Оглавление

ГЛАВА 10

Синхронный и асинхронный ввод-вывод на устройствах

Сложно переоценить важность этой главы, посвященной технологиям Windows, позволяющим создавать быстрые, масштабируемые и надежные приложения. Масштабируемым считается приложение, способное выполнить множество одновременных операций не менее эффективно, чем обработку незначительной вычислительной нагрузки. В случае службы Windows такими операциями являются клиентские запросы, время поступления которых предсказать невозможно, как и количество вычислительных ресурсов, необходимое для обработки запросов. Как правило, эти запросы приходят с устройств ввода-вывода, таких как сетевые платы, а в обработке этих запросов часто участвуют другие устройства, такие как диски.

В Windows-приложениях удобнее всего распределять нагрузку между потоками. Каждый поток приписан к определенному процессору, что позволяет исполнять на многопроцессорном компьютере сразу несколько операций, что повышает общую производительность. Сгенерировав синхронный запрос ввода-вывода на устройстве, поток приостанавливается до завершения обработки этого запроса. При этом страдает производительность, поскольку в данном состоянии поток неспособен выполнять полезную работу, такую как обработка клиентских запросов. Короче говоря, ваша задача — заставить потоки выполнять как можно больше полезных операций, избегая их блокировки.

Чтобы потоки не простаивали, им необходимо взаимодействовать, обмениваться информацией о выполняемых ими операциях. Майкрософт затратила годы работы на исследования и разработки в этой области. Результат этих усилий — весьма совершенный механизм взаимодействия потоков, получивший название *порт завершения ввода-вывода* (I/O completion port). С помощью данного механизма разработчики могут создавать высокопроизводительные масштабируемые приложения. Использование портов завер-

шения ввода-вывода позволяет добиться феноменальной производительности приложений, поскольку оно освобождает их от необходимости ждать отклика устройств ввода-вывода.

Изначально порты завершения ввода предназначены для обработки ввода-вывода на устройствах, но со временем в операционных системах Майкрософт появлялось все больше и больше механизмов, использующих модель портов завершения ввода-вывода. Примером может быть объект ядра «задание», отслеживающий включенные в задание процессы и отправляющий уведомления о событиях в порты завершения ввода-вывода. Совместную работу объектов-заданий и портов завершения ввода-вывода иллюстрирует программа-пример Job Lab из главы 5.

Несмотря на свой многолетний опыт разработки для Windows, я постоянно нахожу новые применения для портов завершения ввода-вывода и считаю, что каждый Windows-разработчик должен досконально разбираться в работе этого механизма. В этой главе я демонстрирую работу портов завершения ввода-вывода при работе с устройствами, но этим их сфера применения далеко не исчерпывается. Проще говоря, порты завершения ввода-вывода — это отличное средства для организации взаимодействия между потоками с безграничной областью применения.

После такой патетики легко заключить, что я — большой фанат использования портов завершения ввода-вывода. Надеюсь, дочитав эту главу до конца, вы тоже поклонником этого средства. Однако я немного отложу детальный разбор портов завершения ввода-вывода, чтобы рассказать о том, какие средства ввода-вывода на устройствах в Windows были исходно доступны разработчикам, чтобы вы глубже осознали все достоинства этого механизма.

Открытие и закрытие устройств

Одна из сильных сторон Windows состоит в разнообразии устройств, поддерживаемых этой операционной системой. Устройством в контексте этого раздела мы будем называть любую сущность, взаимодействующую с системой. Типичные устройства с указанием их применения перечислены в таблице 10-1.

Табл. 10-1. Типичные устройства и их применение

Устройство	Применение
Файл	Постоянное хранилище для любых данных
Каталог	Назначение атрибутов и параметров сжатия файлов
Логический диск	Форматирование дисков
Физический диск	Доступ к таблице разделов
Последовательный порт	Передача данных по телефонным линиям
Параллельный порт	Передача данных на принтер
Почтовый ящик	Передача данных множеству адресатов, обычно Windows-компьютерам по сети

Табл. 10-1. *(окончание)*

Устройство	Применение
Именованный канал	Передача данных отдельному адресату, обычно Windows-компьютеру по сети
Неименованный канал	Обмен данными между парой адресатов на одном и том же компьютере (но ни в коем случае не по сети)
Сокет	Передача данных (потоковая или в виде дейтаграмм, обычно через сеть) на другие поддерживающие сокеты компьютеры (под управлением Windows или других операционных систем), обычно осуществляется через сеть
Консоль	Экранный буфер для текстового окна

В этой главе пойдет речь о том, как наладить взаимодействие потоков с перечисленными выше устройствами так, чтобы потокам не пришлось дожидаться их отклика. Windows пытается по максимуму скрыть от разработчиков различия между устройствами. Другими словами, открыв любое устройство, вы сможете использовать для чтения и записи данных одни и те же Windows-функции. Хотя некоторые функции чтения-записи работают на всех устройствах, устройства все различаются. Так, скорость передачи данных через последовательный порт задают в бод, но этот параметр не имеет никакого смысла для именованных каналов, применяемых для взаимодействия компьютеров сеть (либо компонентов локальной системы) настройка скорости передачи в бод. Я не буду обсуждать все тонкости, которыми устройства отличаются друг от друга, но поподробнее остановлюсь на файлах, поскольку с ними приходится работать очень часто. Для выполнения любых операций ввода-вывода прежде всего необходимо открыть устройство и получить его описатель. Способ получения описателя зависит от типа устройства. Функции, позволяющие открывать различные устройства, перечислены в таблице 10-2.

Табл. 10-2. Функции, открывающие различные устройства

Устройство	Функция
Файл	*CreateFile* (значение *pszName* — путь или UNC-путь)
Каталог	*CreateFile* (значение *pszName* — путь или UNC-путь к каталогу). Windows позволяет открывать каталоги, если при вызове *CreateFile* установлен флаг FILE_FLAG_BACKUP_SEMANTICS. Открыв каталог, можно изменять его атрибуты (например, сделать его скрытым) и временную отметку
Логический диск	*CreateFile* (значение pszName — "\\.\x"). В Windows можно открывать логические диски, ссылаясь на них в формате "\\.\x", где x — буква диска. Например, открыть диск A можно с помощью ссылки \\.\A: Открыв диск, вы сможете его отформатировать либо определить размер носителя

Табл. 10-2. *(окончание)*

Устройство	Функция
Физический диск	*CreateFile* (значение *pszName* — "\\.\PHYSICALDRIVEx"). В Windows можно открывать физические диски, используя ссылку в формате "\\.\PHYSICALDRIVEx", где x — номер физического диска. Например, прочитать физические сектора первого жесткого диска на компьютере пользователя позволит ссылка "\\.\PHYSICALDRIVE0". Открыв физический диск, вы сможете напрямую обращаться к его таблице разделов, правда, это опасно: некорректная модификация содержимого диска сделает его недоступным для операционной системы
Последовательный порт	*CreateFile* (значение *pszName* — "COMx")
Параллельный порт	*CreateFile* (значение *pszName* — "LPTx")
Сервер почтового ящика	*CreateMailslot* (значение pszName — "\\.\mailslot\имя_ящика")
Клиент почтового ящика	*CreateFile* (значение *pszName* — "\\имя_сервера\mailslot\имя_ящика")
Сервер именованного канала	*CreateNamedPipe* (значение *pszName* — "\\.\pipe\имя_канала")
Клиент именованного канала	*CreateFile* (значение *pszName* — "\\имя_сервера\гяре\имя_ канала")
Неименованный канал	*CreatePipe* (для клиента и сервера)
Сокет	*socket*, *accept* и *AcceptEx*
Консоль	*CreateConsoleScreenBuffer* и *GetStdHandle*

Все эти функции возвращают описатели, идентифицирующие устройства. Этот описатель передают различным функциям для взаимодействия с устройствами. Например, функция *SetCommConfig* устанавливается скорость последовательного порта (в бод):

```
BOOL SetCommConfig(
    HANDLE       hCommDev,
    LPCOMMCONFIG pCC,
    DWORD        dwSize);
```

а функция *SetMailslotInfo* — длительность ожидания чтения данных:

```
BOOL SetMailslotInfo(
    HANDLE hMailslot,
    DWORD  dwReadTimeout);
```

Как видно, первым аргументом этих функций является описатель устройства. Закончив работу с устройством, описатель необходимо закрыть.

