u, Kernel=%I64u    ")
        TEXT("Period Time: User=%I64u, Kernel=%I64u"),
        jobai.BasicInfo.TotalUserTime.QuadPart,
        jobai.BasicInfo.TotalKernelTime.QuadPart,
        jobai.BasicInfo.ThisPeriodTotalUserTime.QuadPart,
        jobai.BasicInfo.ThisPeriodTotalKernelTime.QuadPart);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    _stprintf(sz, TEXT("Page Faults=%u, Total Processes=%u, ")
        TEXT("Active Processes=%u, Terminated Processes=%u"),
        jobai.BasicInfo.TotalPageFaultCount,
        jobai.BasicInfo.TotalProcesses,
        jobai.BasicInfo.ActiveProcesses,
        jobai.BasicInfo.TotalTerminatedProcesses);
    ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

    // показываем учетную информацию по вводу-выводу
    _stprintf(sz, TEXT("Reads=%I64u (%I64u bytes), ")
        TEXT("Write=%I64u (%I64u bytes), Other=%I64u (%I64u bytes)"),
        jobai.IOInfo.ReadOperationCount, jobai.IOInfo.ReadTransferCount,
        jobai.IOInfo.WriteOperationCount, jobai.IOInfo.WriteTransferCount,

```

см. след. стр.

Рис. 5-6. *продолжение*

```

        jobai.IoInfo.OtherOperationCount, jobai.IoInfo.OtherTransferCount);
        ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

        // сообщаем пиковые значения объема памяти, использованные
        // процессами и заданиями
        JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli;
        g_job.QueryExtendedLimitInfo(&joeli);
        _stprintf(sz, TEXT("Peak memory used: Process=%I64u, Job=%I64u"),
            (__int64) joeli.PeakProcessMemoryUsed,
            (__int64) joeli.PeakJobMemoryUsed);
        ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));

        // выводим список идентификаторов процессов
        DWORD dwNumProcesses = 50, dwProcessIdList[50];
        g_job.QueryBasicProcessIdList(dwNumProcesses,
            dwProcessIdList, &dwNumProcesses);
        _stprintf(sz, TEXT("PIDs: %s"),
            (dwNumProcesses == 0) ? TEXT("(none)") : TEXT(""));
        for (DWORD x = 0; x < dwNumProcesses; x++) {
            _stprintf(_tcschr(sz, 0), TEXT("%d "), dwProcessIdList[x]);
        }
        ListBox_SetCurSel(hwndLB, ListBox_AddString(hwndLB, sz));
    }
}
return(0);
}

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

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

    chSETDLGICONS(hwnd, IDI_JOBLAB);

    // сохраняем описатель нашего окна, чтобы поток
    // порта завершения мог к нему обращаться
    g_hwnd = hwnd;

    HWND hwndPriorityClass = GetDlgItem(hwnd, IDC_PRIORITYCLASS);
    ComboBox_AddString(hwndPriorityClass, TEXT("No limit"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Idle"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Below normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Above normal"));
    ComboBox_AddString(hwndPriorityClass, TEXT("High"));
    ComboBox_AddString(hwndPriorityClass, TEXT("Realtime"));
    ComboBox_SetCurSel(hwndPriorityClass, 0); // по умолчанию - "No limit"

    HWND hwndSchedulingClass = GetDlgItem(hwnd, IDC_SCHEDULINGCLASS);
    ComboBox_AddString(hwndSchedulingClass, TEXT("No limit"));
    for (int n = 0; n <= 9; n++) {
        TCHAR szSchedulingClass[2] = { (TCHAR) (TEXT('0') + n), 0 };
    }
}

```

Рис. 5-6. продолжение

```

        ComboBox_AddString(hwndSchedulingClass, szSchedulingClass);
    }
    ComboBox_SetCurSel(hwndSchedulingClass, 0); // по умолчанию - "No limit"
    SetTimer(hwnd, 1, 10000, NULL); // обновление каждые 10 секунд
    return(TRUE);
}

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

void Dlg_ApplyLimits(HWND hwnd) {
    const int nNanosecondsPerSecond = 1000000000;
    const int nMillisecondsPerSecond = 1000;
    const int nNanosecondsPerMillisecond =
        nNanosecondsPerSecond / nMillisecondsPerSecond;
    BOOL f;
    __int64 q;
    SIZE_T s;
    DWORD d;

    // устанавливаем базовые и расширенные ограничения
    JOBOBJECT_EXTENDED_LIMIT_INFORMATION joeli = { 0 };
    joeli.BasicLimitInformation.LimitFlags = 0;

    q = GetDlgItemInt(hwnd, IDC_PERPROCESSUSERTIMELIMIT, &f, FALSE);
    if (f) {
        joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PROCESS_TIME;
        joeli.BasicLimitInformation.PerProcessUserTimeLimit.QuadPart =
            q * nNanosecondsPerMillisecond / 100;
    }

    q = GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
    if (f) {
        joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_TIME;
        joeli.BasicLimitInformation.PerJobUserTimeLimit.QuadPart =
            q * nNanosecondsPerMillisecond / 100;
    }

    s = GetDlgItemInt(hwnd, IDC_MINWORKINGSETSIZE, &f, FALSE);
    if (f) {
        joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_WORKINGSET;
        joeli.BasicLimitInformation.MinimumWorkingSetSize = s * 1024 * 1024;
        s = GetDlgItemInt(hwnd, IDC_MAXWORKINGSETSIZE, &f, FALSE);
        if (f) {
            joeli.BasicLimitInformation.MaximumWorkingSetSize = s * 1024 * 1024;
        } else {
            joeli.BasicLimitInformation.LimitFlags &= ~JOB_OBJECT_LIMIT_WORKINGSET;
            chMB("Both minimum and maximum working set sizes must be set.\n"
                "The working set limits will NOT be in effect.");
        }
    }
}

```

см. след. стр.

Рис. 5-6. *продолжение*

```
d = GetDlgItemInt(hwnd, IDC_ACTIVEPROCESSLIMIT, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_ACTIVE_PROCESS;
    joeli.BasicLimitInformation.ActiveProcessLimit = d;
}

s = GetDlgItemInt(hwnd, IDC_AFFINITYMASK, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_AFFINITY;
    joeli.BasicLimitInformation.Affinity = s;
}

joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_PRIORITY_CLASS;
switch (ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_PRIORITYCLASS))) {
    case 0:
        joeli.BasicLimitInformation.LimitFlags &=
            ~JOB_OBJECT_LIMIT_PRIORITY_CLASS;
        break;

    case 1:
        joeli.BasicLimitInformation.PriorityClass =
            IDLE_PRIORITY_CLASS;
        break;

    case 2:
        joeli.BasicLimitInformation.PriorityClass =
            BELOW_NORMAL_PRIORITY_CLASS;
        break;

    case 3:
        joeli.BasicLimitInformation.PriorityClass =
            NORMAL_PRIORITY_CLASS;
        break;

    case 4:
        joeli.BasicLimitInformation.PriorityClass =
            ABOVE_NORMAL_PRIORITY_CLASS;
        break;

    case 5:
        joeli.BasicLimitInformation.PriorityClass =
            HIGH_PRIORITY_CLASS;
        break;

    case 6:
        joeli.BasicLimitInformation.PriorityClass =
            REALTIME_PRIORITY_CLASS;
        break;
}
```


Рис. 5-6. продолжение

```

int nSchedulingClass =
    ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_SCHEDULINGCLASS));
if (nSchedulingClass > 0) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_SCHEDULING_CLASS;
    joeli.BasicLimitInformation.SchedulingClass = nSchedulingClass - 1;
}

s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERJOB, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_JOB_MEMORY;
    joeli.JobMemoryLimit = s * 1024 * 1024;
}

s = GetDlgItemInt(hwnd, IDC_MAXCOMMITPERPROCESS, &f, FALSE);
if (f) {
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_PROCESS_MEMORY;
    joeli.ProcessMemoryLimit = s * 1024 * 1024;
}

if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESCANBREAKAWAYFROMJOB))
    joeli.BasicLimitInformation.LimitFlags |= JOB_OBJECT_LIMIT_BREAKAWAY_OK;

if (IsDlgButtonChecked(hwnd, IDC_CHILDPROCESSESDOBREAKAWAYFROMJOB))
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK;

if (IsDlgButtonChecked(hwnd, IDC_TERMINATEPROCESSONEXCEPTIONS))
    joeli.BasicLimitInformation.LimitFlags |=
        JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION;

f = g_job.SetExtendedLimitInfo(&joeli,
    ((joeli.BasicLimitInformation.LimitFlags & JOB_OBJECT_LIMIT_JOB_TIME)
    != 0) ? FALSE :
    IsDlgButtonChecked(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS));
chASSERT(f);

// устанавливаем ограничения, связанные с пользовательским интерфейсом
DWORD jobuir = JOB_OBJECT_UILIMIT_NONE; // "замысловатый" нуль (0)
if (IsDlgButtonChecked(hwnd, IDC_RESTRICTACCESSTOOUTSIDEUSEROBJECTS))
    jobuir |= JOB_OBJECT_UILIMIT_HANDLES;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTREADINGCLIPBOARD))
    jobuir |= JOB_OBJECT_UILIMIT_READCLIPBOARD;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTWRITINGCLIPBOARD))
    jobuir |= JOB_OBJECT_UILIMIT_WRITECLIPBOARD;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTEXITWINDOW))
    jobuir |= JOB_OBJECT_UILIMIT_EXITWINDOWS;

```

см. след. стр.

Рис. 5-6. *продолжение*

```

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTCHANGINGSYSTEMPARAMETERS))
    jobuir |= JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDESKTOPS))
    jobuir |= JOB_OBJECT_UILIMIT_DESKTOP;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTDISPLAYSETTINGS))
    jobuir |= JOB_OBJECT_UILIMIT_DISPLAYSETTINGS;

if (IsDlgButtonChecked(hwnd, IDC_RESTRICTGLOBALATOMS))
    jobuir |= JOB_OBJECT_UILIMIT_GLOBALATOMS;

chVERIFY(g_job.SetBasicUIRestrictions(jobuir));
}

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

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            // пользователь закрывает наше приложение – закрываем и задание
            KillTimer(hwnd, 1);
            g_job.Terminate(0);
            EndDialog(hwnd, id);
            break;

        case IDC_PERJOBUSERTIMELIMIT:
            {
                // устанавливая новое ограничение по времени,
                // нужно сбросить ранее указанное время для задания
                BOOL f;
                GetDlgItemInt(hwnd, IDC_PERJOBUSERTIMELIMIT, &f, FALSE);
                EnableWindow(
                    GetDlgItem(hwnd, IDC_PRESERVEJOBTIMEWHENAPPLYINGLIMITS), !f);
            }
            break;

        case IDC_APPLYLIMITS:
            Dlg_ApplyLimits(hwnd);
            PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
            break;

        case IDC_TERMINATE:
            g_job.Terminate(0);
            PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
            break;

        case IDC_SPAWNCMDINJOB:
            {
                // порождаем процесс командного процессора и помещаем его в задание

```

Рис. 5-6. продолжение

```

        STARTUPINFO si = { sizeof(si) };
        PROCESS_INFORMATION pi;
        TCHAR sz[] = TEXT("CMD");
        CreateProcess(NULL, sz, NULL, NULL,
            FALSE, CREATE_SUSPENDED, NULL, NULL, &si, &pi);
        g_job.AssignProcess(pi.hProcess);
        ResumeThread(pi.hThread);
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
    }
    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
    break;

case IDC_ASSIGNPROCESSTOJOB:
{
    DWORD dwProcessId = GetDlgItemInt(hwnd, IDC_PROCESSID, NULL, FALSE);
    HANDLE hProcess = OpenProcess(
        PROCESS_SET_QUOTA | PROCESS_TERMINATE, FALSE, dwProcessId);
    if (hProcess != NULL) {
        chVERIFY(g_job.AssignProcess(hProcess));
        CloseHandle(hProcess);
    } else chMB("Could not assign process to job.");
    }
    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
    break;
}

}

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

void WINAPI Dlg_OnTimer(HWND hwnd, UINT id) {

    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_STATUS, NULL);
}

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

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_TIMER, Dlg_OnTimer);
        chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
    }

    return(FALSE);
}

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

```

см. след. стр.

Рис. 5-6. *продолжение*

```
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

    // создаем порт завершения, который будет принимать уведомления от задания
    g_hIOCP = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, 0);

    // создаем поток, ждущий порт завершения
    g_hThreadIOCP = chBEGINTHREADEX(NULL, 0, JobNotify, NULL, 0, NULL);

    // создаем объект-задание
    g_job.Create(NULL, TEXT("JobLab"));
    g_job.SetEndOfJobInfo(JOB_OBJECT_POST_AT_END_OF_JOB);
    g_job.AssociateCompletionPort(g_hIOCP, COMPKEY_JOBOBJECT);

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_JOBLAB), NULL, Dlg_Proc);

    // передаем специальный ключ, заставляющий поток
    // порта завершения закончить работу
    PostQueuedCompletionStatus(g_hIOCP, 0, COMPKEY_TERMINATE, NULL);

    // ждем, когда завершится его поток
    WaitForSingleObject(g_hThreadIOCP, INFINITE);

    // проводим должную очистку
    CloseHandle(g_hIOCP);
    CloseHandle(g_hThreadIOCP);

    // ПРИМЕЧАНИЕ: задание закрывается вызовом деструктора g_job
    return(0);
}

/////////////////////// Конец файла /////////////////////////
```

Job.h

```
/******
Модуль: Job.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****//

#pragma once

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

#include <malloc.h>    // для доступа к _alloca

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

class CJob {
public:
    CJob(HANDLE hJob = NULL);
    ~CJob();
```

Рис. 5-6. продолжение

```

operator HANDLE() const { return(m_hJob); }

// функции, создающие или открывающие объект-задание
BOOL Create(LPSECURITY_ATTRIBUTES psa = NULL, LPCTSTR pszName = NULL);
BOOL Open(LPCTSTR pszName, DWORD dwDesiredAccess,
          BOOL fInheritHandle = FALSE);

// функции, манипулирующие объектом-заданием
BOOL AssignProcess(HANDLE hProcess);
BOOL Terminate(UINT uExitCode = 0);

// функции, налагающие ограничения на задания
BOOL SetExtendedLimitInfo(PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli,
                          BOOL fPreserveJobTime = FALSE);
BOOL SetBasicUIRestrictions(DWORD fdwLimits);
BOOL GrantUserHandleAccess(HANDLE hUserObj, BOOL fGrant = TRUE);
BOOL SetSecurityLimitInfo(PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli);

// функции, запрашивающие сведения об ограничениях
BOOL QueryExtendedLimitInfo(PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli);
BOOL QueryBasicUIRestrictions(PDWORD pfdwRestrictions);
BOOL QuerySecurityLimitInfo(PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli);

// функции, запрашивающие статусную информацию о задании
BOOL QueryBasicAccountingInfo(
    PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai);
BOOL QueryBasicProcessIdList(DWORD dwMaxProcesses,
    PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned = NULL);

// функции, работающие с уведомлениями задания
BOOL AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey);
BOOL QueryAssociatedCompletionPort(
    PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp);
BOOL SetEndOfJobInfo(
    DWORD fdwEndOfJobInfo = JOB_OBJECT_TERMINATE_AT_END_OF_JOB);
BOOL QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo);

private:
    HANDLE m_hJob;
};

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

inline CJob::CJob(HANDLE hJob) {

    m_hJob = hJob;
}

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

```

см. след. стр.

Рис. 5-6. *продолжение*

```

inline CJob::~CJob() {

    if (m_hJob != NULL)
        CloseHandle(m_hJob);
}

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

inline BOOL CJob::Create(PSECURITY_ATTRIBUTES psa, PCTSTR pszName) {

    m_hJob = CreateJobObject(psa, pszName);
    return(m_hJob != NULL);
}

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

inline BOOL CJob::Open(
    PCTSTR pszName, DWORD dwDesiredAccess, BOOL fInheritHandle) {

    m_hJob = OpenJobObject(dwDesiredAccess, fInheritHandle, pszName);
    return(m_hJob != NULL);
}

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

inline BOOL CJob::AssignProcess(HANDLE hProcess) {

    return(AssignProcessToJobObject(m_hJob, hProcess));
}

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

inline BOOL CJob::AssociateCompletionPort(HANDLE hIOCP, ULONG_PTR CompKey) {

    JOBOBJECT_ASSOCIATE_COMPLETION_PORT joacp = { (PVOID) CompKey, hIOCP };
    return(SetInformationJobObject(m_hJob,
        JobObjectAssociateCompletionPortInformation, &joacp, sizeof(joacp)));
}

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

inline BOOL CJob::SetExtendedLimitInfo(
    PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli, BOOL fPreserveJobTime) {

    if (fPreserveJobTime)
        pjoeli->BasicLimitInformation.LimitFlags |=
            JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME;

    // если Вы хотите сохранить информацию о времени задания,
    // флаг JOB_OBJECT_LIMIT_JOB_TIME нужно убрать
    const DWORD fdwFlagTest =

```

Рис. 5-6. продолжение

```

        (JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME | JOB_OBJECT_LIMIT_JOB_TIME);

    if ((pjoeli->BasicLimitInformation.LimitFlags & fdwFlagTest)
        == fdwFlagTest) {
        // ошибка, так как указаны два взаимоисключающих флага
        DebugBreak();
    }

    return(SetInformationJobObject(m_hJob,
        JobObjectExtendedLimitInformation, pjoeli, sizeof(*pjoeli)));
}

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

inline BOOL CJob::SetBasicUIRestrictions(DWORD fdwLimits) {

    JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir = { fdwLimits };
    return(SetInformationJobObject(m_hJob,
        JobObjectBasicUIRestrictions, &jobuir, sizeof(jobuir)));
}

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

inline BOOL CJob::SetEndOfJobInfo(DWORD fdwEndOfJobInfo) {

    JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeojti = { fdwEndOfJobInfo };
    joeojti.EndOfJobTimeAction = fdwEndOfJobInfo;
    return(SetInformationJobObject(m_hJob,
        JobObjectEndOfJobTimeInformation, &joeojti, sizeof(joeojti)));
}

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

inline BOOL CJob::SetSecurityLimitInfo(
    PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosli) {

    return(SetInformationJobObject(m_hJob,
        JobObjectSecurityLimitInformation, pjosli, sizeof(*pjosli)));
}

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

inline BOOL CJob::QueryAssociatedCompletionPort(
    PJOBOBJECT_ASSOCIATE_COMPLETION_PORT pjoacp) {

    return(QueryInformationJobObject(m_hJob,
        JobObjectAssociateCompletionPortInformation, pjoacp, sizeof(*pjoacp),
        NULL));
}

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

```

см. след. стр.

Рис. 5-6. *продолжение*

```
inline BOOL CJob::QueryBasicAccountingInfo(
    PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION pjobai) {

    return(QueryInformationJobObject(m_hJob,
        JobObjectBasicAndIoAccountingInformation, pjobai, sizeof(*pjobai),
        NULL));
}

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

inline BOOL CJob::QueryExtendedLimitInfo(
    PJOBOBJECT_EXTENDED_LIMIT_INFORMATION pjoeli) {

    return(QueryInformationJobObject(m_hJob, JobObjectExtendedLimitInformation,
        pjoeli, sizeof(*pjoeli), NULL));
}

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

inline BOOL CJob::QueryBasicProcessIdList(DWORD dwMaxProcesses,
    PDWORD pdwProcessIdList, PDWORD pdwProcessesReturned) {

    // определяем требуемый объем памяти в байтах
    DWORD cb = sizeof(JOBOBJECT_BASIC_PROCESS_ID_LIST) +
        (sizeof(DWORD) * (dwMaxProcesses - 1));

    // выделяем эти байты из стека
    PJOBOBJECT_BASIC_PROCESS_ID_LIST pjobpil =
        (PJOBOBJECT_BASIC_PROCESS_ID_LIST) _alloca(cb);

    // Успешно ли прошла эта операция? Если да, идем дальше.
    BOOL fOk = (pjobpil != NULL);

    if (fOk) {
        pjobpil->NumberOfProcessIdsInList = dwMaxProcesses;
        fOk = ::QueryInformationJobObject(m_hJob, JobObjectBasicProcessIdList,
            pjobpil, cb, NULL);

        if (fOk) {
            // у нас появилась информация, возвращаем ее тому, кто ее запрашивал
            if (pdwProcessesReturned != NULL)
                *pdwProcessesReturned = pjobpil->NumberOfProcessIdsInList;

            CopyMemory(pdwProcessIdList, pjobpil->ProcessIdList,
                sizeof(DWORD) * pjobpil->NumberOfProcessIdsInList);
        }
    }
    return(fOk);
}

////////////////////////////////////
```

Рис. 5-6. продолжение

```

inline BOOL CJob::QueryBasicUIRestrictions(PDWORD pfdwRestrictions) {

    JOBOBJECT_BASIC_UI_RESTRICTIONS jobuir;
    BOOL fOk = QueryInformationJobObject(m_hJob, JobObjectBasicUIRestrictions,
        &jobuir, sizeof(jobuir), NULL);
    if (fOk)
        *pfdwRestrictions = jobuir.UIRestrictionsClass;
    return(fOk);
}

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

inline BOOL CJob::QueryEndOfJobTimeInfo(PDWORD pfdwEndOfJobTimeInfo) {

    JOBOBJECT_END_OF_JOB_TIME_INFORMATION joeojti;
    BOOL fOk = QueryInformationJobObject(m_hJob, JobObjectBasicUIRestrictions,
        &joeojti, sizeof(joeojti), NULL);
    if (fOk)
        *pfdwEndOfJobTimeInfo = joeojti.EndOfJobTimeAction;
    return(fOk);
}

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

inline BOOL CJob::QuerySecurityLimitInfo(
    PJOBOBJECT_SECURITY_LIMIT_INFORMATION pjosl) {

    return(QueryInformationJobObject(m_hJob, JobObjectSecurityLimitInformation,
        pjosl, sizeof(*pjosl), NULL));
}

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

inline BOOL CJob::Terminate(UINT uExitCode) {

    return(TerminateJobObject(m_hJob, uExitCode));
}

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

inline BOOL CJob::GrantUserHandleAccess(HANDLE hUserObj, BOOL fGrant) {

    return(UserHandleGrantAccess(hUserObj, m_hJob, fGrant));
}

///////////////////////////////////////////////////////////////// Конец файла ///////////////////////////////////////////////////////////////////

```

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 потоков выполняется 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 (*WinMain*, *wWinMain*, *main* или *wmain*) 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 службу индексации данных (content indexing service) Windows 2000. 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 2000. 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.
- Нетрудно представить будущую версию компилятора, способную автоматически компилировать файлы исходного кода в паузах, возникающих при наборе текста программы. Тогда предупреждения и сообщения об ошибках появлялись бы практически в режиме реального времени, и Вы тут же видели бы, в чем Вы ошиблись. Самое интересное, что Microsoft Visual Studio в какой-то мере уже умеет это делать, — обратите внимание на секцию ClassView в Workspace.
- 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.
- Web-браузеры способны взаимодействовать с серверами в фоновом режиме. Благодаря этому пользователь может перейти на другой Web-узел, не дожидаясь, когда будут получены результаты с текущего Web-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 Web-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 от того, 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 интерфей-

са (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 работоспособными, и Вы можете пользоваться соответствующими окнами, пока не сделаете что-нибудь такое, из-за чего рухнут и они. (Подробнее о потоках и пользовательском интерфейсе см. главы 26 и 27.)

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 *main*, *wmain*, *WinMain* или *wWinMain*. Если Вы хотите создать вторичный поток, в нем тоже должна быть входная функция, которая выглядит примерно так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    DWORD 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*, *wmain*, *WinMain* или *wWinMain*, — 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;
    DWORD cbStack;
    PTHREAD_START_ROUTINE pfnStartAddr;
    PVOID pvParam;
    DWORD fdwCreate;
    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.



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.

О'кэй, 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 *blInheritHandle* значением TRUE (см. главу 3).

Параметр *cbStack*

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 *cbStack* 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 *cbStack* ненулевое значение, функция резервирует всю указанную Вами память. Ее объем определяется либо значением параметра *cbStack*, либо значением, заданным в ключе /STACK компоновщика (выбирается большее из них). Но передается стеку лишь тот объем памяти, который соответствует значению в *cbStack*. 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 ограничивать 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 *pfnStartAddr* 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 Web-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, SecondThread, (PVOID) &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 *fdwCreate*

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.)



В Windows 2000 и Windows NT 4 в этом параметре можно передавать *NULL* (обычно так и делается). Тем самым Вы сообщаете функции, что Вас не интересует идентификатор потока. Но в Windows 95/98 это приведет к ошибке, так как функция попытается записать идентификатор потока по нулевому адресу, что недопустимо. И поток не будет создан.

Такое несоответствие между операционными системами может создать разработчикам приложений массу проблем. Допустим, Вы пишете и тестируете программу в Windows 2000 (которая создает поток, даже если Вы передаете *NULL* в *pdwThreadId*). Но вот Вы запускаете приложение в Windows 98, и функция *CreateThread*, естественно, дает ошибку. Вывод один: тщательно тестируйте свое приложение во всех операционных системах, в которых оно будет работать.

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.



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.



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 дескриптор 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.



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.

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 завершён 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 ещё вернёмся 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. Давайте приглядимся к этой схеме повнимательнее. Вызов *CreateThread* заставляет систему создать объект ядра «поток». При этом счётчику 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: счётчик 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*.

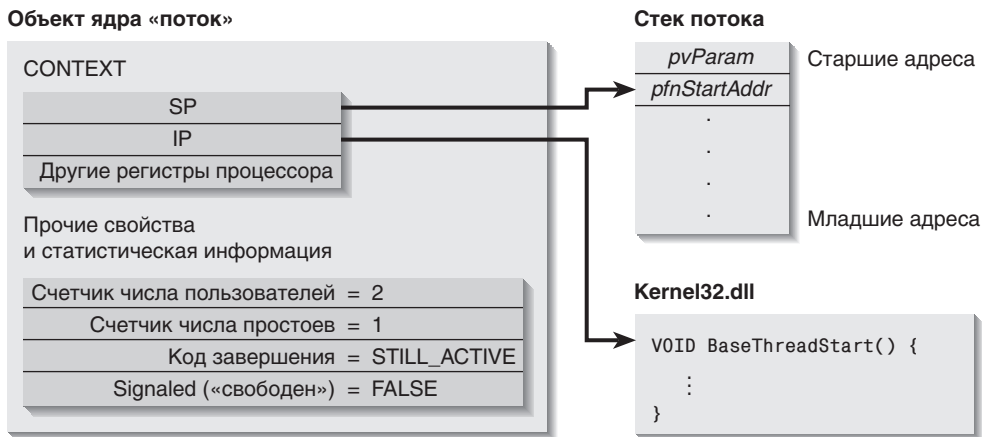


Рис. 6-1. Так создается и инициализируется поток

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 *pfnStartAddr*, а указателю команд — адрес недокументированной (и неэкспортируемой) функции *BaseThreadStart*. Эта функция содержится в модуле `Kernel32.dll`, где, кстати, реализована и функция *CreateThread*.

Вот главное, что делает *BaseThreadStart*:

```
VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
    __try {
        ExitThread((pfnStartAddr)(pvParam));
    }
    __except(UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // ПРИМЕЧАНИЕ: 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 *BaseThreadStart*, именно с этой функции и начнется выполнение потока. Глядя на ее прототип, можно подумать, будто *BaseThreadStart* передаются два параметра, а значит, она вызывается из какой-то другой функции, но это не так. Новый поток просто начинает с нее свою работу. *BaseThreadStart* 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 *BaseThreadStart* — инициализирует нужные регистры процессора.

Когда новый поток выполняет *BaseThreadStart*, происходит 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, *BaseThreadStart* 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 *BaseThreadStart*. Обычно в результате этого появляется окно с каким-нибудь сообщением, и, когда пользователь закрывает его, *BaseThreadStart* вызывает *ExitProcess* и завершает весь процесс, а не только тот поток, в котором произошло исключение.

Обратите внимание, что из *BaseThreadStart* 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 *BaseThreadStart* нет возвращаемого значения — она просто ничего не возвращает.

Кстати, именно благодаря *BaseThreadStart* Ваша функция потока получает возможность вернуть управление по окончании своей работы. *BaseThreadStart*, вызывая функцию потока, заталкивает в стек свой адрес возврата и тем самым сообщает ей, куда надо вернуться. Но сама *BaseThreadStart* 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 другую недокументированную функцию — *BaseProcessStart*. Она почти идентична *BaseThreadStart* и выглядит примерно так:

```
VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr) {
    __try {
        ExitThread((pfnStartAddr)());
    }
    __except(UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // ПРИМЕЧАНИЕ: мы никогда не попадем сюда
}
```

Единственное различие между этими функциями в отсутствии ссылки на параметр *pvParam*. Функция *BaseProcessStart* обращается к стартовому коду библиотеки C/C++, который выполняет необходимую инициализацию, а затем вызывает Вашу входную функцию *main*, *wmain*, *WinMain* или *wWinMain*. Когда входная функция возвращает управление, стартовый код библиотеки C/C++ вызывает *ExitProcess*. Поэтому первичный поток приложения, написанного на C/C++, никогда не возвращается в *BaseProcessStart*.

Некоторые соображения по библиотеке C/C++

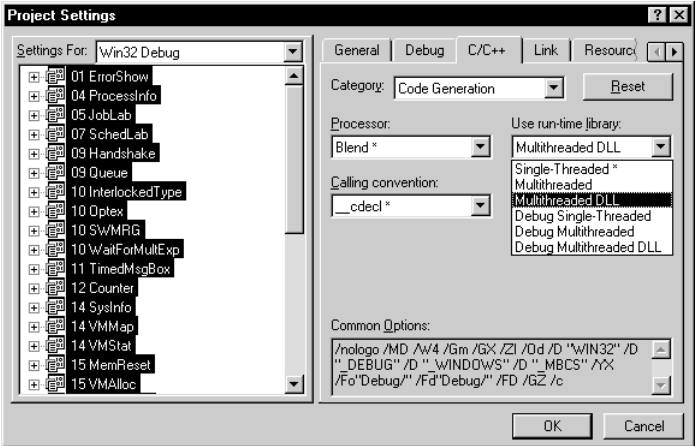
Microsoft поставляет с Visual C++ шесть библиотек C/C++. Их краткое описание представлено в следующей таблице.

| Имя библиотеки | Описание |
|----------------|--|
| LibC.lib | Статически подключаемая библиотека для однопоточных приложений (используется по умолчанию при создании нового проекта) |
| LibCD.lib | Отладочная версия статически подключаемой библиотеки для однопоточных приложений |
| 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 MSVCRT.dll; поддерживает как одно-, так и многопоточные приложения |

продолжение

| Имя библиотеки | Описание |
|----------------|--|
| MSVCRtD.lib | Библиотека импорта для динамического подключения отладочной версии MSVCRtD.dll; поддерживает как одно-, так и многопоточные приложения |

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 ENOMEM:         // 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 выпол-

нения 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 Microsoft 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) (cbStack), \
        (PTHREAD_START) (pfnStartAddr), \
        (void *) (pvParam), \
        (unsigned) (fdwCreate), \
        (unsigned *) (pdwThreadID)))
```

Заметьте, что функция *_beginthreadex* существует только в многопоточных версиях библиотеки C/C++. Связав проект с однопоточной библиотекой, Вы получите от компоновщика сообщение об ошибке «unresolved external symbol». Конечно, это сделано специально, потому что однопоточная библиотека не может корректно работать в многопоточном приложении. Также обратите внимание на то, что при создании нового проекта Visual Studio по умолчанию выбирает однопоточную библиотеку. Этот вариант не самый безопасный, и для многопоточных приложений Вы должны сами выбрать одну из многопоточных версий библиотеки C/C++.

Поскольку Microsoft поставляет исходный код библиотеки C/C++, несложно разобраться в том, что такого делает *_beginthreadex*, чего не делает *CreateThread*. На дистрибутивном компакт-диске Visual Studio ее исходный код содержится в файле Threadex.c. Чтобы не перепечатывать весь код, я решил дать Вам ее версию в псевдокоде, выделив самые интересные места.

```
unsigned long __cdecl _beginthreadex (
    void *psa,
    unsigned cbStack,
    unsigned (__stdcall * pfnStartAddr) (void *),
    void * pvParam,
    unsigned fdwCreate,
    unsigned *pdwThreadId) {

    _ptiddata ptd;          // указатель на блок данных потока
    unsigned long thdl;     // описатель потока

    // выделяется блок данных для нового потока
    if ((ptd = _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;

    // создание нового потока
    thdl = (unsigned long) CreateThread(psa, cbStack,
        _threadstartex, (PVOID) ptd, fdwCreate, pdwThreadId);
    if (thdl == NULL) {
        // 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

    _free_crt(ptd);
    return((unsigned long)0L);
}
```

Несколько важных моментов, связанных с *_beginthreadex*.

- Каждый поток получает свой блок памяти *tiddata*, выделяемый из кучи, которая принадлежит библиотеке C/C++. (Структура *tiddata* определена в файле *Mtdll.h*. Она довольно любопытна, и я привел ее на рис. 6-2.)
- Адрес функции потока, переданный *_beginthreadex*, запоминается в блоке памяти *tiddata*. 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 который указывает *pfnStartAddr*. 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 NULL.

```
struct _tiddata {
    unsigned long    _tid;           /* идентификатор потока */
    unsigned long    _thandle;       /* описатель потока */
    int              _terrno;        /* значение errno */
    unsigned long    _doserrno;      /* значение _doserrno */
    unsigned int     _fpds;          /* сегмент данных Floating Point */
    unsigned long    _holdrand;      /* зародышевое значение для rand() */
    char *           _token;         /* указатель (ptr) на метку strtok() */
#ifdef _WIN32
    wchar_t *        _wtoken;        /* ptr на метку wcstok() */
#endif /* _WIN32 */
    unsigned char *  _mtoken;        /* ptr на метку _mbstok() */

    /* следующие указатели обрабатываются функцией malloc в период выполнения */
    char *           _errmsg;        /* ptr на буфер strerror()/strerror() */
    char *           _namebuf0;      /* ptr на буфер tmpnam() */
#ifdef _WIN32
    wchar_t *        _wnamebuf0;     /* ptr на буфер _wtmpnam() */
#endif /* _WIN32 */
    char *           _namebuf1;      /* ptr на буфер tmpfile() */
#ifdef _WIN32
    wchar_t *        _wnamebuf1;     /* ptr на буфер _wtmpfile() */
#endif /* _WIN32 */
    char *           _asctimebuf;     /* ptr на буфер asctime() */
#ifdef _WIN32
    wchar_t *        _wasctimebuf;    /* ptr на буфер _wasctime() */
#endif /* _WIN32 */
    void *           _gmtimebuf;     /* ptr на структуру gmtime() */
    char *           _cvtbuf;        /* ptr на буфер ecvt()/fcvt */

    /* следующие поля используются кодом _beginthread */
    void *           _initaddr;       /* начальный адрес пользовательского потока */
    void *           _initarg;        /* начальный аргумент пользовательского потока */
}
```

Рис. 6-2. Локальная структура *tiddata* потока, определенная в библиотеке C/C++

Рис. 6-2. продолжение

```

/* следующие три поля нужны для поддержки функции signal и обработки ошибок,
 * возникающих в период выполнения */
void *      _pxcptacttab;    /* ptr на таблицу "исключение-действие" */
void *      _tpxcptinfopts; /* ptr на указатели к информации об исключении */
int         _tfpcode;       /* код исключения для операций над числами
 * с плавающей точкой */

/* 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 *      _curexception; /* текущее исключение */
void *      _curcontext;   /* контекст текущего исключения */
#if defined (_M_MRX000)
void *      _pFrameInfoChain;
void *      _pUnwindContext;
void *      _pExitContext;
int         _MipsPtdDelta;
int         _MipsPtdEpsilon;
#elif defined (_M_PPC)
void *      _pExitContext;
void *      _pUnwindContext;
void *      _pFrameInfoChain;
int         _FrameInfo[6];
#endif /* defined (_M_PPC) */
};

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:

```

static unsigned long WINAPI threadstartex (void* ptd) {
    // Примечание: ptd – это адрес блока tiddata данного потока

    // блок tiddata сопоставляется с данным потоком
    TlsSetValue(__tlsindex, ptd);

    // идентификатор этого потока записывается в tiddata
    ((_ptiddata) ptd)->_tid = GetCurrentThreadId();

    // здесь инициализируется поддержка операций над числами с плавающей точкой
    // (код не показан)
}

```

см. след. стр.

```
// 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
    _exit(GetExceptionCode());
}

// здесь мы тоже никогда не будем, так как в этой функции поток умирает
return(0L);
}
```

Несколько важных моментов, связанных со *_threadstartex*.

- Новый поток начинает выполнение с *BaseThreadStart* (в *Kernel32.dll*), а затем переходит в *_threadstartex*.
- В качестве единственного параметра функции *_threadstartex* передается адрес блока *tiddata* нового потока.
- Windows-функция *TlsSetValue* 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* функцией *_beginthreadex*.
- Значение, возвращаемое функцией потока, считается кодом завершения этого потока. Обратите внимание: *_threadstartex* не возвращается в *BaseThreadStart*. 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();

// высвобождение блока tiddata
_freeptd(ptd);

// завершение потока
ExitThread(retcode);
}

```

Несколько важных моментов, связанных с *_endtbreadex*.

- Библиотечная функция *_getptd* 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 *_endtbreadex* (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 *_beginthbroadex* 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 *_endtbreadex* 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.

Ладно, с функциями все ясно, теперь попробуем проследить, что происходит с глобальными переменными вроде *errno*. В заголовочных файлах C эта переменная определена так:

```

#ifdef _MT || defined(_DLL)
extern int * __cdecl _errno(void);
#define errno (*_errno())
#else /* !defined _MT && !defined _DLL */
extern int errno;
#endif /* !_MT || !_DLL */

```

Создавая многопоточное приложение, надо указывать в командной строке компилятора один из ключей: */MT* (многопоточное приложение) или */MD* (многопоточ-

ная DLL); тогда компилятор определит идентификатор `_MT`. После этого, 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, 9 и 10.)

Конечно, все эти дополнительные операции не могли не отразиться на быстродействии многопоточной версии библиотеки. Поэтому Microsoft, 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 Microsoft 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 *_endtbreadex*, его блок данных не высвободится и произойдет утечка памяти. (Да и кто, интересно, вызовет *_endtbreadex* из потока, созданного с помощью *CreateThread*?)



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 *_endtbreadex*. 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 *_endtbread* 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 *_endtbread* 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:

```
DWORD 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 *_endtbread* уже закрыла описатель нового потока. И, естественно, вызов *CloseHandle* даст ошибку.

Новая функция `_endtbreadex` не закрывает описатель потока, поэтому фрагмент кода, приведенный выше, будет нормально работать (если мы, конечно, заменим вызов `_beginthread` на вызов `_beginthreadex`). И в заключение, напомним еще раз: как только функция потока возвращает управление, `_beginthreadex` самостоятельно вызывает `_endtbreadex`, 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`.

Псевдоописатели можно использовать при вызове функций, которым нужен описатель процесса. Так, поток может запросить все временные показатели своего процесса, вызвав `GetProcessTimes`:

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
GetProcessTimes(GetCurrentProcess(),
    &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
```

Аналогичным образом поток может выяснить собственные временные показатели, вызвав `GetThreadTimes`:

```
FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
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. Вдумаемся в такой фрагмент кода:

```

DWORD WINAPI ParentThread(PVOID pvParam) {
    HANDLE hThreadParent = GetCurrentThread();
    CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
    // далее следует какой-то код...
}

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE) pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, 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 *GetThreadTimes*, 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 fdwAccess,
    BOOL bInheritHandle,
    DWORD fdwOptions);

```

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:

```

DWORD 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
}

```

см. след. стр.

```

        CreateThread(NULL, 0, ChildThread, (PVOID) hThreadParent, 0, NULL);
        // далее следует какой-то код...
    }

DWORD WINAPI ChildThread(PVOID pvParam) {
    HANDLE hThreadParent = (HANDLE) pvParam;
    FILETIME ftCreationTime, ftExitTime, ftKernelTime, ftUserTime;
    GetThreadTimes(hThreadParent,
        &ftCreationTime, &ftExitTime, &ftKernelTime, &ftUserTime);
    CloseHandle(hThreadParent);
    // 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 *GetThreadTimes* дочерний поток вызывает *CloseHandle*, 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 вызывать 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:

```

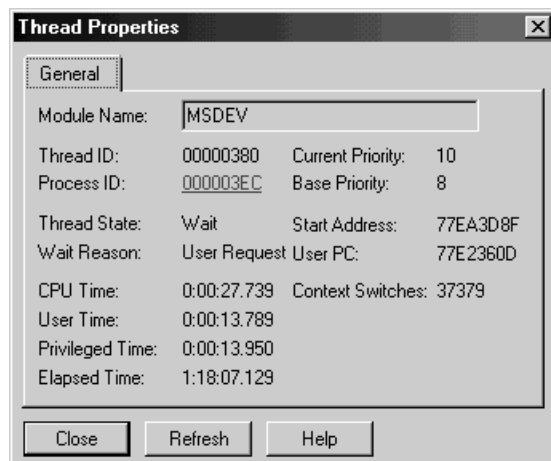
HANDLE hProcess;
DuplicateHandle(
    GetCurrentProcess(),    // описатель процесса, к которому
                           // относится псевдоописатель;
    GetCurrentProcess(),    // псевдоописатель процесса;
    GetCurrentProcess(),    // описатель процесса, к которому
                           // 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

```

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 98 и Windows 2000.

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) 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 37379 раз.



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.



Меня часто спрашивают, как сделать так, чтобы поток гарантированно запустился 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, Microsoft 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.

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 можете 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 — 0xFFFFFFFF.

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). Хотя 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*.

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. Microsoft 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 hSnapshot = CreateToolhelp32Snapshot(
        TH32CS_SNAPTHREAD, dwProcessID);

    if (hSnapshot != INVALID_HANDLE_VALUE) {

        // просматриваем список потоков
        TTHREADENTRY32 te = { sizeof(te) };
        BOOL fOk = Thread32First(hSnapshot, &te);
        for (; fOk; fOk = 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 *dwThreadId*, увеличивает его счетчик пользователей на 1 и возвращает описатель объекта. Получив описатель, я могу передать его в *SuspendThread* (или *ResumeThread*). *OpenThread* имеется только в Windows 2000, поэтому моя функция *SuspendProcess* не будет работать ни в Windows 95/98, ни в Windows NT 4.0.

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 во-

зобновит поток, который собственно 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 сообщить системе 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.

WINDOWS 98 В Windows 98 функция *SwitchToThread* лишь определена, но не реализована.

Определение периодов выполнения потока

Иногда нужно знать, сколько времени затрачивает поток на выполнение той или иной операции. Многие в таких случаях пишут что-то вроде этого:

```
// получаем стартовое время
DWORD dwStartTime = GetTickCount();

// здесь выполняем какой-нибудь сложный алгоритм

// вычитаем стартовое время из текущего
DWORD dwElapsedTime = GetTickCount() - dwStartTime;
```

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:

| Показатель времени | Описание |
|-------------------------------------|--|
| Время создания (creation time) | Абсолютная величина, выраженная в интервалах по 100 нс. Отсчитывается с полуночи 1 января 1601 года по Гринвичу до момента создания потока. |
| Время завершения (exit time) | Абсолютная величина, выраженная в интервалах по 100 нс. 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 нс. Сообщает время, затраченное этим потоком на выполнение кода операционной системы. |
| Время выполнения User (User time) | Относительная величина, выраженная в интервалах по 100 нс. 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, а затем вычитая 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.

WINDOWS 98 К сожалению, в Windows 98 функции *GetThreadTimes* и *GetProcessTimes* определены, но не реализованы. Так что в Windows 98 нет надежного механизма, с помощью которого можно было бы определить, сколько процессорного времени выделяется потоку или процессу.

GetThreadTimes не годится для высокоточного измерения временных интервалов — для этого в Windows предусмотрено две специальные функции:

```
BOOL QueryPerformanceFrequency(LARGE_INTEGER* pliFrequency);
```

```
BOOL QueryPerformanceCounter(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() { QueryPerformanceFrequency(&m_liPerfFreq); Start(); }

    void Start() { QueryPerformanceCounter(&m_liPerfStart); }

    __int64 Now() const { // возвращает число миллисекунд после вызова Start
        LARGE_INTEGER liPerfNow;
        QueryPerformanceCounter(&liPerfNow);
        return(((liPerfNow.QuadPart - m_liPerfStart.QuadPart) * 1000)
            / 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
```

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. В настоящее время такие структуры определены для процессоров Intel, MIPS, Alpha и PowerPC. Соответствующие определения см. в заголовочном файле WinNT.h.»

В документации нет ни слова об элементах этой структуры, набор которых зависит от типа 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., а для процессоров Alpha — *IntV0*, *IntT0*, *IntT1*, *IntS0*, *IntRa*, *IntZero* и др. Структура CONTEXT для процессоров x86 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;
```

см. след. стр.

```

//
// 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 Edi;
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 CONTEXT_EXTENDED_REGISTERS.
// Формат и смысл значений зависят от типа процессора.
//

BYTE ExtendedRegisters[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. (В отличие от *x86*, который при вызове функции помещает адрес возврата в стек, процессор Alpha сохраняет адрес возврата в одном из регистров.) Раздел `CONTEXT_INTEGER` соответствует целочисленным регистрам процессора, `CONTEXT_FLOATING_POINT` — регистрам с плавающей точкой, `CONTEXT_SEGMENTS` — сегментным регистрам (только для *x86*), `CONTEXT_DEBUG_REGISTERS` — регистрам, предназначенным для отладки (только для *x86*), а `CONTEXT_EXTENDED_REGISTERS` — дополнительными регистрами (только для *x86*).

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);
```

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, &Context);

// 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 Microsoft, чаще всего):

```
// сообщаем системе, что нас интересуют
// все значимые регистры
Context.ContextFlags = CONTEXT_FULL;
```

CONTEXT_FULL определен в файле WinNT.h, как показано в таблице.

| Тип процессора | Определение CONTEXT_FULL |
|----------------|--|
| x86 | CONTEXT_CONTROL CONTEXT_INTEGER CONTEXT_SEGMENTS |
| Alpha | CONTEXT_CONTROL CONTEXT_FLOATING_POINT CONTEXT_INTEGER |

После возврата из *GetThreadContext* Вы легко проверите значения любых регистров для потока, но помните, что такой код зависит от типа процессора. В следующей таблице перечислены элементы структуры CONTEXT, соответствующие указателям команд и стека для разных типов процессоров.

| Тип процессора | Указатель команд | Указатель стека |
|----------------|------------------|-----------------|
| x86 | CONTEXT.Eip | CONTEXT.Esp |
| Alpha | CONTEXT.Fir | CONTEXT.IntSp |

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(hThread);

// получаем регистры для контекста потока
Context.ContextFlags = CONTEXT_CONTROL;
GetThreadContext(hThread, &Context);

// устанавливаем указатель команд по своему выбору;
// в нашем примере присваиваем значение 0x00010000
#ifdef _ALPHA_
Context.Fir = 0x00010000;
#elif defined(_X86_)
Context.Eip = 0x00010000;
#else
#error Module contains CPU-specific code; modify and recompile.
#endif

// вносим изменения в регистры потока; ContextFlags
// можно и не инициализировать, так как это уже сделано
Context.ControlFlags = 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 Microsoft 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 Microsoft понимали, что его придется изменять по мере того, как будут расширяться сферы применения компьютеров. Microsoft гарантирует, что наши программы будут работать и в следующих версиях Windows. Как же ей удастся изменять внутреннее устройство системы, не нарушая работоспособность наших программ? Ответ в том, что:

- планировщик документируется не полностью;
- Microsoft не разрешает в полной мере использовать все особенности планировщика;
- Microsoft 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: Microsoft 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.

| 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. |

| Класс приоритета | Описание |
|------------------|--|
| 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. |

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. Может, Вы этого и не знаете, но Explorer выполняется с высоким приоритетом. Большую часть времени его потоки простаивают, готовые пробудиться, как только пользователь нажмет какую-нибудь клавишу или щелкнет кнопку мыши. Пока потоки Explorer простаивают, система не выделяет им процессорное время, что позволяет выполнять потоки с более низким приоритетом. Но вот пользователь нажал, скажем, Ctrl+Esc, и система пробуждает поток Explorer. (Комбинация клавиш Ctrl+Esc попутно открывает меню Start.) Если в данный момент исполняются потоки с более низким приоритетом, они немедленно вытесняются, и начинает работать поток Explorer. Microsoft разработала Explorer именно так потому, что любой пользователь — независимо от текущей ситуации в системе — ожидает мгновенной реакции оболочки на свои команды. В сущности, окна Explorer можно открывать, даже когда все потоки с более низким приоритетом зависают в бесконечных циклах. Обладая более высоким приоритетом, потоки Explorer вытесняют поток, исполняющий бесконечный цикл, и дают возможность закрыть зависший процесс.

Надо отметить высокую степень продуманности Explorer. Основную часть времени он просто «спит», не требуя процессорного времени. Будь это не так, вся система работала бы гораздо медленнее, а многие приложения просто не отзывались бы на действия пользователя.

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 быструю операцию, которую нельзя прерывать ни при каких обстоятельствах.



Процесс с классом приоритета real-time нельзя запустить, если пользователь не имеет привилегии Increase Scheduling Priority. По умолчанию такой привилегией обладает администратор и пользователь с расширенными полномочиями.

Конечно, большинство процессов имеет обычный класс приоритета. В Windows 2000 появилось два новых промежуточных класса — below normal и above normal. Microsoft 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 относятся к классу приоритета процесса. Как обычно, большинство потоков использует обычный приоритет. Относительные приоритеты потоков описаны в следующей таблице.

| 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 |

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 приложений 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 Microsoft не хочет себя ограничивать. И действительно, этот механизм меняется практически в каждой версии системы.

В следующей таблице показано, как формируется уровень приоритета в Windows 2000, 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 относительно класса приоритета его процесса. Изменение класса приоритета процесса не влияет на относительные приоритеты его потоков, но сказывается на уровне их приоритета.

| 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.



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 Microsoft с единственной целью: скрыть от разработчика внутреннее устройство планировщика.



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 следующей таблице.

| Класс приоритета | Идентификатор |
|------------------|-----------------------------|
| Real-time | REALTIME_PRIORITY_CLASS |
| High | HIGH_PRIORITY_CLASS |
| Above normal | ABOVE_NORMAL_PRIORITY_CLASS |
| 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_PRIORITY_CLASS);
```

Парная ей функция *GetPriorityClass* позволяет узнать класс приоритета любого процесса:

```
DWORD GetPriorityClass(HANDLE hProcess);
```

Она возвращает, как Вы догадываетесь, один из ранее перечисленных флагов.

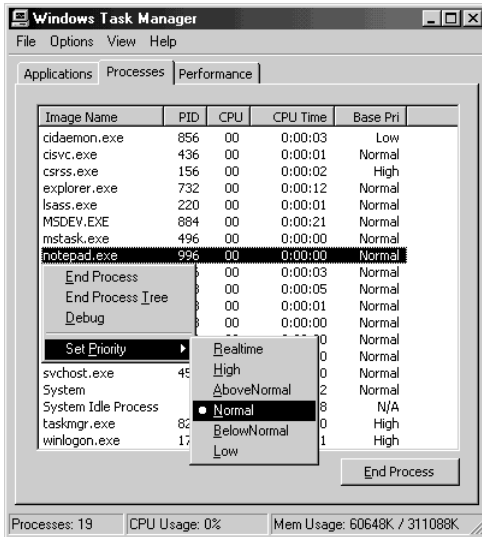
При запуске из оболочки командного процессора начальный приоритет программы тоже обычный. Однако, запуская ее командой *Start*, можно указать ключ, определяющий начальный приоритет. Так, следующая команда, введенная в оболочке командного процессора, заставит систему запустить приложение *Calculator* 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.

WINDOWS 98 В Windows 98 команда *Start* не поддерживает ни один из этих ключей. Из оболочки командного процессора Windows 98 процессы всегда запускаются с классом приоритета *normal*.

Task Manager в Windows 2000 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:

```
BOOL SetThreadPriority(
    HANDLE hThread,
    int nPriority);
```

Разумеется, параметр *hThread* указывает на поток, чей приоритет Вы хотите изменить, а через *nPriority* передается один из идентификаторов (см. таблицу ниже).

| 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, &dwThreadId);
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.



Ни одна Windows-функция не возвращает уровень приоритета потока. Такая ситуация создана преднамеренно. Вспомните, что Microsoft может в любой момент изменить алгоритм распределения процессорного времени. Поэтому при разработке приложений не стоит 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 Microsoft 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 даст приложению нарушить работу операционной системы. И, кстати, система никогда не меняет приоритет потоков с уровнями реального времени (от 16 до 31).

Некоторые разработчики жаловались, что динамическое изменение приоритета системой отрицательно сказывается на производительности их приложений, и поэтому Microsoft добавила две функции, позволяющие отключать этот механизм:

```
BOOL SetProcessPriorityBoost(
    HANDLE hProcess,
    BOOL DisablePriorityBoost);

BOOL SetThreadPriorityBoost(
    HANDLE hThread,
    BOOL DisablePriorityBoost);
```

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 pDisablePriorityBoost);

BOOL GetThreadPriorityBoost(
    HANDLE hThread,
    PBOOL pDisablePriorityBoost);
```

Каждой из этих двух функций Вы передаете описатель нужного процесса или потока и адрес переменной типа BOOL, в которой и возвращается результат.

WINDOWS 98 В Windows 98 эти четыре функции определены, но не реализованы, и при вызове любой из них возвращается FALSE. Последующий вызов *GetLastError* дает `ERROR_CALL_NOT_IMPLEMENTED`.

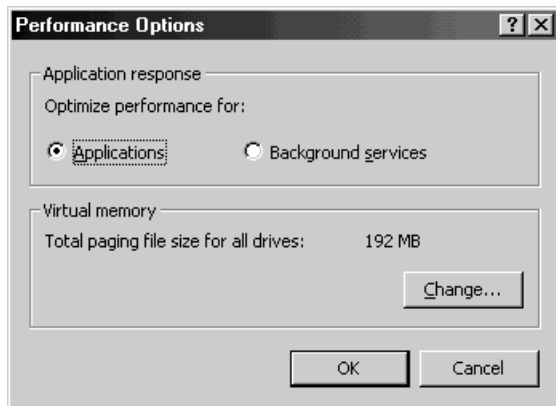
Есть еще одна ситуация, в которой система динамически повышает приоритет потока. Представьте, что поток с приоритетом 4 готов к выполнению, но не может получить доступ к процессору из-за того, что его постоянно занимают потоки с приоритетом 8. Это типичный случай «голодания» потока с более низким приоритетом. 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

ся активным, система выделяет его потокам более длительные кванты времени. Такая регулировка применяется только к процессам с классом приоритета normal.

Windows 2000 позволяет модифицировать работу этого механизма подстройки. Щелкнув кнопку Performance Options на вкладке Advanced диалогового окна System Properties, Вы открываете следующее окно.



Переключатель Applications включает подстройку планировщика для активного процесса, а переключатель Background Services — выключает (в этом случае оптимизируется выполнение фоновых сервисов). В Windows 2000 Professional по умолчанию выбирается переключатель Applications, а в остальных версиях Windows 2000 — переключатель Background Services, так как серверы редко используются в интерактивном режиме.

Windows 98 тоже позволяет подстраивать распределение процессорного времени для потоков активного процесса с классом приоритета normal. Когда процесс этого класса становится активным, система повышает на 1 приоритет его потоков, если их исходные приоритеты были lowest, below normal, normal, above normal или highest; приоритет потоков idle или time-critical не меняется. Поэтому поток с относительным приоритетом normal в активном процессе с классом приоритета normal имеет уровень приоритета 9, а не 8. Когда процесс вновь становится фоновым, приоритеты его потоков автоматически возвращаются к исходным уровням.

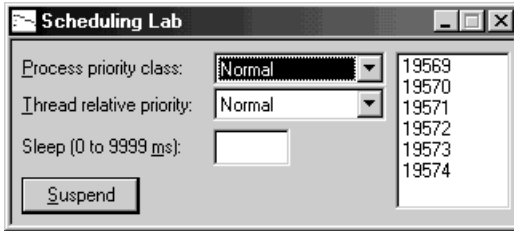
WINDOWS 98 Windows 98 не предусматривает возможности настройки этого механизма, так как не рассчитана на работу в качестве выделенного сервера.

Причина для таких изменений активных процессов очень проста: система дает им возможность быстрее реагировать на пользовательский ввод. Если бы приоритеты их потоков не менялись, то и обычный процесс фоновой печати, и обычный, но активный процесс, принимающий пользовательский ввод, — оба одинаково конкурировали бы за процессорное время. И тогда пользователь, набирая текст в активном приложении, заметил бы, что текст появляется на экране какими-то рывками. Но благодаря тому, что система повышает уровни приоритета потоков активного процесса, они получают преимущество над потоками обычных фоновых процессов.

Программа-пример Scheduling Lab

Эта программа, «07 SchedLab.exe» (см. листинг на рис. 7-1), 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.

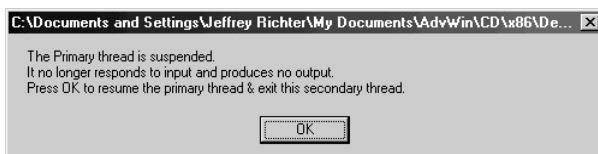


Изначально первичный поток работает очень активно, и степень использования процессора подскакивает до 100%. Все, чем он занимается, — постоянно увеличивает исходное значение на 1 и выводит текущее значение в крайнее справа окно списка. Все эти числа не несут никакой смысловой информации; их появление 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 приостанавливать его хотя бы на 1 мс и посмотрите, сколько процессорного времени это позволит сэкономить. На своем ноутбуке с процессором Pentium II 300 МГц, 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

```

/*****
Модуль: SchedLab.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.H"          /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>            // для доступа к _beginthreadex
#include "Resource.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);

    // во избежание взаимной блокировки после ResumeThread вызываем EnableWindow
    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);

    // инициализируем классы приоритетов процесса
    HWND hwndCtl = GetDlgItem(hwnd, IDC_PROCESSPRIORITYCLASS);

```

Рис. 7-1. Программа-пример SchedLab

Рис. 7-1. продолжение

```
int n = ComboBox_AddString(hwndCtl, TEXT("High"));
ComboBox_SetItemData(hwndCtl, n, HIGH_PRIORITY_CLASS);

// запоминаем текущий класс приоритета своего процесса
DWORD dwpc = GetPriorityClass(GetCurrentProcess());

if (SetPriorityClass(GetCurrentProcess(), BELOW_NORMAL_PRIORITY_CLASS)) {

    // эта система поддерживает класс приоритета below normal

    // восстанавливаем исходный класс приоритета
    SetPriorityClass(GetCurrentProcess(), dwpc);

    // добавляем класс above normal
    n = ComboBox_AddString(hwndCtl, TEXT("Above normal"));
    ComboBox_SetItemData(hwndCtl, n, ABOVE_NORMAL_PRIORITY_CLASS);

    dwpc = 0; // данная система поддерживает класс below normal
}

int nNormal = n = ComboBox_AddString(hwndCtl, TEXT("Normal"));
ComboBox_SetItemData(hwndCtl, n, NORMAL_PRIORITY_CLASS);

if (dwpc == 0) {

    // эта система поддерживает класс приоритета below normal

    // добавляем класс below normal
    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);

// инициализируем относительные приоритеты потоков
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);
```

см. след. стр.

Рис. 7-1. *продолжение*

```

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); // максимум 9999 мс

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:
            // во избежание взаимной блокировки вызываем EnableWindow
            // до создания потока, который вызовет SuspendThread
            EnableWindow(hwndCtl, FALSE);

            HANDLE hThreadPrimary;
            DuplicateHandle(GetCurrentProcess(), GetCurrentThread(),
                GetCurrentProcess(), &hThreadPrimary,
                THREAD_SUSPEND_RESUME, FALSE, DUPLICATE_SAME_ACCESS);
            DWORD dwThreadId;

```

Рис. 7-1. продолжение

```

        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);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR 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 позволяет вовремя реагировать на ввод с клавиатуры
            if (!IsDialogMessage(hwnd, &msg)) {

                if (msg.message == WM_QUIT) {
                    fQuit = TRUE; // принято сообщение WM_QUIT, выходим из цикла
                } else {
                    // принято сообщение, отличное от WM_QUIT;
                    // интерпретируем его и передаем адресату
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                }
            } // if (!IsDialogMessage())

        } else {
            // помещаем число в окно списка
            static int s_n = -1;
            TCHAR sz[20];
            wsprintf(sz, TEXT("%u"), ++s_n);
            HWND hwndWork = GetDlgItem(hwnd, IDC_WORK);
            ListBox_SetCurSel(hwndWork, ListBox_AddString(hwndWork, sz));
        }
    }
}

```

см. след. стр.

Рис. 7-1. *продолжение*

```
// при переполнении списка удаляем лишние строки
while (ListBox_GetCount(hwndWork) > 100)
    ListBox_DeleteString(hwndWork, 0);

// определяем, сколько нужно "спать" потоку
int nSleep = GetDlgItemInt(hwnd, IDC_SLEEPTIME, NULL, FALSE);
if (chINRANGE(1, nSleep, 9999))
    Sleep(nSleep);
}
}
DestroyWindow(hwnd);
return(0);
}

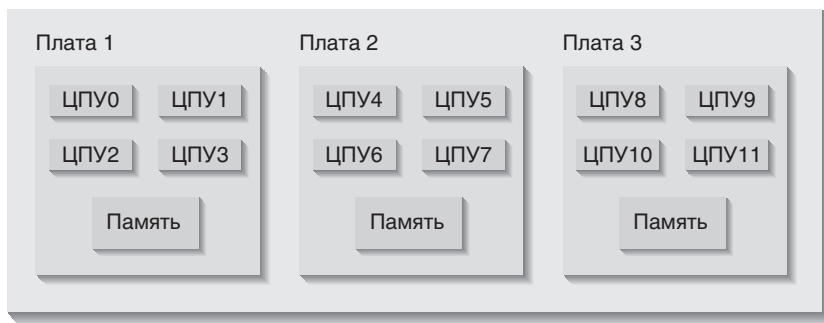
/////////////////////////////////////// Конец файла /////////////////////////////////////////
```

Привязка потоков к процессорам

По умолчанию Windows 2000 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.

Машина с архитектурой NUMA



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,
    DWORD_PTR dwProcessAffinityMask);
```

В первом параметре, *hProcess*, передается описатель процесса. Второй параметр, *dwProcessAffinityMask*, — 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:

```
BOOL GetProcessAffinityMask(
    HANDLE hProcess,
    PDWORD_PTR pdwProcessAffinityMask,
    PDWORD_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.

WINDOWS 98

В Windows 98, которая использует только один процессор независимо от того, сколько их на самом деле, *GetProcessAffinityMask* всегда возвращает в обеих переменных значение 1.

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:

```
DWORD_PTR SetThreadAffinityMask(
    HANDLE hThread,
    DWORD_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);
```

WINDOWS 98 В Windows 98, которая использует только один процессор независимо от того, сколько их на самом деле, параметр *dwThreadAffinityMask* всегда должен быть равен 1.

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.



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.

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. Передавая в нем константу `MAXIMUM_PROCESSORS` (в `WinNT.h` она определена как 32), 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, например, таким кодом:

```
// 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; // нам нужны процессоры 0 и 1

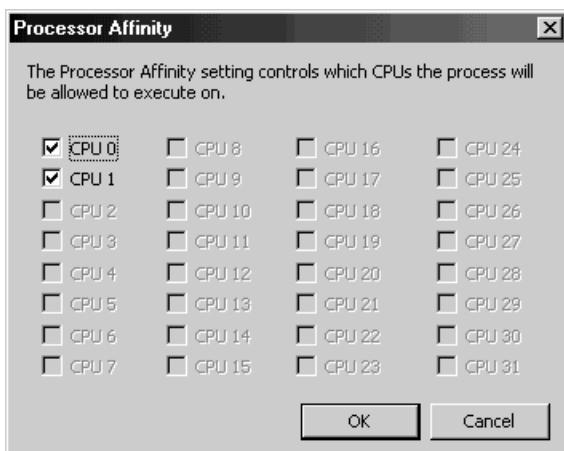
// сохраняем новую информацию о загрузочной конфигурации
SetImageConfigInformation(pLoadedImage, &ilcd);

// выгружаем EXE-файл из памяти
ImageUnload(pLoadedImage);
```

Детально описывать эти функции я не стану — при необходимости Вы найдете их в документации Platform SDK. Кроме того, Вы можете использовать утилиту `ImageCfг.exe`, которая позволяет изменять некоторые флаги в заголовке исполняемого модуля. Подсказку по ее применению Вы получите, запустив `ImageCfг.exe` без ключей.

Указав при запуске `ImageCfг` 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 ключ `-u`, который сообщает системе, что исполняемый файл может выполняться исключительно на однопроцессорной машине.

И, наконец, привязку процесса к процессорам можно изменять с помощью `Task Manager` в `Windows 2000`. В многопроцессорных системах в контекстном меню для процесса появляется команда `Set Affinity` (ее нет на компьютерах с одним процессором). Выбрав эту команду, Вы откроете показанное ниже диалоговое окно и выберите конкретные процессоры для данного процесса.



WINDOWS 2000 При запуске Windows 2000 на машине с процессорами типа x86 можно ограничить число процессоров, используемых системой. В процессе загрузки система считывает файл Boot.ini, который находится в корневом каталоге загрузочного диска. Вот как он выглядит на моем компьютере с двумя процессорами:

```
[boot loader]
timeout=2
default=multi(0)disk(0)rdisk(0)partition(1)\WINNT
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows 2000 Server"
    /fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINNT="Windows 2000 Server"
    /fastdetect /NumProcs=1
```

Этот файл создается при установке Windows 2000; последнюю запись я добавил сам (с помощью Notepad). Она заставляет систему использовать только один процессор. Ключ /NumProcs=1 — как раз то зелье, которое и вызывает все эти магические превращения. Я пользуюсь им иногда для отладки. (Но обычно работаю со всеми своими процессорами.)

Заметьте, что ключи перенесены на отдельные строки с отступом лишь для удобства чтения. На самом деле ключи и путь от загрузочного раздела жесткого диска должны находиться на одной строке.

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 самонадежности 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 *Interlocked*-функций

Большая 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(PVOID pvParam) {
    g_x++;
    return(0);
}

DWORD WINAPI ThreadFunc2(PVOID 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

MOV EAX, [g_x]    ; поток 2: в регистр помещается 1
INC EAX           ; поток 2: значение регистра увеличивается до 2
MOV [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 даст 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*:

```
LONG InterlockedExchangeAdd(
    PLONG plAddend,
    LONG lIncrement);
```

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. На платформе Alpha *Interlocked*-функции действуют примерно так:

1. Устанавливают специальный битовый флаг процессора, указывающий, что данный адрес памяти сейчас занят.
2. Считывают значение из памяти в регистр.
3. Изменяют значение в регистре.
4. Если битовый флаг сброшен, повторяют операции, начиная с п. 2. В ином случае значение из регистра помещается обратно в память.

Вас, наверное, удивило, с какой это стати битовый флаг может оказаться сброшенным? Все очень просто. Его может сбросить другой процессор в системе, пытаясь модифицировать тот же адрес памяти, а это заставляет *Interlocked*-функции вернуться в п. 2.

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.)

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 две функции из этого семейства:

```
LONG InterlockedExchange(
    PLONG plTarget,
    LONG lValue);

PVOID InterlockedExchangePointer(
    PVOID* ppvTarget,
    PVOID 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 (spinlock):

```
// глобальная переменная, используемая как индикатор того, занят ли разделяемый ресурс
BOOL g_fResourceInUse = FALSE;
:
```

```

void Func1() {
    // ожидаем доступа к ресурсу
    while (InterlockedExchange(&g_fResourceInUse, 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. «Крутятся» 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 на вызов *SwitchToThread* (эта функция в Windows 98 не доступна). 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 вой-

дет в цикл, ведь тогда он будет впустую тратить процессорное время. О спин-блокировке мы еще поговорим в этой главе. Кроме того, в главе 10 я покажу, как использовать спин-блокировку на практике.

Последняя пара *Interlocked*-функций выглядит так:

```
PVOID InterlockedCompareExchange(
    PLONG plDestination,
    LONG lExchange,
    LONG lComparand);

PVOID InterlockedCompareExchangePointer(
    PVOID* ppvDestination,
    PVOID pvExchange,
    PVOID 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, LONG lComparand) {

    LONG 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.

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.

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 придется 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 CUSTINFO {
    DWORD    dwCustomerID;        // в основном "только для чтения"
    int      nBalanceDue;         // для чтения и записи
    char     szName[100];         // в основном "только для чтения"
    FILETIME ftLastOrderDate;     // для чтения и записи
};
```

А это усовершенствованная версия той же структуры:

```
// определяем размер кэш-линии используемого процессора
#ifdef _X86_
#define CACHE_ALIGN 32
#endif
#ifdef _ALPHA_
#define CACHE_ALIGN 64
#endif
#ifdef _IA64_
#define CACHE_ALIGN ??
#endif

#define CACHE_PAD(Name, BytesSoFar) \
    BYTE Name[CACHE_ALIGN - ((BytesSoFar) % CACHE_ALIGN)]

struct CUSTINFO {
    DWORD    dwCustomerID;        // в основном "только для чтения"
    char     szName[100];         // в основном "только для чтения"

    // принудительно помещаем следующие элементы в другую кэш-линию
    CACHE_PAD(bPad1, sizeof(DWORD) + 100);

    int      nBalanceDue;         // для чтения и записи
    FILETIME ftLastOrderDate;     // для чтения и записи

    // принудительно помещаем следующую структуру в другую кэш-линию
    CACHE_PAD(bPad2, sizeof(int) + sizeof(FILETIME));
};
```

Макрос `CACHE_ALIGN` неплох, но не идеален. Проблема в том, что байтовый размер каждого элемента придется вводить в макрос вручную, а при добавлении, перемещении или удалении элемента структуры — еще и модифицировать вызов макроса `CACHE_PAD`. В следующих версиях компилятор Microsoft C/C++ будет поддерживать новый синтаксис, упрощающий выравнивание элементов структур. Это будет что-то вроде `__declspec(align(32))`.



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 WinMain(...) {
    CreateThread(..., RecalcFunc, ...);
    :
    // ждем завершения пересчета
    while (!g_fFinishedCalculation)
        ;
    :
}
```

см. след. стр.

```
DWORD WINAPI RecalcFunc(PVOID pvParam) {
    // выполняем пересчет
    :
    g_fFinishedCalculation = TRUE;
    return(0);
}
```

Как видите, первичный поток (он исполняет функцию *WinMain*) 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 *WinMain*, а просто отправили в спячку, это позволило бы системе отдать 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:

```
MOV      Reg0, [g_fFinishedCalculation] ; копируем значение в регистр
Label:   TEST   Reg0, 0                  ; равно ли оно нулю?
JMP      Reg0 == 0, Label                ; в регистре находится 0, повторяем цикл
...      ; в регистре находится ненулевое значение
          ; (выходим из цикла)
```

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_ResourceInUse* 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 одновременно 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 MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];

DWORD WINAPI FirstThread(PVOID pvParam) {

    while (g_nIndex < MAX_TIMES) {
        g_dwTimes[g_nIndex] = GetTickCount();
        g_nIndex++;
    }
    return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {

    while (g_nIndex < MAX_TIMES) {
        g_nIndex++;
        g_dwTimes[g_nIndex - 1] = GetTickCount();
    }
    return(0);
}
```

Здесь предполагается, что функции обоих потоков дают одинаковый результат, хоть они и закодированы с небольшими различиями. Если бы исполнялась только функция *FirstThread*, она заполнила бы массив *g_dwTimes* набором чисел с возрастающими значениями. Это верно и в отношении *SecondThread* — если бы она тоже исполнялась независимо. В идеале обе функции даже при одновременном выполнении должны бы по-прежнему заполнять массив тем же набором чисел. Но в нашем коде возникает проблема: массив *g_dwTimes* не будет заполнен, как надо, потому что функции обоих потоков одновременно обращаются к одним и тем же глобальным переменным. Вот как это может произойти.

Допустим, мы только что начали исполнение обоих потоков в системе с одним процессором. Первым включился в работу второй поток, т. е. функция *SecondThread* (что вполне вероятно), и только она успела увеличить счетчик *g_nIndex* до 1, как си-

стема вытеснила ее поток и перешла к исполнению *FirstThread*. Та заносит в *g_dwTimes[1]* показания системного времени, и процессор вновь переключается на исполнение второго потока. *SecondThread* теперь присваивает элементу *g_dwTimes[1 - 1]* новые показания системного времени. Поскольку эта операция выполняется позже, новые показания, естественно, выше, чем записанные в элемент *g_dwTimes[1]* функцией *FirstThread*. Отметьте также, что сначала заполняется первый элемент массива и только потом нулевой. Таким образом, данные в массиве оказываются ошибочными.

Согласен, пример довольно надуманный, 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 добавляют 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 MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
CRITICAL_SECTION g_cs;

DWORD WINAPI FirstThread(PVOID pvParam) {

    for (BOOL fContinue = TRUE; fContinue; ) {
        EnterCriticalSection(&g_cs);
        if (g_nIndex < MAX_TIMES) {
            g_dwTimes[g_nIndex] = GetTickCount();
            g_nIndex++;
        } else fContinue = FALSE;
        LeaveCriticalSection(&g_cs);
    }
    return(0);
}

DWORD WINAPI SecondThread(PVOID pvParam) {

    for (BOOL fContinue = TRUE; fContinue; ) {
        EnterCriticalSection(&g_cs);
        if (g_nIndex < MAX_TIMES) {
            g_nIndex++;
            g_dwTimes[g_nIndex - 1] = GetTickCount();
        } else fContinue = FALSE;
        LeaveCriticalSection(&g_cs);
    }
    return(0);
}
```

Я создал экземпляр структуры данных *CRITICAL_SECTION* — *g_cs*, а потом «обернул» весь код, работающий с разделяемым ресурсом (в нашем примере это строки с *g_nIndex* и *g_dwTimes*), 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 одновременно 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».



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 переменных *g_nIndex* и *g_dwTimes* 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 удастся 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. Однако в главе 10 я продемонстрирую Вам свой синхронизирующий объект, который я назвал оптексом. На его примере Вы увидите, как реализуются критические секции на уровне операционной системы и как этот объект работает с потоками разных процессов.

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` не относится к недокументированным структурам, Microsoft полагает, что Вам незачем знать, как она устроена. И это правильно. Для нас она является своего рода черным 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 `WinNT.h` как `RTL_CRITICAL_SECTION`, а тип структуры `RTL_CRITICAL_SECTION` определен в файле `WinBase.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 в 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:

```
VOID 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* без 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».

WINDOWS 2000

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:

```
BOOL 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`.

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*.

WINDOWS 2000 В Windows 98 функция *TryEnterCriticalSection* определена, но не реализована. При ее вызове всегда возвращается FALSE.

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 многопроцессорной 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_SECTION 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 со-

стояние 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.

Как реализовать критические секции с применением спин-блокировки, я покажу в главе 10.

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 удастся, 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 удастся.

В работе с критическими секциями может возникнуть еще одна проблема. Когда за доступ к критической секции конкурируют два и более потоков, она использует объект ядра «событие». (Я покажу, как работать с этим объектом при описании C++-класса COptex в главе 10.) 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.

Но если потоки все же будут конкурировать за критическую секцию в условиях нехватки памяти, система не сможет создать нужный объект ядра. И тогда *EnterCriticalSection* возбудит 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 уста-

новленным старшим битом. Тогда функция создает объект «событие» и сопоставляет его с критической секцией. Если создать объект не удастся, 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.)

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

Если в Вашей программе имеется несколько независимых структур данных, создавайте для каждой из них отдельный экземпляр структуры CRITICAL_SECTION. Это лучше, чем защищать все разделяемые ресурсы одной критической секцией. Посмотрите на этот фрагмент кода:

```
int g_nNums[100];      // один разделяемый ресурс
TCHAR g_cChars[100];  // другой разделяемый ресурс
CRITICAL_SECTION g_cs; // защищает оба ресурса

DWORD WINAPI ThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    for (int x = 0; x < 100; x++) {
        g_nNums[x] = 0;
        g_cChars[x] = TEXT('X');
    }
    LeaveCriticalSection(&g_cs);
    return(0);
}
```

Здесь создана единственная критическая секция, защищающая оба массива — *g_nNums* и *g_cChars* — в период их инициализации. Но эти массивы совершенно различны. И при выполнении данного цикла ни один из потоков не получит доступ ни к одному массиву. Теперь посмотрим, что будет, если *ThreadFunc* реализовать так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_cs);
    for (int x = 0; x < 100; x++)
        g_nNums[x] = 0;
    for (x = 0; x < 100; x++)
        g_cChars[x] = TEXT('X');
    LeaveCriticalSection(&g_cs);
    return(0);
}
```

В этом фрагменте массивы инициализируются по отдельности, и теоретически после инициализации *g_nNums* посторонний поток, которому нужен доступ только к первому массиву, сможет начать исполнение — пока *ThreadFunc* занимается вторым массивом. Увы, это невозможно: обе структуры данных защищены одной критической секцией. Чтобы выйти из затруднения, создадим две критические секции:

```
int g_nNum[100];           // разделяемый ресурс
CRITICAL_SECTION g_csNums; // защищает g_nNums
TCHAR g_cChars[100];       // другой разделяемый ресурс
CRITICAL_SECTION g_csChars; // защищает g_cChars

DWORD WINAPI ThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_csNums);
    for (int x = 0; x < 100; x++)
        g_nNums[x] = 0;
    LeaveCriticalSection(&g_csNums);
    EnterCriticalSection(&g_csChars);
    for (x = 0; x < 100; x++)
        g_cChars[x] = TEXT('X');
    LeaveCriticalSection(&g_csChars);
    return(0);
}
```

Теперь другой поток сможет работать с массивом *g_nNums*, как только *ThreadFunc* закончит его инициализацию. Можно сделать и так, чтобы один поток инициализировал массив *g_nNums*, а другой — *g_cChars*.

Одновременный доступ к нескольким ресурсам

Иногда нужен одновременный доступ сразу к двум структурам данных. Тогда *ThreadFunc* следует реализовать так:

```
DWORD WINAPI ThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_csNums);
    EnterCriticalSection(&g_csChars);
    // в этом цикле нужен одновременный доступ к обоим ресурсам
    for (int x = 0; x < 100; x++) g_nNums[x] = g_cChars[x];
    LeaveCriticalSection(&g_csChars);
    LeaveCriticalSection(&g_csNums);
    return(0);
}
```

Предположим, доступ к обоим массивам требуется и другому потоку в данном процессе; при этом его функция написана следующим образом:

```
DWORD WINAPI OtherThreadFunc(PVOID pvParam) {
    EnterCriticalSection(&g_csChars);
    EnterCriticalSection(&g_csNums);
    for (int x = 0; x < 100; x++) g_nNums[x] = g_cChars[x];
    LeaveCriticalSection(&g_csNums);
    LeaveCriticalSection(&g_csChars);
    return(0);
}
```

Я лишь поменял порядок вызовов *EnterCriticalSection* и *LeaveCriticalSection*. Но из-за того, что функции *ThreadFunc* и *OtherThreadFunc* написаны именно так, существу-

ет вероятность *взаимной блокировки* (deadlock). Допустим, *ThreadFunc* начинает исполнение и занимает критическую секцию *g_csNums*. Получив от системы процессорное время, поток с функцией *OtherThreadFunc* захватывает критическую секцию *g_csChars*. 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);
    // посылаем в окно сообщение
    SendMessage(hwndSomeWnd, WM_SOMEMSG, &g_s, 0);
    LeaveCriticalSection(&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:

```
SOMESTRUCT 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.

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 1000 процессорных тактов на платформе x86. Прибавьте сюда еще и время, которое необходимо на выполнение кода этих функций в режиме ядра.

К этому моменту я уже рассказывал 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 Microsoft 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 Microsoft 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.

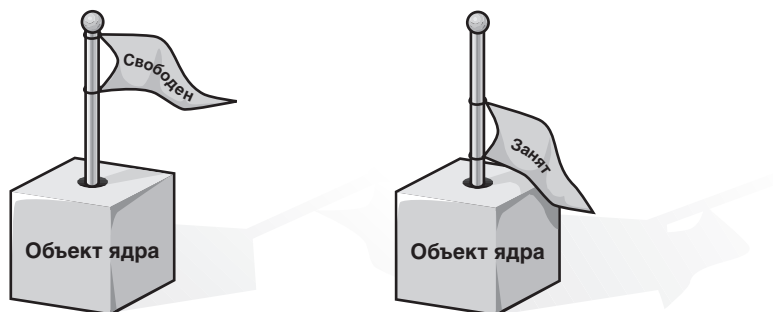
Следующие объекты ядра бывают в свободном или занятом состоянии:

- | | |
|-------------------|-----------------------------------|
| ■ процессы | ■ уведомления об изменении файлов |
| ■ потоки | ■ события |
| ■ задания | ■ ожидаемые таймеры |
| ■ файлы | ■ семафоры |
| ■ консольный ввод | ■ мьютексы |

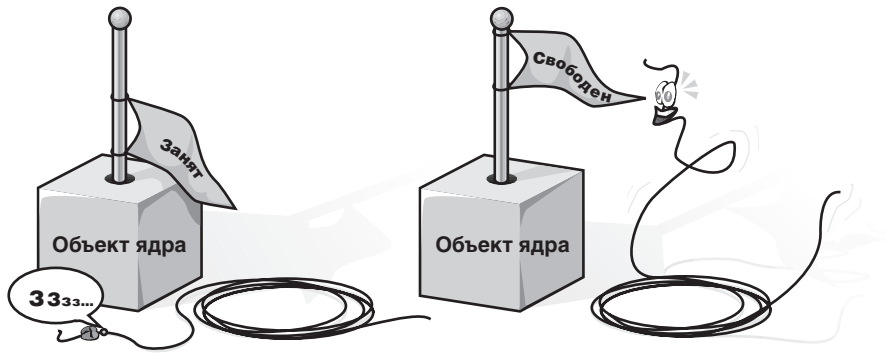
Потоки могут засыпать и в таком состоянии ждать освобождения какого-либо объекта. Правила, по которым объект переходит в свободное или занятое состояние, зависят от типа этого объекта. О правилах для объектов процессов и потоков я упоминал совсем недавно, а правила для заданий были описаны в главе 5.

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. 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);
```

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 тайм-аута, отличным от *INFINITE*:

```
DWORD dw = WaitForSingleObject(hProcess, 5000);
switch (dw) {

    case WAIT_OBJECT_0:
        // процесс завершается
        break;

    case WAIT_TIMEOUT:
        // процесс не завершился в течение 5000 мс
        break;
```

см. след. стр.

```
case WAIT_FAILED:
    // неправильный вызов функции (неверный описатель?)
    break;
}
```

Данный код сообщает системе, что вызывающий поток не 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 закончит 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,
    BOOL fWaitAll,
    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`, вычитите 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_OBJECT_0 + 0:
        // завершился процесс, идентифицируемый h[0], т. е. описателем (hProcess1)
        break;

    case WAIT_OBJECT_0 + 1:
        // завершился процесс, идентифицируемый h[1], т. е. описателем (hProcess2)
        break;

    case WAIT_OBJECT_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, относительно 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 Microsoft 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
h[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 занятое 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 Microsoft отвечает на этот вопрос так: «Алгоритм действует честно.» Что это за алгоритм, Microsoft 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 Microsoft 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 fManualReset,
    BOOL fInitialState,
    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 *fManualReset* (булева переменная) сообщает системе, хотите Вы создать событие со сбросом вручную (TRUE) или с автосбросом (FALSE). Параметр *fInitialState* определяет начальное состояние события — свободное (TRUE) или занятое (FALSE). После того как система создает объект-событие, *CreateEvent* возвращает описатель события, специфичный для конкретного процесса. Потоки из других процессов могут получить доступ к этому объекту: 1) вызовом *CreateEvent* с тем же параметром *pszName*; 2) наследованием описателя; 3) применением функции *DuplicateHandle*; и 4) вызовом *OpenEvent* с передачей в параметре *pszName* имени, совпадающего с указанным в аналогичном параметре функции *CreateEvent*. Вот что представляет собой функция *OpenEvent*.

```
HANDLE OpenEvent(
    DWORD fdwAccess,
    BOOL fInherit,
    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:

```
BOOL SetEvent(HANDLE hEvent);
```

А чтобы поменять его на занятое:

```
BOOL ResetEvent(HANDLE hEvent);
```

Вот так все просто.

Для событий с автосбросом действует следующее правило. Когда его ожидание потоком успешно завершается, этот объект автоматически сбрасывается 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 WinMain(...) {

    // 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(PVOID pvParam) {
    // ждем, когда в память будут загружены данные из файла
    WaitForSingleObject(g_hEvent, INFINITE);
    // обращаемся к блоку памяти
    :
    return(0);
}

DWORD WINAPI SpellCheck(PVOID 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. Обратите внимание, что они получают доступ к памяти в режиме только для чтения. Это единственная причина, по которой все три потока могут выполняться одновременно.

Если событие со сбросом вручную заменить на событие 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 монополярный доступ к блоку памяти, где хранятся данные, считанные из файла. Давайте перепишем функции потоков так, чтобы перед самым возвратом управления они (подобно функции *WinMain*) 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
```

см. след. стр.


```

    WaitForSingleObject(g_hEvent, INFINITE);

    // обращаемся к блоку памяти
    :
    SetEvent(g_hEvent);
    return(0);
}

DWORD WINAPI GrammarCheck(PVOID pvParam) {
    // ждем, когда в память будут загружены данные из файла
    WaitForSingleObject(g_hEvent, INFINITE);

    // обращаемся к блоку памяти
    :
    SetEvent(g_hEvent);
    return(0);
}

```

Закончив свою работу с данными, поток вызывает *SetEvent*, которая разрешает 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);

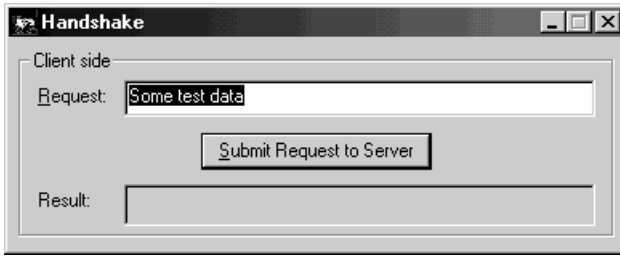
```

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» (см. листинг на рис. 9-1), демонстрирует применение событий с автосбросом. Файлы исходного кода и ресурсов этой программы находятся в каталоге 09-Handshake на компакт-диске, прилагаемом к книге. 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.

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

```

/*****
Модуль: Handshake.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h" /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <process.h> // для доступа к beginthreadex
#include "Resource.h"

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

// это событие освобождается, когда клиент передает запрос серверу
HANDLE g_hevtRequestSubmitted;

// это событие освобождается, когда сервер готов сообщить результат клиенту
HANDLE 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");

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

// это код, выполняемый серверным потоком
DWORD WINAPI ServerThread(PVOID pvParam) {

    // предполагаем, что серверный поток будет выполняться вечно
    BOOL fShutdown = FALSE;

    while (!fShutdown) {

```

Рис. 9-1. Программа-пример Handshake

Рис. 9-1. *продолжение*

```
// ждем от клиента передачи запроса
WaitForSingleObject(g_hevtRequestSubmitted, INFINITE);

// проверяем, не хочет ли клиент, чтобы сервер завершился
fShutdown =
    (lstrcmpi(g_szSharedRequestAndResultBuffer, g_szServerShutdown) == 0);

if (!fShutdown) {
    // обрабатываем клиентский запрос (инвертируем строку)
    _tcsrev(g_szSharedRequestAndResultBuffer);
}

// разрешаем клиенту обработать результат запроса
SetEvent(g_hevtResultReturned);
}

// клиент хочет нас завершить – выходим
return(0);
}

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

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

    chSETDLGICONS(hwnd, IDI_HANDSHAKE);

    // инициализируем поле ввода текстом запроса по умолчанию
    Edit_SetText(GetDlgItem(hwnd, IDC_REQUEST), TEXT("Some test data"));

    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,
                chDIMOF(g_szSharedRequestAndResultBuffer));

            // даем знать серверному потоку, что в буфере появился запрос
            SetEvent(g_hevtRequestSubmitted);
    }
}
```

см. след. стр.

Рис. 9-1. *продолжение*

```

        // 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 hinstExe, HINSTANCE, PTSTR pszCmdLine, 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(hinstExe, MAKEINTRESOURCE(IDD_HANDSHAKE), NULL, Dlg_Proc);

    // пользовательский интерфейс клиента закрывается – надо завершить серверный поток
    lstrcpy(g_szSharedRequestAndResultBuffer, 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);
}

```

Рис. 9-1. продолжение

```

CloseHandle(g_hevtRequestSubmitted);
CloseHandle(g_hevtResultReturned);

// клиентский поток завершается вместе со всем процессом
return(0);
}

/////////////////////////////////////// Конец файла /////////////////////////////////////////

```

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 fManualReset,
    PCTSTR pszName);

```

О параметрах *psa* и *pszName* я уже рассказывал в главе 3. Разумеется, любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «ожидаемый таймер», вызвав *OpenWaitableTimer*:

```

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

```

По аналогии с событиями параметр *fManualReset* 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 fResume);

```

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 2002 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 2002 года в 1:00 PM
// по местному времени
st.wYear      = 2002;    // 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
st.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 сра-

ботать 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.



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 1:00 PM 1 января 2002 года), рассмотрим параметр *lPeriod* функции *SetWaitableTimer*. Этот параметр определяет последующую частоту срабатывания таймера (в мс). Чтобы установить 6 часов, я передаю значение, равное 21 600 000 мс (т. е. 6 часов • 60 минут • 60 секунд • 1000 миллисекунд).

О последних трех параметрах функции *SetWaitableTimer* мы поговорим ближе к концу этого раздела, а сейчас продолжим обсуждение второго и третьего параметров. Вместо того чтобы устанавливать время первого срабатывания таймера в абсолютных единицах, Вы можете задать его в относительных единицах (в интервалах по 100 нс); при этом число должно быть отрицательным. (Одна секунда равна десяти миллионам 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 (он срабатывает сначала через 5 секунд,
// а потом через каждые 6 часов)
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 получают процессорное время, пока компьютер не выйдет из режима сна.

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 *WaitForSingleObject* 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 *pfCompletionRoutine* 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.

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 GetTimeFormat
    FileTimeToSystemTime(&ftLocal, &st);

    // формируем строку с датой и временем, в которое
    // сработал таймер
    GetDateFormat(LOCALE_USER_DEFAULT, DATE_LONGDATE,
        &st, NULL, szBuf, sizeof(szBuf) / sizeof(TCHAR));
    _tcsat(szBuf, _TEXT(" "));
    GetTimeFormat(LOCALE_USER_DEFAULT, 0, &st, NULL, _tcschr(szBuf, 0),
        sizeof(szBuf) / sizeof(TCHAR) - _tcslen(szBuf));

    // показываем время пользователю
    MessageBox(NULL, szBuf, "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);
```

см. след. стр.

```
// 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 дожидаться 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. Приглядитесь 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 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. Как поясняется в главе 26, 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.

Объект ядра «семафор» создается вызовом *CreateSemaphore*:

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

О параметрах *psa* и *pszName* я рассказывал в главе 3. Разумеется, любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «семафор», вызвав *OpenSemaphore*:

```
HANDLE OpenSemaphore(
    DWORD fdwAccess,
    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 (на данный момент). При инициализации моего серверного процесса клиентских запросов нет, поэтому я вызываю *CreateSemaphore* так:

```
HANDLE hSem = 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 hSem,
```

```
LONG lReleaseCount,
PLONG plPreviousCount);
```

Она просто складывает величину *lReleaseCount* со значением счетчика текущего числа ресурсов. Обычно в параметре *lReleaseCount* передают 1, но 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. Однако, если последние являются объектами пользовательского режима, то мьютексы — объектами ядра. Кроме того, единственный объект-мьютекс позволяет синхронизировать доступ к ресурсу нескольких потоков из разных 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 не 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,
    BOOL fInitialOwner,
    PCTSTR pszName);
```


О параметрах *psa* и *pszName* я рассказывал в главе 3. Разумеется, любой процесс может получить свой («процессо-зависимый») описатель существующего объекта «мьютекс», вызвав *OpenMutex*:

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

Параметр *fInitialOwner* определяет начальное состояние мьютекса. Если в нем передается FALSE (что обычно и бывает), объект-мьютекс не принадлежит 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.

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. Если какой-то посторонний поток попытается освободить мьютекс вызовом функции *ReleaseMutex*, то она, проверив идентификаторы потоков и обнаружив их несовпадение, ничего делать не станет, а 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). Если этот мьютекс ждут другие потоки, система, как обычно, «по-честному» выбирает один из потоков и позволяет ему захватить мьютекс. 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 заверченный поток — бывший владелец объекта-мьютекса, увы, невозможно.

В реальности программы никогда специально не 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*.)

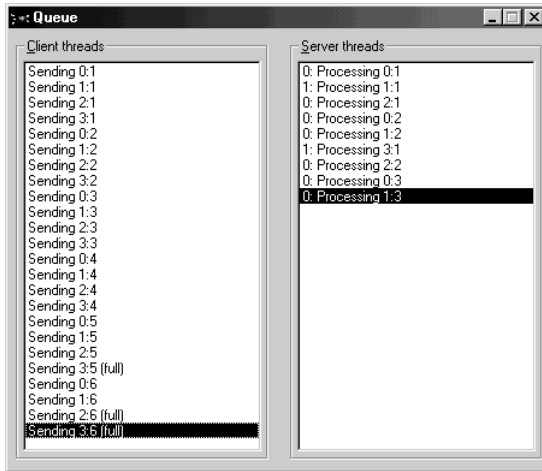
Мьютексы и критические секции

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.

| Характеристики | Объект-мьютекс | Объект — критическая секция |
|---|--|--|
| Быстродействие | Малое | Высокое |
| Возможность использования за границами процесса | Да | Нет |
| Объявление | <i>HANDLE bmtx;</i> | <i>CRITICAL_SECTION cs;</i> |
| Инициализация | <i>bmtx = CreateMutex(NULL, FALSE, NULL);</i> | <i>InitializeCriticalSection(&cs);</i> |
| Очистка | <i>CloseHandle(bmtx);</i> | <i>DeleteCriticalSection(&cs);</i> |
| Бесконечное ожидание | <i>WaitForSingleObject(bmtx, INFINITE);</i> | <i>EnterCriticalSection(&cs);</i> |
| Ожидание в течение 0 мс | <i>WaitForSingleObject(bmtx, 0);</i> | <i>TryEnterCriticalSection(&cs);</i> |
| Ожидание в течение произвольного периода времени | <i>WaitForSingleObject(bmtx, dwMilliseconds);</i> | Невозможно |
| Освобождение | <i>ReleaseMutex(bmtx);</i> | <i>LeaveCriticalSection(&cs);</i> |
| Возможность параллельного ожидания других объектов ядра | Да (с помощью <i>WaitForMultipleObjects</i> или аналогичной функции) | Нет |

Программа-пример Queue

Эта программа, «09 Queue.exe» (см. листинг на рис. 9-2), 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.



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.

О'кэй, что делает программа, Вы поняли; теперь посмотрим — как она это делает (что гораздо интереснее). Очередь 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 монополярный 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 монополярный 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` проходит успешно, в очереди еще есть место, и в нее можно поместить новый элемент. К счастью, `ReleaseSemaphore` возвращает в переменную `lPreviousCount` предыдущее количество элементов в очереди. Благодаря этому Вы точно знаете, в какой элемент массива следует записать новый эле-

мент 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 *WaitForMultipleObjects*, 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

```

/*****
Модуль: Queue.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>      // для доступа к _beginthreadex
#include "Resource.h"

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

class CQueue {
public:
    struct ELEMENT {
        int m_nThreadNum, m_nRequestNum;
        // другие элементы данных должны быть определены здесь
    };
};

```

Рис. 9-2. Программа-пример *Queue*

Рис. 9-2. продолжение

```

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);
};

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

CQueue::CQueue(int nMaxElements)
    : m_hmtxQ(m_h[0]), m_hsemNumElements(m_h[1]) {

    m_pElements = (PELEMENT)
        HeapAlloc(GetProcessHeap(), 0, sizeof(ELEMENT) * nMaxElements);
    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(m_hmtxQ, dwTimeout);

    if (dw == WAIT_OBJECT_0) {
        // этот поток получил монопольный доступ к очереди

        // увеличиваем число элементов в очереди
        LONG lPrevCount;
        fOk = ReleaseSemaphore(m_hsemNumElements, 1, &lPrevCount);
        if (fOk) {

```

см. след. стр.

Рис. 9-2. *продолжение*

```

        // в очереди еще есть место; добавляем новый элемент
        m_pElements[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 информацию
}

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

BOOL CQueue::Remove(PELEMENT pElement, DWORD dwTimeout) {

    // ждем монопольного доступа к очереди
    // и появления в ней хотя бы одного элемента
    BOOL fOk = (WaitForMultipleObjects(chDIMOF(m_h), m_h, TRUE, dwTimeout)
        == WAIT_OBJECT_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 информацию
}

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

```

Рис. 9-2. продолжение

```
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_hThreads[MAXIMUM_WAIT_OBJECTS];
int g_nNumThreads = 0;

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

DWORD WINAPI ClientThread(PVOID pvParam) {

    int nThreadNum = PtrToUlong(pvParam);
    HWND hwndLB = GetDlgItem(g_hwnd, IDC_CLIENTS);

    for (int nRequestNum = 1; !g_fShutdown; 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
            wsprintf(sz, TEXT("Sending %d:%d"), nThreadNum, nRequestNum);
        } else {

            // поставить элемент в очередь не удалось
            wsprintf(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 (!g_fShutdown) {
```

см. след. стр.

Рис. 9-2. *продолжение*

```

TCHAR sz[1024];
CQueue::ELEMENT e;

// пытаемся получить элемент из очереди
if (g_q.Remove(&e, 5000)) {

    // сообщаем, какой поток обрабатывает этот элемент,
    // какой поток поместил его в очередь и какой он по счету
    wprintf(sz, TEXT("%d: 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
    wprintf(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, ClientThread, (PVOID) (INT_PTR) x,
                0, &dwThreadID);

    // создаем серверные потоки
    for (x = 0; x < 2; x++)
        g_hThreads[g_nNumThreads++] =
            chBEGINTHREADEX(NULL, 0, ServerThread, (PVOID) (INT_PTR) x,
                0, &dwThreadID);

    return(TRUE);
}

```

Рис. 9-2. продолжение

```

////////////////////////////////////
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) {

    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_QUEUE), NULL, Dlg_Proc);
    InterlockedExchangePointer((PVOID*) &g_fShutdown, (PVOID) TRUE);
    // ждем завершения всех потоков, а затем проводим очистку
    WaitForMultipleObjects(g_nNumThreads, g_hThreads, TRUE, INFINITE);
    while (g_nNumThreads--)
        CloseHandle(g_hThreads[g_nNumThreads]);

    return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

Сводная таблица объектов, используемых для синхронизации потоков

В следующей таблице суммируются сведения о различных объектах ядра применительно к синхронизации потоков.

| Объект | Находится в занятом состоянии, когда: | Переходит в свободное состояние, когда: | Побочный эффект успешного ожидания |
|---------|---------------------------------------|--|------------------------------------|
| Процесс | процесс еще активен | процесс завершается (<i>ExitProcess</i> , <i>TerminateProcess</i>) | Нет |
| Поток | поток еще активен | поток завершается (<i>ExitThread</i> , <i>TerminateThread</i>) | Нет |

см. след. стр.

| Объект | Находится в занятом состоянии, когда: | Переходит в свободное состояние, когда: | Побочный эффект успешного ожидания |
|---|---|--|------------------------------------|
| 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 <i>ResetEvent</i> , <i>PulseEvent</i> или ожидание успешно завершилось | вызывается <i>SetEvent</i> или <i>PulseEvent</i> | Сбрасывается в исходное состояние |
| Событие со сбросом вручную | вызывается <i>ResetEvent</i> или <i>PulseEvent</i> | вызывается <i>SetEvent</i> или <i>PulseEvent</i> | Нет |
| Ожидаемый таймер с автосбросом | вызывается <i>CancelWaitableTimer</i> или ожидание успешно завершилось | наступает время срабатывания (<i>SetWaitableTimer</i>) | Сбрасывается в исходное состояние |
| Ожидаемый таймер со сбросом вручную | вызывается <i>CancelWaitableTimer</i> | наступает время срабатывания (<i>SetWaitableTimer</i>) | Нет |
| Семафор | ожидание успешно завершилось | счетчик > 0 (<i>ReleaseSemaphore</i>) | Счетчик уменьшается на 1 |
| Мьютекс | ожидание успешно завершилось | поток освобождает мьютекс (<i>ReleaseMutex</i>) | Передается потоку во владение |
| Критическая секция (пользовательского режима) | ожидание успешно завершилось (<i>(Try)EnterCriticalSection</i>) | поток освобождает критическую секцию (<i>LeaveCriticalSection</i>) | Передается потоку во владение |

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 *WaitForSingleObject* 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 сообщить системе 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 — дожидаться, когда тот прекратит обработку любого ввода. Поэтому после вызова *CreateProcess* поток родительского процесса должен вызвать *WaitForInputIdle*.