Для большинства устройств это можно сделать вызовом весьма популярной функции *CloseHandle*:

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

Однако в случае сокета для этого необходимо вызывать функцию *closesocket*:

```
int closesocket(SOCKET s);
```

Кроме того, обладая описателем, можно узнать тип соответствующего устройства, вызвав *GetFileType*:

```
DWORD GetFileType(HANDLE hDevice);
```

Для этого достаточно передать функции *GetFileType* описатель устройства, и она вернет одно из значений, перечисленных в таблице 10-3.

Табл. 10-3. Значения, возвращаемые функцией GetFileType

Значение	Описание
FILE_TYPE_UNKNOWN	Тип заданного файла неизвестен
FILE_TYPE_DISK	Дисковый файл
FILE_TYPE_CHAR	Текстовый файл, консоль или устройство, подключенное к порту LPT
FILE_TYPE_PIPE	Именованный или неименованный канал

Близкое знакомство с функцией *CreateFile*

Естественно, функция *CreateFile* способна создавать и открывать файлы, но этим ее возможности далеко не исчерпываются — она способна открывать и множество других устройств:

```
HANDLE CreateFile(
    PCTSTR pszName,
    DWORD  dwDesiredAccess,
    DWORD  dwShareMode,
    PSECURITY_ATTRIBUTES psa,
    DWORD  dwCreationDisposition,
    DWORD  dwFlagsAndAttributes,
    HANDLE hFileTemplate);
```

Как видно из этого листинга, *CreateFile* принимает изрядное число параметров, что обеспечивает ей немалую гибкость при работе с устройствами. Давайте разберем эти параметры подробно.

При вызове *CreateFile* параметр *pszName* определяет тип и экземпляр устройства. Параметр *dwDesiredAccess* указывает, как должен осуществляться обмен данными с устройством, его значения перечислены в табл. 10-4. Некоторые устройства поддерживают дополнительные флаги, управля-

ющие доступом. Подробнее о каждом из этих флагов см. в документации Platform SDK.

Табл. 10-4. Значения параметра dwDesiredAccess функции *CreateFile*

Значение	Описание
0	Это значение передается, если чтение-запись данных на устройстве не планируется, а требуется только изменить его конфигурацию, например временную отметку файла
GENERIC_READ	Разрешает доступ к устройству только для чтения
GENERIC_WRITE	Разрешает доступ к устройству только для записи. Данное значение используется для работы с принтером либо архивными носителями. Заметьте, что установка GENERIC_WRITE не предполагает автоматическую установку флага GENERIC_READ
GENERIC_READ | GENERIC_WRITE	Разрешает доступ к устройству для чтения и записи. Это значение используется чаще всего, поскольку оно обеспечивает свободный обмен данными

Параметр *dwShareMode* определяет совместным доступом к устройству. Он определяет, как устройство будет открываться другими вызовами *CreateFile*, выполненными, пока вы еще не закрыли это устройство (вызовом *CloseHandle*). Возможные значения параметра *dwShareMode* перечислены в табл. 10-5.

Табл. 10-5. Значения параметра dwShareMode

Значение	Описание
0	Запрашивается монопольный доступ к устройству. Если это устройство уже открыто, вызов *CreateFile* заканчивается неудачей. Если же открыть устройство удастся, неудачей закончатся все последующие вызовы *CreateFile*
FILE_SHARE_READ	Этот флаг запрещает модификацию данных на этом устройстве всем другим объектам ядра. Если устройство уже открыто для записи или для монопольного доступа, вызов *CreateFile* заканчивается неудачей. Если же открыть устройство удастся, неудачей закончатся все последующие вызовы *CreateFile* с флагом GENERIC_WRITE
FILE_SHARE_WRITE	Этот флаг запрещает чтение данных на этом устройстве всем другим объектам ядра. Если устройство уже открыто для чтения или для монопольного доступа, вызов *CreateFile* заканчивается неудачей. Если же открыть устройство удастся, неудачей закончатся все последующие вызовы *CreateFile* с флагом GENERIC_READ

Табл. 10-5. *(продолжение)*

Значение	Описание
FILE_SHARE_READ | FILE_SHARE_ WRITE	Всем объектам ядра разрешен доступ для чтения и записи к этому устройству. Если устройство уже открыто для чтения или для монопольного доступа, вызов *CreateFile* заканчивается неудачей. Если же удалить устройство удастся, неудачей закончатся все последующие вызовы *CreateFile* с запросом монопольного доступа для чтения, записи либо чтения и записи закончатся неудачей
FILE_SHARE_ DELETE	Этот флаг разрешает логическое удаление или перемещение файла, даже если он еще не закрыт. В действительности при этом Windows помечает данный файл для удаления, но удаляет его только когда все описатели этого файла будут закрыты

Примечание. Открывая файл, вы можете передать путь **к** нему, максимальная длина которого определяется значением MAX_PATH (не более 260 символов, согласно определению в файле WinDef.h). Однако, это ограничение можно обойти, вызывая *CreateFileW* (Unicode-версию функции *CreateFile***)** и предваряя путь строкой "*\\?\". При вызове *CreateFileW* префикс удаляется, и максимальная длина пути составляет около 32 ООО Unicode-символов. Однако помните, что с данным префиксом разрешено использовать только полные пути, относительные ссылки, включая «.» и «..», не поддерживаются. При этом длина отдельных компонентов пути по-прежнему ограничена значением MAX_PATH. Не стоит также удивляться, обнаружив в различных исходных кодах константу _MAX_PATH, значение которой в стандартных библиотеках C/C++ определено как 260 (см. stdlib.h).

Параметр *psa* ссылается на структуру SECURITY_ATTRIBUTES, которая позволяет задать атрибуты системы безопасности, а также указать, должен ли быть наследуемым описатель, возвращаемый функцией *CreateFile*. Дескриптор защиты, который содержится в этой структуре, используется только при создании файлов в защищенных файловых системах, таких как NTFS, и игнорируется в остальных случаях. Обычно в параметре *psa* передают просто NULL. При этом создается файл с параметрами защиты по умолчанию, а функция возвращает описатель, не являющийся наследуемым.

Параметр *dwCreationDisposition* особо важен при вызове *CreateFile* для открытия именно файлов, а не других устройств. Возможные значения этого параметра приводятся в табл. 10-6.

Табл. 10-6. Значения параметра *dwCreationDisposition* функции *CreateFile*

Значение	Описание
CREATE_NEW	Вызов *CreateFile* создает новый файл либо заканчивается неудачей, если файл с таким же именем уже существует
CREATE_ALWAYS	Вызов *CreateFile* создает новый файл независимо от того, существует ли уже файл с таким же именем, и перезаписывает существующий файл
OPEN_EXISTING	Вызов *CreateFile* открывает существующий файл либо заканчивается неудачей, если, файл или устройство с таким же именем не существует
OPEN_ALWAYS	Вызов *CreateFile* открывает файл, если он существует, либо создает новый файл в противном случае
TRUNCATE_EXISTING	Вызов *CreateFile* открывает существующий файл, усекая его до 0 байтов, либо заканчивается неудачей, если заданный файл не существует

> **Примечание.** Вызывая *CreateFile*, чтобы открыть любое устройство кроме файла, в параметре *dwCreationDisposition* следует передавать значение OPEN_EXISTING.

Параметр *dwFlagsAndAttributes* функции *CreateFile* используется для установки флагов, оптимизирующих взаимодействие с устройством, а также для установки файловых атрибутов (если устройство является файлом). Большинство флагов-параметров этой функции сообщают системе, как вы планируете обращаться к устройству. Это позволяет оптимизировать алгоритм кэширования и повысить эффективность работы приложения. Начнем с описания этих флагов, а затем перейдем к атрибутам.

Флаги функции *CreateFile*, управляющие кэшированием

В этом разделе описаны различные флаги *CreateFile*, управляющие кэшированием, при этом особое внимание уделяется объектам файловой системы. Подробную информацию о других объектах ядра, таких как почтовые ящики, см. в документации MSDN.

Флаг FILE_FLAG_NO_BUFFERING

Этот флаг запрещает использование буферизации при обращении к файлу. Для повышения быстродействия система кэширует данные при чтении и записи их на диск, и этот флаг обычно не указывают, чтобы позволить диспетчеру кэша хранить в памяти недавно затребованные данные файловой системы. Так что если, прочитав из файла несколько байтов, вы попробуете прочитать из этого файла еще несколько байтов, скорее всего окажется, что система уже загрузила в оперативную память эти данные. В итоге удастся обойтись вместо двух обращений к диску одним, что сильно повышает быс-

тродействие. Однако в действительности при этом в памяти оказывается две копии содержимого файла: в буфере диспетчера кэша и в вашем собственном буфере, куда эти данные копируются вызванной вами функцией (такой как *ReadFile*).

При буферизации данных диспетчер кэша может использовать упреждающее чтение, так что к тому моменту, когда программа обратится к следующей порции данных (при последовательном чтении файла), эти данные, скорее всего, будут уже в памяти. Скорость работы повышается и за счет чтения из файла большего количества данных, чем реально требуется программе. Но если программе не потребуются следующие порции данных из этого файла, часть памяти будет потрачена зря. (Подробнее об упреждающем чтении см. в описании флагов ниже FILE_FLAG_SEQUENTIAL_SCAN и FILE_FLAG_RANDOM_ACCESS.)

Установив флаг FILE_FLAG_NO_BUFFERING, вы запретите диспетчеру кэша буферизацию данных, но знайте, что при этом ответственность за кэширование данных ложится на вас! В зависимости от задачи, этот флаг может повышать быстродействие и эффективность использования памяти. Поскольку драйвер устройства файловой системы пишет содержимое файла прямо в предоставленные вами буферы, придерживайтесь следующих правил:

- в работе с файлом всегда следует использовать только смещения, кратные размеру сектора дискового тома, на котором находится файл (определить его можно с помощью функции *GetDiskFreeSpace*);
- читать и записывать данные всегда следует порциями, размер которых в байтах кратен размеру сектора дискового тома;
- адрес начала буфера, выделенного в адресном пространстве процесса, должен быть кратным размеру диска.

Флаги FILE_FLAG_SEQUENTIAL_SCAN и FILE_FLAG_RANDOM_ACCESS

Эти флаги полезны, только если вы разрешаете системе управлять кэшированием содержимого файла за вас. Если установлен флаг FILE_FLAG_NO_BUFFERING, оба этих флага игнорируются.

Установив флаг FILE_FLAG_SEQUENTIAL_SCAN, вы сообщите системе, что собираетесь читать файл последовательно. В результате система будет читать содержимое файла в память до того, как вы реально обратитесь к нему, что уменьшает число обращений к жесткому диску и повышает быстродействие вашего приложения. Если вам придется обращаться к нужным участкам файла напрямую, система потратит зря небольшую часть процессорного времени и памяти, кэшируя данные, которые вам так и не потребуются. Это вполне нормально, но если такая ситуация повторяется часто, лучше указать флаг FILE_FLAG_RANDOM_ACCESS, запрещающий упреждающее чтение.

Для управления файлом диспетчеру кэша требуется ряд внутренних структур данных, и чем больше файл, тем больше таких структуру нужно

диспетчеру кэша. При работе с чрезвычайно большими файлами диспетчеру кэша может не хватить памяти для необходимых структур, в результате ему не удастся открыть такой файл. Так что при работе с гигантскими файлами устанавливайте флаг FILE_FLAG_NO_BUFFERING.

Флаг FILE_FLAG_WR1TE_THR0UGH

Последний флаг, управляющий кэшем, отключает буферизацию данных, предназначенных для записи в файл, снижая тем самым риск потери данных. Если установлен этот флаг, все модифицированные данные система сразу же записывает в файл. Тем не менее, система поддерживает внутренний кэш содержимого файла, по возможности читая его из кэша, а не напрямую с диска. Когда этот флаг используют, чтобы открыть файл с сетевого диска, Windows-функции для записи в файлы не возвращают управление вызывающему потоку, пока данные не будут записаны в файл на сетевом диске. Вот, собственно, и все о флагах, управляющих кэшированием.

Другие флаги функции *CreateFile*

В этом разделе мы поговорим о других флагах *CreateFile*, не связанных с кэшированием.

Флаг FILE_FLAG_DELETE_ON_CLOSE

Этот флаг заставляет систему удалить файл после того, как будет закрыт последний его описатель. Чаще всего данный флаг используют с флагом-атрибутом FILE_ATTRIBUTE_TEMPORARY. Одновременное использование этих двух флагов позволяет в приложении создавать временные файлы, записывать и читать в них, а затем закрывать временные файлы. Как только временный файл будет закрыт, система автоматически удалит его — как удобно!

Флаг FILE_FLAG_BACKUP_SEMANTICS

Этот флаг используют в программах для архивации и восстановления данных. Прежде чем открыть или создать файл, система обычно проверяет наличие у процесса, пытающегося выполнить это действие, достаточного уровня привилегий. Особенностью программ архивации является их способность обходить проверку прав доступа в отношении некоторых файлов. Если установлен флаг FILE_FLAG_BACKUP_SEMANTICS, система проверяет наличие у маркера доступа вызывающего потока привилегии Backup/Restore File and Directories. Если она имеется; система разрешает открыть файл. Флаг FILE_FLAG_BACKUP_SEMANTICS также используют, чтобы открывать описатели каталогов.

Флаг FILE_FLAG_POSIX_SEMANTICS

В Windows регистр символов в именах файлов сохраняется, но не учитывается при поиске файлов по имени. Однако подсистема POSIX требует

учитывать регистр символов при поиске файлов по имени. Флаг FILE_PLAG_POSIX_SEMANTICS заставляет *CreateFile* учитывать регистр при поиске во время создания или открытия файла. Флаг FILE_FLAG_POSIX_SEMANTICS следует применять чрезвычайно осторожно, поскольку файлы, созданные с таким флагом, могут оказаться недоступными Windows-приложениям.

Флаг FILE_FLAG_OPEN_REPARSE_POINT

По этому, этому флагу больше подходит имя FILE_FLAG_IGNORE_REPARSE_POINT, поскольку он заставляет систему игнорировать атрибут повторного разбора, если таковой есть у файла. Такие атрибуты позволяют фильтрам файловой системы по-другому открывать, читать, записывать и закрывать файлы. Как правило это делается с определенной целью, поэтому использовать флаг FILE_FLAG_OPEN_REPARSE_POINT не рекомендуется.

Флаг FILE_FLAG_OPEN_NO_RECALL

Этот флаг запрещает системе восстанавливать содержимое файла с архивного носителя (такого как картридж с магнитной пленкой) на подключенное к системе хранилище (например, на жесткий диск). Файлы, к которым долгое время никто не обращается, система может перенести на архивные носители, чтобы освободить место на жестком диске. При этом с жесткого диска удаляется только содержимое файла, а сам файл остается на диске. Когда кто-то пытается открыть этот файл, система автоматически восстанавливает его содержимое **с** архивного носителя. Флаг FILE_FLAG_OPEN_NO_RECALL запрещает системе автоматически выполнять такие операции.

FILE_FLAG_OVERLAPPED

Установив этот флаг, вы сообщаете системе, что хотите работать с устройством асинхронно. По умолчанию устройства открываются для синхронного ввода-вывода (без флага FILE_FLAG_OVERLAPPED). Большинство разработчиков привыкло именно к синхронному вводу-выводу, когда поток приостанавливается до завершения чтения данных из файла. После чтения необходимых данных поток вновь получает управления и его исполнение продолжается.

Поскольку операции ввода-вывода на устройства являются довольно медленным по сравнению с другими операциями, стоит подумать об использовании асинхронного ввода-вывода на отдельных устройствах. Вот как он работает: вы вызываете функцию, запрашивающую у системы чтение или запись данных, но вместо того, чтобы ждать завершения этой операции, вызванная функция немедленно возвращает управление, а операционная система сама завершает операцию ввода-вывода, используя собственные потоки. Закончив запрошенную операцию, система уведомляет об этом ваше приложение. Асинхронный ввод-вывод — ключ к созданию производительных, масштабируемых и надежных приложений, быстро реагирующих на

действия пользователя. Windows поддерживает несколько методов асинхронного ввода-вывода, о которых будет рассказано в этой главе.

Флаги файловых атрибутов

А теперь настало время вернуться к флагам атрибутов, которые передаются через параметр *dwFlagsAndAttributes* функции *CreateFile* (см. табл. 10-7). Эти флаги игнорируются во всех случаях кроме одного, когда вы создаете новый файл и передаете в параметре *hFileTemplate* NULL-значение. Многие из перечисленных атрибутов должны быть вам уже знакомы.