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 fWaitAll,
    DWORD dwMilliseconds,
    DWORD dwWakeMask);

DWORD MsgWaitForMultipleObjectsEx(
    DWORD 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. Подробнее эти функции рассматриваются в главе 26.

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 поток 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.

Функция *SignalObjectAndWait*

SignalObjectAndWait 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 fAlertable);
```

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 *fAlertable* 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 1000 процессорных тактов (на платформах x86), и поэтому для выполнения, например, такого кода:

```
ReleaseMutex(hMutex);
WaitForSingleObject(hEvent, INFINITE);
```

понадобится около 2000 тактов. В высокопроизводительных серверных приложениях *SignalObjectAndWait* дает заметную экономию процессорного 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 фиксирует этот импульс (pulse). Я встречал программистов, которые пишут вот такой код:

```
// выполняем какие-то операции
:
SetEvent(hEventWorkerThreadDone);
WaitForSingleObject(hEventMoreWorkToBeDone, 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 ито-

ге рабочий поток не сможет своевременно освободить событие *hEventMoreWorkToBeDone*.

Но если Вы перепишите код рабочего потока с использованием функции *SignalObjectAndWait*:

```
// выполняем какие-то операции
:  
SignalObjectAndWait(hEventWorkerThreadDone,  
    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.

WINDOWS 98 В Windows 98 функция *SignalObjectAndWait* определена, но не реализована.

Полезные средства для синхронизации потоков

За годы своей практики я часто сталкивался с проблемами синхронизации потоков и поэтому написал ряд C++-классов и компонентов, которыми я поделюсь с Вами в этой главе. Надеюсь, этот код Вам пригодится и сэкономит массу времени при разработке приложений — или по крайней мере чему-нибудь научит.

Я начну главу с того, что расскажу о реализации критической секции и расширении ее функциональности. В частности, Вы узнаете, как пользоваться одной критической секцией в нескольких процессах. Далее Вы увидите, как сделать объекты безопасными для применения в многопоточной среде, создав для собственных типов данных оболочку из C++-класса. Используя такие классы, я попутно представлю объект, ведущий себя прямо противоположно семафору.

Потом мы рассмотрим одну из типичных задач программирования: что делать, когда считывает какой-то ресурс несколько потоков, а записывает в него — только один. В Windows нет подходящего на этот случай синхронизирующего объекта, и я написал специальный C++-класс.

Наконец, я продемонстрирую свою функцию *WaitForMultipleExpressions*. Работая по аналогии с *WaitForMultipleObjects*, заставляющей ждать освобождения одного или всех объектов, она позволяет указывать более сложные условия пробуждения потока.

Реализация критической секции: объект-оптекс

Критические секции всегда интересовали меня. В конце концов, если это всего лишь объекты пользовательского режима, то почему бы мне не реализовать их самому? Разве нельзя заставить их работать без поддержки операционной системы? Кроме того, написав собственную критическую секцию, я мог бы расширить ее функциональность и в чем-то даже усовершенствовать. По крайней мере я сделал бы так, чтобы она отслеживала, какой поток захватывает защищаемый ею ресурс. Такая реализация критической секции помогла бы мне устранять проблемы с взаимной блокировкой потоков: с помощью отладчика я узнавал бы, какой из них не освободил тот или иной ресурс.

Так что давайте без лишних разговоров перейдем к тому, как реализуются критические секции. Я все время утверждаю, что они являются объектами пользовательского режима. На самом деле это не совсем так. Любой поток, который пытается войти в критическую секцию, уже захваченную другим потоком, переводится в состояние ожидания. А для этого он должен перейти из пользовательского режима в режим ядра. Поток пользовательского режима может остановиться, просто войдя в цикл ожидания, но это вряд ли можно назвать эффективной реализацией ждущего режима, и поэтому Вы должны всячески избегать ее.

Значит, в критических секциях есть какой-то объект ядра, умеющий переводить поток в эффективный ждущий режим. Критическая секция обладает высоким быстродействием, потому что этот объект ядра используется только при конкуренции потоков за вход в критическую секцию. И он не задействован, пока потоку удастся немедленно захватывать защищаемый ресурс, работать с ним и освобождать его без конкуренции со стороны других потоков, так как выходить из пользовательского режима потоку в этом случае не требуется. В большинстве приложений конкуренция двух (или более) потоков за одновременный вход в критическую секцию наблюдается нечасто.

Мой вариант критической секции содержится в файлах `Optex.h` и `Optex.cpp` (см. листинг на рис. 10-1). Я назвал ее *оптимизированным мьютексом* — *оптексом* и реализовал в виде C++-класса. Разобравшись в этом коде, Вы поймете, почему критические секции работают быстрее объектов ядра «мьютекс».

Поскольку я создавал собственную критическую секцию, у меня была возможность расширить ее функциональность. Например, мой класс `COptex` позволяет синхронизировать потоки из разных процессов. Это фантастически полезная особенность моей реализации: Вы получаете высокоэффективный механизм взаимодействия между потоками из разных процессов.

Чтобы использовать мой оптекс, Вы просто объявляете объект класса `COptex`. Для этого объекта предусмотрено три конструктора:

```
COptex::(DWORD dwSpinCount = 4000);
COptex::(PCSTR pszName, DWORD dwSpinCount = 4000);
COptex::(PCWSTR pszName, DWORD dwSpinCount = 4000);
```

Первый создает объект `COptex`, применимый для синхронизации потоков лишь одного процесса. Оптекс этого типа работает быстрее, чем межпроцессный. Остальные два конструктора создают оптекс, которым могут пользоваться потоки из разных процессов. В параметре *pszName* Вы должны передавать ANSI- или Unicode-строку, уникально идентифицирующую каждый разделяемый оптекс. Чтобы процессы разделяли один оптекс, они должны создать по экземпляру объекта `COptex` с одинаковым именем.

Поток входит в объект `COptex` и покидает его, вызывая методы *Enter* и *Leave*:

```
void COptex::Enter();
void COptex::Leave();
```

Я даже включил методы, эквивалентные функциям *TryEnterCriticalSection* и *SetCriticalSectionSpinCount* критических секций:

```
BOOL COptex::TryEnter();
void COptex::SetSpinCount(DWORD dwSpinCount);
```

Тип оптекса (одно- или межпроцессный) позволяет выяснить последний метод класса `COptex`, показанный ниже. (Необходимость в его вызове возникает очень редко, но внутренние функции класса время от времени к нему обращаются.)

```
BOOL COptex::IsSingleProcessOptex() const;
```

Вот и все (открытые) функции, о которых Вам нужно знать, чтобы пользоваться оптексом. Теперь я объясню, как работает оптекс. Он — как, в сущности, и критическая секция — содержит несколько переменных-членов. Значения этих переменных отражают состояние оптекса. Просмотрев файл `Optex.h`, Вы увидите, что в основном они являются элементами структуры `SHAREDINFO`, а остальные — членами самого класса. Назначение каждой переменной описывается в следующей таблице.

| Переменная | Описание |
|------------------------|--|
| <i>m_lLockCount</i> | Сообщает, сколько раз потоки пытались занять оптекс. Ее значение равно 0, если оптекс не занят ни одним потоком. |
| <i>m_dwTbreadId</i> | Сообщает уникальный идентификатор потока — владельца оптекса. Ее значение равно 0, если оптекс не занят ни одним потоком. |
| <i>m_lRecurseCount</i> | Указывает, сколько раз оптекс был занят потоком-владельцем. Ее значение равно 0, если оптекс не занят ни одним потоком. |
| <i>m_bevt</i> | Содержит описатель объекта ядра «событие», используемого, только если поток пытается войти в оптекс в то время, как им владеет другой поток. Описатели объектов ядра специфичны для конкретных процессов, и именно поэтому данная переменная не включена в структуру SHAREDINFO. |
| <i>m_dwSpinCount</i> | Определяет, сколько попыток входа в оптекс должен предпринять поток до перехода в состояние ожидания на объекте ядра «событие». На однопроцессорной машине значение этой переменной всегда равно 0. |
| <i>m_hfm</i> | Содержит описатель объекта ядра «проекция файла», используемого при разделении оптекса несколькими процессами. Описатели объектов ядра специфичны для конкретных процессов, и именно поэтому данная переменная не включена в структуру SHAREDINFO. В однопроцессном оптексе значение этой переменной всегда равно NULL. |
| <i>m_psi</i> | Содержит указатель на элементы данных оптекса, которые могут использоваться несколькими процессами. Адреса памяти специфичны для конкретных процессов, и именно поэтому данная переменная не включена в структуру SHAREDINFO. В однопроцессном оптексе эта переменная указывает на блок памяти, выделенный из кучи, а в межпроцессном — на файл, спроецированный в память. |

Комментариев в исходном коде вполне достаточно, и у Вас не должно возникнуть трудностей в понимании того, как работает оптекс. Важно лишь отметить, что высокое быстроедействие оптекса достигается за счет интенсивного использования *Interlocked*-функций. Благодаря им код выполняется в пользовательском режиме и переходит в режим ядра только в том случае, когда это действительно необходимо.

Программа-пример Optex

Эта программа, «10 Optex.exe» (см. листинг на рис. 10-1), предназначена для проверки того, что класс COptex работает корректно. Файлы исходного кода и ресурсов этой программы находятся в каталоге 10-Optex на компакт-диске, прилагаемом к книге. Я всегда запускаю такие приложения под управлением отладчика, чтобы наблюдать за всеми функциями и переменными — членами классов.

При запуске программа сначала определяет, является ли она первым экземпляром. Для этого я создаю именованный объект ядра «событие». Реально я им не пользуюсь, а просто смотрю, вернет ли *GetLastError* значение ERROR_ALREADY_EXISTS. Если да, значит, это второй экземпляр программы. Зачем мне два экземпляра этой программы, я объясню позже.

Если же это первый экземпляр, я создаю однопроцессный объект COptex и вызываю свою функцию *FirstFunc*. Она выполняет серию операций с объектом-оптексом и создает второй поток, который манипулирует тем же оптексом. На этом этапе с оптексом работают два потока из одного процесса. Что именно они делают, Вы узнаете, просмотрев исходный код. Я пытался охватить все мыслимые сценарии, чтобы дать шанс на выполнение каждому блоку кода в классе COptex.

После тестирования однопроцессного оптекса я начинаю проверку межпроцессного оптекса. В функции *_tWinMain* по завершении первого вызова *FirstFunc* я создаю

другой объект-оптекс COptex. Но на этот раз я присваиваю ему имя — *CrossOptexTest*. Простое присвоение оптексу имени в момент создания превращает этот объект в межпроцессный. Далее я снова вызываю *FirstFunc*, передавая ей адрес межпроцессного оптекса. При этом *FirstFunc* выполняет в основном тот же код, что и раньше. Но теперь она порождает не второй поток, а дочерний процесс.

Этот дочерний процесс представляет собой всего лишь второй экземпляр той же программы. Однако, создав при запуске объект ядра «событие», она обнаруживает, что такой объект уже существует. Тем самым она узнает, что является вторым экземпляром, и выполняет другой код (отличный от того, который выполняется первым экземпляром). Первое, что делает второй экземпляр, — вызывает *DebugBreak*:

```
VOID DebugBreak();
```

Эта удобная функция инициирует запуск отладчика и его подключение к данному процессу. Это здорово упрощает мне отладку обоих экземпляров данной программы. Далее второй экземпляр создает межпроцессный оптекс, передавая конструктору строку с тем же именем. Поскольку имена идентичны, оптекс становится разделяемым между обоими процессами. Кстати, один оптекс могут разделять более двух процессов.

Наконец, второй экземпляр программы вызывает функцию *SecondFunc*, передавая ей адрес межпроцессного оптекса, и с этого момента выполняется тот же набор тестов. Единственное, что в них меняется, — два потока, манипулирующие оптексом, принадлежат разным процессам.



Optex.cpp

```

/*****
Модуль: Optex.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include "Optex.h"

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

// 0=многопроцессорная машина, 1=однопроцессорная, -1=тип пока не определен
BOOL COptex::sm_fUniprocessorHost = -1;

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

PSTR COptex::ConstructObjectName(PSTR pszResult,
    PCSTR pszPrefix, BOOL fUnicode, PVOID pszName) {

    pszResult[0] = 0;
    if (pszName == NULL)
        return(NULL);

    wsprintfA(pszResult, fUnicode ? "%s%S" : "%s%s", pszPrefix, pszName);
    return(pszResult);
}

```

Рис. 10-1. Программа-пример Optex

Рис. 10-1. продолжение

```

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

void COptex::CommonConstructor(DWORD dwSpinCount,
    BOOL fUnicode, PVOID pszName) {

    if (sm_fUniprocessorHost == -1) {
        // конструируется первый объект; выясняем количество процессоров
        SYSTEM_INFO sinf;
        GetSystemInfo(&sinf);
        sm_fUniprocessorHost = (sinf.dwNumberOfProcessors == 1);
    }

    m_hevt = m_hfm = NULL;
    m_psi = NULL;

    if (pszName == NULL) { // создание однопроцессного оптекса

        m_hevt = CreateEventA(NULL, FALSE, FALSE, NULL);
        chASSERT(m_hevt != NULL);

        m_psi = new SHAREDINFO;
        chASSERT(m_psi != NULL);
        ZeroMemory(m_psi, sizeof(*m_psi));

    } else { // создание межпроцессного оптекса

        // всегда используем ANSI, чтобы программа работала в Win9x и Windows 2000
        char szResult[100];
        ConstructObjectName(szResult, "Optex_Event_", fUnicode, pszName);
        m_hevt = CreateEventA(NULL, FALSE, FALSE, szResult);
        chASSERT(m_hevt != NULL);

        ConstructObjectName(szResult, "Optex_MMF_", fUnicode, pszName);
        m_hfm = CreateFileMappingA(INVALID_HANDLE_VALUE, NULL,
            PAGE_READWRITE, 0, sizeof(*m_psi), szResult);
        chASSERT(m_hfm != NULL);

        m_psi = (PSHAREDINFO) MapViewOfFile(m_hfm,
            FILE_MAP_WRITE, 0, 0, 0);
        chASSERT(m_psi != NULL);

        // Примечание: элементы m_lLockCount, m_dwThreadId и m_lRecurseCount
        // структуры SHAREDINFO надо инициализировать нулевым значением. К счастью,
        // механизм проецирования файлов в память избавляет нас от части работы,
        // связанной с синхронизацией потоков.
    }

    SetSpinCount(dwSpinCount);
}

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

```

см. след. стр.

Рис. 10-1. *продолжение*

```

COptex::~COptex() {

#ifdef _DEBUG
    if (IsSingleProcessOptex() && (m_psi->m_dwThreadId != 0)) {
        // однопроцессный оптекс нельзя разрушать,
        // если им владеет какой-нибудь поток
        DebugBreak();
    }

    if (!IsSingleProcessOptex() &&
        (m_psi->m_dwThreadId == GetCurrentThreadId())) {

        // межпроцессный оптекс нельзя разрушать, если им владеет наш поток
        DebugBreak();
    }
#endif

    CloseHandle(m_hevt);

    if (IsSingleProcessOptex()) {
        delete m_psi;
    } else {
        UnmapViewOfFile(m_psi);
        CloseHandle(m_hfm);
    }
}

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

void COptex::SetSpinCount(DWORD dwSpinCount) {

    // спин-блокировка на однопроцессорных машинах не применяется
    if (!sm_fUniprocessorHost)
        InterlockedExchangePointer((PVOID*) &m_psi->m_dwSpinCount,
            (PVOID) (DWORD_PTR) dwSpinCount);
}

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

void COptex::Enter() {

    // "крутимся", пытаюсь захватить оптекс
    if (TryEnter())
        return; // получилось, возвращаем управление

    // захватить оптекс не удалось, переходим в состояние ожидания
    DWORD dwThreadId = GetCurrentThreadId();

    if (InterlockedIncrement(&m_psi->m_lLockCount) == 1) {
        // оптекс не занят, пусть этот поток захватит его разок
        m_psi->m_dwThreadId = dwThreadId;
    }
}

```

Рис. 10-1. *продолжение*

```

        m_psi->m_lRecurseCount = 1;

    } else {

        if (m_psi->m_dwThreadId == dwThreadId) {

            // если оптекс принадлежит данному потоку, захватываем его еще раз
            m_psi->m_lRecurseCount++;

        } else {

            // оптекс принадлежит другому потоку, ждем
            WaitForSingleObject(m_hevt, INFINITE);

            // оптекс не занят, пусть этот поток захватит его разок
            m_psi->m_dwThreadId = dwThreadId;
            m_psi->m_lRecurseCount = 1;
        }
    }
}

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

BOOL COptex::TryEnter() {

    DWORD dwThreadId = GetCurrentThreadId();

    BOOL fThisThreadOwnsTheOptex = FALSE;    // считаем, что поток владеет оптексом
    DWORD dwSpinCount = m_psi->m_dwSpinCount; // задаем число циклов

    do {
        // если счетчик числа блокировок = 0, оптекс не занят,
        // и мы можем захватить его
        fThisThreadOwnsTheOptex = (0 ==
            InterlockedCompareExchange(&m_psi->m_lLockCount, 1, 0));

        if (fThisThreadOwnsTheOptex) {

            // оптекс не занят, пусть этот поток захватит его разок
            m_psi->m_dwThreadId = dwThreadId;
            m_psi->m_lRecurseCount = 1;

        } else {

            if (m_psi->m_dwThreadId == dwThreadId) {

                // если оптекс принадлежит данному потоку, захватываем его еще раз
                InterlockedIncrement(&m_psi->m_lLockCount);
                m_psi->m_lRecurseCount++;
                fThisThreadOwnsTheOptex = TRUE;
            }
        }
    } while (!fThisThreadOwnsTheOptex);
}

```

см. след. стр.

Рис. 10-1. *продолжение*

```

    }
}

} while (!fThisThreadOwnsTheOptex && (dwSpinCount-- > 0));

// возвращаем управление независимо от того,
// владеет данный поток оптексом или нет
return(fThisThreadOwnsTheOptex);
}

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

void COptex::Leave() {

#ifdef _DEBUG
    // покинуть оптекс может лишь тот поток, который им владеет
    if (m_psi->m_dwThreadId != GetCurrentThreadId())
        DebugBreak();
#endif

    // уменьшаем счетчик числа захватов оптекса данным потоком
    if (--m_psi->m_lRecurseCount > 0) {

        // оптекс все еще принадлежит нам
        InterlockedDecrement(&m_psi->m_lLockCount);

    } else {

        // оптекс нам больше не принадлежит
        m_psi->m_dwThreadId = 0;

        if (InterlockedDecrement(&m_psi->m_lLockCount) > 0) {

            // если оптекс ждут другие потоки,
            // событие с автосбросом пробудит один из них
            SetEvent(m_hevt);
        }
    }
}

//////////////////////////////////// Конеч файл //////////////////////////////////
```

Optex.h

```

/*****
Имя модуля: Optex.h
Написан: Джеффри Рихтером
*****/

#pragma once
```

Рис. 10-1. продолжение

```

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

class COptex {
public:
    COptex(DWORD dwSpinCount = 4000);
    COptex(PCSTR pszName, DWORD dwSpinCount = 4000);
    COptex(PCWSTR pszName, DWORD dwSpinCount = 4000);
    ~COptex();

    void SetSpinCount(DWORD dwSpinCount);
    void Enter();
    BOOL TryEnter();
    void Leave();
    BOOL IsSingleProcessOptex() const;

private:
    typedef struct {
        DWORD m_dwSpinCount;
        long m_lLockCount;
        DWORD m_dwThreadId;
        long m_lRecurseCount;
    } SHAREDINFO, *PSHAREDINFO;

    HANDLE m_hevt;
    HANDLE m_hfm;
    PSHAREDINFO m_psi;

private:
    static BOOL sm_fUniprocessorHost;

private:
    void CommonConstructor(DWORD dwSpinCount, BOOL fUnicode, PVOID pszName);
    PSTR ConstructObjectName(PSTR pszResult,
        PCSTR pszPrefix, BOOL fUnicode, PVOID pszName);
};

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

inline COptex::COptex(DWORD dwSpinCount) {
    CommonConstructor(dwSpinCount, FALSE, NULL);
}

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

inline COptex::COptex(PCSTR pszName, DWORD dwSpinCount) {
    CommonConstructor(dwSpinCount, FALSE, (PVOID) pszName);
}

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

```

см. след. стр.

Рис. 10-1. *продолжение*

```
inline COptex::COptex(PCWSTR pszName, DWORD dwSpinCount) {
    CommonConstructor(dwSpinCount, TRUE, (PVOID) pszName);
}

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

inline COptex::IsSingleProcessOptex() const {
    return(m_hfm == NULL);
}

//////////////////////////////////// Конец файла //////////////////////////////////////
```

OptexTest.cpp

```
/*
Имя модуля: OptexTest.cpp
Написан: Джеффри Рихтером
*/

#include "..\CmnHdr.h" /* см. приложение A */
#include <tchar.h>
#include <process.h>
#include "Optex.h"

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

DWORD WINAPI SecondFunc(PVOID pvParam) {
    COptex& optex = * (COptex*) pvParam;

    // здесь оптексом должен владеть первичный поток,
    // и выполнение этого оператора должно закончиться неудачно
    chVERIFY(optex.TryEnter() == FALSE);

    // ждем, когда первичный поток откажется от оптекса
    optex.Enter();

    optex.Enter(); // проверяем "рекурсию захватов"
    chMB("Secondary: Entered the optex\n(Dismiss me 2nd)");

    // покидаем оптекс, но он все еще принадлежит нам
    optex.Leave();
    chMB("Secondary: The primary thread should not display a box yet");
    optex.Leave(); // теперь первичный поток может работать

    return(0);
}

////////////////////////////////////
```

Рис. 10-1. продолжение

```

VOID FirstFunc(BOOL fLocal, COptex& optex) {

    optex.Enter(); // захватываем оптекс

    // поскольку этот поток уже владеет оптексом, мы сможем заполучить его еще раз
    chVERIFY(optex.TryEnter());

    HANDLE hOtherThread = NULL;
    if (fLocal) {
        // порождаем вторичный поток просто для тестирования (передаем ему оптекс)

        DWORD dwThreadId;
        hOtherThread = chBEGINTHREADEX(NULL, 0,
            SecondFunc, (PVOID) &optex, 0, &dwThreadId);

    } else {
        // порождаем вторичный процесс просто для тестирования
        STARTUPINFO si = { sizeof(si) };
        PROCESS_INFORMATION pi;
        TCHAR szPath[MAX_PATH];
        GetModuleFileName(NULL, szPath, chDIMOF(szPath));
        CreateProcess(NULL, szPath, NULL, NULL,
            FALSE, 0, NULL, NULL, &si, &pi);
        hOtherThread = pi.hProcess;
        CloseHandle(pi.hThread);
    }

    // ждем, когда второй поток захватит оптекс
    chMB("Primary: Hit OK to give optex to secondary");

    // разрешаем второму потоку захватить оптекс
    optex.Leave();
    optex.Leave();

    // ждем, когда второй поток захватит оптекс
    chMB("Primary: Hit OK to wait for the optex\n(Dismiss me 1st)");

    optex.Enter(); // пытаемся вновь захватить оптекс

    WaitForSingleObject(hOtherThread, INFINITE);
    CloseHandle(hOtherThread);
    optex.Leave();
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    // это событие используется только для того, чтобы
    // определить, какой это экземпляр – первый или второй
    HANDLE hev1 = CreateEvent(NULL, FALSE, FALSE, TEXT("OptexTest"));

```

см. след. стр.

Рис. 10-1. продолжение

```

if (GetLastError() != ERROR_ALREADY_EXISTS) {

    // это первый экземпляр тестовой программы

    // сначала тестируем однопроцессный оптекс
    COptex optexSingle; // создаем однопроцессный оптекс
    FirstFunc(TRUE, optexSingle);

    // теперь тестируем межпроцессный оптекс
    COptex optexCross("CrossOptexTest"); // создаем межпроцессный оптекс
    FirstFunc(FALSE, optexCross);

} else {

    // это второй экземпляр тестовой программы
    DebugBreak(); // принудительно подключаем отладчик для трассировки

    // тестируем межпроцессный оптекс
    COptex optexCross("CrossOptexTest"); // создаем межпроцессный оптекс
    SecondFunc((PVOID) &optexCross);
}

CloseHandle(hevt);
return(0);
}

////////////////////////////////////// Конец файла ////////////////////////////////////////

```

Создание инверсных семафоров и типов данных, безопасных в многопоточной среде

Как-то раз я писал одну программу, и мне понадобился объект ядра, который вел бы себя прямо противоположно тому, как ведет себя семафор. Мне нужно было, чтобы он переходил в свободное состояние, когда его счетчик текущего числа ресурсов обнуляется, и оставался в занятом состоянии, пока значение этого счетчика больше 0.

Я мог бы придумать много применений такому объекту. Например, поток должен пробудиться после того, как определенная операция будет выполнена 100 раз. Чтобы осуществить это, нужен объект ядра, счетчик которого можно было бы инициализировать этим значением. Пока он больше 0, объект остается в занятом состоянии. По окончании каждой операции Вы уменьшаете счетчик в объекте ядра на 1. Как только счетчик обнуляется, объект переходит в свободное состояние, сообщая другому потоку, что тот может пробудиться и чем-то заняться. Это типичная задача, и я не понимаю, почему в Windows нет подходящего синхронизирующего объекта.

В сущности, Microsoft могла бы легко решить эту задачу, предусмотрев в семафоре возможность присвоения отрицательных значений его счетчику текущего числа ресурсов. Тогда Вы инициализировали бы счетчик семафора значением -99 и по окончании каждой операции вызывали бы *ReleaseSemaphore*. Как только его счетчик достиг бы значения 1, объект перешел бы в свободное состояние. После этого мог бы пробудиться другой Ваш поток и выполнить свою работу. Увы, Microsoft запрещает

присвоение счетчику семафора отрицательных значений, и вряд ли здесь что-то переменится в обозримом будущем.

В этом разделе я познакомлю Вас со своим набором C++-классов, которые действуют как инверсный семафор и делают уйму всяких других вещей. Исходный код этих классов находится в файле `Interlocked.h` (см. листинг на рис. 10-2).

Когда я впервые взялся за решение этой проблемы, я понимал, что главное в нем — обеспечить безопасность манипуляций над переменной в многопоточной среде. Я хотел найти элегантное решение, которое позволило бы легко писать код, ссылающийся на эту переменную. Очевидно, что самый простой способ обезопасить какой-то ресурс от повреждения в многопоточной среде, — защитить его с помощью критической секции. В C++ это можно сделать без особого труда. Достаточно создать C++-класс, который содержит защищаемую переменную и структуру `CRITICAL_SECTION`. В конструкторе Вы вызываете *InitializeCriticalSection*, а в деструкторе — *DeleteCriticalSection*. Затем для любой переменной-члена Вы вызываете *EnterCriticalSection*, что-то делаете с этой переменной и вызываете *LeaveCriticalSection*. Если Вы именно так реализуете C++-класс, то писать безопасный код, обращающийся к какой-либо структуре данных, будет несложно. Этот принцип положен мной в основу всех C++-классов, о которых я буду рассказывать в данном разделе. (Конечно, вместо критических секций я мог бы использовать оптекс, рассмотренный в предыдущем разделе.)

Первый класс, `CResGuard`, охраняет доступ к ресурсу. Он содержит два элемента данных: `CRITICAL_SECTION` и `LONG`. Последний используется для слежения за тем, сколько раз поток, владеющий ресурсом, входил в критическую секцию. Эта информация полезна при отладке. Конструктор и деструктор объекта `CResGuard` вызывают соответственно *InitializeCriticalSection* и *DeleteCriticalSection*. Поскольку создать объект может лишь единственный поток, конструктор и деструктор какого-либо C++-объекта не обязательно должен быть реентерабельным. Функция-член *IsGuarded* просто сообщает, была ли хоть раз вызвана *EnterCriticalSection* для данного объекта. Как я уже говорил, все это предназначено для отладки. Включение `CRITICAL_SECTION` в C++-объект гарантирует корректность инициализации и удаления критической секции.

Класс `CResGuard` также включает открытый вложенный C++-класс `CGuard`. Объект `CGuard` содержит ссылку на объект `CResGuard` и предусматривает лишь конструктор и деструктор. Конструктор обращается к функции-члену *Guard* класса `CResGuard`, вызывающей *EnterCriticalSection*, а деструктор — к функции-члену *Unguard* того же класса, вызывающей *LeaveCriticalSection*. Такая схема упрощает манипуляции с `CRITICAL_SECTION`. Вот небольшой фрагмент кода, иллюстрирующий применение этих классов:

```
struct SomeDataStruct {
    :
} g_SomeSharedData;

// Создаем объект CResGuard, защищающий g_SomeSharedData.
// Примечание: Конструктор инициализирует критическую секцию, а деструктор удаляет ее.
CResGuard g_rgSomeSharedData;

void AFunction() {
    // эта функция работает с разделяемой структурой данных

    // защищаем ресурс от одновременного доступа со стороны нескольких потоков
    CResGuard::CGuard gDummy(g_rgSomeSharedData); // входим в критическую секцию
```

см. след. стр.

```
// работаем с ресурсом g_SomeSharedData
:
} // Примечание: LeaveCriticalSection вызывается, когда gDummy
// выходит за пределы области видимости.
```

Следующий C++-класс, `CInterlockedType`, содержит все, что нужно для создания объекта данных, безопасного в многопоточной среде. Я сделал `CInterlockedType` классом шаблона, чтобы его можно было применять для любых типов данных. Поэтому Вы можете использовать его, например, с целочисленной переменной, строкой или произвольной структурой данных.

Каждый экземпляр объекта `CInterlockedType` содержит два элемента данных. Первый — это экземпляр шаблонного типа данных, который Вы хотите сделать безопасным в многопоточной среде. Он является закрытым, и им можно манипулировать только через функции-члены класса `CInterlockedType`. Второй элемент данных представляет собой экземпляр объекта `CResGuard`, так что класс, производный от `CInterlockedType`, может легко защитить свои данные.

Предполагается, что Вы всегда будете создавать свой класс, используя класс `CInterlockedType` как базовый. Ранее я уже говорил, что класс `CInterlockedType` предоставляет все необходимое для создания объекта, безопасного в многопоточной среде, но производный класс должен сам позаботиться о корректном использовании элементов `CInterlockedType`.

Класс `CInterlockedType` содержит всего четыре открытые функции: конструктор, инициализирующий объект данных, и конструктор, не инициализирующий этот объект, а также виртуальный деструктор, который ничего не делает, и оператор приведения типа (`cast operator`). Последний просто гарантирует безопасный доступ к данным, охраняя ресурс и возвращая текущее значение объекта. (Ресурс автоматически разблокируется при выходе локальной переменной *x* за пределы ее области видимости.) Этот оператор упрощает безопасную проверку значения объекта данных, содержащегося в классе.

В классе `CInterlockedType` также присутствуют три неvirtуальные защищенные функции, которые будут вызываться производным классом. Две функции *GetVal* возвращают текущее значение объекта данных. В отладочных версиях файла обе эти функции сначала проверяют, охраняется ли объект данных. Если бы он не охранялся, *GetVal* могла бы вернуть значение объекта, а затем позволить другому потоку изменить его до того, как первый поток успеет что-то сделать с этим значением. Я предполагаю, что вызывающий поток получает значение объекта для того, чтобы как-то изменить его. Поэтому функции *GetVal* требуют от вызывающего потока охраны доступа к данным. Определив, что данные охраняются, функции *GetVal* возвращают текущее значение.

Эти функции идентичны с тем исключением, что одна из них манипулирует константной версией объекта. Благодаря этому Вы можете без проблем писать код, работающий как с константными, так и с неконстантными данными.

Третья неvirtуальная защищенная функция-член — *SetVal*. Желая модифицировать данные, любая функция-член производного класса должна защитить доступ к этим данным, а потом вызвать функцию *SetVal*. Как и *GetVal*, функция *SetVal* сначала проводит отладочную проверку, чтобы убедиться, не забыл ли код производного класса защитить доступ к данным. Затем *SetVal* проверяет, действительно ли данные изменяются. Если да, *SetVal* сохраняет старое значение, присваивает объекту новое значение и вызывает виртуальную защищенную функцию-член *OnValChanged*, передавая ей оба значения. В классе `CInterlockedType` последняя функция реализована так, что она

ничего не делает. Вы можете использовать эту функцию-член для того, чтобы расширить возможности своего производного класса, но об этом мы поговорим, когда дойдем до рассмотрения класса `CWhenZero`.

До сих пор речь шла в основном об абстрактных классах и концепциях. Теперь посмотрим, как пользоваться этой архитектурой на благо всего человечества. Я представлю Вам `CInterlockedScalar` — класс шаблона, производный от `CInterlockedType`. С его помощью Вы сможете создавать безопасные в многопоточной среде скалярные (простые) типы данных — байт, символ, 16-, 32- или 64-битное целое, вещественное значение (с плавающей точкой) и т. д. Поскольку `CInterlockedScalar` является производным от класса `CInterlockedType`, у него нет собственных элементов данных. Конструктор `CInterlockedScalar` просто обращается к конструктору `CInterlockedType`, передавая ему начальное значение объекта скалярных данных. Класс `CInterlockedScalar` работает только с числовыми значениями, и в качестве начального значения я выбрал нуль, чтобы наш объект всегда создавался в известном состоянии. Ну а деструктор класса `CInterlockedScalar` вообще ничего не делает.

Остальные функции-члены класса `CInterlockedScalar` отвечают за изменение скалярного значения. Для каждой операции над ним предусмотрена отдельная функция-член. Чтобы класс `CInterlockedScalar` мог безопасно манипулировать своим объектом данных, все функции-члены перед выполнением какой-либо операции блокируют доступ к этому объекту. Функции очень просты, и я не стану подробно объяснять их; просмотрев исходный код, Вы сами поймете, что они делают. Однако я покажу, как пользоваться этими классами. В следующем фрагменте кода объявляется безопасная в многопоточной среде переменная типа `BYTE` и над ней выполняется серия операций:

```
CInterlockedScalar<BYTE> b = 5; // безопасная переменная типа BYTE
BYTE b2 = 10;                  // небезопасная переменная типа BYTE
b2 = b++;                      // b2=5, b=6
b *= 4;                        // b=24
b2 = b;                        // b2=24, b=24
b += b;                        // b=48
b %= 2;                        // b=0
```

Работа с безопасной скалярной переменной так же проста, как и с небезопасной. Благодаря замещению (перегрузке) операторов в C++ даже код в таких случаях фактически одинаков! С помощью C++-классов, о которых я уже рассказывал, любую небезопасную переменную можно легко превратить в безопасную, внося лишь минимальные изменения в исходный код своей программы.

Проектируя все эти классы, я хотел создать объект, чье поведение было бы противоположно поведению семафора. Эту функциональность предоставляет мой C++-класс `CWhenZero`, производный от `CInterlockedScalar`. Когда скалярное значение равно 0, объект `CWhenZero` пребывает в свободном состоянии, а когда оно не равно 0 — в занятом.

Как Вам известно, C++-объекты не поддерживают такие состояния — в них могут находиться только объекты ядра. Значит, в `CWhenZero` нужны дополнительные элементы данных с описателями объектов ядра «событие». Я включил в объект `CWhenZero` два элемента данных: *m_hvtZero* (описатель объекта ядра «событие», переходящего в свободное состояние, когда объект данных содержит нулевое значение) и *m_hvtNotZero* (описатель объекта ядра «событие», переходящего в свободное состояние, когда объект данных содержит ненулевое значение).

Конструктор `CWhenZero` принимает начальное значение для объекта данных, а также позволяет указать, какими должны быть объекты ядра «событие» — со сбросом

вручную (по умолчанию) или с автосбросом. Далее конструктор, вызывая *CreateEvent*, создает два объекта ядра «событие» и переводит их в свободное или занятое состояние в зависимости от того, равно ли нулю начальное значение. Деструктор *CWhenZero* просто закрывает описатели этих двух объектов ядра. Поскольку *CWhenZero* открыто наследует от класса *CInterlockedScalar*, все функции-члены перегруженного оператора доступны и пользователям объекта *CWhenZero*.

Помните защищенную функцию-член *OnValChanged*, объявленную внутри класса *CInterlockedType*? Так вот, класс *CWhenZero* замещает эту виртуальную функцию. Она отвечает за перевод объектов ядра «событие» в свободное или занятое состояние в соответствии со значением объекта данных. *OnValChanged* вызывается при каждом изменении этого значения. Ее реализация в *CWhenZero* проверяет, равно ли нулю новое значение. Если да, функция устанавливает событие *m_bevtZero* и сбрасывает событие *m_bevtNotZero*. Нет — все делается наоборот.

Теперь, если Вы хотите, чтобы поток ждал нулевого значения объекта данных, от Вас требуется лишь следующее:

```
CWhenZero<BYTE> b = 0; // безопасная переменная типа BYTE

// немедленно возвращает управление, так как b равна 0
WaitForSingleObject(b, INFINITE);

b = 5;

// возвращает управление, только если другой поток присваивает b нулевое значение
WaitForSingleObject(b, INFINITE);
```

Вы можете вызывать *WaitForSingleObject* именно таким образом, потому что класс *CWhenZero* включает и функцию-член оператора приведения, которая приводит объект *CWhenZero* к типу *HANDLE* объекта ядра. Иначе говоря, передача C++-объекта *CWhenZero* любой Windows-функции, ожидающей *HANDLE*, приводит к автоматическому вызову функции-члена оператора приведения, возвращаемое значение которой и передается Windows-функции. В данном случае эта функция-член возвращает описатель объекта ядра «событие» *m_bevtZero*.

Описатель события *m_bevtNotZero* внутри класса *CWhenZero* позволяет писать код, ждущий ненулевого значения объекта данных. К сожалению, в класс нельзя включить второй оператор приведения *HANDLE* — для получения описателя *m_bevtNotZero*. Поэтому мне пришлось добавить функцию-член *GetNotZeroHandle*, которая используется так:

```
CWhenZero<BYTE> b = 5; // безопасная переменная типа BYTE

// немедленно возвращает управление, так как b не равна 0
WaitForSingleObject(b.GetNotZeroHandle(), INFINITE);

b = 0;

// возвращает управление, только если другой поток присваивает b ненулевое значение
WaitForSingleObject(b.GetNotZeroHandle(), INFINITE);
```

Программа-пример *InterlockedType*

Эта программа, «10 *InterlockedType.exe*» (см. листинг на рис. 10-2), предназначена для тестирования только что описанных классов. Файлы исходного кода и ресурсов этой

программы находятся в каталоге 10-InterlockedType на компакт-диске, прилагаемом к книге. Как я уже говорил, такие приложения я всегда запускаю под управлением отладчика, чтобы наблюдать за всеми функциями и переменными — членами классов.

Программа иллюстрирует типичный сценарий программирования, который выглядит так. Поток порождает несколько рабочих потоков, а затем инициализирует блок памяти. Далее основной поток пробуждает рабочие потоки, чтобы они начали обработку содержимого этого блока памяти. В данный момент основной поток должен приостановить себя до тех пор, пока все рабочие потоки не выполнят свои задачи. После этого основной поток записывает в блок памяти новые данные и вновь пробуждает рабочие потоки.

На примере этого кода хорошо видно, насколько тривиальным становится решение этой распространенной задачи программирования при использовании C++. Класс CWhenZero дает нам гораздо больше возможностей — не один лишь инверсный семафор. Мы получаем теперь безопасный в многопоточной среде объект данных, который переходит в свободное состояние, когда его значение обнуляется! Вы можете не только увеличивать и уменьшать счетчик семафора на 1, но и выполнять над ним любые математические и логические операции, в том числе сложение, вычитание, умножение, деление, вычисления по модулю! Так что объект CWhenZero намного функциональнее, чем объект ядра «семафор».

С этими классами шаблонов C++ можно много чего придумать. Например, создать класс CInterlockedString, производный от CInterlockedType, и с его помощью безопасно манипулировать символьными строками. А потом создать класс CWhenCertainString, производный от CInterlockedString, чтобы освобождать объект ядра «событие», когда строка принимает определенное значение (или значения). В общем, возможности безграничны.



IntLockTest.cpp

```

/*****
Модуль: IntLockTest.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <tchar.h>
#include "Interlocked.h"

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

// присваиваем TRUE, когда рабочие потоки должны завершиться
volatile BOOL g_fQuit = FALSE;

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

DWORD WINAPI WorkerThread(PVOID pvParam) {
    CWhenZero<BYTE>& bVal = * (CWhenZero<BYTE> *) pvParam;

    // должен ли рабочий поток завершиться?
    while (!g_fQuit) {

```

Рис. 10-2. Программа-пример InterlockedType

см. след. стр.

Рис. 10-2. *продолжение*

```
// ждем какой-нибудь работы
WaitForSingleObject(bVal.GetNotZeroHandle(), INFINITE);

// если мы должны выйти – выходим
if (g_fQuit)
    continue;

// что-то делаем
chMB("Worker thread: We have something to do");

bVal--; // сделали

// ждем, когда остановятся все рабочие потоки
WaitForSingleObject(bVal, INFINITE);
}

chMB("Worker thread: terminating");
return(0);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    // инициализируем объект так, чтобы указать:
    // ни одному рабочему потоку делать нечего
    CWhenZero<BYTE> bVal = 0;

    // создаем рабочие потоки
    const int nMaxThreads = 2;
    HANDLE hThreads[nMaxThreads];
    for (int nThread = 0; nThread < nMaxThreads; nThread++) {
        DWORD dwThreadId;
        hThreads[nThread] = CreateThread(NULL, 0,
            WorkerThread, (PVOID) &bVal, 0, &dwThreadId);
    }

    int n;
    do {
        // делать что-то еще или остановиться?
        n = MessageBox(NULL,
            TEXT("Yes: Give worker threads something to do\nNo: Quit"),
            TEXT("Primary thread"), MB_YESNO);

        // сообщаем рабочим потокам, что мы выходим
        if (n == IDNO)
            InterlockedExchangePointer((PVOID*) &g_fQuit, (PVOID) TRUE);

        bVal = nMaxThreads; // пробуждаем рабочие потоки

        if (n == IDYES) {
```


Рис. 10-2. продолжение

```

        // есть работа, ждем, когда рабочие потоки ее сделают
        WaitForSingleObject(bVal, INFINITE);
    }

    } while (n == IDYES);

    // работы больше нет, процесс надо завершить;
    // ждем завершения рабочих потоков
    WaitForMultipleObjects(nMaxThreads, hThreads, TRUE, INFINITE);

    // закрываем описатели рабочих потоков
    for (nThread = 0; nThread < nMaxThreads; nThread++)
        CloseHandle(hThreads[nThread]);

    // сообщаем пользователю, что процесс завершается
    chMB("Primary thread: terminating");

    return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

Interlocked.h

```

/*****
Модуль: Interlocked.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#pragma once

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

// к экземплярам этого класса будет обращаться множество потоков, поэтому
// все его члены (кроме конструктора и деструктора) должны быть безопасны
// в многопоточной среде
class CResGuard {
public:
    CResGuard() { m_lGrdCnt = 0; InitializeCriticalSection(&m_cs); }
    ~CResGuard() { DeleteCriticalSection(&m_cs); }

    // IsGuarded используется для отладки
    BOOL IsGuarded() const { return(m_lGrdCnt > 0); }

public:
    class CGuard {
    public:
        CGuard(CResGuard& rg) : m_rg(rg) { m_rg.Guard(); };
        ~CGuard() { m_rg.Unguard(); }
    };
};

```

см. след. стр.

Рис. 10-2. *продолжение*

```

private:
    CResGuard& m_rg;
};

private:
    void Guard() { EnterCriticalSection(&m_cs); m_lGrdCnt++; }
    void Unguard() { m_lGrdCnt--; LeaveCriticalSection(&m_cs); }

    // к Guard/Unguard может обращаться только вложенный класс CGuard
    friend class CResGuard::CGuard;

private:
    CRITICAL_SECTION m_cs;
    long m_lGrdCnt; // число вызовов EnterCriticalSection
};

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

// к экземплярам этого класса будет обращаться множество потоков, поэтому
// все его члены (кроме конструктора и деструктора) должны быть безопасны
// в многопоточной среде
template <class TYPE>
class CInterlockedType {

public: // открытые функции-члены
    // Примечание: конструкторы и деструкторы всегда безопасны
    // в многопоточной среде.
    CInterlockedType() { }
    CInterlockedType(const TYPE& TVal) { m_TVal = TVal; }
    virtual ~CInterlockedType() { }

    // оператор приведения, который упрощает написание кода с использованием
    // безопасного в многопоточной среде типа данных
    operator TYPE() const {
        CResGuard::CGuard x(m_rg);
        return(GetVal());
    }

protected: // защищенная функция, которую должен вызывать производный класс
    TYPE& GetVal() {
        chASSERT(m_rg.IsGuarded());
        return(m_TVal);
    }

    const TYPE& GetVal() const {
        assert(m_rg.IsGuarded());
        return(m_TVal);
    }

    TYPE SetVal(const TYPE& TNewVal) {
        chASSERT(m_rg.IsGuarded());

```

Рис. 10-2. *продолжение*

```

        TYPE& TVal = GetVal();
        if (TVal != TNewVal) {
            TYPE TPrevVal = TVal;
            TVal = TNewVal;
            OnValChanged(TNewVal, TPrevVal);
        }
        return(TVal);
    }

protected: // замещаемые функции
    virtual void OnValChanged(
        const TYPE& TNewVal, const TYPE& TPrevVal) const {
        // здесь ничего не делается
    }

protected:
    // защищенный член класса, охраняющий ресурс;
    // используется функциями производного класса
    mutable CResGuard m_rg;

private: // закрытые элементы данных
    TYPE m_TVal;
};

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

// к экземплярам этого класса будет обращаться множество потоков, поэтому
// все его члены (кроме конструктора и деструктора) должны быть безопасны
// в многопоточной среде
template <class TYPE>
class CInterlockedScalar : protected CInterlockedType<TYPE> {

public:
    CInterlockedScalar(TYPE TVal = 0) : CInterlockedType<TYPE>(TVal) {
    }

    ~CInterlockedScalar() { /* ничего не делает */ }

    // C++ не разрешает наследование оператора приведения типа
    operator TYPE() const {
        return(CInterlockedType<TYPE>::operator TYPE());
    }

    TYPE operator=(TYPE TVal) {
        CResGuard::CGuard x(m_rg);
        return(SetVal(TVal));
    }

    TYPE operator++(int) { // постфиксный оператор увеличения на 1
        CResGuard::CGuard x(m_rg);
        TYPE TPrevVal = GetVal();

```

см. след. стр.

Рис. 10-2. *продолжение*

```

        SetVal((TYPE) (TPrevVal + 1));
        return(TPrevVal); // возвращаем значение ДО увеличения
    }

    TYPE operator--(int) { // постфиксный оператор уменьшения на 1
        CResGuard::CGuard x(m_rg);
        TYPE TPrevVal = GetVal();
        SetVal((TYPE) (TPrevVal - 1));
        return(TPrevVal); // возвращаем значение ДО уменьшения
    }

    TYPE operator += (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() + op)); }
    TYPE operator++()
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() + 1)); }
    TYPE operator -= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() - op)); }
    TYPE operator--()
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() - 1)); }
    TYPE operator *= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() * op)); }
    TYPE operator /= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() / op)); }
    TYPE operator %= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() % op)); }
    TYPE operator ^= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() ^ op)); }
    TYPE operator &= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() & op)); }
    TYPE operator |= (TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() | op)); }
    TYPE operator <=<=(TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() << op)); }
    TYPE operator >>=(TYPE op)
    { CResGuard::CGuard x(m_rg); return(SetVal(GetVal() >> op)); }
};

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

// к экземплярам этого класса будет обращаться множество потоков, поэтому
// все его члены (кроме конструктора и деструктора) должны быть безопасны
// в многопоточной среде
template <class TYPE>
class CWhenZero : public CInterlockedScalar<TYPE> {
public:
    CWhenZero(TYPE TVal = 0, BOOL fManualReset = TRUE)
        : CInterlockedScalar<TYPE>(TVal) {

        // это событие должно освобождаться при TVal, равном 0
        m_hevtZero = CreateEvent(NULL, fManualReset, (TVal == 0), NULL);
    }
};

```

Рис. 10-2. *продолжение*

```

        // а это событие должно освобождаться при TVal, НЕ равном 0
        m_hevtNotZero = CreateEvent(NULL, fManualReset, (TVal != 0), NULL);
    }

    ~CWhenZero() {
        CloseHandle(m_hevtZero);
        CloseHandle(m_hevtNotZero);
    }

    // C++ не разрешает наследование оператора =
    TYPE operator=(TYPE x) {
        return(CInterlockedScalar<TYPE>::operator=(x));
    }

    // возвращаем описатель события, которое переходит
    // в свободное состояние при нулевом значении
    operator HANDLE() const { return(m_hevtZero); }
    // возвращаем описатель события, которое переходит
    // в свободное состояние при ненулевом значении
    HANDLE GetNotZeroHandle() const { return(m_hevtNotZero); }
    // C++ не разрешает наследование оператора приведения типа
    operator TYPE() const {
        return(CInterlockedScalar<TYPE>::operator TYPE());
    }

protected:
    void OnValChanged(const TYPE& TNewVal, const TYPE& TPrevVal) const {
        // для большего быстродействия избегайте перехода
        // в режим ядра без веских причин
        if ((TNewVal == 0) && (TPrevVal != 0)) {
            SetEvent(m_hevtZero);
            ResetEvent(m_hevtNotZero);
        }
        if ((TNewVal != 0) && (TPrevVal == 0)) {
            ResetEvent(m_hevtZero);
            SetEvent(m_hevtNotZero);
        }
    }

private:
    HANDLE m_hevtZero;           // освобождается, когда значение равно 0
    HANDLE m_hevtNotZero;       // освобождается, когда значение не равно 0
};

/////////////////////////////////////// Конец файла /////////////////////////////////////////

```

Синхронизация в сценарии «один писатель/группа читателей»

Во многих приложениях возникает одна и та же проблема синхронизации, о которой часто говорят как о сценарии «один писатель/группа читателей» (single-writer/multiple-readers). В чем ее суть? Представьте: произвольное число потоков пытается

получить доступ к некоему разделяемому ресурсу. Каким-то потокам («писателям») нужно модифицировать данные, а каким-то («читателям») — лишь прочесть эти данные. Синхронизация такого процесса необходима хотя бы потому, что Вы должны соблюдать следующие правила:

1. Когда один поток что-то пишет в область общих данных, другие этого делать не могут.
2. Когда один поток что-то пишет в область общих данных, другие не могут ничего считывать оттуда.
3. Когда один поток считывает что-то из области общих данных, другие не могут туда ничего записывать.
4. Когда один поток считывает что-то из области общих данных, другие тоже могут это делать.

Посмотрим на проблему в контексте базы данных. Допустим, с ней работают пять конечных пользователей: двое вводят в нее записи, трое — считывают.

В этом сценарии правило 1 необходимо потому, что мы, конечно же, не можем позволить одновременно обновлять одну и ту же запись. Иначе информация в записи будет повреждена.

Правило 2 запрещает доступ к записи, обновляемой в данный момент другим пользователем. Будь то иначе, один пользователь считывал бы запись, когда другой пользователь изменял бы ее содержимое. Что увидел бы на мониторе своего компьютера первый пользователь, предсказать не берусь. Правило 3 служит тем же целям, что и правило 2. И действительно, какая разница, кто первый получит доступ к данным: тот, кто записывает, или тот, кто считывает, — все равно одновременно этого делать нельзя.

И, наконец, последнее правило. Оно введено для большей эффективности работы баз данных. Если никто не модифицирует записи в базе данных, все пользователи могут свободно читать любые записи. Также предполагается, что количество «читателей» превышает число «писателей».

О'кэй, суть проблемы Вы ухватили. А теперь вопрос: как ее решить?



Я представлю здесь совершенно новый код. Решения этой проблемы, которые я публиковал в прежних изданиях, часто критиковались по двум причинам. Во-первых, предыдущие реализации работали слишком медленно, так как я писал их в расчете на самые разные сценарии. Например, я шире использовал объекты ядра, стремясь синхронизировать доступ к базе данных потоков из разных процессов. Конечно, эти реализации работали и в сценарии для одного процесса, но интенсивное использование объектов ядра приводило в этом случае к существенным издержкам. Похоже, сценарий для одного процесса более распространен, чем я думал.

Во-вторых, в моей реализации был потенциальный риск блокировки потоков-«писателей». Из правил, о которых я рассказал в начале этого раздела, вытекает, что потоки-«писатели» — при обращении к базе данных очень большого количества потоков-«читателей» — могут вообще не получить доступ к этому ресурсу.

Все эти недостатки я теперь устранил. В новой реализации объекты ядра применяются лишь в тех случаях, когда без них не обойтись, и потоки синхронизируются в основном за счет использования критической секции.

Плоды своих трудов я инкапсулировал в C++-класс CSWMRG (я произношу его название как *swimmerge*); это аббревиатура от «single writer/multiple reader guard». Он содержится в файлах SWMRG.h и SWMRG.cpp (см. листинг на рис. 10-3).

Использовать CSWMRG проще простого. Вы создаете объект C++-класса CSWMRG и вызываете нужные в Вашей программе функции-члены. В этом классе всего три метода (не считая конструктора и деструктора):

```
VOID CSWMRG::WaitToRead();    // доступ к разделяемому ресурсу для чтения
VOID CSWMRG::WaitToWrite();   // монопольный доступ к разделяемому ресурсу для записи
VOID CSWMRG::Done();          // вызывается по окончании работы с ресурсом
```

Первый метод (*WaitToRead*) вызывается перед выполнением кода, что-либо считывающего из разделяемого ресурса, а второй (*WaitToWrite*) — перед выполнением кода, который считывает и записывает данные в разделяемом ресурсе. К последнему методу (*Done*) программа обращается, закончив работу с этим ресурсом. Куда уж проще, а?

Объект CSWMRG содержит набор переменных-членов (см. таблицу ниже), отражающих то, как потоки работают с разделяемым ресурсом на данный момент. Остальные подробности Вы узнаете из исходного кода.

| Переменная | Описание |
|--------------------------|--|
| <i>m_cs</i> | Охраняет доступ к остальным членам класса, обеспечивая операции с ними на атомарном уровне. |
| <i>m_nActive</i> | Отражает текущее состояние разделяемого ресурса. Если она равна 0, ни один поток к ресурсу не обращается. Ее значение, большее 0, сообщает текущее число потоков, считывающих данные из ресурса. Отрицательное значение (–1) свидетельствует о том, что какой-то поток записывает данные в ресурс. |
| <i>m_nWaitingReaders</i> | Сообщает количество потоков-«читателей», которым нужен доступ к ресурсу. Значение этой переменной инициализируется 0 и увеличивается на 1 всякий раз, когда поток вызывает <i>WaitToRead</i> в то время, как <i>m_nActive</i> равна –1. |
| <i>m_nWaitingWriters</i> | Сообщает количество потоков-«писателей», которым нужен доступ к ресурсу. Значение этой переменной инициализируется 0 и увеличивается на 1 всякий раз, когда поток вызывает <i>WaitToWrite</i> в то время, как <i>m_nActive</i> больше 0. |
| <i>m_bsemWriters</i> | Когда потоки-«писатели» вызывают <i>WaitToWrite</i> , но получают отказ в доступе, так как <i>m_nActive</i> больше 0, они переходят в состояние ожидания этого семафора. Пока ждет хотя бы один поток-«писатель», новые потоки-«читатели» получают отказ в доступе к ресурсу. Тем самым я не даю потокам-«читателям» монополизировать доступ к этому ресурсу. Когда последний поток-«читатель», работавший с ресурсом, вызывает <i>Done</i> , семафор освобождается со счетчиком, равным 1, и система пробуждает один ждущий поток-«писатель». |
| <i>m_bsemReaders</i> | Когда потоки-«читатели» вызывают <i>WaitToRead</i> , но получают отказ в доступе, так как <i>m_nActive</i> равна –1, они переходят в состояние ожидания этого семафора. Когда последний из ждущих потоков-«писателей» вызывает <i>Done</i> , семафор освобождается со счетчиком, равным <i>m_nWaitingReaders</i> , и система пробуждает все ждущие потоки-«читатели». |

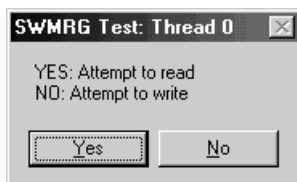
Программа-пример SWMRG

Эта программа, «10 SWMRG.exe» (см. листинг на рис. 10-3), предназначена для тестирования C++-класса CSWMRG. Файлы исходного кода и ресурсов этой программы

находятся в каталоге 10-SWMRG на компакт-диске, прилагаемом к книге. Я запускаю это приложение под управлением отладчика, чтобы наблюдать за всеми функциями и переменными — членами классов.

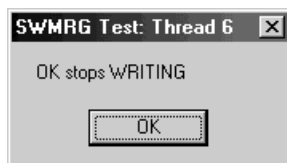
При запуске программы первичный поток создает несколько потоков, выполняющих одну и ту же функцию. Далее первичный поток вызывает *WaitForMultipleObjects* и ждет завершения этих потоков. Когда все они завершаются, их описатели закрываются и процесс прекращает свое существование.

Каждый вторичный поток выводит на экран такое сообщение:



Чтобы данный поток имитировал чтение ресурса, щелкните кнопку Yes, а чтобы он имитировал запись в ресурс — кнопку No. Эти действия просто заставляют его вызвать либо функцию *WaitToRead*, либо функцию *WaitToWrite* объекта CSWMRG.

После вызова одной из этих функций поток выводит соответствующее сообщение.



Пока окно с сообщением открыто, программа приостанавливает поток и делает вид, будто он сейчас работает с ресурсом.

Конечно, если какой-то поток читает данные из ресурса и Вы командуете другому потоку записать данные в ресурс, окно с сообщением от последнего на экране не появится, так как поток-«писатель» ждет освобождения ресурса, вызвав *WaitToWrite*. Аналогичным образом, если Вы командуете потоку считать данные из ресурса в то время, как показывается окно с сообщением от потока-«писателя», первый поток будет ждать в вызове *WaitToRead*, и его окно не появится до тех пор, пока все потоки-«писатели» не закончат имитировать свою работу с ресурсом.

Закрыв окно с сообщением (щелчком кнопки OK), Вы заставите поток, получивший доступ к ресурсу, вызвать *Done*, и объект CSWMRG переключится на другие ждущие потоки.



SWMRG.cpp

```

/*****
Модуль: SWMRG.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include "SWMRG.h"

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

```

Рис. 10-3. Программа-пример SWMRG

Рис. 10-3. *продолжение*

```
CSWMRG::CSWMRG() {
    // изначально ресурс никому не нужен, и никто к нему не обращается
    m_nWaitingReaders = m_nWaitingWriters = m_nActive = 0;
    m_hsemReaders = CreateSemaphore(NULL, 0, MAXLONG, NULL);
    m_hsemWriters = CreateSemaphore(NULL, 0, MAXLONG, NULL);
    InitializeCriticalSection(&m_cs);
}

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

CSWMRG::~CSWMRG() {
#ifdef _DEBUG
    // SWMRG нельзя уничтожать, если потоки пользуются ресурсом
    if (m_nActive != 0)
        DebugBreak();
#endif

    m_nWaitingReaders = m_nWaitingWriters = m_nActive = 0;
    DeleteCriticalSection(&m_cs);
    CloseHandle(m_hsemReaders);
    CloseHandle(m_hsemWriters);
}

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

VOID CSWMRG::WaitToRead() {
    // обеспечиваем монопольный доступ к переменным-членам
    EnterCriticalSection(&m_cs);

    // работает ли сейчас с ресурсом какой-нибудь поток-"писатель"
    // и есть ли "писатели", ждущие этот ресурс?
    BOOL fResourceWritePending = (m_nWaitingWriters || (m_nActive < 0));

    if (fResourceWritePending) {
        // этот "читатель" должен ждать,
        // увеличиваем счетчик числа ждущих "читателей" на 1
        m_nWaitingReaders++;
    } else {
        // этот "читатель" может читать,
        // увеличиваем счетчик числа активных "читателей" на 1
        m_nActive++;
    }

    // разрешаем другим потокам попытаться получить доступ для чтения или записи
    LeaveCriticalSection(&m_cs);

    if (fResourceWritePending) {
```

см. след. стр.

Рис. 10-3. *продолжение*

```

        // этот поток должен ждать
        WaitForSingleObject(m_hsemReaders, INFINITE);
    }
}

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

VOID CSWMRG::WaitToWrite() {

    // обеспечиваем монопольный доступ к переменным-членам
    EnterCriticalSection(&m_cs);

    // работают ли сейчас с ресурсом какие-нибудь потоки?
    BOOL fResourceOwned = (m_nActive != 0);

    if (fResourceOwned) {

        // этот "писатель" должен ждать,
        // увеличиваем счетчик числа ждущих "писателей" на 1
        m_nWaitingWriters++;
    } else {

        // этот "писатель" может писать,
        // уменьшаем счетчик числа активных "писателей" до -1
        m_nActive = -1;
    }

    // разрешаем другим потокам попытаться получить доступ для чтения или записи
    LeaveCriticalSection(&m_cs);

    if (fResourceOwned) {

        // этот поток должен ждать
        WaitForSingleObject(m_hsemWriters, INFINITE);
    }
}

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

VOID CSWMRG::Done() {

    // обеспечиваем монопольный доступ к переменным-членам
    EnterCriticalSection(&m_cs);

    if (m_nActive > 0) {

        // ресурс контролируют "читатели", значит, убираем одного из них так
        m_nActive--;
    } else {

        // ресурс контролируют "писатели", значит, убираем одного из них так

```

Рис. 10-3. *продолжение*

```

        m_nActive++;
    }

    HANDLE hsem = NULL; // предполагаем, что ждущих потоков нет
    LONG lCount = 1;    // предполагаем, что пробуждается только один ждущий поток
                        // (в отношении "писателей" это всегда так)

    if (m_nActive == 0) {

        // Ресурс свободен, кого пробудить?
        // Примечание: "читатели" никогда не получают доступ к ресурсу,
        // если его всегда будут ждать "писатели".

        if (m_nWaitingWriters > 0) {

            // ресурс ждут "писатели", а они имеют приоритет перед "читателями"
            m_nActive = -1;           // писатель получит доступ
            m_nWaitingWriters--;      // одним ждущим "писателем" станет меньше
            hsem = m_hsemWriters;     // "писатели" ждут на этом семафоре
            // Примечание: семафор откроет путь только одному потоку—"писателю".

        } else if (m_nWaitingReaders > 0) {

            // ресурс ждут "читатели", а "писателей" нет
            m_nActive = m_nWaitingReaders; // все "читатели" получают доступ
            m_nWaitingReaders = 0;         // ждущих "читателей" не останется
            hsem = m_hsemReaders;          // "читатели" ждут на этом семафоре
            lCount = m_nActive;            // семафор откроет путь всем "читателям"
        } else {

            // ждущих потоков вообще нет
        }
    }

    // разрешаем другим потокам попытаться получить доступ для чтения или записи
    LeaveCriticalSection(&m_cs);

    if (hsem != NULL) {
        // некоторые потоки следует пробудить
        ReleaseSemaphore(hsem, lCount, NULL);
    }
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

SWMRG.h

```

/*****
Модуль: SWMRG.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

```

см. след. стр.

Рис. 10-3. *продолжение*

```
#pragma once

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

class CSWMRG {
public:
    CSWMRG();           // конструктор
    ~CSWMRG();          // деструктор

    VOID WaitToRead();   // предоставляет доступ к разделяемому ресурсу для чтения
    VOID WaitToWrite();  // предоставляет монопольный доступ к разделяемому
                        // ресурсу для записи
    VOID Done();         // вызывается по окончании работы с ресурсом

private:
    CRITICAL_SECTION m_cs; // обеспечивает монопольный доступ к другим элементам
    HANDLE m_hsemReaders;  // "читатели" ждут на этом семафоре,
                        // если ресурс занят "писателем"
    HANDLE m_hsemWriters;  // "писатели" ждут на этом семафоре,
                        // если ресурс занят "читателем"
    int m_nWaitingReaders; // число ждущих "читателей"
    int m_nWaitingWriters; // число ждущих "писателей"
    int m_nActive;         // текущее число потоков, работающих с ресурсом
                        // (0 – таких потоков нет, >0 – число "читателей",
                        // -1 – один "писатель")
};

//////////////////////////////////// Конец файла //////////////////////////////////////
```

SWMRGTest.cpp

```
/*
*****
Модуль: SWMRGTest.Cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****
*/

#include "..\CmnHdr.h" /* см. приложение A */
#include <tchar.h>
#include <process.h>    // для доступа к _beginthreadex
#include "SWMRG.h"

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

// глобальный синхронизирующий объект Single-Writer/Multiple-Reader Guard
CSWMRG g_swmrg;

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

DWORD WINAPI Thread(PVOID pvParam) {

    TCHAR sz[50];
```

Рис. 10-3. продолжение

```

wprintf(sz, TEXT("SWMRG Test: Thread %d"), PtrToShort(pvParam));
int n = MessageBox(NULL,
    TEXT("YES: Attempt to read\nNO: Attempt to write"), sz, MB_YESNO);

// попытка чтения или записи
if (n == IDYES)
    g_swmrg.WaitToRead();
else
    g_swmrg.WaitToWrite();

MessageBox(NULL,
    (n == IDYES) ? TEXT("OK stops READING") : TEXT("OK stops WRITING"),
    sz, MB_OK);

// прекращаем чтение или запись
g_swmrg.Done();
return(0);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    // порожаем серию потоков, пытающихся читать или записывать
    HANDLE hThreads[MAXIMUM_WAIT_OBJECTS];
    for (int nThreads = 0; nThreads < 8; nThreads++) {
        DWORD dwThreadId;
        hThreads[nThreads] =
            chBEGINTHREADEX(NULL, 0, Thread, (PVOID) (DWORD_PTR) nThreads,
                0, &dwThreadId);
    }

    // ждем завершения всех потоков
    WaitForMultipleObjects(nThreads, hThreads, TRUE, INFINITE);
    while (nThreads--)
        CloseHandle(hThreads[nThreads]);

    return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////

```

Реализация функции *WaitForMultipleExpressions*

Некоторое время назад я разрабатывал одно приложение и столкнулся с весьма непростым случаем синхронизации потоков. Функции *WaitForMultipleObjects*, заставляющей поток ждать освобождения одного или всех объектов, оказалось недостаточно. Мне понадобилась функция, которая позволяла бы задавать более сложные критерии ожидания. У меня было три объекта ядра: процесс, семафор и событие. Мой поток должен был ждать до тех пор, пока не освободится либо процесс и семафор, либо процесс и событие.

Слегка поразмыслив и творчески использовав имеющиеся функции Windows, я создал именно то, что мне требовалось, — функцию *WaitForMultipleExpressions*. Ее прототип выглядит так:

```
DWORD WINAPI WaitForMultipleExpressions(
    DWORD nExpObjects,
    CONST HANDLE* phExpObjects,
    DWORD dwMilliseconds);
```

Перед ее вызовом Вы должны создать массив описателей (HANDLE) и инициализировать все его элементы. Параметр *nExpObjects* сообщает число элементов в массиве, на который указывает параметр *phExpObjects*. Этот массив содержит несколько наборов описателей объектов ядра; при этом каждый набор отделяется элементом, равным NULL. Функция *WaitForMultipleExpressions* считает все объекты в одном наборе объединяемыми логической операцией AND, а сами наборы — объединяемыми логической операцией OR. Поэтому *WaitForMultipleExpressions* приостанавливает вызывающий поток до тех пор, пока не освободятся сразу все объекты в одном из наборов.

Вот пример. Допустим, мы работаем с четырьмя объектами ядра (см. таблицу ниже).

| Объект ядра | Значение описателя |
|-------------|--------------------|
| Поток | 0x1111 |
| Семафор | 0x2222 |
| Событие | 0x3333 |
| Процесс | 0x4444 |

Инициализировав массив описателей, как показано в следующей таблице, мы сообщаем функции *WaitForMultipleExpressions* приостановить вызывающий поток до тех пор, пока не освободятся поток AND семафор OR семафор AND событие AND процесс OR поток AND процесс.

| Индекс | Значение описателя | Набор |
|--------|--------------------|-------|
| 0 | 0x1111 (поток) | 0 |
| 1 | 0x2222 (семафор) | |
| 2 | 0x0000 (OR) | |
| 3 | 0x2222 (семафор) | 1 |
| 4 | 0x3333 (событие) | |
| 5 | 0x4444 (процесс) | |
| 6 | 0x0000 (OR) | 2 |
| 7 | 0x1111 (поток) | |
| 8 | 0x4444 (процесс) | |

Вы, наверное, помните, что функции *WaitForMultipleObjects* нельзя передать массив описателей, число элементов в котором превышает 64 (MAXIMUM_WAIT_OBJECTS). Так вот, при использовании *WaitForMultipleExpressions* массив описателей может быть гораздо больше. Однако у Вас не должно быть более 64 выражений, а в каждом — более 63 описателей. Кроме того, *WaitForMultipleExpressions* будет работать некорректно, если Вы передадите ей хотя бы один описатель мьютекса. (Почему — объясню позже.)

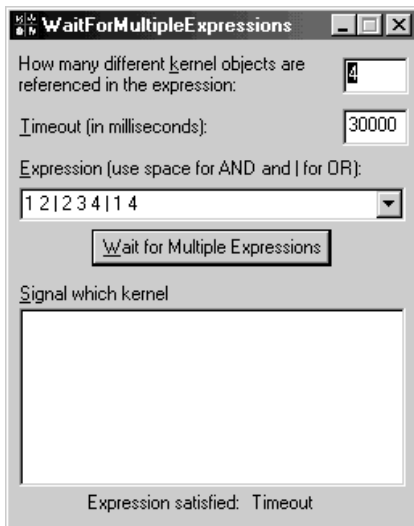
Возвращаемые значения функции *WaitForMultipleExpressions* показаны в следующей таблице. Если заданное выражение становится истинным, *WaitForMultipleExpressions*

возвращает индекс этого выражения относительно `WAIT_OBJECT_0`. Если взять тот же пример, то при освобождении объектов «поток» и «процесс» *WaitForMultipleExpressions* вернет индекс в виде `WAIT_OBJECT_0 + 2`.

| Возвращаемое значение | Описание |
|---|---|
| От <code>WAIT_OBJECT_0</code> до <code>(WAIT_OBJECT_0</code> + число выражений – 1) | Указывает, какое выражение стало истинным. |
| <code>WAIT_TIMEOUT</code> | Ни одно выражение не стало истинным в течение заданного времени. |
| <code>WAIT_FAILED</code> | Произошла ошибка. Чтобы получить более подробную информацию, вызовите <i>GetLastError</i> . Код <code>ERROR_TOO_MANY_SECRETS</code> означает, что Вы указали более 64 выражений, а <code>ERROR_SECRET_TOO_LONG</code> — что по крайней мере в одном выражении указано более 63 объектов. Могут возвращаться коды и других ошибок. |

Программа-пример WaitForMultExp

Эта программа, «10 WaitForMultExp.exe» (см. листинг на рис. 10-4), предназначена для тестирования функции *WaitForMultipleExpressions*. Файлы исходного кода и ресурсов этой программы находятся в каталоге 10-WaitForMultExp на компакт-диске, прилагаемом к книге. После запуска WaitForMultExp открывается диалоговое окно, показанное ниже.

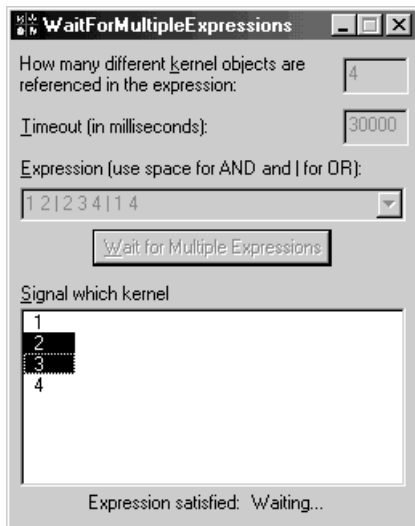


Если Вы не станете изменять предлагаемые параметры, а просто щелкнете кнопку *Wait For Multiple Expressions*, диалоговое окно будет выглядеть так, как показано на следующей иллюстрации.

Программа создает четыре объекта ядра «событие» в занятом состоянии и помещает в многоколоночный список (с возможностью выбора сразу нескольких элементов) по одной записи для каждого объекта ядра. Далее программа анализирует содержимое поля *Expression* и формирует массив описателей. По умолчанию я предлагаю объекты ядра и выражение, как в предыдущем примере.

Поскольку я задал время ожидания равным 30000 мс, у Вас есть 30 секунд на внесение изменений. Выбор элемента в нижнем списке приводит к вызову *SetEvent*, ко-

торая освобождает объект, а отказ от его выбора — к вызову *ResetEvent* и соответственно к переводу объекта в занятое состояние. После выбора достаточного числа элементов (удовлетворяющего одному из выражений) *WaitForMultipleExpressions* возвращает управление, и в нижней части диалогового окна показывается, какому выражению удовлетворяет Ваш выбор. Если Вы не уложитесь в 30 секунд, появится слово «Timeout».



Теперь обсудим мою функцию *WaitForMultipleExpressions*. Реализовать ее было не просто, и ее применение, конечно, приводит к некоторым издержкам. Как Вы знаете, в Windows есть функция *WaitForMultipleObjects*, которая позволяет потоку ждать по единственному AND-выражению:

```
DWORD WaitForMultipleObjects(
    DWORD dwObjects,
    CONST HANDLE* phObjects,
    BOOL fWaitAll,
    DWORD dwMilliseconds);
```

Чтобы расширить ее функциональность для поддержки выражений, объединяемых OR, я должен создать несколько потоков — по одному на каждое такое выражение. Каждый из этих потоков ждет в вызове *WaitForMultipleObjectsEx* по единственному AND-выражению. (Почему я использую эту функцию вместо более распространенной *WaitForMultipleObjects* — станет ясно позже.) Когда какое-то выражение становится истинным, один из созданных потоков пробуждается и завершается.

Поток, который вызвал *WaitForMultipleExpressions* (и который породил все OR-потоки), должен ждать, пока одно из OR-выражений не станет истинным. Для этого он вызывает функцию *WaitForMultipleObjectsEx*. В параметре *dwObjects* передается количество порожденных потоков (OR-выражений), а параметр *phObjects* указывает на массив описателей этих потоков. В параметр *fWaitAll* записывается FALSE, чтобы основной поток пробудился сразу после того, как станет истинным любое из выражений. И, наконец, в параметре *dwMilliseconds* передается значение, идентичное тому, которое было указано в аналогичном параметре при вызове *WaitForMultipleExpressions*.

Если в течение заданного времени ни одно из выражений не становится истинным, *WaitForMultipleObjectsEx* возвращает WAIT_TIMEOUT, и это же значение возвращается функцией *WaitForMultipleExpressions*. А если какое-нибудь выражение становится

ся истинным, *WaitForMultipleObjectsEx* возвращает индекс, указывающий, какой поток завершился. Так как каждый поток представляет отдельное выражение, этот индекс сообщает и то, какое выражение стало истинным; этот же индекс возвращается и функцией *WaitForMultipleExpressions*.

На этом мы, пожалуй, закончим рассмотрение того, как работает функция *WaitForMultipleExpressions*. Но нужно обсудить еще три вещи. Во-первых, нельзя допустить, чтобы несколько OR-потоков одновременно пробудились в своих вызовах *WaitForMultipleObjectsEx*, так как успешное ожидание некоторых объектов ядра приводит к изменению их состояния (например, у семафора счетчик уменьшается на 1). *WaitForMultipleExpressions* ждет лишь до тех пор, пока одно из выражений не станет истинным, а значит, я должен предотвратить более чем однократное изменение состояния объекта.

Решить эту проблему на самом деле довольно легко. Прежде чем порождать OR-потоки, я создаю собственный объект-семафор с начальным значением счетчика, равным 1. Далее каждый OR-поток вызывает *WaitForMultipleObjectsEx* и передает ей не только описатели объектов, связанных с выражением, но и описатель этого семафора. Теперь Вы понимаете, почему в каждом наборе не может быть более 63 описателей? Чтобы OR-поток пробудился, должны освободиться все объекты, которые он ждет, — в том числе мой специальный семафор. Поскольку начальное значение его счетчика равно 1, более одного OR-потока никогда не пробудится, и, следовательно, случайного изменения состояния каких-либо других объектов не произойдет.

Второе, на что нужно обратить внимание, — как заставить ждущий поток прекратить ожидание для корректной очистки. Добавление семафора гарантирует, что пробудится не более чем один поток, но, раз мне уже известно, какое выражение стало истинным, я должен пробудить и остальные потоки, чтобы они корректно завершились. Вызова *TerminateThread* следует избегать, поэтому нужен какой-то другой механизм. Поразмыслив, я вспомнил, что потоки, ждущие в «тревожном» состоянии, принудительно пробуждаются, когда в APC-очереди появляется какой-нибудь элемент.

Моя реализация *WaitForMultipleExpressions* для принудительного пробуждения потоков использует *QueueUserAPC*. После того как *WaitForMultipleObjects*, вызванная основным потоком, возвращает управление, я ставлю APC-вызов в соответствующие очереди каждого из все еще ждущих OR-потоков:

```
// выводим все еще ждущие потоки из состояния сна,
// чтобы они могли корректно завершиться
for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

    if ((WAIT_TIMEOUT == dwWaitRet) || (dwExpNum != (dwWaitRet - WAIT_OBJECT_0))) {
        QueueUserAPC(WFME_ExpressionAPC, ahThreads[dwExpNum], 0);
    }
}
```

Функция обратного вызова, *WFME_ExpressionAPC*, выглядит столь странно потому, что на самом деле от нее не требуется ничего, кроме одного: прервать ожидание потока.

```
// это APC-функция обратного вызова
VOID WINAPI WFME_ExpressionAPC(DWORD dwData) {
    // в тело функции преднамеренно не включено никаких операторов
}
```

Третье (и последнее) — правильная обработка интервалов ожидания. Если никакие выражения так и не стали истинными в течение заданного времени, функция

WaitForMultipleObjects, вызванная основным потоком, возвращает WAIT_TIMEOUT. В этом случае я должен позаботиться о том, чтобы ни одно выражение больше не стало бы истинным и тем самым не изменило бы состояние объектов. За это отвечает следующий код:

```
// ждем, когда выражение станет TRUE или когда истечет срок ожидания
dwWaitRet = WaitForMultipleObjects(dwExpNum, ahThreads, FALSE, dwMilliseconds);

if (WAIT_TIMEOUT == dwWaitRet) {

    // срок ожидания истек; выясняем, не стало ли какое-нибудь выражение
    // истинным, проверяя состояние семафора hsemOnlyOne
    dwWaitRet = WaitForSingleObject(hsemOnlyOne, 0);

    if (WAIT_TIMEOUT == dwWaitRet) {

        // если семафор не был переведен в свободное состояние,
        // какое-то выражение дало TRUE; надо выяснить – какое
        dwWaitRet = WaitForMultipleObjects(dwExpNum,
            ahThreads, FALSE, INFINITE);

    } else {

        // ни одно выражение не стало TRUE,
        // и WaitForSingleObject просто отдала нам семафор
        dwWaitRet = WAIT_TIMEOUT;

    }

}
```

Я не даю другим выражениям стать истинными за счет ожидания на семафоре. Это приводит к уменьшению счетчика семафора до 0, и никакой OR-поток не может пробудиться. Но где-то после вызова функции *WaitForMultipleObjects* из основного потока и обращения той к *WaitForSingleObject* одно из выражений может стать истинным. Вот почему я проверяю значение, возвращаемое *WaitForSingleObject*. Если она возвращает WAIT_OBJECT_0, значит, семафор захвачен основным потоком и ни одно из выражений не стало истинным. Но если она возвращает WAIT_TIMEOUT, какое-то выражение все же стало истинным, прежде чем основной поток успел захватить семафор. Чтобы выяснить, какое именно выражение дало TRUE, основной поток снова вызывает *WaitForMultipleObjects*, но уже с временем ожидания, равным INFINITE; здесь все в порядке, так как я знаю, что семафор захвачен OR-поток и этот поток вот-вот завершится. Теперь я должен пробудить остальные OR-поток, чтобы корректно завершить их. Это делается в цикле, из которого вызывается *QueueUserAPC* (о ней я уже рассказывал).

Поскольку реализация *WaitForMultipleExpressions* основана на использовании группы потоков, каждый из которых ждет на своем наборе объектов, объединяемых по AND, мьютексы в ней неприменимы. В отличие от остальных объектов ядра мьютексы могут передаваться потоку во владение. Значит, если какой-нибудь из моих AND-потоков заполучит мьютекс, то по его завершении произойдет отказ от мьютекса. Вот когда Microsoft добавит в Windows API функцию, позволяющую одному потоку передавать права на владение мьютексом другому потоку, тогда моя функция *WaitForMultipleExpressions* и сможет поддерживать мьютексы. А пока надежного и корректного способа ввести в *WaitForMultipleExpressions* такую поддержку я не вижу.



WaitForMultExp.cpp

```

/*****
Модуль: WaitForMultExp.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h" /* см. приложение A */
#include <malloc.h>
#include <process.h>
#include "WaitForMultExp.h"

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

// внутренняя структура данных, которая представляет одно выражение;
// используется для того, чтобы сообщать OR-потокам,
// на каких объектах они должны ждать
typedef struct {
    PHANDLE m_phExpObjects; // указывает на набор описателей
    DWORD m_nExpObjects; // количество описателей
} EXPRESSION, *PEXPRESSON;

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

// функция OR-потока
DWORD WINAPI WFME_ThreadExpression(PVOID pvParam) {

    // Эта функция просто ждет, когда выражение станет истинным.
    // Ее поток ждет в "тревожном" состоянии, чтобы его ожидание
    // можно было принудительно прервать, поместив в его APC-очередь
    // APC-вызов.
    PEXPRESSION pExpression = (PEXPRESSON) pvParam;
    return(WaitForMultipleObjectsEx(
        pExpression->m_nExpObjects, pExpression->m_phExpObjects,
        TRUE, INFINITE, TRUE));
}

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

// это APC-функция обратного вызова
VOID WINAPI WFME_ExpressionAPC(ULONG_PTR dwData) {

    // в тело функции преднамеренно не включено никаких операторов
}

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

// эта функция ждет по нескольким булевым выражениям
DWORD WINAPI WaitForMultipleExpressions(DWORD nExpObjects,
    CONST HANDLE* phExpObjects, DWORD dwMilliseconds) {

```

Рис. 10-4. Программа-пример WaitForMultExp

см. след. стр.

Рис. 10-4. *продолжение*

```
// создаем временный массив, потому что нам придется модифицировать
// передаваемый массив и добавить в его конец описатель для
// семафора hsemOnlyOne
PHANDLE phExpObjectsTemp = (PHANDLE)
    _alloca(sizeof(HANDLE) * (nExpObjects + 1));
CopyMemory(phExpObjectsTemp, phExpObjects, sizeof(HANDLE) * nExpObjects);
phExpObjectsTemp[nExpObjects] = NULL; // ставим сюда часового

// этот семафор гарантирует, что только одно выражение станет истинным
HANDLE hsemOnlyOne = CreateSemaphore(NULL, 1, 1, NULL);

// информация о выражении
EXPRESSION Expression[MAXIMUM_WAIT_OBJECTS];

DWORD dwExpNum = 0; // номер текущего выражения
DWORD dwNumExps = 0; // общее количество выражений

DWORD dwObjBegin = 0; // первый индекс набора
DWORD dwObjCur = 0; // текущий индекс объекта в наборе

DWORD dwThreadId, dwWaitRet = 0;

// массив описателей потоков
HANDLE ahThreads[MAXIMUM_WAIT_OBJECTS];

// разбираем список описателей вызывающих потоков, инициализируя структуру
// для каждого выражения и добавляя к нему hsemOnlyOne
while ((dwWaitRet != WAIT_FAILED) && (dwObjCur <= nExpObjects)) {

    // пока ошибок нет и описатели объектов находятся в списке вызывающего...

    // находим следующее выражение (OR-выражения разделяются NULL-описателями)
    while (phExpObjectsTemp[dwObjCur] != NULL)
        dwObjCur++;

    // инициализируем структуру Expression, на которой ждет OR-поток
    phExpObjectsTemp[dwObjCur] = hsemOnlyOne;
    Expression[dwNumExps].m_phExpObjects = &phExpObjectsTemp[dwObjBegin];
    Expression[dwNumExps].m_nExpObjects = dwObjCur - dwObjBegin + 1;

    if (Expression[dwNumExps].m_nExpObjects > MAXIMUM_WAIT_OBJECTS) {
        // ошибка: слишком много описателей в одном выражении
        dwWaitRet = WAIT_FAILED;
        SetLastError(ERROR_SECRET_TOO_LONG);
    }

    // переходим к следующему выражению
    dwObjBegin = ++dwObjCur;
    if (++dwNumExps == MAXIMUM_WAIT_OBJECTS) {
        // ошибка: слишком много выражений
        dwWaitRet = WAIT_FAILED;
    }
}
```

Рис. 10-4. *продолжение*

```

        SetLastError(ERROR_TOO_MANY_SECRETS);
    }
}

if (dwWaitRet != WAIT_FAILED) {

    // при разборе списка описателей ошибки не обнаружены

    // порождаем поток для каждого выражения
    for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

        ahThreads[dwExpNum] = chBEGINTHREADEX(NULL,
            1, // нам нужен лишь небольшой стек
            WFME_ThreadExpression, &Expression[dwExpNum],
            0, &dwThreadId);
    }

    // ждем, когда выражение станет TRUE или когда истечет срок ожидания
    dwWaitRet = WaitForMultipleObjects(dwExpNum, ahThreads,
        FALSE, dwMilliseconds);

    if (WAIT_TIMEOUT == dwWaitRet) {

        // срок ожидания истек; выясняем, не стало ли какое-нибудь выражение
        // истинным, проверяя состояние семафора hsemOnlyOne
        dwWaitRet = WaitForSingleObject(hsemOnlyOne, 0);

        if (WAIT_TIMEOUT == dwWaitRet) {

            // если семафор не был переведен в свободное состояние,
            // какое-то выражение дало TRUE; надо выяснить – какое
            dwWaitRet = WaitForMultipleObjects(dwExpNum,
                ahThreads, FALSE, INFINITE);

        } else {

            // ни одно выражение не стало TRUE,
            // и WaitForSingleObject просто отдала нам семафор
            dwWaitRet = WAIT_TIMEOUT;
        }
    }

    // прерываем ожидание всех потоков (ждуших свои выражения),
    // чтобы они могли корректно завершиться
    for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {

        if ((WAIT_TIMEOUT == dwWaitRet) ||
            (dwExpNum != (dwWaitRet - WAIT_OBJECT_0))) {
            QueueUserAPC(WFME_ExpressionAPC, ahThreads[dwExpNum], 0);
        }
    }
}

```

см. след. стр.

Рис. 10-4. *продолжение*

```

#ifdef _DEBUG
    // при отладочной сборке ждем завершения всех потоков выражений,
    // а при окончательной сборке считаем, что все работает, как надо,
    // и не заставляем этот поток ждать их завершения
    WaitForMultipleObjects(dwExpNum, ahThreads, TRUE, INFINITE);
#endif

    // закрываем свои описатели всех потоков выражений
    for (dwExpNum = 0; dwExpNum < dwNumExps; dwExpNum++) {
        CloseHandle(ahThreads[dwExpNum]);
    }
} // при разборе произошла ошибка

CloseHandle(hsemOnlyOne);
return(dwWaitRet);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

WaitForMultExp.h

```

/*****
Модуль: WaitForMultExp.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#pragma once

//////////////////////////////////// Конеч файл //////////////////////////////////////

DWORD WINAPI WaitForMultipleExpressions(DWORD nExpObjects,
    CONST HANDLE* phExpObjects, DWORD dwMilliseconds);

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

WfMETest.cpp

```

/*****
Модуль: WfMETest.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <process.h>
#include "resource.h"
#include "WaitForMultExp.h"

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

см. след. стр.

Рис. 10-4. продолжение

```

// g_ah0bjs содержит список описателей объектов ядра "событие",
// на которые ссылается булево выражение
#define MAX_KERNEL_OBJS 1000
HANDLE g_ah0bjs[MAX_KERNEL_OBJS];

// ahExp0bjs содержит все выражения; каждое выражение состоит
// из набора описателей объектов ядра и становится TRUE, когда
// все эти объекты одновременно переходят в свободное состояние;
// чтобы отделить OR-выражения друг от друга, используем NULL-описатель

// один и тот же описатель может встречаться в AND-выражении только раз,
// но может повторяться в OR-выражениях

// выражение может содержать максимум 64 набора, а каждый набор –
// не более 63 описателей плюс NULL-описатель (разделитель наборов)
#define MAX_EXPRESSION_SIZE ((64 * 63) + 63)

// m_nExp0bjects – число элементов, задействованных в массиве ahExp0bjects
typedef struct {
    HWND m_hwnd; // куда посылать результаты
    DWORD m_dwMilliseconds; // сколько ждать
    DWORD m_nExp0bjects; // количество элементов
                        // в списке объектов
    HANDLE m_ahExp0bjs[MAX_EXPRESSION_SIZE]; // список объектов
} AWFME, *PAWFME;
AWFME g_awfme;

// это сообщение асинхронно посылается потоку пользовательского интерфейса,
// когда какое-то выражение становится истинным или заканчивается заданное
// время ожидания
#define WM_WAITEND (WM_USER + 101)

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

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

    chSETDLGICONS(hwnd, IDI_WFMETEXT);

    // инициализируем элементы управления в диалоговом окне
    SetDlgItemInt(hwnd, IDC_NUMOBJS, 4, FALSE);
    SetDlgItemInt(hwnd, IDC_TIMEOUT, 30000, FALSE);
    SetDlgItemText(hwnd, IDC_EXPRESSION,
        _T("1 2 | 2 3 4 | 1 4"));

    // устанавливаем размер колонки в окне многоколоночного списка
    ListBox_SetColumnWidth(GetDlgItem(hwnd, IDC_OBJLIST),
        LOWORD(GetDialogBaseUnits()) * 4);

    return(TRUE);
}

```

см. след. стр.

Рис. 10-4. *продолжение*

```

////////////////////////////////////
DWORD WINAPI AsyncWaitForMultipleExpressions(PVOID pvParam) {

    PAWFME pawfme = (PAWFME) pvParam;

    DWORD dw = WaitForMultipleExpressions(pawfme->m_nExp0bjects,
        pawfme->m_ahExp0bjs, pawfme->m_dwMilliseconds);
    PostMessage(pawfme->m_hwnd, WM_WAITEND, dw, 0);
    return(0);
}

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

LRESULT Dlg_OnWaitEnd(HWND hwnd, WPARAM wParam, LPARAM lParam) {

    // закрываем все описатели объектов ядра "событие"
    for (int n = 0; g_ah0bjs[n] != NULL; n++)
        CloseHandle(g_ah0bjs[n]);

    // сообщаем пользователю результат выполнения теста
    if (wParam == WAIT_TIMEOUT)
        SetDlgItemText(hwnd, IDC_RESULT, __TEXT("Timeout"));
    else
        SetDlgItemInt(hwnd, IDC_RESULT, (DWORD) wParam - WAIT_OBJECT_0, FALSE);

    // даем возможность изменить значения и повторно выполнить тест
    EnableWindow(GetDlgItem(hwnd, IDC_NUMOBS), TRUE);
    EnableWindow(GetDlgItem(hwnd, IDC_TIMEOUT), TRUE);
    EnableWindow(GetDlgItem(hwnd, IDC_EXPRESSION), TRUE);
    EnableWindow(GetDlgItem(hwnd, IDOK), TRUE);
    SetFocus(GetDlgItem(hwnd, IDC_EXPRESSION));

    return(0);
}

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

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    // получаем пользовательские настройки
    // из элементов управления диалогового окна
    TCHAR szExpression[100];
    ComboBox_GetText(GetDlgItem(hwnd, IDC_EXPRESSION), szExpression,
        sizeof(szExpression) / sizeof(szExpression[0]));

    int nObjects = GetDlgItemInt(hwnd, IDC_NUMOBS, NULL, FALSE);

    switch (id) {
    case IDCANCEL:
        EndDialog(hwnd, id);
        break;
    }
}

```

Рис. 10-4. продолжение

```

case IDC_OBJLIST:
    switch (codeNotify) {
    case LBN_SELCHANGE:
        // состояние элемента изменено, сбрасываем все элементы
        // и устанавливаем только выбранные
        for (int n = 0; n < nObjects; n++)
            ResetEvent(g_ahObjs[n]);

        for (n = 0; n < nObjects; n++) {
            if (ListBox_GetSel(GetDlgItem(hwnd, IDC_OBJLIST), n))
                SetEvent(g_ahObjs[n]);
        }
        break;
    }
    break;

case IDOK:
    // запрещаем изменение значений в процессе выполнения теста
    SetFocus(GetDlgItem(hwnd, IDC_OBJLIST));
    EnableWindow(GetDlgItem(hwnd, IDC_NUMOBS), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDC_TIMEOUT), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDC_EXPRESSION), FALSE);
    EnableWindow(GetDlgItem(hwnd, IDOK), FALSE);

    // уведомляем пользователя, что тест выполняется
    SetDlgItemText(hwnd, IDC_RESULT, TEXT("Waiting..."));

    // создаем все необходимые объекты ядра
    ZeroMemory(g_ahObjs, sizeof(g_ahObjs));
    g_awfme.m_nExpObjects = 0;
    ZeroMemory(g_awfme.m_ahExpObjs, sizeof(g_awfme.m_ahExpObjs));
    g_awfme.m_hwnd = hwnd;
    g_awfme.m_dwMilliseconds = GetDlgItemInt(hwnd, IDC_TIMEOUT, NULL, FALSE);

    ListBox_ResetContent(GetDlgItem(hwnd, IDC_OBJLIST));
    for (int n = 0; n < nObjects; n++) {
        TCHAR szBuf[20];
        g_ahObjs[n] = CreateEvent(NULL, FALSE, FALSE, NULL);

        wsprintf(szBuf, TEXT(" %d"), n + 1);
        ListBox_AddString(GetDlgItem(hwnd, IDC_OBJLIST),
            &szBuf[lstrlen(szBuf) - 3]);
    }

    PTSTR p = _tcstok(szExpression, TEXT(" "));
    while (p != NULL) {
        g_awfme.m_ahExpObjs[g_awfme.m_nExpObjects++] =
            (*p == TEXT('|')) ? NULL : g_ahObjs[_ttoi(p) - 1];
        p = _tcstok(NULL, TEXT(" "));
    }

```

см. след. стр.

Рис. 10-4. *продолжение*

```

        DWORD dwThreadId;
        CloseHandle(chBEGINTHREADEX(NULL, 0,
            AsyncWaitForMultipleExpressions, &g_awfme,
            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);

        case WM_WAITEND:
            return(Dlg_OnWaitEnd(hwnd, wParam, lParam));
    }

    return(FALSE);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_TESTW4ME), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конец файла //////////////////////////////////

```

Пулы потоков

В главе 8 мы обсудили синхронизацию потоков без перехода в режим ядра. Замечательная особенность такой синхронизации — высокое быстродействие. И если Вы озабочены быстродействием потока, сначала подумайте, нельзя ли обойтись синхронизацией в пользовательском режиме.

Вы уже знаете, что создание многопоточных приложений — дело трудное. Вас подстерегают две серьезные проблемы: управление созданием и уничтожением потоков и синхронизация их доступа к ресурсам. Для решения второй проблемы в Windows предусмотрено множество синхронизирующих примитивов: события, семафоры, мьютексы, критические секции и др. Все они довольно просты в использовании. Но если бы система автоматически охраняла разделяемые ресурсы, вот тогда создавать многопоточные приложения было бы по-настоящему легко. Увы, операционной системе Windows до этого еще далеко.