Табл. 10-7. Флаги файловых атрибутов (значения параметра *dwFlagsAndAttributes*)

Флаг	Описание
FILE_ATTRIBUTE_ ARCHIVE	Файл является архивным. Приложения помечают этим флагом файлы, подлежащие удалению. Для новых файлов функция CreateFile устанавливает этот флаг автоматически
FILE_ATTRIBUTE_ ENCRYPTED	Файл зашифрован
FILE_ATTRIBUTE_ HIDDEN	Файл является скрытым и не отображается при просмотре каталога обычными средствами
FILE_ATTRIBUTE_ NORMAL	Другие атрибуты не установлены. Имеет смысл, только если является единственным атрибутом файла
FILE_ATTRIBUTE_ NOT_CONTENT_ INDEXED	Файл обрабатывается службой индексирования
FILE_ATTRIBUTE_ OFFLINE	Файл существует, но его содержимое перемещено на архивный носитель. Этот флаг применяется в иерархических системах хранения данных
FILE_ATTRIBUTE_ READONLY	Файл доступен только для чтения, приложениям разрешено его чтение, но запрещены запись и удаление этого файла
FILE_ATTRIBUTE_ SYSTEM	Файл является часть операционной системы либо используется только ей
FILE_ATTRIBUTE_ TEMPORARY	Временный файл. Операционная система пытается хранить его содержимое в оперативной памяти, чтобы свести к минимуму время доступа

Атрибут FILE_ATTRIBUTE_TEMPORARY служит для создания временных файлов. Система стремится хранить файл с таким атрибутом в оперативной памяти, а не на диске, что существенно ускоряет доступ к нему. Если вы запишете в этот файл много данных, и система не сможет хранить его в RAM, ей придется сбросить часть содержимого этого файла на жесткий диск. Комбинируя флаги FILE_ATTRIBUTE_TEMPORARY и FILE_ FLAG_DELETE_ON_CLOSE (о нем см. выше) можно повысить быстродействие системы. Как правило, после закрытия файла система сбрасывает

его кэшированное содержимое на диск. Если же атрибуты файла говорят системе, что данный файл следует удалить сразу после закрытия, ей не приходится сбрасывать на диск его содержимое.

В дополнение к вышеописанным флагам система поддерживает ряд флагов, управляющих защитой и работой именованных каналов. Подробнее о них см. в описании функции CreateFile в документации Platform SDK.

Последний параметр *CreateFile*, *hFileTemplate,* содержит описатель открытого файла либо NULL. Если в *hFileTemplate* находится описатель файла, *CreateFile* игнорирует любые атрибуты, заданные флагами в *dwFlagsAndAttributes,* используя вместо них атрибуты файла, заданного параметром *hFileTemplate*. Чтобы все это работало, заданный параметром *hFileTemplate* файл должен быть открыт с флагом GENERIC_READ. Если функция *CreateFile* открывает существующий файл (а не создает новый), параметр *hFileTemplate* игнорируется.

Успешно создав или открыв файл либо устройство, функция *CreateFile* возвращает описатель файла или устройства. Если же вызов *CreateFile* заканчивается неудачей, возвращается INVALID_HANDLE_VALUE.

> **Примечание.** Большинство Windows-функций, возвращающих описатели; при неудачном вызове возвращают NULL. В отличие от них, *CreateFile* возвращает в этом случае значение INVALID_HANDLE_VALUE (определенное как -1). Мне часто приходилось видеть такой ошибочный код:
>
> ```
> HANDLE hFile = CreateFile(…);
> if (hFile == NULL) {
>    // программа никогда не окажется здесь
> } else {
>    // этот блок будет исполнен, даже если файл не создан
> }
> ```
>
> Вот как надо правильно обращаться с значением, возвращаемым функцией *CreateFile*:
>
> ```
> HANDLE hFile = CreateFile(…);
> if (hFile == INVALID_HANDLE_VALUE) {
>    // файл не создан
> } else {
>    // файл успешно создан
> }
> ```

Работа с файлами

Поскольку работать с файлами приходится очень часто, я решил подробнее остановиться на проблемах, характерных для файлов, которые в контексте этой главы тоже считаются устройствами. Ниже я расскажу, как устанавливают указатель в файле и изменяют размер файла.

Прежде всего, вам следует знать, что Windows поддерживает работу с чрезвычайно большими файлами. Разработчики в Майкрософт использовали для представления размера файлов 64-разрядные значения вместо 32-разрядных. Это означает, что теоретический максимальный размер файла в Windows составляет 16 экзабайт.

Обработка 64-разрядных значений в 32-разрядной операционной системе несколько затруднительна, поскольку многие Windows-функции требуют передачи 64-разрядных значений как пары 32-разрядных. Однако вы убедитесь, что это не трудно, как кажется. Кроме того, вам редко придется сталкиваться с обработкой файлов, размер которых превышает 4 Гб. Это означает, что 32 старших бита 64-разрядных значений чаще всего будут нулевыми.

Определение размера файла

При работе с файлами довольно часто приходится определять их размеры. Проще всего сделать это вызовом *GetFileSizeEr*.

```
BOOL GetFileSizeEx(
    HANDLE         hFile,
    PLARGE_INTEGER pliFileSize);
```

Первый параметр *hFile* — описатель открытого файла, а *pliFileSize* — адрес объединенной структуры LARGEJNTEGER. Эта структура позволяет ссылаться на 64-разрядное значение с знаком как на пару 32-разрядных либо одно 64-разрядное, что весьма удобно при работе с размерами файлов и смещениями. Вот как выглядит эта объединенная структура:

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;                  // Младшее 82-разрядное значение без знака
        LONG HighPart;                  // Старшее 32-разрядное значение со знаком
    };
    L0NGL0NG QuadPart;                  // Полное 64-разрядное значение со знаком
} LARGE_INTEGER, *PLARGE_INTEGER;
```

В дополнение к LARGE_INTEGER поддерживается структура ULARGE_INTEGER, представляющая 64-разрядное значение без знака:

```
typedef union _ULARGE_INTEGER {
    struct {
        DWORD LowPart;      // Младшее 32-разрядное значение без знака
        DWORD HighPart;     // Старшее 32-разрядное значение без знака
    };
    ULONGLONG QuadPart;     // Полное 64-разрядное значение без знака
} ULARGE_INTEGER, *PULARGE_INTEGER;
```

Есть еще одна весьма полезная функция для определения размера файла — *GetCompressedFileSize*:

```
DWORD GetCompressedFileSize(
    PCTSTR pszFileName,
    PDWORD pdwFileSizeHigh);
```

Эта функция возвращает физический размер файла, а *GetFileSizeEx* — логический размер. Возьмем для примера 100-Кб файл, сжатый до размера 85 Кб. Вызов *GetFileSizeEx* вернет логический размер файла (100 Кб), тогда как *GetCompressedFileSize* вернет реальное число байтов, занятое на диске этим файлом (85 Кб).

В отличие от ***GetFileSizeEx,*** первый параметр функции *GetCompressedFileSize* — это имя файла в виде строки, а не описателя. *GetCompressedFileSize* возвращает 64-разрядное значение, представляющее размер файла, но делает это необычным способом. Младшие 32 бита этого числа передаются в возвращаемом значении функции, а старшие 32 бита записываются в DWORD-значение, на которое ссылается параметр *pdwFileSizeHigh*. Удобнее использовать структуру ULARGEJNTEGER следующим образом:

```
ULARGE_INTEGER ulFileSize;
ulFileSize.LowPart = GetCompressedFileSize(TEXT("SomeFile.dat"),
    &ulFileSize.HighPart);

// Теперь 64-разрядное значение размера файла содержится в ulFileSize.LowPart
```

Установка указателя в файле

Вызов *CreateFile* заставляет систему создать объект ядра «файл», управляющий работой с файлом. Внутри этого объекта ядра содержится указатель, определяющий 64-разрядное смещение внутри файла, по которому будет выполнена следующая синхронная операция чтения или записи. Исходно этот указатель установлен на 0, следовательно, если вызвать *ReadFile* сразу после *CreateFile*, файл будет прочитан с начала (т.е. с нулевой позиции). Если при этом было прочитано 10 байтов, система перемещает на 10 байтов указатель, связанный с описателем файла, поэтому при следующем вызове *ReadFile* чтение файла начнется уже с 11-го байта. Рассмотрим следующий пример кода, считывающего из файла в буфер первые 10 байтов, а затем еще 10 байтов.

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile = CreateFile(TEXT("MyFile.dat"), …);   // Указатель установлен на 0
ReadFlle(hFile, pb, 10, &dwNumBytes, NULL);         // Чтение байтов  0 - 9
ReadFile(hFile, pb, 10, idwNumBytes, NULL);         // Чтение байтов 10 - 19
```