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 2000 появился ряд новых функций для операций с пулами потоков; эти функции упрощают создание, уничтожение и общий контроль за потоками. Конечно, встроенные в них механизмы 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 или при завершении запросов на асинхронный ввод-вывод.

Пул подразделяется на четыре компонента, которые описываются в таблице 11-1.

| | <i>таймера</i> | Компонент поддержки | | |
|-------------------------|--|--|---|------------------------|
| | | <i>ожидания</i> | <i>ввода-вывода</i> | <i>других операций</i> |
| Начальное число потоков | Всегда 1 | 1 | 0 | 0 |
| Когда поток создается | При вызове первой функции таймера пула потоков | Один поток для каждого из 63 зарегистрированных объектов | В системе применяются эвристические методы, но на создание потока влияют следующие факторы: <ul style="list-style-type: none">■ после добавления потока прошло определенное время■ установлен флаг WT_EXECUTEONLONGFUNCTION■ число элементов в очереди превышает пороговое значение | |

Таблица 11-1. Компоненты поддержки пула потоков

см. след. стр.

| | | Компонент поддержки | | |
|--------------------------|---|---|--|---|
| | таймера | ожидания | ввода-вывода | других операций |
| Когда поток разрушается | При завершении процесса | При отсутствии зарегистрированных объектов ожидания | При отсутствии у потока текущих запросов на ввод-вывод и простое в течение определенного порогового времени (около минуты) | При простое потока в течение определенного порогового времени (около минуты) |
| Как поток ждет | В «тревожном» состоянии | <i>WaitForMultipleObjectsEx</i> | В «тревожном» состоянии | <i>GetQueuedCompletionStatus</i> |
| Когда поток пробуждается | При освобождении «ожидаемого таймера», который посылает в очередь APC-вызов | При освобождении объекта ядра | При посылке в очередь APC-вызова или завершении запроса на ввод-вывод | При поступлении запроса о статусе завершения или о завершении ввода-вывода (порт завершения требует, чтобы число потоков не превышало число процессоров более чем в 2 раза) |

При инициализации процесса никаких издержек, связанных с перечисленными в таблице компонентами поддержки, 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 малы: частью Вашего процесса становится целый набор потоков и внутренних структур данных. Так что, прежде чем пользоваться ими, тщательно взвесьте все «за» и «против».

О'кэй, теперь, когда я Вас предупредил, посмотрим, как все это работает.

Сценарий 1: асинхронный вызов функций

Допустим, у Вас есть серверный процесс с основным потоком, который ждет клиентский запрос. Получив его, он порождает отдельный поток для обработки этого запроса. Тем самым основной поток освобождается для приема следующего клиентского запроса. Такой сценарий типичен в клиент-серверных приложениях. Хотя он и так-то незатейлив, при желании его можно реализовать с использованием новых функций пула потоков.

Получая клиентский запрос, основной поток вызывает:

```
BOOL QueueUserWorkItem(  
    PTHREAD_START_ROUTINE pfnCallback,  
    PVOID pvContext,  
    ULONG dwFlags);
```

Эта функция помещает «рабочий элемент» (work item) в очередь потока в пуле и тут же возвращает управление. Рабочий элемент — это просто вызов функции (на которую ссылается параметр *pfnCallback*), принимающей единственный параметр, *pvContext*. В конечном счете какой-то поток из пула займется обработкой этого эле-

мента, в результате чего будет вызвана Ваша функция. У этой функции обратного вызова, за реализацию которой отвечаете Вы, должен быть следующий прототип:

```
DWORD WINAPI WorkItemFunc(PVOID pvContext);
```

Несмотря на то что тип возвращаемого значения определен как `DWORD`, на самом деле оно игнорируется.

Обратите внимание, что Вы сами никогда не вызываете *CreateThread*. Она вызывается из пула потоков, автоматически создаваемого для Вашего процесса, а к функции *WorkItemFunc* обращается один из потоков этого пула. Кроме того, данный поток не уничтожается сразу после обработки клиентского запроса, а возвращается в пул, оставаясь готовым к обработке любых других элементов, помещаемых в очередь. Ваше приложение может стать гораздо эффективнее, так как Вам больше не придется создавать и уничтожать потоки для каждого клиентского запроса. А поскольку потоки связаны с определенным портом завершения, количество одновременно работающих потоков не может превышать число процессоров более чем в 2 раза. За счет этого переключения контекста происходят реже.

Многое в пуле потоков происходит скрытно от разработчика: *QueueUserWorkItem* проверяет число потоков, включенных в сферу ответственности компонента поддержки других операций (не относящихся к вводу-выводу), и в зависимости от текущей нагрузки (количества рабочих элементов в очереди) может передать ему другие потоки. После этого *QueueUserWorkItem* выполняет операции, эквивалентные вызову *PostQueuedCompletionStatus*, пересылая информацию о рабочем элементе в порт завершения ввода-вывода. В конечном счете поток, ждущий на этом объекте, извлекает Ваше сообщение (вызовом *GetQueuedCompletionStatus*) и обращается к Вашей функции. После того как она возвращает управление, поток вновь вызывает *GetQueuedCompletionStatus*, ожидая появления следующего рабочего элемента.

Пул рассчитан на частую обработку асинхронного ввода-вывода — всякий раз, когда поток помещает в очередь запрос на ввод-вывод к драйверу устройства. Пока драйвер выполняет его, поток, поставивший запрос в очередь, не блокируется и может заниматься другой работой. Асинхронный ввод-вывод — ключ к созданию высокоэффективных, масштабируемых приложений, так как позволяет одному потоку обрабатывать запросы от множества клиентов по мере их поступления; ему не приходится обрабатывать их последовательно или останавливаться, ожидая завершения ввода-вывода.

Но Windows накладывает одно ограничение на запросы асинхронного ввода-вывода: если поток, послав такой запрос драйверу устройства, завершается, данный запрос теряется и никакие потоки о его судьбе не уведомляются. В хорошо продуманном пуле, число потоков увеличивается и уменьшается в зависимости от потребностей его клиентов. Поэтому, если поток посылает запрос и уничтожается из-за сокращения пула, то уничтожается и этот запрос. Как правило, это не совсем то, что хотелось бы, и здесь нужно найти какое-то решение.

Если Вы хотите поместить в очередь рабочий элемент, который выдает запрос на асинхронный ввод-вывод, то не сможете передать этот элемент компоненту поддержки других операций в пуле потоков. Его примет лишь компонент поддержки ввода-вывода. Последний включает набор потоков, которые не завершаются, пока есть хотя бы один запрос на ввод-вывод; поэтому для выполнения кода, выдающего запросы на асинхронный ввод-вывод, Вы должны пользоваться только этими потоками.

Чтобы передать рабочий элемент компоненту поддержки ввода-вывода, Вы можете по-прежнему пользоваться функцией *QueueUserWorkItem*, но в параметре *dwFlags*

следует указать флаг `WT_EXECUTEINIOTHREAD`. А обычно Вы будете указывать в этом параметре флаг `WT_EXECUTEDEFAULT` (0) — он заставляет систему передать рабочий элемент компоненту поддержки других операций (не связанных с вводом-выводом).

В Windows есть функции (вроде *RegNotifyChangeKeyValue*), которые асинхронно выполняют операции, не относящиеся к вводу-выводу. Они также требуют, чтобы вызывающий поток не завершался преждевременно. С этой целью Вы можете использовать флаг `WT_EXECUTEINPERSISTENTTHREAD`, который заставляет поток таймера выполнять поставленную в очередь функцию обратного вызова для рабочего элемента. Так как этот компонент существует постоянно, асинхронная операция в конечном счете обязательно будет выполнена. Вы должны позаботиться о том, чтобы функция обратного вызова выполнялась быстро и не блокировала работу компонента поддержки таймера.

Хорошо продуманный пул должен также обеспечивать максимальную готовность потоков к обработке запросов. Если в пуле четыре потока, а в очереди сто рабочих элементов, то одновременно можно обработать только четыре элемента. Это не проблема, если на обработку каждого элемента уходит лишь несколько миллисекунд, но в ином случае программа не сумеет своевременно обслуживать запросы.

Естественно, система не настолько умна, чтобы предвидеть, чем будет заниматься функция Вашего рабочего элемента, но если Вам заранее известно, что на это уйдет длительное время, вызовите *QueueUserWorkItem* с флагом `WT_EXECUTEINLONGFUNCTION` — он заставит пул создать новый поток, если остальные потоки будут в это время заняты. Так, добавив в очередь 10 000 рабочих элементов (с флагом `WT_EXECUTEINLONGFUNCTION`), Вы получите 10 000 новых потоков в пуле. Чтобы избежать этого, делайте перерывы между вызовами *QueueUserWorkItem*, и тогда часть потоков успеет завершиться до порождения новых.

Ограничение на количество потоков в пуле накладывать нельзя, иначе может возникнуть взаимная блокировка потоков. Представьте очередь из 10 000 элементов, заблокированных 10 001-м и ждущих его освобождения. Установив предел в 10 000, Вы запретите выполнение 10 001-го потока, и в результате целых 10 000 потоков останутся навечно заблокированными.

Используя функции пула, будьте осторожны, чтобы не доводить дело до тупиковых ситуаций. Особую осторожность проявляйте, если функция Вашего рабочего элемента использует критические секции, семафоры, мьютексы и др. — это увеличивает вероятность взаимной блокировки. Вы должны всегда точно знать, поток какого компонента пула выполняет Ваш код. Также будьте внимательны, если функция рабочего элемента содержится в DLL, которая может быть динамически выгружена из памяти. Поток, вызывающий функцию из выгруженной DLL, приведет к нарушению доступа. Чтобы предотвратить выгрузку DLL при наличии рабочих элементов в очереди, создайте контрольный счетчик для таких элементов: его значение должно увеличиваться перед вызовом *QueueUserWorkItem* и уменьшаться после выполнения функции рабочего элемента. Выгрузка DLL допустима только после того, как этот счетчик обнулится.

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 какой-то рабочий элемент выполнялся через определенные интервалы времени, первым делом создайте очередь таймеров, вызвав функцию:

```
HANDLE CreateTimerQueue();
```

Очередь таймеров обеспечивает организацию набора таймеров. Представьте, что один исполняемый файл предоставляет несколько сервисов. Каждый сервис может потребовать создания таймеров, скажем, для определения того, какой клиент перестал отвечать, для сбора и обновления некоей статистической информации по расписанию и т. д. Выделять каждому сервису ожидаемый таймер и отдельный поток крайне неэффективно. Вместо этого у каждого сервиса должна быть своя очередь таймеров (занимающая минимум системных ресурсов), а поток компонента поддержки таймера и объект ядра «ожидаемый таймер» должны разделяться всеми сервисами. По окончании работы сервиса его очередь вместе со всеми созданными в ней таймерами просто удаляется.

Вы можете создавать таймеры в очереди, вызывая функцию:

```
BOOL CreateTimerQueueTimer(
    PHANDLE phNewTimer,
    HANDLE hTimerQueue,
    WAITORTIMERCALLBACK pfnCallback,
    PVOID pvContext,
    DWORD dwDueTime,
    DWORD dwPeriod,
    ULONG dwFlags);
```

Во втором параметре Вы передаете описатель очереди, в которую нужно поместить новый таймер. Если таймеров немного, в этом параметре можно передать NULL и вообще не вызывать *CreateTimerQueue*. Такое значение параметра заставит функцию *CreateTimerQueueTimer* использовать очередь по умолчанию и упростит программирование. Параметры *pfnCallback* и *pvContext* указывают на Вашу функцию обратного вызова и данные, передаваемые ей в момент срабатывания таймера. Параметр *dwDueTime* задает время первого срабатывания, а *dwPeriod* — время последующих срабатываний. (Передача в *dwDueTime* нулевого значения заставляет систему вызвать Вашу функцию по возможности немедленно, что делает функцию *CreateTimerQueueTimer* похожей на *QueueUserWorkItem*.) Если *dwPeriod* равен 0, таймер сработает лишь раз, и рабочий элемент будет помещен в очередь только единожды. Описатель нового таймера возвращается в параметре *phNewTimer*.

Прототип Вашей функции обратного вызова должен выглядеть так:

```
VOID WINAPI WaitOrTimerCallback(
    PVOID pvContext,
    BOOL fTimerOrWaitFired);
```

Когда она вызывается, параметр *fTimerOrWaitFired* всегда принимает значение TRUE, сообщая тем самым, что таймер сработал.

Теперь поговорим о параметре *dwFlags* функции *CreateTimerQueueTimer*. Он сообщает функции, как обрабатывать рабочий элемент, помещаемый в очередь. Вы можете указать флаг WT_EXECUTEDefault, если хотите, чтобы рабочий элемент был обработан одним из потоков пула, контролируемых компонентом поддержки других операций, WT_EXECUTEINIOThread — если в определенный момент нужно выдать

асинхронный запрос на ввод-вывод, или `WT_EXECUTEINPERSISTENTTHREAD` — если элементом должен заняться один из постоянных потоков. Для рабочего элемента, требующего длительного времени обработки, следует задать флаг `WT_EXECUTELONGFUNCTION`.

Вы можете пользоваться еще одним флагом, `WT_EXECUTEINTIMERTHREAD`, который нуждается в более подробном объяснении. Как видно из таблицы 11-1, пул потоков включает компонент поддержки таймера. Этот компонент создает единственный объект ядра «ожидаемый таймер», управляя временем его срабатывания, и всегда состоит из одного потока. Вызывая *CreateTimerQueueTimer*, Вы заставляете его пробудиться, добавить Ваш таймер в очередь и перенастроить объект ядра «ожидаемый таймер». После этого поток компонента поддержки таймера переходит в режим «тревожного» ожидания APC-вызова от таймера. Обнаружив APC-вызов в своей очереди, поток пробуждается, обновляет очередь таймеров, перенастраивает объект ядра «ожидаемый таймер», а затем решает, что делать с рабочим элементом, который теперь следует обработать.

Далее поток проверяет наличие следующих флагов: `WT_EXECUTEDEFAULT`, `WT_EXECUTEINIOPTHREAD`, `WT_EXECUTEINPERSISTENTTHREAD`, `WT_EXECUTELONGFUNCTION` и `WT_EXECUTEINTIMERTHREAD`. И сейчас Вы, наверное, поняли, что делает флаг `WT_EXECUTEINTIMERTHREAD`: он заставляет поток компонента поддержки таймера обработать рабочий элемент. Хотя такой механизм обработки элемента более эффективен, он очень опасен! Пока выполняется функция рабочего элемента, поток компонента поддержки таймера ничем другим заниматься не может. Ожидаемый таймер будет по-прежнему ставить APC-вызовы в соответствующую очередь потока, но эти рабочие элементы не удастся обработать до завершения текущей функции. Так что, поток компонента поддержки таймера годится для выполнения лишь «быстрого» кода, не блокирующего этот ресурс надолго.

Флаги `WT_EXECUTEINIOPTHREAD`, `WT_EXECUTEINPERSISTENTTHREAD` и `WT_EXECUTEINTIMERTHREAD` являются взаимоисключающими. Если Вы не передаете ни один из этих флагов (или используете `WT_EXECUTEDEFAULT`), рабочий элемент помещается в очередь одного из потоков в компоненте поддержки других операций (не связанных с вводом-выводом). Кроме того, `WT_EXECUTELONGFUNCTION` игнорируется, если задан флаг `WT_EXECUTEINTIMERTHREAD`.

Ненужный таймер удаляется с помощью функции:

```
BOOL DeleteTimerQueueTimer(
    HANDLE hTimerQueue,
    HANDLE hTimer,
    HANDLE hCompletionEvent);
```

Вы должны вызывать ее даже для «одноразовых» таймеров, если они уже сработали. Параметр *hTimerQueue* указывает очередь, в которой находится таймер, а *hTimer* — удаляемый таймер; последний описатель возвращается *CreateTimerQueueTimer* при создании таймера.

Последний параметр, *hCompletionEvent*, определяет, каким образом Вас следует уведомлять об отсутствии необработанных рабочих элементов, поставленных в очередь этим таймером. Если в нем передать `INVALID_HANDLE_VALUE`, функция *DeleteTimerQueueTimer* вернет управление только после обработки всех поставленных в очередь элементов. Задумайтесь, что это значит: удалив таймер в процессе обработки запущенного им рабочего элемента, Вы создаете тупиковую ситуацию, так? Вы ждете окончания его обработки и сами же прерываете ее! Вот почему поток может

удалить таймер, только если это не он обрабатывает рабочий элемент, поставленный в очередь данным таймером.

Кроме того, используя поток компонента поддержки таймера, никогда не удаляйте какой-либо из таймеров во избежание взаимной блокировки. Попытка удалить таймер приводит к тому, что в очередь этого потока помещается APC-уведомление. Но если поток ждет удаления таймера, то сам удалить его он уже не в состоянии — вот и тупик.

Вместо значения `INVALID_HANDLE_VALUE` в параметре *hCompletionEvent* можно передать `NULL`. Это подскажет функции, что таймер следует удалить — и чем раньше, тем лучше. В таком случае *DeleteTimerQueueTimer* немедленно вернет управление, но Вы не узнаете, когда будут обработаны все элементы, поставленные в очередь этим таймером. И, наконец, в параметре *hCompletionEvent* можно передать описатель объекта ядра «событие». Тогда *DeleteTimerQueueTimer* немедленно вернет управление, а поток компонента поддержки таймера освободит событие, как только будут обработаны все элементы из очереди. Но прежде чем вызывать *DeleteTimerQueueTimer*, Вы должны позаботиться о том, чтобы это событие находилось в занятом состоянии, иначе Ваша программа ошибочно решит, что все элементы уже обработаны.

Вы можете изменять время первого и последующих срабатываний существующего таймера, используя функцию:

```
BOOL ChangeTimerQueueTimer(
    HANDLE hTimerQueue,
    HANDLE hTimer,
    ULONG dwDueTime,
    ULONG dwPeriod);
```

Ей передаются описатели очереди и самого таймера, который надо перенастроить, а также параметры *dwDueTime* и *dwPeriod* (время срабатывания и периодичность). Учтите: эта функция не влияет на уже сработавший «одноразовый» таймер. Вы можете применять ее совершенно свободно, без всяких опасений насчет тупиковых ситуаций.

Для удаления очереди таймеров предназначена функция:

```
BOOL DeleteTimerQueueEx(
    HANDLE hTimerQueue,
    HANDLE hCompletionEvent);
```

Она принимает описатель существующей очереди и удаляет все таймеры в ней, избавляя от необходимости вызова *DeleteTimerQueueTimer* для каждого таймера по отдельности. Параметр *hCompletionEvent* идентичен такому же параметру функции *DeleteTimerQueueTimer*, а это значит, что, как и в предыдущем случае, Вы должны помнить о возможности тупиковых ситуаций, — будьте осторожны.

Прежде чем рассматривать следующий вариант, позвольте обратить Ваше внимание на несколько нюансов. Компонент поддержки таймера создает объект ядра «ожидаемый таймер», и тот посылает APC-вызовы в очередь, а не переходит в свободное состояние. Иначе говоря, операционная система постоянно ставит APC-вызовы в очередь, и события таймера никогда не теряются. Такой механизм гарантирует, что написанная Вами функция обратного вызова будет срабатывать с заданной периодичностью. Только имейте в виду, что все это происходит с использованием множества потоков, а значит, какие-то части этой функции, видимо, потребуют синхронизации.

Если Вас это не устраивает и Вы хотите, чтобы новый вызов помещался в очередь, скажем, через 10 секунд после завершения обработки предыдущего, создавайте в конце функции рабочего элемента однократно срабатывающие таймеры. Или единствен-

ный таймер, но с длительным временем ожидания, а в конце все той же функции вызывайте *ChangeTimerQueueTimer* для перенастройки таймера.

Программа-пример TimedMsgBox

Эта программа, «11-TimedMsgBox.exe» (см. листинг на рис. 11-1), 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 на компакт-диске, прилагаемом к книге.

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 *CreateTimerQueueTimer*, настраивающая 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:



Десятый вызов *MsgBoxTimeout* обнуляет *g_nSecLeft*, и тогда *MsgBoxTimeout* вызывает *EndDialog*, чтобы закрыть окно. После этого функция *MessageBox*, вызванная первичным потоком, возвращает управление, и вызывается *DeleteTimerQueueTimer*, 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.





TimedMsgBox.cpp

```

/*****
Модуль: TimedMsgBox.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <tchar.h>

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

// заголовок нашего окна
TCHAR g_szCaption[] = TEXT("Timed Message Box");

// сколько секунд мы будем показывать это окно
int g_nSecLeft = 0;

// это статический управляющий идентификатор нашего окна
#define ID_MSGBOX_STATIC_TEXT  0x0000ffff

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

VOID WINAPI MsgBoxTimeout(PVOID pvContext, BOOLEAN fTimeout) {

    // Примечание: из-за конкуренции потоков возможно (но маловероятно),
    // что окно еще не будет создано, когда мы попадем сюда.
    HWND hwnd = FindWindow(NULL, g_szCaption);

    if (hwnd != NULL) {
        // окно уже существует; сообщаем, сколько времени осталось
        TCHAR sz[100];
        wsprintf(sz, TEXT("You have %d seconds to respond"), g_nSecLeft--);
        SetDlgItemText(hwnd, ID_MSGBOX_STATIC_TEXT, sz);

        if (g_nSecLeft == 0) {
            // время истекло; закрываем окно
            EndDialog(hwnd, IDOK);
        }
    } else {
        // окна пока нет; сейчас ничего не делаем,
        // попробуем через секунду
    }
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    chWindows9xNotAllowed();
}

```

Рис. 11-1. Программа-пример *TimedMsgBox*

см. след. стр.

Рис. 11-1. *продолжение*

```
// сколько секунд мы даем на ответ
g_nSecLeft = 10;

// создаем многократно срабатывающий таймер
// (первое срабатывание через 1 секунду)
HANDLE hTimerQTimer;
CreateTimerQueueTimer(&hTimerQTimer, NULL, MsgBoxTimeout, NULL,
    1000, 1000, 0);

// выводим окно
MessageBox(NULL, TEXT("You have 10 seconds to respond"), g_szCaption, MB_OK);

// закрываем таймер и удаляем его очередь
DeleteTimerQueueTimer(NULL, hTimerQTimer, NULL);

// выясняем, ответил пользователь или нет
MessageBox(NULL,
    (g_nSecLeft == 0) ? TEXT("Timeout") : TEXT("User responded"),
    TEXT("Result"), MB_OK);

return(0);
}

/////////////////////// Конец файла /////////////////////////
```

Сценарий 3: вызов функций при освобождении отдельных объектов ядра

Microsoft 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:

```
BOOL RegisterWaitForSingleObject(
    PHANDLE phNewWaitObject,
    HANDLE hObject,
    WAITORTIMERCALLBACK pfnCallback,
    PVOID pvContext,
    ULONG dwMilliseconds,
    ULONG dwFlags);
```

Эта функция передает Ваши параметры компоненту поддержки ожидания в пуле потоков. Вы сообщаете ему, что рабочий элемент надо поставить в очередь, как толь-

ко освободится объект ядра (на который указывает *bObject*). Кроме того, Вы можете задать ограничение по времени, т. е. элемент будет помещен в очередь через определенное время, даже если объект ядра так и не освободится. (При этом допустимы значения INFINITE и 0.) В общем, эта функция похожа на хорошо известную функцию *WaitForSingleObject* (см. главу 9). Зарегистрировав рабочий элемент на ожидание указанного объекта, *RegisterWaitForSingleObject* возвращает в параметре *phNewWaitObject* описатель, идентифицирующий объект ожидания.

Данный компонент реализует ожидание зарегистрированных объектов через *WaitForMultipleObjects* и поэтому накладывает те же ограничения, что и эта функция. Одно из них заключается в том, что нельзя ожидать тот же объект несколько раз. Так что придется вызывать *DuplicateHandle* и отдельно регистрировать исходный и продублированный описатель. Вам должно быть известно, что одновременно функция *WaitForMultipleObjects* способна отслеживать не более 64 (MAXIMUM_WAIT_OBJECTS) объектов. А что будет, если попробовать зарегистрировать с ее помощью более 64 объектов? Компонент поддержки ожидания создаст еще один поток, который тоже вызовет *WaitForMultipleObjects*. (На самом деле новый поток создается на каждые 63 объекта, потому что потокам приходится использовать объект ядра «ожидаемый таймер», контролирующий таймауты.)

По умолчанию рабочий элемент, готовый к обработке, помещается в очередь к потокам компонента поддержки других операций (не связанных с вводом-выводом). В конечном счете один из его потоков пробудится и вызовет Вашу функцию, у которой должен быть следующий прототип:

```
VOID WINAPI WaitOrTimerCallbackFunc(
    PVOID pvContext,
    BOOLEAN fTimerOrWaitFired);
```

Параметр *fTimerOrWaitFired* принимает значение TRUE, если время ожидания истекло, или FALSE, если объект освободился раньше.

В параметре *dwFlags* функции *RegisterWaitForSingleObject* можно передать флаг WT_EXECUTEINWAITTHREAD, который заставляет выполнить функцию рабочего элемента в одном из потоков компонента поддержки ожидания. Это эффективнее, потому что тогда рабочий элемент не придется ставить в очередь компонента поддержки других операций. Но в то же время и опаснее, так как этот поток не сможет ждать освобождения других объектов. Используйте этот флаг, только если Ваша функция выполняется быстро.

Вы можете также передать флаг WT_EXECUTEINIOTHREAD, если Ваш рабочий элемент выдаст запрос на асинхронный ввод-вывод, или WT_EXECUTEINPERSISTENTTHREAD, если ему понадобится операция с использованием постоянно существующего потока. В случае длительного выполнения функции рабочего элемента можно применить флаг WT_EXECUTELONGFUNCTION. Указывайте этот флаг, только если рабочий элемент передается компоненту поддержки ввода-вывода или других операций, — функцию, требующую продолжительной обработки, нельзя выполнять в потоке, который относится к компоненту поддержки ожидания.

И последний флаг, о котором Вы должны знать, — WT_EXECUTEONCE. Допустим, Вы зарегистрировались на ожидание объекта ядра «процесс». После перехода в свободное состояние он так и останется в этом состоянии, что заставит компонент поддержки ожидания постоянно включать в очередь рабочие элементы. Так вот, чтобы избежать этого, Вы можете использовать флаг WT_EXECUTEONCE — он сообщает пулу потоков прекратить ожидание объекта после первой обработки рабочего элемента.

Теперь представьте, что Вы ждете объект ядра «событие с автосбросом»: сразу после освобождения он тут же возвращается в занятое состояние; при этом в очередь ставится соответствующий рабочий элемент. На этом этапе пул продолжает отслеживать объект и снова ждет его освобождения или того момента, когда истечет время, выделенное на ожидание. Если состояние объекта Вас больше не интересует, Вы должны снять его с регистрации. Это необходимо даже для отработавших объектов, зарегистрированных с флагом `WT_EXECUTEONCE`. Вот как выглядит требуемая для этого функция:

```
BOOL UnregisterWaitEx(
    HANDLE hWaitHandle,
    HANDLE hCompletionEvent);
```

Первый параметр указывает на объект ожидания (его описатель возвращается *RegisterWaitForSingleObject*), а второй определяет, каким образом Вас следует уведомлять о выполнении последнего элемента в очереди. Как и в *DeleteTimerQueueTimer*, Вы можете передать в этом параметре `NULL` (если уведомление Вас не интересует), `INVALID_HANDLE_VALUE` (функция блокируется до завершения обработки всех элементов в очереди) или описатель объекта-события (переходящего в свободное состояние при завершении обработки очередного элемента). В ответ на неблокирующий вызов *UnregisterWaitEx* возвращает `TRUE`, если очередь пуста, и `FALSE` в ином случае (при этом *GetLastError* возвращает `STATUS_PENDING`).

И вновь будьте осторожны, передавая значение `INVALID_HANDLE_VALUE`. Функция рабочего элемента заблокирует сама себя, если попытается снять с регистрации вызвавший ее объект ожидания. Такая попытка подобна команде: приостановить меня, пока я не закончу выполнение, — полный тупик. Но *UnregisterWaitEx* разработана так, чтобы предотвращать тупиковые ситуации, когда поток компонента поддержки ожидания выполняет рабочий элемент, а тот пытается снять с регистрации запустивший его объект ожидания. И еще один момент: не закрывайте описатель объекта ядра до тех пор, пока не снимете его с регистрации. Иначе недействительный описатель попадет в *WaitForMultipleObjects*, к которой обращается поток компонента поддержки ожидания. Функция моментально завершится с ошибкой, и этот компонент перестанет корректно работать.

И последнее: никогда не вызывайте *PulseEvent* для освобождения объекта-события, зарегистрированного на ожидание. Поток компонента поддержки ожидания скорее всего будет чем-то занят и пропустит этот импульс от *PulseEvent*. Но эта проблема для Вас не нова — *PulseEvent* создает ее почти во всех архитектурах поддержки потоков.

Сценарий 4: вызов функций по завершении запросов на асинхронный ввод-вывод

Последний сценарий самый распространенный. Ваше серверное приложение выдает запросы на асинхронный ввод-вывод, и Вам нужен пул потоков, готовых к их обработке. Это как раз тот случай, на который и были изначально рассчитаны порты завершения ввода-вывода. Если бы Вы управляли собственным пулом потоков, Вы создали бы порт завершения ввода-вывода и пул потоков, ждущих на этом порте. Кроме того, Вы открыли бы пару-тройку устройств ввода-вывода и связали бы их описатели с портом. По мере завершения асинхронных запросов на ввод-вывод, драйверы устройств помещали бы «рабочие элементы» в очередь порта завершения.

Это прекрасная архитектура, позволяющая небольшому количеству потоков эффективно обрабатывать несколько рабочих элементов, и очень хорошо, что она за-

ложена в функции пула потоков. Благодаря этому Вы сэкономите уйму времени и сил. Для использования преимуществ данной архитектуры надо лишь открыть требуемое устройство и сопоставить его с компонентом поддержки других операций (не связанных с вводом-выводом). Учтите, что все потоки в этом компоненте ждут на порте завершения. Чтобы сопоставить устройство с компонентом поддержки других операций, вызовите функцию:

```
BOOL BindIoCompletionCallback(
    HANDLE hDevice,
    POVERLAPPED_COMPLETION_ROUTINE pfnCallback,
    ULONG dwFlags);
```

Эта функция обращается к *CreateIoCompletionPort*, передавая ей *hDevice* и описатель внутреннего порта завершения. Ее вызов также гарантирует, что в компоненте поддержки других операций есть хотя бы один поток. Ключ завершения, сопоставленный с устройством, — это адрес перекрывающейся подпрограммы завершения. Так что, когда ввод-вывод на устройство завершается, компонент пула уже знает, какую функцию надо вызвать для обработки завершеного запроса. У подпрограммы завершения должен быть следующий прототип:

```
VOID WINAPI OverlappedCompletionRoutine(
    DWORD dwErrorCode,
    DWORD dwNumberOfBytesTransferred,
    POVERLAPPED pOverlapped);
```

Заметьте: структура *OVERLAPPED* передается не в *BindIoCompletionCallback*, а в функции типа *ReadFile* и *WriteFile*. Система внутренне отслеживает эту структуру вместе с запросом на ввод-вывод. После его завершения система поместит адрес структуры в порт завершения для последующей передачи Вашей *OverlappedCompletionRoutine*. А поскольку адрес подпрограммы завершения — это и ключ завершения, то для передачи дополнительной контекстной информации в *OverlappedCompletionRoutine* Вы должны прибегнуть к традиционному трюку и разместить эту информацию в конце структуры *OVERLAPPED*.

Также учтите, что закрытие устройства приводит к немедленному завершению всех текущих запросов на ввод-вывод и дает ошибку. Будьте готовы к этому в своей функции обратного вызова. Если Вы хотите, чтобы после закрытия устройства функции обратного вызова больше не выполнялись, создайте в своем приложении контрольный счетчик. При выдаче запроса на ввод-вывод Вы будете увеличивать его значение на 1, а при завершении — соответственно уменьшать.

Каких-то специальных флагов для функции *BindIoCompletionCallback* сейчас не предусматривается, поэтому Вы должны передавать 0 в параметре *dwFlags*. Но, по моему, один флаг, *WT_EXECUTEINIOTHREAD*, ей следовало бы поддерживать. После завершения запроса на ввод-вывод он заставил бы поместить этот запрос в очередь одного из потоков компонента поддержки других операций (не связанных с вводом-выводом). Ведь *OverlappedCompletionRoutine*, вероятно, выдаст еще один запрос на асинхронный ввод-вывод. Однако, если поток завершается, все выданные им запросы на ввод-вывод автоматически уничтожаются. Кроме того, надо учесть, что потоки в компоненте поддержки других операций создаются и уничтожаются в зависимости от текущей нагрузки. При низкой нагрузке поток может быть закрыт, оставив незавершенные запросы. Если бы функция *BindIoCompletionCallback* поддерживала флаг *WT_EXECUTEINIOTHREAD*, то поток, ждущий на порте завершения, мог бы пробудиться и передать результат потоку компонента поддержки ввода-вывода. И поскольку эти

потоки никогда не завершаются при наличии запросов, Вы могли бы выдавать такие запросы, не опасаясь потерять их.

Флаг `WT_EXECUTEINIOTHREAD` был бы, конечно, очень удобен, но Вы можете легко эмулировать все то, о чем я сейчас говорил. В своей функции *OverlappedCompletionRoutine* просто вызовите *QueueUserWorkItem* с флагом `WT_EXECUTEINIOTHREAD` и передайте нужные данные (наверное, как минимум, структуру `OVERLAPPED`). Ничего другого функции пула Вам и не предложили бы.

Волокна

Microsoft 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, Microsoft добавила 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 в Windows реализуются на уровне ядра операционной системы, которое отлично осведомлено об их существовании и «коммутирует» их в соответствии с созданным Microsoft 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 Вами. 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 поток — все то, что можно вытеснить и что выполняет код. Единственно 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*:

```
PVOID 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 исключений;
- начальный и конечный адреса стека волокна; при преобразовании потока в волокно он служит и стеком потока;
- регистры процессора, включая указатели стека и команд.

Создав и инициализировав контекст волокна, Вы сопоставляете его адрес с потоком, преобразованным в волокно, и теперь оно выполняется в этом потоке. *ConvertThreadToFiber* 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 поток.

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(
    DWORD 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.

Создав стек, *CreateFiber* формирует новую структуру контекста волокна и инициализирует ее. При этом первый ее элемент получает значение, переданное функции как параметр *pvParam*, сохраняются начальный и конечный адреса стека, а также адрес функции волокна (переданный как аргумент *pfnStartAddress*).

У функции волокна, реализуемой Вами, должен быть такой прототип:

```
VOID WINAPI FiberFunc(PVOID pvParam);
```

Она выполняется, когда волокно впервые получает управление. В качестве параметра ей передается значение, изначально переданное как аргумент *pvParam* функции *CreateFiber*. Внутри функции волокна можно делать что угодно. Обратите внимание на тип возвращаемого значения — VOID. Причина не в том, что это значение несущественно, а в том, что функция никогда не возвращает управление! А иначе поток и все созданные внутри него волокна были бы тут же уничтожены.

Как и *ConvertThreadToFiber*, *CreateFiber* возвращает адрес контекста волокна, но с тем отличием, что новое волокно начинает работать не сразу, поскольку продолжается выполнение текущего. Единновременно поток может выполнять лишь одно волокно. И, чтобы новое волокно стало работать, надо вызвать *SwitchToFiber*:

```
VOID SwitchToFiber(PVOID pvFiberExecutionContext);
```

Эта функция принимает единственный параметр (*pvFiberExecutionContext*) — адрес контекста волокна, полученный в предшествующем вызове *ConvertThreadToFiber* или *CreateFiber*. 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 *SwitchToFiber* — 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, пока Вы сами не вызовете *SwitchToFiber*.

Для уничтожения волокна предназначена функция *DeleteFiber*:

```
VOID DeleteFiber(PVOID pvFiberExecutionContext);
```

Она удаляет волокно, чей адрес контекста определяется параметром *pvFiberExecutionContext*, освобождает память, занятую стеком волокна, и уничтожает его контекст. Но, если Вы передадите 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.

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 — *GetFiberData*:

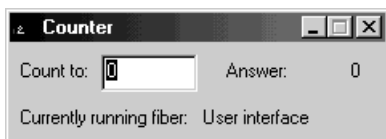
```
PVOID GetFiberData();
```

Как я уже говорил, контекст каждого волокна содержит определяемое программистом значение. Оно инициализируется значением параметра *pvParam*, передаваемого функции *ConvertThreadToFiber* или *CreateFiber*, и служит аргументом функции волокна. *GetFiberData* просто «заглядывает» в контекст текущего волокна и возвращает хранящееся там значение.

Обе функции — *GetCurrentFiber* 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» (см. листинг на рис. 12-1), 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 ней и как она себя ведет.)



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 наблюдайте 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 *ConvertTbthreadToFiber*, 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.

| 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_CONTINUE | 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 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 управление, пока волокно пересчета тоже не обратится к *SwitchToFiber*, передав 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 по-

явились 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 *SwitchToFiber*) интерфейсному волокну. В этот момент ему делать нечего, и оно вызывает *WaitMessage*, которая приостанавливает поток, чтобы не тратить процессорное время понапрасну.



Counter.cpp

```

/*****
Модуль: Counter.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"

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

// возможные состояния фоновой обработки
typedef enum {
    BPS_STARTOVER,    // начинаем фоновую обработку заново
    BPS_CONTINUE,     // продолжаем фоновую обработку
    BPS_DONE          // фоновая обработка не нужна
} BKGNDPROCSTATE;

typedef struct {
    PVOID pFiberUI;    // контекст интерфейсного волокна
    HWND hwnd;         // описатель главного окна
    BKGNDPROCSTATE bps; // состояние фоновой обработки
} FIBERINFO, *PFIBERINFO;

// глобальная переменная, содержащая информацию о состоянии приложения;
// из интерфейсного волокна она доступна напрямую, а из волокна, отвечающего
// за фоновую обработку, – косвенно
FIBERINFO g_FiberInfo;

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

```

Рис. 12-1. Программа-пример Counter

Рис. 12-1. продолжение

```

void WINAPI FiberFunc(PVOID pvParam) {
    PFIBERINFO pFiberInfo = (PFIBERINFO) pvParam;

    // показываем в окне, какое волокно выполняется сейчас
    SetDlgItemText(pFiberInfo->hwnd, IDC_FIBER, TEXT("Recalculation"));

    // получаем текущее значение из поля ввода
    int nCount = GetDlgItemInt(pFiberInfo->hwnd, IDC_COUNT, NULL, FALSE);

    // считаем от 0 до nCount, обновляя статический элемент управления
    for (int x = 0; x <= nCount; x++) {

        // события пользовательского интерфейса приоритетнее расчетов
        // (если такие события есть, обрабатываем их)
        if (HIWORD(GetQueueStatus(QS_ALLEVENTS)) != 0) {

            // интерфейсное волокно чем-то занято; временно приостанавливаем
            // пересчет и обрабатываем события пользовательского интерфейса
            SwitchToFiber(pFiberInfo->pFiberUI);

            // событий больше нет, возобновляем пересчет
            SetDlgItemText(pFiberInfo->hwnd, IDC_FIBER, TEXT("Recalculation"));
        }

        // обновляем статический элемент управления,
        // показывая последнее значение счетчика
        SetDlgItemInt(pFiberInfo->hwnd, IDC_ANSWER, x, FALSE);

        // засыпаем, чтобы "потянуть" время;
        // в рабочем коде вызов Sleep надо убрать
        Sleep(200);
    }

    // сообщаем, что пересчет закончен
    pFiberInfo->bps = BPS_DONE;

    // Выделяем процессорное время интерфейсному волокну. Если
    // событий, подлежащих обработке, нет, приостанавливаем его.
    // Примечание: если разрешить возврат из функции волокна,
    // поток и интерфейсное волокно завершатся, а мы этого не хотим!
    SwitchToFiber(pFiberInfo->pFiberUI);
}

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

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

    chSETDLGICONS(hwnd, IDI_COUNTER);
    SetDlgItemInt(hwnd, IDC_COUNT, 0, FALSE);
    return(TRUE);
}

```

см. след. стр.

Рис. 12-1. *продолжение*

```

////////////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    switch (id) {
        case IDCANCEL:
            PostQuitMessage(0);
            break;

        case IDC_COUNT:
            if (codeNotify == EN_CHANGE) {
                // пользователь изменил значение счетчика,
                // начинаем фоновую обработку заново
                g_FiberInfo.bps = BPS_STARTOVER;
            }
            break;
    }
}

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

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

    switch (uMsg) {
        case WM_INITDIALOG:
            Dlg_OnInitDialog(hwnd, wParam, lParam);
        case WM_COMMAND:
            Dlg_OnCommand(hwnd, wParam, lParam);
    }

    return(FALSE);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    // контекст волокна, отвечающего за пересчет
    PVOID pFiberCounter = NULL;

    // преобразуем поток в волокно
    g_FiberInfo.pFiberUI = ConvertThreadToFiber(NULL);

    // создаем окно программы
    g_FiberInfo.hwnd = CreateDialog(hinstExe, MAKEINTRESOURCE(IDD_COUNTER),
        NULL, Dlg_Proc);

    // обновляем окно, показывая, какое волокно
    // выполняется в данный момент
    SetDlgItemText(g_FiberInfo.hwnd, IDC_FIBER, TEXT("User interface"));

    // изначально фоновая обработка отсутствует
    g_FiberInfo.bps = BPS_DONE;
}

```

Рис. 12-1. *продолжение*

```

// пока пользовательское окно существует...
BOOL fQuit = FALSE;
while (!fQuit) {

    // интерфейсные сообщения приоритетнее фоновой обработки
    MSG msg;
    if (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {

        // если в очереди есть сообщение, обрабатываем его
        fQuit = (msg.message == WM_QUIT);
        if (!IsDialogMessage(g_FiberInfo.hwnd, &msg)) {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }

    } else {

        // если сообщений нет, проверяем состояние фоновой обработки
        switch (g_FiberInfo.bps) {
            case BPS_DONE:
                // нет необходимости в фоновой обработке;
                // ждем событие, связанное с пользовательским интерфейсом
                WaitMessage();
                break;

            case BPS_STARTOVER:
                // пользователь изменил счетчик,
                // начинаем фоновую обработку заново

                if (pFiberCounter != NULL) {
                    // если волокно, отвечающее за пересчет, уже существует,
                    // удаляем его, и фоновая обработка начнется заново
                    DeleteFiber(pFiberCounter);
                    pFiberCounter = NULL;
                }

                // создаем новое волокно для пересчета, которое
                // начнет все с начала
                pFiberCounter = CreateFiber(0, FiberFunc, &g_FiberInfo);

                // фоновая обработка началась, ее нужно продолжать
                g_FiberInfo.bps = BPS_CONTINUE;

                // переходим к BPS_CONTINUE...

            case BPS_CONTINUE:
                // переключаемся на волокно пересчета...
                SwitchToFiber(pFiberCounter);

                // фоновая обработка приостановлена из-за появления сообщения от
                // пользовательского интерфейса или потому, что вычисления закончены

```

см. след. стр.

Рис. 12-1. *продолжение*

```

        // обновляем окно, показывая, какое волокно
        // выполняется в данный момент
        SetDlgItemText(g_FiberInfo.hwnd,
            IDC_FIBER, TEXT("User interface"));

        if (g_FiberInfo.bps == BPS_DONE) {
            // фоновая обработка закончена, удаляем
            // это волокно, чтобы в следующий раз фоновая
            // обработка началась заново
            DeleteFiber(pFiberCounter);
            pFiberCounter = NULL;
        }
        break;
    } // конец оператора switch
} // сообщений от пользовательского интерфейса нет
} // while (окно существует)
DestroyWindow(g_FiberInfo.hwnd);

return(0); // программа завершена
}

////////////////////////////////////// Конец файла ////////////////////////////////////////

```

Ч А С Т Ь III

УПРАВЛЕНИЕ ПАМЯТЬЮ



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 экзбайтам, 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 экзбайтов. 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.



В Windows 2000 память, принадлежащая собственно операционной системе, тоже скрыта от любого выполняемого потока. Иными словами, ни один поток не может случайно повредить ее данные. А в Windows 98 последнее, увы, не реализовано, и есть вероятность, что выполняемый поток, случайно получив доступ к данным операционной системы, тем самым нарушит ее нормальную работу. И все-таки в Windows 98, как и в Windows 2000, ни один поток не может получить доступ к памяти чужого процесса.

Итак, как я уже говорил, адресное пространство процесса закрыто. Отсюда вытекает, что процесс А 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 2000 создают разделы, почти одинаковые по назначению, но отличающиеся по размеру и расположению. Однако ядро Windows 98 формирует другие разделы. Давайте рассмотрим, как система использует каждый из этих разделов.

| Раздел | 32-разрядная Windows 2000 (на x86 и Alpha) | 32-разрядная Windows 2000 (на x86 с ключом /3GB) | 64-разрядная Windows 2000 (на Alpha и IA-64) | Windows 98 |
|---|--|--|--|----------------------------|
| Для выявления нулевых указателей | 0x00000000 0x0000FFFF | 0x00000000 0x0000FFFF | 0x00000000 00000000 0x00000000 0000FFFF | 0x00000000 0x00000FFF |
| Для совместимости с программами DOS и 16-разрядной Windows | Нет | Нет | Нет | 0x00001000 0x003FFFFFFF |
| Для кода и данных пользовательского режима | 0x00010000 0x7FFEFFFF | 0x00010000 0xBFFEFFFF | 0x00000000 00010000 0x000003FF FFFEFFFF | 0x00400000 0x7FFFFFFF |
| Закрытый, размером 64 Кб | 0x7FFF0000 0x7FFFFFFF | 0xBFFF0000 0xBFFFFFFF | 0x000003FF FFFF0000 0x000003FF FFFFFFFF | Нет |
| Для общих MMF (файлов, проеци- руемых в память) | Нет | Нет | Нет | 0x80000000 0xBFFFFFFF |
| Для кода и данных режима ядра | 0x80000000 0xFFFFFFFF | 0xC0000000 0xFFFFFFFF | 0x00000400 00000000 0xFFFFFFFF FFFFFFFF | 0xC0000000 0xFFFFFFFF |

Таблица 13-1. Так адресное пространство процесса разбивается на разделы



Microsoft активно работает над 64-разрядной Windows 2000. На момент написания книги эта система все еще находилась в разработке. Информацию по 64-разрядной Windows 2000 следует учитывать при проектировании и реализации текущих проектов. Однако Вы должны понимать, что какие-то детали скорее всего изменятся к моменту выхода 64-разрядной Windows 2000. То же самое относится и к конкретным диапазонам разделов виртуального адресного пространства и размеру страниц памяти на процессорах IA-64 (64-разрядной архитектуры Intel).

Раздел для выявления нулевых указателей (Windows 2000 и Windows 98)

Этот раздел адресного пространства 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.

Раздел для совместимости с программами DOS и 16-разрядной Windows (только Windows 98)

Этот регион размером 4 Мб в адресном пространстве процесса необходим Windows 98 для поддержки совместимости с программами MS-DOS и 16-разрядной Windows. Не пытайтесь обращаться к нему из 32-разрядных Windows-приложений. В идеале процессор должен был бы генерировать нарушение доступа при обращении потока к этому участку адресного пространства, но по техническим причинам Microsoft не смогла заблокировать эти 4 Мб адресного пространства.

В Windows 2000 программы для MS-DOS и 16-разрядной Windows выполняются в собственных адресных пространствах; 32-разрядные приложения повлиять на них не могут.

Раздел для кода и данных пользовательского режима (Windows 2000 и Windows 98)

В этом разделе располагается закрытая (неразделяемая) часть адресного пространства процесса. Ни один процесс не может получить доступ к данным другого процесса, размещенным в 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.

WINDOWS 2000 В Windows 2000 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.

WINDOWS 98 В Windows 98 основные 32-разрядные системные DLL (Kernel32.dll, AdvAPI32.dll, User32.dll и GDI32.dll) загружаются в раздел для общих MMF (проецируемых в память файлов), а EXE- и остальные DLL-модули — в раздел для кода и данных пользовательского режима. Общие DLL располагаются по одному и тому же виртуальному адресу во всех процессах, но другие DLL могут загружать их (общие DLL) по разным адресам в границах раздела для кода и данных пользова-

тельского режима (хотя это маловероятно). Проецируемые в память файлы в этот раздел никогда не помещаются.

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, Microsoft едва-едва втиснула ядро в эти виртуальные два гигабайта. В 64-разрядной Windows 2000 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 (только Windows 2000)

Многие разработчики уже давно сетовали на нехватку адресного пространства для пользовательского режима. Microsoft пошла навстречу и предусмотрела в версиях Windows 2000 Advanced Server и Windows 2000 Data Center для процессоров x86 возможность увеличения этого пространства до 3 Гб. Чтобы все процессы использовали раздел для кода и данных пользовательского режима размером 3 Гб, а раздел для кода и данных режима ядра — объемом 1 Гб, Вы должны добавить ключ /3GB к нужной записи в системном файле Boot.ini. Как выглядит адресное пространство процесса в этом случае, показано в графе «32-разрядная Windows 2000 (на x86 с ключом /3GB)» таблицы 13-1.

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.

Microsoft 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 в диапазоне адресов от 0x80000000 до 0xBFFFFFFF. Это предотвращает выделение памяти по адресам с установленным старшим битом.

Заметьте, что ядро и так с трудом уместается 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 16 Гб оперативной памяти против 64 Гб в нормальных условиях — из-за нехватки виртуального адресного пространства для кода и данных режима ядра, необходимого для управления дополнительной оперативной памятью.



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 2000

Microsoft 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 делиться 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-разрядное приложение, система резервирует все адресное пространство пользовательского режима, начиная с 0x00000000 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-разрядное приложение могло адресоваться ко всему разделу пользовательского режима (объемом 4 Тб), его следует скомпоновать с ключом /LARGEADDRESSAWARE.



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.

Закрытый раздел размером 64 Кб (только Windows 2000)

Этот раздел заблокирован, и любая попытка обращения к нему приводит к нарушению доступа. Microsoft резервирует этот раздел специально, чтобы упростить внутреннюю реализацию операционной системы. Вспомните: когда Вы передаете Windows-функции адрес блока памяти и его размер, то она (функция), прежде чем приступить к работе, проверяет, действителен ли данный блок. Допустим, Вы написали код:

```
BYTE bBuf[70000];
DWORD dwNumBytesWritten;
WriteProcessMemory(GetCurrentProcess(), (PVOID) 0x7FFEEE90, bBuf,
    sizeof(bBuf), &dwNumBytesWritten);
```

В случае функций типа *WriteProcessMemory* область памяти, в которую предполагается запись, проверяется кодом, работающим в режиме ядра, — только он имеет право обращаться к памяти, выделяемой под код и данные режима ядра (в 32-разрядных системах — по адресам выше 0x80000000). Если по этому адресу есть память, вызов *WriteProcessMemory*, показанный выше, благополучно запишет данные в ту область памяти, которая, по идее, доступна только коду, работающему в режиме ядра. Чтобы предотвратить это и в то же время ускорить проверку таких областей памяти, Microsoft предпочла заблокировать данный раздел, и поэтому любая попытка чтения или записи в нем всегда вызывает нарушение доступа.

Раздел для общих MMF (только Windows 98)

В этом разделе размером 1 Гб система хранит данные, разделяемые всеми 32-разрядными процессами. Сюда, например, загружаются все системные DLL (Kernel32.dll, AdvAPI32.dll, User32.dll и GDI32.dll), и поэтому они доступны любому 32-разрядному процессу. Кроме того, эти DLL загружаются в каждом процессе по одному и тому же адресу памяти. На этот раздел система также отображает все проецируемые в память файлы. Об этих файлах мы поговорим в главе 17.

Раздел для кода и данных режима ядра (Windows 2000 и Windows 98)

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.)

WINDOWS 2000 В 64-разрядной Windows 2000 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 большая 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.

WINDOWS 98 В Windows 98 данные, размещенные в этом разделе, увы, не защищены — любое приложение может что-то считать или записать в нем и нарушить нормальную работу операционной системы.

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 через 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 (x86, 32- и 64-разрядных Alpha и IA-64), — она одинакова и составляет 64 Кб.

Резервируя регион в адресном пространстве, система обеспечивает еще и кратность размера региона размеру *страницы*. 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 он равен 4 Кб, а для Alpha (под управлением как 32-разрядной, так и 64-разрядной Windows 2000) — 8 Кб. На момент написания книги предполагалось, что IA-64

тоже будет работать со страницами размером 8 Кб. Однако в зависимости от результатов тестирования Microsoft может выбрать для него страницы большего размера (от 16 Кб).



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 будет выделен регион размером 12 Кб, а на Alpha — 16 Кб.

И последнее в этой связи. Когда зарезервированный регион адресного пространства становится не нужен, 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 память 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 (страницы по 4 Кб), а справа — на процессоре Alpha (страницы по 8 Кб).

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, используя

ющие 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 64 Мб оперативной памяти, а размер страничного файла на жестком диске составляет 100 Мб, приложение считает, что объем оперативной памяти равен 164 Мб.

Конечно, 164 Мб 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.

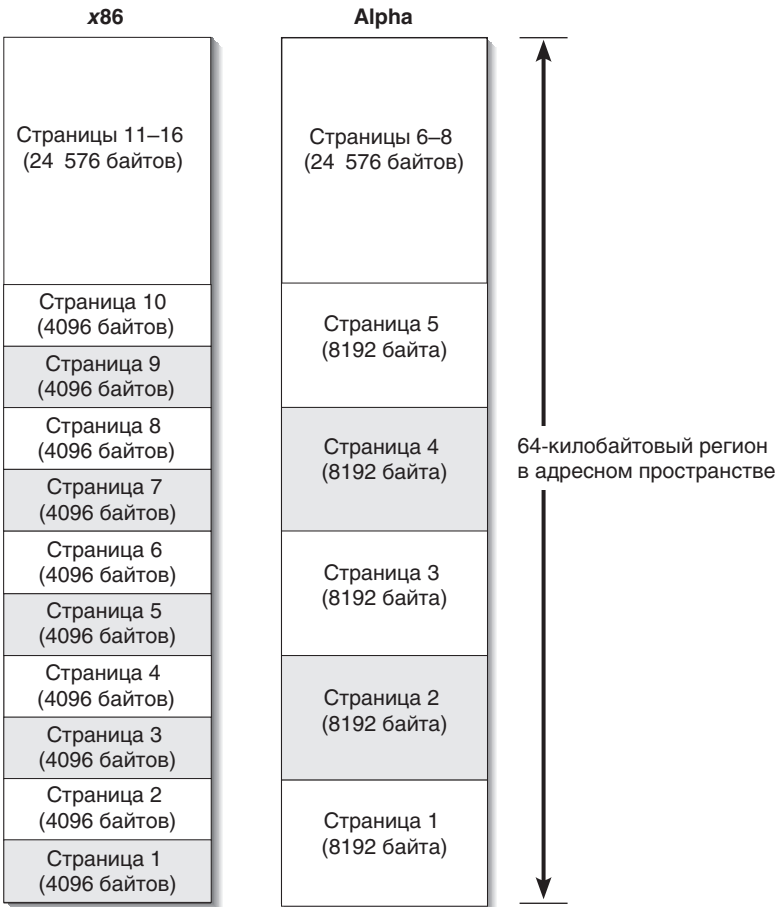


Рис. 13-1. Примеры адресных пространств процессов для разных типов процессоров

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).

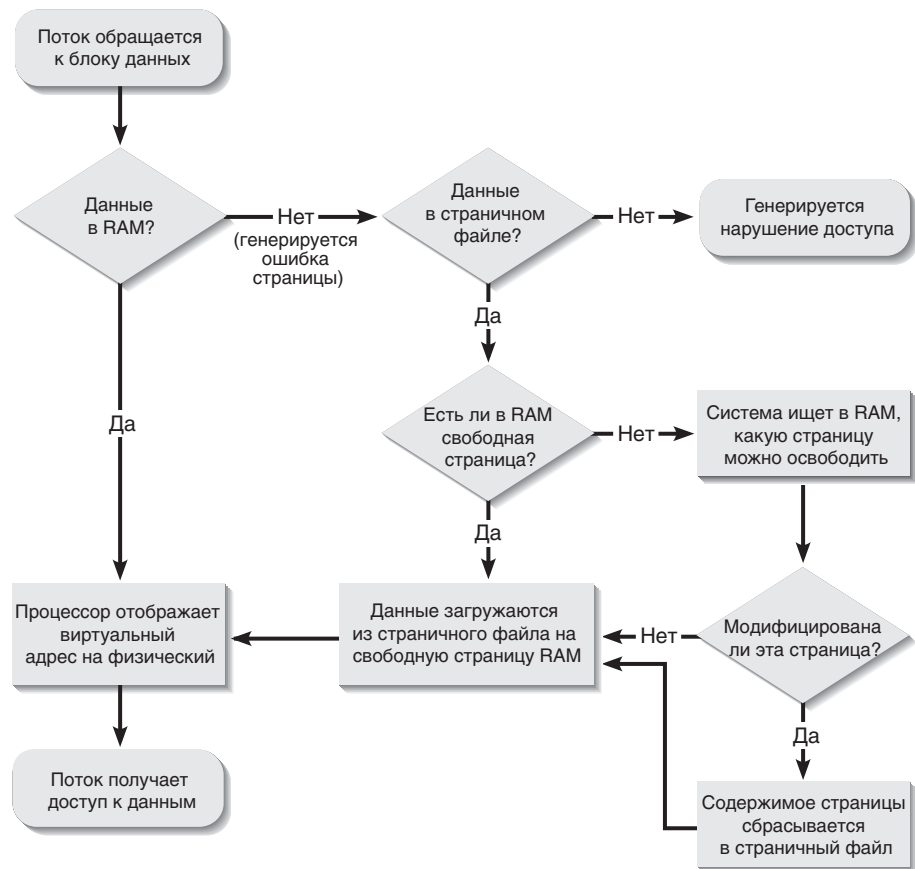


Рис. 13-2. Трансляция виртуального адреса на физический

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 не модифицировалась, она просто освобождается; в ином случае она сначала копируется из оператив-

ной 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.

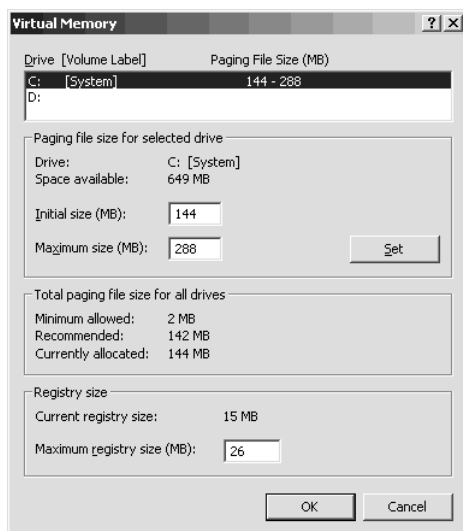
Чем чаще системе приходится копировать страницы памяти в страничный файл и наоборот, тем больше нагрузка на жесткий диск и тем медленнее работает операционная система. (При этом может получиться так, что операционная система будет тратить все свое время на подкачку страниц вместо выполнения программ.) 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.

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.

WINDOWS 2000

Windows 2000 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 апплет System, выберите вкладку Advanced и щелкните кнопку Performance Options. На экране появится следующее диалоговое окно:



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 удастся 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.)



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. (Имейте в виду, что Windows 98 не поддерживает флаги SWAPRUN.)

Атрибуты защиты

Отдельным страницам физической памяти можно присвоить свои атрибуты защиты, показанные в следующей таблице.

| 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 |

продолжение

| Атрибут защиты | Описание |
|------------------------|--|
| 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 |

На процессорных платформах x86 и Alpha атрибут PAGE_EXECUTE не поддерживается, хотя в операционных системах такая поддержка предусмотрена. Перечисленные процессоры воспринимают запрос на чтение как запрос на исполнение. Поэтому присвоение памяти атрибута PAGE_EXECUTE приводит к тому, что на этих процессорах она считается доступной и для чтения. Но полагаться на эту особенность не стоит, поскольку в реализациях Windows на других процессорах все может встать на свои места.

WINDOWS 98 В Windows 98 страницам физической памяти можно присвоить только три атрибута защиты: PAGE_NOACCESS, PAGE_READONLY и PAGE_READWRITE.

Защита типа «копирование при записи»

Атрибуты защиты, перечисленные в 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 поток в одном процессе попытается что-нибудь записать в общий блок памяти, в дело тут же вступит система и проделает следующие операции:

1. Найдёт 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 ещё 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_WRITECOPY или PAGE_EXECUTE_WRITECOPY. 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.

WINDOWS 98 Windows 98 не поддерживает «копирование при записи». Обнаружив запрос на применение такой защиты, Windows 98 тут же делает копии данных, не дожидаясь попытки записи в память.

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 2000 использует этот флаг при создании стека потока. Подробнее на эту тему см. раздел «Стек потока» в главе 16.

WINDOWS 98 Windows 98 игнорирует флаги атрибутов защиты `PAGE_NOCACHE`, `PAGE_WRITECOMBINE` и `PAGE_GUARD`.

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 под управлением Windows 2000 на 32-разрядной процессорной платформе x86. Образец карты адресного пространства VMMap показан в таблице 13-2. На отличиях адресных пространств в Windows 2000 и Windows 98 я остановлюсь чуть позже.

Карта в таблице 13-2 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 простран-

ства, 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.

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|---------------|---------|------------|-------|-------------------|---|
| 00000000 | Free | 65536 | | | |
| 00010000 | Private | 4096 | 1 | -RW- | |
| 00011000 | Free | 61440 | | | |
| 00020000 | Private | 4096 | 1 | -RW- | |
| 00021000 | Free | 61440 | | | |
| 00030000 | Private | 1048576 | 3 | -RW- | Стек потока |
| 00130000 | Private | 1048576 | 2 | -RW- | |
| 00230000 | Mapped | 65536 | 2 | -RW- | |
| 00240000 | Mapped | 90112 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\unicode.nls |
| 00256000 | Free | 40960 | | | |
| 00260000 | Mapped | 208896 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\locale.nls |
| 00293000 | Free | 53248 | | | |
| 002A0000 | Mapped | 266240 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\sortkey.nls |
| 002E1000 | Free | 61440 | | | |
| 002F0000 | Mapped | 16384 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\sorttbls.nls |
| 002F4000 | Free | 49152 | | | |
| 00300000 | Mapped | 819200 | 4 | ER-- | |
| 003C8000 | Free | 229376 | | | |
| 00400000 | Image | 106496 | 5 | ERWC | C:\CD\x86\Debug\14 VMMap.exe |
| 0041A000 | Free | 24576 | | | |
| 00420000 | Mapped | 274432 | 1 | -R-- | |
| 00463000 | Free | 53248 | | | |
| 00470000 | Mapped | 3145728 | 2 | ER-- | |
| 00770000 | Private | 4096 | 1 | -RW- | |
| 00771000 | Free | 61440 | | | |
| 00780000 | Private | 4096 | 1 | -RW- | |
| 00781000 | Free | 61440 | | | |
| 00790000 | Private | 65536 | 2 | -RW- | |
| 007A0000 | Mapped | 8192 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\ctype.nls |
| 007A2000 | Free | 1763893248 | | | |
| 699D0000 | Image | 45056 | 4 | ERWC | C:\WINNT\System32\PSAPI.dll |
| 699DB000 | Free | 238505984 | | | |
| 77D50000 | Image | 450560 | 4 | ERWC | C:\WINNT\system32\RPCRT4.DLL |
| 77DBE000 | Free | 8192 | | | |
| 77DC0000 | Image | 344064 | 5 | ERWC | C:\WINNT\system32\ADVAPI32.dll |
| 77E14000 | Free | 49152 | | | |
| 77E20000 | Image | 401408 | 4 | ERWC | C:\WINNT\system32\USER32.dll |

Таблица 13-2. Образец карты адресного пространства процесса в Windows 2000 на 32-разрядном процессоре типа x86

Таблица 13-2. *продолжение*

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|---------------|---------|-----------|-------|-------------------|--------------------------------|
| 77E82000 | Free | 57344 | | | |
| 77E90000 | Image | 720896 | 5 | ERWC | C:\WINNT\system32\KERNEL32.dll |
| 77F40000 | Image | 241664 | 4 | ERWC | C:\WINNT\system32\GDI32.DLL |
| 77F7B000 | Free | 20480 | | | |
| 77F80000 | Image | 483328 | 5 | ERWC | C:\WINNT\System32\ntdll.dll |
| 77FF6000 | Free | 40960 | | | |
| 78000000 | Image | 290816 | 6 | ERWC | C:\WINNT\system32\MSVCRT.dll |
| 78047000 | Free | 124424192 | | | |
| 7F6F0000 | Mapped | 1048576 | 2 | ER-- | |
| 7F7F0000 | Free | 8126464 | | | |
| 7FFB0000 | Mapped | 147456 | 1 | -R-- | |
| 7FFD4000 | Free | 40960 | | | |
| 7FFDE000 | Private | 4096 | 1 | ERW- | |
| 7FFDF000 | Private | 4096 | 1 | ERW- | |
| 7FFE0000 | Private | 65536 | 2 | -R-- | |

Во втором поле показывается тип региона: Free (свободный), Private (закрытый), Image (образ) или Mapped (проецируемый). Эти типы описаны в следующей таблице.

| 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 поля 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, по результатам которого программа делает обоснованное предположение. Но предположение есть предположение. Если Вы хотите получше разобраться в том, как это делается, просмотрите исходный код VMMap, приведенный в главе 14.

В третьем поле сообщается размер региона в байтах. Например, система спроецировала образ User32.dll по адресу 0x77E20000. Когда она резервировала адресное

пространство для этого образа, ей понадобилось 401 408 байтов. Не забудьте, что в третьем поле всегда содержатся значения, кратные размеру страницы, характерному для данного процессора (4096 байтов для x86).

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 — 401 408 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 типа Image программе VMMap удастся определить полное имя файла, проецируемого на этот регион. Она получает эту информацию с помощью Tool-Helper-функций, о которых я упоминал в конце главы 4. В Windows 2000 программа VMMap может идентифицировать регионы, сопоставленные с файлами данных; для этого она вызывает функцию *GetMappedFileName* (ее нет в Windows 98).

Блоки внутри регионов

Попробуем увеличить детализацию адресного пространства (по сравнению с тем, что показано в таблице 13-2). Например, таблица 13-3 показывает ту же карту адресного пространства, но в другом «масштабе»: по ней можно узнать, из каких блоков состоит каждый регион.

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|---------------|---------|--------|-------|-------------------|----------|
| 00000000 | Free | 65536 | | | |
| 00010000 | Private | 4096 | 1 | -RW- | |
| 00010000 | Private | 4096 | | -RW- --- | |

Таблица 13-3. Образец карты адресного пространства процесса (с указанием блоков внутри регионов) в Windows 2000 на 32-разрядном процессоре типа x86

Таблица 13-3. *продолжение*

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|---------------|---------|---------|-------|-------------------|---|
| 00011000 | Free | 61440 | | | |
| 00020000 | Private | 4096 | 1 | -RW- | |
| 00020000 | Private | 4096 | | -RW- --- | |
| 00021000 | Free | 61440 | | | |
| 00030000 | Private | 1048576 | 3 | -RW- | Стек потока |
| 00030000 | Reserve | 905216 | | -RW- --- | |
| 0010D000 | Private | 4096 | | -RW- G-- | |
| 0010E000 | Private | 139264 | | -RW- --- | |
| 00130000 | Private | 1048576 | 2 | -RW- | |
| 00130000 | Private | 36864 | | -RW- --- | |
| 00139000 | Reserve | 1011712 | | -RW- --- | |
| 00230000 | Mapped | 65536 | 2 | -RW- | |
| 00230000 | Mapped | 4096 | | -RW- --- | |
| 00231000 | Reserve | 61440 | | -RW- --- | |
| 00240000 | Mapped | 90112 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\unicode.nls |
| 00240000 | Mapped | 90112 | | -R-- --- | |
| 00256000 | Free | 40960 | | | |
| 00260000 | Mapped | 208896 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\locale.nls |
| 00260000 | Mapped | 208896 | | -R-- --- | |
| 00293000 | Free | 53248 | | | |
| 002A0000 | Mapped | 266240 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\sortkey.nls |
| 002A0000 | Mapped | 266240 | | -R-- --- | |
| 002E1000 | Free | 61440 | | | |
| 002F0000 | Mapped | 16384 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\sorttbls.nls |
| 002F0000 | Mapped | 16384 | | -R-- --- | |
| 002F4000 | Free | 49152 | | | |
| 00300000 | Mapped | 819200 | 4 | ER-- | |
| 00300000 | Mapped | 16384 | | ER-- --- | |
| 00304000 | Reserve | 770048 | | ER-- --- | |
| 003C0000 | Mapped | 8192 | | ER-- --- | |
| 003C2000 | Reserve | 24576 | | ER-- --- | |
| 003C8000 | Free | 229376 | | | |
| 00400000 | Image | 106496 | 5 | ERWC | C:\CD\x86\Debug\14 VMMap.exe |
| 00400000 | Image | 4096 | | -R-- --- | |
| 00401000 | Image | 81920 | | ER-- --- | |
| 00415000 | Image | 4096 | | -R-- --- | |
| 00416000 | Image | 8192 | | -RW- --- | |
| 00418000 | Image | 8192 | | -R-- --- | |
| 0041A000 | Free | 24576 | | | |
| 00420000 | Mapped | 274432 | 1 | -R-- | |
| 00420000 | Mapped | 274432 | | -R-- --- | |
| 00463000 | Free | 53248 | | | |
| 00470000 | Mapped | 3145728 | 2 | ER-- | |

Таблица 13-3. *продолжение*

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|---------------|---------|------------|-------|-------------------|--|
| 00470000 | Mapped | 274432 | | ER-- --- | |
| 004B3000 | Reserve | 2871296 | | ER-- --- | |
| 00770000 | Private | 4096 | 1 | -RW- | |
| 00770000 | Private | 4096 | | -RW- --- | |
| 00771000 | Free | 61440 | | | |
| 00780000 | Private | 4096 | 1 | -RW- | |
| 00780000 | Private | 4096 | | -RW- --- | |
| 00781000 | Free | 61440 | | | |
| 00790000 | Private | 65536 | 2 | -RW- | |
| 00790000 | Private | 20480 | | -RW- --- | |
| 00795000 | Reserve | 45056 | | -RW- --- | |
| 007A0000 | Mapped | 8192 | 1 | -R-- | \Device\HarddiskVolume1\WINNT\system32\ctype.nls |
| 007A0000 | Mapped | 8192 | | -R-- --- | |
| 007A2000 | Free | 57344 | | | |
| 007B0000 | Private | 524288 | 2 | -RW- | |
| 007B0000 | Private | 4096 | | -RW- --- | |
| 007B1000 | Reserve | 520192 | | -RW- --- | |
| 00830000 | Free | 1763311616 | | | |
| 699D0000 | Image | 45056 | 4 | ERWC | C:\WINNT\System32\PSAPI.dll |
| 699D0000 | Image | 4096 | | -R-- --- | |
| 699D1000 | Image | 16384 | | ER-- --- | |
| 699D5000 | Image | 16384 | | -RWC --- | |
| 699D9000 | Image | 8192 | | -R-- --- | |
| 699DB000 | Free | 238505984 | | | |
| 77D50000 | Image | 450560 | 4 | ERWC | C:\WINNT\system32\RPCRT4.DLL |
| 77D50000 | Image | 4096 | | -R-- --- | |
| 77D51000 | Image | 421888 | | ER-- --- | |
| 77DB8000 | Image | 4096 | | -RW- --- | |
| 77DB9000 | Image | 20480 | | -R-- --- | |
| 77DBE000 | Free | 8192 | | | |
| 77DC0000 | Image | 344064 | 5 | ERWC | C:\WINNT\system32\ADVAPI32.dll |
| 77DC0000 | Image | 4096 | | -R-- --- | |
| 77DC1000 | Image | 307200 | | ER-- --- | |
| 77E0C000 | Image | 4096 | | -RW- --- | |
| 77E0D000 | Image | 4096 | | -RWC --- | |
| 77E0E000 | Image | 24576 | | -R-- --- | |
| 77E14000 | Free | 49152 | | | |
| 77E20000 | Image | 401408 | 4 | ERWC | C:\WINNT\system32\USER32.dll |
| 77E20000 | Image | 4096 | | -R-- --- | |
| 77E21000 | Image | 348160 | | ER-- --- | |
| 77E76000 | Image | 4096 | | -RW- --- | |
| 77E77000 | Image | 45056 | | -R-- --- | |
| 77E82000 | Free | 57344 | | | |

Таблица 13-3. *продолжение*

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|---------------|---------|-----------|-------|-------------------|--------------------------------|
| 77E90000 | Image | 720896 | 5 | ERWC | C:\WINNT\system32\KERNEL32.dll |
| 77E90000 | Image | 4096 | | -R-- --- | |
| 77E91000 | Image | 368640 | | ER-- --- | |
| 77EEB000 | Image | 8192 | | -RW- --- | |
| 77EED000 | Image | 4096 | | -RWC --- | |
| 77EEE000 | Image | 335872 | | -R-- --- | |
| 77F40000 | Image | 241664 | 4 | ERWC | C:\WINNT\system32\GDI32.DLL |
| 77F40000 | Image | 4096 | | -R-- --- | |
| 77F41000 | Image | 221184 | | ER-- --- | |
| 77F77000 | Image | 4096 | | -RW- --- | |
| 77F78000 | Image | 12288 | | -R-- --- | |
| 77F7B000 | Free | 20480 | | | |
| 77F80000 | Image | 483328 | 5 | ERWC | C:\WINT\System32\ntdll.dll |
| 77F80000 | Image | 4096 | | -R-- --- | |
| 77F81000 | Image | 299008 | | ER-- --- | |
| 77FCA000 | Image | 8192 | | -RW- --- | |
| 77FCC000 | Image | 4096 | | -RWC --- | |
| 77FCD000 | Image | 167936 | | -R-- --- | |
| 77FF6000 | Free | 40960 | | | |
| 78000000 | Image | 290816 | 6 | ERWC | C:\WINNT\system32\MSVCRT.dll |
| 78000000 | Image | 4096 | | -R-- --- | |
| 78001000 | Image | 208896 | | ER-- --- | |
| 78034000 | Image | 32768 | | -R-- --- | |
| 7803C000 | Image | 12288 | | -RW- --- | |
| 7803F000 | Image | 16384 | | RWC --- | |
| 78043000 | Image | 16384 | | -R-- --- | |
| 78047000 | Free | 124424192 | | | |
| 7F6F0000 | Mapped | 1048576 | 2 | ER-- | |
| 7F6F0000 | Mapped | 28672 | | ER-- --- | |
| 7F6F7000 | Reserve | 1019904 | | ER-- --- | |
| 7F7F0000 | Free | 8126464 | | | |
| 7FFB0000 | Mapped | 147456 | 1 | -R-- | |
| 7FFB0000 | Mapped | 147456 | | -R-- --- | |
| 7FFD4000 | Free | 40960 | | | |
| 7FFDE000 | Private | 4096 | 1 | ERW- | |
| 7FFDE000 | Private | 4096 | | ERW- --- | |
| 7FFDF000 | Private | 4096 | 1 | ERW- | |
| 7FFDF000 | Private | 4096 | | ERW- --- | |
| 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 0x77E20000 передана единственная страница (4096 байтов) физической памяти с атрибутом защиты, разрешающим только чтение. А по адресу 0x77E21000 присутствует блок размером 85 страниц (348 160 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 86 страниц (352 256 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 подsunет 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_WRITECOMBINE и PAGE_NOCACHE, недопустимые для региона.

Особенности адресного пространства в Windows 98

В таблице 13-4 показана карта адресного пространства при выполнении все той же программы VMMap, но уже под управлением Windows 98. Для экономии места диапазон виртуальных адресов между 0x80018000 и 0x85620000 не приведен.

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|---------------|---------|---------|-------|-------------------|------------------------------|
| 00000000 | Free | 4194304 | | | |
| 00400000 | Private | 131072 | 6 | ---- | C:\CD\X86\DEBUG\14 VMMap.EXE |
| 00400000 | Private | 8192 | | -R-- --- | |
| 00402000 | Private | 8192 | | -RW- --- | |
| 00404000 | Private | 73728 | | -R-- --- | |
| 00416000 | Private | 8192 | | -RW- --- | |
| 00418000 | Private | 8192 | | -R-- --- | |

Таблица 13-4. Образец карты адресного пространства процесса (с указанием блоков внутри регионов) в Windows 98

Таблица 13-4. *продолжение*

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|------------------|---------|------------|-------|----------------------|------------------------------|
| 0041A000 | Reserve | 24576 | | ---- | |
| 00420000 | Private | 1114112 | 4 | ---- | |
| 00420000 | Private | 20480 | | -RW- --- | |
| 00425000 | Reserve | 1028096 | | ---- | |
| 00520000 | Private | 4096 | | -RW- --- | |
| 00521000 | Reserve | 61440 | | ---- | |
| 00530000 | Private | 65536 | 2 | -RW- | |
| 00530000 | Private | 4096 | | -RW- --- | |
| 00531000 | Reserve | 61440 | | -RW- --- | |
| 00540000 | Private | 1179648 | 6 | ---- | Стек потока |
| 00540000 | Reserve | 942080 | | ---- | |
| 00626000 | Private | 4096 | | -RW- --- | |
| 00627000 | Reserve | 24576 | | ---- | |
| 0062D000 | Private | 4096 | | ---- | |
| 0062E000 | Private | 139264 | | -RW- --- | |
| 00650000 | Reserve | 65536 | | ---- | |
| 00660000 | Private | 1114112 | 4 | ---- | |
| 00660000 | Private | 20480 | | -RW- --- | |
| 00665000 | Reserve | 1028096 | | ---- | |
| 00760000 | Private | 4096 | | -RW- --- | |
| 00761000 | Reserve | 61440 | | ---- | |
| 00770000 | Private | 1048576 | 2 | -RW- | |
| 00770000 | Private | 32768 | | -RW- --- | |
| 00778000 | Reserve | 1015808 | | -RW- --- | |
| 00870000 | Free | 2004418560 | | | |
| 78000000 | Private | 262144 | 3 | ---- | C:\WINDOWS\SYSTEM\MSVCRT.DLL |
| 78000000 | Private | 188416 | | -R-- --- | |
| 7802E000 | Private | 57344 | | -RW- --- | |
| 7803C000 | Private | 16384 | | -R-- --- | |
| 78040000 | Free | 133955584 | | | |
| 80000000 | Private | 4096 | 1 | ---- | |
| 80000000 | Reserve | 4096 | | ---- | |
| 80001000 | Private | 4096 | 1 | ---- | |
| 80001000 | Private | 4096 | | -RW- --- | |
| 80002000 | Private | 4096 | 1 | ---- | |
| 80002000 | Private | 4096 | | -RW- --- | |
| 80003000 | Private | 4096 | 1 | ---- | |
| 80003000 | Private | 4096 | | -RW- --- | |
| 80004000 | Private | 65536 | 2 | ---- | |
| 80004000 | Private | 32768 | | -RW- --- | |
| 8000C000 | Reserve | 32768 | | ---- | |
| 80014000 | Private | 4096 | 1 | ---- | |
| 80014000 | Private | 4096 | | -RW- --- | |

Таблица 13-4. *продолжение*

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|------------------|---------|-----------|-------|----------------------|----------|
| 80015000 | Private | 4096 | 1 | ---- | |
| 80015000 | Private | 4096 | | -RW- --- | |
| 80016000 | Private | 4096 | 1 | ---- | |
| 80016000 | Private | 4096 | | -RW- --- | |
| 80017000 | Private | 4096 | 1 | ---- | |
| 80017000 | Private | 4096 | | -RW- --- | |
| 85620000 | Free | 9773056 | | | |
| 85F72000 | Private | 151552 | 1 | ---- | |
| 85F72000 | Private | 151552 | | -R-- --- | |
| 85F97000 | Private | 327680 | 1 | ---- | |
| 85F97000 | Private | 327680 | | -R-- --- | |
| 85FE7000 | Free | 22052864 | | | |
| 874EF000 | Private | 4194304 | 1 | ---- | |
| 874EF000 | Reserve | 4194304 | | ---- --- | |
| 878EF000 | Free | 679219200 | | | |
| B00B0000 | Private | 880640 | 3 | ---- | |
| B00B0000 | Private | 233472 | | -R-- --- | |
| B00E9000 | Private | 20480 | | -RW- --- | |
| B00EE000 | Private | 626688 | | -R-- --- | |
| B0187000 | Free | 177311744 | | | |
| BAAA0000 | Private | 315392 | 7 | ---- | |
| BAAA0000 | Private | 4096 | | -R-- --- | |
| BAAA1000 | Private | 4096 | | -RW- --- | |
| BAAA2000 | Private | 241664 | | -R-- --- | |
| BAADD000 | Private | 4096 | | -RW- --- | |
| BAADE000 | Private | 4096 | | -R-- --- | |
| BAADF000 | Private | 32768 | | -RW- --- | |
| BAAE7000 | Private | 24576 | | -R-- --- | |
| BAAED000 | Free | 86978560 | | | |
| BFDE0000 | Private | 20480 | 1 | ---- | |
| BFDE0000 | Private | 20480 | | -R-- --- | |
| BFDE5000 | Free | 45056 | | | |
| BFDF0000 | Private | 65536 | 3 | ---- | |
| BFDF0000 | Private | 40960 | | -R-- --- | |
| BFDF0000 | Private | 4096 | | -RW- --- | |
| BFDFB000 | Private | 20480 | | -R-- --- | |
| BFE00000 | Free | 131072 | | | |
| BFE20000 | Private | 16384 | 3 | ---- | |
| BFE20000 | Private | 8192 | | -R-- --- | |
| BFE22000 | Private | 4096 | | -RW- --- | |
| BFE23000 | Private | 4096 | | -R-- --- | |
| BFE24000 | Free | 245760 | | | |
| BFE60000 | Private | 24576 | 3 | ---- | |

Таблица 13-4. *продолжение*

| Базовый адрес | Тип | Размер | Блоки | Атрибут(ы) защиты | Описание |
|---------------|---------|--------|-------|-------------------|--------------------------------|
| BFE60000 | Private | 8192 | | -R-- --- | |
| BFE62000 | Private | 4096 | | -RW- --- | |
| BFE63000 | Private | 12288 | | -R-- --- | |
| BFE66000 | Free | 40960 | | | |
| BFE70000 | Private | 24576 | 3 | ---- | |
| BFE70000 | Private | 8192 | | -R-- --- | |
| BFE72000 | Private | 4096 | | -RW- --- | |
| BFE73000 | Private | 12288 | | -R-- --- | |
| BFE76000 | Free | 40960 | | | |
| BFE80000 | Private | 65536 | 3 | ---- | C:\WINDOWS\SYSTEM\ADVAPI32.DLL |
| BFE80000 | Private | 49152 | | -R-- --- | |
| BFE8C000 | Private | 4096 | | -RW- --- | |
| BFE8D000 | Private | 12288 | | -R-- --- | |
| BFE90000 | Private | 573440 | 3 | ---- | |
| BFE90000 | Private | 425984 | | -R-- --- | |
| BFEF8000 | Private | 4096 | | -RW- --- | |
| BFEF9000 | Private | 143360 | | -R-- --- | |
| BFF1C000 | Free | 16384 | | | |
| BFF20000 | Private | 155648 | 5 | ---- | C:\WINDOWS\SYSTEM\GDI32.DLL |
| BFF20000 | Private | 126976 | | -R-- --- | |
| BFF3F000 | Private | 8192 | | -RW- --- | |
| BFF41000 | Private | 4096 | | -R-- --- | |
| BFF42000 | Private | 4096 | | -RW- --- | |
| BFF43000 | Private | 12288 | | -R-- --- | |
| BFF46000 | Free | 40960 | | | |
| BFF50000 | Private | 69632 | 3 | ---- | C:\WINDOWS\SYSTEM\USER32.DLL |
| BFF50000 | Private | 53248 | | -R-- --- | |
| BFF5D000 | Private | 4096 | | -RW- --- | |
| BFF5E000 | Private | 12288 | | -R-- --- | |
| BFF61000 | Free | 61440 | | | |
| BFF70000 | Private | 585728 | 5 | ---- | C:\WINDOWS\SYSTEM\KERNEL32.DLL |
| BFF70000 | Private | 352256 | | -R-- --- | |
| BFFC6000 | Reserve | 12288 | | ---- | |
| BFFC9000 | Private | 16384 | | -RW- --- | |
| BFFCD000 | Private | 90112 | | -R-- --- | |
| BFFE3000 | Reserve | 114688 | | ---- | |
| BFFFF000 | Free | 4096 | | | |

Главное отличие двух карт адресного пространства в том, что под управлением Windows 98 информации получаешь значительно меньше. Например, о регионах и блоках можно узнать лишь, свободные они, резервные или закрытые. Распознать тип физической памяти Mapped или Image нельзя; Windows 98 не позволяет получить дополнительную информацию, по которой можно было бы судить, что с регионом связан проецируемый в память файл или образ исполняемого файла.

Наверное, Вы заметили, что размер большинства регионов кратен 64 Кб (это значение определяется гранулярностью выделения памяти). Если размеры блоков, составляющих регион, не дают в сумме величины, кратной 64 Кб, то в конце региона часто появляется резервный блок адресного пространства. Его размер выбирается системой так, чтобы довести общий объем региона до величины, кратной 64 Кб. Например, регион, который начинается с адреса 0x00530000, включает в себя два блока: четырехкилобайтовый блок переданной памяти и резервный блок, занимающий 60 Кб адресного пространства.

Заметьте также, что на последней карте не встречаются атрибуты защиты, разрешающие исполнение или копирование при записи, поскольку Windows 98 не поддерживает их. Кроме того, она не поддерживает и флаги атрибутов защиты (PAGE_GUARD, PAGE_WRITECOMBINE и PAGE_NOCACHE). Из-за этого программе VMMap приходится использовать более сложный метод, чтобы определить, не выделен ли данный регион под стек потока.

И последнее. В Windows 98 (в отличие от Windows 2000) можно исследовать регион адресного пространства 0x80000000–0xBFFFFFFF. Это раздел, в котором находится адресное пространство, общее для всех 32-разрядных приложений. По карте видно, что в него загружены четыре системные DLL, и поэтому они доступны любому процессу.

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 вызовет 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 процессорах Alpha 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 выров-

ненному! 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 2000 для процессоров типа x86 и Windows 98 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.

Теперь обратим внимание на процессор Alpha. Он не умеет оперировать с невыровненными данными. Когда происходит попытка доступа к таким данным, этот процессор уведомляет операционную систему. Далее Windows 2000 решает, что делать — генерировать соответствующее исключение или самой устранить возникшую проблему, выдав процессору дополнительные инструкции. По умолчанию Windows 2000, установленная на компьютере с процессором Alpha, сама исправляет все ошибки обращения к невыровненным данным. Однако Вы можете изменить ее поведение. При загрузке Windows 2000 проверяет раздел реестра:

```
HKEY_LOCAL_MACHINE\CurrentControlSet\Control\Session Manager
```

В этом разделе может присутствовать параметр EnableAlignmentFaultExceptions. Если его нет (что чаще всего и бывает), Windows 2000 сама исправляет ошибки, связанные с доступом к невыровненным данным. Но, если он есть, система учитывает его значение. При его нулевом значении система действует так же, как и в отсутствие этого параметра. Если же он равен 1, система не исправляет такие ошибки, а генерирует исключения. Никогда не модифицируйте этот параметр в реестре без особой необходимости, потому что иначе некоторые приложения будут вызывать исключения из-за доступа к невыровненным данным и аварийно завершаться.

Чтобы упростить изменение этого параметра реестра, с Microsoft Visual C++ для платформы Alpha поставляется утилита AXPAlign.exe. Она используется так, как показано ниже.

```
Alpha AXP alignment fault exception control
```

```
Usage: axpalign [option]
```

```
Options:
```

```
  /enable  to enable alignment fault exceptions.
  /disable to disable alignment fault exceptions.
  /show    to display the current alignment exception setting.
```

```
Enable alignment fault exceptions:
```

В этом режиме любое обращение к невыровненным данным приведет к исключению. Приложение может быть закрыто. В своем коде Вы можете найти источник ошибок, связанных с выравниванием данных, с помощью отладчика. Действие этого параметра распространяется на все выполняемые процессы, и использовать его следует с осторожностью, так как в старых приложениях могут возникать необрабатываемые ими исключения. Заметьте, что SetErrorMode(SEM_NOALIGNMENTFAULTEXCEPT) позволяет подавить генерацию таких исключений даже в этом режиме.

Disable alignment fault exceptions:

Этот режим действует по умолчанию в Windows NT for Alpha AXP версий 3.1 и 3.5. Операционная система сама исправляет любые ошибки, связанные с доступом к невыровненным данным (если таковые ошибки возникают), и приложения или отладчики их не замечают. Если программа часто обращается к невыровненным данным, производительность системы может заметно снизиться. Для наблюдения за частотой появления таких ошибок можно использовать `Perfmon` или `wperf`.

Эта утилита просто модифицирует нужный параметр реестра или показывает его текущее значение. Изменив значение этого параметра, перезагрузите компьютер, чтобы изменения вступили в силу.

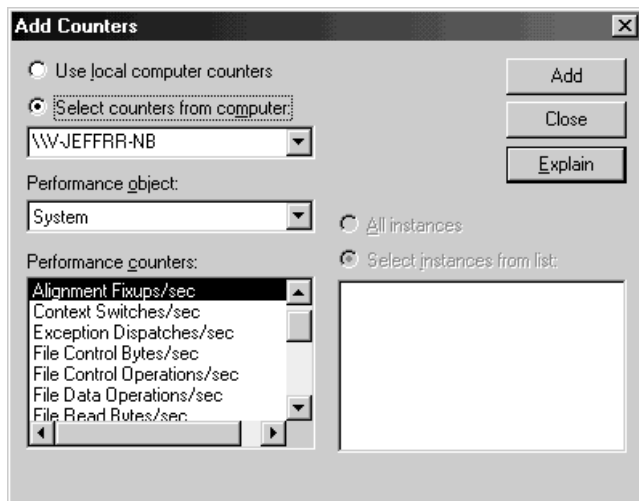
Но, даже не пользуясь утилитой `AXPAlign`, Вы все равно можете заставить систему молча исправлять ошибки обращения к невыровненным данным во всех потоках Вашего процесса. Для этого один из потоков должен вызвать функцию `SetErrorMode`:

```
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. 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 платформе `x86` сбросить этот флаг просто нельзя, а на платформе Alpha его разрешается сбросить, только если параметр `EnableAlignmentFaultExceptions` в реестре равен 1.

Для наблюдения за частотой возникновения ошибок, связанных с доступом к невыровненным данным, в Windows 2000 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.

Как видите, простого вызова *SetErrorMode* вполне достаточно для того, чтобы Ваше приложение работало корректно. Но это решение явно не самое эффективное. Так, в *Alpha Architecture Reference Manual*, опубликованном Digital Press, утверждается, что системный код, автоматически устраняющий ошибки обращения к невыровненным данным, может снизить быстродействие в 100 раз! Издержки слишком велики. К счастью, есть более эффективное решение этой проблемы.

Компилятор Microsoft C/C++ для процессоров Alpha 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*.

```
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 = * (__unaligned DWORD *) pvDataBuffer;

    // Предыдущая строка заставит компилятор сгенерировать дополнительные
    // машинные инструкции, которые позволят считать значение типа DWORD
    // в несколько приемов. При этом исключение из-за попытки доступа
    // к невыровненным данным не возникнет.
    :
}
```

При компиляции следующей строки на процессоре Alpha, генерируется 7 машинных инструкций.

```
DWORD dw = * (__unaligned DWORD *) pvDataBuffer;
```

Но если я уберу ключевое слово *__unaligned*, то получу всего 3 машинные инструкции. Как видите, модификатор *__unaligned* на процессорах Alpha 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.

И последнее. Ключевое слово *__unaligned* на процессорах типа x86 компилятором Visual C/C++ не поддерживается. На этих процессорах оно просто 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 для обеих процессорных платформ, используйте вместо *__unaligned* макрос `UNALIGNED`. Он определен в файле `WinNT.h` так:

```
#if defined(_M_MRX000) || defined(_M_ALPHA) || defined(_M_IA64)
#define UNALIGNED __unaligned
#endif
#define UNALIGNED64 __unaligned
#else
#define UNALIGNED64
#endif
#define UNALIGNED
#define UNALIGNED64
#endif
```

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 psinf);
```

Вы должны передать в *GetSystemInfo* адрес структуры SYSTEM_INFO, и функция инициализирует элементы этой структуры:

```
typedef struct _SYSTEM_INFO {
    union {
        DWORD dwOemId; // не используйте этот элемент, он устарел
        struct {
            WORD wProcessorArchitecture;
            WORD wReserved;
        };
    };
};

DWORD      dwPageSize;
LPVOID     lpMinimumApplicationAddress;
LPVOID     lpMaximumApplicationAddress;
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.

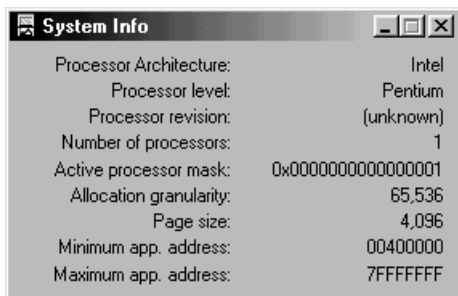
| Элемент | Описание |
|------------------------------------|---|
| <i>dwPageSize</i> | Размер страницы памяти. На процессорах x86 это значение равно 4096, а на процессорах Alpha — 8192 байтам. |
| <i>lpMinimumApplicationAddress</i> | Минимальный адрес памяти доступного адресного пространства для каждого процесса. В Windows 98 это значение равно 4 194 304, или 0x00400000, поскольку нижние 4 Мб адресного пространства каждого процесса недоступны. В Windows 2000 это значение равно 65 536, или 0x00010000, так как в этой системе резервируются лишь первые 64 Кб адресного пространства каждого процесса. |
| <i>lpMaximumApplicationAddress</i> | Максимальный адрес памяти доступного адресного пространства, отведенного в «личное пользование» каждому процессу. В Windows 98 этот адрес равен 2 147 483 647, или 0x7FFFFFFF, так как верхние 2 Гб занимают общие файлы, проецируемые в память, и разделяемый код операционной системы. В Windows 2000 этот адрес соответствует началу раздела для кода и данных режима ядра за вычетом 64 Кб. |
| <i>dwAllocationGranularity</i> | 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.

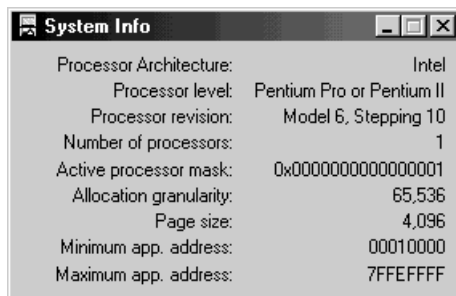
| Элемент | Описание |
|-------------------------------|--|
| <i>dwOemId</i> | Устарел; больше не используется |
| <i>wReserved</i> | Зарезервирован на будущее; пока не используется |
| <i>dwNumberOfProcessors</i> | Число процессоров в компьютере |
| <i>dwActiveProcessorMask</i> | Битовая маска, которая сообщает, какие процессоры активны (выполняют потоки) |
| <i>dwProcessorType</i> | Используется только в Windows 98; сообщает тип процессора, например Intel 386, 486 или Pentium |
| <i>wProcessorArchitecture</i> | Используется только в Windows 2000; сообщает тип архитектуры процессора, например Intel, Alpha, 64-разрядный Intel или 64-разрядный Alpha |
| <i>wProcessorLevel</i> | Используется только в Windows 2000; сообщает дополнительные подробности об архитектуре процессора, например Intel Pentium Pro или Pentium II |
| <i>wProcessorRevision</i> | Используется только в Windows 2000; сообщает дополнительные подробности об уровне данной архитектуры процессора |

Программа-пример SysInfo

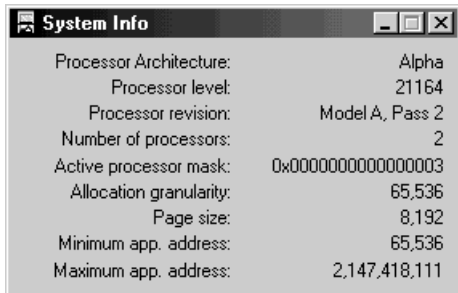
Эта программа, «14 SysInfo.exe» (см. листинг на рис. 14-1), весьма проста; она вызывает функцию *GetSystemInfo* и выводит на экран информацию, возвращенную в структуре *SYSTEM_INFO*. Файлы исходного кода и ресурсов этой программы находятся в каталоге 14-SysInfo на компакт-диске, прилагаемом к книге. Диалоговые окна с результатами выполнения программы SysInfo на разных процессорных платформах показаны ниже.



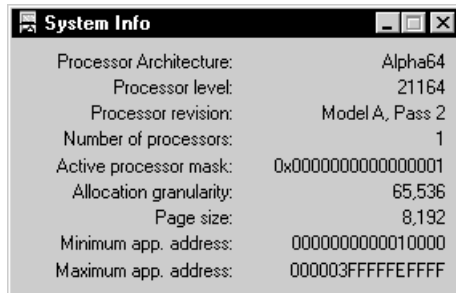
Windows 98 на процессоре x86



32-разрядная Windows 2000 на процессоре x86



32-разрядная Windows 2000 на процессоре Alpha



64-разрядная Windows 2000 на процессоре Alpha



SysInfo.cpp

```

/*****
Модуль: SysInfo.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h" /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h>
#include "Resource.h"

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

// устанавливаем TRUE, если программа выполняется в Windows 9x
BOOL g_fWin9xIsHost = FALSE;

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

// эта функция принимает число и преобразует его в строку,
// вставляя в нужных местах запятые
PTSTR BigNumToString(LONG lNum, PTSTR szBuf) {

    TCHAR szNum[100];

```

Рис. 14-1. Программа-пример SysInfo

Рис. 14-1. *продолжение*

```

    wsprintf(szNum, TEXT("%d"), lNum);
    NUMBERFMT 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, 100);
    return(szBuf);
}

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

void ShowCPUInfo(HWND hwnd, WORD wProcessorArchitecture, WORD wProcessorLevel,
    WORD wProcessorRevision) {

    TCHAR szCPUArch[64] = TEXT("(unknown)");
    TCHAR szCPULevel[64] = TEXT("(unknown)");
    TCHAR szCPURev[64] = TEXT("(unknown)");

    switch (wProcessorArchitecture) {
        case PROCESSOR_ARCHITECTURE_INTEL:
            lstrcpy(szCPUArch, TEXT("Intel"));
            switch (wProcessorLevel) {
                case 3: case 4:
                    wsprintf(szCPULevel, TEXT("80%c86"), wProcessorLevel + '0');
                    if (!g_fWin9xIsHost)
                        wsprintf(szCPURev, TEXT("%c%d"),
                            HIBYTE(wProcessorRevision) + TEXT('A'), LOBYTE(wProcessorRevision));
                    break;

                case 5:
                    wsprintf(szCPULevel, TEXT("Pentium"));
                    if (!g_fWin9xIsHost)
                        wsprintf(szCPURev, TEXT("Model %d, Stepping %d"),
                            HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
                    break;

                case 6:
                    wsprintf(szCPULevel, TEXT("Pentium Pro or Pentium II"));
                    if (!g_fWin9xIsHost)
                        wsprintf(szCPURev, TEXT("Model %d, Stepping %d"),
                            HIBYTE(wProcessorRevision), LOBYTE(wProcessorRevision));
                    break;
            }
            break;

        case PROCESSOR_ARCHITECTURE_ALPHA:
            lstrcpy(szCPUArch, TEXT("Alpha"));
            wsprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
    }
}

```

см. след. стр.

Рис. 14-1. *продолжение*

```

        wprintf(szCPURev, TEXT("Model %c, Pass %d"),
            HIBYTE(wProcessorRevision) + TEXT('A'), LOBYTE(wProcessorRevision));
        break;

    case PROCESSOR_ARCHITECTURE_IA64:
        lstrcpy(szCPUArch, TEXT("IA-64"));
        wprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
        wprintf(szCPURev, TEXT("Model %c, Pass %d"),
            HIBYTE(wProcessorRevision) + TEXT('A'), LOBYTE(wProcessorRevision));
        break;

    case PROCESSOR_ARCHITECTURE_ALPHA64:
        lstrcpy(szCPUArch, TEXT("Alpha64"));
        wprintf(szCPULevel, TEXT("%d"), wProcessorLevel);
        wprintf(szCPURev, TEXT("Model %c, Pass %d"),
            HIBYTE(wProcessorRevision) + TEXT('A'), LOBYTE(wProcessorRevision));
        break;

    case PROCESSOR_ARCHITECTURE_UNKNOWN:
    default:
        wprintf(szCPUArch, TEXT("Unknown"));
        break;
}
SetDlgItemText(hwnd, IDC_PROCCARCH, szCPUArch);
SetDlgItemText(hwnd, IDC_PROCLEVEL, szCPULevel);
SetDlgItemText(hwnd, IDC_PROCREV, szCPURev);
}

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

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

    chSETDLGICONS(hwnd, IDI_SYSINFO);

    SYSTEM_INFO sinf;
    GetSystemInfo(&sinf);

    if (g_fWin9xIsHost) {
        sinf.wProcessorLevel = (WORD) (sinf.dwProcessorType / 100);
    }

    ShowCPUInfo(hwnd, sinf.wProcessorArchitecture,
        sinf.wProcessorLevel, sinf.wProcessorRevision);

    TCHAR szBuf[50];
    SetDlgItemText(hwnd, IDC_PAGESIZE, BigNumToString(sinf.dwPageSize, szBuf));

    _stprintf(szBuf, TEXT("%p"), sinf.lpMinimumApplicationAddress);
    SetDlgItemText(hwnd, IDC_MINAPPADDR, szBuf);

    _stprintf(szBuf, TEXT("%p"), sinf.lpMaximumApplicationAddress);
    SetDlgItemText(hwnd, IDC_MAXAPPADDR, szBuf);
}

```

Рис. 14-1. *продолжение*

```

    _stprintf(szBuf, TEXT("0x%016I64X"), (__int64) sinf.dwActiveProcessorMask);
    SetDlgItemText(hwnd, IDC_ACTIVEPROCMASK, szBuf);

    SetDlgItemText(hwnd, IDC_NUMOFPROCS,
        BigNumToString(sinf.dwNumberOfProcessors, szBuf));

    SetDlgItemText(hwnd, IDC_ALLOCGRAN,
        BigNumToString(sinf.dwAllocationGranularity, szBuf));

    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) {
        case WM_INITDIALOG: Dlg_OnInitDialog(hDlg);
        case WM_COMMAND:    Dlg_OnCommand(hDlg, LOWORD(wParam), HIWORD(wParam));
    }
    return(FALSE);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    OSVERSIONINFO vi = { sizeof(vi) };
    GetVersionEx(&vi);
    g_fWin9xIsHost = (vi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS);
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SYSINFO), NULL, Dlg_Proc);
    return(0);
}

///////////////////////////////////////////////////////////////// Конец файла ///////////////////////////////////////////////////////////////////

```

Статус виртуальной памяти

Windows-функция *GlobalMemoryStatus* позволяет отслеживать текущее состояние памяти:

```
VOID GlobalMemoryStatus(LPMEMORYSTATUS pmst);
```

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 dwMemoryLoad;
    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 Microsoft 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(LPMEMORYSTATUSEX pmst);
```

Вы должны передать ей адрес новой структуры *MEMORYSTATUSEX*:

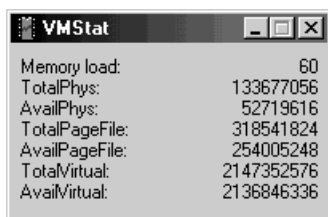
```
typedef struct _MEMORYSTATUSEX {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    DWORDLONG ullTotalPhys;
    DWORDLONG ullAvailPhys;
    DWORDLONG ullTotalPageFile;
    DWORDLONG ullAvailPageFile;
    DWORDLONG ullTotalVirtual;
    DWORDLONG ullAvailVirtual;
    DWORDLONG ullAvailExtendedVirtual;
} 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.

Программа-пример *VMStat*

Эта программа, «14 *VMStat.exe*» (см. листинг на рис. 14-2), выводит на экран окно с результатами вызова *GlobalMemoryStatus*. Информация в окне обновляется каждую секунду, так что *VMStat* вполне пригодна для мониторинга памяти в системе. Файлы

исходного кода и ресурсов этой программы находятся в каталоге 14-VMStat на компакт-диске, прилагаемом к книге. Окно этой программы после запуска в Windows 2000 на машине с процессором Intel Pentium II и 128 Мб оперативной памяти показано ниже.



| | |
|----------------|------------|
| VMStat | |
| Memory load: | 60 |
| TotalPhys: | 133677056 |
| AvailPhys: | 52719616 |
| TotalPageFile: | 318541824 |
| AvailPageFile: | 254005248 |
| TotalVirtual: | 2147352576 |
| AvailVirtual: | 2136846336 |

Элемент *dwMemoryLoad* (показываемый как Memory Load) позволяет оценить, насколько занята подсистема управления памятью. Это число может быть любым в диапазоне от 0 до 100. В Windows 98 и Windows 2000 алгоритмы, используемые для его подсчета, различны. 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 Pentium II и 128 Мб оперативной памяти его значение составляет 133 677 056, что на 540 672 байта меньше 128 Мб. Причина, по которой *GlobalMemoryStatus* не сообщает о полных 128 Мб, 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 318 574 592 байта, система может варьировать его по своему усмотрению. Элемент *dwAvailPageFile* (показываемый как AvailPageFile) подсказывает, что в данный момент 233 046 016 байтов 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. Запустив VMStat в Windows 98, Вы увидите, что значение этого элемента поменялось на 2 143 289 344 (2 Гб за вычетом 4 Мб). Разница в 4 Мб возникает из-за того, что Windows 98 блокирует нижний раздел от 0x00000000 до 0x003FFFFF (размером в 4 Мб).

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 2 136 846 336 байтов свободного адресного пространства. Вычтя из значения *dwTotalVirtual* величину *dwAvailVirtual*, Вы получите 10 506 240 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.



VMStat.cpp

```

/*****
Модуль: VMStat.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h>
#include "Resource.h"

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

// идентификатор таймера, отвечающего за обновление информации
#define IDT_UPDATE 1

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

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

    chSETDLGICONS(hwnd, IDI_VMSTAT);

    // устанавливаем таймер так, чтобы периодически обновлять информацию
    SetTimer(hwnd, IDT_UPDATE, 1 * 1000, NULL);
    // выдаем сообщение таймера для первого обновления
    FORWARD_WM_TIMER(hwnd, IDT_UPDATE, SendMessage);
    return(TRUE);
}

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

void Dlg_OnTimer(HWND hwnd, UINT id) {

    // прежде чем передать структуру функции GlobalMemoryStatus,
    // заносим в элемент dwLength ее длину
    MEMORYSTATUS ms = { sizeof(ms) };
    GlobalMemoryStatus(&ms);

    TCHAR szData[512] = { 0 };
    _stprintf(szData, TEXT("%d\n%d\n%I64d\n%I64d\n%I64d\n%I64d\n%I64d"),
        ms.dwMemoryLoad, ms.dwTotalPhys,
        (__int64) ms.dwAvailPhys,    (__int64) ms.dwTotalPageFile,
        (__int64) ms.dwAvailPageFile, (__int64) ms.dwTotalVirtual,
        (__int64) ms.dwAvailVirtual);
    SetDlgItemText(hwnd, IDC_DATA, szData);
}

```

Рис. 14-2. Программа-пример VMStat

Рис. 14-2. *продолжение*

```

////////////////////////////////////
void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {
    switch (id) {
        case IDCANCEL:
            KillTimer(hwnd, IDT_UPDATE);
            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);
        chHANDLE_DLGMSG(hwnd, WM_TIMER, Dlg_OnTimer);
    }
    return(FALSE);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMSTAT), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////

```

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 (ее листинг см. на рис. 14-4) выводит карты виртуальной памяти, с которыми мы познакомились в главе 13. Вот эта функция:

```

DWORD VirtualQuery(
    LPCVOID pvAddress,
    PMEMORY_BASIC_INFORMATION pmbi,
    DWORD dwLength);

```

Парная ей функция, *VirtualQueryEx*, сообщает ту же информацию о памяти, но в другом процессе:

```

DWORD 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;
    DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_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.

| Элемент | Описание |
|--------------------------|---|
| <i>BaseAddress</i> | Сообщает то же значение, что и параметр <i>pvAddress</i> , но округленное до ближайшего меньшего адреса, кратного размеру страницы. |
| <i>AllocationBase</i> | Идентифицирует базовый адрес региона, включающего в себя адрес, указанный в параметре <i>pvAddress</i> . |
| <i>AllocationProtect</i> | Идентифицирует атрибут защиты, присвоенный региону при его резервировании. |
| <i>RegionSize</i> | Сообщает суммарный размер (в байтах) группы страниц, которые начинаются с базового адреса <i>BaseAddress</i> и имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>pvAddress</i> . |
| <i>State</i> | Сообщает состояние (<code>MEM_FREE</code> , <code>MEM_RESERVE</code> или <code>MEM_COMMIT</code>) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>pvAddress</i> . При <code>MEM_FREE</code> элементы <i>AllocationBase</i> , <i>AllocationProtect</i> , <i>Protect</i> и <i>Type</i> содержат неопределенные значения, а при <code>MEM_RESERVE</code> неопределенное значение содержит элемент <i>Protect</i> . |
| <i>Protect</i> | Идентифицирует атрибут защиты (<code>PAGE_*</code>) всех смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>pvAddress</i> . |
| <i>Type</i> | Идентифицирует тип физической памяти (<code>MEM_IMAGE</code> , <code>MEM_MAPPED</code> или <code>MEM_PRIVATE</code>), связанной с группой смежных страниц, которые имеют те же атрибуты защиты, состояние и тип, что и страница, расположенная по адресу, указанному в параметре <i>pvAddress</i> . В Windows 98 этот элемент всегда дает <code>MEM_PRIVATE</code> . |

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, содержащую элементы структуры MEMORY_BASIC_INFORMATION. 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,
    PVOID 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 {
    // информация о регионе
    PVOID pvRgnBaseAddress;
    DWORD dwRgnProtection;    // PAGE_*
    SIZE_T RgnSize;
    DWORD dwRgnStorage;        // MEM_*: Free, Image, Mapped, Private
    DWORD dwRgnBlocks;
    DWORD dwRgnGuardBlks;     // если > 0, регион содержит стек потока
    BOOL fRgnIsAStack;        // TRUE, если регион содержит стек потока

    // информация о блоке
    PVOID pvBlkBaseAddress;
    DWORD dwBlkProtection;    // PAGE_*
    SIZE_T BlkSize;
    DWORD dwBlkStorage;        // MEM_*: Free, Reserve, Image, Mapped, 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.

| Элемент | Описание |
|-------------------------|---|
| <i>pvRgnBaseAddress</i> | Идентифицирует базовый адрес региона виртуального адресного пространства, включающего адрес, указанный в параметре <i>pvAddress</i> . |
| <i>dwRgnProtection</i> | Сообщает атрибут защиты, присвоенный региону при его резервировании. |
| <i>RgnSize</i> | Указывает размер (в байтах) зарезервированного региона. |
| <i>dwRgnStorage</i> | Идентифицирует тип физической памяти, используемой группой блоков данного региона: MEM_FREE, MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE. Поскольку Windows 98 не различает типы памяти, в этой операционной системе данный элемент содержит либо MEM_FREE, либо MEM_PRIVATE. |
| <i>dwRgnBlocks</i> | Содержит значение — число блоков в указанном регионе. |
| <i>dwRgnGuardBlks</i> | 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. В Windows 98 этот элемент всегда равен 0. |
| <i>fRgnIsAStack</i> | 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. |
| <i>pvBlkBaseAddress</i> | Идентифицирует базовый адрес блока, включающего адрес, указанный в параметре <i>pvAddress</i> . |
| <i>dwBlkProtection</i> | Идентифицирует атрибут защиты блока, включающего адрес, указанный в параметре <i>pvAddress</i> . |
| <i>BlkSize</i> | Содержит значение — размер блока (в байтах), включающего адрес, указанный в параметре <i>pvAddress</i> . |
| <i>dwBlkStorage</i> | Идентифицирует содержимое блока, включающего адрес, указанный в параметре <i>pvAddress</i> . Принимает одно из значений: MEM_FREE, MEM_RESERVE, MEM_IMAGE, MEM_MAPPED или MEM_PRIVATE. В Windows 98 этот элемент никогда не содержит значения MEM_IMAGE и MEM_MAPPED. |

Чтобы получить всю эту информацию, *VMQuery*, естественно, приходится выполнять гораздо больше операций (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 из этих функций. 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 комментариев, вольно разбросанных по всему коду.

VMQuery.cpp

```

/*****
Модуль: VMQuery.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */

```

Рис. 14-3. Функция *VMQuery*

Рис. 14-3. *продолжение*

```

#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 fRgnIsAStack;    // 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 VMQuery
static DWORD gs_dwAllocGran = 0;

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

// эта функция проходит по всем блокам в регионе
// и инициализирует структуру найденными значениями
static BOOL VMQueryHelp(HANDLE hProcess, LPCVOID pvAddress,
    VMQUERY_HELP *pVMQHelp) {

    // каждый элемент содержит атрибут защиты страницы
    // (например, 0=зарезервирована, PAGE_NOACCESS, PAGE_READWRITE и т. д.)
    DWORD dwProtectBlock[4] = { 0 };

    ZeroMemory(pVMQHelp, sizeof(*pVMQHelp));

    // получаем базовый адрес региона, включающего переданный адрес памяти
    MEMORY_BASIC_INFORMATION mbi;
    BOOL fOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi)) == sizeof(mbi));

    if (!fOk)
        return(fOk); // неверный адрес памяти, сообщаем об ошибке

    // проходим по региону, начиная с его базового адреса
    // (который никогда не изменится)
    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

```

см. след. стр.

Рис. 14-3. *продолжение*

```

fOk = (VirtualQueryEx(hProcess, pvAddressBlk, &mbi, sizeof(mbi)) == sizeof(mbi));
if (!fOk)
    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

// следующий оператор if служит для обнаружения стеков в Windows 98; в этой
// системе стеки размещаются в последних 4 блоках региона: "зарезервированный",
// PAGE_NOACCESS, PAGE_READWRITE и еще один "зарезервированный"
if (pVMQHelp->dwRgnBlocks < 4) {
    // если это блок 0-3, запоминаем тип защиты блока в массиве
    dwProtectBlock[pVMQHelp->dwRgnBlocks] =
        (mbi.State == MEM_RESERVE) ? 0 : mbi.Protect;
} else {
    // мы уже просмотрели 4 блока в этом регионе;
    // смещаем вниз элементы массива с атрибутами защиты
    MoveMemory(&dwProtectBlock[0], &dwProtectBlock[1],
        sizeof(dwProtectBlock) - sizeof(DWORD));

    // добавляем новые значения атрибутов защиты в конец массива
    dwProtectBlock[3] = (mbi.State == MEM_RESERVE) ? 0 : mbi.Protect;
}

pVMQHelp->dwRgnBlocks++; // увеличиваем счетчик блоков
// в этом регионе на 1
pVMQHelp->RgnSize += mbi.RegionSize; // добавляем размер блока к размеру региона

// 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 из MEM_IMAGE
// в MEM_PRIVATE или из MEM_MAPPED в MEM_PRIVATE; MEM_PRIVATE в любой
// момент может быть замещен на MEM_IMAGE или MEM_MAPPED.
if (pVMQHelp->dwRgnStorage == MEM_PRIVATE)
    pVMQHelp->dwRgnStorage = mbi.Type;

// получаем адрес следующего блока
pvAddressBlk = (PVOID) ((PBYTE) pvAddressBlk + mbi.RegionSize);
}

// Обследовав регион, думаем: не стек ли это?
// Windows 2000: да – если в регионе содержится хотя бы 1 блок с флагом PAGE_GUARD.
// Windows 9x: да – если в регионе содержится хотя бы 4 блока,

```

Рис. 14-3. *продолжение*

```

//          и они имеют такие атрибуты:
//          3-й блок от конца: зарезервирован
//          2-й блок от конца: PAGE_NOACCESS
//          1-й блок от конца: PAGE_READWRITE
//          последний блок: зарезервирован
pVMQHelp->fRgnIsAStack =
    (pVMQHelp->dwRgnGuardBlks > 0)      ||
    ((pVMQHelp->dwRgnBlocks >= 4)        &&
     (dwProtectBlock[0] == 0)           &&
     (dwProtectBlock[1] == PAGE_NOACCESS) &&
     (dwProtectBlock[2] == PAGE_READWRITE) &&
     (dwProtectBlock[3] == 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 fOk = (VirtualQueryEx(hProcess, pvAddress, &mbi, sizeof(mbi))
        == sizeof(mbi));

    if (!fOk)
        return(fOk); // неверный адрес памяти, сообщаем об ошибке

    // структура MEMORY_BASIC_INFORMATION содержит действительную
    // информацию – пора заполнить элементы нашей структуры VMQUERY

    // во-первых, заполним элементы, описывающие состояние блока;
    // данные по региону получим позже
    switch (mbi.State) {
        case MEM_FREE: // свободный блок (незарезервированный)
            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;
    }
}

```

см. след. стр.

Рис. 14-3. *продолжение*

```

    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 содержится.
    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->fRgnIsAStack = 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->fRgnIsAStack = VMQHelp.fRgnIsAStack;
    break;

case MEM_COMMIT: // зарезервированный блок, которому передана физическая память
    pVMQ->pvRgnBaseAddress = mbi.AllocationBase;

```

Рис. 14-3. *продолжение*

```

    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->fRgnIsAStack      = VMQHelp.fRgnIsAStack;
    break;

default:
    DebugBreak();
    break;
}

return(fOk);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

VMQuery.h

```

/*****
Модуль: VMQuery.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

typedef struct {
    // информация о регионе
    PVOID  pvRgnBaseAddress;
    DWORD  dwRgnProtection;    // PAGE_*
    SIZE_T RgnSize;
    DWORD  dwRgnStorage;       // MEM_*: Free, Image, Mapped, Private
    DWORD  dwRgnBlocks;
    DWORD  dwRgnGuardBlks;    // если > 0, регион содержит стек потока
    BOOL   fRgnIsAStack;      // TRUE, если регион содержит стек потока
    // информация о блоке
    PVOID  pvBlkBaseAddress;
    DWORD  dwBlkProtection;    // PAGE_*
    SIZE_T BlkSize;
    DWORD  dwBlkStorage;       // MEM_*: Free, Reserve, Image, Mapped, Private
} VMQUERY, *PVMQUERY;

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

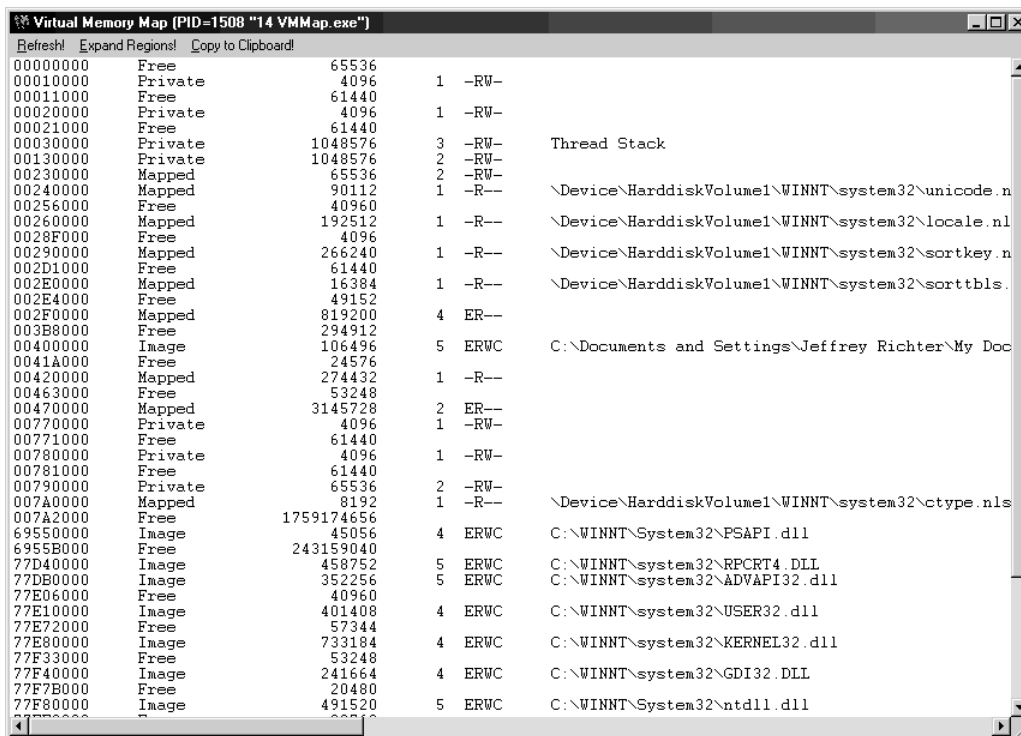
BOOL VMQuery(HANDLE hProcess, LPCVOID pvAddress, PVMQUERY pVMQ);

//////////////////////////////////// Конеч файл //////////////////////////////////////

```


Программа-пример VMMap

Эта программа, «14 VMMap.exe» (см. листинг на рис. 14-4), 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 на компакт-диске, прилагаемом к книге. После запуска VMMap на экране появляется следующее окно.



Карты виртуальной памяти, представленные в главе 13 в таблицах 13-2, 13-3 и 13-4, 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 fOk = TRUE;
PVOID pvAddress = NULL;
:
while (fOk) {

    VMQUERY vmq;
    fOk = VMQuery(hProcess, pvAddress, &vmq);

    if (fOk) {
        // формируем строку для вывода на экран
        // и добавляем ее в окно списка
        TCHAR szLine[1024];
        ConstructRgnInfoLine(hProcess, &vmq, szLine, sizeof(szLine));
        ListBox_AddString(hwndLB, szLine);

        if (fExpandRegions) {

```

```

for (DWORD dwBlock = 0; fOk && (dwBlock < vmq.dwRgnBlocks);
    dwBlock++) {

    ConstructBlkInfoLine(&vmq, szLine, sizeof(szLine));
    ListBox_AddString(hwndLB, szLine);

    // получаем адрес следующего региона
    pvAddress = ((PBYTE) pvAddress + vmq.BlkSize);
    if (dwBlock < vmq.dwRgnBlocks - 1) {
        // нельзя запрашивать информацию о памяти за последним блоком
        fOk = VMQuery(hProcess, pvAddress, &vmq);
    }
}

// получаем адрес следующего региона
pvAddress = ((PBYTE) vmq.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 *VMQuery* просматривать адресное пространство процесса очень легко.



VMMap.cpp

```

/*****
Модуль: VMMap.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h" /* см. приложение A */
#include <psapi.h>
#include <windowsx.h>
#include <tchar.h>
#include <stdio.h> // для доступа к sprintf
#include "..\04-ProcessInfo\Toolhelp.h"
#include "Resource.h"
#include "VMQuery.h"

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

DWORD g_dwProcessId = 0; // какой процесс надо пройти?
BOOL g_fExpandRegions = FALSE;
CToolhelp g_toolhelp;

```

Рис. 14-4. Программа-пример VMMap

см. след. стр.

Рис. 14-4. *продолжение*

```
// GetMappedFileName имеется только в Windows 2000 (в PSAPI.DLL);
// если эта функция в системе есть, используем именно ее
typedef DWORD (WINAPI* PFNGETMAPPEDFILENAME)(HANDLE, PVOID, PTSTR, DWORD);
static PFNGETMAPPEDFILENAME g_pfnGetMappedFileName = NULL;

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

// я использовал эту функцию, чтобы получить карты памяти, приведенные в книге
void CopyControlToClipboard(HWND hwnd) {
    TCHAR szClipData[128 * 1024] = { 0 };

    int nCount = ListBox_GetCount(hwnd);
    for (int nNum = 0; nNum < nCount; nNum++) {
        TCHAR szLine[1000];
        ListBox_GetText(hwnd, nNum, szLine);
        _tcscat(szClipData, szLine);
        _tcscat(szClipData, TEXT("\r\n"));
    }

    OpenClipboard(NULL);
    EmptyClipboard();

    // буфер обмена принимает только данные, находящиеся в блоке, выделенном
    // функцией GlobalAlloc с флагами GMEM_MOVEABLE и GMEM_DDESHARE
    HGLOBAL hClipData = GlobalAlloc(GMEM_MOVEABLE | GMEM_DDESHARE,
        sizeof(TCHAR) * (_tcslen(szClipData) + 1));
    PTSTR pClipData = (PTSTR) GlobalLock(hClipData);

    _tcscpy(pClipData, szClipData);

#ifdef UNICODE
    BOOL fOk = (SetClipboardData(CF_UNICODETEXT, hClipData) == hClipData);
#else
    BOOL fOk = (SetClipboardData(CF_TEXT, hClipData) == hClipData);
#endif
    CloseClipboard();

    if (!fOk) {
        GlobalFree(hClipData);
        chMB("Error putting text on the clipboard");
    }
}

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

PCTSTR GetMemStorageText(DWORD dwStorage) {

    PCTSTR p = TEXT("Unknown");
    switch (dwStorage) {
        case MEM_FREE:    p = TEXT("Free   "); break;
        case MEM_RESERVE: p = TEXT("Reserve"); break;
        case MEM_IMAGE:   p = TEXT("Image  "); break;
    }
}
```

Рис. 14-4. *продолжение*

```

    case MEM_MAPPED: p = TEXT("Mapped "); break;
    case MEM_PRIVATE: p = TEXT("Private"); break;
    }
    return(p);
}

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

PTSTR GetProtectText(DWORD dwProtect, PTSTR szBuf, BOOL fShowFlags) {

    PCTSTR p = TEXT("Unknown");
    switch (dwProtect & ~(PAGE_GUARD | PAGE_NOCACHE | PAGE_WRITECOMBINE)) {
    case PAGE_READONLY:      p = TEXT("-R-"); break;
    case PAGE_READWRITE:     p = TEXT("-RW-"); break;
    case PAGE_WRITECOPY:     p = TEXT("-RWC"); break;
    case PAGE_EXECUTE:       p = TEXT("E---"); break;
    case PAGE_EXECUTE_READ:  p = TEXT("ER--"); break;
    case PAGE_EXECUTE_READWRITE: p = TEXT("ERW-"); break;
    case PAGE_EXECUTE_WRITECOPY: p = TEXT("ERWC"); break;
    case PAGE_NOACCESS:      p = TEXT("----"); break;
    }
    _tcscpy(szBuf, p);
    if (fShowFlags) {
        _tcscat(szBuf, TEXT(" "));
        _tcscat(szBuf, (dwProtect & PAGE_GUARD)      ? TEXT("G") : TEXT("-"));
        _tcscat(szBuf, (dwProtect & PAGE_NOCACHE)    ? TEXT("N") : TEXT("-"));
        _tcscat(szBuf, (dwProtect & PAGE_WRITECOMBINE) ? TEXT("W") : TEXT("-"));
    }
    return(szBuf);
}

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

void ConstructRgnInfoLine(HANDLE hProcess, PVMQUERY pVMQ, PTSTR szLine, int nMaxLen) {

    _stprintf(szLine, TEXT("%p    %s %16u  "), pVMQ->pvRgnBaseAddress,
        GetMemStorageText(pVMQ->dwRgnStorage), pVMQ->RgnSize);
    if (pVMQ->dwRgnStorage != MEM_FREE) {
        wsprintf(_tcschr(szLine, 0), TEXT("%5u  "), pVMQ->dwRgnBlocks);
        GetProtectText(pVMQ->dwRgnProtection, _tcschr(szLine, 0), FALSE);
    }

    _tcscat(szLine, TEXT("    "));

    // пытаемся получить полное имя модуля для этого региона
    int nLen = _tcslen(szLine);
    if (pVMQ->pvRgnBaseAddress != NULL) {
        MODULEENTRY32 me = { sizeof(me) };
        if (g_toolhelp.ModuleFind(pVMQ->pvRgnBaseAddress, &me)) {
            lstrcat(&szLine[nLen], me.szExePath);
        } else {

```

см. след. стр.

Рис. 14-4. *продолжение*

```

        // это не модуль; проверяем, не является ли он файлом, проецируемым в память
        if (g_pfnGetMappedFileName != NULL) {
            DWORD d = g_pfnGetMappedFileName(hProcess,
                pVMQ->pvRgnBaseAddress, szLine + nLen, nMaxLen - nLen);
            if (d == 0) {
                // Примечание: GetMappedFileName модифицирует строку при неудаче
                szLine[nLen] = 0;
            }
        }
    }
}

if (pVMQ->fRgnIsAStack) {
    _tcsat(szLine, TEXT("Thread Stack"));
}
}

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

void ConstructBlkInfoLine(PVMQUERY pVMQ, PTSTR szLine, int nMaxLen) {

    _stprintf(szLine, TEXT("    %p %s %16u      "),
        pVMQ->pvBlkBaseAddress, GetMemStorageText(pVMQ->dwBlkStorage),
        pVMQ->BlkSize);

    if (pVMQ->dwBlkStorage != MEM_FREE) {
        GetProtectText(pVMQ->dwBlkProtection, _tcschr(szLine, 0), TRUE);
    }
}

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

void Refresh(HWND hwndLB, DWORD dwProcessId, BOOL fExpandRegions) {

    // очищаем окно списка и создаем в нем горизонтальную полосу прокрутки
    ListBox_ResetContent(hwndLB);
    ListBox_SetHorizontalExtent(hwndLB, 300 * LOWORD(GetDialogBaseUnits()));

    // выполняется ли еще процесс?
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, dwProcessId);

    if (hProcess == NULL) {
        ListBox_AddString(hwndLB, TEXT("")); // пустая строка (так лучше на вид)
        ListBox_AddString(hwndLB,
            TEXT("    The process ID identifies a process that is not running"));
        return;
    }

    // получаем новый снимок процесса
    g_toolhelp.CreateSnapshot(TH32CS_SNAPALL, dwProcessId);

    // просматриваем виртуальное адресное пространство, добавляя элементы в окно списка

```

Рис. 14-4. *продолжение*

```

    BOOL fOk = TRUE;
    PVOID pvAddress = NULL;

    SetWindowRedraw(hwndLB, FALSE);
    while (fOk) {

        VMQUERY vmq;
        fOk = VMQuery(hProcess, pvAddress, &vmq);

        if (fOk) {
            // формируем строку для вывода на экран и добавляем ее в окно списка
            TCHAR szLine[1024];
            ConstructRgnInfoLine(hProcess, &vmq, szLine, sizeof(szLine));
            ListBox_AddString(hwndLB, szLine);

            if (fExpandRegions) {
                for (DWORD dwBlock = 0; fOk && (dwBlock < vmq.dwRgnBlocks);
                    dwBlock++) {

                    ConstructBlkInfoLine(&vmq, szLine, sizeof(szLine));
                    ListBox_AddString(hwndLB, szLine);

                    // получаем адрес следующего региона
                    pvAddress = ((PBYTE) pvAddress + vmq.BlkSize);
                    if (dwBlock < vmq.dwRgnBlocks - 1) {
                        // нельзя запрашивать информацию о памяти за последним блоком
                        fOk = VMQuery(hProcess, pvAddress, &vmq);
                    }
                }
            }
            // получаем адрес следующего региона
            pvAddress = ((PBYTE) vmq.pvRgnBaseAddress + vmq.RgnSize);
        }
    }
    SetWindowRedraw(hwndLB, TRUE);
    CloseHandle(hProcess);

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

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

        chSETDLGICONS(hwnd, IDI_VMMAP);

        // показываем в заголовке окна, какой процесс мы просматриваем
        TCHAR szCaption[MAX_PATH * 2];
        GetWindowText(hwnd, szCaption, chDIMOF(szCaption));
        g_toolhelp.CreateSnapshot(TH32CS_SNAPALL, g_dwProcessId);
        PROCESSENTRY32 pe = { sizeof(pe) };
        wsprintf(&szCaption[lstrlen(szCaption)], TEXT(" (PID=%u \\"%s\\")"),
            g_dwProcessId, g_toolhelp.ProcessFind(g_dwProcessId, &pe)
                ? pe.szExeFile : TEXT("unknown"));
    }

```

см. след. стр.

Рис. 14-4. *продолжение*

```

SetWindowText(hwnd, szCaption);

// VMMar сообщает столько информации, что
// окно лучше сразу отмасштабировать по максимуму
ShowWindow(hwnd, SW_MAXIMIZE);

// принудительно обновляем окно списка
Refresh(GetDlgItem(hwnd, IDC_LISTBOX), g_dwProcessId, g_fExpandRegions);
return(TRUE);
}

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

void Dlg_OnSize(HWND hwnd, UINT state, int cx, int cy) {

    // окно списка всегда занимает всю клиентскую область
    SetWindowPos(GetDlgItem(hwnd, IDC_LISTBOX), 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;

        case ID_REFRESH:
            Refresh(GetDlgItem(hwnd, IDC_LISTBOX),
                g_dwProcessId, g_fExpandRegions);
            break;

        case ID_EXPANDREGIONS:
            g_fExpandRegions = g_fExpandRegions ? FALSE: TRUE;
            Refresh(GetDlgItem(hwnd, IDC_LISTBOX),
                g_dwProcessId, g_fExpandRegions);
            break;

        case ID_COPYTOCLIPBOARD:
            CopyControlToClipboard(GetDlgItem(hwnd, IDC_LISTBOX));
            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);
    }
}

```

Рис. 14-4. *продолжение*

```

        chHANDLE_DLGMSG(hwnd, WM_SIZE,      Dlg_OnSize);
    }
    return(FALSE);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    CToolhelp::EnableDebugPrivilege();

    // пытаемся загрузить PSAPI.DLL и получить адрес GetMappedFileName
    HMODULE hmodPSAPI = LoadLibrary(TEXT("PSAPI"));
    if (hmodPSAPI != NULL) {
#ifdef UNICODE
        g_pfnGetMappedFileName = (PFNGETMAPPEDFILENAME)
            GetProcAddress(hmodPSAPI, "GetMappedFileNameW");
#else
        g_pfnGetMappedFileName = (PFNGETMAPPEDFILENAME)
            GetProcAddress(hmodPSAPI, "GetMappedFileNameA");
#endif
    }

    g_dwProcessId = _ttoi(pszCmdLine);
    if (g_dwProcessId == 0) {
        g_dwProcessId = GetCurrentProcessId();
    }

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMMAP), NULL, Dlg_Proc);

    if (hmodPSAPI != NULL)
        FreeLibrary(hmodPSAPI); // выгружаем PSAPI.DLL, если мы ее загружали

    return(0);
}

///////////////////////////////////////////////////////////////// Конец файла ///////////////////////////////////////////////////////////////////

```


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 ($50 \times 1024 \times 1024$). 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 *pvAddress*, 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 ($300 \times 65\,536 + 8192$), система округлит этот адрес до ближайшего меньшего числа, кратного 64 Кб, и на самом деле зарезервирует регион по адресу 19 660 800 ($300 \times 65\,536$).

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 *pvAddress* передайте NULL, а в параметре *fdwAllocationType* — флаг MEM_RESERVE, скомбинированный с флагом MEM_TOP_DOWN.



В Windows 98 флаг MEM_TOP_DOWN игнорируется.

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 защиты региона совпадает с атрибутом защиты 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.



Windows 98 поддерживает лишь атрибуты защиты `PAGE_NOACCESS`, `PAGE_READONLY` и `PAGE_READWRITE`. Попытка резервирования региона с атрибутом `PAGE_EXECUTE` или `PAGE_EXECUTE_READ` дает регион с атрибутом `PAGE_READONLY`. А указав `PAGE_EXECUTE_READWRITE`, Вы получите регион с атрибутом `PAGE_READWRITE`.

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, что и при резервировании региона, хотя можно задать и другой.

Затем 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:

```
VirtualAlloc((PVOID) (5242880 + (2 * 1024)), 6 * 1024,
    MEM_COMMIT, 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 защиты `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_COMMIT, PAGE_READWRITE);
```

Этот вызов содержит запрос на выделение региона размером 99 Кб и передачу ему 99 Кб физической памяти. Обработывая 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. Оказавшись он недостаточным, функция *VirtualAlloc* вернула бы NULL.

Если системе удастся зарезервировать подходящий регион, она передает ему физическую память. И регион, и переданная память получают один атрибут защиты — в данном случае `PAGE_READWRITE`.

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, *VirtualAlloc* 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 ($200 \times 256 \times 128$) 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 расходом 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 изменении структуры CELLDATA придется вызывать функцию *VirtualAlloc*, что, естественно, скажется на скорости работы программы.
- Определять (с помощью *VirtualQuery*), 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 *Virtual-*

Alloc, 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 в 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 MEM_DECOMMIT.

Как и передача, возврат памяти осуществляется с учетом размерности страниц. Иначе говоря, задание адреса, указывающего на середину страницы, 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 уместится 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. приравняваете 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, *flnUse*) 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 *flnUse*. 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.

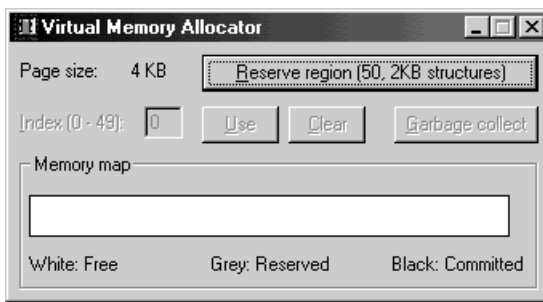
Функцию сбора мусора можно вызывать сразу после того, как необходимость в одной из структур отпадет, но делать так не стоит, поскольку функция каждый раз просматривает все структуры — и существующие, и те, которые могут быть созданы. Оптимальный путь — реализовать эту функцию как поток с бо-

лее 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.

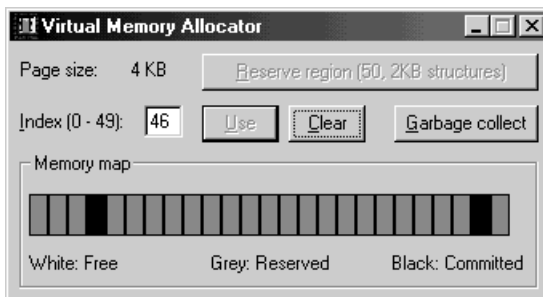
Программа-пример VMAlloc

Эта программа, «15 VMAlloc.exe» (см. листинг на рис. 15-1), демонстрирует применение механизма виртуальной памяти для управления массивом структур. Файлы исходного кода и ресурсов этой программы находятся в каталоге 15-VMAlloc на компакт-диске, прилагаемом к книге. 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 *VirtualAlloc* 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:

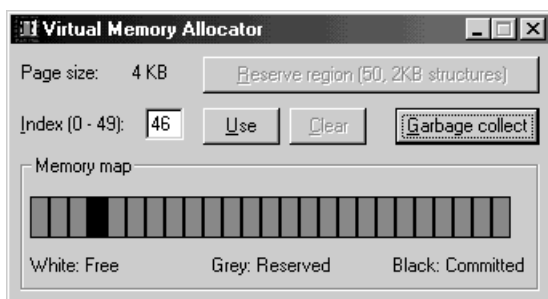


Любой элемент массива, помеченный как занятый, можно освободить щелчком кнопки Clear. Но это не приведет к возврату физической памяти, переданной под элемент массива. Дело в том, что каждая страница содержит несколько структур и освобождение одной структуры не влечет за собой освобождения других. Если бы память была возвращена, то пропали бы и данные, содержащиеся в остальных струк-

турах. 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 *flnUse* 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 уместиться 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 *flnUse*, 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

```

/*****
Модуль: VMAlloc.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

```

Рис. 15-1. Программа-пример VMAlloc

Рис. 15-1. *продолжение*

```

#include "..\CmnHdr.h"      /* см. приложение А */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"

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

// переменная для хранения размера страниц на данном процессоре
UINT g_uPageSize = 0;

// пример структуры данных, используемой для массива
typedef struct {
    BOOL fInUse;
    BYTE bOtherData[2048 - sizeof(BOOL)];
} SOMEDATA, *PSOMEDATA;

// число структур в массиве
#define MAX_SOMEDATA      (50)

// указатель на массив структур данных
PSOMEDATA g_pSomeData = NULL;

// прямоугольная область в окне, занимаемая картой памяти
RECT g_rcMemMap;

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

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

    chSETDLGICONS(hwnd, IDI_VMALLOC);

    // инициализируем диалоговое окно с отключением
    // пока недоступных элементов управления
    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);

    // получаем координаты поля вывода карты памяти
    GetWindowRect(GetDlgItem(hwnd, IDC_MEMMAP), &g_rcMemMap);
    MapWindowPoints(NULL, hwnd, (LPPPOINT) &g_rcMemMap, 2);

    // уничтожаем временное окно, которое определяет позицию
    // поля вывода для карты памяти
    DestroyWindow(GetDlgItem(hwnd, IDC_MEMMAP));

    // выводим в диалоговое окно размер страницы (просто для сведения)
    TCHAR szBuf[10];
    wsprintf(szBuf, TEXT("(%d KB)"), g_uPageSize / 1024);
    SetDlgItemText(hwnd, IDC_PAGESIZE, szBuf);

```

см. след. стр.

Рис. 15-1. *продолжение*

```

// инициализируем поле ввода
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) {

    static DWORD s_uPageSize = 0;

    if (s_uPageSize == 0) {
        // получаем размер страниц на данном процессоре
        SYSTEM_INFO si;
        GetSystemInfo(&si);
        s_uPageSize = si.dwPageSize;
    }

    UINT uMaxPages = dwNum * dwStructSize / g_uPageSize;

    for (UINT uPage = 0; uPage < uMaxPages; uPage++) {
        BOOL fAnyAllocsInThisPage = 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));
            fAnyAllocsInThisPage = ((mbi.State == MEM_COMMIT) &&
                * (PBOOL) ((PBYTE) pvBase + dwStructSize * uIndex));

            // прекращаем проверку этой страницы, так как ее нельзя вернуть
            if (fAnyAllocsInThisPage) break;
        }

        if (!fAnyAllocsInThisPage) {
            // на этой странице нет структур, возвращаем ее
            VirtualFree(&g_pSomeData[uIndexLast - 1], dwStructSize, MEM_DECOMMIT);
        }
    }
}

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

```

Рис. 15-1. *продолжение*

```

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:
            // резервируем адресное пространство,
            // достаточное для размещения массива структур
            g_pSomeData = (PSOMEDATA) VirtualAlloc(NULL,
                MAX_SOMEDATA * sizeof(SOMEDATA), MEM_RESERVE, PAGE_READWRITE);

            // отключаем кнопку Reserve и включаем остальные
            // элементы управления
            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);

            // переводим фокус в поле ввода индекса
            SetFocus(GetDlgItem(hwnd, IDC_INDEX));

            // объявляем поле вывода карты памяти недействительным (для его перерисовки)
            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 fOk = (mbi.State == MEM_COMMIT);
                if (fOk)
                    fOk = g_pSomeData[uIndex].fInUse;

                EnableWindow(GetDlgItem(hwnd, IDC_USE), !fOk);
                EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), fOk);

            } else {
                EnableWindow(GetDlgItem(hwnd, IDC_USE), FALSE);
                EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);
            }
            break;
    }
}

```

см. след. стр.

Рис. 15-1. *продолжение*

```

case IDC_USE:
    uIndex = GetDlgItemInt(hwnd, IDC_INDEX, NULL, FALSE);
    if (chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {

        // Примечание: новые страницы всегда обнуляются системой
        VirtualAlloc(&g_pSomeData[uIndex], sizeof(SOMEDATA),
            MEM_COMMIT, PAGE_READWRITE);

        g_pSomeData[uIndex].fInUse = TRUE;

        EnableWindow(GetDlgItem(hwnd, IDC_USE), FALSE);
        EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), TRUE);

        // переводим фокус на кнопку Clear
        SetFocus(GetDlgItem(hwnd, IDC_CLEAR));

        // объявляем поле вывода карты памяти недействительным
        InvalidateRect(hwnd, &g_rcMemMap, FALSE);
    }
    break;

case IDC_CLEAR:
    uIndex = GetDlgItemInt(hwnd, IDC_INDEX, NULL, FALSE);
    if (chINRANGE(0, uIndex, MAX_SOMEDATA - 1)) {
        g_pSomeData[uIndex].fInUse = FALSE;
        EnableWindow(GetDlgItem(hwnd, IDC_USE), TRUE);
        EnableWindow(GetDlgItem(hwnd, IDC_CLEAR), FALSE);

        // переводим фокус на кнопку Use
        SetFocus(GetDlgItem(hwnd, IDC_USE));
    }
    break;

case IDC_GARBAGECOLLECT:
    GarbageCollect(g_pSomeData, MAX_SOMEDATA, sizeof(SOMEDATA));

    // объявляем поле вывода карты памяти недействительным
    InvalidateRect(hwnd, &g_rcMemMap, FALSE);
    break;
}
}

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

void Dlg_OnPaint(HWND hwnd) { // перерисовывает карту памяти

    PAINTSTRUCT ps;
    BeginPaint(hwnd, &ps);

    UINT uMaxPages = MAX_SOMEDATA * sizeof(SOMEDATA) / g_uPageSize;
    UINT uMemMapWidth = g_rcMemMap.right - g_rcMemMap.left;

```

Рис. 15-1. *продолжение*

```

    if (g_pSomeData == NULL) {

        // память еще предстоит зарезервировать
        Rectangle(ps.hdc, g_rcMemMap.left, g_rcMemMap.top,
            g_rcMemMap.right - uMemMapWidth % uMaxPages, g_rcMemMap.bottom);

    } else {

        // проходим виртуальное адресное пространство, создавая карту памяти
        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);
}

```

см. след. стр.

Рис. 15-1. *продолжение*

```

////////////////////////////////////
int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

    // получаем размер страниц для данного процессора
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    g_uPageSize = si.dwPageSize;

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_VMALLOC), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конец файла //////////////////////////////////

```

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 *VirtualProtect*:

```

BOOL VirtualProtect(
    PVOID pvAddress,
    SIZE_T dwSize,
    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.

WINDOWS 98 Windows 98 поддерживает лишь атрибуты защиты PAGE_NOACCESS, PAGE_READ-ONLY и PAGE_READWRITE. Попытка изменить атрибут защиты страницы на PAGE_EXECUTE или PAGE_EXECUTE_READ приведет к тому, что эта область памяти получит атрибут PAGE_READONLY. А указав атрибут PAGE_EXECUTE_READWRITE, Вы получите страницу с атрибутом PAGE_READWRITE.

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

WINDOWS 98 Windows 98 не поддерживает сброс 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 2000 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 неизменяющуюся, 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 pData = (PINT) VirtualAlloc(NULL, 1024,
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
p[0] = 100;
p[1] = 200;
VirtualAlloc((PVOID) pData, 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). 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 значения даст 0, а эта величина недопустима.

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:

```
PVOID pMem = 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, «15 MemReset.exe» (см. листинг на рис. 15-2), демонстрирует, как работает флаг MEM_RESET. Файлы исходного кода и ресурсов этой программы находятся в каталоге 15-MemReset на компакт-диске, прилагаемом к книге.

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 *lstrcpy* 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

```

/*****
Модуль: MemReset.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <tchar.h>

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

    chWindows9xNotAllowed();

    TCHAR szAppName[] = TEXT("MEM_RESET tester");
    TCHAR szTestData[] = TEXT("Some text data");

    // передаем страницу памяти и модифицируем ее содержимое
    LPTSTR pszData = (LPTSTR) VirtualAlloc(NULL, 1024,
        MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
    lstrcpy(pszData, szTestData);

```

Рис. 15-1. Программа-пример *MemReset*

см. след. стр.

Рис. 15-1. *продолжение*

```

if (MessageBox(NULL, TEXT("Do you want to access this data later?"),
    szAppName, MB_YESNO) == IDNO) {

    // Мы хотим сохранить эту страницу физической памяти в нашем
    // процессе, но ее данные нас больше не интересуют.
    // Скажем системе, что данные на этой странице не изменялись.

    // Примечание: поскольку MEM_RESET разрушает данные, VirtualAlloc округляет
    // параметры с базовым адресом и размером до наиболее безопасных значений.
    // Вот пример:
    // VirtualAlloc(pvData, 5000, MEM_RESET, PAGE_READWRITE)
    // сбросит 0 страниц на процессорах с размером страниц более 4 Кб
    // и 1 страницу на процессорах с четырехкилобайтовыми страницами.
    // Поэтому, чтобы вызов VirtualAlloc всегда был успешным, надо
    // сначала вызвать VirtualQuery и определить точный размер страницы.

    MEMORY_BASIC_INFORMATION mbi;
    VirtualQuery(pszData, &mbi, sizeof(mbi));
    VirtualAlloc(pszData, mbi.RegionSize, MEM_RESET, PAGE_READWRITE);
}

// передаем объем памяти, равный размеру оперативной памяти
MEMORYSTATUS mst;
GlobalMemoryStatus(&mst);
PVOID pvDummy = VirtualAlloc(NULL, mst.dwTotalPhys,
    MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);

// изменяем содержимое всех страниц в регионе, чтобы все страницы,
// модифицированные в оперативной памяти, записывались в страничный файл
ZeroMemory(pvDummy, mst.dwTotalPhys);

// сравниваем нашу страницу данных с тем,
// что было записано туда изначально
if (lstrcmp(pszData, szTestData) == 0) {

    // Данные на этой странице совпали с тем, что мы туда записывали.
    // Функция ZeroMemory заставила систему записать нашу
    // страницу в страничный файл.
    MessageBox(NULL, TEXT("Modified data page was saved."),
        szAppName, MB_OK);
} else {

    // Данные на этой странице не совпадают с тем, что мы туда записывали.
    // ZeroMemory не заставила систему записать измененную страницу
    // в страничный файл.
    MessageBox(NULL, TEXT("Modified data page was NOT saved."),
        szAppName, MB_OK);
}
return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

Механизм Address Windowing Extensions (только Windows 2000)

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 2000 предлагает новый механизм — Address Windowing Extensions (AWE). Создавая AWE, Microsoft 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 микросекунд).

Через одно адресное окно одновременно 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;
GetSystemInfo(&sinf);

// вычисляем, сколько страниц памяти нужно для нашего количества байтов
ULONG_PTR ulRAMPages = (ulRAMBytes + sinf.dwPageSize - 1) / sinf.dwPageSize;

// создаем соответствующий массив для номеров фреймов страниц
ULONG_PTR aRAMPages[ulRAMPages];

// выделяем страницы оперативной памяти (в полномочиях пользователя
// должна быть разрешена блокировка страниц в памяти)
AllocateUserPhysicalPages(
    GetCurrentProcess(), // выделяем память для нашего процесса
    &ulRAMPages,          // на входе: количество запрошенных страниц RAM,
                        // на выходе: количество выделенных страниц RAM
```

см. след. стр.

```

aRAMPages);          // на выходе: специфический массив,
                      // идентифицирующий выделенные страницы

// 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(
    GetCurrentProcess(), // освобождаем RAM, выделенную нашему процессу
    &ulRAMPages,          // на входе: количество страниц RAM,
                        // на выходе: количество освобожденных страниц RAM
    aRAMPages);          // на входе: массив, идентифицирующий освобождаемые
                        // страницы RAM

// уничтожаем адресное окно
VirtualFree(pvWindow, 0, MEM_RELEASE);

```

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 MEM_PHYSICAL — 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_READWRITE. 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 бо-

лее 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.



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.

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.



Функция *MapUserPhysicalPages* отключает текущий блок оперативной памяти от адресного окна, если вместо параметра *aRAMPages* передается NULL. Вот пример:

```
// отключаем текущий блок RAM от адресного окна
MapUserPhysicalPages(pvWindow, ulRAMPages, NULL);
```

WINDOWS 2000

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 *FreeUserPhysicalPages*:

```
BOOL FreeUserPhysicalPages(
    HANDLE hProcess,
    PULONG_PTR puLRAMPages,
    PULONG_PTR aRAMPages);
```

В Windows 2000 право «Lock Pages in Memory» активизируется так:

1. Запустите консоль Computer Management MMC. Для этого щелкните кнопку Start, выберите команду Run, введите «compmgmt.msc /a» и щелкните кнопку OK.
2. Если в левой секции нет элемента Local Computer Policy, выберите из меню Console команду Add/Remove Snap-in. На вкладке Standalone в списке Snap-ins Added To укажите строку Computer Management (Local). Теперь щелкните кнопку Add, чтобы открыть диалоговое окно Add Standalone Snap-in. В списке Available Standalone Snap-ins укажите Select Group Policy и выберите кнопку Add. В диалоговом окне Select Group Policy Object просто щелкните кнопку Finish. Наконец, в диалоговом окне Add Standalone Snap-in щелкните кнопку Close, а в диалоговом окне Add/Remove Snap-in — кнопку OK. После этого в левой секции консоли Computer Management должен появиться элемент Local Computer Policy.
3. В левой секции консоли последовательно раскройте следующие элементы: Local Computer Policy, Computer Configuration, Windows Settings, Security Settings и Local Policies. Выберите User Rights Assignment.
4. В правой секции выберите атрибут Lock Pages in Memory.
5. Выберите из меню Action команду Select Security, чтобы открыть диалоговое окно Lock Pages in Memory. Щелкните кнопку Add. В диалоговом окне Select Users or Groups добавьте пользователей и/или группы, которым Вы хотите разрешить блокировку страниц в памяти. Затем закройте все диалоговые окна, щелкая в каждом из них кнопку OK.

Новые права вступают в силу при следующей регистрации в системе. Если Вы только что сами себе предоставили право «Lock Pages in Memory», выйдите из системы и вновь зарегистрируйтесь в ней.

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 2000 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» (см. листинг на рис. 15-3), 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 на компакт-диске, прилагаемом к книге. 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, CAddrWindowStorage, 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 CAddrWindowStorage. Остальной код просто вызывает нужные методы в нужное время.



AWE.cpp

```

/*****
Модуль: AWE.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <Windowsx.h>
#include <tchar.h>
#include "AddrWindow.h"
#include "Resource.h"

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

CAddrWindow g_aw[2];        // два адресных окна
CAddrWindowStorage g_aws[2]; // два блока памяти
const ULONG_PTR g_nChars = 1024; // буфер на 1024 символа

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

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

    chSETDLGICONS(hwnd, IDI_AWE);

    // создаем два адресных окна
    chVERIFY(g_aw[0].Create(g_nChars * sizeof(TCHAR)));
    chVERIFY(g_aw[1].Create(g_nChars * sizeof(TCHAR)));

    // создаем два блока памяти
    if (!g_aws[0].Allocate(g_nChars * sizeof(TCHAR))) {
        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)));

    // помещаем в первый блок текст по умолчанию
    g_aws[0].MapStorage(g_aw[0]);
    lstrcpy((PSTR) (PVOID) g_aw[0], TEXT("Text in Storage 0"));

    // помещаем во второй блок текст по умолчанию
    g_aws[1].MapStorage(g_aw[0]);
    lstrcpy((PSTR) (PVOID) g_aw[0], TEXT("Text in Storage 1"));

    // заполняем элементы управления диалогового окна
    for (int n = 0; n <= 1; n++) {
        // настраиваем поле со списком для каждого адресного окна
    }
}

```

Рис. 15-3. Программа-пример AWE

Рис. 15-3. *продолжение*

```

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"));

// окно 0 показывает Storage 0, а окно 1 – 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) {
            // показываем для каждого окна другой блок
            int nWindow = ((id == IDC_WINDOW0STORAGE) ? 0 : 1);
            int nStorage = ComboBox_GetCurSel(hwndCtl) - 1;
            if (nStorage == -1) { // с этим окном блок памяти не связан
                chVERIFY(g_aw[nWindow].UnmapStorage());
            } else {
                if (!g_aws[nStorage].MapStorage(g_aw[nWindow])) {
                    // не удалось связать блок с окном
                    chVERIFY(g_aw[nWindow].UnmapStorage());
                    ComboBox_SetCurSel(hwndCtl, 0); // устанавливаем "No storage"
                    chMB("This storage can be mapped only once.");
                }
            }
        }

        // обновляем текст в поле, относящемся к адресному окну
        HWND hwndText = GetDlgItem(hwnd,
            ((nWindow == 0) ? IDC_WINDOW0TEXT : IDC_WINDOW1TEXT));
        MEMORY_BASIC_INFORMATION mbi;
        VirtualQuery(g_aw[nWindow], &mbi, sizeof(mbi));
        // Примечание: mbi.State == MEM_RESERVE, если с адресным окном
        // не связан блок памяти
        EnableWindow(hwndText, (mbi.State == MEM_COMMIT));
    }
}

```

см. след. стр.

Рис. 15-3. *продолжение*

```

        Edit_SetText(hwndText, IsWindowEnabled(hwndText)
            ? (PCSTR) (PVOID) g_aw[nWindow] : TEXT("(No storage)"));
    }
    break;
case IDC_WINDOW0TEXT:
case IDC_WINDOW1TEXT:
    if (codeNotify == EN_CHANGE) {
        // обновляем память, связанную с этим адресным окном
        int nWindow = ((id == IDC_WINDOW0TEXT) ? 0 : 1);
        Edit_GetText(hwndCtl, (PSTR) (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 pszCmdLine, int) {

    chWindows2000Required();

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_AWE), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конец файла //////////////////////////////////////

```

AddrWindow.h

```

/*****
Модуль: AddrWindow.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#pragma once

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

```

Рис. 15-3. *продолжение*

```

#include "..\CmnHdr.h"      /* см. приложение А */
#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) {
        // резервируем регион для адресного окна
        m_pvWindow = VirtualAlloc(pvPreferredWindowBase, dwBytes,
            MEM_RESERVE | MEM_PHYSICAL, PAGE_READWRITE);
        return(m_pvWindow != NULL);
    }

    BOOL Destroy() {
        BOOL fOk = TRUE;
        if (m_pvWindow != NULL) {
            // удаляем регион, выделенный для адресного окна
            fOk = VirtualFree(m_pvWindow, 0, MEM_RELEASE);
            m_pvWindow = NULL;
        }
        return(fOk);
    }

    BOOL UnmapStorage() {
        // отключаем всю память от адресного окна
        MEMORY_BASIC_INFORMATION mbi;
        VirtualQuery(m_pvWindow, &mbi, sizeof(mbi));
        return(MapUserPhysicalPages(m_pvWindow,
            mbi.RegionSize / sm_sinf.dwPageSize, NULL));
    }

    // возвращаем виртуальный адрес адресного окна
    operator PVOID() { return(m_pvWindow); }

private:
    PVOID m_pvWindow; // виртуальный адрес адресного окна
    static CSystemInfo sm_sinf;
};

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

```

см. след. стр.

Рис. 15-3. *продолжение*

```

CSystemInfo CAddrWindow::sm_sinf;

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

class CAddrWindowStorage {
public:
    CAddrWindowStorage() { m_ulPages = 0; m_pulUserPfnArray = NULL; }
    ~CAddrWindowStorage() { Free(); }

    BOOL Allocate(ULONG_PTR ulBytes) {
        // выделяем память, предназначенную для адресного окна

        Free(); // очищаем существующее адресное окно,
                // принадлежащее этому объекту

        // вычисляем количество страниц по числу байтов
        m_ulPages = (ulBytes + sm_sinf.dwPageSize) / sm_sinf.dwPageSize;

        // создаем массив для номеров фреймов страниц
        m_pulUserPfnArray = (PULONG_PTR)
            HeapAlloc(GetProcessHeap(), 0, m_ulPages * sizeof(ULONG_PTR));

        BOOL fOk = (m_pulUserPfnArray != NULL);
        if (fOk) {
            // должно быть предоставлено право "Lock Pages in Memory"
            EnablePrivilege(SE_LOCK_MEMORY_NAME, TRUE);
            fOk = AllocateUserPhysicalPages(GetCurrentProcess(),
                &m_ulPages, m_pulUserPfnArray);
            EnablePrivilege(SE_LOCK_MEMORY_NAME, FALSE);
        }
        return(fOk);
    }

    BOOL Free() {
        BOOL fOk = TRUE;
        if (m_pulUserPfnArray != NULL) {
            fOk = FreeUserPhysicalPages(GetCurrentProcess(),
                &m_ulPages, m_pulUserPfnArray);
            if (fOk) {
                // освобождаем массив для номеров фреймов страниц
                HeapFree(GetProcessHeap(), 0, m_pulUserPfnArray);
                m_ulPages = 0;
                m_pulUserPfnArray = NULL;
            }
        }
        return(fOk);
    }

    ULONG_PTR HowManyPagesAllocated() { return(m_ulPages); }

```

Рис. 15-3. *продолжение*

```

    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 fEnable = TRUE) {

        BOOL fOk = FALSE; // считаем, что функция потерпит неудачу
        HANDLE hToken;

        // пытаемся открыть маркер доступа у этого процесса
        if (OpenProcessToken(GetCurrentProcess(),
            TOKEN_ADJUST_PRIVILEGES, &hToken)) {

            // пытаемся разрешить блокировать страницы в памяти
            TOKEN_PRIVILEGES tp = { 1 };
            LookupPrivilegeValue(NULL, pszPrivName, &tp.Privileges[0].Luid);
            tp.Privileges[0].Attributes = fEnable ? SE_PRIVILEGE_ENABLED : 0;
            AdjustTokenPrivileges(hToken, FALSE, &tp, sizeof(tp), NULL, NULL);
            fOk = (GetLastError() == ERROR_SUCCESS);
            CloseHandle(hToken);
        }
        return(fOk);
    }

private:
    ULONG_PTR m_ulPages;           // число страниц памяти
    PULONG_PTR m_pulUserPfnArray; // массив для номеров фреймов страниц

private:
    static CSystemInfo sm_sinf;
};

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

CSystemInfo CAddrWindowStorage::sm_sinf;

//////////////////////////////////// Конеч файл //////////////////////////////////

```

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`:

```
/STACK:reserve[,commit]
```

Тогда при создании стека потока система зарезервирует регион адресного пространства, размер которого указан в параметре `/STACK` компоновщика. 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, указанное в параметре `/STACK`. 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).

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.


| Адрес | Состояние страницы |
|---|--|
| 0x080FF000 | Верхняя часть стека (переданная страница) |
| 0x080FE000 | Переданная страница с флагом PAGE_GUARD |
| 0x080FD000 | Зарезервированная страница |
|  | |
| 0x08003000 | Зарезервированная страница |
| 0x08002000 | Зарезервированная страница |
| 0x08001000 | Зарезервированная страница |
| 0x08000000 | Нижняя часть стека (зарезервированная страница) |

Рис. 16-1. Так выглядит регион стека потока сразу после его создания


| Адрес | Состояние страницы |
|---|--|
| 0x080FF000 | Верхняя часть стека (переданная страница) |
| 0x080FE000 | Переданная страница |
| 0x080FD000 | Переданная страница |
|  | |
| 0x08003000 | Переданная страница |
| 0x08002000 | Переданная страница с флагом PAGE_GUARD |
| 0x08001000 | Зарезервированная страница |
| 0x08000000 | Нижняя часть стека (зарезервированная страница) |

Рис. 16-2. Почти заполненный регион стека потока

| Адрес | Состояние страницы |
|------------|--|
| 0x080FF000 | Верхняя часть стека (переданная страница) |
| 0x080FE000 | Переданная страница |
| 0x080FD000 | Переданная страница |
| | |
| 0x08003000 | Переданная страница |
| 0x08002000 | Переданная страница |
| 0x08001000 | Переданная страница |
| 0x08000000 | Нижняя часть стека (зарезервированная страница) |

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 возьмет управление на себя и завершит не только данный поток, но и весь процесс. И даже не сообщит об этом пользователю; процесс просто исчезнет!

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 суме-

ла 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.

Стек потока в Windows 98

В Windows 98 стеки ведут себя почти так же, как и в Windows 2000. Но отличия все же есть.

На рис. 16-4 показано, как в Windows 98 может выглядеть регион стека (зарезервированный с адреса 0x00530000) размером 1 Мб.

| Адрес | Размер | Состояние страницы |
|------------|-----------------------------------|---|
| 0x00640000 | 16 страниц (65 536 байтов) | Верхняя часть стека (зарезервирована для перехвата обращений к несуществующей области стека) |
| 0x0063F000 | 1 страница (4096 байтов) | Переданная страница с атрибутом PAGE_READWRITE (задействованная область стека) |
| 0x0063E000 | 1 страница (4096 байтов) | Страница с атрибутом PAGE_NOACCESS (заменяет флаг PAGE_GUARD) |
| 0x00638000 | 6 страниц (24 576 байтов) | Страницы, зарезервированные для перехвата переполнения стека |
| 0x00637000 | 1 страница (4096 байтов) | Переданная страница с атрибутом PAGE_READWRITE (для совместимости с 16-разрядными компонентами) |
| 0x00540000 | 247 страниц (1 011 712 байтов) | Страницы, зарезервированные для дальнейшего расширения стека |
| 0x00530000 | 16 страниц (65 536 байтов) | Нижняя часть стека (зарезервирована для перехвата переполнения стека) |

Рис. 16-4. Так выглядит регион стека сразу после его создания под управлением Windows 98

Во-первых, размер региона на самом деле 1 Мб плюс 128 Кб, хотя мы хотели создать стек объемом всего 1 Мб. В Windows 98 система резервирует под стек на 128 Кб больше, чем было запрошено. Собственно стек располагается в середине этого региона, а по обеим его границам размещаются блоки по 64 Кб каждый.

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,
    PSTR pszCmdLine, int nCmdShow) {

    char szBuf[100];
    szBuf[10000] = 0; // обращение к несуществующей области стека

    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, но, если приложение работает под управлением Win-

dows 98, выполнение этого оператора вызовет нарушение доступа. Это одна из приятных особенностей Windows 98, отсутствующих в Windows 2000, в которой сразу за стеком потока может быть расположен 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*.

Второе отличие: в стеке нет страниц с флагом атрибутов защиты PAGE_GUARD. Поскольку Windows 98 такой флаг не поддерживает, при расширении стека потока она действует несколько иначе. Она помечает страницу переданной памяти, располагаемой под стеком, атрибутом PAGE_NOACCESS (на рис. 16-4 — по адресу 0x0063E000). Когда поток обращается к этой странице, происходит нарушение доступа. Система перехватывает это исключение, меняет атрибут защиты страницы с PAGE_NOACCESS на PAGE_READWRITE и передает память новой «сторожевой» странице, размещаемой сразу за предыдущей.

Третье: обратите внимание на единственную страницу с атрибутом PAGE_READWRITE по адресу 0x00637000. Она создается для совместимости с 16-разрядной Windows. Хотя Microsoft нигде не говорит об этом, разработчики обнаружили, что первые 16 байтов сегмента стека 16-разрядной программы содержат информацию о ее стеке, локальной куче и локальной таблице атомарного доступа. Поскольку Win32-приложения в Windows 98 часто обращаются к 16-разрядным DLL и некоторые из этих DLL предполагают наличие тех самых 16 байтов в начале сегмента стека, Microsoft пришлось эмулировать подобные данные и в Windows 98. Когда 32-разрядный код обращается к 16-разрядному, Windows 98 отображает 16-битный селектор процессора на 32-разрядный стек и записывает в регистр сегмента стека (SS) такое значение, чтобы он указывал на страницу по адресу 0x00637000. И тогда 16-разрядный код, получив доступ к своим 16 байтам в начале сегмента стека, продолжает выполнение без всяких проблем.

По мере роста стека потока, выполняемого под управлением Windows 98, блок памяти по адресу 0x0063F000 постепенно увеличивается, а сторожевая страница смещается вниз до тех пор, пока не будет достигнут предел в 1 Мб, после чего она исчезает так же, как и в Windows 2000. Одновременно система смещает позицию страницы, предназначенной для совместимости с компонентами 16-разрядной Windows, и она, в конце концов, попадает в 64-килобайтовый блок, расположенный в начале региона стека. Поэтому целиком заполненный стек в Windows 98 выглядит так, как показано на рис. 16-5.

| Адрес | Размер | Состояние страницы |
|------------|-------------------------------|---|
| 0x00640000 | 16 страниц (65 536 байтов) | Верхняя часть стека (зарезервирована для перехвата обращений к несуществующей области стека) |
| 0x00540000 | 256 страниц (1 Мб) | Переданная страница с атрибутом PAGE_READWRITE (задействованная область стека) |
| 0x00539000 | 7 страниц (28 672 байта) | Страницы, зарезервированные для перехвата переполнения стека |
| 0x00538000 | 1 страница (4096 байтов) | Переданная страница с атрибутом PAGE_READWRITE (для совместимости с 16-разрядными компонентами) |
| 0x00530000 | 8 страниц (32 768 байтов) | Нижняя часть стека (зарезервирована для перехвата переполнения стека) |

Рис. 16-5. Целиком заполненный регион стека потока в Windows 98

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 x86 и 8 Кб для Alpha). 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_ALPHA
#define PAGE_SIZE (8 * 1024) // страницы по 8 Кб
#else
#define PAGE_SIZE (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 >= PAGE_SIZE) {
        // смещаем страницу вниз по стеку - должна быть сторожевой
        pbStackPtr -= PAGE_SIZE;
```

см. след. стр.

```
// 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 -= PAGE_SIZE;
}

// перед возвратом управления функция StackCheck устанавливает регистр
// указателя стека на адрес, следующий за локальными переменными функции
}
```

В компиляторе Microsoft Visual C++ предусмотрен параметр, позволяющий контролировать пороговый предел числа страниц, начиная с которого компилятор автоматически вставляет в программу вызов функции *StackCheck*. Используйте этот параметр, только если Вы точно знаете, что делаете, и если это действительно нужно. В 99,99999 процентах из ста приложения и DLL не требуют применения упомянутого параметра.

Программа-пример Summation

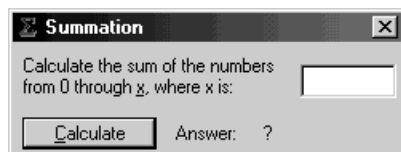
Эта программа, «16 Summation.exe» (см. листинг на рис. 16-6), демонстрирует использование фильтров и обработчиков исключений для корректного восстановления после переполнения стека. Файлы исходного кода и ресурсов этой программы находятся в каталоге 16-Summation на компакт-диске, прилагаемом к книге. 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:

$$\text{Sum} = (x * (x + 1)) / 2;$$

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 дескриптор 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 три причины.

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.

И последнее, почему я использую отдельный поток: 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 не-

сколько 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.



Summation.cpp

```

/*****
Модуль: Summation.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <windowsx.h>
#include <limits.h>
#include <process.h>      // для доступа к _beginthreadex
#include <tchar.h>
#include "Resource.h"

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

// Пример вызова Sum для uNum от 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);

```

Рис. 16-6. Программа-пример *Summation*

Рис. 16-6. *продолжение*

```

// 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 целые 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 чисел, которые
            // пользователь хочет просуммировать
            UINT uSum = GetDlgItemInt(hwnd, IDC_SUMNUM, NULL, FALSE);

            // создаем поток (с собственным стеком), отвечающий за суммирование
            DWORD dwThreadId;
            HANDLE hThread = chBEGINTHREADEX(NULL, 0,
                SumThreadFunc, (PVOID) (UINT_PTR) uSum, 0, &dwThreadId);

```

см. след. стр.

Рис. 16-6. *продолжение*

```
// 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 pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_SUMMATION), NULL, Dlg_Proc);
    return(0);
}

///////////////////////////////////////////////////////////////// Конец файла ///////////////////////////////////////////////////////////////////
```

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 2000 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 удастся (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 98 эта операция осуществляется по мере подкачки страниц в оперативную память. Но в Windows 2000 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 2000 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.

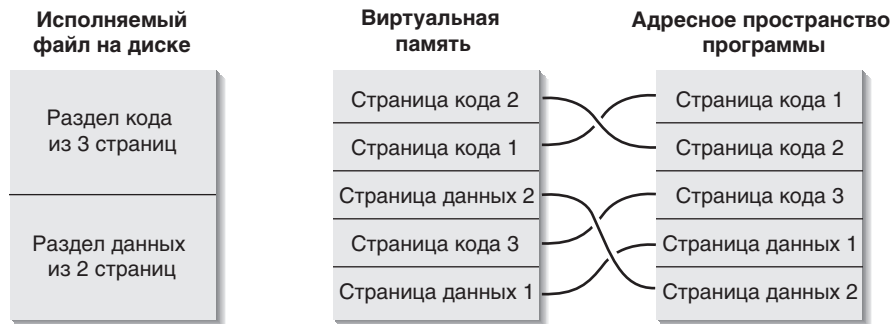
Если система почему-либо не свяжет EXE-файл с необходимыми ему DLL, на экране появится соответствующее сообщение, а адресное пространство процесса и объект «процесс» будут освобождены. При этом *CreateProcess* вернет FALSE; прояснить причину сбоя поможет функция *GetLastError*.

После увязки EXE- и DLL-файлов с адресным пространством процесса начинается исполняемый 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.

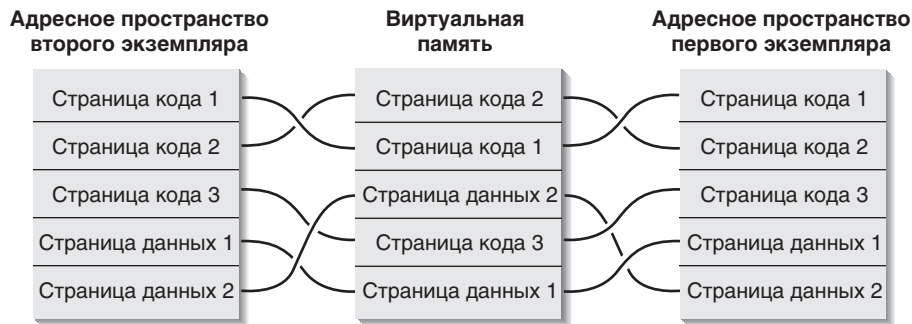
Статические данные не разделяются несколькими экземплярами EXE или DLL

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-файле¹. Вот упрощенная иллюстрация того, как код и данные приложения загружаются в виртуальную память, а затем отображаются на адресное пространство процесса:



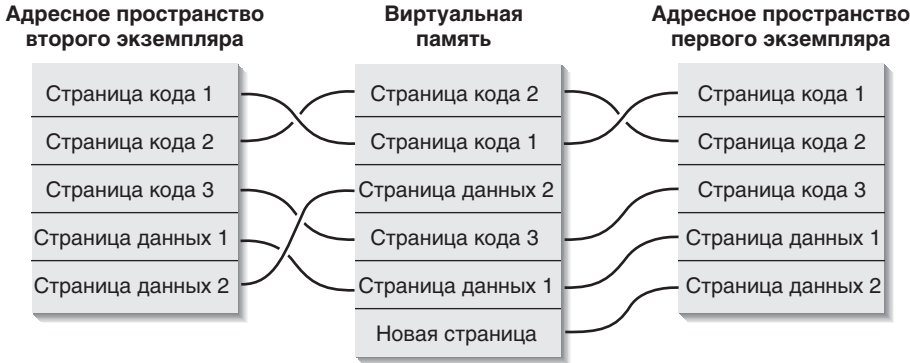
Теперь допустим, что запущен второй экземпляр программы. Система просто-напросто проецирует страницы виртуальной памяти, содержащие код и данные файла, на адресное пространство второго экземпляра приложения:



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.

¹ На самом деле содержимое файла разбито на отдельные разделы (sections). Код находится в одном разделе, а глобальные переменные — в другом. Разделы выравниваются по границам страниц. Приложение определяет размер страницы через функцию `GetSystemInfo`. В EXE- или DLL-файле раздел кода обычно предшествует разделу данных.

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.

WINDOWS
98

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.

Поведение Windows 2000 и Windows 98 в подобных случаях одинаково, кроме ситуации, когда в память загружено два экземпляра одного модуля и никаких данных не изменено. Тогда процессы под управлением Windows 2000 могут совместно использовать данные, а в Windows 98 каждый процесс получает свою копию этих данных. Но если в память загружен лишь один экземпляр модуля или же данные были модифицированы (что чаще всего и бывает), Windows 2000 и Windows 98 ведут себя одинаково.

Статические данные разделяются несколькими экземплярами EXE или DLL

По умолчанию для большей безопасности глобальные и статические данные не 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.

| 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
```

см. след. стр.

```
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 Only
```

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
580 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 #5

```
.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.

| Имя раздела | Описание |
|-------------|---|
| .bss | Неинициализированные данные |
| .CRT | Неизменяемые данные библиотеки C |
| .data | Инициализированные данные |
| .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 | Настроенная 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("имя_раздела")
```

Например, можно создать раздел Shared, в котором содержится единственная переменная типа LONG:

```
#pragma data_seg("Shared")
LONG 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_InstanceCount`. 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")
LONG g_InstanceCount;
#pragma data_seg()
```

Однако в компиляторе Microsoft Visual C++ 6.0 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;

// сообщаем 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 в 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:

`/SECTION: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 *.directve*. 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, `Microsoft` 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.

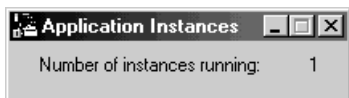
Все выглядит вполне невинно. В конце концов только Ваши приложения загружают данную DLL, и только они знают, где искать пароль, содержащийся в общем разделе памяти. Но хакеры не дремлют, и если им захочется узнать Ваш пароль, то максимум, что им понадобится, — написать небольшую программу, загружающую 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.

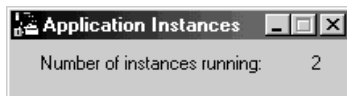
Программа-пример `AppInst`

Эта программа, «17 `AppInst.exe`» (см. листинг на рис. 17-1), демонстрирует, как выяснить, сколько экземпляров приложения уже выполняется в системе. Файлы исходного кода и ресурсов этой программы находятся в каталоге `17-AppInst` на компакт-дис-

ке, прилагаемом к книге. После запуска AppInst на экране появляется диалоговое окно, в котором сообщается, что сейчас выполняется только один ее 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_aMsgAppInstCountUpdate*):

```
PostMessage(HWND_BROADCAST, g_aMsgAppInstCountUpdate, 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*).



ApplInst.cpp

```

/*****
Модуль: AppInst.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"

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

// общесистемное оконное сообщение с уникальным идентификатором
UINT g_uMsgAppInstCountUpdate = INVALID_ATOM;

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

// 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")

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

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

    chSETDLGICONS(hwnd, IDI_APPINST);

    // инициализируем статический элемент управления
    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;
    }
}

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

```

Рис. 17-1. Программа-пример ApplInst

см. след. стр

Рис. 17-1. *продолжение*

```

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, LPTSTR pszCmdLine, 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((PLONG) &g_lApplicationInstances, 1);

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_APPINST), NULL, Dlg_Proc);

    // данный экземпляр закрывается
    InterlockedExchangeAdd((PLONG) &g_lApplicationInstances, -1);

    // сообщаем об этом остальным экземплярам программы
    PostMessage(HWND_BROADCAST, g_uMsgAppInstCountUpdate, 0, 0);

    return(0);
}

//////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////// Конец файла

```

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 первом методе: на каждой итерации перед считыванием приходится находить нужный фрагмент исходного файла. Во-вторых, может понадобиться огромное пространство на жестком диске. Если длина исходного файла 400 Мб, новый файл постепенно вырастет до этой величины, и перед самым удалением исходного файла будет занято 800 Мб, т. е. на 400 Мб больше, чем следовало бы. Так что все пути ведут... к третьему методу.

Метод 3: один файл, два буфера

Программа инициализирует два отдельных 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 пугает 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.

| 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).

| Значение | Описание |
|---------------------------------------|---|
| 0 | Другие попытки открыть файл закончатся неудачно |
| FILE_SHARE_READ | Попытка постороннего процесса открыть файл с флагом <code>GENERIC_WRITE</code> не удастся |
| FILE_SHARE_WRITE | Попытка постороннего процесса открыть файл с флагом <code>GENERIC_READ</code> не удастся |
| FILE_SHARE_READ FILE_SHARE_WRITE | Посторонний процесс может открывать файл без ограничений |

Создав или открыв указанный файл, *CreateFile* возвращает его дескриптор, в ином случае — идентификатор `INVALID_HANDLE_VALUE`.



Большинство функций Windows, возвращающих те или иные дескрипторы, 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,
    PCSTR pszName);
```

Первый параметр, *hFile*, идентифицирует дескриптор файла, проецируемого на адресное пространство процесса. Этот дескриптор 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 память, аналогично резервированию региона адресного пространства с последующей 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 следующей таблице.

| Атрибут защиты | Описание |
|----------------|---|
| PAGE_READONLY | Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла. При этом Вы должны были передать в <i>CreateFile</i> флаг <i>GENERIC_READ</i> . |
| PAGE_READWRITE | Отобразив объект «проекция файла» на адресное пространство, можно считывать данные из файла и записывать их. При этом Вы должны были передать в <i>CreateFile</i> комбинацию флагов <i>GENERIC_READ GENERIC_WRITE</i> . |
| 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 <i>CreateFile</i> либо <i>GENERIC_READ</i> , либо <i>GENERIC_READ GENERIC_WRITE</i> . |

WINDOWS 98 В Windows 98 функции *CreateFileMapping* можно передать флаг *PAGE_WRITECOPY*; тем самым Вы скажете системе передать физическую память из страничного файла. Эта память резервируется для копии информации из файла данных, и лишь модифицированные страницы действительно записываются в страничный файл. Изменения не распространяются на исходный файл данных. Результат применения флага *PAGE_WRITECOPY* одинаков в Windows 2000 и в Windows 98.

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.

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.

WINDOWS 98 Windows 98 игнорирует флаг *SEC_NOCACHE*.

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 проецируе-

мого 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.

WINDOWS 98 Windows 98 игнорирует флаг `SEC_IMAGE`.

98

Последние два атрибута (`SEC_RESERVE` и `SEC_COMMIT`) взаимоисключают друг друга и неприменимы для проецирования в память файла данных. Эти флаги мы рассмотрим ближе к концу главы. *CreateFileMapping* их игнорирует.

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 *`dwMaximumSizeHigh`* 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 *`dwMaximumSizeHigh`* 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 большими 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 hinstExe, HINSTANCE,
    PTSTR pszCmdLine, int nCmdShow) {

    // перед выполнением этого оператора, в каталоге C:\
    // еще нет файла "MMFTest.dat"
    HANDLE hfile = CreateFile("C:\\MMFTest.dat",
```

см. след. стр.

```

    GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ | FILE_SHARE_WRITE, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    // перед выполнением этого оператора файл MMFTTest.dat существует,
    // но имеет нулевую длину
    HANDLE hfilemap = CreateFileMapping(hfile, NULL, PAGE_READWRITE, 0, 100, NULL);

    // после выполнения предыдущего оператора размер файла MMFTTest.dat
    // возрастает до 100 байтов

    // очистка
    CloseHandle(hfilemap);
    CloseHandle(hfile);

    // по завершении процесса файл MMFTTest.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 `INVALID_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 под данные файла, передала их как физическую память, отображенную на регион. Это делает функция *MapViewOfFile*:

```

PVOID MapViewOfFile(
    HANDLE hFileMappingObject,
    DWORD dwDesiredAccess,
    DWORD dwFileOffsetHigh,
    DWORD dwFileOffsetLow,
    SIZE_T dwNumberOfBytesToMap);

```

Параметр *bFileMappingObject* 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.

| Значение | Описание |
|---------------------|---|
| FILE_MAP_WRITE | Файловые данные можно считывать и записывать; Вы должны были передать функции <i>CreateFileMapping</i> атрибут PAGE_READWRITE |
| FILE_MAP_READ | Файловые данные можно только считывать; Вы должны были вызвать <i>CreateFileMapping</i> 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 <i>CreateFileMapping</i> с любым из следующих атрибутов: PAGE_READONLY, PAGE_READWRITE или PAGE_WRITECOPY (Windows 98 требует вызывать <i>CreateFileMapping</i> с атрибутом PAGE_WRITECOPY) |

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 экзбайтов, придется 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 составляет 64 Кб.) О гранулярности выделения памяти см. раздел «Системная информация» в главе 14.

Во-вторых, от Вас потребуется указать размер представления, т. е. сколько байтов 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.

WINDOWS 2000 В Windows 2000 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 получают 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_WRITECOPY

// 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 процесса.
// 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 файлом)

// 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);
```

WINDOWS
98

Как уже упоминалось, Windows 98 сначала передает проецируемому файлу физическую память из страничного файла. Однако запись модифицированных страниц в страничный файл происходит только при необходимости.

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, *FlushViewOfFile* 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 «про-

екция 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.

WINDOWS 98 Windows 98 не поддерживает атрибут защиты «копирование при записи», поэтому при просмотре первого представления файла, проецируемого в память, Вы не сможете проверить страницы по флагу `PAGE_READWRITE`. Вам придется разработать свой метод.

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 по-работает 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);
```

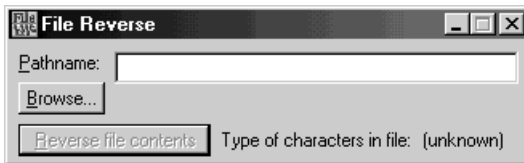
При операциях с проецируемыми файлами обычно открывают файл, создают объект «проекция файла» и с его помощью проецируют представление файловых

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» (см. листинг на рис. 17-2), 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 на компакт-диске, прилагаемом к книге. 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).

WINDOWS 98 В Windows 98 функция *IsTextUnicode* определена, но не реализована; она просто возвращает FALSE, а последующий вызов *GetLastError* дает ERROR_CALL_NOT_IMPLEMENTED. Это значит, что программа FileRev, выполняемая в Windows 98, всегда считает, что файл содержит текст в ANSI-кодировке.

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 вызывает-ся 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;

// "переворачиваем" содержимое файла
_strrev(pchANSI);
```

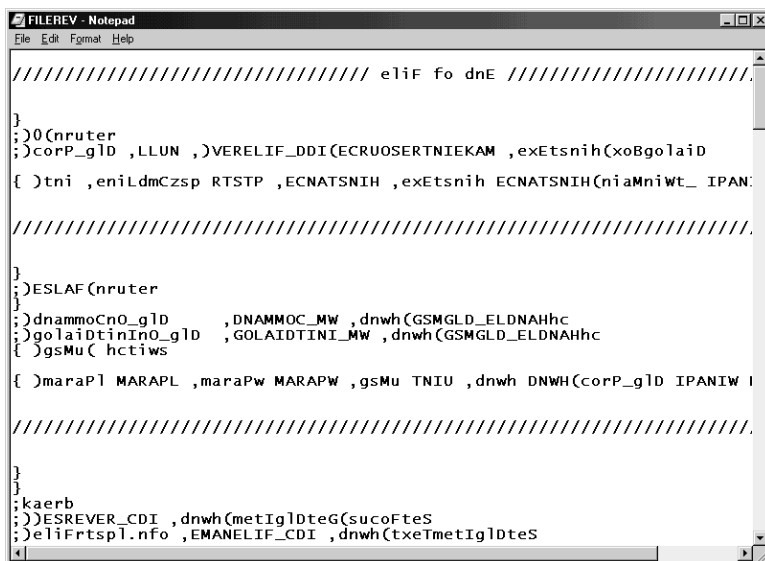
В текстовом файле каждая строка завершается символами возврата каретки («\r») и перевода строки («\n»). К сожалению, после вызова функции *_strrev* эти символы тоже меняются местами. Поэтому для загрузки преобразованного файла в текстовый редактор придется заменить все пары «\n\r» на исходные «\r\n». В программе этим занимается следующий цикл:

```
while (pchANSI != NULL) {
    // вхождение найдено...
    *pchANSI++ = '\r'; // заменяем '\n' на '\r'
    *pchANSI++ = '\n'; // заменяем '\r' на '\n'
    pchANSI = strchr(pchANSI, '\n'); // 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 полученный файл оказался бы на 1 символ длиннее, и тогда повторный запуск программы FileRev не 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);
```





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

```

/*****
Модуль: FileRev.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\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 pfIsTextUnicode) {

    *pfIsTextUnicode = 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 предполагаю худшее и добавляю размер WCHAR вместо CHAR.

```

Рис. 17-2. Программа-пример FileRev

см. след. стр.

Рис. 17-2. *продолжение*

```

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; // выполнить все проверки
*pfIsTextUnicode = IsTextUnicode(pvFile, dwFileSize, &iUnicodeTestFlags);

if (!*pfIsTextUnicode) {
    // при дальнейших операциях с файлом явно используем ANSI-функции,
    // так как мы имеем 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 "\r\n", чтобы сохранить
    // нормальную последовательность кодов завершения строки в текстовом файле
    pchANSI = strchr(pchANSI, '\n'); // ищем первое вхождение '\n'

    while (pchANSI != NULL) {
        // вхождение найдено...
        *pchANSI++ = '\r'; // заменяем '\n' на '\r'
        *pchANSI++ = '\n'; // заменяем '\r' на '\n'
        pchANSI = strchr(pchANSI, '\n'); // 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;
}

```

Рис. 17-2. *продолжение*

```

    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 = wcschr(pchUnicode, L'\n'); // ищем первое вхождение '\n'

    while (pchUnicode != NULL) {
        // вхождение найдено...
        *pchUnicode++ = L'\r'; // заменяем '\n' на '\r'
        *pchUnicode++ = L'\n'; // заменяем '\r' на '\n'
        pchUnicode = wcschr(pchUnicode, L'\n'); // 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, отключая кнопку Reverse
    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) {

```

см. след. стр.

Рис. 17-2. *продолжение*

```

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, chDIMOF(szPathname));

    // делаем копию исходного файла, чтобы случайно не повредить его
    if (!CopyFile(szPathname, FILENAME, FALSE)) {
        chMB("New file could not be created.");
        break;
    }

    BOOL fIsTextUnicode;
    if (FileReverse(FILENAME, &fIsTextUnicode)) {
        SetDlgItemText(hwnd, IDC_TEXTTYPE,
            fIsTextUnicode ? 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 = chDIMOF(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;
}
}

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

```

Рис. 17-2. *продолжение*

```

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_FILEREV), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конец файла //////////////////////////////////////

```

Обработка больших файлов

Я обещал рассказать, как спроецировать на небольшое адресное пространство файл длиной 16 экзбайтов. 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 самого начала 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 память, 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 Count0s(void) {
    // начальные границы представлений всегда начинаются по адресам,
    // кратным гранулярности выделения памяти
    SYSTEM_INFO sinf;
    GetSystemInfo(&sinf);

    // открываем файл данных
    HANDLE hFile = CreateFile("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);
}

```

см. след. стр.

```

__int64 qwFileOffset = 0, qwNumOf0s = 0;
while (qwFileSize > 0) {
    // определяем, сколько байтов надо спроецировать
    DWORD dwBytesInBlock = sinf.dwAllocationGranularity;
    if (qwFileSize < sinf.dwAllocationGranularity)
        dwBytesInBlock = (DWORD) qwFileSize;

    PBYTE pbFile = (PBYTE) MapViewOfFile(hFileMapping, FILE_MAP_READ,
        (DWORD) (qwFileOffset >> 32),           // начальный байт
        (DWORD) (qwFileOffset & 0xFFFFFFFF),    // в файле
        dwBytesInBlock);                        // число проецируемых байтов

    // подсчитываем количество нулевых байтов в этом блоке
    for (DWORD dwByte = 0; dwByte < dwBytesInBlock; dwByte++) {
        if (pbFile[dwByte] == 0)
            qwNumOf0s++;
    }
    // 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(qwNumOf0s);
}

```

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 по-прежнему 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.



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 дает 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), а в Windows 98 базовый адрес будет округлен до ближайшего меньшего значения, кратного гранулярности выделения памяти.

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 (так 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 Microsoft поддерживают указатели со смещением относительно базового значения (*based-pointers*), для чего предусмотрено ключевое слово `__based`.

WINDOWS 98 В Windows 98 при вызове *MapViewOfFileEx* следует указывать адрес в диапазоне от 0x80000000 до 0xBFFFFFFF, иначе функция вернет NULL.

WINDOWS 2000 В Windows 2000 при вызове *MapViewOfFileEx* следует указывать адрес в границах пользовательского раздела адресного пространства процесса, иначе функция вернет NULL.

Особенности проецирования файлов на разных платформах

Механизм проецирования файлов в Windows 2000 и Windows 98 реализован по-разному. Вы должны знать об этих отличиях, поскольку они могут повлиять на код программ и целостность используемых ими данных.

В Windows 98 представление всегда проецируется на раздел адресного пространства, расположенный в диапазоне от 0x80000000 до 0xBFFFFFFF. Значит, после успешного вызова функция *MapViewOfFile* вернет какой-нибудь адрес из этого диапазона. Но вспомните: данные в этом разделе доступны всем процессам. Если один из процессов отображает сюда представление объекта «проекция файла», то принадлежащие этому объекту данные физически доступны всем процессам, и уже неважно: проецируют ли они сами представление того же объекта. Если другой процесс вызывает *MapViewOfFile*, используя тот же объект «проекция файла», Windows 98 возвращает адрес памяти, идентичный тому, что она сообщила первому процессу. Поэтому два процесса обращаются к одним и тем же данным и представления их объектов когерентны.

В Windows 98 один процесс может вызвать *MapViewOfFile* и, воспользовавшись какой-либо формой межпроцессной связи, передать возвращенный ею адрес памяти потоку другого процесса. Как только этот поток получит нужный адрес, ему уже ничто не мешает получить доступ к тому же представлению объекта «проекция файла». Но прибегать к такой возможности не следует по двум причинам:

- приложение не будет работать в Windows 2000 (и я только что рассказал — почему);
- если первый процесс вызовет *UnmapViewOfFile*, регион адресного пространства освободится. А значит, при попытке потока второго процесса обратиться к участку памяти, где когда-то находилось представление, возникнет нарушение доступа.

Чтобы второй процесс получил доступ к представлению проецируемого файла, его поток тоже должен вызвать *MapViewOfFile*. Тогда система увеличит счетчик числа пользователей объекта «проекция файла». И если первый процесс обратится к *UnmapViewOfFile*, регион адресного пространства, занятый представлением, не будет освобожден, пока второй процесс тоже не вызовет *UnmapViewOfFile*. А вызвав *MapViewOfFile*, второй процесс получит тот же адрес, что и первый. Таким образом, необходимость в передаче адреса от первого процесса второму отпадает.

В Windows 2000 механизм проецирования файлов реализован удачнее, чем в Windows 98, потому что Windows 2000 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.

Адрес, полученный при вызове *MapViewOfFile* первым процессом, скорее всего не совпадет с тем, что получит при ее вызове второй процесс, — даже несмотря на то что оба процесса проецируют представление одного и того же объекта. И хотя в Windows 98 адреса, получаемые процессами при вызове *MapViewOfFile*, совпадают,

лучше не полагаться на эту особенность — иначе приложение не станет работать в Windows 2000!

Рассмотрим еще одно различие механизмов проецирования файлов у Windows 2000 и Windows 98. Взгляните на текст программы, проецирующей два представления единственного объекта «проекция файла».

```
#include <Windows.h>

int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE,
    PTSTR pszCmdLine, int nCmdShow) {

    // открываем существующий файл; он должен быть больше 64 Кб
    HANDLE hFile = CreateFile(pszCmdLine, GENERIC_READ | GENERIC_WRITE, 0,
        NULL, OPEN_ALWAYS, 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_WRITE, 0, 65536, 0);

    if ((pbFile + 65536) == pbFile2) {
        // если адреса перекрываются, оба представления проецируются на один
        // регион, и мы работаем в Windows 98
        MessageBox(NULL, "We are running under Windows 98", NULL, MB_OK);
    } else {
        // если адреса не перекрываются, каждое представление размещается в
        // своем регионе адресного пространства, и мы работаем в Windows 2000
        MessageBox(NULL, "We are running under Windows 2000", NULL, MB_OK);
    }
    UnmapViewOfFile(pbFile2);
    UnmapViewOfFile(pbFile);
    CloseHandle(hFileMapping);
    CloseHandle(hFile);

    return(0);
}
```

Когда приложение в Windows 98 отображает на адресное пространство представление объекта «проекция файла», ему отводится регион, достаточно большой для размещения всего объекта. Это происходит, даже если Вы просите *MapViewOfFile* спроецировать лишь малую часть такого объекта. Поэтому спроецировать объект размером 1 Гб не удастся, даже если указать, что представление должно быть не более 64 Кб.

При вызове каким-либо процессом функции *MapViewOfFile* ему возвращается адрес в пределах региона, зарезервированного для *целого* объекта «проекция файла». Так что в показанной выше программе первый вызов этой функции дает базовый адрес региона, содержащего весь спроецированный файл, а второй — адрес, смещенный «вглубь» того же региона на 64 Кб.

Windows 2000 и здесь ведет себя совершенно иначе. 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». А в Windows 98 такие представления когерентны потому, что они расположены в одном участке памяти.