У каждого объекта ядра имеется свой собственный указатель, поэтому, открыв файл два раза подряд, вы получите немного неожиданные результаты:

```
BYTE pb[10];
```

```
DWORD dwNumBytes;
HANDLE hFile1 = CreateFile(TEXT( "MyFile.dat"), …);          // Указатель
                                                             // установлен на 0
HANDLE hFile2 = CreateFile(TEXT("MyFile.dat"), ...);         // Указатель
                                                             // установлен на 0
ReadFile(hFile1, pb, 10, MwNumBytes, NULL);                  // Чтение байтов 0-9
ReadFile(hFile2, pb, 10, &dwNumBytes, NULL);                 // Чтение байтов 0-9
```

В этом примере два объекта ядра управляют одним и тем же файлом. Поскольку у каждого объекта ядра «файл» есть свой указатель, манипулирование файлом с помощью одного из объектов никак не влияет на указатель в другом объекте. В итоге первые 10 байтов будут прочитаны дважды.

Приведу еще один пример, который поможет вам разобраться в этом:

```
BYTE pb[10];
DWORD dwNumBytes;
HANDLE hFile1 = CreateFile(TEXT("MyFile.dat"), …);           // Указатель
                                                             // установлен на 0
HANDLE hFile2;
DuplicateHandle(
    GetCurrentProcess(), hFile1,
    GetCurrentProcess(), &hFile2,
    0, FALSE, 0UPLICATE_SANE_ACCESS);
ReadFile(hFile1, pb, 10, &dwNumBytes, NULL);                 // Чтение байтов 0-9
ReadFile(hFile2, pb, 10, &dwNumBytes, NULL);                 // Чтение байтов 10-19
```

В этом примере один объект ядра «файл» связан с двумя описателями одного и того же файла. В результате положение указателя в файле будет обновляться независимо от того, какой из объектов ядра используется для манипулирования файлом.

Для случайного доступа к файлу требуется возможность модификации указателя, связанного с объектом ядра «файл». Делается это вызовом функции *SetFilePointerEx*:

```
B00L SetFilePointerEx(
    HANDLE         hFile,
    LARGE_INTEGER  liDistanceToMove,
    PLARGE_INTEGER pliNewFilePointer,
    DWORD          dwMoveMethod);
```

Параметр *hFile* определяет объект ядра «файл», указатель которого требуется изменить. Параметр *liDistanceToMove* сообщает системе, на сколько байт следует переместить указатель. Значение этого параметра прибавляется к текущему значению указателя, поэтому отрицательные значения *liDistanceToMove* перемещают указатель в обратном направлении (к началу файла). Последний параметр, *dwMoveMethod*, указывает, как функция *SetFilePointerEx* должна интерпретировать параметр *liDistanceToMove*. Возможные значения этого параметра перечислены в табл. 10-8.

Табл. 10-8. Значения параметра *dwMoveMethod* функции *SetFilePointerEx*

Значение	Описание
FILE_BEGIN	Указатель объекта «файл» перемещается в положение, заданное параметром *HDistanceTbMove*, который интерпретируется как 64-разрядное значение без знака
FILE_CURRENT	Значение параметра *HDistanceTbMove* прибавляется к текущему значению указателя объекта «файл». Учтите, что отрицательные значения интерпретируются как смещения к началу файла, что позволяет вести поиск в обоих направлениях
FILE_END	Новая позиция указателя вычисляется как сумма логического размера файла и значения пара метра *HDistanceToMove*, которое интерпретируется как 64-разрядное значение со знаком. Это позволяет вести поиск в файле в обоих направлениях

Обновив указатель в объекте «файл», функция *SetFilePointerEx* возвращает новое значение указателя в параметре *pliNewFilePointer*, содержащем ссылку на LARGEINTEGER. Можно отказаться от получения нового значение указателя, передайте NULL в параметре *pliNewFilePointer*.

Обратите внимание не несколько моментов, касающихся *SetFilePointerEx*:

- допускается устанавливать указатель дальше конца файла. При этом размер файла не увеличивается, если только вы не запишете в файл данные или не вызовете функцию *SetEndOfFile*;
- при использовании функции *SetFilePointerEx* с файлом, открытым с флагом FILE_FLAG_NO_BUFFERING, устанавливать указатель можно только на границах, выровненных по размеру сектору (как это делается, я покажу на примере программы *FileCopy* ниже в этой главе);
- Windows не поддерживает функцию *GetFilePointerEx* для определения положения указателя. Но можно получить желаемое значение, переместив указатель на 0 байтов вызовом *SetFilePointerEx*, как показано ниже:

```
LARGE_IHTEGER liCurrentPosition = { 0 };
SetFilePointerEx(hFile, liCurrentPosition, &liCurrentPosition, FILE_CURRENT);
```

Установка конца файла

Обычно система автоматически устанавливает конец файла при его закрытии. Однако в некоторых случаях требуется принудительно сделать файл больше или меньше. Для этого вызывают следующую функцию:

```
BOOL SetEndOfFile(HANDLE hFile);
```

Функция *SetEndOfFile* уменьшает или увеличивает файл до размера, заданного текущим положением указателя в файле. Так, если вы хотите принудительно назначить файлу размер 1024 байта, вызовите *SetEndOfFile* следующим образом:

```
HANDLE hFile = CreateFile(…);
LARGE_INTEGER liDistanceToMove;
liDistanceToMove.QuadPart = 1024;
SetFilePointerEx(hFile, liDistanceToMove, NULL, FILE_BEGIN);
SetEndOfFile(hFile);
CloseHandle(hFile);
```

Если проверить свойства такого файла с помощью Windows Explorer, его размер окажется равным в точности 1024 байтам.

Синхронный ввод-вывод на устройствах

В этом разделе рассказывается о Windows-функциях для синхронного ввода-вывода на устройствах. Помните, что к устройствам относятся файлы, почтовые ящики, каналы, сокеты и пр. Независимо от типа устройства для синхронного ввода-вывода используются одни и те же функции.

Несомненно, самыми простыми и востребованными функциям для чтения-записи на устройствах являются *ReadFile* и *WriteFile*:

```
BOOL ReadFile(
    HANDLE       hFile,
    PVOID        pvBuffer,
    DWORD        nNumBytesToRead,
    PDWORD       pdwNumBytes,
    OVERLAPPED*  pOverlapped);

BOOL WriteFile(
    HANDLE       hFile,
    CONST VOID   *pvBuffer,
    DWORD        nNumBytesToWrite,
    PDWORD       pdwNumBytes,
    OVERLAPPED*  pOverlapped);
```

Параметр *hFile* задает описатель нужного устройства. Для открытого устройства нельзя задавать флаг FILE_FLAG_OVERLAPPED, иначе система подумает, что вы хотите работать с устройством асинхронно. Параметр *pvBuffer* ссылается на буфер для хранения прочитанных данных либо данных, подлежащих записи. Параметры *nNumBytesToRead* и *nNumBytesToWrite* сообщают функциям *ReadFile* и *WriteFile,* сколько байтов следует прочитать с устройства либо записать на него, соответственно.

Параметр *pdwNumBytes указывает адрес* DWORD-переменной, в которую функция записывает число байтов, успешно полученных с устройства либо переданных на него. Последний параметр, *pOverlapped,* в случае синхронного ввода-вывода устанавливают в NULL. Подробнее о нем — в разделе, посвященном асинхронному вводу-выводу.

При успешном завершении вызовы *ReadFile* и *WriteFile* возвращают TRUE. Кстати, *ReadFile* можно вызывать только для устройств, открытых с

флагом GENERIC_READ. Аналогично, *WriteFile* вызывают только для **уст**ройств, открытых с флагом GENERIC_WRITE.

Сброс данных на устройство

Вспомните, что функция *CreateFile* принимает целый ряд флагов, которые определяют, как система каптирует содержимое файлов. Другие устройства, такие как последовательные порты, почтовые ящики и канала, также способны кэшировать файлы. Чтобы заставить систему записать кэшированные данные на устройство, можно воспользоваться функцией *FlushFileBuffers:*

```
BOOL FlushFileBuffers(HANDLE hFile);
```

Эта функция принуждает систему сбросить все содержимое буферов на устройство, заданное параметром *hFile.* Для этого устройство должно быть открыто с флагом GENERIC_WRITE. При успешном завершении вызова функция возвращает TRUE.