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, сокет 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. *MapViewOfFileEx* вызывается вместо *MapViewOfFile*, чтобы представление файла было спроецировано по базовому адресу, значение которого хранится в самом EXE-файле. Потом создается первичный поток 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, Microsoft 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 *dwMaximumSizeHigh* 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.



Есть одна интересная ловушка, в которую может попасть неискушенный программист. Попробуйте догадаться, что неверно в этом фрагменте кода:

```
HANDLE hFile = CreateFile(...);
HANDLE hMap = CreateFileMapping(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!

Программа-пример MMFShare

Эта программа, «17 MMFShare.exe» (см. листинг на рис. 17-3), 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 на компакт-диске, прилагаемом к книге.

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 стра-

ничного 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

```

/*****
Модуль: MMFShare.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение 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 тестовыми данными

```

Рис. 17-3. Программа-пример MMFShare

см. след. стр.

Рис. 17-3. *продолжение*

```

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 Кб 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),
                            (LPTSTR) pView, 4 * 1024);

                        // прекращаем проецирование; это защитит
                        // данные от "блуждающих" указателей
                        UnmapViewOfFile(pView);
                    }
                }
            }
        }
    }
}

```


Рис. 17-3. *продолжение*

```

        // 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: не 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), (LPTSTR) pView);
            UnmapViewOfFile(pView);
        } else {
            chMB("Can't map view.");
        }
    }
}

```

см. след. стр.

Рис. 17-3. *продолжение*

```

        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 pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_MMFSHARE), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

Частичная передача физической памяти проецируемым файлам

До сих пор мы видели, что система требует передавать проецируемым файлам всю физическую память либо из файла данных на диске, либо из страничного файла. Это значит, что память 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 (200 × 256 × 128) 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 Вы 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 *MapViewOfFileEx*, 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 память региону по-прежнему не передается, так что, если потоки в других процессах попытаются обратиться по одному из адресов представления 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 pvAddress,
    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 получают доступ к страницам физической памяти, переданным региону.

Итак, флаг SEC_RESERVE и функция *VirtualAlloc* позволяют сделать табличную матрицу *CellData* «общедоступной» и эффективнее использовать память.

WINDOWS 98 Обычно *VirtualAlloc* не срабатывает, если Вы передаете ей адрес памяти, выходящий за пределы диапазона от 0x00400000 до 0x7FFFFFFF. Однако при передаче физической памяти проецируемому файлу, созданному с флагом SEC_RESERVE, в *VirtualAlloc* нужно передать адрес, укладывающийся в диапазон от 0x80000000 до 0xBFFFFFFF. Только тогда Windows 98 поймет, что физическая память передается региону, зарезервированному под проецируемый файл, и даст благополучно выполнить вызов функции.

WINDOWS 2000 В Windows 2000 функция *VirtualFree* не годится для возврата физической памяти, переданной в свое время проецируемому файлу (созданному с флагом SEC_RESERVE). Однако в Windows 98 такого ограничения нет.

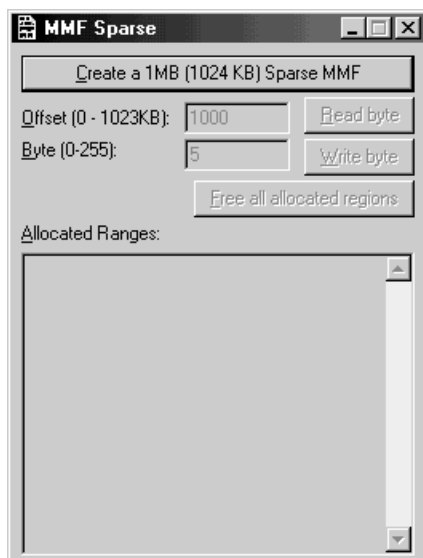
Файловая система NTFS 5 поддерживает так называемые разреженные файлы (sparse files). Это потрясающая новинка. Она позволяет легко создавать и использовать

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» (см. листинг на рис. 17-4), демонстрирует, как создать проецируемый в память файл, связанный с разреженным файлом NTFS 5. Файлы исходного кода и ресурсов этой программы находятся в каталоге 17-MMFSparse на компакт-диске, прилагаемом к книге. После запуска MMFSparse на экране появляется окно, показанное ниже.

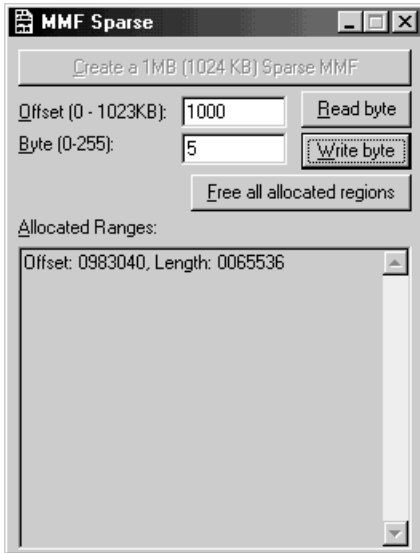


Когда Вы щелкнете кнопку Create a 1MB (1024 KB) Sparse MMF, программа попытается создать разреженный файл «C:\MMFSparse». Если Ваш диск C не является томом NTFS 5, у программы ничего не получится, и ее процесс завершится. А если Вы создали том NTFS 5 на каком-то другом диске, модифицируйте мою программу и перекомпилируйте ее, чтобы посмотреть, как она работает.

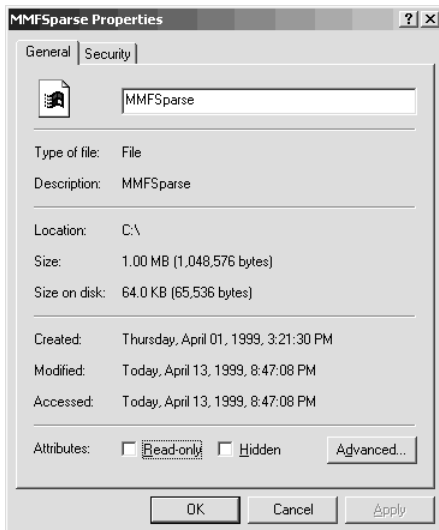
После создания разреженный файл проецируется на адресное пространство процесса. В поле Allocated Ranges (внизу окна) показывается, какие части файла действительно связаны с дисковой памятью. Изначально файл не связан ни с какой памятью, и в этом поле сообщается «No allocated ranges in the file» («В файле нет 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 (1000×1024).



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 C:\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 `CSparseStream`. 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.

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*.



MMFSparse.cpp

```

/*****
Модуль: MMFSparse.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <tchar.h>
#include <WindowsX.h>
#include <WinIoctl.h>
#include "SparseStream.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, SIZE_T dwStreamSizeMax = 0);

    // закрывает разреженный MMF
    virtual ~CMMFSparse() { ForceClose(); }

    // создает разреженный MMF и проецирует его на адресное пространство процесса
    BOOL Initialize(HANDLE hstream, SIZE_T dwStreamSizeMax);

    // 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, SIZE_T dwStreamSizeMax) {
    Initialize(hstream, dwStreamSizeMax);
}

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

```

Рис. 17-4. Программа-пример MMFSparse

см. след. стр.

Рис. 17-4. *продолжение*

```

BOOL CMMFSparse::Initialize(HANDLE hstream, SIZE_T dwStreamSizeMax) {

    if (m_hfilemap != NULL)
        ForceClose();

    // инициализируем значением NULL на случай, если что-то пойдет не так
    m_hfilemap = m_pvFile = NULL;

    BOOL fOk = TRUE; // предполагаем, что все будет хорошо

    if (hstream != NULL) {
        if (dwStreamSizeMax == 0) {
            DebugBreak(); // недопустимый размер потока данных
        }

        CSparseStream::Initialize(hstream);
        fOk = MakeSparse(); // делаем поток разреженным
        if (fOk) {
            // создаем объект "проекция файла"
            m_hfilemap = ::CreateFileMapping(hstream, NULL, PAGE_READWRITE,
                (DWORD) (dwStreamSizeMax >> 32i64), (DWORD) dwStreamSizeMax, 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 файл не удалось; проводим очистку
                CSparseStream::Initialize(NULL);
                ForceClose();
                fOk = FALSE;
            }
        }
    }
    return(fOk);
}

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

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;
    }
}

```

Рис. 17-4. *продолжение*

```

////////////////////////////////////
#define STREAMSIZE      (1 * 1024 * 1024) // 1 Мб (1024 Кб)
TCHAR szPathname[] = TEXT("C:\\MMFSparse.");
HANDLE g_hstream = INVALID_HANDLE_VALUE;
CMMFSparse g_mmf;

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

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);

    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++) {
            wsprintf(_tcschr(sz, 0), 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);
}

```

см. след. стр.

Рис. 17-4. *продолжение*

```

////////////////////////////////////
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:
            // создаем файл
            g_hstream = CreateFile(szPathname, GENERIC_READ | GENERIC_WRITE,
                0, NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
            if (g_hstream == INVALID_HANDLE_VALUE) {
                chFAIL("Failed to create file.");
            }

            // используя этот файл, создаем MMF размером 1 Мб (1024 Кб)
            if (!g_mmf.Initialize(g_hstream, STREAMSIZE)) {
                chFAIL("Failed to initialize Sparse MMF.");
            }
            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 fTranslated;
                DWORD dwOffset = GetDlgItemInt(hwnd, IDC_OFFSET, &fTranslated, FALSE);
                if (fTranslated) {
                    g_mmf[dwOffset * 1024] = (BYTE)
                        GetDlgItemInt(hwnd, IDC_BYTE, NULL, FALSE);
                    Dlg_ShowAllocatedRanges(hwnd);
                }
            }
            break;

        case IDC_READBYTE:
            {
                BOOL fTranslated;

```

Рис. 17-4. *продолжение*

```

        DWORD dwOffset = GetDlgItemInt(hwnd, IDC_OFFSET, &fTranslated, FALSE);
        if (fTranslated) {
            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);
    g_mmf.Initialize(g_hstream, STREAMSIZE);
    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) {

    chWindows2000Required();

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_MMFSPARSE), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

см. след. стр.

Рис. 17-4. *продолжение*

SparseStream.h

```

/*****
Модуль: SparseStream.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\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);
    }
};

```

Рис. 17-4. *продолжение*

```

////////////////////////////////////
inline BOOL CSparseStream::DoesFileSystemSupportSparseStreams(
    PCTSTR pszVolume) {

    DWORD dwFileSystemFlags = 0;
    BOOL fOk = GetVolumeInformation(pszVolume, NULL, 0, NULL, NULL,
        &dwFileSystemFlags, NULL, 0);
    fOk = fOk && AreFlagsSet(dwFileSystemFlags, FILE_SUPPORTS_SPARSE_FILES);
    return(fOk);
}

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

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 работает, если файл проецируется 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, (LPVOID) &fzdi,
        sizeof(fzdi), NULL, 0, &dw, NULL));
}

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

inline BOOL CSparseStream::DoesFileContainAnySparseStreams(
    PCTSTR pszPathname) {

    DWORD dw = GetFileAttributes(pszPathname);
    return((dw == 0xffffffff)

```

см. след. стр.

Рис. 17-4. *продолжение*

```

        ? 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 попытки 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));
}

//////////////////////////////////// Конец файла //////////////////////////////////

```

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), странично передает ей физическую память (из страничного файла). А при освобождении блоков в куче диспетчер возвращает системе соответствующие страницы физической памяти.

Microsoft не документирует правила, по которым диспетчер передает или отбирает физическую память. Эти правила различны в Windows 98 и Windows 2000. Могу сказать Вам лишь следующее: Windows 98 больше озабочена эффективностью использования памяти и поэтому старается как можно быстрее отобрать у куч физическую память. Однако Windows 2000 нацелена главным образом на максимальное быстродействие, в связи с чем возвращает физическую память в страничный файл, только если страницы не используются в течение определенного времени. Microsoft 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 в его адресном пространстве стандартную кучу (process's default heap). Ее размер по умолчанию — 1 Мб. Но система позволяет увеличивать этот размер, для чего надо указать компоновщику при сборке про-

граммы ключ /HEAP. (Однако при сборке DLL этим ключом пользоваться нельзя, так как для DLL куча не создается.)

`/HEAP:reserve[,commit]`

Стандартная куча процесса необходима многим Windows-функциям. Например, функции ядра Windows 2000 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 быстрого 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.

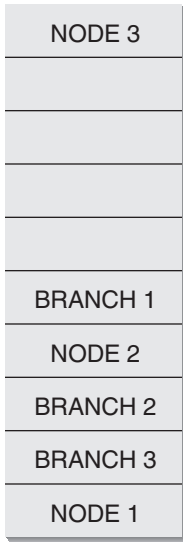


Рис. 18-1. Единая куча, в которой размещены структуры NODE и BRANCH

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 испорчена. 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 не 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.

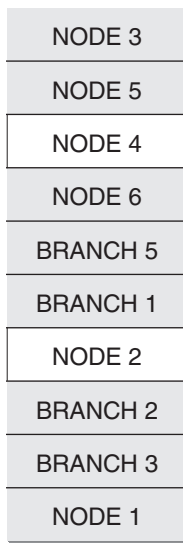


Рис. 18-2. Фрагментированная куча, содержащая несколько объектов *NODE* и *BRANCH*

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 одновременно 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 dwMaximumSize);
```

Параметр *fdwOptions* модифицирует способ выполнения операций над кучей. В нем можно указать 0, HEAP_NO_SERIALIZE, HEAP_GENERATE_EXCEPTIONS или комбинацию последних двух 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 поток 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 взаимoisключения, 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.

Второй параметр функции *HeapCreate* — *dwInitialSize* — 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 не 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*:

```
PVOID 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.

| Идентификатор | Описание |
|--------------------------------------|--|
| <code>STATUS_NO_MEMORY</code> | Попытка выделения памяти не удалась из-за ее нехватки |
| <code>STATUS_ACCESS_VIOLATION</code> | 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.

WINDOWS 98 Вызов *HeapAlloc* с требованием выделить блок размером более 256 Мб Windows 98 считает ошибкой. Заметьте, что в этом случае функция всегда возвращает `NULL`, а исключение никогда не генерируется, даже если при создании кучи или попытке выделить блок Вы указали флаг `HEAP_GENERATE_EXCEPTIONS`.



Для выделения больших блоков памяти (от 1 Мб) рекомендуется использовать функцию *VirtualAlloc*, а не функции, оперирующие с кучами.

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 *fdwFlags* указывает флаги, используемые при изменении разме-

ра блока: `HEAP_GENERATE_EXCEPTIONS`, `HEAP_NO_SERIALIZE`, `HEAP_ZERO_MEMORY` или `HEAP_REALLOC_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, включающиеся 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 *fdwFlags* 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 адрес-

ного 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 библиотеке C. Когда необходимость в данном объекте класса отпадает, вместо библиотечной C-функции *free* следует применять оператор *delete*. Скажем, у нас есть класс *CSomeClass*, 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 *delete*:

```
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_uNumAllocsInHeap;

    // 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_uNumAllocsInHeap*, 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 C++-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 память 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_uNumAllocsInHeap* 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 Вам 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*. Самое ценное в этих функциях то, что они доступны как в Windows 98, так и в Windows 2000. Прочие функции, о которых пойдет речь в этом разделе, есть только в Windows 2000.

В адресном пространстве процесса может быть несколько куч, и функция *GetProcessHeaps* позволяет получить их описатели «одним махом»:

```
DWORD GetProcessHeaps(
    DWORD dwNumHeaps,
    PHANDLE pHeaps);
```

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 *fdwFlags* передают 0, но можно передать и `HEAP_NO_SERIALIZE`.

Следующие две функции — *HeapLock* и *HeapUnlock* — используются парно:

```
BOOL HeapLock(HANDLE hHeap);
BOOL 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*:

```
BOOL 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.

Ч А С Т Ь I V

ДИНАМИЧЕСКИ ПОДКЛЮЧАЕМЫЕ БИБЛИОТЕКИ



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.
- **Возможность использования разных языков программирования.** У Вас есть выбор, на каком языке писать ту или иную часть приложения. Так, пользовательский интерфейс приложения Вы скорее всего будете создавать на Microsoft Visual Basic, но прикладную логику лучше всего реализовать на C++. Программа на Visual Basic может загружать DLL, написанные на C++, Коболе, Фортране и др.
- **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 экземпляр, 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 komponуются 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 Web-браузером ActiveX-элементам, позволяющим создавать Web-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 конечный 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 98 это достигается за счет выделения специальной области памяти для таких переменных при проецировании EXE-файла на адресное пространство процесса, а 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.



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 сегодняшний день метод. (Windows поддерживает и явное связывание, но об этом — в главе 20.)

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 функций, 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 (глобаль-

ные 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.

СОЗДАНИЕ DLL

- 1) Заголовочный файл с экспортируемыми прототипами, структурами и идентификаторами (символьными именами)
- 2) Исходные файлы C/C++, в которых реализованы функции и определены переменные
- 3) Компилятор создает OBJ-файл из каждого исходного файла C/C++
- 4) Компоновщик собирает DLL из OBJ-модулей
- 5) Если DLL экспортирует хотя бы одну переменную или функцию, компоновщик создает и LIB-файл

СОЗДАНИЕ EXE

- 6) Заголовочный файл с импортируемыми прототипами, структурами и идентификаторами
- 7) Исходные файлы C/C++, из которых вызываются импортируемые функции и переменные
- 8) Компилятор создает OBJ-файл из каждого исходного файла C/C++
- 9) Используя OBJ-модули и LIB-файл и учитывая ссылки на импортируемые идентификаторы, компоновщик собирает EXE-модуль (в котором также размещается таблица импорта — список необходимых DLL и импортируемых идентификаторов)

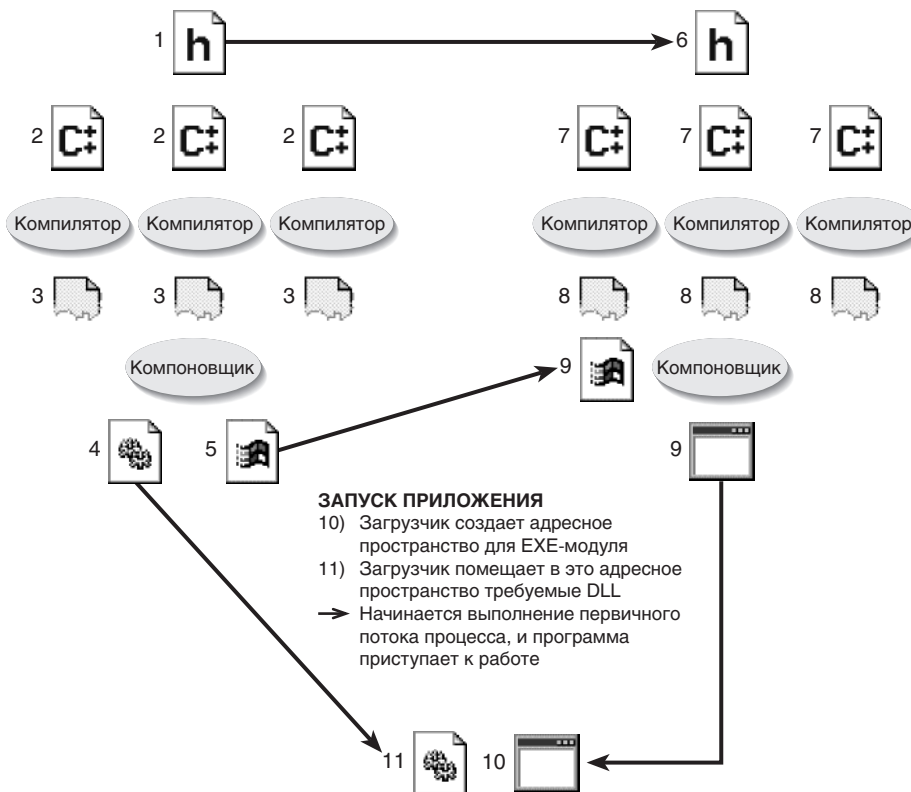


Рис. 19-1. Так DLL создается и неявно связывается с приложением

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-модуль, в который в конечном итоге помещаются двоичный код и переменные (глобальные и статические), относящиеся к данному EXE. В нем также создается раздел импорта, где перечисляются имена всех необходимых DLL-модулей (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 из 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. Еди-

ный 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.

```

/*****
Модуль: MyLib.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);

//////////////////////////////////// Конеч файла //////////////////////////////////

```

Этот заголовочный файл надо включать в самое начало исходных файлов Вашей DLL следующим образом.

```

/*****
Модуль: MyLibFile1.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);
}

//////////////////////////////////// Конец файла //////////////////////////////////////
```

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 MyLibFile1.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 исказить имена переменных или функций, и они становятся доступными исполняемому модулю, написанному 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 Microsoft.

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:\WINNT\SYSTEM32>DUMPBIN -exports Kernel32.DLL
```

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

```
Dump of file kernel32.dll
```

```
File Type: DLL
```

```
Section contains the following exports for KERNEL32.dll
```

```

    0 characteristics
36DB3213 time date stamp Mon Mar 01 16:34:27 1999
    0.00 version
    1 ordinal base
    829 number of functions
    829 number of names
```

| ordinal | hint | RVA | name |
|---------|------|----------|---------------------------|
| 1 | 0 | 0001A3C6 | AddAtomA |
| 2 | 1 | 0001A367 | AddAtomW |
| 3 | 2 | 0003F7C4 | AddConsoleAliasA |
| 4 | 3 | 0003F78D | AddConsoleAliasW |
| 5 | 4 | 0004085C | AllocConsole |
| 6 | 5 | 0002C91D | AllocateUserPhysicalPages |
| 7 | 6 | 00005953 | AreFileApisANSI |
| 8 | 7 | 0003F1A0 | AssignProcessToJobObject |
| 9 | 8 | 00021372 | BackupRead |
| 10 | 9 | 000215CE | BackupSeek |
| 11 | A | 00021F21 | BackupWrite |

```
:
```

```

828 33B 00003200 lstrlenA
829 33C 000040D5 lstrlenW
```

```
Summary
```

```

3000 .data
4000 .reloc
4D000 .rsrc
59000 .text
```

Как видите, идентификаторы расположены по алфавиту; в графе RVA указывается смещение в образе DLL-файла, по которому можно найти экспортируемый иденти-

фикатор. 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.



Многие разработчики — особенно те, у кого большой опыт программирования для 16-разрядной Windows, — привыкли экспортировать функции из DLL, присваивая им порядковые номера. Но Microsoft 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.

Кстати, именно это и случилось со мной. В журнале *Microsoft Systems Journal* я опубликовал программу, построенную на применении порядковых номеров. В Windows NT 3.1 программа работала прекрасно, но сбила при запуске в Windows NT 3.5. Чтобы избавиться от сбоев, пришлось заменить порядковые номера именами функций, и все встало на свои места.

Я поинтересовался, почему Microsoft отказывается от порядковых номеров, и получил такой ответ: «Мы (Microsoft) 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. Microsoft 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 скомпоновать 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 способами. 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 скомпилирует 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.

```

/*****
Модуль: MyExeFile1.cpp
*****/

// сюда включаются стандартные заголовочные файлы Windows и библиотеки C
#include <windows.h>

// включаем экспортируемые структуры данных, идентификаторы, функции и переменные
#include "MyLib\MyLib.h"

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

int WINAPI WinMain(HINSTANCE hinstExe, HINSTANCE, LPTSTR pszCmdLine, int) {

```

```

int nLeft = 10, nRight = 25;

TCHAR sz[100];
wsprintf(sz, TEXT("%d + %d = %d"), nLeft, nRight, Add(nLeft, nRight));
MessageBox(NULL, sz, TEXT("Calculation"), MB_OK);

wsprintf(sz, TEXT("The result from the last Add is: %d"), g_nResult);
MessageBox(NULL, sz, TEXT("Last Result"), MB_OK);
return(0);
}

/////////////////////// Конец файла /////////////////////////

```

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:\WINNT\SYSTEM32>DUMPBIN -imports Calc.EXE
```

```
Microsoft (R) COFF Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.
```

```
Dump of file calc.exe
```

```
File Type: EXECUTABLE IMAGE
```

```
Section contains the following imports:
```

```
SHELL32.dll
```

```
    10010F4 Import Address Table
    1012820 Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference
```

```
    77C42983    7A ShellAboutW
```

```
MSVCRT.dll
```

```
    1001094 Import Address Table
    10127C0 Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference
```

```
    78010040    295 memmove
    78018124    42  _EH_prolog
    78014C34    2D1 toupper
    78010F6E    2DD wcschr
    78010668    2E3 wcslen
```

```
:
```

```
ADVAPI32.dll
```

```
    1001000 Import Address Table
    101272C Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference
```

```
    779858F4    19A RegQueryValueExA
    77985196    190 RegOpenKeyExA
    77984BA1    178 RegCloseKey
```

```
KERNEL32.dll
```

```
    100101C Import Address Table
    1012748 Import Name Table
    FFFFFFFF time date stamp
    FFFFFFFF Index of first forwarder reference
```

```
    77ED4134    336 lstrcpyW
    77ED33E8    1E5 LocalAlloc
    77EDEF36    DB  GetCommandLineW
    77ED1610    15E GetProfileIntW
    77ED4BA4    1EC LocalReAlloc
```


:

Header contains the following bound import information:

```
Bound to SHELL32.dll [36E449E0] Mon Mar 08 14:06:24 1999
Bound to MSVCRT.dll [36BB8379] Fri Feb 05 15:49:13 1999
Bound to ADVAPI32.dll [36E449E1] Mon Mar 08 14:06:25 1999
Bound to KERNEL32.dll [36DDAD55] Wed Mar 03 13:44:53 1999
Bound to GDI32.dll [36E449E0] Mon Mar 08 14:06:24 1999
Bound to USER32.dll [36E449E0] Mon Mar 08 14:06:24 1999
```

Summary

```
2000 .data
3000 .rsrc
13000 .text
```

Как видите, в разделе есть записи по каждой DLL, необходимой Calc.exe: Shell32.dll, MSVCRT.dll, AdvAPI32.dll, Kernel32.dll, GDI32.dll и User32.dll. Под именем DLL-модуля выводится список идентификаторов, импортируемых программой Calc.exe. Например, Calc.exe обращается к следующим функциям из Kernel32.dll: *lstrcpyW*, *LocalAlloc*, *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

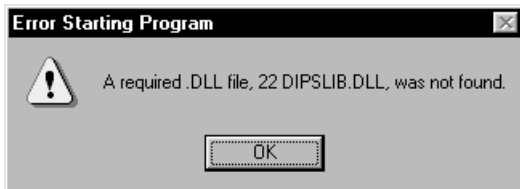
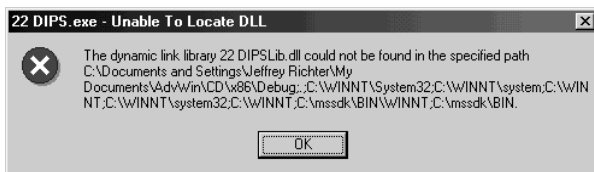
При запуске EXE-файла загрузчик операционной системы создает для его процесса виртуальное адресное пространство и проецирует на него исполняемый модуль. Далее загрузчик анализирует раздел импорта и пытается спроецировать все необходимые DLL на адресное пространство процесса.

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.

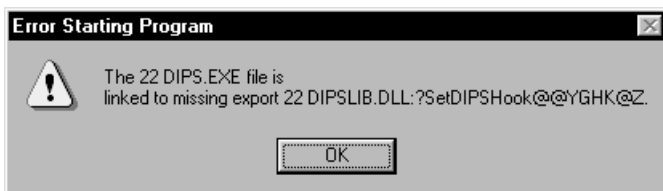
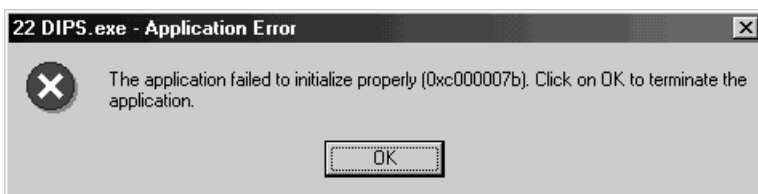
1. Каталог, содержащий EXE-файл.
2. Текущий каталог процесса.
3. Системный каталог Windows.
4. Основной каталог Windows.
5. Каталоги, указанные в переменной окружения PATH.

Учтите, что на процесс поиска библиотек могут повлиять и другие факторы (см. главу 20). 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 удастся, загрузчик выводит одно из двух сообщений (первое — в Windows 2000, а второе — в Windows 98).



Найдя и спроецировав на адресное пространство процесса все необходимые DLL-модули, загрузчик настраивает ссылки на импортируемые идентификаторы. 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 одно из двух сообщений (первое — в Windows 2000, а второе — в Windows 98):



Было бы неплохо, если бы в версии этого окна для Windows 2000 сообщалось имя недостающей функции, а не маловразумительный для пользователя код ошибки вроде 0xC000007B. Ну да ладно, может, в следующей версии Windows это будет исправлено.

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 загруз-

ки 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.

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.

Явная загрузка DLL

В любой момент поток может спроецировать DLL на адресное пространство процесса, вызвав одну из двух функций:

```
HINSTANCE LoadLibrary(PCTSTR pszDLLPathName);
```

```
HINSTANCE 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, со-

общает 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* и *LOAD_WITH_ALTERED_SEARCH_PATH*, о которых мы сейчас и поговорим.

СОЗДАНИЕ DLL

- 1) Заголовочный файл с экспортируемыми прототипами, структурами и идентификаторами (символьными именами)
- 2) Исходные файлы C/C++, в которых реализованы экспортируемые функции и определены переменные
- 3) Компилятор создает OBJ-файл из каждого исходного файла C/C++
- 4) Компоновщик собирает DLL из OBJ-модулей
- 5) Если DLL экспортирует хотя бы одну переменную или функцию, компоновщик создает и LIB-файл (при явном связывании этот файл не используется)

СОЗДАНИЕ EXE

- 6) Заголовочный файл с импортируемыми прототипами, структурами и идентификаторами
- 7) Исходные файлы C/C++, в которых нет ссылок на импортируемые функции и переменные
- 8) Компилятор создает OBJ-файл из каждого исходного файла C/C++
- 9) Компоновщик собирает EXE-модуль из OBJ-модулей (LIB-файл DLL не нужен, так как нет прямых ссылок на экспортируемые идентификаторы; раздел импорта в EXE-модуле отсутствует)

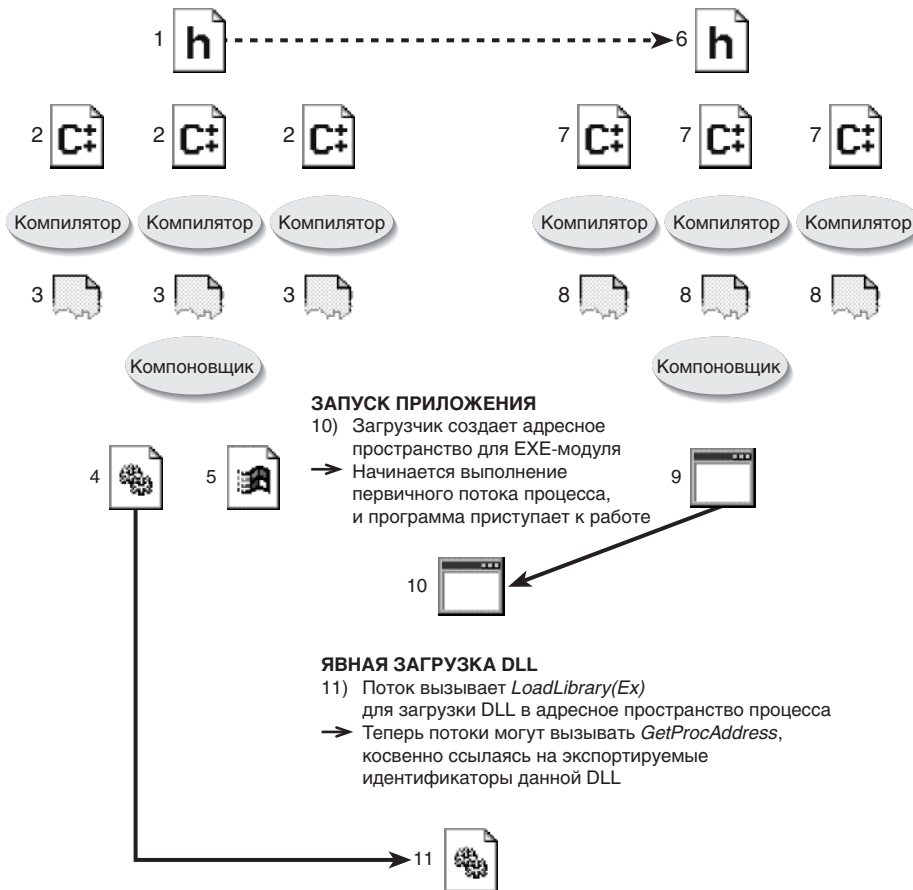


Рис. 20-1. Так DLL создается и явно связывается с приложением

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.

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 файла. Если флаг LOAD_LIBRARY_AS_DATAFILE не указан, атрибуты защиты устанавливаются такими, чтобы код из данного файла можно было выполнять.

Этот флаг может понадобиться по нескольким причинам. Во-первых, его стоит указать, если DLL содержит только ресурсы и никаких функций. Тогда DLL проецируется на адресное пространство процесса, после чего при вызове функций, загружающих ресурсы, можно использовать значение HINSTANCE, 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 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_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, просматривая каталоги в таком порядке:

1. Каталог, заданный в параметре *pszDLLPathName*.
2. Текущий каталог процесса.
3. Системный каталог Windows.
4. Основной каталог Windows.
5. Каталоги, перечисленные в переменной окружения PATH.

Явная загрузка DLL

Если необходимость в DLL отпадает, ее можно выгрузить из адресного пространства процесса, вызвав функцию:

```
BOOL FreeLibrary(HINSTANCE hinstDll);
```

Вы должны передать в *FreeLibrary* значение типа HINSTANCE, которое идентифицирует выгружаемую DLL. Это значение Вы получаете после вызова *LoadLibrary(Ex)*.

DLL можно выгрузить и с помощью другой функции:

```
VOID FreeLibraryAndExitThread(
    HINSTANCE hinstDll,
    DWORD dwExitCode);
```

Она реализована в Kernel32.dll так:

```
VOID FreeLibraryAndExitThread(HINSTANCE hinstDll, DWORD dwExitCode) {
    FreeLibrary(hinstDll);
    ExitThread(dwExitCode);
}
```

На первый взгляд, в ней нет ничего особенного, и Вы, наверное, удивляетесь, с чего это Microsoft 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 не известно 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 исполняемая 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.

Впрочем, *FreeLibraryAndExitThread* может и не понадобиться. Мне она пригодилась лишь раз, когда я занимался весьма нетипичной задачей. Да и код я писал под Windows NT 3.1, где этой функции не было. Наверное, поэтому я так обрадовался, обнаружив ее в более новых версиях Windows.

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 А вызывает приведенную ниже функцию, а затем тот же вызов делает поток в процессе В, то MyLib.dll проецируется на адресное пространство обоих процессов, а счетчики числа пользователей DLL в каждом из них приравниваются 1.

```
HINSTANCE hinstDll = LoadLibrary("MyLib.dll");
```

Если же поток процесса В вызовет далее:

```
FreeLibrary(hinstDll);
```

счетчик числа пользователей DLL в процессе В обнулится, что приведет к отключению DLL от адресного пространства процесса В. Но проекция DLL на адресное пространство процесса А 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*:

```
HINSTANCE GetModuleHandle(PCTSTR pszModuleName);
```

Например, следующий код загружает MyLib.dll, только если она еще не спроецирована на адресное пространство процесса:

```
HINSTANCE hinstDll = GetModuleHandle("MyLib"); // подразумевается расширение .dll
if (hinstDll == NULL) {
    hinstDll = LoadLibrary("MyLib"); // подразумевается расширение .dll
}
```

Если у Вас есть значение HINSTANCE для DLL, можно определить полное (вместе с путем) имя DLL или EXE с помощью *GetModuleFileName*:

```
DWORD GetModuleFileName(
    HINSTANCE hinstModule,
    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.

Явное подключение экспортируемого идентификатора

Поток получает адрес экспортируемого идентификатора из явно загруженной DLL вызовом *GetProcAddress*:

```
FARPROC GetProcAddress(
    HINSTANCE 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 повторяю, что Microsoft

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.)

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
}
```



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, *flmpLoad*, 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.



Не забывайте, что DLL инициализируют себя, используя 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*.

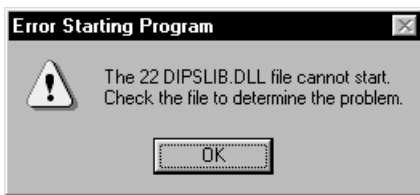
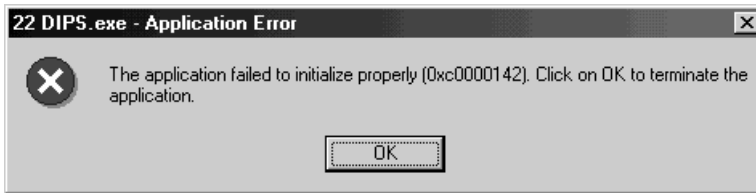
Уведомление `DLL_PROCESS_ATTACH`

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.

Обработывая `DLL_PROCESS_ATTACH`, библиотека должна выполнить в процессе инициализацию, необходимую ее 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 (*main*, *wmain*, *WinMain* или *wWinMain*). 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 показаны соответствующие окна для Windows 2000 и Windows 98.



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*. Обработывая 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`.

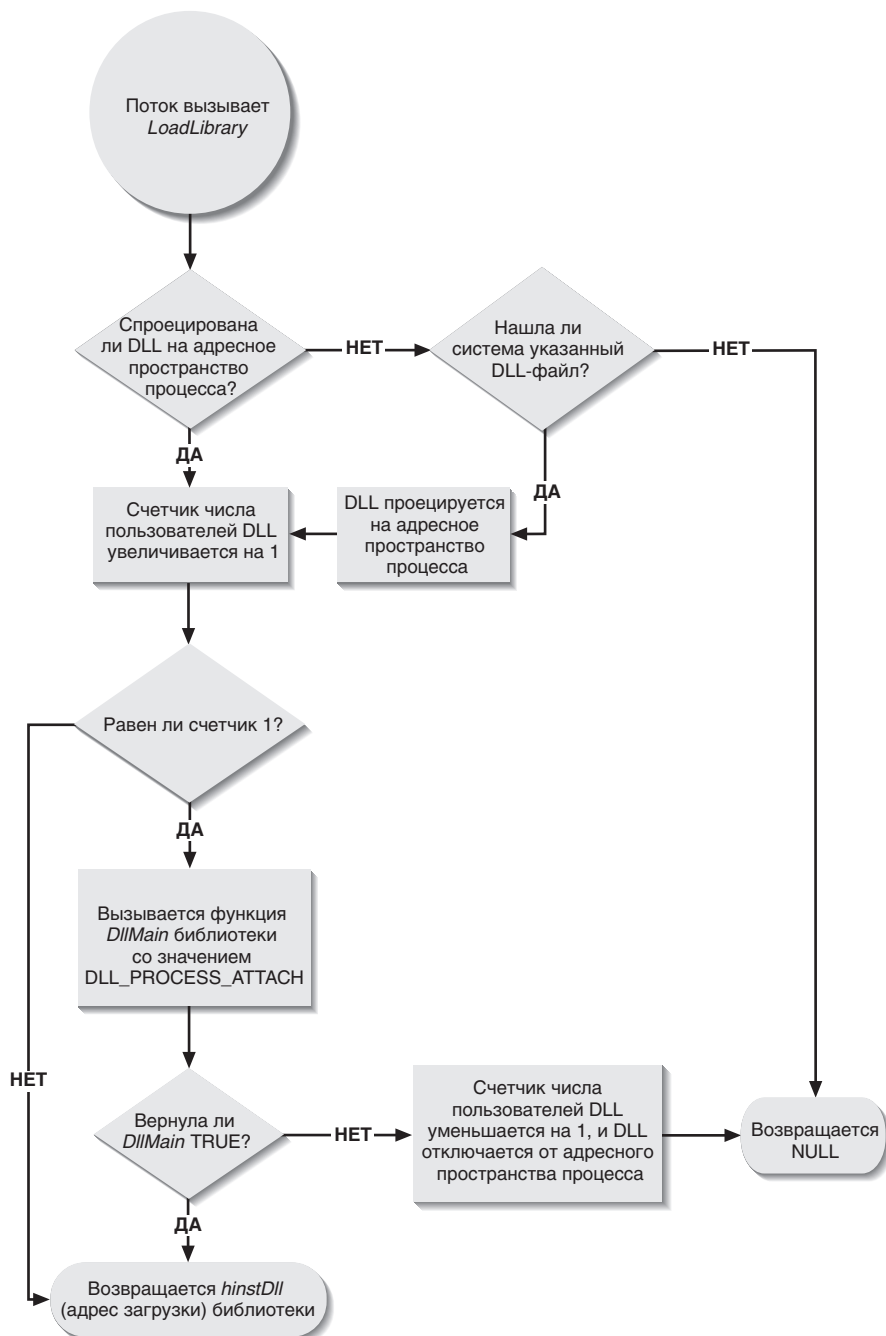


Рис. 20-2. Операции, выполняемые системой при вызове потоком функции `LoadLibrary`



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*.

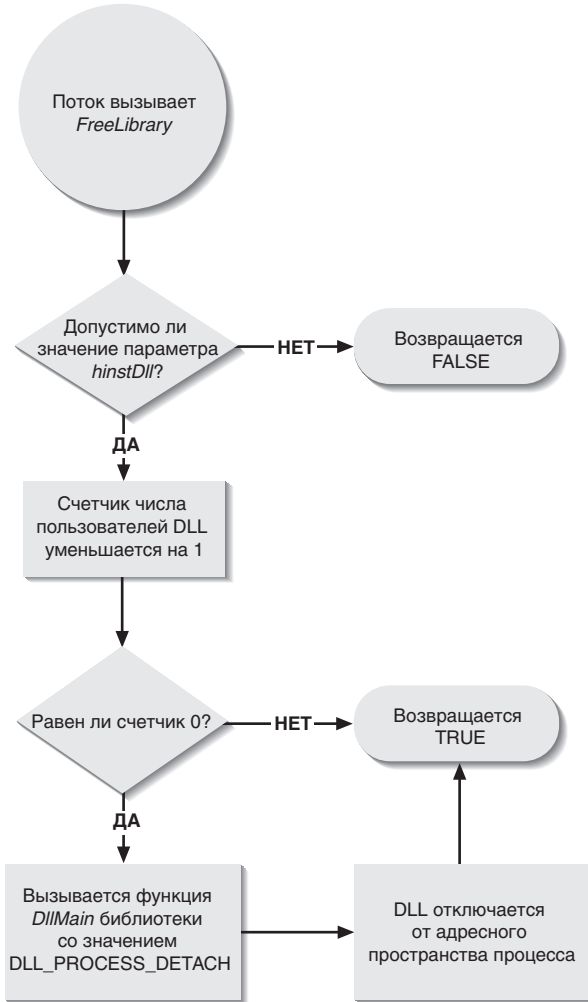


Рис. 20-3. Операции, выполняемые системой при вызове потоком функции *FreeLibrary*

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 значением `DLL_THREAD_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.



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 поток, загрузивший DLL, завершается, что приводит к новому вызову *DllMain* — на этот раз со значением `DLL_THREAD_DETACH`. Библиотека уведомляется 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 А имеет два потока: А и В. На его адресное пространство проецируется DLL-модуль *SomeDLL.dll*. Оба потока собираются вызвать *CreateThread*, чтобы создать еще два потока: С и D.

Когда поток А вызывает для создания потока С функцию *CreateThread*, система обращается к *DllMain* из *SomeDLL.dll* со значением `DLL_THREAD_ATTACH`. Пока поток С исполняет код *DllMain*, поток В 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 С не завершит обработку кода *DllMain* и не выйдет из этой функции.

Закончив выполнение *DllMain*, поток С 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(HINSTANCE hinstDll);
```

Вызывая ее, Вы сообщаете системе, что уведомления `DLL_THREAD_ATTACH` и `DLL_THREAD_DETACH` не должны посылааться *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, DWORD 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 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);
}

```

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 *WaitForSingleObject*, 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 ее 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*.

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`, функция *_DllMainCRTStartup* не делает ничего особенного. Но в случае уведомления `DLL_THREAD_DETACH`, она освобождает в потоке блок памяти *tiddata*, если он к тому времени еще не удален. Обычно в корректно написанной функции потока этот блок отсутствует, потому что она возвращает управление в *_threadstartex* из библиотеки C/C++ (см. главу 6). Функция *_threadstartex* сама вызывает *_endthreadex*, которая освобождает блок *tiddata* до того, как поток обращается к *ExitThread*.

Но представьте, что приложение, написанное на Паскале, вызывает функции из DLL, написанной на C/C++. В этом случае оно создаст поток, не прибегая к *_begin-*

threadex, и такой поток никогда не узнает о библиотеке C/C++. Далее поток вызовет функцию из DLL, которая в свою очередь обратится к библиотечной C-функции. Как Вы помните, подобные функции «на лету» создают блок *tiddata* и сопоставляют его с вызывающим потоком. Получается, что приложение, написанное на Паскале, может создавать потоки, способные без проблем обращаться к функциям из библиотеки C! Когда его функция потока возвращает управление, вызывается *ExitThread*, а библиотека C/C++ получает уведомление *DLL_THREAD_DETACH* и освобождает блок памяти *tiddata*, так что никакой утечки памяти не происходит. Здорово придумано, да?

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 так (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_THREAD_ATTACH* и *DLL_THREAD_DETACH*. Функция *DisableThreadLibraryCalls* вызывается для ускорения создания и разрушения потоков.

Отложенная загрузка DLL

Microsoft Visual C++ 6.0 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 в Windows 2000 должна использовать функции PSAPI, а в Windows 98 — ToolHelp-функции (вроде *Process32Next*). 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 98 приведет к тому, что загрузчик сообщит об ошибке, поскольку в этой системе нет модуля PSAPI.dll. Так вот, и эта проблема легко решается за счет DLL отложенной загрузки.

Я довольно долго экспериментировал с DLL отложенной загрузки в Visual C++ 6.0 и должен сказать, что Microsoft прекрасно справилась со своей задачей. DLL отложенной загрузки открывают массу дополнительных возможностей и корректно работают как в Windows 98, так и в Windows 2000.

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 пару ключей компоновщика и повторить сборку исполняемого файла. Вот эти ключи:

```
/Lib:DelayImp.lib  
/DelayLoad:MyDll.dll
```

Первый ключ заставляет компоновщик внедрить в EXE-модуль специальную функцию, `__delayLoadHelper`, 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 (.didat) со списком функций, импортируемых из MyDll.dll;
- привести вызовы функций из DLL отложенной загрузки к вызовам `__delayLoadHelper`.

При выполнении приложения вызов функции из DLL отложенной загрузки (далее для краткости — DLL-функции) фактически переадресуется к `__delayLoadHelper`. Последняя, просмотрев раздел отложенного импорта, знает, что нужно вызывать `LoadLibrary`, а затем `GetProcAddress`. Получив адрес DLL-функции, `__delayLoadHelper` 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 `__delayLoadHelper` 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 `__delayLoadHelper` 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_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND)` и `VcppException(ERROR_SEVE-`

RITY_ERROR, *ERROR_PROC_NOT_FOUND*). Они уведомляют соответственно об отсутствии DLL и DLL-функции. Моя функция филътра исключений *DelayLoadDllExceptionFilter* реагирует на оба кода. При возникновении любого другого исключения она, как и положено корректно написанному филътру, возвращает *EXCEPTION_CONTINUE_SEARCH*. (Программа не должна «глотать» исключения, которые не умеет обрабатывать.) Однако, если генерируется один из приведенных выше кодов, функция *__delayLoadHelper* 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 загрузить
    LPCSTR          szDll;       // 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-филътра 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 fImportByName;
    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 филътра исключения код ошибки, видимо, не понадобится, поскольку код исключения и так информирует о том, что произошло. Элемент *pfnCur* содержит адрес DLL-функции, и филътр исключения устанавливает его в *NULL*, так как само исключение говорит о том, что *__delayLoadHelper* 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 *__delayLoadHelper* 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 команду 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 точку выгрузки DLL вызов функции `__FUnloadDelayLoadedDLL`:

```
BOOL __FUnloadDelayLoadedDLL(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 `__FUnloadDelayLoadedDLL`, Вы передаете имя выгружаемой DLL. После этого она просматривает раздел выгрузки (unload section) и сбрасывает адреса всех DLL-функций. И, наконец, `__FUnloadDelayLoadedDLL` 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 `__FUnloadDelayLoadedDLL` 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 `__FUnloadDelayLoadedDLL` 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 `__FUnloadDelayLoadedDLL` из модуля, собранного без ключа `/Delay:unload`, ничего страшного не случится: `__FUnloadDelayLoadedDLL` 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 Microsoft внимание к деталям. Функция `__delayLoadHelper` может вызывать предоставленные Вами функции-ловушки (hook functions), и они будут получать уведомления о том, как идет выполнение `__delayLoadHelper`, 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 `__delayLoadHelper`, нужно внести два изменения в свой 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 `__delayLoadHelper`. Если Вы хотите изменить поведение `__delayLoadHelper`, начните с `DliHook` и модифицируйте ее код так, как Вам требуется. Потом передайте ее адрес функции `__delayLoadHelper`.

В статически подключаемой библиотеке DelayImp.lib определены две глобальные переменные типа *PfnDliHook*: *__pfnDliNotifyHook* и *__pfnDliFailureHook*:

```
typedef FARPROC (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 *__delayLoadHelper*, 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 __pfnDliNotifyHook = DliHook;
PfnDliHook __pfnDliFailureHook = 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.

Механизм отложенной загрузки DLL, введенный в Visual C++ 6.0, — вещь весьма интересная, и я знаю многих разработчиков, которые давно мечтали о нем. Он будет полезен в очень большом числе приложений (особенно от Microsoft).

Программа-пример DelayLoadApp

Эта программа, «20 DelayLoadApp.exe» (см. листинг на рис. 20-6), показывает, как использовать все преимущества DLL отложенной загрузки. Для демонстрации нам понадобится небольшой DLL-файл; он находится в каталоге 20-DelayLoadLib на компакт-диске, прилагаемом к книге.

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» не загружен

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 `__FUnloadDelayLoadedDLL`, 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

```

/*****
Модуль: DelayLoadApp.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <Windowsx.h>
#include <tchar.h>

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

#include <Delayimp.h>      // для обработки ошибок и доступа
                        // к дополнительной функциональности
#include "..\20-DelayLoadLib\DelayLoadLib.h" // прототипы функций в моей DLL

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

// статическое связывание с __delayLoadHelper и __FUnloadDelayLoadedDLL
#pragma comment(lib, "Delayimp.lib")

// сообщаем компоновщику, что загрузка DLL должна быть отложена;
// обратите внимание на две группы символов (\"), введенных
// из-за пробела в имени файла
#pragma comment(linker, "/DelayLoad:\\"20 DelayLoadLib.dll\\")

// сообщаем компоновщику, что DLL должна при необходимости выгружаться
#pragma comment(linker, "/Delay:unload")

// сообщаем компоновщику, что связывание DLL отложенной загрузки не требуется;
// но обычно это все же требуется, поэтому я закомментировал следующую строку
//#pragma comment(linker, "/Delay:nobind")

```

Рис. 20-6. Программа-пример DelayLoadApp

см. след. стр.

Рис. 20-6. *продолжение*

```
// имя модуля отложенной загрузки (используется только этой программой)
TCHAR g_szDelayLoadModuleName[] = TEXT("20 DelayLoadLib");

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

// упреждающий прототип функции
LONG WINAPI DelayLoadDllExceptionFilter(PEXCEPTION_POINTERS pep);

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

void IsModuleLoaded(PCTSTR pszModuleName) {

    HMODULE hmod = GetModuleHandle(pszModuleName);
    char sz[100];
#ifdef UNICODE
    wsprintfA(sz, "Module \"%S\" is %Sloaded.",
        pszModuleName, (hmod == NULL) ? L"not " : L "");
#else
    wsprintfA(sz, "Module \"%s\" is %sloaded.",
        pszModuleName, (hmod == NULL) ? "not " : "");
#endif
    chMB(sz);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    // помещаем все вызовы функций DLL отложенной загрузки в SEH-фрейм
    __try {
        int x = 0;

        // если Вы запустили программу под отладчиком, попробуйте выбрать
        // из меню Debug новую команду Modules, и Вы убедитесь, что
        // до выполнения следующей строки наша DLL еще не загружалась
        IsModuleLoaded(g_szDelayLoadModuleName);

        x = fnLib(); // пытаемся вызвать DLL-функцию

        // выберите из меню Debug команду Modules, чтобы убедиться: DLL загружена
        IsModuleLoaded(g_szDelayLoadModuleName);

        x = fnLib2(); // пытаемся вызвать DLL-функцию

        // Выгружаем DLL отложенной загрузки.
        // Примечание: имя должно быть идентично указанному в /DelayLoad.
        __FUnloadDelayLoadedDLL("20 DelayLoadLib.dll");

        // выберите из меню Debug команду Modules, чтобы убедиться: DLL выгружена
        IsModuleLoaded(g_szDelayLoadModuleName);

        x = fnLib(); // пытаемся вызвать DLL-функцию
```


Рис. 20-6. *продолжение*

```

        // выберите из меню Debug команду Modules, чтобы убедиться:
        // DLL снова загружена
        IsModuleLoaded(g_szDelayLoadModuleName);
    }
    __except (DelayLoadDllExceptionHandler(GetExceptionInformation())) {
        // здесь нам делать нечего, поток продолжает работу
    }

    // сюда можно добавить еще чуточку кода...

    return(0);
}

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

LONG WINAPI DelayLoadDllExceptionHandler(PEXCEPTION_POINTERS pep) {

    // предполагаем, что мы распознаем это исключение
    LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;

    // если проблема связана с отложенной загрузкой, ExceptionInformation[0]
    // указывает на структуру DelayLoadInfo, в которой содержится
    // подробная информация об ошибке
    PDelayLoadInfo pdli =
        PDelayLoadInfo(pep->ExceptionRecord->ExceptionInformation[0]);

    // создаем буфер, в котором мы будем формировать сообщения об ошибках
    char sz[500] = { 0 };

    switch (pep->ExceptionRecord->ExceptionCode) {
    case VcppException(ERROR_SEVERITY_ERROR, ERROR_MOD_NOT_FOUND):
        // DLL найти не удалось
        wsprintfA(sz, "Dll not found: %s", pdli->szDll);
        break;

    case VcppException(ERROR_SEVERITY_ERROR, ERROR_PROC_NOT_FOUND):
        // DLL найдена, но нужной функции в ней нет
        if (pdli->dlp.fImportByName) {
            wsprintfA(sz, "Function %s was not found in %s",
                pdli->dlp.szProcName, pdli->szDll);
        } else {
            wsprintfA(sz, "Function ordinal %d was not found in %s",
                pdli->dlp.dwOrdinal, pdli->szDll);
        }
        break;

    default:
        // это исключение мы не распознаем
        lDisposition = EXCEPTION_CONTINUE_SEARCH;
        break;
    }
}

```

см. след. стр.

Рис. 20-6. *продолжение*

```

    if (lDisposition == EXCEPTION_EXECUTE_HANDLER) {
        // мы распознали ошибку и сформировали сообщение; показываем его
        chMB(sz);
    }

    return(lDisposition);
}

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

// схематическая функция DliHook, которая не делает ничего интересного
FARPROC WINAPI DliHook(unsigned dliNotify, PDelayLoadInfo pdli) {

    FARPROC fp = NULL; // значение, возвращаемое по умолчанию

    // Примечание: элементы структуры DelayLoadInfo, на которые указывает pdli,
    // содержат сведения о ходе выполнения работы

    Switch (dliNotify) {
    Case dliStartProcessing:
        // Вызывается, когда __delayLoadHelper пытается найти DLL или функцию.
        // Возвращаем 0, если изменять поведение не требуется, и ненулевое
        // значение, если изменить поведение все же необходимо (Вы все равно
        // получите dliNoteEndProcessing).
        break;

    case dliNotePreLoadLibrary:
        // Вызывается непосредственно перед вызовом LoadLibrary.
        // Возвращаем NULL, чтобы __delayLoadHelper вызвала LoadLibrary.
        // Но Вы можете сами вызвать LoadLibrary и вернуть HMODULE.
        fp = (FARPROC) (HMODULE) NULL;
        break;

    case dliFailLoadLib:
        // Вызывается, если LoadLibrary терпит неудачу.
        // И на этот раз Вы можете сами вызвать LoadLibrary и вернуть HMODULE.
        // Если Вы вернете NULL, __delayLoadHelper возбудит исключение
        // ERROR_MOD_NOT_FOUND.
        fp = (FARPROC) (HMODULE) NULL;
        break;

    case dliNotePreGetProcAddress:
        // Вызывается непосредственно перед вызовом GetProcAddress.
        // Возвращаем NULL, чтобы __delayLoadHelper вызвала GetProcAddress.
        // Но Вы можете сами вызвать GetProcAddress и вернуть адрес.
        fp = (FARPROC) NULL;
        break;

    case dliFailGetProcAddress:
        // Вызывается, если GetProcAddress терпит неудачу.
        // Вы можете сами вызвать GetProcAddress и вернуть адрес.
        // Если Вы вернете NULL, __delayLoadHelper возбудит исключение

```

Рис. 20-6. *продолжение*

```

        // ERROR_PROC_NOT_FOUND.
        fp = (FARPROC) NULL;
        break;

    case dliNoteEndProcessing:
        // Простое уведомление об окончании работы __delayLoadHelper.
        // Вы можете пользоваться элементами структуры DelayLoadInfo,
        // на которую указывает pdli, и при необходимости возбудить исключение.
        break;
    }

    return(fp);
}

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

// сообщаем __delayLoadHelper вызвать мою функцию-ловушку
PfnDliHook __pfnDliNotifyHook = DliHook;
PfnDliHook __pfnDliFailureHook = DliHook;

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// конец файла

```

DelayLoadLib.cpp

```

/*****
Модуль: DelayLoadLib.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <Windowsx.h>
#include <tchar.h>

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

#define DELAYLOADLIBAPI extern "C" __declspec(dllexport)
#include "DelayLoadLib.h"

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

int fnLib() {
    return(321);
}

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

int fnLib2() {
    return(123);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// конец файла

```

см. след. стр.

Рис. 20-6. *продолжение*

DelayLoadLib.h

```

/*****
Модуль: DelayLoadLib.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#ifndef DELAYLOADLIBAPI
#define DELAYLOADLIBAPI extern "C" __declspec(dllimport)
#endif

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

DELAYLOADLIBAPI int fnLib();
DELAYLOADLIBAPI int fnLib2();

//////////////////////////////////// Конец файла //////////////////////////////////

```

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 2000, Вы среди прочей информации увидите и следующее.

```

C:\winnt\system32>DumpBin -Exports Kernel32.dll
(часть вывода опущена)
360 167 HeapAlloc (forwarded to NTDLL.RtlAllocateHeap)
361 168 HeapCompact (000128D9)
362 169 HeapCreate (000126EF)
363 16A HeapCreateTagsW (0001279E)
364 16B HeapDestroy (00012750)
365 16C HeapExtend (00012773)
366 16D HeapFree (forwarded to NTDLL.RtlFreeHeap)
367 16E HeapLock (000128ED)
368 16F HeapQueryTagW (000127B8)
369 170 HeapReAlloc (forwarded to NTDLL.RtlReAllocateHeap)
370 171 HeapSize (forwarded to NTDLL.RtlSizeHeap)
(остальное тоже опущено)

```

Здесь есть четыре переадресованные функции. Всякий раз, когда Ваше приложение вызывает *HeapAlloc*, *HeapFree*, *HeapReAlloc* или *HeapSize*, 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 *HeapAlloc*, программа фактически вызывает функцию *RtlAllocateHeap* из NTDLL.dll. А функции *HeapAlloc* вообще нет!

При вызове *HeapAlloc* (см. ниже) функция *GetProcAddress* просмотрит раздел экспорта Kernel32.dll и, выяснив, что *HeapAlloc* — переадресуемая функция, рекурсивно вызовет сама себя для поиска *RtlAllocateHeap* в разделе экспорта NTDLL.dll.

```
GetProcAddress(GetModuleHandle("Kernel32"), "HeapAlloc");
```

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")
```

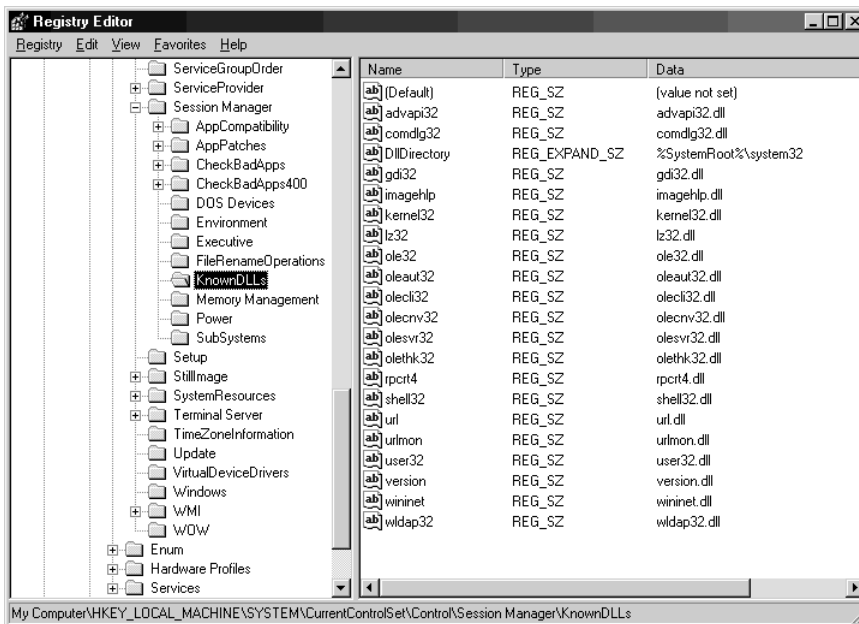
Эта директива сообщает компоновщику, что DLL должна экспортировать функцию *SomeFunc*, которая на самом деле реализована как функция *SomeOtherFunc* в модуле *DllWork.dll*. Такая запись нужна для каждой переадресуемой функции.

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 2000 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("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("SomeLib.dll");
```

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

WINDOWS 98 Windows 98 не поддерживает перенаправление DLL.

Когда разрабатывались первые версии Windows, оперативная память 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 Microsoft 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 Microsoft 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 Microsoft ввела в Windows 2000 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.

Перенаправление DLL исключительно полезно для работы с зарегистрированными СОМ-объектами. Оно позволяет приложению размещать DLL с СОМ-объектами в своем каталоге, и другие программы, регистрирующие те же объекты, не будут мешать его нормальной работе.

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 Binary File Dumper Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. 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
3588004A time date stamp Wed Jun 17 10:43:38 1998
0 file pointer to symbol table
0 number of symbols
E0 size of optional header
10F characteristics
    Relocations stripped
    Executable
    Line numbers stripped
    Symbols stripped
    32 bit word machine
```

OPTIONAL HEADER VALUES

```
10B magic #
6.00 linker version
1000 size of code
2000 size of initialized data
0 size of uninitialized data
1320 RVA of entry point
1000 base of code
2000 base of data
400000 image base    <-- предпочтительный базовый адрес модуля
1000 section alignment
```

см. след. стр.

```

1000 file alignment
4.00 operating system version
0.00 image version
4.00 subsystem version
    0 Win32 version
4000 size of image
1000 size of headers
127E2 checksum
    3 subsystem (Windows CUI)
    0 DLL characteristics
100000 size of stack reserve
1000 size of stack commit
:

```

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:

```
MOV [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.

О'кэй, 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.

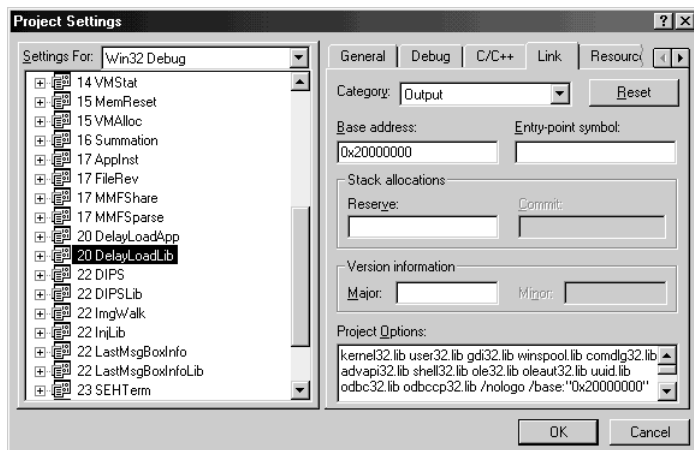
Кстати, Вы можете создать EXE- или DLL-модуль без раздела переадресации, указав при сборке ключ `/FIXED` компоновщика. Тогда у модуля будет меньший размер, но загрузить его по другому базовому адресу, кроме предпочтительного, уже не удастся. Если загрузчику понадобится модифицировать адреса в модуле, 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 IMAGE_FILE_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 лишь в 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.



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.

О'кэй, 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. 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* для каждого указанного файла. Вот что представляет собой эта функция:

```

BOOL ReBaseImage(
    PSTR CurrentImageName, // полное имя обрабатываемого файла
    PSTR SymbolPath,       // символьный путь к файлу (необходим для
                          // корректности отладочной информации)
    BOOL fRebase,          // TRUE = выполнить реальную модификацию адреса;
                          // FALSE = имитировать такую модификацию
    BOOL fRebaseSysFileOk, // FALSE = не модифицировать адреса системных файлов
    BOOL fGoingDown,       // TRUE = модифицировать адрес модуля,
                          // продвигаясь в сторону уменьшения адресов
    ULONG CheckImageSize,  // ограничение на размер получаемого в итоге модуля
    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.

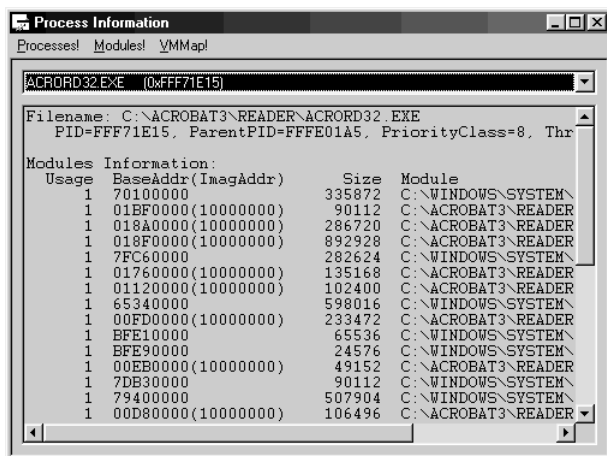
1. Моделирует создание адресного пространства процесса.
2. Открывает все модули, которые загружались бы в это адресное пространство, и получает предпочтительный базовый адрес и размер каждого модуля.
3. Моделирует переадресацию модулей в адресном пространстве, добиваясь того, чтобы модули не перекрывались.
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 BaseAddr расположена колонка ImageAddr. 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 ImageAddr показывается базовый адрес, взятый из заголовка его файла на диске.

Ниже приведена информация о процессе Acord32.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, 0x10000000; 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 потребоваться 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. 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. Вот что представляет собой эта функция:

```
BOOL BindImageEx(
    DWORD dwFlags,           // управляющие флаги
    PSTR pszImageName,       // полное имя обрабатываемого файла
    PSTR pszDllPath,         // путь для поиска образов файлов
    PSTR pszSymbolPath,      // путь для поиска отладочной информации
    PIMAGEHLP_STATUS_ROUTINE StatusRoutine); // 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
    PSTR pszImageName,           // полное имя обрабатываемого файла
    PSTR pszDllName,             // полное имя DLL
    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 заголовки, 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 [36E449E0] Mon Mar 08 14:06:24 1999
Bound to MSVCRT.dll [36BB8379] Fri Feb 05 15:49:13 1999
Bound to ADVAPI32.dll [36E449E1] Mon Mar 08 14:06:25 1999
Bound to KERNEL32.dll [36DDAD55] Wed Mar 03 13:44:53 1999
Bound to GDI32.dll [36E449E0] Mon Mar 08 14:06:24 1999
Bound to USER32.dll [36E449E0] Mon Mar 08 14:06:24 1999
```

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.

О'кэй, 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 98, Windows NT или Windows 2000) 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 98 и Windows 2000, 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. Microsoft следовало бы поставлять с операционной системой утилиту, которая автоматически проводила бы повторное связывание всех модулей после обновления системы. Но, увы, такой утилиты нет.

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 *strtok*. При первом вызове она получает адрес строки и запоминает его в собственной статической переменной. Когда при следующих вызовах *strtok* Вы передаете ей NULL, она оперирует с адресом, записанным в своей переменной.

В многопоточной среде вероятна такая ситуация: один поток вызывает *strtok*, и, не успев 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 *strtok*, 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 *strtok*. 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 *strtok* 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.

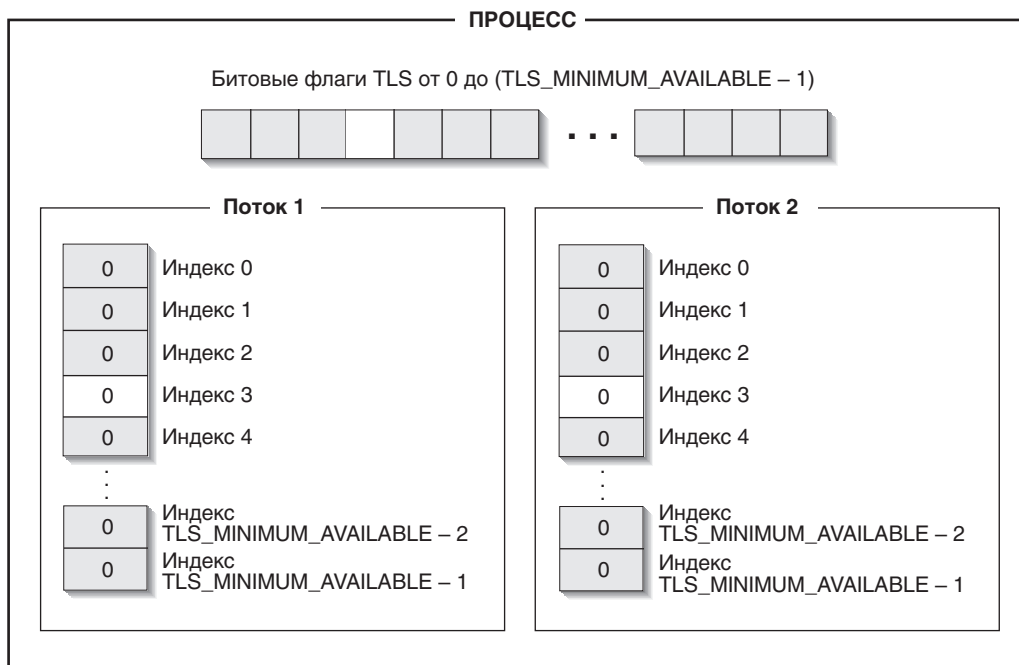


Рис. 21-1. Внутренние структуры данных, предназначенные для управления локальной памятью потока

Каждый флаг выполняемого в системе процесса может находиться в состоянии FREE или INUSE, указывая, свободна или занята данная область локальной памяти потока (TLS-область). Microsoft гарантирует доступность по крайней мере TLS_MINIMUM_AVAILABLE 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();
```

Она заставляет систему сканировать битовые флаги в текущем процессе и искать флаг FREE. Отыскав, система меняет его на INUSE, 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_INDEXES (определенный в файле WinBase.h как 0xFFFFFFFF).

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 поток (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 *dwTlsIndex*, значение типа PVOID, содержащееся в параметре *pvTlsValue*. Содержимое *pvTlsValue* сопоставляется с потоком, вызвавшим *TlsSetValue*. В случае успеха возвращается TRUE.

Обращаясь к *TlsSetValue*, 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.

Вызывая *TlsSetValue*, будьте осторожны и передавайте только тот индекс, который получен предыдущим вызовом *TlsAlloc*. Чтобы максимально увеличить быстродействие этих функций, Microsoft 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 *TlsGetValue*:

```
PVOID TlsGetValue(DWORD dwTlsIndex);
```

Она возвращает значение, сопоставленное с TLS-областью под индексом *dwTlsIndex*. Как и *TlsSetValue*, функция *TlsGetValue* обращается только к массиву, который принадлежит вызывающему потоку. Она тоже не контролирует 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 в DLL.

Вот один из способов работы с TLS-памятью: Вы создаете ее только по необходимости. Например, в DLL может быть функция, работающая аналогично *strtok*. При первом ее вызове поток передает этой функции указатель на 40-байтовую структуру, которую надо сохранить, чтобы сослаться 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) TlsGetValue(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, напомним: приложение может динамически подключать несколько DLL. Одна DLL займет, допустим, 10 TLS-индексов, 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-байтовую 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 2000 количество TLS-областей увеличено до более чем 1000. Microsoft 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 *TlsAlloc*. Взгляните на фрагмент кода:

```
DWORD dwTlsIndex;
PVOID pvSomeValue;
:
dwTlsIndex = TlsAlloc();
TlsSetValue(dwTlsIndex, (PVOID) 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 невозможно.

Статическая локальная память потока

Статическая локальная память потока основана на 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 лишь 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_`.

Обработывая 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 дополнительных машинных 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 программа, 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.

Это слишком большая нагрузка на операционную систему. Кроме того, допуская явную загрузку DLL, содержащих статические TLS-переменные, система не в состоянии должным образом инициализировать TLS-данные, что при попытке обращения к ним может вызвать нарушение доступа. Это, пожалуй, единственный недостаток статической TLS; при использовании динамической TLS такой проблемы нет. DLL, работающие с динамической TLS, могут загружаться и выгружаться из выполняемой программы в любой момент и без всяких проблем.

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, принадлежащую 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.

**WINDOWS
98**

В Windows 98 процессы фактически совместно используют 2 Гб адресного пространства (от 0x80000000 до 0xFFFFFFFF). На этот регион отображаются только системные компоненты и файлы, проецируемые в память (подробнее на эту тему см. главы 13, 14 и 17).

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 А 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 А. 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.



Рис. 22-1. Поток процесса В пытается создать подкласс окна, сформированного потоком процесса А

Теперь допустим, что процесс В хочет создать подкласс окна, порожденного одним из потоков процесса А. Сначала код процесса В должен определить описатель этого окна, что можно сделать самыми разными способами. В примере на рис. 22-1 поток процесса В просто вызывает *FindWindow*, затем — *SetWindowLongPtr*, 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 А извлек 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 В, а активен — процесс А. Очевидно, если бы *User32* обратился по данному адресу, то на самом деле он обратился бы к какому-то участку памяти в адресном пространстве процесса А, что, естественно, привело бы к нарушению доступа к памяти.

Чтобы избежать этого, было бы неплохо сообщить системе, что *MySubclassProc* находится в адресном пространстве процесса В, 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 В, 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, Microsoft 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 А, это позволило бы вызвать *SetWindowLongPtr* и передать ей адрес *MySubclassProc* в процессе А. Я называю такой прием внедрением (injecting) DLL в адресное пространство процесса. Мне известно несколько способов подобного внедрения. Рассмотрим их по порядку, начиная с простейшего.

Внедрение DLL с использованием реестра

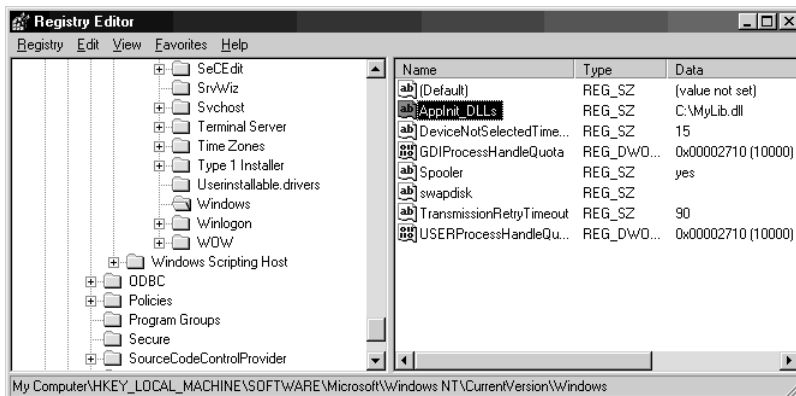
Если Вы уже работали с Windows, то знаете, что такое реестр. В нем хранится конфигурация всей системы, и, модифицируя в реестре те или иные параметры, можно изменить поведение системы. Я намерен поговорить о параметре реестра:

```
HKEY_LOCAL_MACHINE\Software\Microsoft
\Windows NT\CurrentVersion\Windows\AppInit_DLLs
```

WINDOWS 98 Windows 98 игнорирует этот параметр реестра, поэтому для нее такой способ внедрения DLL не работает.

Список параметров в разделе реестра, где находится *AppInit_DLLs*, можно просмотреть с помощью программы Registry Editor (Редактор реестра). Значением параметра *AppInit_DLLs* 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 указал 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 *fvdReason*, 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. Правда, в Windows 2000 модуль User32.dll ведет себя несколько иначе, но об этом — чуть позже.

Это простейший способ внедрения DLL. Все, что от Вас требуется, — добавить значение в уже существующий параметр реестра. Однако он не лишен недостатков.

- Так как система считывает значение параметра при инициализации, после его изменения придется перезагружать компьютер. Выход и повторный вход в систему не сработает — Вы должны перезагрузить компьютер. Впрочем, сказанное относится лишь к Windows NT версии 4.0 (или ниже). В Windows 2000 модуль User32.dll повторно считывает параметр реестра AppInit_DLLs при каждой загрузке в процесс, и перезапуска системы не требуется.
- 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 заикнется или некорректно обратится к памяти, Вы повлияете на поведение и устойчивость соответствующих про-

цессов. 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 WordPad в тот момент, когда пользователь запускает Ваше приложение. Естественно, пока пользователь не откроет Ваше приложение, внедрять DLL в адресное пространство WordPad не требуется. Когда пользователь закроет Ваше приложение, целесообразно отменить переопределение оконной процедуры WordPad. И в этом случае DLL тоже не зачем «держат» в адресном пространстве WordPad. 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, Microsoft 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 GUI-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 А. Если же `hinstDll` различ-

ны, система определяет адрес функции *GetMsgProc* в адресном пространстве процесса В по формуле:

$\text{GetMsgProc } B = \text{hinstDll } B + (\text{GetMsgProc } A - \text{hinstDll } A)$

Вычитая *hinstDll A* из *GetMsgProc A*, Вы получаете смещение (в байтах) адреса функции *GetMsgProc*. Добавляя это смещение к *hinstDll B*, Вы получаете адрес *GetMsgProc*, соответствующий проекции DLL в адресном пространстве процесса В.

6. Счетчик блокировок проекции DLL в процессе В увеличивается на 1.
7. Вызывается *GetMsgProc* в адресном пространстве процесса В.
8. После возврата из *GetMsgProc* счетчик блокировок проекции DLL в адресном пространстве процесса В 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 В, 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 момент, когда поток процесса В 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

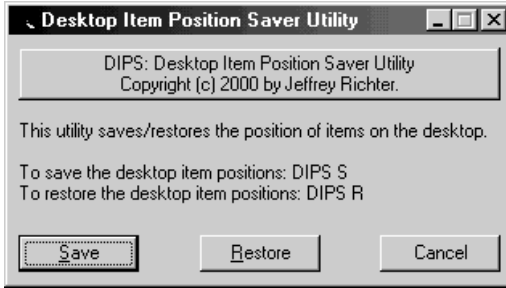
Эта утилита, «22 DIPS.exe» (см. листинг на рис. 22-2), использует ловушки окон для внедрения DLL в адресное пространство *Explorer.exe*. Файлы исходного кода и ресурсов этой программы и DLL находятся в каталогах 22-DIPS и 22-DIPSLib на компакт-диске, прилагаемом к книге.

Компьютер я использую в основном для работы, и, на мой взгляд, самое оптимальное в этом случае разрешение экрана — 1152 × 864. Иногда я запускаю на своем компьютере кое-какие игры, но большинство из них рассчитано на разрешение 640 × 480. Когда у меня появляется настроение поиграть, приходится открывать апплет *Display* в *Control Panel* и устанавливать разрешение 640 × 480, а закончив игру, вновь возвращаться в *Display* и восстанавливать разрешение 1152 × 864.

Возможность изменять экранное разрешение «на лету» — очень удобная функция *Windows*. Единственное, что мне не по душе, — при смене экранного 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 перемещать 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 *S* в командной строке, в реестре создается подраздел:

HKEY_CURRENT_USER\Software\Richter\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 перед установкой более низкого экранного разрешения. Вслась 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, *LVM_GETITEM* 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 *LVM_GETITEMPOSITION* элементу управления *ListView* рабочего стола.



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*,

см. след. стр.

и, если да, сама создает проецируемый в память файл и копирует строковые данные из адресного пространства одного процесса в адресное пространство другого.

Почему Microsoft 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, Microsoft и пошла на эти ухищрения. А поскольку в 16-разрядной Windows нет новых элементов управления, то проблемы их переноса тоже нет, и Microsoft ничего подобного для них делать не стала.

Сразу после запуска DIPS получает описатель окна элемента управления `ListView` 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_DefView`, у которого тоже одно дочернее окно — класса `SysListView32`. Оно-то и служит элементом управления `ListView` рабочего стола. (Кстати, всю эту информацию я выудил благодаря `Spy++`.)

Получив описатель окна `ListView`, 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 этом 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, fSave);
```

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 адресному 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!



Dips.cpp

```

/*****
Модуль: DIPS.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h" /* см. приложение A */
#include <WindowsX.h>
#include <tchar.h>
#include "Resource.h"
#include "..\22-DIPSLib\DIPSLib.h"

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

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

```

Рис. 22-2. Утилита DIPS

см. след. стр.

Рис. 22-2. *продолжение*

```

        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;
        }
    }

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

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

        switch (uMsg) {
            case WM_INITDIALOG:
                Dlg_OnInitDialog(hwnd, wParam, lParam);
            case WM_COMMAND:
                Dlg_OnCommand(hwnd, wParam, lParam);
        }
        return(FALSE);
    }

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

    int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

        // преобразуем символ в командной строке в верхний регистр
        CharUpperBuff(pszCmdLine, 1);
        TCHAR cWhatToDo = pszCmdLine[0];

        if ((cWhatToDo != TEXT('S')) && (cWhatToDo != TEXT('R'))) {

            // неверный аргумент в командной строке; сообщаем пользователю
            cWhatToDo = 0;
        }

        if (cWhatToDo == 0) {
            // аргумента в командной строке нет;
            // выводим диалоговое окно с инструкциями
            switch (DialogBox(hinstExe, MAKEINTRESOURCE(IDD_DIPS), NULL, Dlg_Proc)) {
                case IDC_SAVE:
                    cWhatToDo = TEXT('S');
                    break;
            }
        }
    }

```

Рис. 22-2. *продолжение*

```

        case IDC_RESTORE:
            cWhatToDo = TEXT('R');
            break;
    }
}

if (cWhatToDo == 0) {
    // пользователь не хочет ничего делать
    return(0);
}

// окно ListView рабочего стола – “внук” окна ProgMan
HWND hwndLV = GetFirstChild(GetFirstChild(
    FindWindow(TEXT("ProgMan"), NULL)));
chASSERT(IsWindow(hwndLV));

// Устанавливаем ловушку, внедряющую нашу DLL в адресное пространство
// Explorer. После этого создаем скрытое немодальное диалоговое окно.
// Мы посылаем ему сообщения, которые говорят, что делать.
chVERIFY(SetDIPSHook(GetWindowThreadProcessId(hwndLV, NULL)));

// ждем создания серверного окна DIPS
MSG msg;
GetMessage(&msg, NULL, 0, 0);

// получаем описатель скрытого диалогового окна
HWND hwndDIPS = FindWindow(NULL, TEXT("Richter DIPS"));

// проверяем, действительно ли это окно создано
chASSERT(IsWindow(hwndDIPS));

// сообщаем окну DIPS, с каким окном ListView следует работать,
// а также что делать: сохранять или восстанавливать элементы
SendMessage(hwndDIPS, WM_APP, (LPARAM) hwndLV, (cWhatToDo == TEXT('S')));

// Сообщаем окну DIPS о необходимости самоуничтожения. Вместо
// PostMessage используем SendMessage, чтобы уничтожить окно
// до снятия ловушки.
SendMessage(hwndDIPS, WM_CLOSE, 0, 0);

// проверяем, действительно ли окно уничтожено
chASSERT(!IsWindow(hwndDIPS));

// убираем ловушку, удаляя процедуру диалогового окна DIPS
// из адресного пространства процесса Explorer
SetDIPSHook(0);

return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

см. след. стр.

Рис. 22-2. *продолжение*

DIPSLib.cpp

```

/*****
Модуль: DIPSLib.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <WindowsX.h>
#include <CommCtrl.h>

#define DIPSLIBAPI __declspec(dllexport)
#include "DIPSLib.h"
#include "Resource.h"

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

#ifdef _DEBUG
// эта функция активизирует отладчик
void ForceDebugBreak() {
    __try { DebugBreak(); }
    __except(UnhandledExceptionFilter(GetExceptionInformation())) { }
}
#else
#define ForceDebugBreak()
#endif

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

// упреждающие ссылки
LRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam);

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

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

// требуем от компилятора разместить переменную g_hhook в отдельном разделе
// данных Shared, а компоновщику сообщаем, что данные из этого раздела должны
// быть доступны всем экземплярам приложения
#pragma data_seg("Shared")
HHOOK g_hhook = NULL;
DWORD g_dwThreadIdDIPS = 0;
#pragma data_seg()

// сообщаем компоновщику, что раздел Shared должен быть общим
// и доступным для чтения и записи
#pragma comment(linker, "/section:Shared,rws")

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

// неразделяемые переменные
HINSTANCE g_hinstDll = NULL;

```


Рис. 22-2. *продолжение*

```

/////////////////////////////////////////////////////////////////
BOOL WINAPI DllMain(HINSTANCE hinstDll, DWORD fdwReason, PVOID fImpLoad) {

    switch (fdwReason) {

        case DLL_PROCESS_ATTACH:
            // к адресному пространству текущего процесса подключается DLL
            g_hinstDll = hinstDll;
            break;

        case DLL_THREAD_ATTACH:
            // в текущем процессе создается поток
            break;

        case DLL_THREAD_DETACH:
            // поток корректно завершается
            break;

        case DLL_PROCESS_DETACH:
            // вызывающий процесс отключает DLL от своего
            // адресного пространства
            break;
    }
    return(TRUE);
}

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

BOOL WINAPI SetDIPSHook(DWORD dwThreadId) {

    BOOL fOk = FALSE;

    if (dwThreadId != 0) {
        // убеждаемся, что ловушка еще не установлена
        chASSERT(g_hhook == NULL);

        // сохраняем идентификатор потока в разделяемой переменной, чтобы после создания
        // окна сервера наша функция GetMsgProc могла послать асинхронный ответ потоку
        g_dwThreadIdDIPS = GetCurrentThreadId();

        // устанавливаем ловушку для указанного потока
        g_hhook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc, g_hinstDll,
            dwThreadId);

        fOk = (g_hhook != NULL);
        if (fOk) {
            // ловушка установлена; отправляем сообщение в очередь
            // потока, провоцируя вызов функции ловушки
            fOk = PostThreadMessage(dwThreadId, WM_NULL, 0, 0);
        }
    }
}

```

см. след. стр.

Рис. 22-2. *продолжение*

```

    } else {

        // убеждаемся, что ловушка действительно была установлена
        chASSERT(g_hhook != NULL);
        fOk = UnhookWindowsHookEx(g_hhook);
        g_hhook = NULL;
    }
    return(fOk);
}

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

LRESULT WINAPI GetMsgProc(int nCode, WPARAM wParam, LPARAM lParam) {

    static BOOL fFirstTime = TRUE;

    if (fFirstTime) {
        // DLL только что внедрена
        fFirstTime = FALSE;

        // раскомментируйте следующую строку, чтобы вызвать отладчик
        // для процесса, в который только что внедрена DLL
        // ForceDebugBreak();

        // создаем окно сервера DIPS, обрабатывающее клиентские запросы
        CreateDialog(g_hinstDll, MAKEINTRESOURCE(IDD_DIPS), NULL, Dlg_Proc);

        // сообщаем утилите DIPS, что сервер готов к обработке запросов
        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\\Richter\\Desktop Item Position Saver");

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

void SaveListViewItemPositions(HWND hwndLV) {

    int nMaxItems = ListView_GetItemCount(hwndLV);

    // сохраняя новые позиции, удаляем из реестра информацию о старых
    LONG l = RegDeleteKey(HKEY_CURRENT_USER, g_szRegSubKey);

```

Рис. 22-2. *продолжение*

```

// создаем нужный раздел реестра
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++) {

    // получаем имя и позицию элемента в ListView
    TCHAR szName[MAX_PATH];
    ListView_GetItemText(hwndLV, nItem, 0, szName, chDIMOF(szName));

    POINT pt;
    ListView_GetItemPosition(hwndLV, nItem, &pt);

    // сохраняем полученные данные в реестре
    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) {

        // если в ListView установлен режим автоупорядочения (Auto Arrange),
        // временно отключаем его
        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 = chDIMOF(szName);

            POINT pt;
            DWORD cbData = sizeof(pt), nItem;

            // считываем данные из реестра
            DWORD dwType;
            l = RegEnumValue(hkey, nIndex, szName, &cbValueName,
                NULL, &dwType, (PBYTE) &pt, &cbData);

            if (l == ERROR_NO_MORE_ITEMS)

```

см. след. стр.

Рис. 22-2. *продолжение*

```

        continue;

    if ((dwType == REG_BINARY) && (cbData == sizeof(pt))) {
        // если данный параметр нам известен, пытаемся найти
        // одноименный элемент в ListView
        LV_FINDINFO lvfi;
        lvfi.flags = LVFI_STRING;
        lvfi.psz = szName;
        nItem = ListView_FindItem(hwndLV, -1, &lvfi);
        if (nItem != -1) {
            // элемент найден; изменяем его позицию
            ListView_SetItemPosition(hwndLV, nItem, pt.x, pt.y);
        }
    }
}
// если раньше действовал режим автоупорядочения,
// восстанавливаем его
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:
            // раскомментируйте следующую строку, чтобы вызвать отладчик
            // для процесса, в который только что внедрена DLL
            // ForceDebugBreak();

            if (lParam)
                SaveListViewItemPositions((HWND) wParam);
            else
                RestoreListViewItemPositions((HWND) wParam);
            break;
    }
    return(FALSE);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

DIPSLib.h

```

/*****
Модуль: DIPSLib.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

```

Рис. 22-2. продолжение

```

#ifdef DIPSLIBAPI
#define DIPSLIBAPI __declspec(dllimport)
#endif

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

// прототипы внешних функций
DIPSLIBAPI BOOL WINAPI SetDIPSHook(DWORD dwThreadId);

//////////////////////////////////// Конец файла //////////////////////////////////

```

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 pdwThreadId);

```

Она идентична *CreateThread*, 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.



В Windows 2000 чаще используемая функция *CreateThread*, между прочим, реализована через вызов *CreateRemoteThread*:

```

HANDLE CreateThread(PSECURITY_ATTRIBUTES psa, DWORD dwStackSize,
    PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam,
    DWORD fdwCreate, PDWORD pdwThreadId) {

    return (CreateRemoteThread(GetCurrentProcess(), psa, dwStackSize,
        pfnStartAddr, pvParam, fdwCreate, pdwThreadId));
}

```

WINDOWS 98 В Windows 98 функция *CreateRemoteThread* определена, но не реализована и просто возвращает FALSE; последующий вызов *GetLastError* дает код ERROR_CALL_NOT_IMPLEMENTED. (Но функция *CreateThread*, которая создает поток в вызывающем процессе, реализована полностью.) Так что описываемый здесь метод внедрения DLL в Windows 98 не работает.

О'кэй, 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*:

```
HINSTANCE LoadLibrary(PCTSTR pszLibFile);
```

Заглянув в заголовочный файл WinBase.h, Вы увидите, что для *LoadLibrary* там есть такие строки:

```
HINSTANCE WINAPI LoadLibraryA(LPCSTR pszLibFileName);
HINSTANCE WINAPI LoadLibraryW(LPCWSTR pszLibFileName);
#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);
```

О'кэй, 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,
    LoadLibraryA, "C:\\MyLib.dll", 0, NULL);
```

Или, если Вы предпочитаете Unicode:

```
HANDLE hThread = CreateRemoteThread(hProcessRemote, NULL, 0,
    LoadLibraryW, L"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 шлю-

зов 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 *LoadLibraryA* в вызове *CreateRemoteThread* преобразуется в обращение к шлюзу *LoadLibraryA* 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 *LoadLibraryA*, минуя шлюз, Вы должны выяснить ее точный адрес в памяти с помощью *GetProcAddress*.

Вызов *CreateRemoteThread* 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 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);
```

Или, если Вы предпочитаете Unicode:

```
// получаем истинный адрес LoadLibraryA 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);
```

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 *LoadLibraryA*. Но, когда *LoadLibraryA* 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 необработываемом 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,
    PVOID pvAddressRemote,
    PVOID pvBufferLocal,
    DWORD dwSize,
    PDWORD pdwNumBytesRead);

BOOL WriteProcessMemory(
    HANDLE hProcess,
    PVOID pvAddressRemote,
    PVOID pvBufferLocal,
    DWORD dwSize,
    PDWORD 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 *CreateRemoteThread*, создайте в удаленном процессе поток, который вызовет *FreeLibrary* с передачей *HINSTANCE* внедренной DLL.

Вот, собственно, и все. Единственный недостаток этого метода внедрения DLL (самого универсального из уже рассмотренных) — многие нужные функции в Windows 98 не поддерживаются. Так что данный метод применим только в Windows 2000.

Программа-пример InjLib

Эта программа, «22 InjLib.exe» (см. листинг на рис. 22-3), внедряет DLL с помощью функции *CreateRemoteThread*. Файлы исходного кода и ресурсов этой программы и DLL находятся в каталогах 22-InjLib и 22-ImgWalk на компакт-диске, прилагаемом к книге. 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 CreateRemoteThread
    PROCESS_VM_OPERATION  | // 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. Функция *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

```

/*****
Модуль: InjLib.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <stdio.h>
#include <tchar.h>
#include <malloc.h>         // для доступа к alloca
#include <TlHelp32.h>
#include "Resource.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 fOk = FALSE; // считаем, что функция потерпит неудачу
    HANDLE hProcess = NULL, hThread = NULL;
    PWSTR pszLibFileRemote = NULL;

    __try {
        // получаем дескриптор целевого процесса
        hProcess = OpenProcess(
            PROCESS_CREATE_THREAD | // для CreateRemoteThread
            PROCESS_VM_OPERATION  | // для VirtualAllocEx/VirtualFreeEx
            PROCESS_VM_WRITE,      // для WriteProcessMemory
            FALSE, dwProcessId);
        if (hProcess == NULL) __leave;
    }
}

```

Рис. 22-3. Программа-пример *InjLib*

Рис. 22-3. *продолжение*

```

    // определяем, сколько байтов нужно для строки с полным именем DLL
    int cch = 1 + lstrlenW(pszLibFile);
    int cb = cch * sizeof(WCHAR);

    // выделяем блок памяти под эту строку
    pszLibFileRemote = (PWSTR)
        VirtualAllocEx(hProcess, NULL, cb, MEM_COMMIT, PAGE_READWRITE);
    if (pszLibFileRemote == NULL) __leave;

    // копируем эту строку в адресное пространство удаленного процесса
    if (!WriteProcessMemory(hProcess, pszLibFileRemote,
        (PVOID) pszLibFile, cb, NULL)) __leave;

    // получаем истинный адрес LoadLibraryW в Kernel32.dll
    PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
        GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "LoadLibraryW");
    if (pfnThreadRtn == NULL) __leave;

    // создаем удаленный поток, вызывающий LoadLibraryW
    hThread = CreateRemoteThread(hProcess, NULL, 0,
        pfnThreadRtn, pszLibFileRemote, 0, NULL);
    if (hThread == NULL) __leave;

    // ждем завершения удаленного потока
    WaitForSingleObject(hThread, INFINITE);

    fOk = TRUE; // все прошло успешно
}
__finally { // проводим очистку

    // освобождаем память, выделенную под строку для полного имени DLL
    if (pszLibFileRemote != NULL)
        VirtualFreeEx(hProcess, pszLibFileRemote, 0, MEM_RELEASE);

    if (hThread != NULL)
        CloseHandle(hThread);

    if (hProcess != NULL)
        CloseHandle(hProcess);
}
return(fOk);
}

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

BOOL WINAPI InjectLibA(DWORD dwProcessId, PCSTR pszLibFile) {

    // выделяем буфер (в стеке) для Unicode-строки с полным именем DLL
    PWSTR pszLibFileW = (PWSTR)
        _alloca((lstrlenA(pszLibFile) + 1) * sizeof(WCHAR));

```

см. след. стр.