Отмена синхронного ввода-вывода

Функции для синхронного ввода-вывода просты в использовании, но до завершения их вызова поток прекращает все остальные операции. Типичным примером может служить функция *CreateFile*. Когда пользователь вводит информацию с помощью мыши или клавиатуры происходит вставка оконных сообщений в очередь потока — создателя окна, в которое выполняется ввод. Если поток оказался заблокированным в ожидании завершения вызова *CreateFile*, обработка оконных сообщений останавливается и все окна, созданные этим потоком, «застывают». Блокирование потоков в ожидании завершения синхронного ввода-вывода — наиболее распространенная причина «зависаний» приложений!

Майкрософт сделала в Windows Vista ряд заметных нововведений, призванных решить эту проблему. Так, при зависании консольного приложения из-за синхронного ввода-вывода пользователь теперь может, нажав Ctrl+C, вернуть себе контроль над консолью и продолжить работу с ней, не «убивая» процесс консоли. Кроме того, диалоги сохранения и открытия файлов содержат кнопку Cancel, которой можно отменить чрезмерно затянувшуюся операцию (например, обращение к файлу, расположенному на сетевом сервере).

Чтобы приложение реагировало на действие пользователя без задержек, следует по максимуму использовать асинхронный ввод-вывод. Кроме того, этот подход позволяет обойтись в приложении минимумом потоков, что экономит ресурсы (такие как объекты ядра «поток» и «стек»). Для асинхронных операций также проще реализовать возможность отмены. Например, Internet Explorer позволяет отменять веб-запросы (щелчком красной кнопки с белым крестом либо нажатием клавиши Esc), если их исполнение слишком затянулось и пользователь больше не хочет ждать.

К сожалению, некоторые API-функции, такие как *CreateFile*, не поддерживают асинхронный вызов методов. Некоторые из этих методов могут завершаться по тайм-ауту (если их исполнение заняло слишком много времени, например, обращение к сетевому серверу), но лучше бы иметь функцию, принудительно завершающую ожидание или просто отменяющую синхронный ввод-вывод. В Windows Vista следующая функция позволяет отменить незаконченные операции синхронного ввода-вывода для заданного потока:

```
BOOL CancelSynchronouslo(HANDLE hThread);
```

Параметр *hThread* — это описатель потока, приостановленного в ожидании завершения синхронного запроса ввода-вывода. Данный описатель должен разрешать завершение (THREAD_TERMINATE) потока. В противном случае вызов *CancelSynchronousIo* заканчивается неудачей, а GetLastError возвращает ERROR_ACCESS_DENIED. Описатели созданных вами потоков разрешают полный доступ к ним (THREAD_ALL_ACCESS), в том числе для завершения (THREAD_TERMINATE). Если же вы используете поток из пула либо код для отмены вызывается по таймеру, как правило, приходится вызывать *OpenThread*, чтобы получить описатель потока с соответствующим идентификатором; при этом не забудьте передать THREAD_TERMINATE как первый параметр.

Если заданный поток приостановлен в ожидании завершения синхронной операции ввода-вывода, вызов *CancelSynchronousIo* пробудит этот поток, а операция, которую он ожидал, завершится неудачей. При этом вызов *GetLastError* вернет ERROR_OPERATION_ABORTED, а *CancelSynchronousIo* вернет TRUE вызвавшему его потоку.

Заметьте, что вызывающий *CancelSynchronousIo* поток не «знает», что именно делает поток, заблокированный на синхронной операции. Возможно, он просто был вытеснен, так и не успев обратиться к устройству. Этот поток также может ожидать ответа устройства либо устройство ответило, и поток уже выходит их синхронной операции. Если функция *CancelSynchronousIo* вызвана на потоке, который не ждал отклика устройства, вызов *CancelSynchronousIo* возвращает FALSE, а *GetLastError* — ERROR_NOT_FOUND.

По этой причине имеет смысл использовать дополнительные средства синхронизации потоков (см. главы 8 и 9), чтобы отменять только синхронные операции. Однако на практике это обычно не требуется, поскольку инициатором отмены обычно является пользователь, обнаруживший, что приложение «зависло». Кроме того, пользователь может попытаться еще раз отменить операцию, если ему покажется, что первая попытка не удалась. Кстати, Windows автоматически вызывает *CancelSynchronousIo*, чтобы вернуть пользователю контроль над консолью либо диалогом сохранения или открытия файла.

Внимание! Возможность отмены запросов ввода-вывода зависит от реализации соответствующего уровня в драйвере. Драйвер может и не поддерживать отмену. В этом случае *CancelSynchronousIo* все равно вернет TRUE, поскольку функция обнаружила запрос, который необходимо пометить для отмены. Собственно, отмена находится в компетенции драйвера. Примером обновленного драйвера, поддерживающего отмену синхронных операций в Windows Vista, может быть сетевой редиректор.

Асинхронный ввод-вывод на устройствах: основы

Ввод-вывод на устройствах — самая медленная и наименее предсказуемая категория операций, выполняемых компьютером. Процессор выполняет арифметические действия и даже закрашивает экран намного быстрее, чем чтение-запись данных в файлы, в том числе через сеть. Однако применение асинхронного ввода-вывода позволяет оптимизировать использование ресурсов и создавать более эффективные приложения.

Рассмотрим поток, генерирующий асинхронный запрос ввода-вывода. Этот запрос передается драйверу устройства, который в итоге берет на себя ответственность за исполнение» собственно, ввода-вывода. Пока драйвер ждет отклика устройства, поток приложения не будет приостановлен в ожидании завершения запроса ввода-вывода, а продолжит исполнение других полезных операций.

В какой-то момент драйвер устройства завершит обработку запросов ввода-вывода и ему придется уведомить приложение об успешной отправке или приеме данных либо об ошибке. Подробнее об этом рассказывается далее, а сейчас разберем постановку запросов ввода-вывода в очередь. Очереди запросов ввода-вывода — важнейший механизм для проектирования высокопроизводительных масштабируемых приложений, о котором и пойдет речь в остальных разделах этой главы.

Для асинхронного доступа к устройству последнее сначала необходимо открыть вызовом *CreateFile* с передачей флага FILE_FLAG_OVERLAPPED в параметре dwFlagsAndAttributes. Этот флаг уведомляет систему о том, что вы планируете работать с устройством асинхронно.

Для постановки запросов ввода-вывода в очередь драйвера устройства применяют уже известные вам функции *ReadFile* и *WriteFile* (см. раздел о синхронном вводе-выводе выше). Для удобства я снова приведу прототипы этих функций:

```
BOOL ReadFile(
   HANDLE       hFile,
   PVOID        pvBuffer,
   DWORD        nNumBytesToRead,
```

```
    PDWORD       pdwNumBytes,
    OVERLAPPED*  pOverlapped);

BOOL WriteFile(
    HANDLE       hFile,
    CONST VOID   *pvBuffer,
    DWORD        nNumBytesToWrite,
    PDWORD       pdwNumBytes,
    OVERLAPPED* pOverlapped);
```

При вызове любая из этих функций проверяет, не открыто ли заданное параметром *hFile* устройство с флагом FILE_FLAG_OVERLAPPED. Если это так, функция выполняет запрошенную операцию ввода-вывода асинхронно. Кстати, при вызове этих функций для асинхронного ввода-вывода в параметре *pdwNumBytes* можно передать NULL (обычно так и делается), что и понятно: этот вызов должен вернуть управление до завершения запроса ввода-вывода, следовательно, считать число переданных байтов в этот момент бессмысленно.