Рис. 22-3. *продолжение*

```
// преобразуем ANSI-строку в Unicode
wprintfW(pszLibFileW, L"%S", pszLibFile);

// вызываем Unicode-версию функции, которая, собственно, и выполняет работу
return(InjectLibW(dwProcessId, pszLibFileW));
}

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

BOOL WINAPI EjectLibW(DWORD dwProcessId, PCWSTR pszLibFile) {

    BOOL fOk = FALSE; // считаем, что функция потерпит неудачу
    HANDLE hthSnapshot = NULL;
    HANDLE hProcess = NULL, hThread = NULL;

    __try {
        // получаем новый "моментальный снимок" процесса
        hthSnapshot = CreateToolhelp32Snapshot(TH32CS_SNAPMODULE, dwProcessId);
        if (hthSnapshot == NULL) __leave;

        // получаем HMODULE требуемой DLL
        MODULEENTRY32W me = { sizeof(me) };
        BOOL fFound = FALSE;
        BOOL fMoreMods = Module32FirstW(hthSnapshot, &me);
        for (; fMoreMods; fMoreMods = Module32NextW(hthSnapshot, &me)) {
            fFound = (lstrcmpiW(me.szModule, pszLibFile) == 0) ||
                (lstrcmpiW(me.szExePath, pszLibFile) == 0);
            if (fFound) break;
        }
        if (!fFound) __leave;

        // получаем описатель целевого процесса
        hProcess = OpenProcess(
            PROCESS_CREATE_THREAD |
            PROCESS_VM_OPERATION, // для CreateRemoteThread
            FALSE, dwProcessId);
        if (hProcess == NULL) __leave;

        // получаем истинный адрес LoadLibraryW в Kernel32.dll
        PTHREAD_START_ROUTINE pfnThreadRtn = (PTHREAD_START_ROUTINE)
            GetProcAddress(GetModuleHandle(TEXT("Kernel32")), "FreeLibrary");
        if (pfnThreadRtn == NULL) __leave;

        // создаем удаленный поток, вызывающий LoadLibraryW
        hThread = CreateRemoteThread(hProcess, NULL, 0,
            pfnThreadRtn, me.modBaseAddr, 0, NULL);
        if (hThread == NULL) __leave;

        // ждем завершения удаленного потока
        WaitForSingleObject(hThread, INFINITE);
    }
}
```

Рис. 22-3. *продолжение*

```

        fOk = TRUE; // все прошло успешно
    }
    __finally { // проводим очистку

        if (hthSnapshot != NULL)
            CloseHandle(hthSnapshot);

        if (hThread != NULL)
            CloseHandle(hThread);

        if (hProcess != NULL)
            CloseHandle(hProcess);
    }
    return(fOk);
}

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

BOOL WINAPI EjectLibA(DWORD dwProcessId, PCSTR pszLibFile) {

    // выделяем буфер (в стеке) для Unicode-строки с полным именем DLL
    PWSTR pszLibFileW = (PWSTR)
        _alloca((lstrlenA(pszLibFile) + 1) * sizeof(WCHAR));

    // преобразуем ANSI-строку в Unicode
    wprintfW(pszLibFileW, L"%S", pszLibFile);

    // вызываем Unicode-версию функции, которая, собственно, и выполняет работу
    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);

```

см. след. стр.

Рис. 22-3. *продолжение*

```

        if (dwProcessId == 0) {
            // если идентификатор процесса равен 0, DLL
            // внедряется в локальный процесс (это облегчает отладку)
            DwProcessId = GetCurrentProcessId();
        }

        TCHAR szLibFile[MAX_PATH];
        GetModuleFileName(NULL, szLibFile, sizeof(szLibFile));
        _tcscpy(_tcsrchr(szLibFile, TEXT('\\')) + 1, 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) {

    chWindows9xNotAllowed();
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_INJLIB), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конец файла //////////////////////////////////

```

Библиотека ImgWalk.dll

ImgWalk.dll (см. листинг на рис. 22-4) — 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 на компакт-диске, прилагаемом к книге. 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 `MEMORY_BASIC_INFORMATION`. На каждой итерации цикла `ImgWalk` проверяет, нет ли строки с полным именем файла, которую можно было бы добавить в список, выводимый на экран.

```
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)) {
        // Имя модуля не включается в список, если
        // истинно хотя бы одно из следующих условий:
        // 1. Данный регион содержит нашу DLL.
        // 2. Данный блок НЕ является началом региона.
        // 3. Адрес равен NULL.
        nLen = 0;
    } else {
        nLen = GetModuleFileNameA((HINSTANCE) mbi.AllocationBase,
            szModName, chDIMOF(szModName));
    }

    if (nLen > 0) {
        wsprintfA(strchr(szBuf, 0), "\\n%08X-%s",
            mbi.AllocationBase, szModName);
    }
    pb += mbi.RegionSize;
}
chMB(&szBuf[1]);
```

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.

ImgWalk.cpp

```

/*****
Модуль: ImgWalk.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение 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)) {
                // Имя модуля не включается в список, если
                // истинно хотя бы одно из следующих условий:
                // 1. Данный регион содержит нашу DLL.
                // 2. Данный блок НЕ является началом региона.
                // 3. Адрес равен NULL.
                nLen = 0;
            } else {
                nLen = GetModuleFileNameA((HINSTANCE) mbi.AllocationBase,
                    szModName, chDIMOF(szModName));
            }

            if (nLen > 0) {
                wsprintfA(strchr(szBuf, 0), "\\n%p-%s",

```

Рис. 22-4. Исходный код *ImgWalk.dll*

Рис. 22-4. *продолжение*

```

        mbi.AllocationBase, szModName);
    }

    pb += mbi.RegionSize;
}
chMB(&szBuf[1]);
}

return(TRUE);
}

/////////////////////////////////////// Конец файла /////////////////////////////////////////

```

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 Microsoft 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-модюлю. 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 зависить 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. Избежать этого нельзя.

Внедрение кода в среде Windows 98 через проецируемый в память файл

Эта задача в Windows 98, по сути, тривиальна. В ней все 32-разрядные приложения делят верхние два гигабайта своих адресных пространств. Выделенный там блок памяти доступен любому приложению. С этой целью Вы должны использовать проецируемые в память файлы (см. главу 17). Сначала Вы создаете проекцию файла, а потом вызываете *MapViewOfFile* и делаете ее видимой. Далее Вы записываете нужную информацию в эту область своего адресного пространства (она одинакова во всех адресных пространствах). Чтобы все это работало, Вам, вероятно, придется вручную писать машинные коды, а это затруднит перенос программы на другую процессорную платформу. Но вряд ли это должно Вас волновать — все равно Windows 98 работает только на процессорах типа x86.

Данный метод тоже довольно труден, потому что Вам нужно будет заставить поток другого процесса выполнять код в проекции файла. Для этого понадобятся какие-то средства управления удаленным потоком. Здесь пригодилась бы функция *CreateRemoteThread*, но Windows 98 ее не поддерживает. Увы, готового решения этой проблемы у меня нет.

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 за тем, 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 внед-

ренной 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!

WINDOWS 98 В Windows 98 основные системные DLL (Kernel32, AdvAPI32, User32 и GDI32) защищены так, что приложение не может что-либо изменить на их страницах кода. Это ограничение можно обойти, только написав специальный драйвер виртуального устройства (VxD).

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 ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
    PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) {
```

```

ULONG ulSize;
PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)
    ImageDirectoryEntryToData(hmodCaller, TRUE,
        IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);

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)
        break;
}

if (pImportDesc->Name == 0)
    // этот модуль не импортирует никаких функций из данной DLL
    return;

// 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 fFound = (*ppfn == pfnCurrent);

    // см. текст программы-примера, в котором
    // содержится трюковый код для Windows 98

    if (fFound) {
        // адреса сходятся; изменяем адрес в разделе импорта
        WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
            sizeof(pfnNew), NULL);
        return; // получилось; выходим
    }
}

// если мы попали сюда, значит, в разделе импорта
// нет ссылки на нужную функцию
}

```

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 *ReplaceIATEntryInOneMod* следующим образом.

```
PROC pfnOrig = GetProcAddress(GetModuleHandle("Kernel32"), "ExitProcess");
HMODULE hmodCaller = GetModuleHandle("DataBase.exe");

void 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.

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 *ReplaceIATEntryInOneMod*). 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 *Kernel32.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 такого адреса нет, значит, данный модуль не импортирует нужный идентификатор, и *ReplaceIATEntryInOneMod* 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 *ReplaceIATEntryInOneMod* 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 *ReplaceATEntryInOneMod* для каждого загружаемого модуля.

И, наконец, есть еще одна проблема, связанная с *GetProcAddress*. Допустим, поток выполняет такой код:

```
typedef int (WINAPI *PFNEXITPROCESS)(UINT uExitCode);
PFNEXITPROCESS pfnExitProcess = (PFNEXITPROCESS) GetProcAddress(
    GetModuleHandle("Kernel32"), "ExitProcess");
pfnExitProcess(0);
```

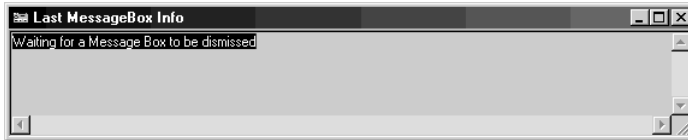
Этот код сообщает системе, что надо получить истинный адрес *ExitProcess* в *Kernel32.dll*, а затем сделать вызов по этому адресу. Данный код будет выполнен в обход Вашей подставной функции. Проблема решается перехватом обращений к *GetProcAddress*. При ее вызове Вы должны возвращать адрес своей функции.

В следующем разделе я покажу, как на практике реализовать перехват API-вызовов и решить все проблемы, связанные с использованием *LoadLibrary* и *GetProcAddress*.

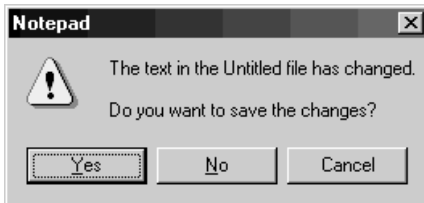
Программа-пример LastMsgBoxInfo

Эта программа, «22 LastMsgBoxInfo.exe» (см. листинг на рис. 22-5), 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-LastMsgBox-Info и 22-LastMsgBoxInfoLib на компакт-диске, прилагаемом к книге.*

После запуска LastMsgBoxInfo открывает диалоговое окно, показанное ниже.



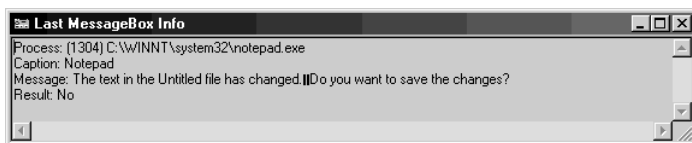
В этот момент программа находится в состоянии ожидания. Запустите какое-нибудь приложение и заставьте его открыть окно с тем или иным сообщением. Тестируя свою программу, я запускал Notepad, набирал произвольный текст, а затем пытался закрыть его окно, не сохранив набранный текст. Это заставляло Notepad выводить вот такое окно с предложением сохранить документ.



* Как сообщил сам автор, эта программа, к сожалению, работает не со всеми приложениями. Чтобы исправить эту ошибку, Вы должны модифицировать метод *CAPiHook::ReplaceIATEntryInOneMod* так, чтобы перед вызовом *WriteProcessMemory* он обращался к *VirtualProtect*:

```
DWORD dwDummy;
VirtualProtect(ppfn, sizeof(ppfn), PAGE_EXECUTE_READWRITE, &dwDummy);
```

После отказа от сохранения документа диалоговое окно `LastMsgBoxInfo` приобретает следующий вид.



Как видите, `LastMsgBoxInfo` позволяет наблюдать за вызовами функции `MessageBox` из других процессов.

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_MessageBoxA, 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 `ReplaceIATEntryInAllMods`, 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_MessageBoxA(HWND hWnd, PCSTR pszText,
    PCSTR pszCaption, UINT uType) {

    int nResult = ((PFNMESSAGEBOXA)(PROC) g_MessageBoxA)
        (hWnd, pszText, pszCaption, uType);
    SendLastMsgBoxInfo(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 просмотра файла `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.



LastMsgBoxInfo.cpp

```

/*****
Модуль: LastMsgBoxInfo.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение 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) {

    // какой-то из перехватываемых процессов прислал информацию
    // об открытом им окне с сообщением; выводим ее на экран

```

Рис. 22-5. Программа-пример LastMsgBoxInfo

см. след. стр.

Рис. 22-5. *продолжение*

```

        SetDlgItemTextA(hwnd, IDC_INFO, (PCSTR) 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;
#ifdef _DEBUG
    HWND hwnd = FindWindow(NULL, TEXT("Untitled - Paint"));
    dwThreadId = GetWindowThreadProcessId(hwnd, NULL);
#endif

    LastMsgBoxInfo_HookAllApps(TRUE, dwThreadId);
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_LASTMSGBOXINFO), NULL, Dlg_Proc);
    LastMsgBoxInfo_HookAllApps(FALSE, 0);
    return(0);
}

    ////////////////////////////////////// Конеч файла //////////////////////////////////////

```

LastMsgBoxInfoLib.cpp

```

/*****
Модуль: LastMsgBoxInfoLib.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#define WINVER      0x0500
#include " \CmnHdr.h"
#include <WindowsX.h>
#include <tchar.h>
#include <stdio.h>
#include "APIHook.h"

#define LASTMSGBOXINFOLIBAPI extern "C" __declspec(dllexport)
#include "LastMsgBoxInfoLib.h"

```

Рис. 22-5. продолжение

```

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

// прототипы перехватываемых функций
typedef int (WINAPI *PFNMESSAGEBOXA)(HWND hWnd, PCSTR pszText,
    PCSTR pszCaption, UINT uType);
typedef int (WINAPI *PFNMESSAGEBOXW)(HWND hWnd, PCWSTR pszText,
    PCWSTR pszCaption, UINT uType);

// нам приходится ссылаться на эти переменные до их создания
extern CAPIHook g_MessageBoxA;
extern CAPIHook g_MessageBoxW;

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

// эта функция отправляет сообщение MessageBox в наше диалоговое окно
void SendLastMsgBoxInfo(BOOL fUnicode,
    PVOID pvCaption, PVOID pvText, int nResult) {

    // получаем полное имя процесса, открывшего окно с сообщением
    char szProcessPathname[MAX_PATH];
    GetModuleFileNameA(NULL, szProcessPathname, MAX_PATH);

    // преобразуем возвращенное значение в строку, понятную человеку
    PCSTR pszResult = "(Unknown)";
    switch (nResult) {
        case IDOK:           pszResult = "Ok";           break;
        case IDCANCEL:       pszResult = "Cancel";       break;
        case IDABORT:        pszResult = "Abort";        break;
        case IDRETRY:        pszResult = "Retry";        break;
        case IDIGNORE:       pszResult = "Ignore";       break;
        case IDYES:          pszResult = "Yes";          break;
        case IDNO:           pszResult = "No";           break;
        case IDCLOSE:        pszResult = "Close";        break;
        case IDHELP:         pszResult = "Help";         break;
        case IDTRYAGAIN:     pszResult = "Try Again";    break;
        case IDCONTINUE:     pszResult = "Continue";     break;
    }

    // формируем строку для отсылки в наше диалоговое окно
    char sz[2048];
    wsprintfA(sz, fUnicode
        ? "Process: (%d) %s\r\nCaption: %S\r\nMessage: %S\r\nResult: %s"
        : "Process: (%d) %s\r\nCaption: %s\r\nMessage: %s\r\nResult: %s",
        GetCurrentProcessId(), szProcessPathname,
        pvCaption, pvText, pszResult);

    // отправляем ее в наше диалоговое окно
    COPYDATASTRUCT cds = { 0, lstrlenA(sz) + 1, sz };
    FORWARD_WM_COPYDATA(FindWindow(NULL, TEXT("Last MessageBox Info")),
        NULL, &cds, SendMessage);
}

```

см. след. стр.

Рис. 22-5. *продолжение*

```

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

// функция, заменяющая MessageBoxW
int WINAPI Hook_MessageBoxW(HWND hWnd, PCWSTR pszText, LPCWSTR pszCaption,
    UINT uType) {

    // вызываем исходную функцию MessageBoxW
    int nResult = ((PFNMESSAGEBOXW)(PROC) g_MessageBoxW)
        (hWnd, pszText, pszCaption, uType);

    // посылаем полученную информацию в наше диалоговое окно
    SendLastMsgBoxInfo(TRUE, (PVOID) pszCaption, (PVOID) pszText, nResult);

    // возвращаем результат вызвавшему модулю
    return(nResult);
}

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

// функция, заменяющая MessageBoxA
int WINAPI Hook_MessageBoxA(HWND hWnd, PCSTR pszText, PCSTR pszCaption,
    UINT uType) {

    // вызываем исходную функцию MessageBoxA
    int nResult = ((PFNMESSAGEBOXA)(PROC) g_MessageBoxA)
        (hWnd, pszText, pszCaption, uType);

    // посылаем полученную информацию в наше диалоговое окно
    SendLastMsgBoxInfo(FALSE, (PVOID) pszCaption, (PVOID) pszText, nResult);

    // возвращаем результат вызвавшему модулю
    return(nResult);
}

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

// ставим ловушки на функции MessageBoxA и MessageBoxW
CAPIHook g_MessageBoxA("User32.dll", "MessageBoxA",
    (PROC) Hook_MessageBoxA, TRUE);

CAPIHook g_MessageBoxW("User32.dll", "MessageBoxW",
    (PROC) Hook_MessageBoxW, TRUE);

// поскольку мы внедряем DLL, пользуясь Windows-ловушками,
// нам придется сохранить описатель ловушки в общем блоке памяти
// (кстати, в Windows 2000 этого не требуется)
#pragma data_seg("Shared")
HHOOK g_hhook = NULL;
#pragma data_seg()
#pragma comment(linker, "/Section:Shared, rws")

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

```

Рис. 22-5. *продолжение*

```

static LRESULT WINAPI GetMsgProc(int code, WPARAM wParam, LPARAM lParam) {

    // Примечание: в Windows 2000 первый параметр CallNextHookEx может
    // принимать значение NULL. В Windows 98 он обязательно должен содержать
    // описатель ловушки.
    return(CallNextHookEx(g_hhook, code, wParam, lParam));
}

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

// возвращает HMODULE, содержащий указанный адрес памяти
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 fInstall, DWORD dwThreadId) {

    BOOL fOk;

    if (fInstall) {

        chASSERT(g_hhook == NULL); // повторная установка ловушки недопустима
        // устанавливаем ловушку
        g_hhook = SetWindowsHookEx(WH_GETMESSAGE, GetMsgProc,
            ModuleFromAddress(LastMsgBoxInfo_HookAllApps), dwThreadId);
        fOk = (g_hhook != NULL);

    } else {

        chASSERT(g_hhook != NULL); // нельзя снять ловушку, если она не установлена
        fOk = UnhookWindowsHookEx(g_hhook);
        g_hhook = NULL;
    }
    return(fOk);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

LastMsgBoxInfoLib.h

```

/*****
Модуль: LastMsgBoxInfoLib.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#ifdef LASTMSGBOXINFOLIBAPI

```

см. след. стр.

Рис. 22-5. *продолжение*

```
#define LASTMSGBOXINFOLIBAPI extern "C" __declspec(dllimport)
#endif

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

LASTMSGBOXINFOLIBAPI BOOL WINAPI LastMsgBoxInfo_HookAllApps(BOOL fInstall,
    DWORD dwThreadId);

//////////////////////////////////// Конец файла //////////////////////////////////////
```

APIHook.cpp

```
/*
Модуль: APIHook.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*/

#include "..\CmnHdr.h"
#include <ImageHlp.h>
#pragma comment(lib, "ImageHlp")

#include "APIHook.h"
#include "..\04-ProcessInfo\Toolhelp.h"

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

// При выполнении приложения в Windows 98 под управлением отладчика
// последний записывает в раздел импорта модуля указатель на заглушку,
// которая вызывает желательную функцию. Чтобы учесть это, придется
// выделять сумасшедшие трюки, и для них понадобятся следующие переменные.

// старший адрес закрытой памяти (только в Windows 98)
PVOID CAPIHook::sm_pvMaxAppAddr = NULL;
const BYTE cPushOpCode = 0x68; // код PUSH-инструкции на платформе x86

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

// заголовок связанного списка объектов CAPIHook
CAPIHook* CAPIHook::sm_pHead = NULL;

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

CAPIHook::CAPIHook(PSTR pszCalleeModName, PSTR pszFuncName, PROC pfnHook,
    BOOL fExcludeAPIHookMod) {

    if (sm_pvMaxAppAddr == NULL) {
        // функции с адресами выше lpMaximumApplicationAddress
        // требуют особой обработки (только в Windows 98)
        SYSTEM_INFO si;
        GetSystemInfo(&si);
        sm_pvMaxAppAddr = si.lpMaximumApplicationAddress;
    }
}
```

Рис. 22-5. *продолжение*

```

    m_pNext = sm_pHead;    // этот узел был вверху списка
    sm_pHead = this;       // теперь вверху списка находится этот узел

    // сохраняем информацию о перехватываемой функции
    m_pszCalleeModName = pszCalleeModName;
    m_pszFuncName       = pszFuncName;
    m_pfnHook           = pfnHook;
    m_fExcludeAPIHookMod = fExcludeAPIHookMod;
    m_pfnOrig           = GetProcAddress(
        GetModuleHandleA(pszCalleeModName), m_pszFuncName);
    chASSERT(m_pfnOrig != NULL); // такой функции нет

    if (m_pfnOrig > sm_pvMaxAppAddr) {
        // адрес находится в общей DLL; его надо подправить
        PBYTE pb = (PBYTE) m_pfnOrig;
        if (pb[0] == cPushOpCode) {
            // пропускаем команду PUSH и получаем истинный адрес
            PVOID pv = * (PVOID*) &pb[1];
            m_pfnOrig = (PROC) pv;
        }
    }

    // перехватываем эту функцию во всех загруженных модулях
    ReplaceIATEntryInAllMods(m_pszCalleeModName, m_pfnOrig, m_pfnHook,
        m_fExcludeAPIHookMod);
}

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

CAPIHook::~CAPIHook() {

    // отменяем перехват этой функции во всех модулях
    ReplaceIATEntryInAllMods(m_pszCalleeModName, m_pfnHook, m_pfnOrig,
        m_fExcludeAPIHookMod);

    // удаляем этот объект из связанного списка
    CAPIHook* p = sm_pHead;
    if (p == this) { // удаляем верхний узел
        sm_pHead = p->m_pNext;
    } else {

        BOOL fFound = FALSE;

        // проходим по списку сверху и исправляем указатели
        for (; !fFound && (p->m_pNext != NULL); p = p->m_pNext) {
            if (p->m_pNext == this) {
                // переадресуем узел с нашей функции на следующий узел
                p->m_pNext = p->m_pNext->m_pNext;
                break;
            }
        }
    }
}

```

см. след. стр.

Рис. 22-5. *продолжение*

```

        chASSERT(fFound);
    }
}

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

// Примечание: эту функцию компилятор НЕ должен подставлять в конечный код
FARPROC WINAPIHook::GetProcAddressRaw(HMODULE hmod, PCSTR pszProcName) {

    return(::GetProcAddress(hmod, pszProcName));
}

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

// возвращает HMODULE, содержащий указанный адрес памяти
static HMODULE ModuleFromAddress(PVOID pv) {

    MEMORY_BASIC_INFORMATION mbi;
    return((VirtualQuery(pv, &mbi, sizeof(mbi)) != 0)
        ? (HMODULE) mbi.AllocationBase : NULL);
}

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

void WINAPIHook::ReplaceIATEntryInAllMods(PCSTR pszCalleeModName,
    PROC pfnCurrent, PROC pfnNew, BOOL fExcludeAPIHookMod) {

    HMODULE hmodThisMod = fExcludeAPIHookMod
        ? ModuleFromAddress(ReplaceIATEntryInAllMods) : NULL;

    // получаем список модулей в данном процессе
    CToolhelp th(TH32CS_SNAPMODULE, GetCurrentProcessId());

    MODULEENTRY32 me = { sizeof(me) };
    for (BOOL fOk = th.ModuleFirst(&me); fOk; fOk = th.ModuleNext(&me)) {

        // Примечание: мы не перехватываем функции в собственном модуле
        if (me.hModule != hmodThisMod) {

            // перехватываем данную функцию в этом модуле
            ReplaceIATEntryInOneMod(
                pszCalleeModName, pfnCurrent, pfnNew, me.hModule);
        }
    }
}

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

void WINAPIHook::ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
    PROC pfnCurrent, PROC pfnNew, HMODULE hmodCaller) {

    // получаем адрес раздела импорта модуля

```


Рис. 22-5. *продолжение*

```

ULONG ulSize;
PIMAGE_IMPORT_DESCRIPTOR pImportDesc = (PIMAGE_IMPORT_DESCRIPTOR)
    ImageDirectoryEntryToData(hmodCaller, TRUE,
        IMAGE_DIRECTORY_ENTRY_IMPORT, &ulSize);

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)
        break; // найден
}

if (pImportDesc->Name == 0)
    return; // этот модуль не импортирует никаких функций из данной DLL

// 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 fFound = (*ppfn == pfnCurrent);

    if (!fFound && (*ppfn > sm_pvMaxAppAddr)) {

        // Если это "не та" функция и ее адрес находится в общей DLL,
        // то, вероятно, мы работаем под отладчиком в Windows 98.
        // Тогда адрес указывает на инструкцию, в которой, может быть,
        // есть правильный адрес.

        PBYTE pbInFunc = (PBYTE) *ppfn;
        if (pbInFunc[0] == cPushOpCode) {
            // это инструкция PUSH; за ней следует истинный адрес функции
            ppfn = (PROC*) &pbInFunc[1];

            // та ли это функция, которая нас интересует?
            fFound = (*ppfn == pfnCurrent);
        }
    }

    if (fFound) {
        // адреса сходятся; изменяем адрес в разделе импорта
    }
}

```

см. след. стр.

Рис. 22-5. *продолжение*

```

        WriteProcessMemory(GetCurrentProcess(), ppfn, &pfnNew,
            sizeof(pfnNew), NULL);
        return; // получилось; выходим
    }
}
// если мы попали сюда, значит, в разделе импорта
// нет ссылки на нужную функцию
}

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

// перехватываем LoadLibrary и GetProcAddress, чтобы наши ловушки
// работали корректно и при вызове этих функций

CAPIHook CAPIHook::sm_LoadLibraryA ("Kernel32.dll", "LoadLibraryA",
    (PROC) CAPIHook::LoadLibraryA, TRUE);

CAPIHook CAPIHook::sm_LoadLibraryW ("Kernel32.dll", "LoadLibraryW",
    (PROC) CAPIHook::LoadLibraryW, TRUE);

CAPIHook CAPIHook::sm_LoadLibraryExA("Kernel32.dll", "LoadLibraryExA",
    (PROC) CAPIHook::LoadLibraryExA, TRUE);

CAPIHook CAPIHook::sm_LoadLibraryExW("Kernel32.dll", "LoadLibraryExW",
    (PROC) CAPIHook::LoadLibraryExW, TRUE);

CAPIHook CAPIHook::sm_GetProcAddress("Kernel32.dll", "GetProcAddress",
    (PROC) CAPIHook::GetProcAddress, TRUE);

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

void CAPIHook::FixupNewlyLoadedModule(HMODULE hmod, DWORD dwFlags) {

    // если загружается новый модуль, его вызовы функций тоже перехватываются
    if ((hmod != NULL) && ((dwFlags & LOAD_LIBRARY_AS_DATAFILE) == 0)) {

        for (CAPIHook* p = sm_pHead; p != NULL; p = p->m_pNext) {
            ReplaceIATEntryInOneMod(p->m_pszCalleeModName,
                p->m_pfnOrig, p->m_pfnHook, hmod);
        }
    }
}

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

HMODULE WINAPI CAPIHook::LoadLibraryA(PCSTR pszModulePath) {

    HMODULE hmod = ::LoadLibraryA(pszModulePath);
    FixupNewlyLoadedModule(hmod, 0);
    return(hmod);
}

```

Рис. 22-5. *продолжение*

```

////////////////////////////////////
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) {

    // получаем истинный адрес этой функции
    FARPROC pfn = GetProcAddressRaw(hmod, pszProcName);

    // относится ли эта функция к числу тех, которые мы хотим перехватывать?
    CAPIHook* p = sm_pHead;
    for (; (pfn != NULL) && (p != NULL); p = p->m_pNext) {

        if (pfn == p->m_pfnOrig) {
            // возвращаем адрес перехватываемой функции
            pfn = p->m_pfnHook;
            break;
        }
    }

    return(pfn);
}

//////////////////////////////////// Конеч файл //////////////////////////////////

```

см. след. стр.

Рис. 22-5. *продолжение*

APIHook.h

```

/*****
Модуль: APIHook.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#pragma once

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

class CAPIHook {
public:
    // перехватывает какую-либо функцию во всех модулях
    CAPIHook(PSTR pszCalleeModName, PSTR pszFuncName, PROC pfnHook,
        BOOL fExcludeAPIHookMod);

    // отменяет перехват функции во всех модулях
    ~CAPIHook();

    // возвращает исходный адрес перехватываемой функции
    operator PROC() { return(m_pfnOrig); }

public:
    // вызывает настоящую GetProcAddress
    static FARPROC WINAPI GetProcAddressRaw(HMODULE hmod, PCSTR pszProcName);

private:
    static PVOID sm_pvMaxAppAddr; // старший адрес закрытой памяти
    static CAPIHook* sm_pHead;    // адрес первого объекта
    CAPIHook* m_pNext;           // адрес следующего объекта

    PCSTR m_pszCalleeModName;    // модуль, содержащий функцию (ANSI)
    PCSTR m_pszFuncName;         // имя функции в вызываемой DLL (ANSI)
    PROC m_pfnOrig;              // исходный адрес функции в вызываемой DLL
    PROC m_pfnHook;              // адрес заменяющей функции
    BOOL m_fExcludeAPIHookMod;   // не реализует ли данный модуль CAPIHook?

private:
    // заменяет адрес идентификатора в разделах импорта всех модулей
    static void WINAPI ReplaceIATEntryInAllMods(PCSTR pszCalleeModName,
        PROC pfnOrig, PROC pfnHook, BOOL fExcludeAPIHookMod);

    // заменяет адрес идентификатора в разделе импорта одного модуля
    static void WINAPI ReplaceIATEntryInOneMod(PCSTR pszCalleeModName,
        PROC pfnOrig, PROC pfnHook, HMODULE hmodCaller);

private:
    // применяется, если после установки ловушки загружается новая DLL
    static void WINAPI FixupNewlyLoadedModule(HMODULE hmod, DWORD dwFlags);

```

Рис. 22-5. *продолжение*

```
// используется для отслеживания загружаемых DLL
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);

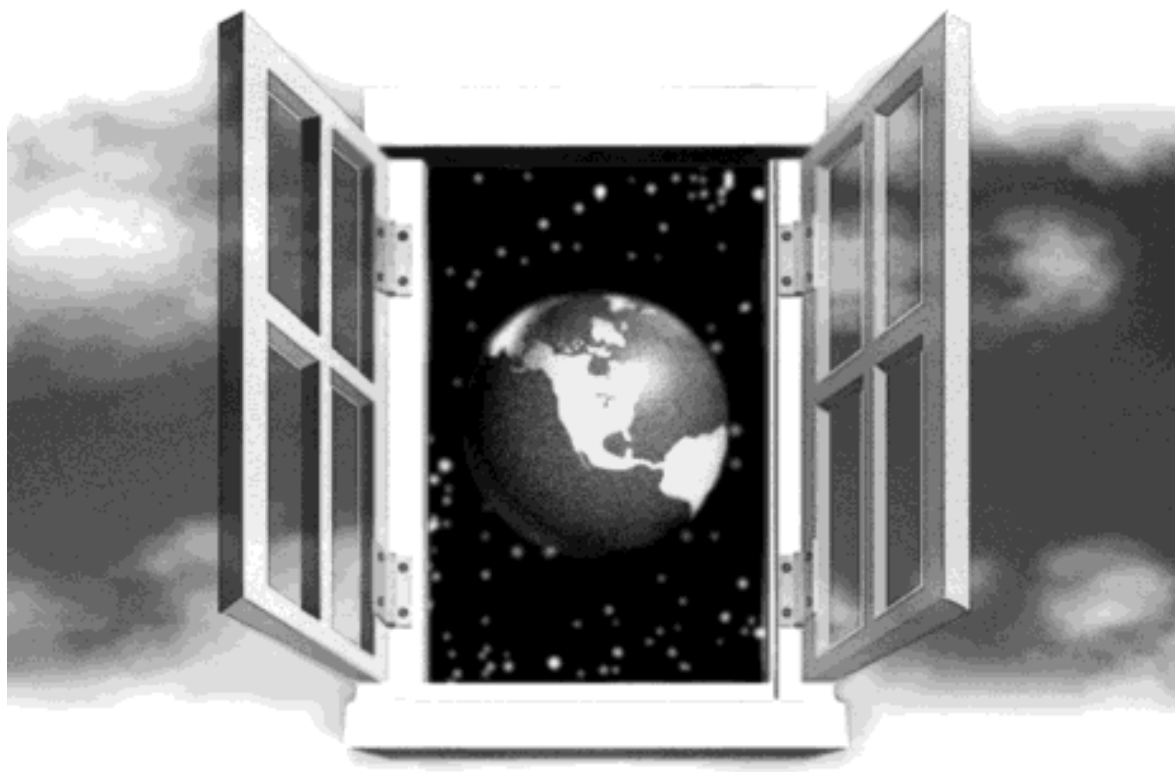
// возвращает адрес заменяющей функции при вызове перехватываемой
static FARPROC WINAPI GetProcAddress(HMODULE hmod, PCSTR pszProcName);

private:
    static CAPIHook sm_LoadLibraryA;
    static CAPIHook sm_LoadLibraryW;
    static CAPIHook sm_LoadLibraryExA;
    static CAPIHook sm_LoadLibraryExW;
    static CAPIHook sm_GetProcAddress;
};

/////////////////////////////////////// Конец файла /////////////////////////////////////////
```

Ч А С Т Ь V

СТРУКТУРНАЯ ОБРАБОТКА ИСКЛЮЧЕНИЙ



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 Microsoft 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 Microsoft. Синтаксис и ключевые слова в моих примерах могут отличаться от применяемых в других компиляторах, но основные концепции SEH везде одинаковы. В этой главе я использую синтаксис компилятора Microsoft Visual C++.



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;
    }
```

см. след. стр.


```

__finally {
    // 3. Даем и другим попользоваться защищенными данными
    ReleaseSemaphore(g_hSem, 1, NULL);
}

// 4. Продолжаем что-то делать
return(dwTemp);
}

```

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 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);
    }

    // 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*, компи-

лятор 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
```

см. след. стр.

```

        WaitForSingleObject(g_hSem, 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*, быстроедействие 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 Error. 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.

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*, приравняв 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 *FuncuDoodleDoo*. 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 *TerminateThread* или *TerminateProcess*. Некоторые функции библиотеки C (вроде *abort*), в свою очередь вызывающие *ExitProcess*, тоже исключают выполнение блока *finally*. Раз Вы не 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;
    PVOID pvBuf = NULL;
    DWORD dwNumBytesRead;
    BOOL fOk;

    hFile = CreateFile("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, PAGE_READWRITE);
    if (pvBuf == NULL) {
        CloseHandle(hFile);
        return(FALSE);
    }

    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead == 0)) {
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
        CloseHandle(hFile);
    }
}

```

см. след. стр.

```

        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 Funcarama2() {
    HANDLE hFile = INVALID_HANDLE_VALUE;
    PVOID pvBuf = NULL;
    DWORD dwNumBytesRead;
    BOOL fOk, fSuccess = FALSE;

    hFile = CreateFile("SOMEDATA.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) {
            fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
            if (fOk && (dwNumBytesRead != 0)) {
                // что-то делаем с данными
                :
                fSuccess = TRUE;
            }
        }
        VirtualFree(pvBuf, MEM_RELEASE | MEM_DECOMMIT);
    }
    CloseHandle(hFile);
    return(fSuccess);
}

```

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:

```

BOOL Funcarama3() {
    // Внимание: инициализируйте все переменные, предполагая худшее

```

```

HANDLE hFile = INVALID_HANDLE_VALUE;
PVOID pvBuf = NULL;

__try {
    DWORD dwNumBytesRead;
    BOOL fOk;

    hFile = CreateFile("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, PAGE_READWRITE);
    if (pvBuf == NULL) {
        return(FALSE);
    }

    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead != 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, Microsoft ввела еще одно ключевое слово в свой компилятор C++: *__leave*. Вот новая версия (*Funcarama4*), построенная на применении нового ключевого слова:

```

BOOL Funcarama4() {
    // Внимание: инициализируйте все переменные, предполагая худшее

```

см. след. стр.


```

HANDLE hFile = INVALID_HANDLE_VALUE;
PVOID pvBuf = NULL;

// предполагаем, что выполнение функции будет неудачным
BOOL fFunctionOk = FALSE;

__try {
    DWORD dwNumBytesRead;
    BOOL fOk;

    hFile = CreateFile("SOMEDATA.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;
    }

    fOk = ReadFile(hFile, pvBuf, 1024, &dwNumBytesRead, NULL);
    if (!fOk || (dwNumBytesRead == 0)) {
        __leave;
    }

    // что-то делаем с данными
    :
    // функция выполнена успешно
    fFunctionOk = 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(fFunctionOk);
}

```

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 выражено. 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 следующей главе.

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 функцию¹ *AbnormalTermination*:

```
BOOL AbnormalTermination();
```

Ее можно вызвать только из блока *finally*; она возвращает булево значение, которое сообщает, был ли преждевременный выход из блока *try*, связанного 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
    :
```

см. след. стр.

¹ Встраиваемая функция (intrinsic function) — особая функция, распознаваемая компилятором. Вместо генерации вызова такой функции он подставляет в точку вызова ее код. Примером встраиваемой функции является *memset* (если указан ключ компилятора /Oi). Встречая вызов *memset*, компилятор подставляет ее код непосредственно в вызывающую функцию. Обычно это ускоряет работу программы ценой увеличения ее размера. Функция *AbnormalTermination* отличается от *memset* тем, что существует только во встраиваемом варианте. Ее нет ни в одной библиотеке C.

```

__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);
}

```

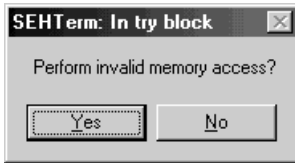
Теперь Вы знаете, как создавать обработчики завершения. Вскоре Вы увидите, что они могут быть еще полезнее и важнее, — когда мы дойдем до фильтров и обработчиков исключений (в следующей главе). А пока давайте суммируем причины, по которым следует применять обработчики завершения.

- Упрощается обработка ошибок — очистка гарантируется и проводится в одном месте.
- Улучшается восприятие текста программ.
- Облегчается сопровождение кода.
- Удастся добиться минимальных издержек по скорости и размеру кода — при условии правильного применения обработчиков.

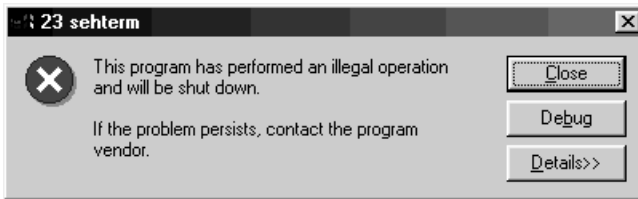
Программа-пример SEHTerm

Эта программа, «23 SEHTerm.exe» (см. листинг на рис. 23-1), демонстрирует обработчики завершения. Файлы исходного кода и ресурсов этой программы находятся в каталоге 23-SEHTerm на компакт-диске, прилагаемом к книге.

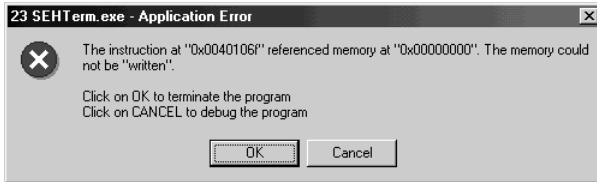
После запуска SEHTerm ее первичный поток входит в блок *try*. Из него открывается следующее окно.



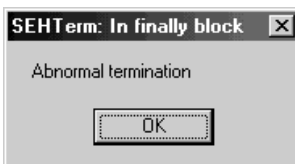
В этом окне предлагается обратиться к памяти по недопустимому адресу. (Большинство приложений не столь тактично — они обращаются по недопустимым адресам, никого не спрашивая.) Давайте обсудим, что случится, если Вы щелкнете кнопку Yes. В этом случае поток попытается записать значение 5 по нулевому адресу памяти. Запись по нулевому адресу всегда вызывает исключение, связанное с нарушением доступа. А когда поток возбуждает такое исключение, Windows 98 выводит окно, показанное ниже.



В Windows 2000 аналогичное окно выглядит иначе.

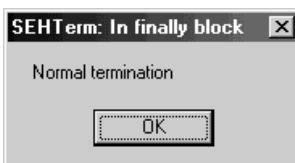


Если Вы теперь щелкнете кнопку Close (в Windows 98) или OK (в Windows 2000), процесс завершится. Однако в исходном коде этой программы присутствует блок *finally*, который будет выполнен до того, как процесс завершится. Из этого блока открывается следующее окно.



Блок *finally* выполняется потому, что происходит ненормальный выход из связанного с ним блока *try*. После закрытия этого окна процесс завершается.

О'кэй, а сейчас снова запустим эту программу. Но на этот раз попробуйте щелкнуть кнопку No, чтобы избежать обращения к памяти по недопустимому адресу. Тогда поток естественным образом перейдет из блока *try* в блок *finally*, откуда будет открыто следующее окно.



Обратите внимание, что на этот раз в окне сообщается о нормальном выходе из блока *try*. Когда Вы закроете это окно, поток выйдет из блока *finally* и покажет последнее окно.



После того как Вы закроете и это окно, процесс нормально завершится, поскольку функция *WinMain* вернет управление. Заметьте, что данное окно не появляется при аварийном завершении процесса.



SEHTerm.cpp

```

/*****
Модуль: SEHTerm.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h" /* см. приложение A */
#include <tchar.h>

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    __try {
        int n = MessageBox(NULL, TEXT("Perform invalid memory access?"),
            TEXT("SEHTerm: In try block"), MB_YESNO);

        if (n == IDYES) {
            * (PBYTE) NULL = 5; // это приводит к нарушению доступа
        }
    }
    __finally {
        PCTSTR psz = AbnormalTermination()
            ? TEXT("Abnormal termination") : TEXT("Normal termination");
        MessageBox(NULL, psz, TEXT("SEHTerm: In finally block"), MB_OK);
    }

    MessageBox(NULL, TEXT("Normal process termination"),
        TEXT("SEHTerm: After finally block"), MB_OK);

    return(0);
}

//////////////////////////////////// Конеч файл //////////////////////////////////////

```

Рис. 23-1. Программа-пример SEHTerm

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 (фильтр исключений) {
    // обработчик исключений
    :
}
```

Обратите внимание на ключевое слово *__except*. За блоком *try* всегда должен следовать либо блок *finally*, 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
    return(dwTemp);
}
```

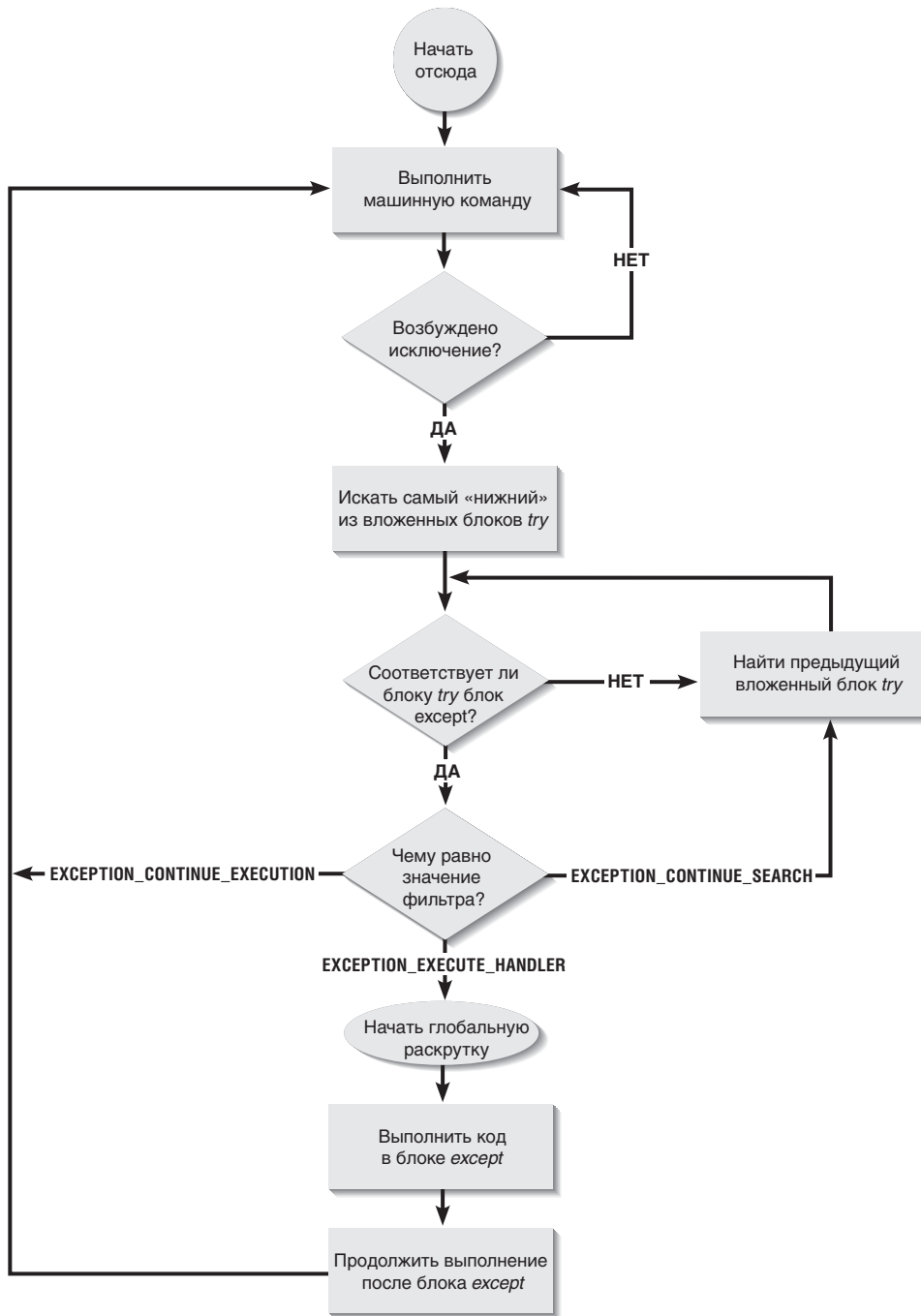


Рис. 24-1. Так система обрабатывает исключения

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*.

| 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.

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, следующие 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 выделить 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 C, 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, Microsoft 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 возбуждившей исключение команды. На этот раз Вы поделили бы 5 на 2, и программа спокойно продолжила бы свою работу. Иначе говоря, Вы что-то меняете и заставляете систему повторить выполнение команды, возбуждившей 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!

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 неожиданными. 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 *strlen* 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 *free* с передачей ей NULL. Стандарт ANSI C допускает вызов *free* с передачей NULL, в каком-то 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 RobustMemDup(PBYTE pbSrc, size_t cb) {

    PBYTE 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.

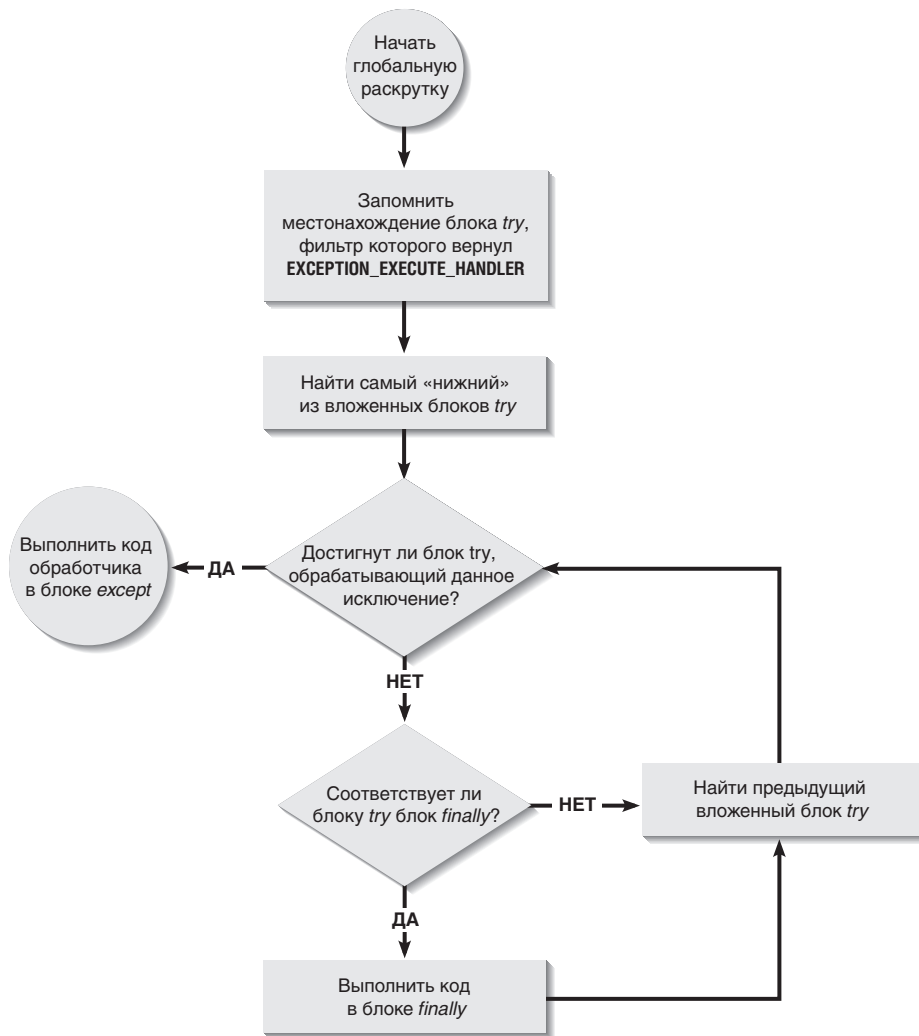


Рис. 24-2. Так система проводит глобальную раскрутку

```

void Func0Stimpy1() {
    // 1. Что-то делаем здесь
    :
    __try {
        // 2. Вызываем другую функцию
        Func0Ren1();

        // 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 ноль 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 *finally* 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, SEH — штука весьма трудная для понимания: в 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.

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.

Microsoft намеренно вложила в SEH такую логику. Иногда ведь нужно прекратить раскрутку и продолжить выполнение программы. Хотя в большинстве случаев так все же не делают. А значит, будьте внимательны и избегайте операторов *return* в блоках *finally*.

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* «защит» (простоты ради) в код самого фильтра, но Вы могли бы вызывать там функцию, которая определяла бы нужный идентификатор. Взгляните:

```
char g_szBuffer[100];

void FunclnRoosevelt1() {
    int x = 0;
    char *pchBuffer = NULL;

    __try {
        *pchBuffer = 'J';
        x = 5 / x;
    }
    __except (OilFilter1(&pchBuffer)) {
        MessageBox(NULL, "An exception occurred", NULL, MB_OK);
    }
    MessageBox(NULL, "Function completed", NULL, MB_OK);
}

LONG OilFilter1(char **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 **ppchBuffer* 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 *J* 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 ис-

ключений. На этот раз **pcbBuffer* 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. Но, разумеется, в итоге фильтр должен вернуть один из трех идентификаторов.

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:

```
*pcbBuffer = 'J';
```

которые выглядят так:

```
MOV EAX, [pcbBuffer] // адрес помещается в регистр EAX
MOV [EAX], 'J'       // символ J записывается по адресу из регистра EAX
```

Последняя команда и возбудила бы исключение. Фильтр исключений, перехватив его, исправил бы значение *pcbBuffer* и указал бы системе повторить эту команду. Но проблема в том, что содержимое регистра не изменится так, чтобы отразить новое значение *pcbBuffer*, 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:

```
char g_szBuffer[100];

void FunclinRoosevelt2() {
    char *pchBuffer = NULL;

    __try {
        FuncAtude2(pchBuffer);
    }
    __except (OilFilter2(&pchBuffer)) {
        MessageBox(...);
    }
}

void FuncAtude2(char *sz) {
    *sz = 0;
}

LONG OilFilter2(char **ppchBuffer) {
    if (*ppchBuffer == NULL) {
        *ppchBuffer = g_szBuffer;
        return(EXCEPTION_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*:

```
char g_szBuffer[100];

void FunclinRoosevelt3() {
    char *pchBuffer = NULL;

    __try {
        FuncAtude3(pchBuffer);
    }
    __except (OilFilter3(&pchBuffer)) {
        MessageBox(...);
    }
}

void FuncAtude3(char *sz) {
```

см. след. стр.

```

__try {
    *sz = 0;
}
__except (EXCEPTION_CONTINUE_SEARCH) {
    // ЭТОТ КОД НИКОГДА НЕ ВЫПОЛНЯЕТСЯ
    :
}
}

LONG OilFilter3(char **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* (в функции *FuncInRoosevelt3*) и вычисляет его фильтр *OilFilter3*. 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!

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 *FuncInRoosevelt3*.

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 ноль, 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;
    y = 4 / x;
}

```

```
__except ((GetExceptionCode() == EXCEPTION_INT_DIVIDE_BY_ZERO) ?
    EXCEPTION_EXECUTE_HANDLER : EXCEPTION_CONTINUE_SEARCH) {
    // обработка деления на ноль
}
```

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 WinBase.h. Я сгруппировал исключения по категориям.

Исключения, 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_INT_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 фильгра 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 ноль
        :
        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 24-1.

| Биты | 31–30 | 29 | 28 | 27–16 | 15–0 |
|-------------|---|---|----------------------------------|--------------------------------|--|
| Содержимое: | Код степени «тяжести» (severity) | Кем определен — Microsoft или пользователем | Зарезервирован | Код подсистемы (facility code) | Код исключения |
| Значение: | 0 = успех 1 = информация 2 = предупреждение 3 = ошибка | 0 = Microsoft 1 = пользователь | Должен быть 0 (см. таблицу ниже) | Определяется Microsoft | Определяется Microsoft или пользователем |

Таблица 24-1. Поля кода ошибки

На сегодняшний день определены такие коды подсистемы.

| Код подсистемы | Значение | Код подсистемы | Значение |
|-------------------|----------|----------------------|----------|
| FACILITY_NULL | 0 | FACILITY_CONTROL | 10 |
| FACILITY_RPC | 1 | FACILITY_CERT | 11 |
| FACILITY_DISPATCH | 2 | FACILITY_INTERNET | 12 |
| FACILITY_STORAGE | 3 | FACILITY_MEDIASERVER | 13 |
| FACILITY_ITF | 4 | FACILITY_MSMQ | 14 |
| FACILITY_WIN32 | 7 | FACILITY_SETUPAPI | 15 |
| FACILITY_WINDOWS | 8 | FACILITY_SCARD | 16 |
| FACILITY_SECURITY | 9 | FACILITY_COMPLUS | 17 |

Разберем на части, например, код исключения EXCEPTION_ACCESS_VIOLATION. Если Вы посмотрите его значение в файле WinBase.h, то увидите, что оно равно 0xC0000005:

 C 0 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 Microsoft. 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 Microsoft 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_POINTERS всего два элемента — указатели на помещенные в стек структуры EXCEPTION_RECORD и CONTEXT:

```
typedef struct _EXCEPTION_POINTERS {  
    PEXCEPTION_RECORD ExceptionRecord;
```

```

    PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;

```

Чтобы получить эту информацию и использовать ее в программе, вызовите *GetExceptionInformation*:

```
PEXCEPTION_POINTERS GetExceptionInformation();
```

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 *Saved*

Context. 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*.
- *ExceptionFlags* — флаги исключения. На данный момент определено только два значения: 0 (возобновляемое исключение) и `EXCEPTION_NONCONTINUABLE` (невозобновляемое исключение). Любая попытка возобновить 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.

Эта структура позволяет писать фильтры исключений, сообщающие значительный объем информации о работе программы. Можно создать, например, такой фильтр:

```
--try {
:
}
__except (ExpFiltr(GetExceptionInformation()->ExceptionRecord)) {
:
}

LONG ExpFiltr(PEXCEPTION_RECORD pER) {
    char szBuf[300], *p;
    DWORD dwExceptionCode = pER->ExceptionCode;

    sprintf(szBuf, "Code = %x, Address = %p",
        dwExceptionCode, pER->ExceptionAddress);

    // находим конец строки
    p = strchr(szBuf, 0);

    // я использовал оператор switch на тот случай, если Microsoft
    // в будущем добавит информацию для других исключений
    switch (dwExceptionCode) {
        case EXCEPTION_ACCESS_VIOLATION:
            sprintf(p, "Attempt to %s data at address %p",
                pER->ExceptionInformation[0] ? "write" : "read",
                pER->ExceptionInformation[1]);
            break;

        default:
            break;
    }
    MessageBox(NULL, szBuf, "Exception", MB_OK | MB_ICONEXCLAMATION);
    return(EXCEPTION_CONTINUE_SEARCH);
}
```

Элемент *ContextRecord* структуры `EXCEPTION_POINTERS` 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 апплодирую 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,
    DWORD 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 Microsoft, вроде STATUS_NO_MEMORY для *HeapAlloc*);
- бит 28 должен быть равен 0;
- биты 27–16 должны указывать один из кодов подсистемы, предопределенных Microsoft;
- 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 Microsoft 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 *Exception-*

Record. Он указывает на другую структуру EXCEPTION_RECORD, где содержится информация о предыдущем исключении.

Обычно система одновременно 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.

Необработанные исключения 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).

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.

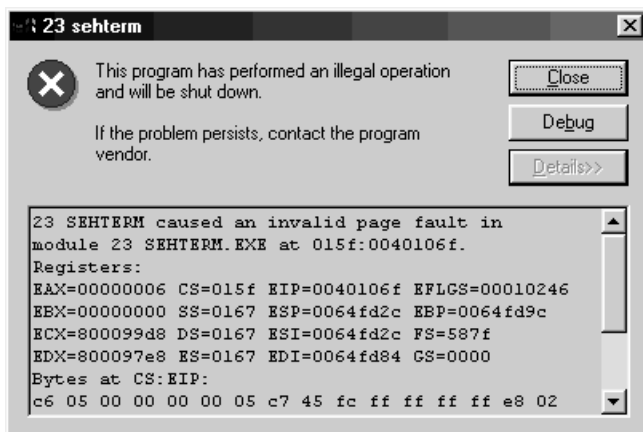
```
VOID BaseProcessStart(PPROCESS_START_ROUTINE pfnStartAddr) {
    __try {
        ExitThread((pfnStartAddr)());
    }
    __except (UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // Примечание: сюда мы никогда не попадем
}

VOID BaseThreadStart(PTHREAD_START_ROUTINE pfnStartAddr, PVOID pvParam) {
    __try {
        ExitThread((pfnStartAddr)(pvParam));
    }
    __except (UnhandledExceptionFilter(GetExceptionInformation())) {
        ExitProcess(GetExceptionCode());
    }
    // Примечание: сюда мы никогда не попадем
}
```

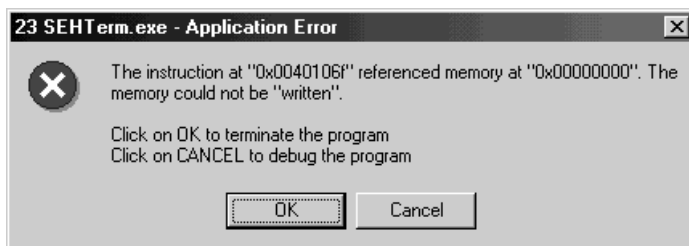
Обратите внимание, что обе функции содержат SEH-фрейм: поток запускается из блока *try*. Если поток возбудит исключение, в ответ на которое все Ваши фильтры вернут `EXCEPTION_CONTINUE_SEARCH`, будет вызвана особая функция фильтра, предоставляемая операционной системой:

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

Она выводит окно, указывающее на то, что поток в процессе вызвал необрабатываемое им исключение, и предлагает либо закрыть процесс, либо начать его отладку. В Windows 98 это окно выглядит следующим образом.



А в Windows 2000 оно имеет другой вид.



В Windows 2000 первая часть текста в этом окне подсказывает тип исключения и адрес вызвавшей его инструкции в адресном пространстве процесса. У меня окно появилось из-за нарушения доступа к памяти, поэтому система сообщила адрес, по которому произошла ошибка, и тип доступа к памяти — чтение. *UnhandledExceptionFilter* получает эту информацию из элемента *ExceptionInformation* структуры *EXCEPTION_RECORD*, инициализированной для этого исключения.

В данном окне можно сделать одно из двух. Во-первых, щелкнуть кнопку OK, и тогда *UnhandledExceptionFilter* вернет *EXCEPTION_EXECUTE_HANDLER*. Это приведет к глобальной раскрутке и соответственно к выполнению всех имеющихся блоков *finally*, а затем и к выполнению обработчика в *BaseProcessStart* или *BaseThreadStart*. Оба обработчика вызывают *ExitProcess*, поэтому-то Ваш процесс и закрывается. Причем кодом завершения процесса становится код исключения. Кроме того, процесс закрывается его же потоком, а не операционной системой! А это означает, что Вы можете вмешаться в ход завершения своего процесса.

Во-вторых, Вы можете щелкнуть кнопку Cancel (сбываются самые смелые мечты программистов). В этом случае *UnhandledExceptionFilter* попытается запустить отладчик и подключить его к процессу. Тогда Вы сможете просматривать состояние глобальных, локальных и статических переменных, расставлять точки прерывания, перезапускать процесс и вообще делать все, что делается при отладке процесса.

Но самое главное, что сбой в программе можно исследовать в момент его возникновения. В большинстве других операционных систем для отладки процесса сначала запускается отладчик. При генерации исключения в процессе, выполняемом в любой из таких систем, этот процесс надо завершить, запустить отладчик и прогнать программу уже под отладчиком. Проблема, правда, в том, что ошибку надо сначала воспроизвести; лишь потом можно попытаться ее исправить. А кто знает, какие значения были у переменных, когда Вы впервые заметили ошибку? Поэтому найти ее та-

ким способом гораздо труднее. Возможность динамически подключать отладчик к уже запущенному процессу — одно из лучших качеств Windows.

WINDOWS 2000

В этой книге рассматривается разработка приложений, работающих только в пользовательском режиме. Но, наверное, Вас интересует, что происходит, когда необработанное исключение возникает в потоке, выполняемом в режиме ядра. Так вот, исключения в режиме ядра обрабатываются так же, как и исключения пользовательского режима. Если низкоуровневая функция для работы с виртуальной памятью возбуждает исключение, система проверяет, есть ли фильтр режима ядра, готовый обработать это исключение. Если такого фильтра нет, оно остается необработанным. В этом случае необработанное исключение окажется в операционной системе или (что вероятнее) в драйвере устройства, а не в приложении. А это уже серьезно!

Так как дальнейшая работа системы после необработанного исключения в режиме ядра небезопасна, Windows не вызывает *UnhandledExceptionFilter*. Вместо этого появляется так называемый «синий экран смерти»: экран переключается в текстовый режим, окрашивается в синий фон, выводится информация о модуле, вызвавшем необработанное исключение, и система останавливается. Вам следует записать эту информацию и отправить ее в Microsoft или поставщику драйвера устройства. Прежде чем продолжить работу, придется перезагрузить машину; при этом все несохраненные данные теряются.

Отладка по запросу

Windows позволяет подключать отладчик к любому процессу в любой момент времени — эта функциональность называется отладкой по запросу (just-in-time debugging). В этом разделе я расскажу, как она работает. Щелкнув кнопку Cancel, Вы сообщаете функции *UnhandledExceptionFilter* о том, что хотите начать отладку процесса.

Для активизации отладчика *UnhandledExceptionFilter* просматривает раздел реестра:

```
HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug
```

Если Вы установили Visual Studio, то содержащийся в этом разделе параметр Debugger имеет следующее значение:

```
"C:\Program Files\Microsoft Visual Studio\Common\MSDev98\Bin\msdev.exe"  
-p %ld -e %ld
```

WINDOWS 98

В Windows 98 соответствующие значения хранятся не в реестре, а в файле Win.ini.

Строка, приведенная выше, сообщает системе, какой отладчик надо запустить (в данном случае — MSDev.exe). Естественно, Вы можете изменить это значение, указав другой отладчик. *UnhandledExceptionFilter* передает отладчику два параметра в командной строке. Первый — это идентификатор процесса, который нужно отладить, а второй — наследуемое событие со сбросом вручную, которое создается функцией *UnhandledExceptionFilter* в занятом состоянии. Отладчик должен распознавать ключи *-p* и *-e* как идентификатор процесса и описатель события.

Сформировав командную строку из идентификатора процесса и описателя события, *UnhandledExceptionFilter* запускает отладчик вызовом *CreateProcess*. Отладчик про-

веряет аргументы в командной строке и, обнаружив ключ *-p*, подключается к соответствующему процессу вызовом *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 поток *UnhandledExceptionFilter* должен находиться в режиме ожидания. Для этого функция вызывает *WaitForSingleObject* и передает описатель созданного ею события со сбросом вручную. Как Вы помните, оно было создано в занятом состоянии, поэтому поток отлаживаемого процесса немедленно приостанавливается и ждет освобождения этого события.

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, поскольку процесс отладчика является дочерним по отношению к отлаживаемому процессу, который и породил его, вызвав *UnhandledExceptionFilter*.

Переход события в свободное состояние пробуждает поток отлаживаемого процесса, и он передает отладчику информацию о необработанном исключении. Получив эти данные, отладчик загружает соответствующий файл исходного кода и переходит к команде, которая вызвала исключение. Вот это действительно круто!

Кстати, совсем не обязательно дожидаться исключения, чтобы начать отладку. Отладчик можно подключить в любой момент командой «MSDEV -p *PID*», где *PID* — идентификатор отлаживаемого процесса. Task Manager в Windows 2000 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.

Отключение вывода сообщений об исключении

Иногда нужно, чтобы окно с сообщением об исключении не появлялось на экране, — например, в готовом программном продукте. Ведь если такое окно появится, пользователь может случайно перейти в режим отладки Вашей программы. Стоит ему только щелкнуть кнопку Cancel, и он шагнет на незнакомую и страшную территорию — попадет в отладчик. Поэтому предусмотрено несколько способов, позволяющих избежать появления этого окна на экране.

Принудительное завершение процесса

Запретить функции *UnhandledExceptionFilter* вывод окна с сообщением об исключении можно вызовом *SetErrorMode* с передачей идентификатора SEM_NOGPFAULT-ERRORBOX:

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

Тогда *UnhandledExceptionFilter*, вызванная для обработки исключения, немедленно вернет EXCEPTION_EXECUTE_HANDLER, что приведет к глобальной раскрутке и выполнению обработчика в *BaseProcessStart* или *BaseThreadStart*, который закроет процесс.

Лично мне этот способ не нравится, так как пользователь не получает никакого предупреждения — приложение просто исчезает.

Создание оболочки вокруг функции потока

Другой способ состоит в том, что Вы помещаете входную функцию первичного потока (*main*, *wmain*, *WinMain* или *wWinMain*) в блок *try-except*. Фильтр исключений должен всегда возвращать `EXCEPTION_EXECUTE_HANDLER`, чтобы исключение действительно обрабатывалось; это предотвратит вызов *UnhandledExceptionFilter*.

В обработчике исключений Вы выводите на экран диалоговое окно с какой-нибудь диагностической информацией. Пользователь может скопировать эту информацию и передать ее в службу технической поддержки Вашего приложения, что поможет выявить источник проблем. Это диалоговое окно надо разработать так, чтобы пользователь мог завершить приложение, но не отлаживать.

Этому способу присущ один недостаток: он позволяет перехватывать только те исключения, которые возникают в первичном потоке. Если исключение происходит в любом другом потоке процесса, система вызывает функцию *UnhandledExceptionFilter*. Чтобы вывернуться из этой ситуации, придется также включить блоки *try-except* во входные функции всех вторичных потоков Вашего процесса.

Создание оболочки вокруг всех функций потоков

Функция *SetUnhandledExceptionFilter* позволяет включать все функции потоков в SEH-фрейм:

```
PTOP_LEVEL_EXCEPTION_FILTER SetUnhandledExceptionFilter(
    PTOP_LEVEL_EXCEPTION_FILTER pTopLevelExceptionFilter);
```

После ее вызова необработанное исключение, возникшее в любом из потоков процесса, приведет к вызову Вашего фильтра исключений. Адрес фильтра следует передать в единственном параметре функции *SetUnhandledExceptionFilter*. Прототип этой функции-фильтра должен выглядеть так:

```
LONG UnhandledExceptionFilter(PEXCEPTION_POINTERS pExceptionInfo);
```

По форме она идентична функции *UnhandledExceptionFilter*. Внутри фильтра можно проводить любую обработку, а возвращаемым значением должен быть один из трех идентификаторов типа `EXCEPTION_*`. В следующей таблице описано, что происходит в случае возврата каждого из идентификаторов.

| Идентификатор | Действие |
|---|--|
| <code>EXCEPTION_EXECUTE_HANDLER</code> | Процесс просто завершается, так как система не выполняет никаких операций в своем обработчике исключений |
| <code>EXCEPTION_CONTINUE_EXECUTION</code> | Выполнение продолжается с инструкции, вызвавшей исключение; Вы можете модифицировать информацию об исключении, на которую указывает параметр типа <code>PEXCEPTION_POINTERS</code> |
| <code>EXCEPTION_CONTINUE_SEARCH</code> | Выполняется обычная Windows-функция <i>UnhandledExceptionFilter</i> |

Чтобы функция *UnhandledExceptionFilter* вновь стала фильтром по умолчанию, вызовите *SetUnhandledExceptionFilter* со значением `NULL`. Заметьте также, что всякий раз, когда устанавливается новый фильтр для необработанных исключений, *SetUn-*

handledExceptionFilter возвращает адрес ранее установленного фильтра. Если таким фильтром была *UnhandledExceptionFilter*, возвращается NULL. Если Ваш фильтр возвращает EXCEPTION_CONTINUE_SEARCH, Вы должны вызывать ранее установленный фильтр, адрес которого вернула *SetUnhandledExceptionFilter*.

Автоматический вызов отладчика

Это последний способ отключения окна с сообщением об исключении. В уже упомянутом разделе реестра есть еще один параметр — Auto; его значение может быть либо 0, либо 1. В последнем случае *UnhandledExceptionFilter* не выводит окно, но сразу же вызывает отладчик. А при нулевом значении функция выводит сообщения и работает так, как я уже рассказывал.

Явный вызов функции *UnhandledExceptionFilter*

Функция *UnhandledExceptionFilter* полностью задокументирована, и Вы можете сами вызывать ее в своих программах. Вот пример ее использования:

```
void Funcadelic() {
    __try {
        :
    }
    __except (ExpFltr(GetExceptionInformation())) {
        :
    }
}

LONG ExpFltr(PEXCEPTION_POINTERS pEP) {
    DWORD dwExceptionCode = pEP->ExceptionRecord.ExceptionCode;

    if (dwExceptionCode == EXCEPTION_ACCESS_VIOLATION) {
        // что-то делаем здесь...
        return(EXCEPTION_CONTINUE_EXECUTION);
    }
    return(UnhandledExceptionFilter(pEP));
}
```

Исключение в блоке *try* функции *Funcadelic* приводит к вызову *ExpFltr*. Ей передается значение, возвращаемое *GetExceptionInformation*. Внутри фильтра определяется код исключения и сравнивается с EXCEPTION_ACCESS_VIOLATION. Если было нарушение доступа, фильтр исправляет ситуацию и возвращает EXCEPTION_CONTINUE_EXECUTION. Это значение заставляет систему возобновить выполнение программы с инструкции, вызвавшей исключение.

Если произошло какое-то другое исключение, *ExpFltr* вызывает *UnhandledExceptionFilter*, передавая ей адрес структуры EXCEPTION_POINTERS. Функция *UnhandledExceptionFilter* открывает окно, позволяющее завершить процесс или начать отладку. Ее возвращаемое значение становится в результате функции *ExpFltr*.

Функция *UnhandledExceptionFilter* изнутри

Начав работать с исключениями, я решил, что можно извлечь массу информации, если детально вникнуть в механизм работы функции *UnhandledExceptionFilter*. Поэтому я тщательно его исследовал. Вот что делает функция *UnhandledExceptionFilter*.

1. Если возникло нарушение доступа и его причина связана с попыткой записи, система проверяет, не пытались ли Вы модифицировать ресурс в EXE- или DLL-модуле. По умолчанию такие ресурсы предназначены только для чтения. Однако 16-разрядная Windows разрешала модифицировать эти ресурсы, и из соображений обратной совместимости такие операции должны поддерживаться как в 32-, так и в 64-разрядной Windows. Поэтому, когда Вы пытаетесь модифицировать ресурс, *UnhandledExceptionFilter* вызывает *VirtualProtect* для изменения атрибута защиты страницы с этим ресурсом на PAGE_READWRITE и возвращает EXCEPTION_CONTINUE_EXECUTION.
2. Если Вы установили свой фильтр вызовом *SetUnhandledExceptionFilter*, функция *UnhandledExceptionFilter* обращается к Вашей функции фильтра. И если она возвращает EXCEPTION_EXECUTE_HANDLER или EXCEPTION_CONTINUE_EXECUTION, *UnhandledExceptionFilter* передает его системе. Но, если Вы не устанавливали свой фильтр необработанных исключений или если функция фильтра возвращает EXCEPTION_CONTINUE_SEARCH, *UnhandledExceptionFilter* переходит к операциям, описанным в п. 3.

WINDOWS 98

Из-за ошибки в Windows 98 Ваша функция фильтра необработанных исключений вызывается, только если к процессу не подключен отладчик. По той же причине в Windows 98 невозможна отладка программы Spreadsheet, представленной в следующем разделе.

3. Если Ваш процесс выполняется под управлением отладчика, то возвращается EXCEPTION_CONTINUE_SEARCH. Это может показаться странным, так как система уже выполняет самый «верхний» блок *try* или *except* и другого фильтра выше по дереву вызовов просто нет. Но, обнаружив этот факт, система сообщит отладчику о необработанном исключении в подопечном ему процессе. В ответ на это отладчик выведет окно, где предложит начать отладку. (Кстати, функция *IsDebuggerPresent* позволяет узнать, работает ли данный процесс под управлением отладчика.)
4. Если поток в Вашем процессе вызовет *SetErrorMode* с флагом SEM_NOGPFAULTERRORBOX, то *UnhandledExceptionFilter* вернет EXCEPTION_EXECUTE_HANDLER.
5. Если процесс включен в задание (см. главу 5), на которое наложено ограничение JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION, то *UnhandledExceptionFilter* также вернет EXCEPTION_EXECUTE_HANDLER.

WINDOWS 98

Windows 98 не поддерживает задания, и в ней этот этап пропускается.

6. *UnhandledExceptionFilter* считывает в реестре значение параметра Auto. Если оно равно 1, происходит переход на этап 7, в ином случае выводится окно с информацией об исключении. Если в реестре присутствует и параметр Debugger, в этом окне появляются кнопки OK и Cancel. А если этого параметра нет — только кнопка OK. Как только пользователь щелкнет кнопку OK, функция *UnhandledExceptionFilter* вернет EXCEPTION_EXECUTE_HANDLER. Щелчок кнопки Cancel (если она есть) вызывает переход на следующий этап.

WINDOWS 98

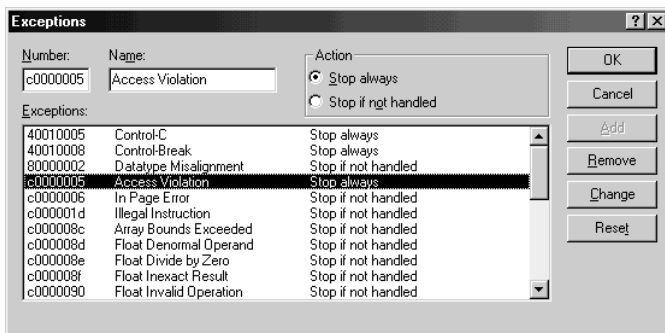
В Windows 98 упомянутые параметры хранятся не в реестре, а в файле Win.ini.

7. На этом этапе *UnhandledExceptionFilter* запускает отладчик как дочерний процесс. Но сначала создает событие со сбросом вручную в занятом состоянии и наследуемым описателем. Затем извлекает из реестра значение параметра Debugger и вызывает *sprintf* для вставки идентификатора процесса (полученного через функцию *GetCurrentProcessId*) и описателя события в командную строку. Элементу *lpDesktop* структуры STARTUPINFO присваивается значение «Winsta0\Default», чтобы отладчик был доступен в интерактивном режиме на рабочем столе. Далее вызывается *CreateProcess* со значением TRUE в параметре *blInheritHandles*, благодаря чему отладчик получает возможность наследовать описатель объекта «событие». После этого *UnhandledExceptionFilter* ждет завершения инициализации отладчика, вызвав *WaitForSingleObjectEx* с передачей ей описателя события. Заметьте, что вместо *WaitForSingleObject* используется *WaitForSingleObjectEx*. Это заставляет поток ждать в «тревожном» состоянии, которое позволяет ему обрабатывать все поступающие APC-вызовы.
8. Закончив инициализацию, отладчик освобождает событие, и поток *UnhandledExceptionFilter* пробуждается. Теперь, когда процесс находится под управлением отладчика, *UnhandledExceptionFilter* возвращает EXCEPTION_CONTINUE_SEARCH. Обратите внимание: все, что здесь происходит, точно соответствует этапу 3.

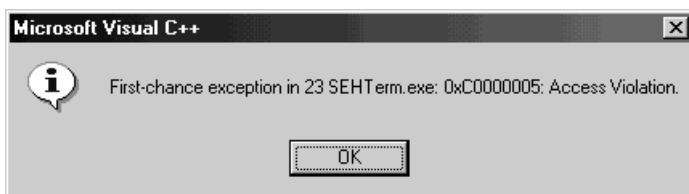
Исключения и отладчик

Отладчик Microsoft Visual C++ предоставляет фантастические возможности для отладки после исключений. Когда поток процесса вызывает исключение, операционная система немедленно уведомляет об этом отладчик (если он, конечно, подключен). Это уведомление называется «первым предупреждением» (first-chance notification). Реагируя на него, отладчик обычно заставляет поток искать фильтры исключений. Если все фильтры возвращают EXCEPTION_CONTINUE_SEARCH, операционная система вновь уведомляет отладчик, но на этот раз дает «последнее предупреждение» (last-chance notification). Существование этих двух типов предупреждений обеспечивает больший контроль за отладкой при исключениях.

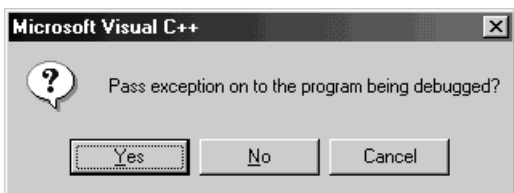
Чтобы сообщить отладчику, как реагировать на первое предупреждение, используйте диалоговое окно Exceptions отладчика.



Как видите, оно содержит список всех исключений, определенных в системе. Для каждого из них сообщаются 32-битный код, текстовое описание и ответные действия отладчика. Я выбрал исключение Access Violation (нарушение доступа) и указал для него Stop Always. Теперь, если поток в отлаживаемом процессе вызовет это исключение, отладчик выведет при первом предупреждении следующее окно.

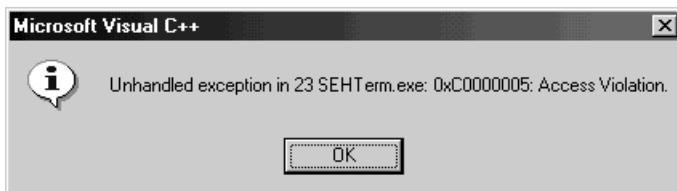


К этому моменту поток еще *не* получал шанса на поиск фильтров исключений. Сейчас я могу поместить в исходный код точки прерывания, просмотреть значения переменных или проверить стек вызовов потока. Пока ни один фильтр не выполнялся — исключение произошло только что. Когда я попытаюсь начать пошаговую отладку программы, на экране появится новое окно.



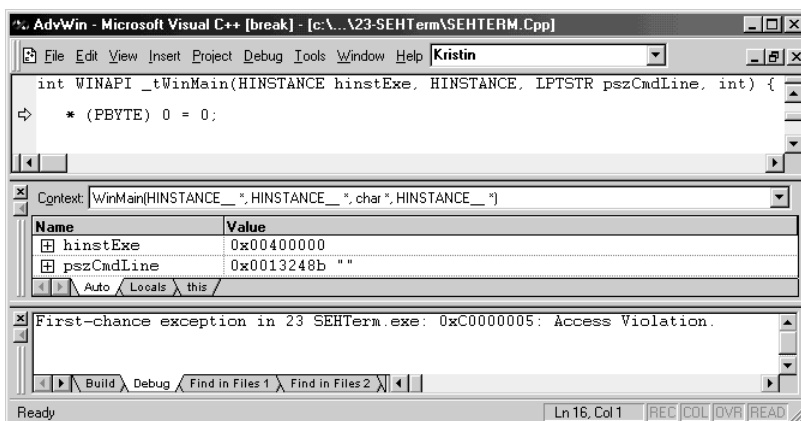
Кнопка Cancel вернет нас в отладчик. Кнопка No заставит поток отлаживаемого процесса повторить выполнение неудавшейся машинной команды. При большинстве исключений повторное выполнение команды ничего не даст, так как вновь вызовет исключение. Однако, если исключение было сгенерировано с помощью функции *RaiseException*, это позволит возобновить выполнение потока, и он продолжит работу, как ни в чем ни бывало. Данный метод может быть особенно полезен при отладке программ на C++: получится так, будто оператор *throw* никогда не выполнялся. (К обработке исключений в C++ мы вернемся в конце главы.)

И, наконец, кнопка Yes разрешит потоку отлаживаемого процесса начать поиск фильтров исключений. Если фильтр исключения, возвращающий `EXCEPTION_EXECUTE_HANDLER` или `EXCEPTION_CONTINUE_EXECUTION`, найден, то все хорошо и поток продолжает работу. Если же все фильтры вернут `EXCEPTION_CONTINUE_SEARCH`, отладчик получит последнее предупреждение и выведет окно с сообщением, аналогичным тому, которое показано ниже.

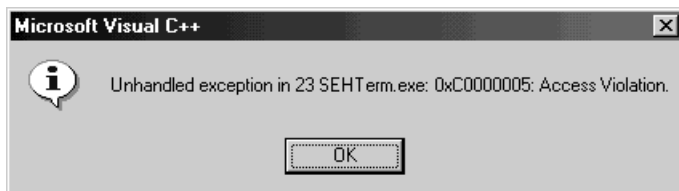


Здесь Вам придется либо начать отладку, либо закрыть приложение.

Я продемонстрировал Вам, что случится, если ответным действием отладчика выбран вариант Stop Always. Но для большинства исключений по умолчанию предлагается вариант Stop If Not Handled. В этом случае отладчик, получив первое предупреждение, просто сообщает о нем в своем окне Output.

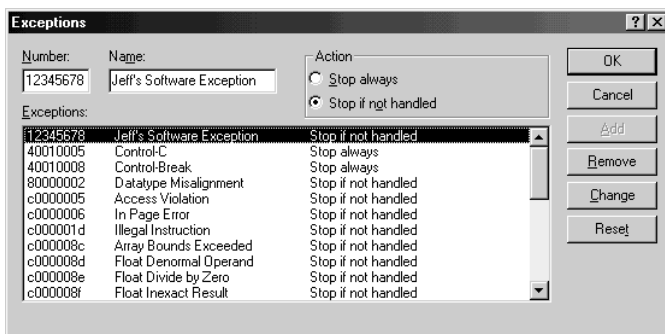


После этого отладчик разрешит потоку искать подходящие фильтры и, только если исключение не будет обработано, откроет следующее окно.



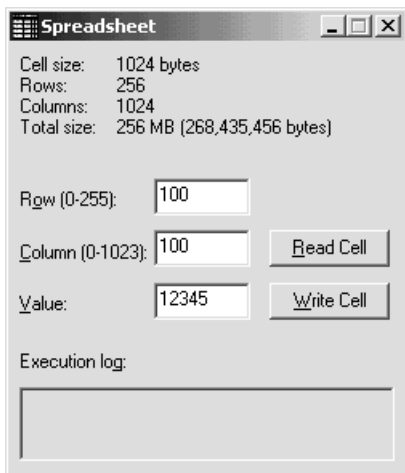
Очень важно помнить, что первое предупреждение вовсе не говорит о каких-либо проблемах или «жучках» в приложении. В сущности, оно появляется только при отладке. Отладчик просто сообщает о возникновении исключения, и, если после этого он не выводит уже известное Вам окно, это означает лишь одно: фильтр обработал исключение, приложение продолжает нормально работать. А вот последнее предупреждение говорит о том, что в Вашей программе есть некая проблема, которую надо устранить.

Прежде чем закончить обсуждение этой темы, хотелось бы упомянуть еще об одной особенности диалогового окна Exceptions отладчика. Оно полностью поддерживает любые определяемые Вами программные исключения. От Вас требуется лишь указать уникальный числовой код исключения, его название и ответное действие отладчика, а затем, щелкнув кнопку Add, добавить это новое исключение в список. Посмотрите, как это сделал я, определив собственное исключение.



Программа-пример Spreadsheet

Эта программа, «25 Spreadsheet.exe» (см. листинг на рис. 25-1), 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.



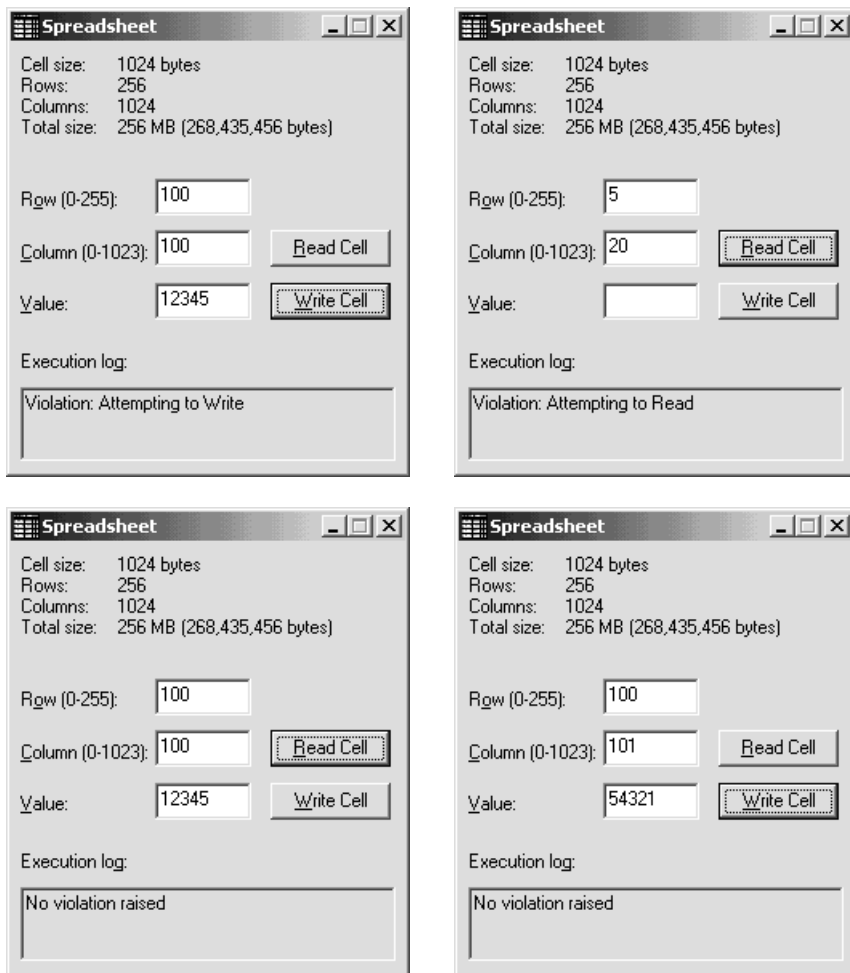
Программа Spreadsheet резервирует регион для двумерной таблицы, содержащей 256 строк и 1024 колонки, с размером ячеек по 1024 байта. Если бы программа ранее передавала физическую память под всю таблицу, то ей понадобилось бы 268 435 456 байтов, или 256 Мб. Поэтому для экономии драгоценных ресурсов программа 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_CONTINUE_EXECUTION`. Такое применение класса `CVMArray` позволяет работать с разреженной памятью (*sparse storage*), не забивая SEH-фреймами исходный код программы. Единственный недоста-

ток 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

```

/*****
Модуль: Spreadsheet.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include "Resource.h"
#include "VMArray.h"

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

HWND g_hwnd; // глобальный описатель окна, применяемого для SEH-отчетов

const int g_NumRows = 256;
const int g_NumCols = 1024;

// определяем структуру ячеек таблицы
typedef struct {
    DWORD dwValue;
    BYTE bDummy[1020];
} CELL, *PCELL;

// объявляем тип данных для всей таблицы
typedef CELL SPREADSHEET[g_NumRows][g_NumCols];
typedef SPREADSHEET *PSPREADSHEET;

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

// таблица является двумерным массивом переменных типа CELL
class CVMSpreadsheet : public CVMArray<CELL> {
public:
    CVMSpreadsheet() : CVMArray<CELL>(g_NumRows * g_NumCols) {}

private:
    LONG OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
        PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful);
};

```

Рис. 25-1. Программа-пример Spreadsheet

см. след. стр.

Рис. 25-1. *продолжение*

```

////////////////////////////////////
LONG CVMSpreadsheet::OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
    PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful) {

    TCHAR sz[200];
    wsprintf(sz, TEXT("Violation: Attempting to %s"),
        fAttemptedRead ? TEXT("Read") : TEXT("Write"));
    SetDlgItemText(g_hwnd, IDC_LOG, sz);

    LONG lDisposition = EXCEPTION_EXECUTE_HANDLER;
    if (!fAttemptedRead) {

        // возвращаем значение, определяемое базовым классом
        lDisposition = CVMArray<CELL>::OnAccessViolation(pvAddrTouched,
            fAttemptedRead, pep, fRetryUntilSuccessful);
    }
    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;

```

Рис. 25-1. *продолжение*

```

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 определено
            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;
}
}

```

см. след. стр.

Рис. 25-1. *продолжение*

```

////////////////////////////////////
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_SPREADSHEET), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конец файла //////////////////////////////////

```

VMArray.h

```

/*****
Модуль: VMArray.h
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#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); }

```

Рис. 25-1. *продолжение*

```

    // может вызываться для более "тонкой" обработки,
    // если передать память не удалось
    LONG ExceptionFilter(PEXCEPTION_POINTERS pep,
        BOOL fRetryUntilSuccessful = FALSE);

protected:
    // замещается для более "тонкой" обработки исключения,
    // связанного с нарушением доступа
    virtual LONG OnAccessViolation(PVOID pvAddrTouched, BOOL fAttemptedRead,
        PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful);

private:
    static CVMArray* sm_pHead;    // адрес первого объекта
    CVMArray* m_pNext;           // адрес следующего объекта
    TYPE* m_pArray;              // указатель на регион, зарезервированный
                                // под массив (разреженную матрицу)
    DWORD m_cbReserve;           // размер зарезервированного региона (в байтах)

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

```

см. след. стр.

Рис. 25-1. *продолжение*

```

    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>
CVMArrау<TYPE>::~CVMArrау() {

    // освобождаем регион массива (возвращаем всю переданную ему память)
    VirtualFree(m_pArray, 0, MEM_RELEASE);

    // удаляем этот объект из связанного списка
    CVMArrау* p = sm_pHead;
    if (p == this) { // удаляем верхний узел
        sm_pHead = p->m_pNext;
    } else {

        BOOL fFound = FALSE;

        // проходим по списку сверху и модифицируем указатели
        for (; !fFound && (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(fFound);
    }
}

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

// предлагаемый по умолчанию механизм обработки нарушений доступа,
// возникающих при попытках передачи физической памяти
template <class TYPE>
LONG CVMArrау<TYPE>::OnAccessViolation(PVOID pvAddrTouched,
    BOOL fAttemptedRead, PEXCEPTION_POINTERS pep, BOOL fRetryUntilSuccessful) {

    BOOL fCommittedStorage = FALSE; // считаем, что передать память не удалось

    do {
        // пытаемся передать физическую память
        fCommittedStorage = (NULL != VirtualAlloc(pvAddrTouched,
            sizeof(TYPE), MEM_COMMIT, PAGE_READWRITE));
    }

```

Рис. 25-1. *продолжение*

```

        // если передать память не удастся, предлагаем пользователю освободить память
        if (!fCommittedStorage && fRetryUntilSuccessful) {
            MessageBox(NULL,
                TEXT("Please close some other applications and Press OK."),
                TEXT("Insufficient Memory Available"), MB_ICONWARNING | MB_OK);
        }
    } while (!fCommittedStorage && fRetryUntilSuccessful);

    // если память передана, пытаемся возобновить выполнение программы;
    // в ином случае активизируем обработчик
    return(fCommittedStorage
        ? EXCEPTION_CONTINUE_EXECUTION : EXCEPTION_EXECUTE_HANDLER);
}

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

// фильтр, связываемый с отдельным объектом класса CVMArray
template <class TYPE>
LONG CVMArray<TYPE>::ExceptionFilter(PEXCEPTION_POINTERS pep,
    BOOL fRetryUntilSuccessful) {

    // 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 fAttemptedRead = (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, fAttemptedRead,
            pep, fRetryUntilSuccessful);
    }
    return(lDisposition);
}

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

// фильтр, связываемый со всеми объектами класса CVMArray
template <class TYPE>
LONG WINAPI CVMArray<TYPE>::UnhandledExceptionFilter(PEXCEPTION_POINTERS pep) {

```

см. след. стр.

Рис. 25-1. *продолжение*

```
// по умолчанию пытаемся использовать другой фильтр (самый безопасный путь)
LONG lDisposition = EXCEPTION_CONTINUE_SEARCH;

// мы обрабатываем только нарушение доступа
if (pex->ExceptionRecord->ExceptionCode == EXCEPTION_ACCESS_VIOLATION) {

    // проходим все узлы связанного списка
    for (CVMArrary* p = sm_pHead; p != NULL; p = p->m_pNext) {

        // Интересуемся, может ли данный узел обработать исключение.
        // Примечание: исключение НАДО обработать, иначе процесс будет закрыт!
        lDisposition = p->ExceptionFilter(pex, 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(pex);

    return(lDisposition);
}

/////////////////////// Конец файла /////////////////////////
```

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.

```

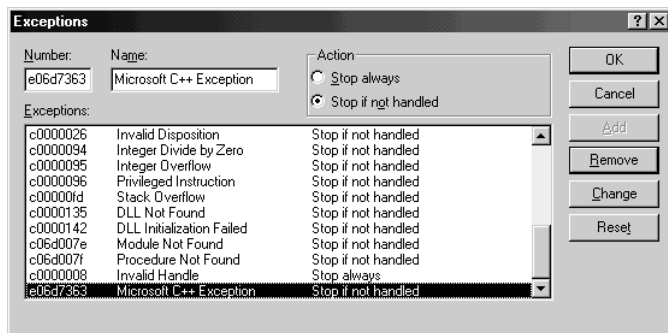
void ChunkyFunky() {
    try {
        // тело блока try
        ...
        throw 5;
    }
    catch (int x) {

        // тело блока catch
        ...
    }
}

void ChunkyFunky() {
    __try {
        // тело блока try
        ...
        RaiseException(Code=0xE06D7363,
            Flag=EXCEPTION_NONCONTINUABLE, Args=5);
    }
    __except ((ArgType == Integer) ?
        EXCEPTION_EXECUTE_HANDLER :
        EXCEPTION_CONTINUE_SEARCH) {
        // тело блока catch
        ...
    }
}

```

Обратите внимание на несколько интересных особенностей этого кода. Во-первых, *RaiseException* вызывается с кодом исключения 0xE06D7363. Это код программного исключения, выбранный разработчиками Visual C++ на случай выталкивания (throwing) исключений C++. Вы можете сами в этом убедиться, открыв диалоговое окно Exceptions отладчика и прокрутив его список до конца, как на следующей иллюстрации.



Заметьте также, что при выталкивании 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 выталкивает (throw) указанную переменную. Точный механизм того, как данные из переменной передаются *RaiseException*, не задокументирован, но догадаться о его реализации в компиляторе не так уж трудно.

И последнее, о чем хотелось бы сказать. Назначение фильтра *__except* — сравнивать тип *throw*-переменных с типом переменной, используемой в C++-операторе *catch*. Если их типы совпадают, фильтр возвращает EXCEPTION_EXECUTE_HANDLER, вызывая выполнение операторов в блоке *catch* (*__except*). А если они не совпадают, фильтр возвращает EXCEPTION_CONTINUE_SEARCH для проверки «вышестоящих» по дереву вызовов фильтров *catch*.



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++.

Перехват структурных исключений в C++

Обычно механизм обработки исключений в C++ не позволяет приложению восстановиться после таких серьезных исключений, как нарушение доступа или деление на ноль. Однако Microsoft добавила поддержку соответствующей функциональности в свой компилятор. Так, следующий код предотвратит аварийное завершение процесса.

```
void main() {
    try {
        * (PBYTE) 0 = 0; // нарушение доступа
    }
    catch (...) {
        // этот код обрабатывает исключения, связанные с нарушением доступа
    }
    // процесс завершается корректно
}
```

И это прекрасно, так как приложение может корректно справляться с серьезными исключениями. Но было бы еще лучше, если бы блок `catch` как-то различал коды исключений — чтобы мы могли писать, например, такой исходный код:

```
void Functastic() {

    try {
        * (PBYTE) 0 = 0;      // нарушение доступа

        int x = 0;
        x = 5 / x;            // деление на ноль
    }
    catch (StructuredException) {
        switch (StructuredExceptionCode) {
            case EXCEPTION_ACCESS_VIOLATION:
                // здесь обрабатывается нарушение доступа
                break;

            case EXCEPTION_INT_DIVIDE_BY_ZERO:
                // здесь обрабатывается деление на ноль
                break;

            default:
                // другие исключения мы не обрабатываем
                throw; // может, какой-нибудь другой блок catch
                       // обработает это исключение
        }
    }
}
```

```
        break;        // никогда не выполняется
    }
}
}
```

Так вот, хочу Вас порадовать. В Visual C++ теперь возможно и такое. От Вас требуется создать C++-класс, используемый специально для идентификации структурных исключений. Например:

```
#include <eh.h>        // для доступа к _set_se_translator
:
class CSE {
public:
    // вызовите эту функцию для каждого потока
    static void MapSEtoCE() { _set_se_translator(TranslateSEtoCE); }
    operator DWORD() { return(m_er.ExceptionCode); }

private:
    CSE(PEXCEPTION_POINTERS pep) {
        m_er      = *pep->ExceptionRecord;
        m_context = *pep->ContextRecord;
    }
    static void _cdecl TranslateSEtoCE(UINT dwEC,
        PEXCEPTION_POINTERS pep) {
        throw CSE(pep);
    }

private:
    EXCEPTION_RECORD m_er;        // машинно-независимая информация об исключении
    CONTEXT          m_context;   // машинно-зависимая информация об исключении
};
```

Внутри входных функций потоков вызывайте статическую функцию-член *MapSEtoCE*. В свою очередь она обращается к библиотечной C-функции *_set_se_translator*, передавая ей адрес функции *TranslateSEtoCE* класса CSE. Вызов *_set_se_translator* сообщает C++, что при возбуждении структурных исключений Вы хотите вызывать *TranslateSEtoCE*. Эта функция вызывает конструктор CSE-объекта и инициализирует два элемента данных машинно-зависимой и машинно-независимой информацией об исключении. Созданный таким образом CSE-объект может быть вытолкнут так же, как и любая другая переменная. И теперь Ваш C++-код способен обрабатывать структурные исключения, захватывая (catching) переменную этого типа.

Вот пример захвата такого C++-объекта:

```
void Functastic() {

    CSE::MapSEtoCE(); // должна быть вызвана до возникновения исключений

    try {
        * (PBYTE) 0 = 0;        // нарушение доступа
        int x = 0;
        x = 5 / x;              // деление на ноль
    }
    catch (CSE se) {
        switch (se) { // вызывает функцию-член оператора DWORD()
```

см. след. стр.

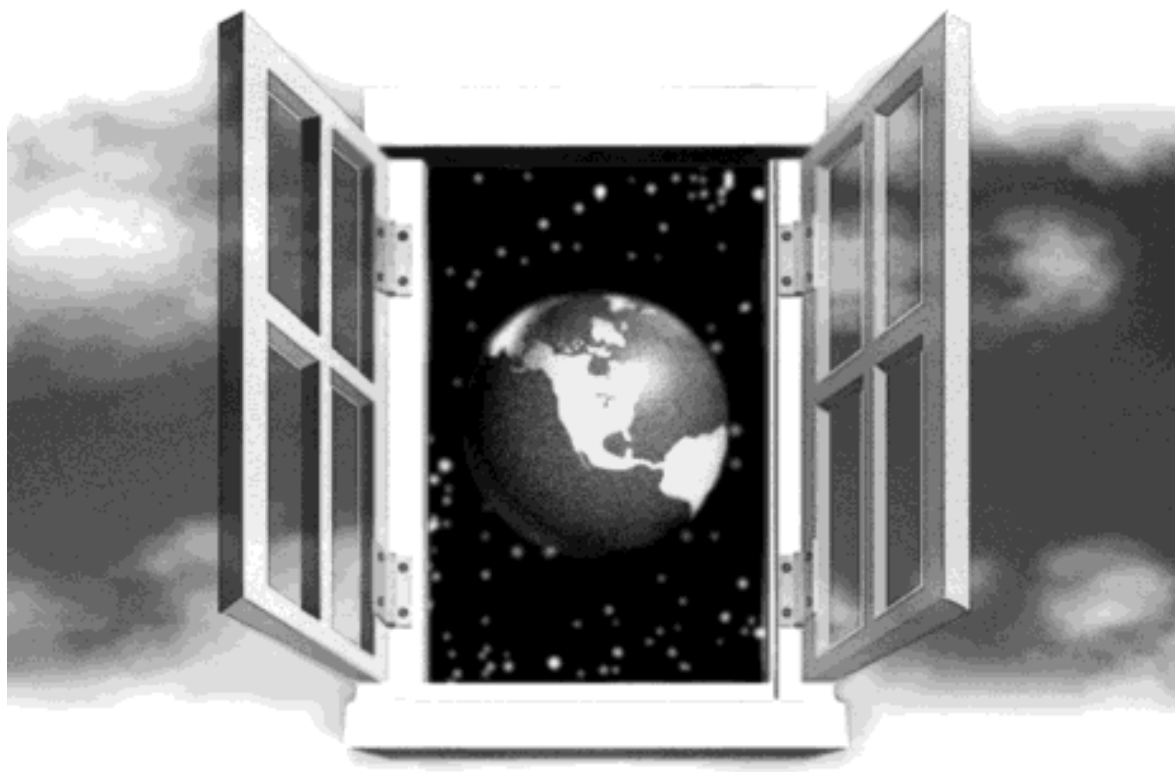
```
case EXCEPTION_ACCESS_VIOLATION:
    // здесь обрабатывается исключение, вызванное нарушением доступа
    break;

case EXCEPTION_INT_DIVIDE_BY_ZERO:
    // здесь обрабатывается исключение, вызванное делением на нуль
    break;

default:
    // другие исключения мы не обрабатываем
    throw;    // может, какой-нибудь другой блок catch
              // обработает это исключение
    break;   // никогда не выполняется
}
}
}
```

ЧАСТЬ VI

ОПЕРАЦИИ С ОКНАМИ



Оконные сообщения

В этой главе я расскажу, как работает подсистема передачи сообщений в Windows применительно к приложениям с графическим пользовательским интерфейсом. Разрабатывая подсистему управления окнами в Windows 2000 и Windows 98, Microsoft преследовала две основные цели:

- обратная совместимость с 16-разрядной Windows, облегчающая перенос существующих 16-разрядных приложений;
- отказоустойчивость подсистемы управления окнами, чтобы ни один поток не мог нарушить работу других потоков в системе.

К сожалению, эти цели прямо противоречат друг другу. В 16-разрядной Windows передача сообщения в окно всегда осуществляется синхронно: отправитель не может продолжить работу, пока окно не обработает полученное сообщение. Обычно так и нужно. Но, если на обработку сообщения потребуется длительное время или если окно «зависнет», выполнение отправителя просто прекратится. А значит, такая операционная система не вправе претендовать на устойчивость к сбоям.

Это противоречие было серьезным вызовом для команды разработчиков из Microsoft. В итоге было выбрано компромиссное решение, отвечающее двум вышеупомянутым целям. Помните о них, читая эту главу, и Вы поймете, почему Microsoft сделала именно такой выбор.

Для начала рассмотрим некоторые базовые принципы. Один процесс в Windows может создать до 10 000 User-объектов различных типов — значков, курсоров, оконных классов, меню, таблиц клавиш-акселераторов и т. д. Когда поток из какого-либо процесса вызывает функцию, создающую один из этих объектов, последний переходит во владение процесса. Поэтому, если процесс завершается, не уничтожив данный объект явным образом, операционная система делает это за него. Однако два User-объекта (окна и ловушки) принадлежат только создавшему их потоку. И вновь, если поток создает окно или устанавливает ловушку, а потом завершается, операционная система автоматически уничтожает окно или удаляет ловушку.

Этот принцип принадлежности окон и ловушек создавшему их потоку оказывает существенное влияние на механизм функционирования окон: поток, создавший окно, должен обрабатывать все его сообщения. Поясню данный принцип на примере. Допустим, поток создал окно, а затем прекратил работу. Тогда его окно уже не получит сообщение WM_DESTROY или WM_NCDESTROY, потому что поток уже завершился и обрабатывать сообщения, посылаемые этому окну, больше некому.

Это также означает, что каждому потоку, создавшему хотя бы одно окно, система выделяет очередь сообщений, используемую для их диспетчеризации. Чтобы окно в конечном счете получило эти сообщения, поток *должен* иметь собственный цикл выборки сообщений. В этой главе мы детально рассмотрим, что представляют собой

очереди сообщений потоков. В частности, я расскажу, как сообщения помещаются в эту очередь и как они извлекаются из нее, а потом обрабатываются.

Очередь сообщений потока

Как я уже говорил, одна из главных целей Windows — предоставить всем приложениям отказоустойчивую среду. Для этого любой поток должен выполняться в такой среде, где он может считать себя единственным. Точнее, у каждого потока должны быть очереди сообщений, полностью независимые от других потоков. Кроме того, для каждого потока нужно смоделировать среду, позволяющую ему самостоятельно управлять фокусом ввода с клавиатуры, активизировать окна, захватывать мышь и т. д.

Создавая какой-либо поток, система предполагает, что он не будет иметь отношения к поддержке пользовательского интерфейса. Это позволяет уменьшить объем выделяемых ему системных ресурсов. Но, как только поток обратится к той или иной GUI-функции (например, для проверки очереди сообщений или создания окна), система автоматически выделит ему дополнительные ресурсы, необходимые для выполнения задач, связанных с пользовательским интерфейсом. А если конкретнее, то система создает структуру `THREADINFO` и сопоставляет ее с этим потоком.

Элементы этой структуры используются, чтобы обмануть поток — заставить его считать, будто он выполняется в среде, принадлежащей только ему. `THREADINFO` — это внутренняя (недокументированная) структура, идентифицирующая очередь асинхронных сообщений потока (`posted-message queue`), очередь синхронных сообщений потока (`sent-message queue`), очередь ответных сообщений (`reply-message queue`), очередь виртуального ввода (`virtualized input queue`) и флаги пробуждения (`wake flags`); она также включает ряд других переменных-членов, характеризующих локальное состояние ввода для данного потока. На рис. 26-1 показаны структуры `THREADINFO`, сопоставленные с тремя потоками.

Структура `THREADINFO` — фундамент всей подсистемы передачи сообщений; читая следующие разделы, время от времени посматривайте на эту иллюстрацию.

Посылка асинхронных сообщений в очередь потока

Когда с потоком связывается структура `THREADINFO`, он получает свой набор очередей сообщений. Если процесс создает три потока и все они вызывают функцию *CreateWindow*, то и наборов очередей сообщений будет тоже три. Сообщения ставятся в очередь асинхронных сообщений вызовом функции *PostMessage*:

```
BOOL PostMessage(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

При вызове этой функции система определяет, каким потоком создано окно, идентифицируемое параметром *hwnd*. Далее система выделяет блок памяти, сохраняет в нем параметры сообщения и записывает этот блок в очередь асинхронных сообщений данного потока. Кроме того, функция устанавливает флаг пробуждения `QS_POSTMESSAGE` (о нем — чуть позже). Возврат из *PostMessage* происходит сразу после того, как сообщение поставлено в очередь, поэтому вызывающий поток остается в неведении, обработано ли оно процедурой соответствующего окна. На самом деле вполне вероятно, что окно даже не получит это сообщение. Такое возможно, если поток, создавший это окно, завершится до того, как обработает все сообщения из своей очереди.

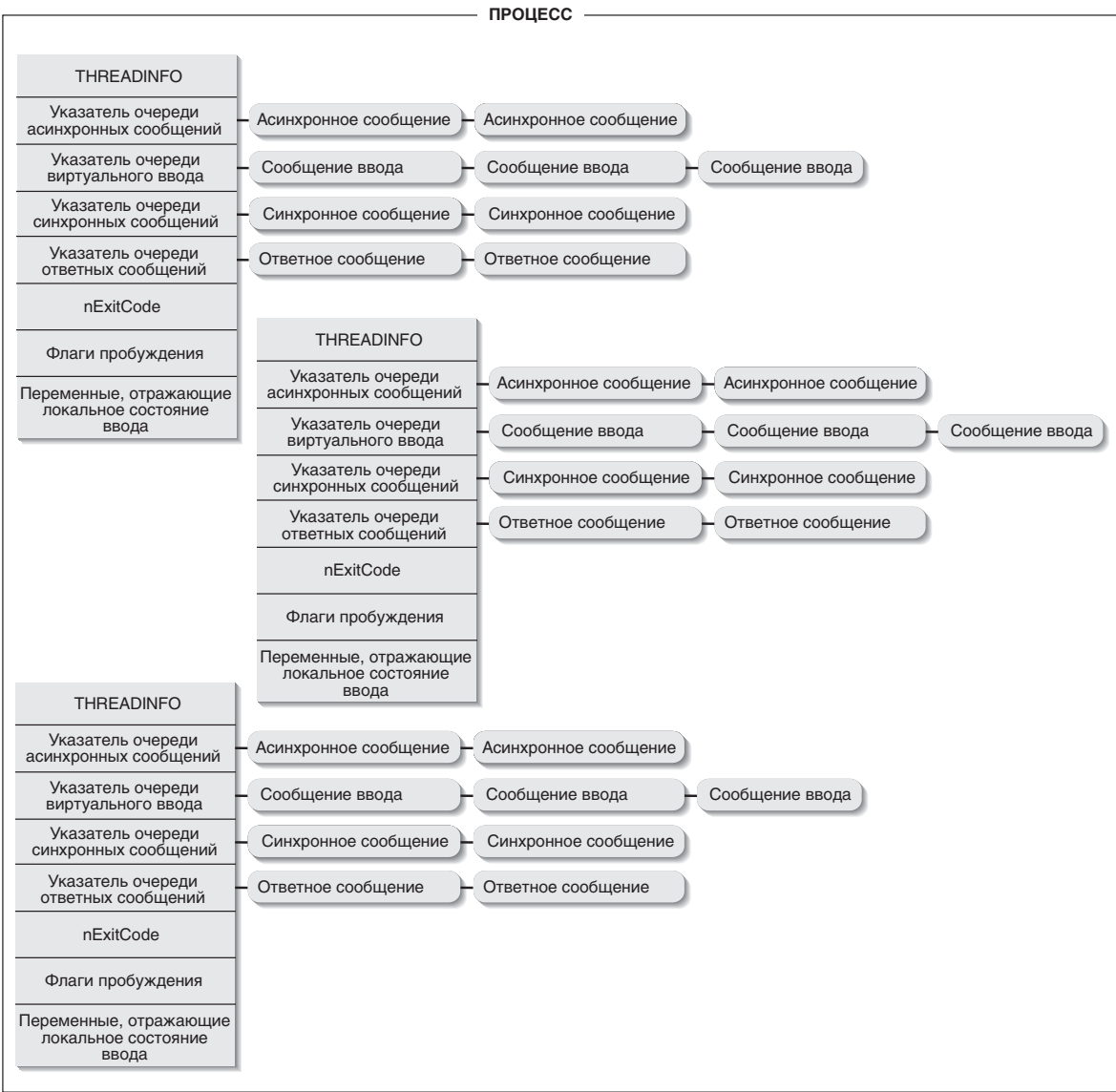


Рис. 26-1. Три потока и соответствующие им структуры *THREADINFO*

Сообщение можно поставить в очередь асинхронных сообщений потока и вызовом *PostThreadMessage*:

```
BOOL PostThreadMessage(  
    DWORD dwThreadId,  
    UINT uMsg,  
    WPARAM wParam,  
    LPARAM lParam);
```



Какой поток создал окно, можно определить с помощью *GetWindowThreadProcessId*:

```
DWORD GetWindowThreadProcessId(
    HWND hwnd,
    PDWORD pdwProcessId);
```

Она возвращает уникальный общесистемный идентификатор потока, который создал окно, определяемое параметром *hwnd*. Передав адрес переменной типа `DWORD` в параметре *pdwProcessId*, можно получить и уникальный общесистемный идентификатор процесса, которому принадлежит этот поток. Но обычно такой идентификатор не нужен, и мы просто передаем `NULL`.

Нужный поток идентифицируется первым параметром, *dwThreadId*. Когда сообщение помещено в очередь, элемент *hwnd* структуры `MSG` устанавливается как `NULL`. Применяется эта функция, когда приложение выполняет какую-то особую обработку в основном цикле выборки сообщений потока, — в этом случае он пишется так, чтобы после выборки сообщения функцией *GetMessage* (или *PeekMessage*) код в цикле сравнивал *hwnd* с `NULL` и, выполняя эту самую особую обработку, мог проверить значение элемента *msg* структуры `MSG`. Если поток определил, что сообщение не адресовано какому-либо окну, *DispatchMessage* не вызывается, и цикл переходит к выборке следующего сообщения.

Как и *PostMessage*, функция *PostThreadMessage* возвращает управление сразу после того, как сообщение поставлено в очередь потока. И вновь вызывающий поток остается в неведении о дальнейшей судьбе сообщения.

И, наконец, еще одна функция, позволяющая поместить сообщение в очередь асинхронных сообщений потока:

```
VOID PostQuitMessage(int nExitCode);
```

Она вызывается для того, чтобы завершить цикл выборки сообщений потока. Ее вызов аналогичен вызову:

```
PostThreadMessage(GetCurrentThreadId(), WM_QUIT, nExitCode, 0);
```

Но в действительности *PostQuitMessage* не помещает сообщение ни в одну из очередей структуры `THREADINFO`. Эта функция просто устанавливает флаг пробуждения `QS_QUIT` (о нем я тоже расскажу чуть позже) и элемент *nExitCode* структуры `THREADINFO`. Так как эти операции не могут вызвать ошибку, функция *PostQuitMessage* не возвращает никаких значений (`VOID`).

Посылка синхронных сообщений окну

Оконное сообщение можно отправить непосредственно оконной процедуре вызовом *SendMessage*:

```
LRESULT SendMessage(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

Оконная процедура обработает сообщение, и только по окончании обработки функция *SendMessage* вернет управление. Благодаря этому ее используют гораздо чаще, чем *PostMessage* или *PostThreadMessage*. При переходе к выполнению следующей строки кода поток, вызвавший *SendMessage*, может быть уверен, что сообщение уже обработано.

Вот как работает *SendMessage*. Если поток вызывает *SendMessage* для отправки сообщения окну, созданному им же, то функция просто обращается к оконной процедуре соответствующего окна как к подпрограмме. Закончив обработку, оконная процедура передает функции *SendMessage* некое значение, а та возвращает его вызвавшему потоку.

Однако, если поток посылает сообщение окну, созданному другим потоком, операции, выполняемые функцией *SendMessage*, значительно усложняются.¹ Windows требует, чтобы оконное сообщение обрабатывалось потоком, создавшим окно. Поэтому, если вызвать *SendMessage* для отправки сообщения окну, созданному в другом процессе и, естественно, другим потоком, Ваш поток не сможет обработать это сообщение — ведь он не работает в адресном пространстве чужого процесса, а потому не имеет доступа к коду и данным соответствующей оконной процедуры. И действительно, Ваш поток приостанавливается, пока другой поток обрабатывает сообщение. Поэтому, чтобы один поток мог отправить сообщение окну, созданному другим потоком, система должна выполнить следующие действия.

Во-первых, переданное сообщение присоединяется к очереди сообщений потока-приемника, в результате чего для этого потока устанавливается флаг `QS_SENDMESSAGE`. Во-вторых, если поток-приемник в данный момент выполняет какой-то код и не ожидает сообщений (через вызов *GetMessage*, *PeekMessage* или *WaitMessage*), переданное сообщение обработать не удастся — система не прервет работу потока для немедленной обработки сообщения. Но когда поток-приемник ждет сообщений, система сначала проверяет, установлен ли флаг пробуждения `QS_SENDMESSAGE`, и, если да, просматривает очередь синхронных сообщений, отыскивая первое из них. В очереди может находиться более одного сообщения. Скажем, несколько потоков одновременно послали сообщение одному и тому же окну. Тогда система просто ставит эти сообщения в очередь синхронных сообщений потока.

Итак, когда поток ждет сообщений, система извлекает из очереди синхронных сообщений первое и вызывает для его обработки нужную оконную процедуру. Если таких сообщений больше нет, флаг `QS_SENDMESSAGE` сбрасывается. Пока поток-приемник обрабатывает сообщение, поток, отправивший сообщение через *SendMessage*, простаивает, ожидая появления сообщения в очереди ответных сообщений. По окончании обработки значение, возвращенное оконной процедурой, передается асинхронно в очередь ответных сообщений потока-отправителя. Теперь он пробудится и извлечет упомянутое значение из ответного сообщения. Именно это значение и будет результатом вызова *SendMessage*. С этого момента поток-отправитель возобновляет работу в обычном режиме.

Ожидая возврата управления функцией *SendMessage*, поток в основном простаивает. Но кое-чем он может заняться: если другой поток посылает сообщение окну, созданному первым (ожидаящим) потоком, система тут же обрабатывает это сообщение, не дожидаясь, когда поток вызовет *GetMessage*, *PeekMessage* или *WaitMessage*.

Поскольку Windows обрабатывает межпоточные сообщения описанным выше образом, Ваш поток может зависнуть. Допустим, в потоке, обрабатывающем синхронное сообщение, имеется «жучок», из-за которого поток входит в бесконечный цикл. Что же произойдет с потоком, вызвавшим *SendMessage*? Возобновится ли когда-нибудь его выполнение? Значит ли это, что ошибка в одном приложении «подвесит» другое? Ответ — да!

¹ Это верно даже в том случае, если оба потока принадлежат одному процессу.

Избегать подобных ситуаций позволяют четыре функции, и первая из них — *SendMessageTimeout*:

```
LRESULT SendMessageTimeout(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam,
    UINT fuFlags,
    UINT uTimeout,
    PDWORD_PTR pdwResult);
```

Она позволяет задавать отрезок времени, в течение которого Вы готовы ждать ответа от другого потока на Ваше сообщение. Ее первые четыре параметра идентичны параметрам функции *SendMessage*. В параметре *fuFlags* можно передавать флаги *SMTO_NORMAL* (0), *SMTO_ABORTIFHUNG*, *SMTO_BLOCK*, *SMTO_NOTIMEOUTIFNOTHUNG* или комбинацию этих флагов.

Флаг *SMTO_ABORTIFHUNG* заставляет *SendMessageTimeout* проверить, не завис ли поток-приемник², и, если да, немедленно вернуть управление. Флаг *SMTO_NOTIMEOUTIFNOTHUNG* сообщает функции, что она должна игнорировать ограничение по времени, если поток-приемник не завис. Флаг *SMTO_BLOCK* предотвращает обработку вызывающим потоком любых других синхронных сообщений до возврата из *SendMessageTimeout*. Флаг *SMTO_NORMAL* определен в файле *WinUser.h* как 0; он используется в том случае, если Вы не указали другие флаги.

Я уже говорил, что ожидание потоком окончания обработки синхронного сообщения может быть прервано для обработки другого синхронного сообщения. Флаг *SMTO_BLOCK* предотвращает такое прерывание. Он применяется, только если поток, ожидая окончания обработки своего сообщения, не в состоянии обрабатывать прочие синхронные сообщения. Этот флаг иногда приводит к взаимной блокировке потоков до конца таймаута. Так, если Ваш поток отправит сообщение другому, а тому нужно послать сообщение Вашему, ни один из них не сможет продолжить обработку, и оба зависнут.

Параметр *uTimeout* определяет таймаут — время (в миллисекундах), в течение которого Вы готовы ждать ответного сообщения. При успешном выполнении функция возвращает *TRUE*, а результат обработки сообщения копируется по адресу, указанному в параметре *pdwResult*.

Кстати, прототип этой функции в заголовочном файле *WinUser.h* неверен. Функцию следовало бы определить как возвращающую значение типа *BOOL*, поскольку значение типа *LRESULT* на самом деле возвращается через ее параметр. Это создает определенные проблемы, так как *SendMessageTimeout* вернет *FALSE*, если Вы передадите неверный описатель окна или если закончится заданный период ожидания. Единственный способ узнать причину неудачного завершения функции — вызвать *GetLastError*. Последняя вернет 0 (*ERROR_SUCCESS*), если ошибка связана с окончанием периода ожидания. А если причина в неверном описателе, *GetLastError* даст код 1400 (*ERROR_INVALID_WINDOW_HANDLE*).

Если Вы обращаетесь к *SendMessageTimeout* для отправки сообщения окну, созданному вызывающим потоком, система просто вызывает оконную процедуру, помещая возвращаемое значение в *pdwResult*. Из-за этого код, расположенный за вызовом

² Операционная система считает поток зависшим, если он прекращает обработку сообщений более чем на 5 секунд.

SendMessageTimeout, не выполняется до тех пор, пока не заканчивается обработка сообщения, — ведь все эти операции осуществляются одним потоком.

Теперь рассмотрим вторую функцию, предназначенную для отправки межпоточных сообщений:

```
BOOL SendMessageCallback(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam,
    SENDASYNCPROC pfnResultCallback,
    ULONG_PTR dwData);
```

И вновь первые четыре параметра идентичны параметрам функции *SendMessage*. При вызове Вашим потоком *SendMessageCallback* отправляет сообщение в очередь синхронных сообщений потока-приемника и тут же возвращает управление вызывающему (т. е. Вашему) потоку. Закончив обработку сообщения, поток-приемник асинхронно отправляет свое сообщение в очередь ответных сообщений Вашего потока. Позже система уведомит Ваш поток об этом, вызвав написанную Вами функцию; у нее должен быть следующий прототип:

```
VOID CALLBACK ResultCallback(
    HWND hwnd,
    UINT uMsg,
    ULONG_PTR dwData,
    LRESULT lResult);
```

Адрес этой функции обратного вызова передается *SendMessageCallback* в параметре *pfnResultCallback*. А при вызове *ResultCallback* в первых двух параметрах передаются описатель окна, закончившего обработку сообщения, и код (значение) самого сообщения. Параметр *dwData* функции *ResultCallback* всегда получает значение, переданное *SendMessageCallback* в одноименном параметре. (Система просто берет то, что указано там, и передает Вашей функции *ResultCallback*.) Последний параметр функции *ResultCallback* сообщает результат обработки сообщения, полученный от оконной процедуры.

Поскольку *SendMessageCallback*, передавая сообщение другому потоку, немедленно возвращает управление, *ResultCallback* вызывается после обработки сообщения потоком-приемником не сразу, а с задержкой. Сначала поток-приемник асинхронно ставит сообщение в очередь ответных сообщений потока-отправителя. Затем при первом же вызове потоком-отправителем любой из функций *GetMessage*, *PeekMessage*, *WaitMessage* или одной из *Send*-функций сообщение извлекается из очереди ответных сообщений, и лишь потом вызывается Ваша функция *ResultCallback*.

Существует и другое применение функции *SendMessageCallback*. В Windows предусмотрен метод, позволяющий разослать сообщение всем перекрывающимся окнам (overlapped windows) в системе; он состоит в том, что Вы вызываете *SendMessage* и в параметре *hwnd* передаете ей *HWND_BROADCAST* (определенный как *-1*). Этот метод годится только для ширококвещательной рассылки сообщений, возвращаемые значения которых Вас не интересуют, поскольку функция способна вернуть лишь одно значение, *LRESULT*. Но, используя *SendMessageCallback*, можно получить результаты обработки «широковещательного» сообщения от каждого перекрытого окна. Ваша функция *SendMessageCallback* будет вызываться с результатом обработки сообщения от каждого из таких окон.

Если *SendMessageCallback* вызывается для отправки сообщения окну, созданному вызывающим потоком, система немедленно вызывает оконную процедуру, а после обработки сообщения — функцию *ResultCallBack*. После возврата из *ResultCallback* выполнение начинается со строки, следующей за вызовом *SendMessageCallback*.

Третья функция, предназначенная для передачи межпоточных сообщений:

```
BOOL SendNotifyMessage(
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

Поместив сообщение в очередь синхронных сообщений потока-приемника, она немедленно возвращает управление вызывающему потоку. Так ведет себя и *PostMessage*, помните? Но два отличия *SendNotifyMessage* от *PostMessage* все же есть.

Во-первых, если *SendNotifyMessage* посылает сообщение окну, созданному другим потоком, приоритет данного синхронного сообщения выше приоритета асинхронных сообщений, находящихся в очереди потока-приемника. Иными словами, сообщения, помещаемые в очередь с помощью *SendNotifyMessage*, всегда извлекаются до выборки сообщений, отправленных через *PostMessage*.

Во-вторых, если сообщение посылается окну, созданному вызывающим потоком, *SendNotifyMessage* работает точно так же, как и *SendMessage*, т. е. не возвращает управление до окончания обработки сообщения.

Большинство синхронных сообщений посылается окну для уведомления — чтобы сообщить ему об изменении состояния и чтобы оно как-то отреагировало на это, прежде чем Вы продолжите свою работу. Например, *WM_ACTIVATE*, *WM_DESTROY*, *WM_ENABLE*, *WM_SIZE*, *WM_SETFOCUS*, *WM_MOVE* и многие другие сообщения — это просто уведомления, посылаемые системой окну в синхронном, а не асинхронном режиме. Поэтому система не прерывает свою работу только ради того, чтобы оконная процедура могла их обработать. Прямо противоположный эффект дает отправка сообщения *WM_CREATE* — тогда система ждет, когда окно закончит его обработку. Если возвращено значение *-1*, значит, окно не создано.

И, наконец, четвертая функция, связанная с обработкой межпоточных сообщений:

```
BOOL ReplyMessage(LRESULT lResult);
```

Она отличается от трех описанных выше. В то время как *Send*-функции используются посылающим сообщения потоком для защиты себя от зависания, *ReplyMessage* вызывается потоком, принимающим оконное сообщение. Вызвав ее, поток как бы говорит системе, что он уже получил результат обработки сообщения и что этот результат нужно упаковать и асинхронно отправить в очередь ответных сообщений потока-отправителя. Последний сможет пробудиться, получить результат и возобновить работу.

Поток, вызывающий *ReplyMessage*, передает результат обработки сообщения через параметр *lResult*. После вызова *ReplyMessage* выполнение потока-отправителя возобновляется, а поток, занятый обработкой сообщения, продолжает эту обработку. Ни один из потоков не приостанавливается — оба работают, как обычно. Когда поток, обрабатывающий сообщение, выйдет из своей оконной процедуры, любое возвращаемое значение просто игнорируется.

Заметьте: *ReplyMessage* надо вызывать из оконной процедуры, получившей сообщение, но не из потока, вызвавшего одну из *Send*-функций. Поэтому, чтобы написать «защищенный от зависаний» код, следует заменить все вызовы *SendMessage* вызовами

одной из трех *Send*-функций и не полагаться на то, что оконная процедура будет вызывать именно *ReplyMessage*.

Учтите также, что вызов *ReplyMessage* при обработке сообщения, посланного этим же потоком, не влечет никаких действий. На это и указывает значение, возвращаемое *ReplyMessage*: TRUE — при обработке межпоточного сообщения и FALSE — при попытке вызова функции для обработки внутривиточного сообщения.

Если Вас интересует, является обрабатываемое сообщение внутривиточным или межпоточным, вызовите функцию *InSendMessage*:

```
BOOL InSendMessage();
```

Имя этой функции не совсем точно соответствует тому, что она делает в действительности. На первый взгляд, функция должна возвращать TRUE, если поток обрабатывает синхронное сообщение, и FALSE — при обработке им асинхронного сообщения. Но это не так. Она возвращает TRUE, если поток обрабатывает межпоточное синхронное сообщение, и FALSE — при обработке им внутривиточного сообщения (синхронного или асинхронного). Возвращаемые значения функций *InSendMessage* и *ReplyMessage* идентичны.

Есть еще одна функция, позволяющая определить тип сообщения, которое обрабатывается Вашей оконной процедурой:

```
DWORD InSendMessageEx(PVOID pvReserved);
```

Вызывая ее, Вы должны передать NULL в параметре *pvReserved*. Возвращаемое значение указывает на тип обрабатываемого сообщения. Значение ISMEX_NOSEND (0) говорит о том, что поток обрабатывает внутривиточное синхронное или асинхронное сообщение. Остальные возвращаемые значения представляют собой комбинацию битовых флагов, описанных в следующей таблице.

| Флаг | Описание |
|----------------|--|
| ISMEX_SEND | Поток обрабатывает межпоточное синхронное сообщение, посланное через <i>SendMessage</i> или <i>SendMessageTimeout</i> ; если флаг ISMEX_REPLIED не установлен, поток-отправитель блокируется в ожидании ответа |
| ISMEX_NOTIFY | Поток обрабатывает межпоточное синхронное сообщение, посланное через <i>SendMessage</i> ; поток-отправитель не ждет ответа и не блокируется |
| ISMEX_CALLBACK | Поток обрабатывает межпоточное синхронное сообщение, посланное через <i>SendMessageCallback</i> ; поток-отправитель не ждет ответа и не блокируется |
| ISMEX_REPLIED | Поток обрабатывает межпоточное синхронное сообщение и уже вызвал <i>ReplyMessage</i> ; поток-отправитель не блокируется |

Пробуждение потока

Когда поток вызывает *GetMessage* или *WaitMessage* и никаких сообщений для него или созданных им окон нет, система может приостановить выполнение потока, и тогда он уже не получает процессорное время. Как только потоку будет отправлено синхронное или асинхронное сообщение, система установит флаг пробуждения, указывающий, что теперь поток должен получать процессорное время и обработать сообщение. Если пользователь ничего не набирает на клавиатуре и не трогает мышь, то обычно в таких обстоятельствах никаких сообщений окнам не посылается. А это значит, что большинство потоков в системе не получает процессорное время.

Флаги состояния очереди

Во время выполнения поток может опросить состояние своих очередей вызовом *GetQueueStatus*:

```
DWORD GetQueueStatus(UINT fuFlags);
```

Параметр *fuFlags* — флаг или группа флагов, объединенных побитовой операцией OR; он позволяет проверить значения отдельных битов пробуждения (wake bits). Допустимые значения флагов и их смысл описаны в следующей таблице.

| Флаг | Сообщение в очереди |
|-------------------|--|
| QS_KEY | WM_KEYUP, WM_KEYDOWN, WM_SYSKEYUP или WM_SYSKEYDOWN |
| QS_MOUSEMOVE | WM_MOUSEMOVE |
| QS_MOUSEBUTTON | WM_?BUTTON* (где знак вопроса заменяет букву L, M или R, а звездочка — DOWN, UP или DBLCLK) |
| QS_MOUSE | To же, что QS_MOUSEMOVE QS_MOUSEBUTTON |
| QS_INPUT | To же, что QS_MOUSE QS_KEY |
| QS_PAINT | WM_PAINT |
| QS_TIMER | WM_TIMER |
| QS_HOTKEY | WM_HOTKEY |
| QS_POSTMESSAGE | Асинхронное сообщение (отличное от события аппаратного ввода); этот флаг идентичен QS_ALLPOSTMESSAGE с тем исключением, что сбрасывается при отсутствии асинхронных сообщений в диапазоне действия фильтра сообщений |
| QS_ALLPOSTMESSAGE | Асинхронное сообщение (отличное от события аппаратного ввода); этот флаг идентичен QS_POSTMESSAGE с тем исключением, что сбрасывается лишь при полном отсутствии каких-либо асинхронных сообщений (вне зависимости от фильтра сообщений) |
| QS_ALLEVENTS | To же, что QS_INPUT QS_POSTMESSAGE QS_TIMER QS_PAINT QS_HOTKEY |
| QS_QUIT | Сообщает о вызове <i>PostQuitMessage</i> ; этот флаг не задокументирован, его нет в WinUser.h, и он используется самой системой |
| QS_SENDMESSAGE | Синхронное сообщение, посланное другим потоком |
| QS_ALLINPUT | To же, что QS_ALLEVENTS QS_SENDMESSAGE |

При вызове *GetQueueStatus* параметр *fuFlags* сообщает функции, наличие каких типов сообщений в очереди следует проверить. Чем меньше идентификаторов QS_* объединено побитовой операцией OR, тем быстрее отрабатывается вызов. Результат сообщается в старшем слове значения, возвращаемого функцией. Возвращаемый набор флагов всегда представляет собой подмножество того набора, который Вы запросили от функции. Например, если Вы делаете такой вызов:

```
BOOL fPaintMsgWaiting = HIWORD(GetQueueStatus(QS_TIMER)) & QS_PAINT;
```

то значение *fPaintMsgWaiting* всегда будет равно FALSE независимо от наличия в очереди сообщения WM_PAINT, так как флаг QS_PAINT функции не передан.

Младшее слово возвращаемого значения содержит типы сообщений, которые помещены в очередь, но не обработаны с момента последнего вызова *GetQueueStatus*, *GetMessage* или *PeekMessage*.

Не все флаги пробуждения обрабатываются системой одинаково. Флаг QS_MOUSEMOVE устанавливается, если в очереди есть необработанное сообщение WM_MOUSE-

MOVE. Когда *GetMessage* или *PeekMessage* (с флагом `PM_REMOVE`) извлекают последнее сообщение `WM_MOUSEMOVE`, флаг сбрасывается и остается в таком состоянии, пока в очереди ввода снова не окажется сообщение `WM_MOUSEMOVE`. Флаги `QS_KEY`, `QS_MOUSEBUTTON` и `QS_HOTKEY` действуют при соответствующих сообщениях аналогичным образом.

Флаг `QS_PAINT` обрабатывается иначе. Он устанавливается, если в окне, созданном данным потоком, имеется недействительная, требующая перерисовки область. Когда область, занятая всеми окнами, созданными одним потоком, становится действительной (обычно в результате вызова *ValidateRect*, *ValidateRegion* или *BeginPaint*), флаг `QS_PAINT` сбрасывается. Еще раз подчеркну: данный флаг сбрасывается, только если становятся действительными все окна, принадлежащие потоку. Вызов *GetMessage* или *PeekMessage* на этот флаг пробуждения не влияет.

Флаг `QS_POSTMESSAGE` устанавливается, когда в очереди асинхронных сообщений потока есть минимум одно сообщение. При этом не учитываются аппаратные сообщения, находящиеся в очереди виртуального ввода потока. Этот флаг сбрасывается после обработки всех сообщений из очереди асинхронных сообщений.

Флаг `QS_TIMER` устанавливается после срабатывания таймера (созданного потоком). После того как функция *GetMessage* или *PeekMessage* вернет `WM_TIMER`, флаг сбрасывается и остается в таком состоянии, пока таймер вновь не сработает.

Флаг `QS_SENDMESSAGE` указывает, что сообщение появилось в очереди синхронных сообщений. Он используется системой для идентификации и обработки межпоточных синхронных сообщений, а для внутривиджетных синхронных сообщений не применяется. Вы можете указывать флаг `QS_SENDMESSAGE`, но необходимость в нем возникает крайне редко. Я ни разу не видел его ни в одном приложении.

Есть еще один (недокументированный) флаг состояния очереди — `QS_QUIT`. Он устанавливается при вызове потоком *PostQuitMessage*. Сообщение `WM_QUIT` при этом не добавляется к очереди сообщений. И учтите, что *GetQueueStatus* не возвращает состояние этого флага.

Алгоритм выборки сообщений из очереди потока

Когда поток вызывает *GetMessage* или *PeekMessage*, система проверяет флаги состояния очередей потока и определяет, какое сообщение надо обработать (рис. 26-2).

1. Если флаг `QS_SENDMESSAGE` установлен, система отправляет сообщение соответствующей оконной процедуре. *GetMessage* и *PeekMessage* контролируют процесс обработки и не передают управление потоку сразу после того, как оконная процедура обрабатывает сообщение; вместо этого обе функции ждут следующего сообщения.
2. Если очередь асинхронных сообщений потока не пуста, *GetMessage* и *PeekMessage* заполняют переданную им структуру `MSG` и возвращают управление. Цикл выборки сообщений (расположенный в потоке) в этот момент обычно обращается к *DispatchMessage*, чтобы соответствующая оконная процедура обработала сообщение.
3. Если флаг `QS_QUIT` установлен, *GetMessage* и *PeekMessage* возвращают сообщение `WM_QUIT` (параметр *wParam* которого содержит указанный код завершения) и сбрасывают этот флаг.
4. Если в очереди виртуального ввода потока есть какие-то сообщения, *GetMessage* и *PeekMessage* возвращают сообщение, связанное с аппаратным вводом.

5. Если флаг `QS_PAINT` установлен, *GetMessage* и *PeekMessage* возвращают сообщение `WM_PAINT` для соответствующего окна.
6. Если флаг `QS_TIMER` установлен, *GetMessage* и *PeekMessage* возвращают сообщение `WM_TIMER`.

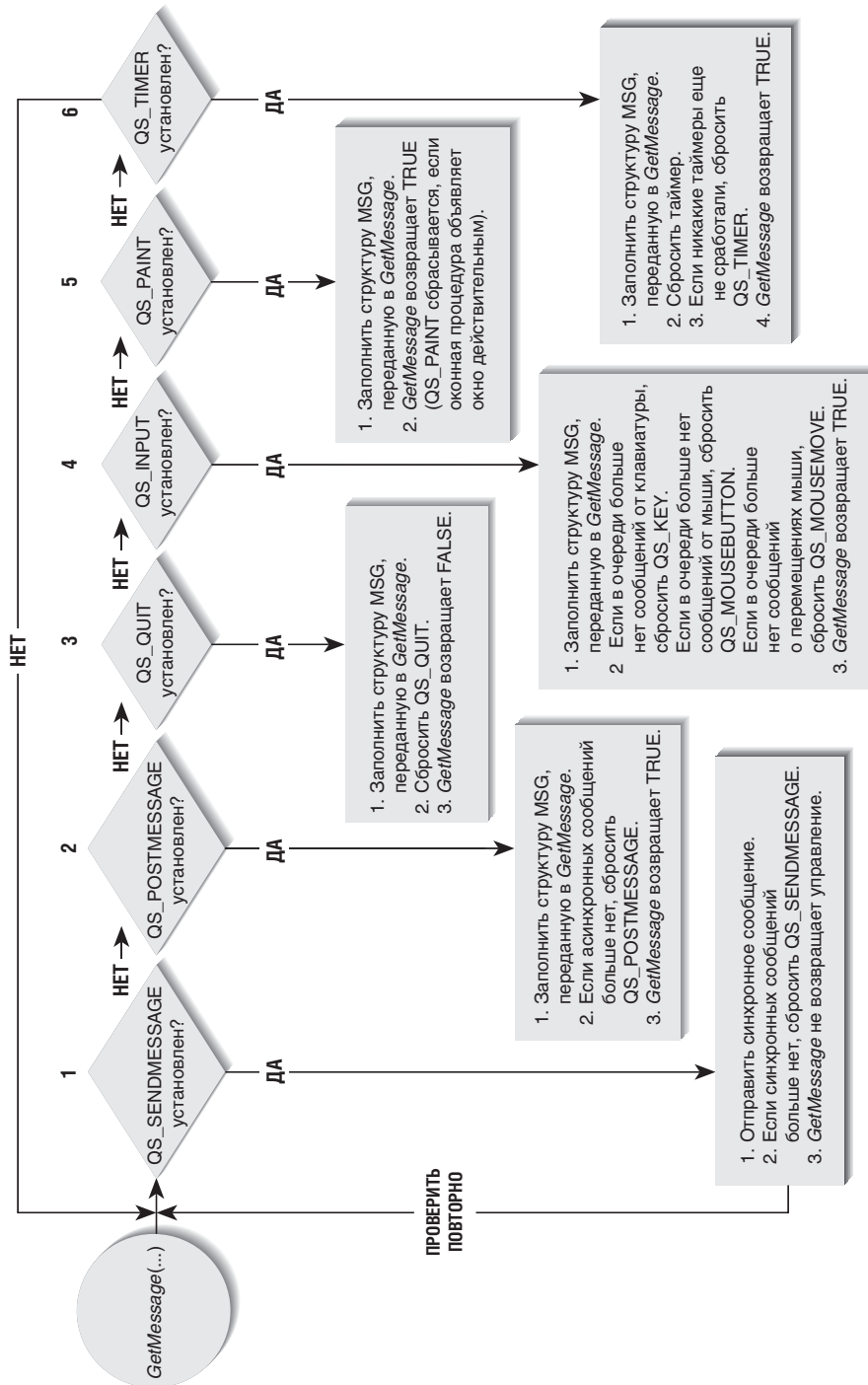


Рис. 26-2. Алгоритм выборки сообщений из очереди потока

Хоть и трудно в это поверить, но для такого безумия есть своя причина. Главное, из чего исходила Microsoft, разрабатывая описанный алгоритм, — приложения должны слушаться пользователя, и именно его действия (с клавиатурой и мышью) управляют программой, порождая события аппаратного ввода. Работая с программой, пользователь может нажать кнопку мыши, что приводит к генерации последовательности определенных событий. А программа порождает отдельные события, асинхронно отправляя сообщения в очередь потока.

Так, нажатие кнопки мыши могло бы заставить окно, которое обрабатывает сообщение WM_LBUTTONDOWN, послать три асинхронных сообщения разным окнам. Поскольку эти три программных события возникают в результате аппаратного события, система обрабатывает их до того, как принимает новое аппаратное событие, инициируемое пользователем. И именно поэтому очередь асинхронных сообщений проверяется раньше очереди виртуального ввода.

Прекрасный пример такой последовательности событий — вызов функции *TranslateMessage*, проверяющей, не было ли выбрано из очереди ввода сообщение WM_KEYDOWN или WM_SYSKEYDOWN. Если одно из этих сообщений выбрано, система проверяет, можно ли преобразовать информацию о виртуальной клавише в символьный эквивалент. Если это возможно, *TranslateMessage* вызывает *PostMessage*, чтобы поместить в очередь асинхронных сообщений WM_CHAR или WM_SYSCHAR. При следующем вызове *GetMessage* система проверяет содержимое очереди асинхронных сообщений и, если в ней есть сообщение, извлекает его и возвращает потоку. Возвращается либо WM_CHAR, либо WM_SYSCHAR. При следующем вызове *GetMessage* система обнаруживает, что очередь асинхронных сообщений пуста. Тогда она проверяет очередь ввода, где и находит сообщение WM_(SYS)KEYUP; именно оно и возвращается функцией *GetMessage*.

Поскольку система устроена так, а не иначе, последовательность аппаратных событий:

```
WM_KEYDOWN
WM_KEYUP
```

генерирует следующую последовательность сообщений для оконной процедуры (при этом предполагается, что информацию о виртуальной клавише можно преобразовать в ее символьный эквивалент):

```
WM_KEYDOWN
WM_CHAR
WM_KEYUP
```

Вернемся к тому, как система решает, что за сообщение должна вернуть функция *GetMessage* или *PeekMessage*. Просмотрев очередь асинхронных сообщений, система, прежде чем перейти к проверке очереди виртуального ввода, проверяет флаг QS_QUIT. Вспомните: этот флаг устанавливается, когда поток вызывает *PostQuitMessage*. Вызов *PostQuitMessage* дает примерно тот же эффект, что и вызов *PostMessage*, которая помещает сообщение в конец очереди и тем самым заставляет обрабатывать его до проверки очереди ввода. Так почему же *PostQuitMessage* устанавливает флаг вместо того, чтобы поместить WM_QUIT в очередь сообщений? На то есть две причины.

Во-первых, в условиях нехватки памяти может получиться так, что асинхронное сообщение не удастся поместить в очередь. Но, если приложение хочет завершиться, оно должно завершиться — тем более при нехватке памяти. Вторая причина в том, что этот флаг позволяет потоку закончить обработку остальных асинхронных сообщений до завершения его цикла выборки сообщений. Поэтому в следующем фрагмен-

те кода сообщение WM_USER будет извлечено до WM_QUIT, даже если WM_USER асинхронно помещено в очередь после вызова *PostQuitMessage*.

```
case WM_CLOSE:
    PostQuitMessage(0);
    PostMessage(hwnd, WM_USER, 0, 0);
```

А теперь о последних двух сообщениях: WM_PAINT и WM_TIMER. Сообщение WM_PAINT имеет низкий приоритет, так как прорисовка экрана — операция не самая быстрая. Если бы это сообщение посылалось всякий раз, когда окно становится недействительным, быстродействие системы снизилось бы весьма ощутимо. Но помещая WM_PAINT после ввода с клавиатуры, система работает гораздо быстрее. Например, из меню можно вызвать какую-нибудь команду, открывающую диалоговое окно, выбрать в нем что-то, нажать клавишу Enter — и проделать все это даже до того, как окно появится на экране. Достаточно быстро нажимая клавиши, Вы наверняка заметите, что сообщения об их нажатии извлекаются прежде, чем дело доходит до сообщений WM_PAINT. А когда Вы нажимаете клавишу Enter, подтверждая тем самым значения параметров, указанных в диалоговом окне, система разрушает окно и сбрасывает флаг QS_PAINT.

Приоритет WM_TIMER еще ниже, чем WM_PAINT. Почему? Допустим, какая-то программа обновляет свое окно всякий раз, когда получает сообщение WM_TIMER. Если бы оно поступало слишком часто, программа просто не смогла бы обновлять свое окно. Но поскольку сообщения WM_PAINT обрабатываются до WM_TIMER, такая проблема не возникает.



Функции *GetMessage* и *PeekMessage* проверяют флаги пробуждения только для вызывающего потока. Это значит, что потоки никогда не смогут извлечь сообщения из очереди, присоединенной к другому потоку, включая сообщения для потоков того же процесса.

Пробуждение потока с использованием объектов ядра или флагов состояния очереди

Функции *GetMessage* и *PeekMessage* приостанавливают поток до тех пор, пока ему не понадобится выполнить какую-нибудь задачу, связанную с пользовательским интерфейсом. Иногда то же самое было бы удобно и при обработке других задач. Для этого поток должен как-то узнавать о завершении операции, не относящейся к пользовательскому интерфейсу.

Чтобы поток ждал собственных сообщений, вызовите функцию *MsgWaitForMultipleObjects* или *MsgWaitForMultipleObjectsEx*:

```
DWORD MsgWaitForMultipleObjects(
    DWORD nCount,
    PHANDLE phObjects,
    BOOL fWaitAll,
    DWORD dwMilliseconds,
    DWORD dwWakeMask);

DWORD MsgWaitForMultipleObjectsEx(
    DWORD nCount,
    PHANDLE phObjects,
    DWORD dwMilliseconds,
```

см. след. стр.

```
DWORD dwWakeMask,
DWORD dwFlags);
```

Эти функции аналогичны *WaitForMultipleObjects* (см. главу 9). Разница в том, что при их использовании поток становится планируемым, когда освобождается какой-нибудь из указанных объектов ядра или когда оконное сообщение нужно переслать окну, созданному этим потоком.

Внутренне система просто добавляет объект ядра «событие» в массив описателей ядра. Параметр *dwWakeMask* сообщает системе, в какой момент объект-событие должно переходить в свободное состояние. Его допустимые значения идентичны тем, которые можно передавать в функцию *GetQueueStatus*.

WaitForMultipleObjects обычно возвращает индекс освобожденного объекта (в диапазоне от *WAIT_OBJECT_0* до *WAIT_OBJECT_0 + nCount - 1*). Задание параметра *dwWakeMask* равносильно добавлению еще одного описателя. При выполнении условия, определенного маской пробуждения, *MsgWaitForMultipleObjects(Ex)* возвращает значение *WAIT_OBJECT_0 + nCount*.

Вот пример вызова *MsgWaitForMultipleObjects*:

```
MsgWaitForMultipleObjects(0, NULL, TRUE, INFINITE, QS_INPUT);
```

Описатели синхронизирующих объектов в этом операторе не передаются — параметры *nCount* и *pbObjects* равны соответственно 0 и NULL. Мы указываем функции ждать освобождения всех объектов. Но в действительности задан лишь один объект, и с тем же успехом параметру *fWaitAll* можно было бы присвоить значение FALSE. Мы также сообщаем, что будем ждать — сколько бы времени это ни потребовало — появления в очереди ввода потока сообщения от клавиатуры или мыши.

Начав пользоваться функцией *MsgWaitForMultipleObjects* в своих программах, Вы быстро поймете, что она лишена многих важных качеств. Вот почему Microsoft пришлось создать более совершенную функцию *MsgWaitForMultipleObjectsEx*, которая позволяет задать в параметре *dwFlags* любую комбинацию следующих флагов.

| Флаг | Описание |
|---------------------|---|
| MWMO_WAITALL | Функция ждет освобождения всех объектов ядра и появления в очереди потока указанных сообщений (без этого флага функция ждет освобождения одного из объектов ядра или появления в очереди одного из указанных сообщений) |
| MWMO_ALERTABLE | Функция ждет в «тревожном» состоянии |
| MWMO_INPUTAVAILABLE | Функция ждет появления в очереди потока одного из указанных сообщений |

Если Вам не нужны эти дополнительные возможности, передайте в *dwFlags* нулевое значение.

При использовании *MsgWaitForMultipleObjects(Ex)* учитывайте, что:

- эти функции лишь включают описатель внутреннего объекта ядра «событие» в массив описателей объектов ядра, и значение параметра *nCount* не должно превышать 63 (*MAXIMUM_WAIT_OBJECTS - 1*);
- если в параметре *fWaitAll* передается FALSE, функции возвращают управление при освобождении объекта ядра *или* при появлении в очереди потока сообщения заданного типа;
- если в параметре *fWaitAll* передается TRUE, функции возвращают управление при освобождении всех объектов ядра *и* появлении в очереди потока сообще-

ния заданного типа. Такое поведение этих функций преподносит сюрприз многим разработчикам. Ведь очень часто поток надо пробуждать при освобождении всех объектов ядра *или* при появлении сообщения указанного типа. Но функции, действующей именно так, нет;

- при вызове любая из этих функций на самом деле проверяет в очереди потока только *новые* сообщения заданного типа.

Заметьте, что и последняя особенность этих функций — не очень приятный сюрприз для многих разработчиков. Возьмем простой пример. Допустим, в очереди потока находятся два сообщения о нажатии клавиш. Если теперь вызвать *MsgWaitForMultipleObjects(Ex)* и задать в *dwWakeMask* значение *QS_INPUT*, поток пробудится, извлечет из очереди первое сообщение и обработает его. Но на повторный вызов *MsgWaitForMultipleObjects(Ex)* поток никак не отреагирует — ведь *новых* сообщений в очереди нет.

Этот механизм создал столько проблем разработчикам, что Microsoft пришлось добавить в *MsgWaitForMultipleObjectsEx* поддержку флага *MWMO_INPUTAVAILABLE*.

Вот как надо писать цикл выборки сообщений при использовании *MsgWaitForMultipleObjectsEx*:

```

BOOL fQuit = FALSE; // надо ли завершить цикл?

while (!fQuit) {

    // поток пробуждается при освобождении объекта ядра ИЛИ
    // для обработки сообщения от пользовательского интерфейса
    DWORD dwResult = MsgWaitForMultipleObjectsEx(1, &hEvent,
        INFINITE, QS_ALLEVENTS, MWMO_INPUTAVAILABLE);

    switch (dwResult) {
        case WAIT_OBJECT_0:           // освободилось событие
            break;

        case WAIT_OBJECT_0 + 1:       // в очереди появилось сообщение
            // разослать все сообщения
            MSG msg;

            while (PeekMessage(&msg, NULL, 0, 0, PM_REMOVE)) {
                if (msg.message == WM_QUIT) {
                    // сообщение WM_QUIT – выходим из цикла
                    fQuit = TRUE;
                } else {
                    // транслируем и пересылаем сообщение
                    TranslateMessage(&msg);
                    DispatchMessage(&msg);
                }
            }
            // наша очередь пуста
            break;
    }
}
// конец цикла while

```


Передача данных через сообщения

Здесь мы обсудим, как система обеспечивает передачу данных между процессами с помощью сообщений. В некоторых оконных сообщениях параметр *lParam* задает адрес блока памяти. Например, сообщение WM_SETTEXT использует *lParam* как указатель на строку (с нулевым символом в конце), содержащую новый текст для окна. Рассмотрим такой вызов:

```
SendMessage(FindWindow(NULL, "Calculator"), WM_SETTEXT,
    0, (LPARAM) "A Test Caption");
```

Вроде бы все достаточно безобидно: определяется описатель окна Calculator и делается попытка изменить его заголовок на «A Test Caption». Но приглядимся к тому, что тут происходит.

В *lParam* передается адрес строки (с новым заголовком), расположенной в адресном пространстве Вашего процесса. Получив это сообщение, оконная процедура программы Calculator берет *lParam* и пытается манипулировать чем-то, что, «по ее мнению», является указателем на строку с новым заголовком.

Но адрес в *lParam* указывает на строку в адресном пространстве Вашего процесса, а не программы Calculator. Вот Вам и долгожданная неприятность — нарушение доступа к памяти. Но если Вы все же выполните показанную ранее строку, все будет работать нормально. Что за наваждение?

А дело в том, что система отслеживает сообщения WM_SETTEXT и обрабатывает их не так, как большинство других сообщений. При вызове *SendMessage* внутренний код функции проверяет, не пытаетесь ли Вы послать сообщение WM_SETTEXT. Если это так, функция копирует строку из Вашего адресного пространства в проекцию файла и делает его доступным другому процессу. Затем сообщение посылается потоку другого процесса. Когда поток-приемник готов к обработке WM_SETTEXT, он определяет адрес общей проекции файла (содержащей копию строки) в адресном пространстве своего процесса. Параметру *lParam* присваивается значение именно этого адреса, и WM_SETTEXT направляется нужной оконной процедуре. После обработки этого сообщения, проекция файла уничтожается. Не слишком ли тут накручено, а?

К счастью, большинство сообщений не требует такой обработки — она осуществляется, только если сообщение посылается другому процессу. (Заметьте: описанная обработка выполняется и для любого сообщения, параметры *wParam* или *lParam* которого содержат указатель на какую-либо структуру данных.)

А вот другой случай, когда от системы требуется особая обработка, — сообщение WM_GETTEXT. Допустим, Ваша программа содержит код:

```
char szBuf[200];
SendMessage(FindWindow(NULL, "Calculator"), WM_GETTEXT,
    Sizeof(szBuf), (LPARAM) szBuf);
```

WM_GETTEXT требует, чтобы оконная процедура программы Calculator поместила в буфер, на который указывает *szBuf*, заголовок своего окна. Когда Вы посылаете это сообщение окну другого процесса, система должна на самом деле послать два сообщения. Сначала — WM_GETTEXTLENGTH. Оконная процедура возвращает число символов в строке заголовка окна. Это значение система использует при создании проекции файла, разделяемой двумя процессами.

Создав проекцию файла, система посылает для его заполнения сообщение WM_GETTEXT. Затем переключается обратно на процесс, первым вызвавший функцию *SendMessage*.

sage, копирует данные из общей проекции файла в буфер, на который указывает *szBuf*, и заставляет *SendMessage* вернуть управление.

Что ж, все хорошо, пока Вы посылаете сообщения, известные системе. А если мы определим собственное сообщение (*WM_USER + x*), собираясь отправить его окну другого процесса? Система не «поймет», что нам нужна общая проекция файла для корректировки указателей при их пересылке. Но выход есть — это сообщение *WM_COPYDATA*:

```
COPYDATASTRUCT cds;
```

```
SendMessage(hwndReceiver, WM_COPYDATA, (LPARAM) hwndSender, (LPARAM) &cds);
```

COPYDATASTRUCT — структура, определенная в *WinUser.h*:

```
typedef struct tagCOPYDATASTRUCT {
    ULONG_PTR dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT;
```

Чтобы переслать данные окну другого процесса, нужно сначала инициализировать эту структуру. Элемент *dwData* резервируется для использования в Вашей программе. В него разрешается записывать любое значение. Например, передавая в другой процесс данные, в этом элементе можно указывать тип данных.

Элемент *cbData* задает число байтов, пересылаемых в другой процесс, а *lpData* указывает на первый байт данных. Адрес, идентифицируемый элементом *lpData*, находится, конечно же, в адресном пространстве отправителя.

Увидев, что Вы посылаете сообщение *WM_COPYDATA*, *SendMessage* создает проекцию файла размером *cbData* байтов и копирует данные из адресного пространства Вашей программы в эту проекцию. Затем отправляет сообщение окну-приемнику. При обработке этого сообщения принимающей оконной процедурой параметр *lParam* указывает на структуру *COPYDATASTRUCT*, которая находится в адресном пространстве процесса-приемника. Элемент *lpData* этой структуры указывает на проекцию файла в адресном пространстве процесса-приемника.

Вам следует помнить о трех важных вещах, связанных с сообщением *WM_COPYDATA*.

- Отправляйте его всегда синхронно; никогда не пытайтесь делать этого асинхронно. Последнее просто невозможно: как только принимающая оконная процедура обработает сообщение, система должна освободить проекцию файла. При передаче *WM_COPYDATA* как асинхронного сообщения появится неопределенность в том, когда оно будет обработано, и система не сможет освободить память, занятую проекцией файла.
- На создание копии данных в адресном пространстве другого процесса неизбежно уходит какое-то время. Значит, пока *SendMessage* не вернет управление, нельзя допускать изменения содержимого общей проекции файла каким-либо другим потоком.
- Сообщение *WM_COPYDATA* позволяет 16-разрядным приложениям взаимодействовать с 32-разрядными (и наоборот), как впрочем и 32-разрядным — с 64-разрядными (и наоборот). Это удивительно простой способ общения между новыми и старыми приложениями. К тому же, *WM_COPYDATA* полностью поддерживается как в Windows 2000, так и в Windows 98. Но, если Вы все еще пишете 16-разрядные Windows-приложения, учтите, что сообщение *WM_COPY-*

DATA и структура COPYDATASTRUCT в Microsoft Visual C++ версии 1.52 не определены. Вам придется добавить их определения самостоятельно:

```
// включите этот код в свою 16-разрядную Windows-программу
#define WM_COPYDATA 0x004A

typedef VOID FAR* PVOID;
typedef struct tagCOPYDATASTRUCT {
    DWORD dwData;
    DWORD cbData;
    PVOID lpData;
} COPYDATASTRUCT, FAR* PCOPYDATASTRUCT;
```

Сообщение WM_COPYDATA — мощный инструмент, позволяющий разработчикам экономить массу времени при решении проблем связи между процессами. И очень жаль, что применяется оно нечасто. Насколько полезно это сообщение, иллюстрирует программа-пример LastMsgBoxInfo из главы 22.

Программа-пример CopyData

Эта программа, «26 CopyData.exe» (см. листинг на рис. 26-3), демонстрирует применение сообщения WM_COPYDATA при пересылке блока данных из одной программы в другую. Файлы исходного кода и ресурсов этой программы находятся в каталоге 26-CopyData на компакт-диске, прилагаемом к книге. Чтобы увидеть программу CopyData в действии, запустите минимум две ее копии; при этом каждая копия открывает диалоговое окно, показанное ниже.



Если Вы хотите посмотреть, как данные копируются из одного приложения в другое, то сначала измените содержимое полей Data1 и Data2. Затем щелкните одну из двух кнопок Send Data* To Other Windows. Программа отправит данные всем выполняемым экземплярам CopyData, и в их полях появятся новые данные.

А теперь обсудим принцип работы программы. Щелчок одной из двух кнопок приводит к:

1. Инициализации элемента *dwData* структуры COPYDATASTRUCT нулевым значением (если выбрана кнопка Send Data1 To Other Windows) или единицей (если выбрана кнопка Send Data2 To Other Windows).
2. Подсчету длины текстовой строки (в символах) из соответствующего поля с добавлением единицы, чтобы учесть нулевой символ в конце. Полученное число символов преобразуется в количество байтов умножением на *sizeof(TCHAR)*, и результат записывается в элемент *cbData* структуры COPYDATASTRUCT.
3. Вызову *_alloca*, чтобы выделить блок памяти, достаточный для хранения строки с учетом конечного нулевого символа. Адрес этого блока записывается в элемент *lpData* все той же структуры.
4. Копированию текста из поля в выделенный блок памяти.

Теперь все готово для пересылки в другие окна. Чтобы определить, каким окнам следует посылать сообщение WM_COPYDATA, программа вызывает *FindWindowEx* и передает заголовок своего диалогового окна — благодаря этому перечисляются только другие экземпляры данной программы. Найдя окна всех экземпляров, программа пересылает им сообщение WM_COPYDATA, что заставляет их обновить содержимое своих полей.



CopyData.cpp

```

/*****
Модуль: CopyData.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"      /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <malloc.h>
#include "Resource.h"

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

// в WindowsX.h нет прототипа Cls_OnCopyData, поэтому определяем его здесь
/* BOOL Cls_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT pcds) */

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

BOOL Dlg_OnCopyData(HWND hwnd, HWND hwndFrom, PCOPYDATASTRUCT cds) {

    Edit_SetText(GetDlgItem(hwnd, cds->dwData ? IDC_DATA2 : IDC_DATA1),
        (PTSTR) cds->lpData);

    return(TRUE);
}

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

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

    chSETDLGICONS(hwnd, IDI_COPYDATA);

    // инициализируем поля тестовыми данными
    Edit_SetText(GetDlgItem(hwnd, IDC_DATA1), TEXT("Some test data"));
    Edit_SetText(GetDlgItem(hwnd, IDC_DATA2), TEXT("Some more test data"));
    return(TRUE);
}

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

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

```

Рис. 26-3. Программа-пример CopyData

см. след. стр.

Рис. 26-3. *продолжение*

```

switch (id) {
    case IDCANCEL:
        EndDialog(hwnd, id);
        break;

    case IDC_COPYDATA1:
    case IDC_COPYDATA2:
        if (codeNotify != BN_CLICKED)
            break;

        HWND hwndEdit = GetDlgItem(hwnd,
            (id == IDC_COPYDATA1) ? IDC_DATA1 : IDC_DATA2);

        // создаем экземпляр структуры COPYDATASTRUCT
        COPYDATASTRUCT cds;

        // указываем, из какого поля мы пересылаем данные (0=ID_DATA1, 1=ID_DATA2)
        cds.dwData = (DWORD) ((id == IDC_COPYDATA1) ? 0 : 1);

        // получаем длину пересылаемого блока данных (в байтах)
        cds.cbData = (Edit_GetTextLength(hwndEdit) + 1) * sizeof(TCHAR);

        // выделяем блок памяти для хранения строки
        cds.lpData = _alloca(cds.cbData);

        // копируем текст из поля ввода в выделенный блок
        Edit_GetText(hwndEdit, (PTSTR) cds.lpData, cds.cbData);

        // получаем заголовок нашего окна
        TCHAR szCaption[100];
        GetWindowText(hwnd, szCaption, chDIMOF(szCaption));

        // перечисляем все окна верхнего уровня с тем же заголовком
        HWND hwndT = NULL;
        do {
            hwndT = FindWindowEx(NULL, hwndT, NULL, szCaption);
            if (hwndT != NULL) {
                FORWARD_WM_COPYDATA(hwndT, hwnd, &cds, SendMessage);
            }
        } while (hwndT != NULL);
        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);
        chHANDLE_DLGMSG(hwnd, WM_COPYDATA, Dlg_OnCopyData);
    }
}

```

Рис. 26-3. продолжение

```

    }
    return(FALSE);
}
/////////////////////////////////////////////////////////////////

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_COPYDATA), NULL, Dlg_Proc);
    return(0);
}
///////////////////////////////////////////////////////////////// Конец файла ///////////////////////////////////////////////////////////////////

```

Как Windows манипулирует с ANSI/Unicode-символами и строками

WINDOWS 98 Windows 98 поддерживает классы и процедуры окон только в формате ANSI.

Регистрируя новый класс окна, Вы должны сообщить системе адрес оконной процедуры, которая отвечает за обработку сообщений для этого класса. В некоторых сообщениях (например, WM_SETTEXT) параметр *lParam* является указателем на строку. Для корректной обработки сообщения система должна заранее знать, в каком формате оконная процедура принимает строки — ANSI или Unicode.

Выбирая конкретную функцию для регистрации класса окна, Вы сообщаете системе формат, приемлемый для Вашей оконной процедуры. Если Вы создаете структуру WNDCLASS и вызываете *RegisterClassA*, система считает, что процедура ожидает исключительно ANSI-строки и символы. А регистрация класса окна через *RegisterClassW* заставит систему полагать, что процедуре нужен Unicode. И, конечно же, в зависимости от того, определен ли UNICODE при компиляции модуля исходного кода, макрос *RegisterClass* будет раскрыт либо в *RegisterClassA*, либо в *RegisterClassW*.

Располагая описателем окна, Вы можете выяснить, какой формат символов и строк требует оконная процедура. Для этого вызовите функцию:

```
BOOL IsWindowUnicode(HWND hwnd);
```

Если оконная процедура ожидает передачи данных только в Unicode, эта функция возвращает TRUE; в ином случае — FALSE.

Если Вы сформировали ANSI-строку и посылаете сообщение WM_SETTEXT окну, чья процедура принимает только Unicode-строки, то система перед отсылкой сообщения автоматически преобразует его в нужный формат. Так что необходимость в вызове *IsWindowUnicode* возникает нечасто.

Система автоматически выполняет все преобразования и при создании подкласса окна. Допустим, что для заполнения своего поля ввода оконная процедура ожидает передачи символов и строк в Unicode. Кроме того, где-то в программе Вы создаете поле ввода и подкласс оконной процедуры, вызывая:

```

LONG_PTR SetWindowLongPtrA(
    HWND hwnd,
    int nIndex,
    LONG_PTR dwNewLong);

```

или

```
LONG_PTR SetWindowLongPtrW(
    HWND hwnd,
    int nIndex,
    LONG_PTR dwNewLong);
```

При этом Вы передаете в параметре *nIndex* значение `GCLP_WNDPROC`, а в параметре *dwNewLong* — адрес своей процедуры подкласса. Но что будет, если Ваша процедура ожидает передачи символов и строк в формате ANSI? В принципе, это чревато проблемами. Система определяет, как преобразовывать строки и символы в зависимости от функции, вызванной Вами для создания подкласса. Используя *SetWindowLongPtrA*, Вы сообщаете Windows, что новая оконная процедура (Вашего подкласса) принимает строки и символы только в ANSI. (Вызвав *IsWindowUnicode* после *SetWindowLongPtrA*, Вы получили бы FALSE, так как новая процедура не принимает строки и символы в Unicode.)

Но теперь у нас новая проблема: как сделать так, чтобы исходная процедура получила символы и строки в своем формате? Для корректного преобразования системе нужно знать две вещи. Во-первых, текущий формат символов и строк. Эту информацию мы предоставляем, вызывая одну из двух функций — *CallWindowProcA* или *CallWindowProcW*:

```
LRESULT CallWindowProcA(
    WNDPROC wndprcPrev,
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

```
LRESULT CallWindowProcW(
    WNDPROC wndprcPrev,
    HWND hwnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam);
```

При передаче исходной оконной процедуре ANSI-строк процедура подкласса должна вызывать *CallWindowProcA*, а при передаче Unicode-строк — *CallWindowProcW*.

Второе, о чем должна знать система, — тип символов и строк, ожидаемый исходной оконной процедурой. Система получает эту информацию по адресу этой процедуры. Когда Вы вызываете *SetWindowLongPtrA* или *SetWindowLongPtrW*, система проверяет, создаете ли Вы ANSI-подкласс Unicode-процедуры окна или наоборот. Если при создании подкласса тип строк не меняется, *SetWindowLongPtr* просто возвращает адрес исходной процедуры. В ином случае *SetWindowLongPtr* вместо этого адреса возвращает описатель внутренней структуры данных.

Эта структура содержит адрес исходной оконной процедуры и значение, которое указывает на ожидаемый ею формат строк. При вызове *CallWindowProc* система проверяет, что Вы передаете — адрес оконной процедуры или описатель внутренней структуры данных. В первом случае система сразу обращается к исходной оконной процедуре, так как никаких преобразований не требуется, а во втором случае система сначала преобразует символы и строки в соответствующую кодировку и только потом вызывает исходную оконную процедуру.

Модель аппаратного ввода и локальное состояние ввода

В этой главе мы рассмотрим модель аппаратного ввода. В частности, я расскажу, как события от клавиатуры и мыши попадают в систему и пересылаются соответствующим оконным процедурам. Создавая модель ввода, Microsoft стремилась главным образом к тому, чтобы ни один поток не мог нарушить работу других потоков. Вот пример из 16-разрядной Windows: задача (так в этой системе назывались выполняемые программы), зависшая в бесконечном цикле, приводила к тому, что зависали и остальные задачи — их дальнейшее выполнение становилось невозможным. Пользователю ничего не оставалось, как только перезагрузить компьютер. А все потому, что операционная система слишком много разрешала отдельно взятой задаче. Отказоустойчивые операционные системы вроде Windows 2000 и Windows 98 не дают зависшему потоку блокировать другим потокам прием аппаратного ввода.

Поток необработанного ввода

Общая схема модели аппаратного ввода в системе показана на рис. 27-1. При запуске система создает себе особый *поток необработанного ввода* (raw input thread, RIT) и *системную очередь аппаратного ввода* (system hardware input queue, SHIQ). RIT и SHIQ — это фундамент, на котором построена вся модель аппаратного ввода.

Обычно RIT бездействует, ожидая появления какого-нибудь элемента в SHIQ. Когда пользователь нажимает и отпускает клавишу на клавиатуре или кнопку мыши, либо перемещает мышь, соответствующий драйвер устройства добавляет аппаратное событие в SHIQ. Тогда RIT пробуждается, извлекает этот элемент из SHIQ, преобразует его в сообщение (WM_KEY*, WM_?BUTTON* или WM_MOUSEMOVE) и ставит в конец очереди виртуального ввода (virtualized input queue, VIQ) нужного потока. Далее RIT возвращается в начало цикла и ждет появления следующего элемента в SHIQ. RIT никогда не перестает реагировать на события аппаратного ввода — весь его код написан самой Microsoft и очень тщательно протестирован.

Как же RIT узнает, в чью очередь надо пересылать сообщения аппаратного ввода? Ну, с сообщениями от мыши все ясно: RIT просто выясняет, в каком окне находится ее курсор, и, вызвав *GetWindowThreadProcessId*, определяет поток, создавший это окно. Поток с данным идентификатором и получит сообщение от мыши.

В случае сообщений от клавиатуры все происходит несколько иначе. В любой момент с RIT «связан» лишь какой-то один поток, называемый *активным* (foreground thread). Именно ему принадлежит окно, с которым работает пользователь в данное время.

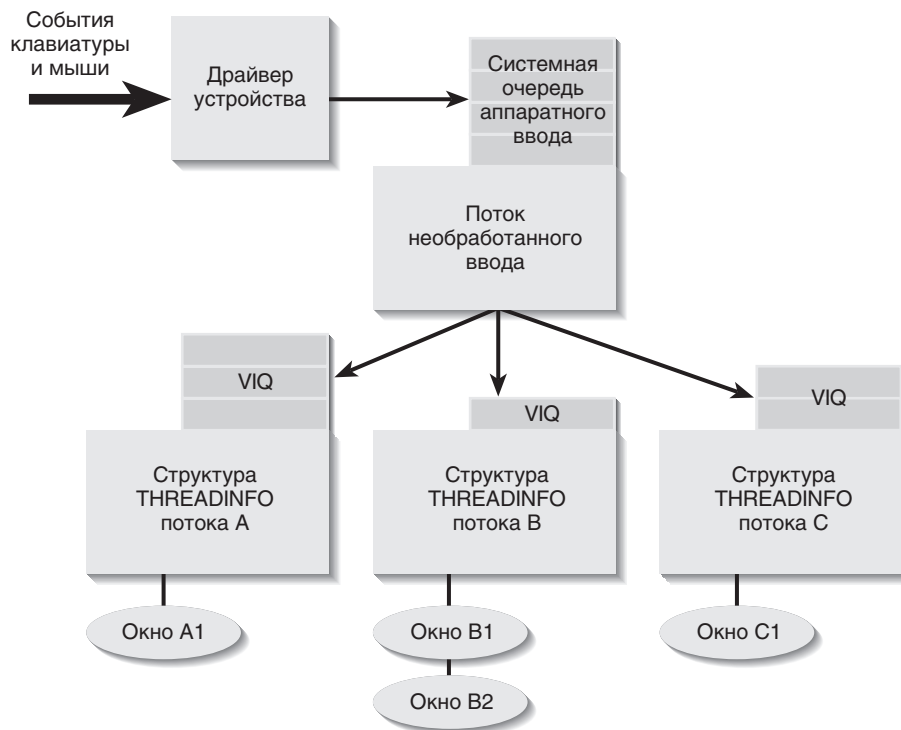


Рис. 27-1. Модель аппаратного ввода

Когда пользователь входит в систему, процесс Windows Explorer порождает поток, который создает панель задач и рабочий стол. Этот поток привязывается к RIT. Если Вы запустите Calculator, то его поток, создавший окно, немедленно подключится к RIT. После этого поток, принадлежащий Explorer, отключается от RIT, так как одновременно с RIT может быть связан только один поток. При нажатии клавиши в SHIQ появится соответствующий элемент. Это приведет к тому, что RIT пробудится, преобразует событие аппаратного ввода в сообщение от клавиатуры и поместит его в VIQ потока Calculator.

Каким образом различные потоки подключаются к RIT? Если при создании процесса его поток создает окно, последнее автоматически появляется на переднем плане (становится активным), и этот поток присоединяется к RIT. Кроме того, RIT отвечает за обработку особых комбинаций клавиш: Alt+Tab, Alt+Esc и Ctrl+Alt+Del. Поскольку эти комбинации клавиш RIT обрабатывает самостоятельно, пользователи могут в любой момент активизировать соответствующие окна с клавиатуры; ни одно приложение не в состоянии перехватить упомянутые комбинации клавиш. Как только пользователь нажимает одну из таких комбинаций клавиш, RIT активизирует выбранное окно, и в результате его поток подключается к RIT. Кстати, в Windows есть функции, позволяющие программно активизировать окно, присоединив его поток к RIT. Мы обсудим их несколько позже.

На рис. 27-1 видно, как работает механизм защиты потоков друг от друга. Посылая сообщение в окно B1 или B2, RIT помещает его в очередь виртуального ввода потока B. Обработывая это сообщение, поток — при синхронизации на каком-либо объекте ядра — может войти в бесконечный цикл или попасть в ситуацию взаимной блокировки. Если так и случится, он все равно останется присоединенным к RIT, и сообщения будут поступать именно в его очередь виртуального ввода.

Однако пользователь, заметив, что ни окно В1, ни окно В2 не реагируют на его действия, может переключиться, например, в окно А1 нажатием клавиш Alt+Tab. Поскольку RIT сам обрабатывает комбинацию клавиш Alt+Tab, переключение пройдет без всяких проблем. После активизации окна А1 к RIT будет подключен поток А. Теперь пользователь может спокойно работать с окном А1, даже несмотря на то что поток В и оба его окна зависли.

Локальное состояние ввода

Независимая обработка ввода потоками, предотвращающая неблагоприятное воздействие одного потока на другой, — это лишь часть того, что обеспечивает отказоустойчивость модели аппаратного ввода. Но этого недостаточно для надежной изоляции потоков друг от друга, и поэтому система поддерживает дополнительную концепцию — *локальное состояние ввода* (local input state).

Каждый поток обладает собственным состоянием ввода, сведения о котором хранятся в структуре THREADINFO (глава 26). В информацию об этом состоянии включаются данные об очереди виртуального ввода потока и группа переменных. Последние содержат управляющую информацию о состоянии ввода.

Для клавиатуры поддерживаются следующие сведения:

- какое окно находится в фокусе клавиатуры;
- какое окно активно в данный момент;
- какие клавиши нажаты;
- состояние курсора ввода.

Для мыши учитывается такая информация:

- каким окном захвачена мышь;
- какова форма курсора мыши;
- видим ли этот курсор.

Так как у каждого потока свой набор переменных состояния ввода, то и представления об окне, находящемся в фокусе, об окне, захватившем мышь, и т. п. у них тоже сугубо свои. С точки зрения потока, клавиатурный фокус либо есть у одного из его окон, либо его нет ни у одного окна во всей системе. То же самое относится и к мыши: либо она захвачена одним из его окон, либо не захвачена никем. В общем, перечислять можно еще долго. Так вот, подобный сепаратизм приводит к некоторым последствиям — о них мы и поговорим.

Ввод с клавиатуры и фокус

Как Вы уже знаете, ввод с клавиатуры направляется потоком необработанного ввода (RIT) в очередь виртуального ввода какого-либо потока, но только не в окно. RIT помещает события от клавиатуры в очередь потока безотносительно конкретному окну. Когда поток вызывает *GetMessage*, событие от клавиатуры извлекается из очереди и перенаправляется окну (созданному потоком), на котором в данный момент сосредоточен фокус ввода (рис. 27-2).

Чтобы направить клавиатурный ввод в другое окно, нужно указать, в очередь какого потока RIT должен помещать события от клавиатуры, а также «сообщить» переменным состояния ввода потока, какое окно будет находиться в фокусе. Одним вызовом *SetFocus* эти задачи не решить. Если в данный момент ввод от RIT получает поток 1, то вызов *SetFocus* с передачей описателей окон А, В или С приведет к смене фокуса.

Окно, теряющее фокус, убирает используемый для обозначения фокуса прямоугольник или гасит курсор ввода, а окно, получающее фокус, рисует такой прямоугольник или показывает курсор ввода.

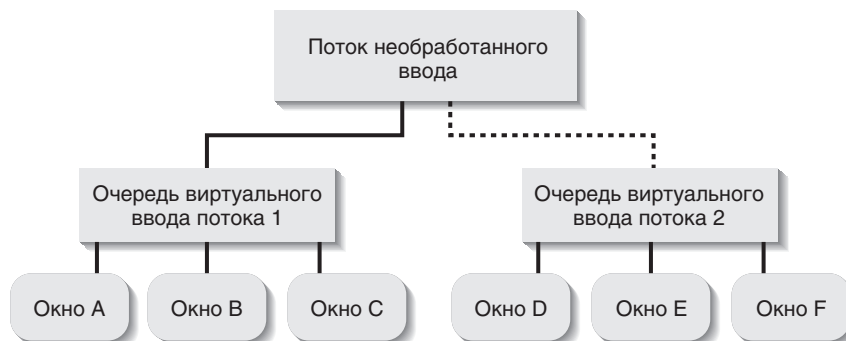


Рис. 27-2. RIT направляет пользовательский ввод с клавиатуры в очередь виртуального ввода только одного из потоков одновременно

Предположим, однако, что поток 1 по-прежнему получает ввод от RIT и вызывает *SetFocus*, передавая ей описатель окна E. В этом случае система не дает функции что-либо сделать, так как окно, на которое Вы хотите перевести фокус, не использует очередь виртуального ввода, подключенную в данный момент к RIT. Когда поток 1 выполнит этот вызов, на экране не произойдет ни смены фокуса, ни каких-либо изменений.

Возьмем другую ситуацию: поток 1 подключен к RIT, а поток 2 вызывает *SetFocus*, передавая ей описатель окна E. На этот раз значения переменных локального состояния ввода потока 2 изменяются так, что — когда RIT в следующий раз направит события от клавиатуры этому потоку — ввод с клавиатуры получит окно E. Этот вызов не заставит RIT направить клавиатурный ввод в очередь виртуального ввода потока 2.

Так как фокус теперь сосредоточен на окне E потока 2, оно получает сообщение WM_SETFOCUS. Если окно E — кнопка, на нем появляется прямоугольник, обозначающий фокус, и в результате на экране могут появиться два окна с такими прямоугольниками (окна A и E). Сами понимаете, это вряд ли кому понравится. Поэтому вызывать *SetFocus* следует с большой осторожностью — чтобы не создавать подобных ситуаций. Вызов *SetFocus* безопасен, только если Ваш поток подключен к RIT.

Кстати, если Вы переведете фокус на окно, которое, получив сообщение WM_SETFOCUS, показывает курсор ввода, не исключено одновременное появление на экране нескольких окон с таким курсором. Это тоже вряд ли кого обрадует.

Когда фокус переводится с одного окна на другое обычным способом (например, щелчком окна), теряющее фокус окно получает сообщение WM_KILLFOCUS. Если окно, получающее фокус, принадлежит другому потоку, переменные локального состояния ввода потока, который владеет окном, теряющим фокус, обновляются так, чтобы показать: окон в фокусе нет. И вызов *GetFocus* возвращает при этом NULL, заставляя поток считать, что окон в фокусе нет.

Функция *SetActiveWindow* активизирует в системе окно верхнего уровня и переводит на него фокус:

```
HWND SetActiveWindow(HWND hwnd);
```

Как и *SetFocus*, эта функция ничего не делает, если поток вызывает ее с описателем окна, созданного другим потоком.

Функцию *SetActiveWindow* дополняет *GetActiveWindow*:

```
HWND GetActiveWindow();
```

Она работает так же, как и *GetFocus*, но возвращает описатель активного окна, указанного в переменных локального состояния ввода вызывающего потока. Так что, если активное окно принадлежит другому потоку, функция возвращает NULL.

Есть и другие функции, влияющие на порядок размещения окон, их статус (активно или неактивно) и фокус:

```
BOOL BringWindowToTop(HWND hwnd);
```

```
BOOL SetWindowPos(
    HWND hwnd,
    HWND hwndInsertAfter,
    int x,
    int y,
    int cx,
    int cy,
    UINT fuFlags);
```

Обе эти функции работают одинаково (фактически *BringWindowToTop* вызывает *SetWindowPos*, передавая ей *HWND_TOP* во втором параметре). Когда поток, вызывающий любую из этих функций, не связан с RIT, они ничего не делают. В ином случае (когда поток связан с RIT) система активизирует указанное окно. Обратите внимание, что здесь не имеет значения, принадлежит ли это окно вызвавшему потоку. Окно становится активным, а к RIT подключается тот поток, который создал данное окно. Кроме того, значения переменных локального состояния ввода обоих потоков обновляются так, чтобы отразить эти изменения.

Иногда потоку нужно вывести свое окно на передний план. Например, Вы запланировали какую-то встречу, используя Microsoft Outlook. Где-то за полчаса до назначенного времени Outlook выводит на экран диалоговое окно с напоминанием о встрече. Если поток Outlook не связан с RIT, это диалоговое окно появится под другими окнами, и Вы его не увидите. Поэтому нужен какой-то способ, который позволил бы привлекать внимание к определенному окну, даже если в данный момент пользователь работает с окном другого приложения.

Вот функция, которая выводит окно на передний план и подключает его поток к RIT:

```
BOOL SetForegroundWindow(HWND hwnd);
```

Одновременно система активизирует окно и переводит на него фокус. Функция, парная *SetForegroundWindow*:

```
HWND GetForegroundWindow();
```

Она возвращает описатель окна, находящегося сейчас на переднем плане.

В более ранних версиях Windows функция *SetForegroundWindow* срабатывала всегда. То есть поток, вызвавший ее, всегда мог перевести указанное окно на передний план (даже если оно было создано другим потоком). Однако разработчики стали злоупотреблять этой функцией и нагромождать окна друг на друга. Представьте, я пишу журнальную статью, и вдруг выскакивает окно с сообщением о завершении печати. Если бы я не смотрел на экран, то начал бы вводить текст не в свой документ, а в это окно. Еще больше раздражает, когда пытаешься выбрать команду в меню, а на экране появляется какое-то окно и закрывает меню.

Чтобы прекратить всю эту неразбериху, Microsoft сделала *SetForegroundWindow* чуть поумнее. В частности, эта функция срабатывает, только если вызывающий поток уже подключен к RIT или если поток, связанный с RIT в данный момент, не получал ввода на протяжении определенного периода (который задается функцией *SystemParametersInfo* и значением SPI_SETFOREGROUNDLOCKTIMEOUT). Кроме того, *SetForegroundWindow* терпит неудачу, когда активно какое-нибудь меню.

Если *SetForegroundWindow* не удастся переместить окно на передний план, то его кнопка на панели задач начинает мигать. Заметив это, пользователь будет в курсе, что окно требует его внимания. Чтобы выяснить, в чем дело, пользователю придется активизировать это окно вручную. Управлять режимом мигания окна позволяет функция *SystemParametersInfo* со значением SPI_SETFOREGROUNDFLASHCOUNT.

Из-за такого поведения *SetForegroundWindow* в систему встроено несколько новых функций. Первая из них, *AllowSetForegroundWindow*, разрешает потоку указанного процесса успешно вызвать *SetForegroundWindow*, но только если и вызывающий ее поток может успешно вызвать *SetForegroundWindow*. Чтобы любой процесс мог вывести окно «поверх» остальных окон, открытых Вашим потоком, передайте в параметре *dwProcessId* значение ASFW_ANY (определенное как -1):

```
BOOL AllowSetForegroundWindow(DWORD dwProcessId);
```

Кроме того, можно полностью заблокировать работу *SetForegroundWindow*, вызвав *LockSetForegroundWindow*:

```
BOOL LockSetForegroundWindow(UINT uLockCode);
```

В параметре *uLockCode* она принимает либо LSFW_LOCK, либо LSFW_UNLOCK. Данная функция вызывается системой, когда на экране активно какое-нибудь системное меню, — чтобы никакое окно не могло его закрыть. (Поскольку меню Start не является встроенным, то при его открытии Windows Explorer сам вызывает эти функции.)

Система автоматически снимает блокировку с функции *SetForegroundWindow*, когда пользователь нажимает клавишу Alt или активизирует какое-либо окно. Так что приложение не может навечно заблокировать *SetForegroundWindow*.

Другой аспект управления клавиатурой и локальным состоянием ввода связан с массивом синхронного состояния клавиш (synchronous key state array). Этот массив включается в переменные локального состояния ввода каждого потока. В то же время массив асинхронного состояния клавиш (asynchronous key state array) — только один, и он разделяется всеми потоками. Эти массивы отражают состояние всех клавиш на данный момент, и функция *GetAsyncKeyState* позволяет определить, нажата ли сейчас заданная клавиша:

```
SHORT GetAsyncKeyState(int nVirtKey);
```

Параметр *nVirtKey* задает код виртуальной клавиши, состояние которой нужно проверить. Старший бит результата определяет, нажата в данный момент клавиша (1) или нет (0). Я часто пользовался этой функцией, определяя при обработке сообщения, отпустил ли пользователь основную (обычно левую) кнопку мыши. Передав значение VK_LBUTTON, я ждал, когда обнулится старший бит. Заметьте, что *GetAsyncKeyState* всегда возвращает 0 (не нажата), если ее вызывает другой поток, а не тот, который создал окно, находящееся сейчас в фокусе ввода.

Функция *GetKeyState* отличается от *GetAsyncKeyState* тем, что возвращает состояние клавиатуры на момент, когда из очереди потока извлечено последнее сообщение от клавиатуры:

```
SHORT GetKeyState(int nVirtKey);
```

Эту функцию можно вызвать в любой момент; для нее неважно, какое окно в фокусе.

Управление курсором мыши

В концепцию локального состояния ввода входит и управление состоянием курсора мыши. Поскольку мышь, как и клавиатура, должна быть доступна всем потокам, Windows не позволяет какому-то одному потоку монопольно распоряжаться курсором мыши, изменяя его форму или ограничивая область его перемещения. Посмотрим, как система управляет этим курсором.

Один из аспектов управления курсором мыши заключается в его отображении или гашении. Если поток вызывает *ShowCursor(FALSE)*, то система скрывает курсор, когда он оказывается на любом окне, созданном этим потоком, и показывает курсор всякий раз, когда он попадает в окно, созданное другим потоком.

Другой аспект управления курсором мыши — возможность ограничить его перемещение каким-либо прямоугольным участком. Для этого надо вызвать функцию *ClipCursor*:

```
BOOL ClipCursor(CONST RECT *prc);
```

Она ограничивает перемещение курсора мыши прямоугольником, на который указывает параметр *prc*. И опять система разрешает потоку ограничить перемещение курсора заданным прямоугольником. Но, когда возникает событие асинхронной активизации, т. е. когда пользователь переключается в окно другого приложения, нажимает клавиши Ctrl+Esc или же поток вызывает *SetForegroundWindow*, система снимает ограничения на передвижение курсора, позволяя свободно перемещать его по экрану.

И здесь мы подошли к концепции *захвата мыши* (mouse capture). «Захватывающая» мышь (вызовом *SetCapture*), окно требует, чтобы все связанные с мышью сообщения RIT отправлял в очередь виртуального ввода вызывающего потока, а из нее — установившему захват окну до тех пор, пока программа не вызовет *ReleaseCapture*.

Как и в предыдущих случаях, это тоже снижает отказоустойчивость системы, но без компромиссов, увы, не обойтись. Вызывая *SetCapture*, поток заставляет RIT помещать все сообщения от мыши в свою очередь виртуального ввода. При этом *SetCapture* соответственно настраивает переменные локального состояния ввода данного потока.

Обычно приложение вызывает *SetCapture*, когда пользователь нажимает кнопку мыши. Но поток может вызвать эту функцию, даже если нажатия кнопки мыши не было. Если *SetCapture* вызывается при нажатой кнопке, захват действует для всей системы. Как только система определяет, что все кнопки мыши отпущены, RIT перестает направлять сообщения от мыши исключительно в очередь виртуального ввода данного потока. Вместо этого он передает сообщения в очередь ввода, связанную с окном, «поверх» которого курсор находится в данный момент. И это нормальное поведение системы, когда захват мыши не установлен.

Однако для вызвавшего *SetCapture* потока ничего не меняется. Всякий раз, когда курсор оказывается на любом из окон, созданных установившим захват потоком, сообщения от мыши направляются в окно, применительно к которому этот захват и установлен. Иначе говоря, когда пользователь отпускает все кнопки мыши, захват осуществляется на уровне лишь данного потока, а не всей системы.

Если пользователь попытается активизировать окно, созданное другим потоком, система автоматически отправит установившему захват потоку сообщения о нажатии и отжатии кнопок мыши. Затем она изменит переменные локального состояния ввода потока, чтобы отразить тот факт, что поток более не работает в режиме захвата. Словом, Microsoft считает, что захват мыши чаще всего применяется для выполнения таких операций, как щелчок и перетаскивание экранного объекта.

Последняя переменная локального состояния ввода, связанная с мышью, относится к форме курсора. Всякий раз, когда поток вызывает *SetCursor* для изменения формы курсора, переменные локального состояния ввода соответствующим образом обновляются. То есть переменные локального состояния ввода всегда запоминают последнюю форму курсора, установленную потоком.

Допустим, пользователь перемещает курсор мыши на окно Вашей программы, окно получает сообщение WM_SETCURSOR, и Вы вызываете *SetCursor*, чтобы преобразовать курсор в «песочные часы». Вызвав *SetCursor*, программа начинает выполнять какую-то длительную операцию. (Бесконечный цикл — лучший пример длительной операции. Шутка.) Далее пользователь перемещает курсор из окна Вашей программы в окно другого приложения, и это окно может изменить форму курсора.

Для такого изменения переменные локального состояния ввода не нужны. Но переведем курсор обратно в то окно, поток которого по-прежнему занят обработкой. Системе «хочется» послать окну сообщения WM_SETCURSOR, но процедура этого окна не может выбрать их из очереди, так как его поток продолжает свою операцию. Тогда система определяет, какая форма была у курсора в прошлый раз (информация об этом содержится в переменных локального состояния ввода данного потока), и автоматически восстанавливает ее (в нашем примере — «песочные часы»). Теперь пользователю четко видно, что в этом окне работа еще не закончена и придется подождать.

Подключение к очередям виртуального ввода и переменным локального состояния ввода

Как Вы уже убедились, отказоустойчивость модели ввода достигается благодаря тому, что у каждого потока имеются собственные переменные локального состояния ввода, а подключение потока к RIT и отключение от него происходит по мере необходимости. Иногда нужно, чтобы два потока (или более) разделяли один набор переменных локального состояния ввода или одну очередь виртуального ввода.

Вы можете заставить два и более потока совместно использовать одну и ту же очередь виртуального ввода и переменные локального состояния ввода с помощью функции *AttachThreadInput*:

```
BOOL AttachThreadInput(
    DWORD idAttach,
    DWORD idAttachTo,
    BOOL fAttach);
```

Параметр *idAttach* задает идентификатор потока, чьи переменные локального состояния ввода и очередь виртуального ввода Вам больше не нужны, а параметр *idAttachTo* — идентификатор потока, чьи переменные локального состояния ввода и виртуальная очередь ввода должны совместно использоваться потоками. И, наконец, параметр *fAttach* должен быть или TRUE, чтобы инициировать совместное использование одной очереди, или FALSE — тогда каждый поток будет вновь использовать свои переменные состояния ввода и очередь. А чтобы одну очередь (и переменные состояния ввода) разделяли более двух потоков, вызовите *AttachThreadInput* соответствующее число раз.

Вернемся к одному из предыдущих примеров и допустим, что поток А вызывает *AttachThreadInput*, передавая в первом параметре свой идентификатор, во втором — идентификатор потока В и в последнем — TRUE:

```
AttachThreadInput(idThreadA, idThreadB, TRUE);
```

Теперь любое событие аппаратного ввода, адресованное окну A1, B1 или B2, будет добавлено в конец очереди виртуального ввода потока B. Аналогичная очередь потока A больше не получит новых событий, если только Вы не разъедините очереди, повторно вызвав *AttachThreadInput* с передачей FALSE в параметре *fAttach*.

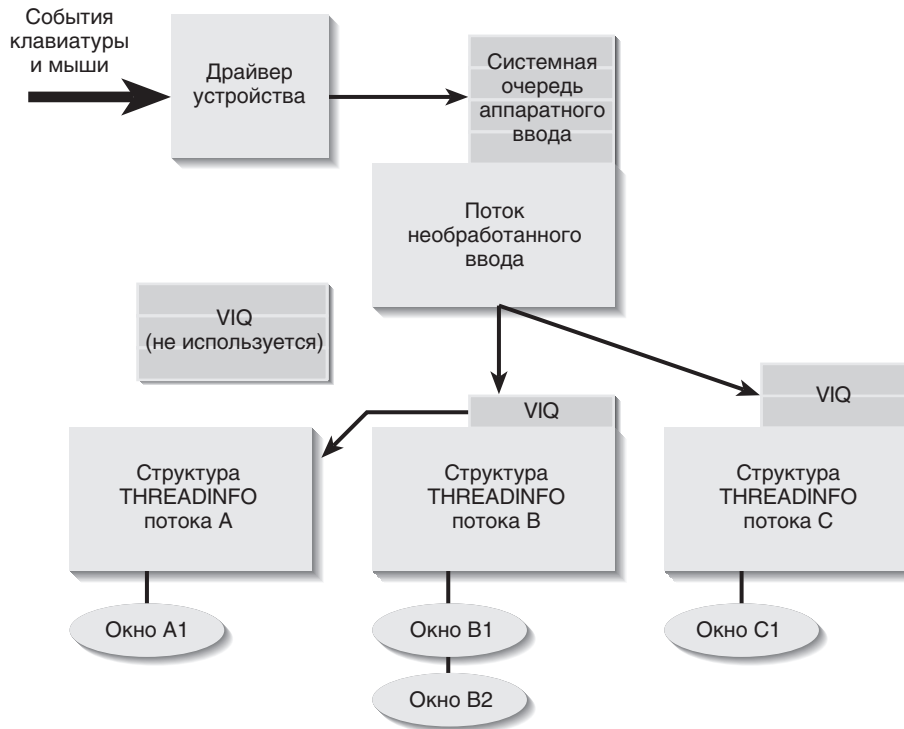


Рис. 27-3. Аппаратные сообщения для окон A1, B1 и B2 помещаются в очередь виртуального ввода потока B

Потоки, присоединенные к одной очереди виртуального ввода (VIQ), сохраняют индивидуальные очереди сообщений (синхронных, асинхронных и ответных), а также флаги пробуждения. Однако Вы серьезно снизите надежность системы, если заставите все потоки использовать одну очередь сообщений. Если какой-нибудь поток зависнет при обработке нажатия клавиши, другие потоки не получат никакого ввода. Поэтому использования *AttachThreadInput* следует по возможности избегать.

Система неявно соединяет очереди виртуального ввода двух потоков, если какой-то из них устанавливает ловушку регистрации (journal record hook) или ловушку воспроизведения (journal playback hook). Когда ловушка снимается, система восстанавливает схему организации очереди ввода, существовавшую до установки ловушки.

Установкой ловушки регистрации поток сообщает, что хочет получать уведомления о всех аппаратных событиях, вызываемых пользователем. Поток обычно сохраняет или регистрирует эту информацию в файле. Так как пользовательский ввод должен быть зарегистрирован в том порядке, в каком он происходил, все потоки в системе начинают разделять одну очередь виртуального ввода для синхронизации обработки ввода.