Структура OVERLAPPED

При выполнении асинхронного ввода-вывода на устройствах необходимо передавать адрес инициализированной структуры OVERLAPPED через параметр *pOverlapped*. В этом контексте название структуры (overlapped — по-английски «перекрывающийся») означает, что исполнение запросов ввода-вывода перекрывается по времени с исполнением потоком других операций. Вот как выглядит структура OVERLAPPED:

```
typedef struct _OVERLAPPED {
    DWORD  Internal;         // [вывод] код ошибки
    DWORD  InternalHigh;     // [вывод] число переданных байтов
    DWORD  Offset;           // [ввод]  смещение в файле, младшие 32 бита
    DWORD  OffsetHigh;       // [ввод]  смещение в файле, старшие 32 бита
    HANDLE hEvent;           // [ввод]  описатель события или данных
} OVERLAPPED, *LPOVERLAPPED;
```

Эта структура включает пять элементов. Три из них — *Offset, OffsetHigh* и *hEvent* — должны быть инициализированы до вызова *ReadFile или WriteFile.* Остальные два, Internal и *InternalHigh,* устанавливаются драйвером устройства и могут быть прочитаны по завершении операции ввода-вывода. Рассмотрим их поподробнее.

- *Offset* и *OffsetHigh*

  При обращении к файлу в эти элементы записывают 64-разрядное смещение, определяющее позицию в файле, с которой начнется операция ввода-вывода. Вспомните, что с каждым объектом ядра «файл» связан указатель. При синхронном запросе ввода этот указатель определяет, с какого места начнется обращение к файлу. По завершении операции система ав-

томатически обновляет указатель, чтобы начать следующую операцию с того места, на котором закончилась предыдущая. При асинхронных операциях ввода-вывода система игнорирует этот указатель. Представьте, что будет, если ваш код два раза асинхронно вызовет *ReadFile* (на одном и том же объекте ядра). В этом случае система не узнает, откуда следует начать чтение при втором вызове *ReadFile*. Ясно только, что читать область файла, уже прочитанную при первом вызове *ReadFile*, не нужно. Поэтому во избежание путаницы при повторных асинхронных вызовах на одном и том же объекте с каждым асинхронным запросом ввода-вывода необходимо передавать в составе структуры OVERLAPPED смещение в файле. Заметьте, что элементы *Offset* и *OffsetHigh* не игнорируются, даже если устройство не является файлом. Эти элементы необходимо установить в 0, в противном случае запрос ввода-вывода закончится неудачей, а *GetLastError* вернет ERROR_INVALID_PARAMETER.

- *hEvent*
  Этот элемент используется одним в одном из методов получения уведомлений о завершении ввода-вывода. В методе, основанном на вводе-выводе с оповещениями, это элемент можно использовать для собственных целей. Я знаю многих разработчиков, хранящих в *hEvent* адреса различных C++-объектов (подробнее об этом элементе см. далее в этой главе.)
- *Internal*
  Это поле хранит код ошибки, сгенерированный при обработке запроса ввода-вывода. В случае асинхронного запроса драйвер устанавливает для *Internal* значение STATUS_PENDING, свидетельствующее об отсутствии ошибок (поскольку ввод-вывод еще не начался). Проверить, завершена ли асинхронная операция ввода-вывода, можно при помощи макроса *HasOverlappedloCompleted*, определенного в WmBase.h. Если операция еще не закончилась, макрос вернет FALSE, а в противном случае — TRUE. Вот определение этого макроса:

```
#define HasOverlappedloCompleted(pOverlapped) \
   ((pOverlapped)->Internal != STATUS_PENDING)
```

- *InternalHigh*
  По завершении асинхронного запроса ввода-вывода в это поле записывается число переданных байтов.
  Изначально при разработке структуры OVERLAPPED Майкрософт решила не документировать поля *Internal* и *InternalHigh* (отсюда их названия). Со временем стало ясно, что хранящаяся в этих поля информация может оказаться полезной разработчикам, в результате поля были задокументированы. Несмотря на это, полям оставили прежние имена, поскольку они часто используются в коде операционной систем, а его в Майкрософт изменять не хотели.

Примечание. По завершении асинхронного запроса ввода-вывода вы получаете адрес структуры OVERLAPPED, использованной при генерации запроса. Сведения о контексте, переданные в OVERLAPPED, часто оказываются полезными. Например, в этой структуре можно сохранить описатель устройства, которому адресован запрос ввода-вывода. У этой структуры нет ни специального поля для хранения описателя устройства, ни других полей для хранения потенциально полезных сведений о контексте, но это проблему довольного легко решить.

Я часто создаю на основе структуры OVERLAPPED производные C++-классы, способные хранить любую необходимую мне информацию. Получив в своих приложениях адрес структуры OVERLAPPED, я просто привожу его к указателю на экземпляр моего C++-класса. Так я получаю доступ ко всем элементам OVERLAPPED и любой контекстной информации, необходимой моему приложению. Этот метод демонстрируется на примере программы *FileCopy*, показанной в конце этой главы (см. код C++-класса *FileCopy*).

Асинхронный ввод-вывод на устройствах: «подводные камни»

Используя асинхронный ввод-вывод, следует быть в курсе пары сложностей. Во-первых, драйверы устройств не обязаны обрабатывать очередь запросов ввода-вывода по принципу «первый вошел, первый вышел» (first-in first-out, FIFO). Так, в случае потока, исполняющего показанный ниже код, драйвер устройства вполне способен сначала записать данные файла, а потом прочитать этот файл:

```
OVERLAPPED o1 = { 0 };
OVERLAPPED o2 = { 0 };
BYTE bBuffer[100];
ReadFile (hFile, bBuffer, 100, NULL, &o1);
WriteFile(hFile, bBuffer, 100, NULL, &o2);
```

Как правило, драйверы устройств исполняют запросы в порядке, оптимальном с точки зрения производительности. Например, стремясь уменьшить число перемещений головки жесткого диска и времени доступа, драйвер файловой системы может выбрать из очереди запросы ввода-вывода, обращающихся к физически близким областям диска.

Вторая сложность связана с проверкой наличия ошибок. Большинство Windows-функций сообщают об ошибке, возвращая FALSE, а об успехе — возвращая значение, отличное от нуля. Однако функции *ReadFile* и *WriteFile* в этом плане отличаются. Чем — разберемся на примере.

При попытке поставить в очередь асинхронный запрос ввода-вывода драйвер устройства может исполнить этот запрос синхронно. Так бывает, например, если при чтении файла система обнаружила, что запрошенные данные уже находятся в кэше. В этом случае запрос ввода-вывода не попа-

дает в очередь драйвера: система просто копирует нужные данные из кэша в ваш буфер — и все, запрос обработаю. Некоторые операции, такие как сжатие NTFS, увеличение размеров и дописывание данных к файлам драйверы всегда выполняют синхронно (подробнее см. по ссылке http://support.microsoft.com/default.aspx?scid=kb%3Ben-us%3B156932).

Функции *ReadFile* и *WriteFile* возвращают отличное от нуля значение, если запрос ввода-вывода был выполнен Синхронно. Если же он выполнен синхронно, либо при вызове *ReadFile* или *WriteFile* возникла ошибка, возвращается FALSE. Получив FALSE, вы должны вызвать *GetLastError*, чтобы выяснить, что именно случилось. Если *GetLastError* возвращает ERROR_IO_PENDING, запрос ввода вывода успешно поставлен в очередь и будет выполнен позже.

Если же *GetLastError* вернет значение, отличное от ERROR_IO_PENDING, поставить запрос ввода-вывода в очередь драйвера не удалось. Ниже приводятся коды ошибок при постановке запросов ввода-вывода в очередь драйвера устройства, которые *GetLastError* возвращает чаще всего:

- ERROR_INVALID_USER_BUFFER или ERROR_NOT_ENOUGH_MEMORY
  Каждый драйвер устройства поддерживает список незавершенных запросов ввода-вывода, хранящийся в невыгружаемом пуле и включающий фиксированное число элементов. Если этот список заполнен, добавить в него новые запросы невозможно, в результате *ReadFile* и *WriteFile* возвращают FALSE, а *GetLastError* — один из показанных выше кодов ошибок.
- ERROR_NOT_ENOUGH_QUOTA
  Некоторые устройства требуют блокировать в ОЗУ страницы памяти, составляющие выделенный вами буфер, во избежание выкачивания их на диск в то время, пока запрос стоит в очереди. Типичный пример — файловый ввод-вывод при использовании флага FILE_FLAG_NO_BUFFERING. Однако система ограничивает число страниц, которое может заблокировать отдельный процесс. Если функциям *ReadFile* и *WriteFile* не удается заблокировать нужное число страниц, эти функции возвращают FALSE, а *GetLastError* — ERROR_NOT_ENOUGH_QUOTA. Увеличить квоту процесса можно, вызвав *SetProcessWorkingSetSize*.