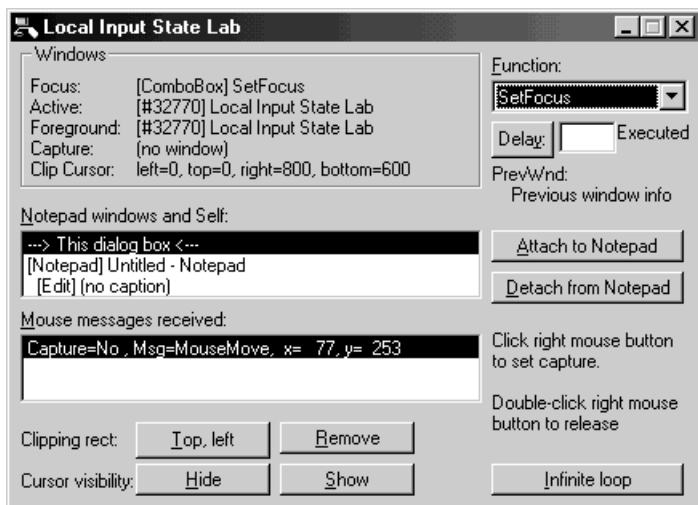
Есть еще один случай, когда система неявно вызывает *AttachThreadInput*. Допустим, приложение создает два потока. Первый открывает на экране диалоговое окно. Затем второй поток вызывает *CreateWindow*, указывая стиль *WS_CHILD* и передавая описа-

тель этого диалогового окна, чтобы оно стало «родителем» дочернего окна. Тогда система сама вызывает *AttachThreadInput*, чтобы поток (которому принадлежит дочернее окно) использовал ту же очередь ввода, что и поток, создавший исходное диалоговое окно. Это приводит к синхронизации ввода во всех дочерних окнах исходного диалогового окна.

Программа-пример LISLab

Эта программа, «27 LISLab.exe» (см. листинг на рис. 27-4), — своего рода лаборатория, в которой Вы сможете поэкспериментировать с локальным состоянием ввода. Файлы исходного кода и ресурсов этой программы находятся в каталоге 27-LISLab на компакт-диске, прилагаемом к книге.

В качестве подопытных кроликов нам понадобятся два потока. Один поток есть в нашей LISLab, а вторым будет Notepad. Если на момент запуска LISLab программа Notepad не выполняется, LISLab сама запустит эту программу. После инициализации LISLab Вы увидите следующее диалоговое окно.



В левом верхнем углу окна — раздел Windows; его поля обновляются дважды в секунду, т. е. дважды в секунду диалоговое окно получает сообщение WM_TIMER и в ответ вызывает функции *GetFocus*, *GetActiveWindow*, *GetForegroundWindow*, *GetCapture* и *GetClipCursor*. Первые четыре функции возвращают описатели окна (считываемые из переменных локального состояния ввода моего потока), через которые я могу определить класс и заголовок окна и вывести эту информацию на экран.

Если я активизирую другое приложение (тот же Notepad), названия полей Focus и Active меняются на (No Window), а поля Foreground — на [Notepad] Untitled - Notepad. Обратите внимание, что активизация Notepad заставляет LISLab считать, что ни активных, ни находящихся в фокусе окон нет.

Теперь поэкспериментируем со сменой фокуса. Выберем SetFocus в списке Function — в правом верхнем углу диалогового окна. Затем в поле Delay введем время (в секундах), в течение которого LISLab будет ждать, прежде чем вызвать SetFocus. В данном случае, видимо, лучше установить нулевое время задержки. Позже я объясню, как используется поле Delay.

Выберем окно (описатель которого мы хотим передать функции SetFocus) в списке Notepad Windows And Self, расположенном в левой части диалогового окна. Для эксперимента укажем [Notepad] Untitled - Notepad. Теперь все готово к вызову SetFocus.

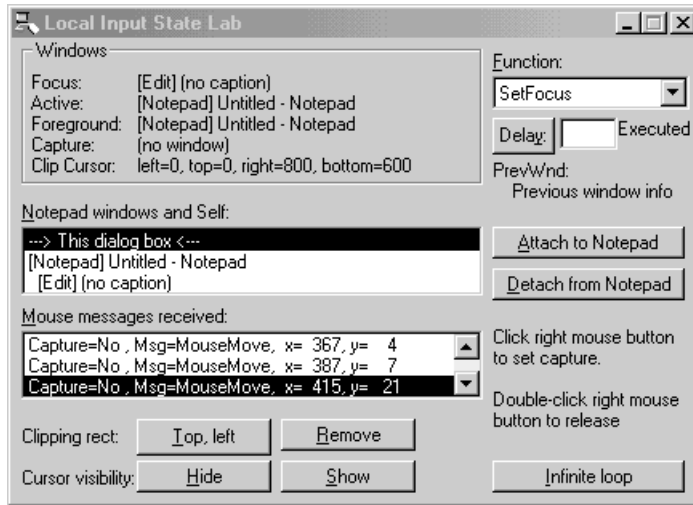
Щелкните кнопку Delay и наблюдайте, что произойдет в разделе Windows. Ничего. Система отказалась менять фокус.

Если Вы действительно хотите перевести фокус на Notepad, щелкните кнопку Attach To Notepad, что заставит LISLab вызвать:

```
AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),
    GetCurrentThreadId(), TRUE);
```

В результате этого вызова поток LISLab станет использовать ту же очередь виртуального ввода, что и Notepad. Кроме того, поток LISLab «разделит» переменные локального состояния ввода Notepad.

Если после щелчка кнопки Attach To Notepad Вы щелкнете окно Notepad, диалоговое окно LISLab примет следующий вид.



Теперь, когда очереди ввода соединены друг с другом, LISLab способна отслеживать изменения фокуса, происходящие в Notepad. В приведенном выше диалоговом окне показано, что в данный момент фокус установлен на поле ввода. А если мы откроем в Notepad диалоговое окно File Open, то LISLab, продолжая следить за Notepad, покажет нам, какое окно Notepad получило фокус ввода, какое окно активно и т. д.

Теперь можно вернуться в LISLab, щелкнуть кнопку Delay и вновь попытаться заставить *SetFocus* перевести фокус на Notepad. На этот раз все пройдет успешно, потому что очереди ввода соединены.

Если хотите, поэкспериментируйте с *SetActiveWindow*, *SetForegroundWindow*, *BringWindowToTop* и *SetWindowPos*, выбирая нужную функцию в списке Function. Попробуйте вызывать их и когда очереди соединены, и когда они разъединены; при этом обращайте внимание на различия в поведении функций.

А сейчас я поясню, зачем предусмотрена задержка. Она заставляет LISLab вызывать указанную функцию только по истечении заданного числа секунд. Для иллюстрации возьмем такой пример. Но прежде отключите LISLab от Notepad, щелкнув кнопку Detach From Notepad. Затем в списке Notepad Windows And Self выберите --->This Dialog Box<---, а в списке Function — SetFocus и установите задержку на 10 секунд. Наконец «нажмите» кнопку Delay и быстро щелкните окно Notepad, чтобы оно стало активным. Вы должны активизировать Notepad до того, как истекнут заданные 10 секунд.

Пока идет отсчет времени задержки, справа от счетчика высвечивается слово Pending. По истечении 10 секунд слово Pending меняется на Executed, и появляется

результат вызова функции. Если Вы внимательно следите за работой программы, то увидите, что фокус теперь установлен на окно списка Function. Но ввод с клавиатуры по-прежнему направляется в Notepad. Таким образом, и поток LISLab, и поток Notepad — оба считают, что в фокусе находится одно из их окон. Но на самом деле RIT остается связанным с потоком Notepad.

И последнее замечание в этой связи: *SetFocus* и *SetActiveWindow* возвращают описатель окна, которое изначально находилось в фокусе или было активным. Информация об этом окне отображается в поле PrevWnd. Кроме того, непосредственно перед вызовом *SetForegroundWindow* программа обращается к *GetForegroundWindow*, чтобы получить описатель окна, которое располагалось «поверх» остальных окон. Эта информация также отображается в поле PrevWnd.

Далее поэкспериментируем с курсором мыши. Всякий раз, когда курсор проходит над диалоговым окном LISLab (но не над каким-либо из его дочерних окон), он изображается в виде вертикальной стрелки. По мере поступления диалоговому окну сообщения от мыши добавляются в список Mouse Messages Received. Таким образом, Вы всегда в курсе того, когда диалоговое окно получает сообщения от мыши. Сдвинув курсор за пределы основного окна или поместив его на одно из дочерних окон, Вы увидите, что сообщения больше не вносятся в список.

Теперь переместите курсор в правую часть диалогового окна, установив его над текстом Click Right Mouse Button To Set Capture, а затем нажмите и удерживайте правую кнопку мыши. После этого LISLab вызовет функцию *SetCapture*, передав ей описатель своего диалогового окна. Заметьте: факт захвата мыши программой LISLab отразится в разделе Windows.

Не отпуская правую кнопку, проведите курсор над дочерними окнами LISLab и понаблюдайте за сообщениями от мыши, добавляемыми к списку. Если курсор выведен за пределы диалогового окна LISLab, программа по-прежнему получает сообщения от мыши. Курсор сохраняет форму вертикальной стрелки независимо от того, над каким участком экрана Вы его перемещаете.

Теперь мы можем увидеть другой эффект. Отпустите правую кнопку и следите, что произойдет. Окно, в свое время захватившее мышь, показывает, что LISLab по-прежнему считает мышь захваченной. Но сдвиньте курсор за пределы диалогового окна LISLab, и он больше не останется вертикальной стрелкой, а сообщения от мыши перестанут появляться в списке Mouse Messages Received. Установив же курсор на какое-либо из дочерних окон LISLab, Вы сразу увидите: захват по-прежнему действует, потому что все эти окна используют один набор переменных локального состояния ввода.

Закончив эксперименты, отключите режим захвата одним из двух способов:

- двойным щелчком правой кнопки мыши в любом месте диалогового окна LISLab (чтобы программа вызвала функцию *ReleaseCapture*);
- щелчком окна, созданного любым другим потоком (отличным от потока LISLab). В этом случае система автоматически передаст диалоговому окну LISLab сообщения о нажатии и отпуске кнопки мыши.

Какой бы способ Вы ни выбрали, обратите внимание на поле Capture в разделе Windows — теперь оно отражает тот факт, что больше ни одно окно не захватывает мышь.

И еще два эксперимента, связанных с мышью: в одном мы ограничим поле перемещения курсора заданным прямоугольником, а в другом — изменим «видимость» курсора. Если Вы щелкнете кнопку «Top, Left», программа LISLab выполнит такой код:

```
RECT rc;
:
```

```
SetRect(&rc, 0, 0, GetSystemMetrics(SM_CXSCREEN) / 2,  
GetSystemMetrics(SM_CYSCREEN) / 2);
```

Это ограничит поле перемещения курсора верхней левой четвертью экрана. Переключившись в окно другого приложения нажатием клавиш Alt+Tab, Вы заметите, что ограничение по-прежнему действует. Но система автоматически снимет его в результате одного из таких действий:

| | |
|--------------|--|
| Windows 98 | щелчка заголовка окна другого приложения и последующего перемещения этого окна; |
| Windows 2000 | щелчка заголовка окна другого приложения (последующего перемещения этого окна не требуется); |
| Windows 2000 | активизации Task Manager нажатием клавиш Ctrl+Shift+Esc и последующей его отмены. |

Для снятия ограничения на перемещение курсора можно также щелкнуть кнопку Remove в диалоговом окне LISLab (если эта кнопка находится в пределах текущего поля перемещения курсора).

Щелчок кнопки Hide или Show вызывает выполнение соответственно:

```
ShowCursor(FALSE);
```

или

```
ShowCursor(TRUE);
```

Когда курсор скрыт, его не видно при перемещении над диалоговым окном LISLab. Но как только курсор оказывается за пределами окна, он снова видим. Для нейтрализации действия кнопки Hide используйте кнопку Show. Заметьте, что скрытие курсора носит кумулятивный характер: пять раз щелкнув кнопку Hide, придется столько же раз щелкнуть кнопку Show, прежде чем курсор станет видимым.

И последний эксперимент — с кнопкой Infinite Loop. При ее щелчке выполняется код:

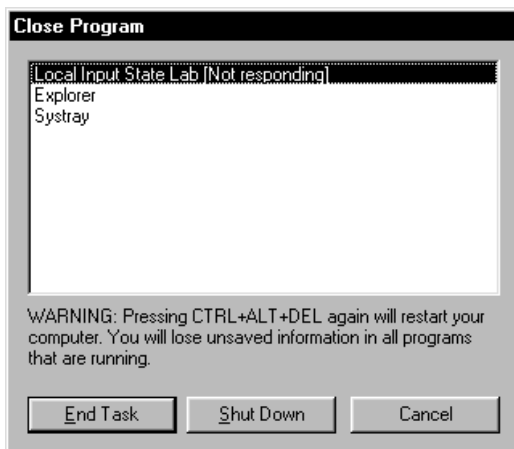
```
SetCursor(LoadCursor(NULL, IDC_NO));  
for (;;) ;
```

Первая строка меняет форму курсора на перечеркнутый круг, а вторая выполняет бесконечный цикл. После щелчка кнопки Infinite Loop программа перестает реагировать на какой-либо ввод. Перемещая курсор в пределах диалогового окна LISLab, Вы увидите, что курсор остается перечеркнутым кругом. Если же Вы сместите его в другое окно, он получит форму, заданную в текущем окне.

Если теперь вернуть курсор в диалоговое окно LISLab, система обнаружит, что программа не отвечает, и автоматически восстановит прежнюю форму курсора — перечеркнутый круг. Так что вошедший в бесконечный цикл поток хоть и не вызывает положительных эмоций, но пользоваться другими окнами не мешает.

Заметьте: если Вы переместите на окно зависшей программы LISLab другое окно, а потом уберете его, система отправит LISLab сообщение WM_PAINT и обнаружит, что данный поток не отвечает. Из этой ситуации система выходит элементарно: перерисовывает окно нереагирующего приложения. Конечно, перерисовать окно правильно она не в состоянии, так как ей не известно, что именно делало приложение, и поэтому она просто затирает окно цветом фона и перерисовывает рамку его окна.

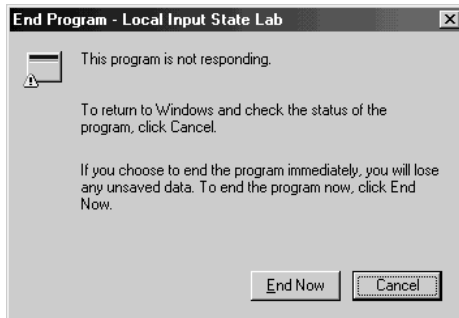
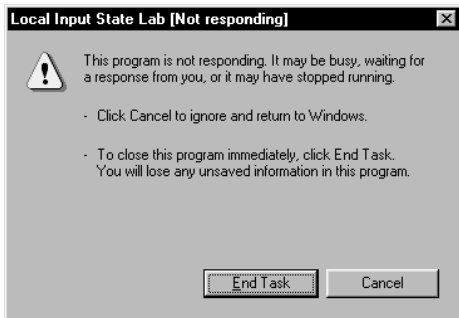
Проблема теперь в том, что на экране есть ни на что не отвечающее окно. Как от него избавиться? Под управлением Windows 98 нужно сначала нажать клавиши Ctrl+Alt+Del, чтобы на экране появилось окно Close Program.



В Windows 2000 можно либо щелкнуть правой кнопкой мыши кнопку приложения на панели задач, либо открыть окно Task Manager.



Затем следует выбрать из списка название программы, которую нужно завершить (в данном случае — Local Input State Lab), и щелкнуть кнопку End Task. Система попытается завершить LISLab «по-хорошему» (послав сообщение WM_CLOSE), но обнаружит, что приложение не отвечает. Это заставит ее вывести одно из окон: первое — в Windows 98, второе — в Windows 2000.



Если Вы выберете кнопку End Task (в Windows 98) или End Now (в Windows 2000), система завершит LISLab принудительно. Кнопка Cancel сообщит системе, что Вы передумали завершать приложение. Так что щелкните кнопку End Task или End Now, чтобы удалить LISLab из системы.

Общий смысл этих экспериментов — продемонстрировать отказоустойчивость системы. Ни одно приложение практически не способно привести систему в такое состояние, когда работа с другими приложениями станет невозможной. Кроме того, и Windows 98, и Windows 2000 автоматически освобождают все ресурсы, выделявшиеся потокам завершенного процесса, — утечки памяти не происходит!



LISLab.cpp

```

/*****
Модуль: LISLab.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h" /* см. приложение A */
#include <windowsx.h>
#include <tchar.h>
#include <string.h>
#include "Resource.h"

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

#define TIMER_DELAY (500) // полсекунды

UINT_PTR g_uTimerId = 1;
Int g_nEventId = 0;
DWORD g_dwEventTime = 0;
HWND g_hwndSubject = NULL;
HWND g_hwndNotepad = NULL;

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

void CalcWndText(HWND hwnd, PTSTR szBuf, int nLen) {

    TCHAR szClass[50], szCaption[50], szBufT[150];

    if (hwnd == (HWND) NULL) {
        _tcscpy(szBuf, TEXT("(no window)"));
        return;
    }
    if (!IsWindow(hwnd)) {
        _tcscpy(szBuf, TEXT("(invalid window)"));
        return;
    }

    GetClassName(hwnd, szClass, chDIMOF(szClass));
    GetWindowText(hwnd, szCaption, chDIMOF(szCaption));

```

Рис. 27-4. Программа-пример LISLab

см. след. стр

Рис. 27-4. *продолжение*

```

wprintf(szBufT, TEXT("[%s] %s"), (PTSTR) szClass,
    (*szCaption == 0) ? (PTSTR) TEXT("(no caption)") : (PTSTR) szCaption);
_tcsncpy(szBuf, szBufT, nLen - 1);
szBuf[nLen - 1] = 0; // принудительно завершаем строку нулевым символом
}

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

// для уменьшения используемого размера стека один экземпляр
// WALKWINDOWTREEDATA создаем как локальную переменную в WalkWindowTree()
// и указатель на нее передаем функции WalkWindowTreeRecurse

// данные, используемые WalkWindowTreeRecurse
typedef struct {
    HWND  hwndLB;           // описатель окна списка для вывода
    HWND  hwndParent;       // описатель родительского окна
    Int    nLevel;          // глубина рекурсии
    Int    nIndex;          // индекс элемента списка
    TCHAR  szBuf[100];       // буфер вывода
    Int    iBuf;            // индекс в szBuf
} WALKWINDOWTREEDATA, *PWALKWINDOWTREEDATA;

void WalkWindowTreeRecurse(PWALKWINDOWTREEDATA pWWT) {

    if (!IsWindow(pWWT->hwndParent))
        return;

    pWWT->nLevel++;
    const int nIndexAmount = 2;

    for (pWWT->iBuf = 0; pWWT->iBuf < pWWT->nLevel * nIndexAmount; pWWT->iBuf++)
        pWWT->szBuf[pWWT->iBuf] = TEXT(' ');

    CalcWndText(pWWT->hwndParent, &pWWT->szBuf[pWWT->iBuf],
        chDIMOF(pWWT->szBuf) - pWWT->iBuf);
    pWWT->nIndex = ListBox_AddString(pWWT->hwndLB, pWWT->szBuf);
    ListBox_SetItemData(pWWT->hwndLB, pWWT->nIndex, pWWT->hwndParent);

    HWND hwndChild = GetFirstChild(pWWT->hwndParent);
    while (hwndChild != NULL) {
        pWWT->hwndParent = hwndChild;
        WalkWindowTreeRecurse(pWWT);
        hwndChild = GetNextSibling(hwndChild);
    }

    pWWT->nLevel--;
}

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

void WalkWindowTree(HWND hwndLB, HWND hwndParent) {

```

Рис. 27-4. *продолжение*

```
WALKWINDOWTREEDATA WWT;

WWT.hwndLB = hwndLB;
WWT.hwndParent = hwndParent;
WWT.nLevel = -1;

WalkWindowTreeRecurse(&WWT);
}

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

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

    ChSETDLGICONS(hwnd, IDI_LISLAB);

    // связываем курсор в форме "стрелка вверх" с клиентской областью
    // диалогового окна
    SetClassLongPtr(hwnd, GCLP_HCURSOR,
        (LONG_PTR) LoadCursor(NULL, IDC_UPARROW));

    g_uTimerId = SetTimer(hwnd, g_uTimerId, TIMER_DELAY, NULL);

    HWND hwndT = GetDlgItem(hwnd, IDC_WNDFUNC);
    ComboBox_AddString(hwndT, TEXT("SetFocus"));
    ComboBox_AddString(hwndT, TEXT("SetActiveWindow"));
    ComboBox_AddString(hwndT, TEXT("SetForegroundWnd"));
    ComboBox_AddString(hwndT, TEXT("BringWindowToTop"));
    ComboBox_AddString(hwndT, TEXT("SetWindowPos-TOP"));
    ComboBox_AddString(hwndT, TEXT("SetWindowPos-BTM"));
    ComboBox_SetCurSel(hwndT, 0);

    // помещаем в раздел Windows информацию о нашем окне
    hwndT = GetDlgItem(hwnd, IDC_WNDS);
    ListBox_AddString(hwndT, TEXT("----> This dialog box <---"));

    ListBox_SetItemData(hwndT, 0, hwnd);
    ListBox_SetCurSel(hwndT, 0);

    // теперь добавляем информацию об окнах Notepad
    g_hwndNotepad = FindWindow(TEXT("Notepad"), NULL);
    if (g_hwndNotepad == NULL) {

        // Notepad не выполняется; запускаем его
        STARTUPINFO si = { sizeof(si) };
        PROCESS_INFORMATION pi;
        TCHAR szCommandLine[] = TEXT("Notepad");
        if (CreateProcess(NULL, szCommandLine, NULL, NULL, FALSE, 0,
            NULL, NULL, &si, &pi)) {

            // ждем, когда Notepad создаст все свои окна
            WaitForInputIdle(pi.hProcess, INFINITE);
        }
    }
}
```

см. след. стр.

Рис. 27-4. *продолжение*

```

        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
        g_hwndNotepad = FindWindow(TEXT("Notepad"), NULL);
    }
}
WalkWindowTree(hwndT, g_hwndNotepad);

return(TRUE);
}

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

void Dlg_OnCommand(HWND hwnd, int id, HWND hwndCtl, UINT codeNotify) {

    HWND hwndT;

    switch (id) {

        case IDCANCEL:
            if (g_uTimerId != 0)
                KillTimer(hwnd, g_uTimerId);
            EndDialog(hwnd, 0);
            break;

        case IDC_FUNCSTART:
            g_dwEventTime = GetTickCount() + 1000 *
                GetDlgItemInt(hwnd, IDC_DELAY, NULL, FALSE);
            hwndT = GetDlgItem(hwnd, IDC_WNDS);
            g_hwndSubject = (HWND)
                ListBox_GetItemData(hwndT, ListBox_GetCurSel(hwndT));
            g_nEventId = ComboBox_GetCurSel(GetDlgItem(hwnd, IDC_WNDFUNC));
            SetWindowText(GetDlgItem(hwnd, IDC_EVENTPENDING), TEXT("Pending"));
            break;

        case IDC_THREADATTACH:
            AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),
                GetCurrentThreadId(), TRUE);
            break;

        case IDC_THREADDETACH:
            AttachThreadInput(GetWindowThreadProcessId(g_hwndNotepad, NULL),
                GetCurrentThreadId(), FALSE);
            break;

        case IDC_REMOVECLIPRECT:
            ClipCursor(NULL);
            break;

        case IDC_HIDECURSOR:
            ShowCursor(FALSE);
            break;
    }
}

```

Рис. 27-4. *продолжение*

```

        case IDC_SHOWCURSOR:
            ShowCursor(TRUE);
            break;

        case IDC_INFINITELOOP:
            SetCursor(LoadCursor(NULL, IDC_NO));
            for (;;)
                ;
            break;

        case IDC_SETCLIPRECT:
            RECT rc;
            SetRect(&rc, 0, 0, GetSystemMetrics(SM_CXSCREEN) / 2,
                GetSystemMetrics(SM_CYSCREEN) / 2);
            ClipCursor(&rc);
            break;
    }
}

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

void AddStr(HWND hwndLB, PCTSTR szBuf) {

    int nIndex;

    do {
        nIndex = ListBox_AddString(hwndLB, szBuf);
        if (nIndex == LB_ERR)
            ListBox_DeleteString(hwndLB, 0);
    } while (nIndex == LB_ERR);
    ListBox_SetCurSel(hwndLB, nIndex);
}

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

int Dlg_OnRButtonDown(HWND hwnd, BOOL fDoubleClick,
    int x, int y, UINT keyFlags) {

    TCHAR szBuf[100];
    wsprintf(szBuf,
        TEXT("Capture=%-3s, Msg=RButtonDown, DblClk=%-3s, x=%5d, y=%5d"),
        (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"),
        fDoubleClick ? TEXT("Yes") : TEXT("No"), x, y);

    AddStr(GetDlgItem(hwnd, IDC_MOUSEMSG, szBuf);
    if (!fDoubleClick) SetCapture(hwnd);
    else ReleaseCapture();
    return(0);
}

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

```

см. след. стр.

Рис. 27-4. *продолжение*

```
int Dlg_OnRButtonUp(HWND hwnd, int x, int y, UINT keyFlags) {

    TCHAR szBuf[100];
    wsprintf(szBuf, TEXT("Capture=%-3s, Msg=RButtonUp, x=%5d, y=%5d"),
        (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"), x, y);

    AddStr(GetDlgItem(hwnd, IDC_MOUSEMSGs), szBuf);
    return(0);
}

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

int Dlg_OnLButtonDown(HWND hwnd, BOOL fDoubleClick,
    int x, int y, UINT keyFlags) {

    TCHAR szBuf[100];
    wsprintf(szBuf, TEXT("Capture=%-3s, Msg=LButtonDown, DblClk=%-3s, x=%5d, y=%5d"),
        (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"),
        fDoubleClick ? TEXT("Yes") : TEXT("No"), x, y);

    AddStr(GetDlgItem(hwnd, IDC_MOUSEMSGs), szBuf);
    return(0);
}

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

void Dlg_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags) {

    TCHAR szBuf[100];
    wsprintf(szBuf,
        TEXT("Capture=%-3s, Msg=LButtonUp, x=%5d, y=%5d"),
        (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"), x, y);

    AddStr(GetDlgItem(hwnd, IDC_MOUSEMSGs), szBuf);
}

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

void Dlg_OnMouseMove(HWND hwnd, int x, int y, UINT keyFlags) {

    TCHAR szBuf[100];
    wsprintf(szBuf, TEXT("Capture=%-3s, Msg=MouseMove, x=%5d, y=%5d"),
        (GetCapture() == NULL) ? TEXT("No") : TEXT("Yes"), x, y);

    AddStr(GetDlgItem(hwnd, IDC_MOUSEMSGs), szBuf);
}

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

void Dlg_OnTimer(HWND hwnd, UINT id) {

    TCHAR szBuf[100];
```

Рис. 27-4. *продолжение*

```

CalcWndText(GetFocus(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDFOCUS), szBuf);

CalcWndText(GetCapture(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDCAPTURE), szBuf);

CalcWndText(GetActiveWindow(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDACTIVE), szBuf);

CalcWndText(GetForegroundWindow(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDFOREGROUND), szBuf);

RECT rc;
GetClipCursor(&rc);
wsprintf(szBuf, TEXT("left=%d, top=%d, right=%d, bottom=%d"),
        rc.left, rc.top, rc.right, rc.bottom);
SetWindowText(GetDlgItem(hwnd, IDC_CLIPCURSOR), szBuf);

if ((g_dwEventTime == 0) || (GetTickCount() < g_dwEventTime))
    return;

HWND hwndT;
switch (g_nEventId) {
    case 0: // SetFocus
        g_hwndSubject = SetFocus(g_hwndSubject);
        break;

    case 1: // SetActiveWindow
        g_hwndSubject = SetActiveWindow(g_hwndSubject);
        break;

    case 2: // SetForegroundWindow
        hwndT = GetForegroundWindow();
        SetForegroundWindow(g_hwndSubject);
        g_hwndSubject = hwndT;
        break;

    case 3: // BringWindowToTop
        BringWindowToTop(g_hwndSubject);
        break;

    case 4: // SetWindowPos c HWND_TOP
        SetWindowPos(g_hwndSubject, HWND_TOP, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
        g_hwndSubject = (HWND) 1;
        break;

    case 5: // SetWindowPos c HWND_BOTTOM
        SetWindowPos(g_hwndSubject, HWND_BOTTOM, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
        g_hwndSubject = (HWND) 1;
        break;
}

```

см. след. стр.

Рис. 27-4. *продолжение*

```

if (g_hwndSubject == (HWND) 1) {
    SetWindowText(GetDlgItem(hwnd, IDC_PREVWND), TEXT("Can't tell.));
} else {
    CalcWndText(g_hwndSubject, szBuf, chDIMOF(szBuf));
    SetWindowText(GetDlgItem(hwnd, IDC_PREVWND), szBuf);
}
g_hwndSubject = NULL; g_nEventId = 0; g_dwEventTime = 0;
SetWindowText(GetDlgItem(hwnd, IDC_EVENTPENDING), TEXT("Executed"));
}

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

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_MOUSEMOVE, Dlg_OnMouseMove);
        chHANDLE_DLGMSG(hwnd, WM_LBUTTONDOWN, Dlg_OnLButtonDown);
        chHANDLE_DLGMSG(hwnd, WM_LBUTTONDBLCLK, Dlg_OnLButtonDown);
        chHANDLE_DLGMSG(hwnd, WM_LBUTTONUP, Dlg_OnLButtonUp);
        chHANDLE_DLGMSG(hwnd, WM_RBUTTONDOWN, Dlg_OnRButtonDown);
        chHANDLE_DLGMSG(hwnd, WM_RBUTTONDBLCLK, Dlg_OnRButtonDown);
        chHANDLE_DLGMSG(hwnd, WM_RBUTTONUP, Dlg_OnRButtonUp);
        chHANDLE_DLGMSG(hwnd, WM_TIMER, Dlg_OnTimer);
    }
    return(FALSE);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

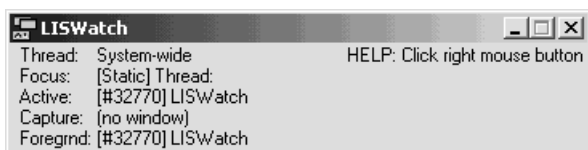
    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_LISLAB), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конец файла //////////////////////////////////////

```

Программа-пример LISWatch

Эта программа, «27 LISWatch.exe» (см. листинг на рис. 27-5), — полезная утилита, которая отслеживает следующие окна: активное, находящееся в фокусе и захватившее мышь. Файлы исходного кода и ресурсов этой программы находятся в каталоге 27-LISWatch на компакт-диске, прилагаемом к книге. После запуска LISWatch открывает диалоговое окно, показанное ниже.



Получив сообщение WM_INITDIALOG, это окно вызывает *SetTimer*, чтобы создать таймер, срабатывающий два раза в секунду. А когда от таймера поступает сообщение WM_TIMER, обновляется содержимое диалогового окна — сведения об активном окне, окне в фокусе и окне, захватившем мышь. В нижней части диалогового окна также сообщается, какое окно находится сейчас на переднем плане. Поэкспериментируйте с этой утилитой, просто щелкая окна, созданные различными приложениями. При этом Вы заметите, что LISWatch корректно сообщает информацию об окнах независимо от того, какой поток создал то или иное окно.

Самая интересная часть кода этой программы выполняется при приеме сообщения WM_TIMER, поэтому, когда я буду давать пояснения, посматривайте на исходный код функции *Dlg_OnTimer*. И пока считайте, что глобальная переменная *g_dwThreadIdAttachTo* равна 0. Ее назначение я объясню позже.

Так как у каждого потока свой набор переменных локального состояния ввода, *Dlg_OnTimer* сначала вызывает *GetForegroundWindow* для определения окна, с которым имеет дело пользователь. *GetForegroundWindow* всегда возвращает действительный описатель окна независимо от того, какой поток создал это окно или вызвал эту функцию.

Полученный таким образом описатель передается в *GetWindowThreadProcessId*, чтобы выяснить, какой поток связан с RIT. Далее LISWatch, вызвав *AttachThreadInput*, подключается к переменным локального состояния ввода потока, связанного с RIT. После этого поток LISWatch может вызывать *GetFocus*, *GetActiveWindow* и *GetCapture* — любая из них возвращает действительный описатель окна. Вспомогательная функция *CalcWndText* формирует строки с именами классов и заголовками окон. Потом эти строки показываются в диалоговом окне LISWatch. И, наконец, перед самым возвратом управления *Dlg_OnTimer* снова вызывает *AttachThreadInput*, на этот раз передавая FALSE в последнем параметре и тем самым отключая два потока друг от друга.

Ну вот, общее представление о программе LISWatch Вы получили. Однако она поддерживает и другую функциональность, о которой я хочу сейчас рассказать. После запуска LISWatch отслеживает активизацию окон во всех частях системы. На это и указывает надпись System-wide в верхней части диалогового окна. Но LISWatch может наблюдать за локальным состоянием ввода и только в одном потоке. Если Вы выберете именно этот режим, LISWatch будет сообщать Вам о том, что «думает» поток о своем состоянии ввода.

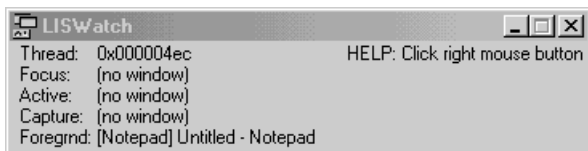
Чтобы LISWatch контролировала состояние ввода лишь одного потока, щелкните окно LISWatch левой кнопкой мыши и, не отпуская кнопку, переместите курсор мыши в окно, созданное другим потоком, а затем отпустите кнопку. Как только Вы это делаете, LISWatch присвоит глобальной переменной *g_dwThreadIdAttachTo* идентификатор выбранного потока, значение которого заменит надпись System-wide в верхней части окна LISWatch. При ненулевом значении этой переменной функция *Dlg_OnTimer* ведет себя несколько иначе: теперь она подключает LISWatch к переменным локального состояния ввода выбранного Вами потока.

Прodelаем один эксперимент. Запустите Calculator и выберите его окно в LISWatch. Активизировав окно Calculator, Вы увидите следующую информацию.



В моей системе идентификатор потока Calculator получил значение 0x000004ec. В данном случае LISWatch настроена на наблюдение за локальным состоянием ввода этого потока. Если я щелкну любую кнопку или флажок в Calculator, LISWatch моментально отреагирует на это, сообщив о соответствующей смене фокуса.

Однако, если Вы теперь активизируете окно, созданное другим приложением (у меня — Notepad), окно LISWatch будет выглядеть так, как показано ниже.



Как видите, поток Calculator считает, что нет ни одного окна, которое находилось бы в фокусе, было бы активно или захватило бы мышь.

Чтобы по-настоящему разобраться в концепции локального состояния ввода, запустите несколько копий LISWatch и настройте их на мониторинг отдельных потоков. После этого щелкайте в их окнах и следите, что происходит с локальным состоянием ввода каждого из потоков.

LISWatch.cpp

```

/*****
Модуль: LISWatch.cpp
Автор: Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
*****/

#include "..\CmnHdr.h"    /* см. приложение A */
#include <tchar.h>
#include <windowsx.h>
#include "Resource.h"

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

#define TIMER_DELAY (500)    // полсекунды

UINT_PTR g_uTimerId = 1;
DWORD g_dwThreadIdAttachTo = 0;    // 0 – в масштабе всей системы,
                                   // ненулевое значение – конкретный поток

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

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

    chSETDLGICONS(hwnd, IDI_LISWATCH);

    // периодически обновляем информацию
    g_uTimerId = SetTimer(hwnd, g_uTimerId, TIMER_DELAY, NULL);
    // помещаем наше окно "поверх" остальных
    SetWindowPos(hwnd, HWND_TOPMOST, 0, 0, 0, 0, SWP_NOMOVE | SWP_NOSIZE);
    return(TRUE);
}

```

Рис. 27-5. Программа-пример LISWatch

Рис. 27-5. *продолжение*

```

////////////////////////////////////
void Dlg_OnRButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y,
    UINT keyFlags) {

    chMB("To monitor a specific thread, click the left mouse button in "
        "the main window and release it in the desired window.\n"
        "To monitor all threads, double-click the left mouse button in "
        "the main window.");
}

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

void Dlg_OnLButtonDown(HWND hwnd, BOOL fDoubleClick, int x, int y,
    UINT keyFlags) {

    // если мы присоединены к потоку, отсоединяемся от него
    if (g_dwThreadIdAttachTo != 0)
        AttachThreadInput(GetCurrentThreadId(), g_dwThreadIdAttachTo, FALSE);

    // захватываем мышь и изменяем форму ее курсора
    SetCapture(hwnd);
    SetCursor(LoadCursor(GetModuleHandle(NULL), MAKEINTRESOURCE(IDC_EYES)));
}

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

void Dlg_OnLButtonUp(HWND hwnd, int x, int y, UINT keyFlags) {

    if (GetCapture() == hwnd) {

        // если мышь захвачена нами, определяем идентификатор потока,
        // создавшего окно, над которым находится сейчас курсор
        POINT pt;
        pt.x = LOWORD(GetMessagePos());
        pt.y = HIWORD(GetMessagePos());
        ReleaseCapture();
        g_dwThreadIdAttachTo = GetWindowThreadProcessId(
            ChildWindowFromPointEx(GetDesktopWindow(), pt, CWP_SKIPINVISIBLE), NULL);

        if (g_dwThreadIdAttachTo == GetCurrentThreadId()) {

            // кнопка мыши отпущена в одном из наших окон;
            // отражаем локальное состояние ввода на общесистемном уровне
            g_dwThreadIdAttachTo = 0;

        } else {

            // кнопка мыши отпущена в окне, которое не принадлежит нашему потоку;
            // отражаем локальное состояние ввода только выбранного потока
            AttachThreadInput(GetCurrentThreadId(), g_dwThreadIdAttachTo, TRUE);

        }

    }
}

```

см. след. стр.

Рис. 27-5. *продолжение*

```

    }
}

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

static void CalcWndText(HWND hwnd, PTSTR szBuf, int nLen) {

    if (hwnd == (HWND) NULL) {
        lstrcpy(szBuf, TEXT("(no window)"));
        return;
    }

    if (!IsWindow(hwnd)) {
        lstrcpy(szBuf, TEXT("(invalid window)"));
        return;
    }

    TCHAR szClass[50], szCaption[50], szBufT[150];
    GetClassName(hwnd, szClass, chDIMOF(szClass));
    GetWindowText(hwnd, szCaption, chDIMOF(szCaption));
    wsprintf(szBufT, TEXT("[%s] %s"), (PTSTR) szClass,
        (szCaption[0] == 0) ? (PTSTR) TEXT("(no caption)") : (PTSTR) szCaption);
    _tcsncpy(szBuf, szBufT, nLen - 1);
    szBuf[nLen - 1] = 0; // принудительно завершаем строку нулевым символом
}

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

void Dlg_OnTimer(HWND hwnd, UINT id) {

    TCHAR szBuf[100] = TEXT("System-wide");
    HWND hwndForeground = GetForegroundWindow();
    DWORD dwThreadIdAttachTo = g_dwThreadIdAttachTo;

    if (dwThreadIdAttachTo == 0) {

        // если мы отслеживаем локальное состояние ввода на общесистемном уровне,
        // присоединяем наши переменные локального состояния ввода к аналогичным
        // переменным потока, которому принадлежит активное окно
        DwThreadIdAttachTo = GetWindowThreadProcessId(hwndForeground, NULL);
        AttachThreadInput(GetCurrentThreadId(), dwThreadIdAttachTo, TRUE);

    } else {

        wsprintf(szBuf, TEXT("0x%08x"), dwThreadIdAttachTo);
    }

    SetWindowText(GetDlgItem(hwnd, IDC_THREADID), szBuf);

    CalcWndText(GetFocus(), szBuf, chDIMOF(szBuf));
    SetWindowText(GetDlgItem(hwnd, IDC_WNDFOCUS), szBuf);
}

```

Рис. 27-5. *продолжение*

```

CalcWndText(GetActiveWindow(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDACTIVE), szBuf);

CalcWndText(GetCapture(), szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDCAPTURE), szBuf);

CalcWndText(hwndForeground, szBuf, chDIMOF(szBuf));
SetWindowText(GetDlgItem(hwnd, IDC_WNDFOREGRND), szBuf);

if (g_dwThreadIdAttachTo == 0) {
    // если мы отслеживаем локальное состояние ввода на общесистемном уровне,
    // отсоединяем наши переменные локального состояния ввода от аналогичных
    // переменных потока, которому принадлежит активное окно
    AttachThreadInput(GetCurrentThreadId(), dwThreadIdAttachTo, FALSE);
}
}

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

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) {

    switch (uMsg) {
        chHANDLE_DLGMSG(hwnd, WM_INITDIALOG, Dlg_OnInitDialog);
        chHANDLE_DLGMSG(hwnd, WM_COMMAND, Dlg_OnCommand);
        chHANDLE_DLGMSG(hwnd, WM_TIMER, Dlg_OnTimer);
        chHANDLE_DLGMSG(hwnd, WM_RBUTTONDOWN, Dlg_OnRButtonDown);
        chHANDLE_DLGMSG(hwnd, WM_LBUTTONDOWN, Dlg_OnLButtonDown);
        chHANDLE_DLGMSG(hwnd, WM_LBUTTONUP, Dlg_OnLButtonUp);
    }
    return(FALSE);
}

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

int WINAPI _tWinMain(HINSTANCE hinstExe, HINSTANCE, PTSTR pszCmdLine, int) {

    DialogBox(hinstExe, MAKEINTRESOURCE(IDD_LISWATCH), NULL, Dlg_Proc);
    return(0);
}

//////////////////////////////////// Конеч файла //////////////////////////////////////

```


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 `Project Settings` и изменял следующие настройки:

- на вкладке `General` в поле `Output Files` я указывал конкретный каталог, чтобы в него помещались все `EXE`- и `DLL`-файлы;
- на вкладке `C/C++` в списке `Category` я выбирал `Code Generation`, а в списке `Use Run-Time Library` — `Multithreaded DLL`.

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 (см. листинг на рис. А-1), 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 на компакт-диске, прилагаемом к книге.

Далее я расскажу обо всех разделах заголовочного файла `CmnHdr.h` 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 2000`, в этом разделе определяется идентификатор `_WIN32_WINNT`:

```
#define _WIN32_WINNT 0x0500
```

Мне пришлось это сделать, так как прототипы новых функций в заголовочных файлах `Windows` задаются так:

```

#if (_WIN32_WINNT >= 0x0500)
:
WINBASEAPI
BOOL
WINAPI
AssignProcessToJobObject(
    IN HANDLE hJob,
    IN HANDLE hProcess
);
:
#endif /* _WIN32_WINNT >= 0x0500 */

```

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. Microsoft специально защитила эти функции идентификатором `_WIN32_WINNT`, чтобы разработчики, ориентирующиеся на разные версии Windows, случайно не использовали функции, существующие пока только в Windows 2000.

Раздел Unicode Build Option

Все примеры написаны так, чтобы их можно было компилировать с использованием как ANSI, так и Unicode. При компиляции для процессоров x86 — чтобы программы работали и в Windows 98 — по умолчанию принимается кодировка ANSI. Но для других процессорных платформ при сборке применяется Unicode, что ускоряет работу программ.

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 Microsoft 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, включая 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\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.

Макросы `chINRANGE` и `chDIMOF`

Это весьма полезные макросы, которыми я часто пользуюсь в своих программах. Первый макрос проверяет, укладывается ли какое-то значение в заданный диапазон, а второй просто возвращает число элементов в массиве. Делается это так: оператор `sizeof` вычисляет размер массива в байтах, а результат делится на число байтов, занимаемых одним элементом.

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,
    DWORD cbStack,
    PTHREAD_START_ROUTINE pfnStartAddr,
    PVOID pvParam,
    DWORD fdwCreate,
    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 Microsoft 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 *_beginthreadex* 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 Microsoft следовало бы объявить прототип *_beginthreadex* так, чтобы предупреждения не генерировались:

```
unsigned long __cdecl _beginthreadex(
    void *psa,
    unsigned long cbStack,
    unsigned (__stdcall *) (void *pvParam),
    void *pvParam,
    unsigned long fdwCreate,
    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, cbStack, pfnStartAddr, \
    pvParam, fdwCreate, pdwThreadId) \
    ((HANDLE)_beginthreadex(
```

см. след. стр.

```
(void *)          (psa),                \  
(unsigned)       (cbStack),            \  
(PTHREAD_START) (pfnStartAddr),        \  
(void *)         (pvParam),            \  
(unsigned)       (fdwCreate),           \  
(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 x86, 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 *MAKESOFTWAREEXCEPTION*.

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.

Встраиваемые функции для проверки версии операционной системы

Большинство программ-примеров работает на всех платформах, но некоторые из них требуют функций, не поддерживаемых Windows 95 и Windows 98, а какая-то часть использует функции, имеющиеся только в Windows 2000. Поэтому каждая программа проверяет при инициализации версию системы и выводит соответствующее сообщение, если ей нужна более современная операционная система.

В функции `_tWinMain` программ-примеров, не работающих в Windows 95 и Windows 98, Вы увидите вызов моей функции `cbWindows9xNotAllowed`, а в аналогичном месте программ-примеров, работающих только в Windows 2000, — вызов моей функции `cbWindows2000Required`.

Проверка на поддержку Unicode

Windows 98 не поддерживает Unicode в той мере, как Windows 2000, и фактически приложения, вызывающие Unicode-функции, просто не будут работать в ней! К сожалению, Windows 98 не предупреждает о запуске приложения, использующего Unicode. Применительно к моим программам это означало бы, что они будут немедленно завершаться без всякой диагностики. Спать можно!

Поскольку надо было как-то узнавать, что программа, скомпилированная под Unicode, запускается в Windows 98, я создал C++-класс `CUnicodeSupported`. Его конструктор проверяет, полностью ли поддерживает данная операционная система Unicode, и, если нет, сообщает об этом — только после этого процесс завершается.

Вы заметите, что в `CmnHdr.h` я создаю глобальный статический экземпляр этого класса. Когда моя программа запускается, стартовый код из библиотеки C/C++ вызывает конструктор данного объекта. Если конструктор обнаруживает, что операционная система полностью поддерживает Unicode, он просто возвращает управление, и программа продолжает выполнение. Поскольку я создаю глобальный экземпляр класса, мне не приходится включать специальный код в каждый модуль исходного кода программы. В программах, не использующих Unicode, C++-класс `CUnicodeSupported` не объявляется, и его экземпляр не создается.

Принудительное указание компоновщику входной функции (*w*)`WinMain`

Некоторые читатели предыдущих изданий этой книги, включавшие мои модули исходного кода в новые проекты Visual C++, 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 Visual C++ 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 C++ 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.

CmnHdr.h

```

/*****
Модуль:      CmnHdr.h
Автор:       Copyright (c) 2000, Джеффри Рихтер (Jeffrey Richter)
Назначение:  Общий заголовочный файл, содержащий полезные макросы
              и определения, используемые во всех примерах из этой книги
*****/

#pragma once // включайте этот заголовочный файл только в главный модуль

//////////////////// Windows Version Build Option //////////////////////

#define _WIN32_WINNT 0x0500
// #define WINVER      0x0500

//////////////////// Unicode Build Option //////////////////////

// если мы компилируем не для x86, используем только Unicode
#ifndef _M_IX86
#define UNICODE
#endif

// чтобы компилировать на x86 с использованием Unicode,
// раскомментируйте следующую строку
// #define UNICODE

// работая с Unicode-версиями Windows-функций, надо применять
// и Unicode-функции из библиотеки C
#ifdef UNICODE
#define _UNICODE
#endif

//////////////////// Windows Definitions //////////////////////

#pragma warning(push, 3)
#include <Windows.h>
#pragma warning(pop)
#pragma warning(push, 4)

////////// проверка на использование корректных заголовочных файлов //////////

#ifndef WT_EXECUTEINPERSISTENTIOTHREAD

```

Рис. А-1. Заголовочный файл *CmnHdr.h*

Рис. А-1. *продолжение*

```

#pragma message("You are not using the latest Platform SDK header/library ")
#pragma message("files. This may prevent the project from building correctly.")
#endif

////////// обеспечиваем корректную компиляцию при диагностике уровня 4 //////////

/* нестандартное расширение - однострочный комментарий */
#pragma warning(disable:4001)

// неиспользуемый формальный параметр
#pragma warning(disable:4100)

// Внимание: создается предкомпилированный заголовочный файл
#pragma warning(disable:4699)

// функция не подставлена
#pragma warning(disable:4710)

// подставляемая функция, на которую нет ссылок, удалена
#pragma warning(disable:4514)

// не удалось создать оператор присвоения
#pragma warning(disable:4512)

////////////////////////////////// вспомогательный макрос Pragma Message //////////////////////////////////

/*
когда компилятор встречает строку вида:
    #pragma chMSG(Fix this later)

он генерирует сообщение вроде:

    c:\CD\CmnHdr.h(82):Fix this later

Вы можете перейти прямо на эту строку и проверить окружающий ее код
*/

#define chSTR2(x)      #x
#define chSTR(x)      chSTR2(x)
#define chMSG(desc) message(__FILE__ "(" chSTR(__LINE__) "):" #desc)

////////////////////////////////// макрос chINRANGE //////////////////////////////////

// этот макрос возвращает TRUE, если число укладывается в заданный диапазон
#define chINRANGE(low, Num, High) (((low) <= (Num)) && ((Num) <= (High)))

////////////////////////////////// макрос chDIMOF //////////////////////////////////

// этот макрос сообщает количество элементов в массиве
#define chDIMOF(Array) (sizeof(Array) / sizeof(Array[0]))

```

см. след. стр.

Рис. А-1. *продолжение*

```

//////////////////// макрос chBEGINTHREADEX //////////////////////

// Макрос chBEGINTHREADEX вызывает библиотечную функцию _beginthreadex.
// Библиотека построена так, чтобы не зависеть от типов данных Windows, например
// HANDLE. Таким образом, при вызове _beginthreadex Windows-программист должен
// приводить значения к типам данных Windows. Это жутко неудобно, поэтому
// я написал макрос, который сам выполняет необходимые преобразования типов.
typedef unsigned (__stdcall *PTHREAD_START) (void *);

#define chBEGINTHREADEX(psa, cbStack, pfnStartAddr, \
    pvParam, fdwCreate, pdwThreadId) \
    ((HANDLE)_beginthreadex( \
        (void *) (psa), \
        (unsigned) (cbStack), \
        (PTHREAD_START) (pfnStartAddr), \
        (void *) (pvParam), \
        (unsigned) (fdwCreate), \
        (unsigned *) (pdwThreadId)))

//////////////////// моя реализация DebugBreak для платформы x86 //////////////////////

#ifdef _X86_
#define DebugBreak() _asm { int 3 }
#endif

//////////////////// макрос для определения кодов программных исключений //////////////////////

#define MAKESOFTWAREEXCEPTION(Severity, Facility, Exception) \
    ((DWORD) ( \
        /* код степени "тяжести" */ (Severity) | \
        /* MS (0) или польз. (1) */ (1) << 29) | \
        /* Зарезервирован (0) */ (0) << 28) | \
        /* Код подсистемы */ (Facility << 16) | \
        /* Код исключения */ (Exception << 0)))

//////////////////// макрос Quick MessageBox //////////////////////

inline void chMB(PCSTR s) {
    char szTMP[128];
    GetModuleFileNameA(NULL, szTMP, chDIMOF(szTMP));
    MessageBoxA(GetActiveWindow(), s, szTMP, MB_OK);
}

//////////////////// макросы chASSERT и chVERIFY //////////////////////

inline void chFAIL(PSTR szMsg) {
    chMB(szMsg);
    DebugBreak();
}

// выводим окно, которое сообщает о невыполнении условия
inline void chASSERTFAIL(LPCSTR file, int line, PCSTR expr) {

```

Рис. А-1. *продолжение*

```

    char sz[128];
    wprintfA(sz, "File %s, line %d : %s", file, line, expr);
    chFAIL(sz);
}

// выводим окно, если условие не выполнено в отладочном режиме
#ifdef _DEBUG
#define chASSERT(x) if (!(x)) chASSERTFAIL(__FILE__, __LINE__, #x)
#else
#define chASSERT(x)
#endif

// chASSERT работает только в режиме отладки,
// а chVERIFY из готовой программы не удаляется
#ifdef _DEBUG
#define chVERIFY(x) chASSERT(x)
#else
#define chVERIFY(x) (x)
#endif

////////// макрос chHANDLE_DLGMSG //////////

// Обычный макрос HANDLE_MSG из WindowsX.h для диалоговых окон работает
// некорректно, так как DlgProc-функции возвращают BOOL вместо LRESULT
// (подобно WndProc). Макрос chHANDLE_DLGMSG устраняет эту проблему.
#define chHANDLE_DLGMSG(hwnd, message, fn) \
    case (message): return (SetDlgMsgResult(hwnd, uMsg, \
        HANDLE_##message((hwnd), (wParam), (lParam), (fn))))

////////// макрос chSETDLGICONS //////////

// настраивает значки диалоговых окон
inline void chSETDLGICONS(HWND hwnd, int idi) {
    SendMessage(hwnd, WM_SETICON, TRUE, (LPARAM)
        LoadIcon((HINSTANCE) GetWindowLongPtr(hwnd, GWLP_HINSTANCE),
            MAKEINTRESOURCE(idi)));
    SendMessage(hwnd, WM_SETICON, FALSE, (LPARAM)
        LoadIcon((HINSTANCE) GetWindowLongPtr(hwnd, GWLP_HINSTANCE),
            MAKEINTRESOURCE(idi)));
}

////////// макросы для проверки версии операционной системы //////////

inline void chWindows9xNotAllowed() {
    OSVERSIONINFO vi = { sizeof(vi) };
    GetVersionEx(&vi);
    if (vi.dwPlatformId == VER_PLATFORM_WIN32_WINDOWS) {
        chMB("This application requires features not present in Windows 9x.");
        ExitProcess(0);
    }
}

```

см. след. стр.

Рис. А-1. *продолжение*

```
inline void chWindows2000Required() {
    OSVERSIONINFO vi = { sizeof(vi) };
    GetVersionEx(&vi);
    if ((vi.dwPlatformId != VER_PLATFORM_WIN32_NT) && (vi.dwMajorVersion < 5)) {
        chMB("This application requires features present in Windows 2000.");
        ExitProcess(0);
    }
}

////////// макрос для проверки поддержки UNICODE //////////

// Поскольку Windows 98 не поддерживает Unicode, выдаем сообщение об ошибке
// и завершаем процесс – если в Windows 98 запущена программа, рассчитанная
// на Unicode. Для этого создается глобальный C++-объект. Его конструктор
// выполняется до WinMain.

#ifdef UNICODE

class CUnicodeSupported {
public:
    CUnicodeSupported() {
        if (GetWindowsDirectoryW(NULL, 0) <= 0) {
            chMB("This application requires an OS that supports Unicode.");
            ExitProcess(0);
        }
    }
};

// так как объект объявлен статическим, компоновщик не жалуется на наличие
// нескольких экземпляров объекта в тех случаях, когда в одном проекте
// содержится несколько файлов исходного кода
static CUnicodeSupported g_UnicodeSupported;

#endif

////////// используем подсистему Windows //////////

#pragma comment(linker, "/subsystem:Windows")

////////// Конец файла //////////
```

Распаковщики сообщений, макросы для дочерних элементов управления и API-макросы

На конференциях я часто спрашиваю: «Применяете ли Вы распаковщики сообщений?» — и обычно получаю один ответ — нет. Оказывается, многие даже не знают, что это такое. 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 16-, 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 WM_COMMAND, 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 LBN_SELCHANGE
            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 `HANDLE_MSG` определен так:

```
#define HANDLE_MSG(hwnd, message, fn) \
    case (message): \
        return HANDLE_##message((hwnd), (wParam), (lParam), (fn));
```

Для сообщения `WM_COMMAND` 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 *Cls_OnCommand*). Вот макрос `HANDLE_WM_COMMAND`:

```
#define HANDLE_WM_COMMAND(hwnd, wParam, lParam, fn) \
    ((fn)((hwnd), (int)(LOWORD(wParam)), (HWND)(lParam), \
    (UINT)HIWORD(wParam)), 0L)
```

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)(lParam), \
    (UINT)HIWORD(wParam)), 0L)
#define 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 предопределенный макрос (message forwarder). Допустим, при обработке сообщения `WM_COMMAND` 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)SendMessage((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, автоматически 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, "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 почаще их используйте.

Предметный указатель

A

Address Windowing Extensions 387–391

D

DLL 476–526

— адресное пространство процесса 477–479

— внедрение в чужое адресное пространство 533–538, 539–553, 561–563

— известные 517–518

— отложенная загрузка 506–509

— перенаправление 518–5191

— поиск системой 479

— проецирование 409–413

— связывание

— — неявное 479–481

— — явное 492–493

— создание 481–485

— — для использования со средствами разработки, отличными от Visual C++ 485–486

— функция входа/выхода; см. функция, *DllMain*

— экспорт функций по порядковым номерам 485

— явная выгрузка 494–496

E

EXE-модуль

— запуск 479–490

— проецирование 409–413

— связывание 524–526

— создание 486–479

P

Performance Data Helper 83

Performance Monitor 111, 339

T

Terminal Server 42

U

Unicode 11–27

— COM 15

— Windows 2000 13

— Windows 98 13–14

— Windows CE 14

— библиотека C 15–16

— типы данных в Windows 17–18

V

VDM 70

A

адрес

— базовый 54, 410

— — модификация 519–523

— — предпочтительный 519–522

— — файла, проецируемого в память 439–440

— виртуальный

— — относительный 484

— — трансляция на физический 322

архитектура

— NUMA 182–183

— памяти 314–341

B

библиотека

— C/C++ 142–151

— — глобальные переменные, доступные программам 52–53

— — многопоточная версия 150

— *ImgWalk.dll* 578–561

блок

— *catch* 642

— *except* 602

— *finally* 595

— *try* 642

— переменных окружения

— — потока 320

— — процесса 320

блокировка, взаимная 206

B

ввод-вывод, асинхронный (на устройствах) 240–241

волокно 303–312

выравнивание данных 337–341

Г

«голодание» 167

Д

директива

— *#pragma comment* 417, 487, 517— *#pragma data_seg* 415–416

доступ, атомарный 187–192

З

задание 98–129

— завершение всех процессов 108

— ограничения 101–107

захват мыши 677

защита

— дескрипторы 29

— копирование при записи 325–326

— страниц 324–325

И

идентификатор

— `_MT` 150— `_WIN32_WINNT` 699— `CONTEXT_FULL` 165–166— `EXCEPTION_ACCESS_VIOLATION` 613— `EXCEPTION_ARRAY_BOUNDS_EXCEEDED` 613— `EXCEPTION_BREAKPOINT` 614— `EXCEPTION_CONTINUE_EXECUTION` 602, 609–610— `EXCEPTION_CONTINUE_SEARCH` 602, 611–612— `EXCEPTION_DATATYPE_MISALIGNMENT` 613— `EXCEPTION_EXECUTE_HANDLER` 602–608— `EXCEPTION_FLT_DENORMAL_OPERAND` 614— `EXCEPTION_FLT_DIVIDE_BY_ZERO` 614— `EXCEPTION_FLT_INEXACT_RESULT` 614— `EXCEPTION_FLT_INVALID_OPERATION` 614— `EXCEPTION_FLT_OVERFLOW` 614— `EXCEPTION_FLT_STACK_CHECK` 614— `EXCEPTION_FLT_UNDERFLOW` 614— `EXCEPTION_GUARD_PAGE` 613— `EXCEPTION_ILLEGAL_INSTRUCTION` 613— `EXCEPTION_IN_PAGE_ERROR` 613— `EXCEPTION_INT_DIVIDE_BY_ZERO` 614— `EXCEPTION_INT_OVERFLOW` 614— `EXCEPTION_INVALID_DISPOSITION` 613— `EXCEPTION_INVALID_HANDLE` 614— `EXCEPTION_NONCONTINUABLE_EXCEPTION` 613— `EXCEPTION_PRIV_INSTRUCTION` 613— `EXCEPTION_SINGLE_STEP` 614— `EXCEPTION_STACK_OVERFLOW` 613— `HWND_BROADCAST` 654— `MAXIMUM_PROCESSORS` 185— `MAXIMUM_SUSPEND_COUNT` 157— `MAXIMUM_WAIT_OBJECTS` 210— `STATUS_ACCESS_VIOLATION` 467— `STATUS_NO_MEMORY` 467— `STILL_ACTIVE` 80— `TLS_MINIMUM_AVAILABLE` 528

— класса приоритета 172

— локализации 21

— относительного приоритета потока 173

исключение 599

— C++ 642–644

— `EXCEPTION_STACK_OVERFLOW` 400

— аппаратное 599

— необработанное 623–625

— обработка

— — системой 601

— — структурная (SEH) 583–646

— обработчик завершения 584–588

— отключение вывода сообщений 626–620

— отладчик 630–630

— программное 620–622

— структурное в C++ 644–646

— фильтры и обработчики 599–622

— формат кодов 616

К

каталог, текущий 60–61

класс

— `CAddrWindow` 391— `CAddrWindowStorage` 391— `CAPIHook` 588–589— `CGuard` 257— `CInterlockedScalar` 259— `CInterlockedType` 258— `CJob` 115— `CMMFSParse` 452— `COptex` 246–247— `CQueue` 232–234— `CResGuard` 257–258— `CSE` 645— `CSomeClass` 469–470— `CSparseStream` 452— `CStopwatch` 162— `CSWMRG` 269— `CSystemInfo` 391— `CVMArray` 634–635— `CWhenZero` 259–261

ключ

— `/3GB` 317— `/ABOVENORMAL` 172— `/BELOWNORMAL` 172— `/Delay:unload` 509— `/DelayLoad` 507— `/DLL` 477— `/FIXED` 410, 522

- /Gf 66
- /GF 66
- /HEAP 462
- /HIGH 172
- /LARGEADDRESSAWARE 317–318
- /Lib:DelayImp.lib 507
- /NORMAL 172
- /REALTIME 172
- /SECTION 417
- /STACK 135, 398
- /SUBSYSTEM:CONSOLE 50–51
- /SUBSYSTEM:CONSOLE, 5.0 522
- /SUBSYSTEM:WINDOWS 50–51
- /SUBSYSTEM:WINDOWS, 5.0 522
- /SWAPRUN 324
- ключевое слово
 - *__except* 599
 - *__finally* 585
 - *__leave* 594
 - *__try* 585
- код подсистемы 616
- куча 461–473
 - выделение блока памяти 466–467
 - дополнительная 462–464
 - — создание 465–466
 - изменение размера блока 467–468
 - использование в программах на C++ 469–471
 - определение размера блока 468
 - освобождение блока 468
 - стандартная 461–462
 - уничтожение 468–469
- кэш-линия 193–194

M

макрос

- *_TEXT* 17
- API 713
- *CACHE_ALIGN* 194
- *chASSERT* 702
- *chBEGINTHREADEX* 144, 700–701
- *chDIMOF* 700
- *chHANDLE_DLGMSG* 702
- *chINRANGE* 700
- *chMB* 702
- *chSETDLGICONS* 703
- *chVERIFY* 702
- *Pragma Message* 700
- *UNALIGNED* 341
- *VER_SET_CONDITION* 63–64
- для дочерних элементов управления 712
- распаковщик сообщений 710–712
- модель аппаратного ввода 672
- модификатор
 - *__declspec(dllexport)* 483

- *__declspec(dllimport)* 483, 487
- *__declspec(thread)* 532
- *__unaligned* 340
- *allocate* 416
- *extern* 483
- *volatile* 196

O

обработка ошибок 2–10

- режимы 59–60

объект

- дублирование описателей 42–46
- критическая секция 197–206
 - — обработка ошибок 203–204
 - — спин-блокировка 191, 202
- наследование описателя 35–38
- — пример 68–69
- оптекс 245–256
- проверка на принадлежность ядру 31
- таймер (User) 226

объект ядра 28–46

- закрытие 33–34
- защита 29–31
- именование 38–42
- мьютекс 229–231
- — и критические секции 231
- — отказ 231
- ожидаемый таймер 221–224
- — APC-очередь 224–227
- побочные эффекты успешного ожидания 211–213
- проекция файла 411
- — закрытие 430–431
- — создание 423–426
- семафор 227–229
- — инверсный 257
- синхронизирующий 239–240
- событие 213–216
- совместное использование несколькими процессами 34–46
- создание 32–33
- состояние 207–208
- таблица описателей 31–33
- — флаги 35
- учет пользователей 29
- файл
 - — закрытие 430–431
 - — создание или открытие 422–423

окно, адресное 387

оператор

- *delete* 469, 471
- *new* 469, 471
- запятая 617

описатель 29

- преобразование из псевдоописателя 152–154
- экземпляра процесса 53, 54
- предыдущего 54
- определение версии системы 61–64
- отладка по запросу 625–626
- очередь
 - аппаратного ввода, системная (SHIQ) 671
 - виртуального ввода (VIQ) 671
 - сообщений потока 649

П

память

- гранулярность выделения 319
- динамически распределяемая; см. куча
- физическая 322–324
- — возврат 320, 373–374
- — передача 320, 370
- — сброс содержимого 383–384
- переменная окружения 56–59
- перехват API-вызовов 563–567
- позиция, кодовая 12–13
- поток 130–154
- завершение 137–140
- контекст 140–141
- локальная память (TLS) 527–532
- — динамическая 530–531
- — статическая 531–532
- необработанного ввода (RIT) 671–673
- определение периодов выполнения 160–162
- первичный 49
- переключение 159–160
- планирование 155–156
- привязка к процессорам 182–186
- — жесткая 182
- — нежесткая 182
- приостановка и возобновление 156–157
- пул 289–302
- — компоненты поддержки 289–290
- распределение процессорного времени 48–49
- синхронизация
 - — в пользовательском режиме 187–206
 - — в сценарии «один писатель/группа читателей» 267–269
 - — полезные средства 245–288
 - — с использованием объектов ядра 207–244
- создание и инициализация 140–142
- стек 398–408
- — особенности в Windows 98 401–402
- приложение
 - многопоточное 132
 - типы 49–50
- приоритет 167–171
- абстрагирование 168–169

- динамическое изменение 174–175
- классы 71, 168–170
- относительный 170–171, 174
- программирование 171–174
- уровень 167, 174
- программа-пример
 - AppInst 417–420
 - AWE 391–397
 - CopyData 666–669
 - Counter 306–312
 - DelayLoadApp 510–516
 - ErrorShow 7–10
 - FileRev 431–437
 - Handshake 216–221
 - InjLib 553–558
 - InterlockedType 260–267
 - JobLab 114–129
 - LastMsgBoxInfo 567–581
 - LISLab 680–692
 - LISWatch 692–697
 - MemReset 384–386
 - MMFShare 444–448
 - MMFSparse 450–460
 - Optex 247–256
 - ProcessInfo 83–97
 - Queue 232–239
 - SchedLab 176–182
 - SEHTerm 596–698
 - Spreadsheet 633–642
 - Summation 404–408
 - SWMRG 269–275
 - SysInfo 343–347
 - TimedMsgBox 296–298
 - VMAlloc 375–382
 - VMMap 360–367
 - VMStat 348–351
 - WaitForMultExp 277–288
- процесс 48–97
 - адресное пространство 314–319
 - — особенности в Windows 98 333
 - — отключение файла данных 429
 - — разделы 315
 - — регионы 319–320
 - — увеличение пользовательского раздела до 3 Гб (в 32-разрядной Windows 2000) 317
 - — уменьшение пользовательского раздела до 2 Гб (в 64-разрядной Windows 2000) 318
 - активный 175
 - — подстройка планировщика 175–176
 - включение в задание 107–108
 - дочерний 80–82
 - — обособленный 82
 - завершение 77–80
 - карта адресного пространства

- в Windows 2000 327–333
- в Windows 98 333–337
- командная строка 55–56
- перечисление 82–83
- приостановка и возобновление 157–159
- совместный доступ к данным через проецирование 443
- псевдоописатель 152
- преобразование в настоящий описатель 152–154

Р

раздел

- в EXE и DLL-файлах 415
- импорта 487, 564
- — отложенного 507
- переадресации 521
- раскрутка
 - глобальная 595, 605–609
 - — остановка 609
 - локальная 587
- регион
 - блоки 329
 - освобождение 373
 - передача физической памяти 320, 370
 - резервирование 319, 368–370
 - — с одновременной передачей физической памяти 370–371
 - типы 328

С

сообщение

- LB_GETTEXT 539–540
- LVM_GETITEM 539
- LVM_GETITEMPOSITION 539
- WM_COPYDATA 665–666
- WM_GETTEXT 664
- WM_KILLFOCUS 674
- WM_SETFOCUS 674
- WM_SETTEXT 664
- WM_SETTINGCHANGE 57
- алгоритм выборки из очереди потока 658–661
- оконное 648–670
- передача данных 664–666
- состояние ввода, локальное 673–675
- страница 319
 - атрибуты защиты 324–325
 - — изменение 382–383
 - номер фрейма 388
 - ошибка 322
- структура
 - CONTEXT 141, 162–167

- COPYDATASTRUCT 665–666
- CRITICAL_SECTION 198–200
- DelayLoadInfo 508
- DelayLoadProc 508
- ELEMENT 233
- EXCEPTION_POINTERS 616
- EXCEPTION_RECORD 618–619
- FILETIME 222–223
- IO_COUNTERS 110
- JOB_OBJECT_BASIC_LIMIT_INFORMATION 101–102
- JOBOBJECT_BASIC_ACCOUNTING_INFORMATION 109
- JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION 109
- JOBOBJECT_BASIC_LIMIT_INFORMATION 103
- JOBOBJECT_BASIC_PROCESS_ID_LIST 110
- JOBOBJECT_BASIC_UI_RESTRICTIONS 105
- JOBOBJECT_EXTENDED_LIMIT_INFORMATION 104
- JOBOBJECT_SECURITY_LIMIT_INFORMATION 106–107
- LARGE_INTEGER 222–223
- LV_ITEM 539
- MEMORY_BASIC_INFORMATION 352
- MEMORYSTATUS 348
- OSVERSIONINFOEX 62–64
- OVERLAPPED 301
- PROCESS_HEAP_ENTRY 473
- PROCESS_INFORMATION 75–76
- SECURITY_ATTRIBUTES 29–30, 67
- SHAREDINFO 247
- STARTUPINFO 71–75
- SYSTEM_INFO 342–343
- SYSTEMTIME 222
- THREADINFO 649, 651
- tiddata 146–147
- VMQUERY 353–354

У

уведомление

- DLL_PROCESS_ATTACH 498–499
- DLL_PROCESS_DETACH 499–501
- DLL_THREAD_ATTACH 501–502
- DLL_THREAD_DETACH 502
- задания 112–113

утилита

- AXPAlign 338–339
- Bind 524–525
- DIPS 539–539
- DumpBin 413, 516
- Error Lookup 5

- ImageCfg 185
- Microsoft Spy++ 105–106
- Rebase 523

Ф

файл

- APIHook.cpp 568
- APIHook.h 568
- CmnHdr.h 698–708
- DelayImp.lib 510
- Excpt.h 601
- Mtdll.h 146
- PDH.dll 83
- String.h 15–16
- TChar.h 16
- Threadex.c 147–148
- WinError.h 3–4, 616
- WinNT.h 157
- WinUser.h 653
- данных (проецируемый в память) 420–422
- проецируемый 409–460
- — когерентность 438–439
- — на физическую память из страничного файла 443–444
- — особенности на разных платформах 441–442
- — применение 409
- — с частичной передачей физической памяти 448–450
- разреженный 449–450
- страничный 320–324

флаг

- AC 338
- CREATE_BREAKAWAY_FROM_JOB 70
- CREATE_DEFAULT_ERROR_MODE 60, 70
- CREATE_FORCEDOS 70
- CREATE_NEW_CONSOLE 69
- CREATE_NEW_PROCESS_GROUP 70
- CREATE_NO_WINDOW 69
- CREATE_SEPARATE_WOW_VDM 70
- CREATE_SHARED_WOW_VDM 70
- CREATE_SUSPENDED 69, 107, 137, 141
- CREATE_UNICODE_ENVIRONMENT 70
- DEBUG_ONLY_THIS_PROCESS 69
- DEBUG_PROCESS 69
- DETACHED_PROCESS 69
- DONT_RESOLVE_DLL_REFERENCES 494
- DUPLICATE_CLOSE_SOURCE 43
- DUPLICATE_SAME_ACCESS 43
- EXCEPTION_NONCONTINUABLE 618, 621
- FILE_MAP_READ 30
- FORMAT_MESSAGE_ALLOCATE_BUFFER 7
- FORMAT_MESSAGE_FROM_SYSTEM 7
- FREE 528

- GENERIC_READ 423
- GENERIC_WRITE 423
- HANDLE_FLAG_INHERIT 38
- HANDLE_FLAG_PROTECT_FROM_CLOSE 38
- HEAP_GENERATE_EXCEPTIONS 465–467, 620
- HEAP_NO_SERIALIZE 465–467, 470
- HEAP_REALLOC_IN_PLACE_ONLY 468
- HEAP_ZERO_MEMORY 467
- IMAGE_FILE_RELOCS_STRIPPED 522
- INUSE 528
- ISMEX_CALLBACK 656
- ISMEX_NOTIFY 656
- ISMEX_REPLIED 656
- ISMEX_SEND 656
- JOB_OBJECT_LIMIT_ACTIVE_PROCESS 102
- JOB_OBJECT_LIMIT_AFFINITY 102–103
- JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION 104
- JOB_OBJECT_LIMIT_JOB_TIME 102–103
- JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME 103
- JOB_OBJECT_LIMIT_PRIORITY_CLASS 103
- JOB_OBJECT_LIMIT_PROCESS_TIME 102
- JOB_OBJECT_LIMIT_SCHEDULING_CLASS 103
- JOB_OBJECT_LIMIT_WORKINGSET 102
- JOB_OBJECT_UILIMIT_DESKTOP 105
- JOB_OBJECT_UILIMIT_DISPLAYSETTINGS 105
- JOB_OBJECT_UILIMIT_EXITWINDOWS 105
- JOB_OBJECT_UILIMIT_GLOBALATOMS 105
- JOB_OBJECT_UILIMIT_HANDLES 105
- JOB_OBJECT_UILIMIT_READCLIPBOARD 105
- JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS 105
- JOB_OBJECT_UILIMIT_WRITECLIPBOARD 105
- KEY_ALL_ACCESS 31
- KEY_QUERY_VALUE 30
- LOAD_LIBRARY_AS_DATAFILE 494
- LOAD_WITH_ALTERED_SEARCH_PATH 494
- MEM_COMMIT 370
- MEM_DECOMMIT 373
- MEM_PHYSICAL 388
- MEM_RELEASE 373
- MEM_RESERVE 388
- MEM_RESET 383–384
- MEM_TOP_DOWN 369
- MWMO_ALERTABLE 662
- MWMO_INPUTAVAILABLE 662–663
- MWMO_WAITALL 662
- NORM_IGNORECASE 21
- NORM_IGNOREKANATYPE 21
- NORM_IGNORENONSPACE 21
- NORM_IGNORESYMBOLS 21
- NORM_IGNOREWIDTH 21
- PAGE_GUARD 326, 398, 400
- PAGE_NOCACHE 326

- PAGE_WRITECOMBINE 326
- QS_PAINT 658
- QS_POSTMESSAGE 658
- QS_QUIT 658
- QS_SENDMESSAGE 652, 658
- QS_TIMER 658
- SEC_IMAGE 425
- SEC_NOCACHE 424
- SEM_FAILCRITICALERRORS 59
- SEM_NOALIGNMENTFAULTEXCEPT 59, 339
- SEM_NOGPFAULTERRORBOX 59
- SEM_NOOPENFILEERRORBOX 59
- SMTO_ABORTIFHUNG 653
- SMTO_BLOCK 653
- SMTO_NORMAL 653
- SMTO_NOTIMEOUTIFNOTHUNG 653
- SORT_STRINGSORT 21
- STARTF_FORCEOFFFEEDBACK 74
- STARTF_FORCEONFEEDBACK 74
- STARTF_RUN_FULLSCREEN 74
- STARTF_USECOUNTCHARS 73
- STARTF_USEFILLATTRIBUTE 73
- STARTF_USEPOSITION 73
- STARTF_USESHOWWINDOW 73
- STARTF_USESIZE 73
- STARTF_USESTDHANDLES 73
- WT_EXECUTEINPERSISTENTTHREAD 294
- WT_EXECUTEDEFAULT 292–293
- WT_EXECUTEINIOTHREAD 292–293, 299
- WT_EXECUTEINPERSISTENTTTHREAD 299
- WT_EXECUTEINTIMERTHREAD 294
- WT_EXECUTEINWAITTHREAD 299
- WT_EXECUTEONLONGFUNCTION 292, 294, 299
- WT_EXECUTEONLYONCE 299–300
- атрибута защиты 326
- наследования, связанный с описателем 37
- состояния очереди 657–658
- функция
 - *_delayLoadHelper* 507–509
 - *_FUnloadDelayLoadedDLL* 509
 - *_beginthread* 151
 - *_beginthreadex* 144–146, 150–151
 - *_DllMainCRTStartup* 505
 - *_endthread* 151
 - *_endthreadex* 148–149, 152
 - *_getptd* 149
 - *_threadstartex* 147–148
 - *AbnormalTermination* 595
 - *AllocateUserPhysicalPages* 388–389
 - *AllowSetForegroundWindow* 676
 - APC 224–225
 - *AssignProcessToJobObject* 108
 - *AttachThreadInput* 678–679
 - *BaseProcessStart* 142, 623
 - *BaseThreadStart* 141–142, 623
 - *BindImageEx* 524–525
 - *BindIoCompletionCallback* 301
 - *BringWindowToTop* 675
 - *CallWindowProcA* 670
 - *CallWindowProcW* 670
 - *CancelWaitableTimer* 224
 - *ChangeTimerQueueTimer* 295
 - *CharLower* 21–22
 - *CharLowerBuff* 22
 - *CharNext* 12
 - *CharPrev* 12
 - *CharUpper* 21
 - *CharUpperBuff* 22
 - *ClipCursor* 677
 - *CloseHandle* 33–34
 - *CommandLineToArgvW* 55
 - *CompareString* 21
 - *ConvertThreadToFiber* 304
 - *CreateEvent* 213
 - *CreateFiber* 304
 - *CreateFile* 422–423
 - *CreateFileMapping* 423–426
 - *CreateJobObject* 100
 - *CreateMutex* 40, 229
 - *CreateProcess* 64–76, 409
 - — поиск заданного файла 66
 - *CreateRemoteThread* 549–551
 - *CreateSemaphore* 228
 - *CreateThread* 134–137
 - *CreateTimerQueue* 293
 - *CreateTimerQueueTimer* 293
 - *CreateWaitableTimer* 221
 - *DebugActiveProcess* 627
 - *DebugBreak* 248
 - *DeleteCriticalSection* 200
 - *DeleteFiber* 305
 - *DeleteTimerQueueEx* 295
 - *DeleteTimerQueueTimer* 294–295
 - *DisableThreadLibraryCalls* 504, 506
 - *DllMain* 497–498
 - — библиотека C/C++ 505–506
 - — упорядочение вызовов 503–505
 - *DuplicateHandle* 43–45, 153–154
 - *EnterCriticalSection* 199–201
 - *ExitProcess* 77–79
 - *ExitThread* 138
 - *ExpandEnvironmentStrings* 58
 - *FlushViewOfFile* 429
 - *FormatMessage* 5
 - *FreeEnvironmentStrings* 71
 - *FreeLibrary* 496
 - *FreeLibraryAndExitThread* 495
 - *FreeUserPhysicalPages* 390

- *GetActiveWindow* 675
- *GetAsyncKeyState* 676
- *GetCommandLine* 55
- *GetCurrentDirectory* 60
- *GetCurrentFiber* 305
- *GetCurrentProcess* 152
- *GetCurrentThread* 152
- *GetEnvironmentStrings* 71
- *GetEnvironmentVariable* 58
- *GetExceptionCode* 612–615
- *GetExceptionInformation* 616–617, 619
- *GetExitCodeProcess* 80
- *GetExitCodeThread* 140
- *GetFiberData* 305
- *GetFileSize* 431
- *GetForegroundWindow* 675
- *GetFullPathName* 61
- *GetHandleInformation* 38
- *GetKeyState* 676
- *GetLastError* 2–4
- *GetMessage* 540
- *GetModuleFileName* 496
- *GetModuleHandle* 54, 496
- *GetPriorityClass* 172
- *GetProcAddress* 496–497
- *GetProcessAffinityMask* 183
- *GetProcessHeap* 462
- *GetProcessHeaps* 472
- *GetProcessIoCounters* 110
- *GetProcessPriorityBoost* 175
- *GetProcessTimes* 152, 161
- *GetQueuedCompletionStatus* 113
- *GetQueueStatus* 657
- *GetStartupInfo* 75
- *GetSystemInfo* 342
- *GetThreadContext* 165–166
- *GetThreadLocale* 21
- *GetThreadPriority* 173–174
- *GetThreadPriorityBoost* 175
- *GetThreadTimes* 152, 160–161
- *GetVersion* 61
- *GetVersionEx* 61–62
- *GetWindowThreadProcessId* 650–651
- *GlobalMemoryStatus* 347–348
- *GlobalMemoryStatusEx* 348
- *HeapAlloc* 466–467
- *HeapCompact* 472
- *HeapCreate* 465–466
- *HeapDestroy* 468–469
- *HeapFree* 468
- *HeapLock* 473
- *HeapReAlloc* 467–468
- *HeapSize* 468
- *HeapUnlock* 473
- *HeapValidate* 472
- *HeapWalk* 473
- *InitializeCriticalSection* 200
- *InitializeCriticalSectionAndSpinCount* 202–203
- *InSendMessage* 656
- *InSendMessageEx* 656
- *InterlockedCompareExchange* 192
- *InterlockedCompareExchangePointer* 192
- *InterlockedDecrement* 193
- *InterlockedExchange* 190
- *InterlockedExchangeAdd* 189
- *InterlockedExchangePointer* 190
- *InterlockedIncrement* 193
- *IsCharAlpha* 22
- *IsCharAlphaNumeric* 22
- *IsCharLower* 22
- *IsCharUpper* 22
- *IsDBCSLeadByte* 12
- *IsTextUnicode* 23–24
- *IsWindowUnicode* 669
- *LeaveCriticalSection* 199, 202
- *LoadIcon* 53
- *LoadLibrary* 410, 492, 495, 519, 550
- *LoadLibraryEx* 492–495, 519
- *LockSetForegroundWindow* 676
- *MapUserPhysicalPages* 389
- *MapViewOfFile* 426–429, 441–442
- *MapViewOfFileEx* 439–440
- *MoveMemory* 430
- *MsgWaitForMultipleObjects* 242, 661–662
- *MsgWaitForMultipleObjectsEx* 242, 661–663
- *MultiByteToWideChar* 24–25
- *OpenEvent* 214
- *OpenJobObject* 100
- *OpenMutex* 230
- *OpenProcess* 553
- *OpenSemaphore* 228
- *OpenThread* 158
- *OpenWaitableTimer* 221
- *OverlappedCompletionRoutine* 301
- *PostMessage* 619
- *PostQuitMessage* 651
- *PostThreadMessage* 540, 650–651
- *PulseEvent* 216
- *QueryInformationJobObject* 107
- *QueryPerformanceCounter* 162
- *QueryPerformanceFrequency* 162
- *QueueUserWorkItem* 290
- *RaiseException* 621
- *ReadProcessMemory* 552
- *ReBaseImage* 523
- *RegisterWaitForSingleObject* 298–299
- *RegOpenKeyEx* 30–31
- *ReleaseMutex* 230

- *ReleaseSemaphore* 228–229
- *ReplaceLATEntryInOneMod* 564–566
- *ReplyMessage* 655
- *ResetEvent* 214
- *ResultCallback* 654
- *ResumeThread* 157
- *RobustHowManyToken* 604–605
- *RobustMemDup* 605
- *RobustStrCpy* 603
- *SendMessage* 651–652, 664–665
- *SendMessageCallback* 654
- *SendMessageTimeout* 653
- *SendNotifyMessage* 655
- *SetActiveWindow* 674
- *SetCriticalSectionSpinCount* 203
- *SetCurrentDirectory* 60
- *SetEndOfFile* 432–433
- *SetEnvironmentVariable* 59
- *SetErrorMode* 59, 339
- *SetEvent* 214
- *SetFilePointer* 432
- *SetForegroundWindow* 675–676
- *SetHandleInformation* 37–38
- *SetInformationJobObject* 101
- *SetLastError* 6
- *SetPriorityClass* 172
- *SetProcessAffinityMask* 183
- *SetProcessPriorityBoost* 175
- *SetThreadAffinityMask* 183–184
- *SetThreadContext* 166
- *SetThreadIdealProcessor* 185
- *SetThreadPriority* 173
- *SetThreadPriorityBoost* 175
- *SetUnhandledExceptionFilter* 627–629
- *SetWaitableTimer* 221, 223–224
- *SetWindowLongPtr* 534–535
- *SetWindowLongPtrA* 619
- *SetWindowLongPtrW* 670
- *SetWindowPos* 675
- *SetWindowsHookEx* 537
- *SignalObjectAndWait* 242–244
- *Sleep* 159
- *StackCheck* 403–404
- *StartRestrictedProcess* 98–100
- *StatusRoutine* 525
- *StringReverse* 26–27
- *strlen* 12
- *SuspendProcess* 157–158
- *SuspendThread* 157
- *SwitchToFiber* 304–305
- *SwitchToThread* 159–160
- *TerminateJobObject* 108
- *TerminateProcess* 79
- *TerminateThread* 138–139
- *TlsAlloc* 528–529
- *TlsFree* 529
- *TlsGetValue* 529
- *TlsSetValue* 529
- *ToolHelp* (вспомогательные информационные) 76
- *TranslateMessage* 660
- *TryEnterCriticalSection* 201–202
- *UnhandledExceptionFilter* 623–630
- *UnhookWindowsHookEx* 538
- *Unicode- и ANSI-версии* 18–19
- *UnmapViewOfFile* 429–430
- *UnregisterWaitEx* 300
- *UserHandleGrantAccess* 106
- *VerifyVersionInfo* 63–64
- *VirtualAlloc* 368–371, 449
- *VirtualAllocEx* 551
- *VirtualFree* 373
- *VirtualFreeEx* 552
- *VirtualProtect* 382–383
- *VirtualQuery* 351–352
- *VirtualQueryEx* 351–352
- *VMQuery* 353–359
- *WaitForDebugEvent* 242
- *WaitForInputIdle* 241
- *WaitForMultipleExpressions* 275–280
- *WaitForMultipleObjects* 210–211
- *WaitForSingleObject* 81, 209–210
- *WaitOrTimerCallback* 293
- *WaitOrTimerCallbackFunc* 299
- *WideCharToMultiByte* 25
- *WorkItemFunc* 291
- *WriteProcessMemory* 318, 552
- *ZeroMemory* 385
- *волокна* 304
- *входная* 50–52
- *запись о переадресации вызова* 516–517
- *значение, свидетельствующее об ошибке* 2
- *переадресованная* 516
- *потока* 133–134
- *семейства*
- *— Interlocked* 187–193
- *— printf* 22
- *стартовая (из библиотеки C/C++)* 50–53



Об авторе

Джеффри Рихтер ведет семинары по программированию (<http://www.SolSem.com>) и консультирует разработчиков программного обеспечения по различным вопросам (<http://www.JeffreyRichter.com>). В число его клиентов входят такие компании, как Allen-Bradley, AT&T, Caterpillar, Digital, DreamWorks, GE Medical Systems, Hewlett-Packard, IBM, Intel, Intuit, Microsoft, Pitney Bowes, Sybase, Tandem и Unisys.

Джефф регулярно выступает на отраслевых конференциях, в том числе на Software Development, COMDEX, WinDev (при Бостонском Университете), Microsoft Professional Developer's Conference и Tech-Ed. Кроме того, Джефф — один из «пишущих» редакторов журнала *Microsoft Systems Journal*, в котором ведет колонку «Вопросы и ответы по Win32» и публикует свои статьи.

Сейчас Джефф живет в Белльвю (штат Вашингтон). Его хобби: полеты на вертолете, фокусы и игра на барабанах. Ему очень нравятся телесериал *The Simpsons*, классический рок и джаз в стиле «фьюжн».

ЛИЦЕНЗИОННОЕ СОГЛАШЕНИЕ MICROSOFT

(прилагаемый к книге компакт-диск)

ЭТО ВАЖНО – ПРОЧИТАЙТЕ ВНИМАТЕЛЬНО. Настоящее лицензионное соглашение (далее «Соглашение») является юридическим документом, оно заключается между Вами (физическим или юридическим лицом) и Microsoft Corporation (далее «корпорация Microsoft») на указанный выше продукт Microsoft, который включает программное обеспечение и может включать сопутствующие мультимедийные и печатные материалы, а также электронную документацию (далее «Программный Продукт»). Любой компонент, входящий в Программный Продукт, который сопровождается отдельным Соглашением, подпадает под действие именно того Соглашения, а не условий, изложенных ниже. Установка, копирование или иное использование данного Программного Продукта означает принятие Вами данного Соглашения. Если Вы не принимаете его условия, то не имеете права устанавливать, копировать или как-то иначе использовать этот Программный Продукт.

ЛИЦЕНЗИЯ НА ПРОГРАММНЫЙ ПРОДУКТ

Программный Продукт защищен законами Соединенных Штатов по авторскому праву и международными договорами по авторскому праву, а также другими законами и договорами по правам на интеллектуальную собственность.

1. ОБЪЕМ ЛИЦЕНЗИИ. Настоящее Соглашение дает Вам право:

- а) **Программный продукт.** Вы можете установить и использовать одну копию Программного Продукта на одном компьютере. Основной пользователь компьютера, на котором установлен данный Программный Продукт, может сделать только для себя вторую копию и использовать ее на портативном компьютере.
- б) **Хранение или использование в сети.** Вы можете также скопировать или установить экземпляр Программного Продукта на устройстве хранения, например на сетевом сервере, исключительно для установки или запуска данного Программного Продукта на других компьютерах в своей внутренней сети, но тогда Вы должны приобрести лицензии на каждый такой компьютер. Лицензию на данный Программный продукт нельзя использовать совместно или одновременно на других компьютерах.
- с) **License Pak.** Если Вы купили эту лицензию в составе Microsoft License Pak, можете сделать ряд дополнительных копий программного обеспечения, входящего в данный Программный Продукт, и использовать каждую копию так, как было описано выше. Кроме того, Вы получаете право сделать соответствующее число вторичных копий для портативного компьютера в целях, также оговоренных выше.
- д) **Примеры кода.** Это относится исключительно к отдельным частям Программного Продукта, заявленным как примеры кода (далее «Примеры»), если таковые входят в состав Программного Продукта.
 - i) **Использование и модификация.** Microsoft дает Вам право использовать и модифицировать исходный код Примеров при условии соблюдения пункта (d)(iii) ниже. Вы не имеете права распространять в виде исходного кода ни Примеры, ни их модифицированную версию.
 - ii) **Распространяемые файлы.** При соблюдении пункта (d)(iii) Microsoft дает Вам право на свободное от отчислений копирование и распространение в виде объектного кода Примеров или их модифицированной версии, кроме тех частей (или их модифицированных версий), которые оговорены в файле Readme, относящемся к данному Программному Продукту, как не подлежащие распространению.
 - iii) **Требования к распространению файлов.** Вы можете распространять файлы, разрешенные к распространению, при условии, что: а) распространяете их в виде объектного кода только в сочетании со своим приложением и как его часть; б) не используете название, эмблему или товарные знаки Microsoft для продвижения своего приложения; в) включаете имеющуюся в Программном Продукте ссылку на авторские права в состав этикетки и заставки своего приложения; г) согласны освободить от ответственности и взять на себя защиту корпорации Microsoft от любых претензий или преследований по закону, включая судебные издержки, если таковые возникнут в результате использования или распространения Вашего приложения; и д) не допускаете дальнейшего распространения конечным пользователем своего приложения. По поводу отчислений и других условий лицензии применительно к иным видам использования или распространения распространяемых файлов обращайтесь в Microsoft.

2. ПРОЧИЕ ПРАВА И ОГРАНИЧЕНИЯ

- **Ограничения на реконструкцию, декомпиляцию и дизассемблирование.** Вы не имеете права реконструировать, декомпилировать или дизассемблировать данный Программный Продукт, кроме того случая, когда такая деятельность (только в той мере, которая необходима) явно разрешается соответствующим законом, несмотря на это ограничение.
- **Разделение компонентов.** Данный Программный Продукт лицензируется как единый продукт. Его компоненты нельзя отделять друг от друга для использования более чем на одном компьютере.
- **Аренда.** Данный Программный Продукт нельзя сдавать в прокат, передавать во временное пользование или уступать для использования в иных целях.
- **Услуги по технической поддержке.** Microsoft может (но не обязана) предоставить Вам услуги по технической поддержке данного Программного Продукта (далее «Услуги»). Предоставление Услуг регулируется соответствующими правилами и программами Microsoft, описанными в руководстве пользователя, электронной документации и/или других материалах, публикуемых Microsoft. Любой дополнительный программный код, предоставленный в рамках Услуг, следует считать частью данного Программного Продукта и подпадающим под действие настоящего Соглашения. Что касается технической информации, предоставляемой Вами корпорации Microsoft при использовании ее Услуг, то Microsoft может задерживать эту информацию в деловых целях, в том числе для технической поддержки продукта и разработки. Используя такую техническую информацию, Microsoft не будет ссылаться на Вас.
- **Передача прав на программное обеспечение.** Вы можете безвозвратно уступить все права, регулируемые настоящим Соглашением, при условии, что не оставите себе никаких копий, передадите все составные части данного Программного Продукта (включая компоненты, мультимедийные и печатные материалы, любые обновления, Соглашение и сертификат подлинности, если таковой имеется) и принимающая сторона согласится с условиями настоящего Соглашения.
- **Прекращение действия Соглашения.** Без ущерба для любых других прав Microsoft может прекратить действие настоящего Соглашения, если Вы нарушите его условия. В этом случае Вы должны будете уничтожить все копии данного Программного Продукта вместе со всеми его компонентами.

3. **АВТОРСКОЕ ПРАВО.** Все авторские права и право собственности на Программный Продукт (в том числе любые изображения, фотографии, анимации, видео, аудио, музыку, текст, примеры кода, распространяемые файлы и апплеты, включенные в состав Программного Продукта) и любые его копии принадлежат корпорации Microsoft или ее поставщикам. Программный Продукт охраняется законодательством об авторских правах и положениями международных договоров. Таким образом, Вы должны обращаться с данным Программным Продуктом, как с любым другим материалом, охраняемым авторскими правами, **с тем исключением**, что Вы можете установить Программный Продукт на один компьютер при условии, что храните оригинал исключительно как резервную или архивную копию. Копирование печатных материалов, поставляемых вместе с Программным Продуктом, запрещается.

ОГРАНИЧЕНИЕ ГАРАНТИИ

ДАННЫЙ ПРОГРАММНЫЙ ПРОДУКТ (ВКЛЮЧАЯ ИНСТРУКЦИИ ПО ЕГО ИСПОЛЬЗОВАНИЮ) ПРЕДОСТАВЛЯЕТСЯ БЕЗ КАКОЙ-ЛИБО ГАРАНТИИ. КОРПОРАЦИЯ MICROSOFT СНИМАЕТ С СЕБЯ ЛЮБУЮ ВОЗМОЖНУЮ ОТВЕТСТВЕННОСТЬ, В ТОМ ЧИСЛЕ ОТВЕТСТВЕННОСТЬ ЗА КОММЕРЧЕСКУЮ ЦЕННОСТЬ ИЛИ СООТВЕТСТВИЕ ОПРЕДЕЛЕННЫМ ЦЕЛЯМ. ВСЕ РИСК ПО ИСПОЛЬЗОВАНИЮ ИЛИ РАБОТЕ С ПРОГРАММНЫМ ПРОДУКТОМ ЛОЖИТСЯ НА ВАС.

НИ ПРИ КАКИХ ОБСТОЯТЕЛЬСТВАХ КОРПОРАЦИЯ MICROSOFT, ЕЕ РАЗРАБОТЧИКИ, А ТАКЖЕ ВСЕ, ЗАНЯТЫЕ В СОЗДАНИИ, ПРОИЗВОДСТВЕ И РАСПРОСТРАНЕНИИ ДАННОГО ПРОГРАММНОГО ПРОДУКТА, НЕ НЕСУТ ОТВЕТСТВЕННОСТИ ЗА КАКОЙ-ЛИБО УЩЕРБ (ВКЛЮЧАЯ ВСЕ, БЕЗ ИСКЛЮЧЕНИЯ, СЛУЧАИ УПУЩЕННОЙ ВЫГОДЫ, НАРУШЕНИЯ ХОЗЯЙСТВЕННОЙ ДЕЯТЕЛЬНОСТИ, ПОТЕРИ ИНФОРМАЦИИ ИЛИ ДРУГИХ УБЫТКОВ) ВСЛЕДСТВИЕ ИСПОЛЬЗОВАНИЯ ИЛИ НЕВОЗМОЖНОСТИ ИСПОЛЬЗОВАНИЯ ДАННОГО ПРОГРАММНОГО ПРОДУКТА ИЛИ ДОКУМЕНТАЦИИ, ДАЖЕ ЕСЛИ КОРПОРАЦИЯ MICROSOFT БЫЛА ИЗВЕЩЕНА О ВОЗМОЖНОСТИ ТАКИХ ПОТЕРЬ, ТАК КАК В НЕКОТОРЫХ СТРАНАХ НЕ РАЗРЕШЕНО ИСКЛЮЧЕНИЕ ИЛИ ОГРАНИЧЕНИЕ ОТВЕТСТВЕННОСТИ ЗА НЕПРЕДНАМЕРЕННЫЙ УЩЕРБ, УКАЗАННОЕ ОГРАНИЧЕНИЕ МОЖЕТ ВАС НЕ КОСНУТЬСЯ.

РАЗНОЕ

Настоящее Соглашение регулируется законодательством штата Вашингтон (США), кроме случаев (и лишь в той мере, насколько это необходимо) исключительной юрисдикции того государства, на территории которого используется Программный Продукт. Если у Вас возникли какие-либо вопросы, касающиеся настоящего Соглашения, или если Вы желаете связаться с Microsoft по любой другой причине, пожалуйста, обращайтесь в местное представительство Microsoft или пишите по адресу: Microsoft Sales Information Center, One Microsoft Way, Redmond, WA 98052-6399.