Как обрабатывать такие ошибки? В сущности, они возникают из-за превышения лимита необработанных запросов ввода-вывода, поэтому следует дождаться завершения некоторых запросов и повторно вызвать *ReadFile* или *WriteFile*.

И третье, что нужно знать об асинхронном вводе-выводе: буфер с данными и структуру OVERLAPPED нельзя ни перемещать, ни уничтожать до завершения соответствующего запроса ввода-вывода. Дело в том, что при постановке запроса в очередь драйвера устройства последний получает адреса буфера и структуры OVERLAPPED, именно адреса, а не соответствующие блоки памяти. Почему это так, вполне понятно: копирование блоков памяти отнимает массу ресурсов и процессорного времени.

Подготовившись к обработке очередного запроса, драйвер передает данные, которые находятся по адресу, заданному параметром *pvBuffer*, а затем обращается к полям, содержащим смещение в файле, и другим элементам структуры OVERLAPPED, адрес которой задан параметром *pOverlapped*. В частности, драйвер записывает в поле *Internal* код ошибки ввода-вывода, а в поле *InternalHigh* — число переданных байтов.

> **Примечание.** Во избежание повреждения памяти ни в коем случае не перемещайте и не разрушайте эти буферы до завершения запроса ввода-вывода. Кроме того, для каждого запроса ввода-вывода вы должны создать и инициализировать отдельную структуру OVERLAPPED.

Предыдущее замечание чрезвычайно важно, поскольку именно по этой причине разработчики чаще всего допускают ошибки, реализуя асинхронный ввод-вывод в приложениях. Вот пример того, как не нужно писать код:

```
VOID ReadData(HANDLE hFile) {
    OVERLAPPED o = { 0 };
    BYTE b[100];
    ReadFile(hFile, b. 100, NULL, &o);
}
```

Этот код выглядит довольно безобидно, и вызов *ReadFile* написан вроде бы безупречно. Проблема лишь в том, что эта функция возвращает управление после постановки в очередь асинхронного запроса ввода-вывода. При этом буфер и структура OVERLAPPED удаляются из стека потока, а драйвер устройства не «в курсе», что вызов *ReadData* завершен и по-прежнему использует переданные ему адреса двух блоков памяти в стеке потока. Когда запрос ввода-вывода завершится, драйвер запишет результаты в стек потока. В результате будут затерты все данные, оказавшиеся к моменту завершения запроса в переданных драйверу областях памяти. Диагностировать такие ошибки особенно сложно, поскольку модификация памяти происходит асинхронно. Иногда драйвер ввод-вывод синхронно, и тогда ошибка не проявляется. В других случаях ввод-вывод может завершиться сразу после вызова *ReadData*, а может через час и более — кто знает, что окажется в стеке к этому моменту?

Отмена запросов ввода-вывода, ожидающих в очереди

Иногда требуется отменить поставленный в очередь запрос ввода-вывода до того, как он будет обработан драйвером устройства. *Windows позволяет* сделать это несколькими способами:

- функция *CancelIo* позволяет отменить все запросы ввода-вывода, сгенерированные вызывающим потоком для заданного описателя (если только этот описатель не связан с портом завершения ввода-вывода):

```
BOOL Cancello(HANDLE hFile);
```

- закрыв описатель устройства, можно отменить все адресованные ему запросы ввода-вывода, независимо от сгенерировавшего их потока;
- при завершении потока система автоматически отменяет все сгенерированные им запросы ввода-вывода за исключением запросов на описатели, связанные с портами завершения ввода-вывода;
- отменить отдельный запрос ввода-вывода можно вызовом функции *CancelIoEx*:

```
BOOL CancelIoEx(HANDLE hFile, LPOVERLAPPED pOverlapped);
```

CancelIoEx позволяет отменять незавершенные запросы ввода-вывода, сгенерированные потоком, отличным от вызывающего потока. Эта функция помечает как отмененные все запросы, заданные значениями параметров *hFile* и *pOverlapped*. Поскольку каждому из незавершенных запросов ввода-вывода соответствует уникальная структура OVERLAPPED, вызов *CancelloEx* отменяет один-единственный запрос. Если же параметр pOverlapped содержит NULL, *CancelloEx* отменяет все незавершенные запросы ввода-вывода, адресованные устройству с описателем, заданным параметром *hFile*.

> **Примечание.** Отмененные запросы ввода-вывода завершаются с кодом ошибки ERROR_OPERATION_ABORTED.

Уведомление о завершении ввода-вывода

Итак, вы уже знаете, как поставить в очередь асинхронный запрос ввода-вывода, а теперь поговорим о том, как драйвер устройства уведомляет приложения о том, что он завершил обработку запросов ввода-вывода.

Windows поддерживает четыре способа уведомления о завершении ввода-вывода (см. табл. 10-9), ниже мы разберем каждый из них подробно. Эти методы осуждаются в порядке от простейшего в реализации (освобождение объекта ядра «устройство») до самого сложного (порты завершения ввода-вывода).

Табл. 10-9. Методы уведомления о завершении ввода-вывода

Метод	Описание
Освобождение объекта ядра	Неудобен в ситуациях, когда одному устройству адресовано «устройство» сразу несколько запросов ввода-вывода. Позволяет потоку обработать результаты запроса, сгенерированного другим потоком
Освобождение объекта ядра	Поддерживает ситуации, когда одному устройству «событие» адресовано сразу несколько запросов ввода-вывода. Позволяет потоку обработать результаты запроса, сгенерированного другим потоком

Табл. 10-9. (окончание)

Метод	Описание
Ввод-вывод с оповещением	Поддерживает ситуации, когда одному устройству адресовано сразу несколько запросов ввода-вывода. Результаты запроса ввода-вывода должен обработать поток, который сгенерировал этот запрос
Порты завершения ввода-вывода	Поддерживает ситуации, когда одному устройству адресовано сразу несколько запросов ввода-вывода. Позволяет потокам обрабатывать результаты запросов, сгенерированных другими потоками. Обеспечивает наибольшую масштабируемость и гибкость

Как сказано выше, порты завершения ввода-вывода — наилучший метод получения уведомлений о завершении ввода-вывода. Однако рекомендую вам изучить все четыре метода, чтобы понять, зачем Майкрософт добавила порты завершения ввода-вывода и этот механизм позволяет решать проблемы, характернее для менее совершенных методов.

Освобождение объекта ядра «устройство»

Сгенерировав асинхронный запрос ввода-вывода, поток продолжает работать, исполняя различные полезные операции. Однако в итоге поток должен синхронизироваться с исполнением запрошенной им операции ввода-вывода. Иными словами, рано или поздно настанет момент, когда поток не сможет продолжить работу, не получив в полном объеме с устройства необходимые ему данные.

В Windows объекты ядра «устройство» используются для синхронизации потоков, поэтому такой объект может быть свободен либо занят. Функции *ReadFile* и *WriteFile* переводят объект ядра устройство в состояние «занят» непосредственно перед постановкой в очередь запроса ввода-вывода. Обработав запрос ввода-вывода, драйвер устройства переводит этот объект ядра в состояние «свободен».

Поток может узнать, завершен ли запрос ввода-вывода, вызвав функцию *WaitForSingleObject* либо *WaitForMultipleObjects.* Вот простой пример:

```
HANDLE hFile = CreateFile(…, FILE_FLAG_OVERLAPPED, …);
BYTE bBuffer[100];
OVERLAPPED o = { 0 };
o.Offset = 345;

BOOL bReadDone = ReadFile(hFile, bBuffer, 100, NULL, &o);
DWORD dwError = GetLastError();

if ((bReadDone && (dwError == ERR0R_I0_PENDING)) {
    // Ввод-вывод выполняется синхронно, ждем его завершения
```

```
    WaitForSingleObject(hFile, INFINITE);
    bReadDone = TRUE;
}

if (bReadDone) {
    // o.Internal содержит код ошибки ввода-вывода
    // o.InternalHigh содержит число переданных байтов
    // bBuffer содержит прочитанные данные
} else {
    // An error occurred; see dwError
}
```

Этот код генерирует асинхронный запрос ввода-вывода, после чего ожидает его завершения, что абсолютно противоречит логике асинхронного ввода-вывода! Ясно, что вы никогда не напишете такой код для реального приложения, я же использую его как иллюстрацию важных моментов, о которых говорится ниже.

- Устройство должно быть открыто для асинхронного ввода-вывода с флагом FILE_FLAG_OVERLAPPED.
- Элементы *Offset*, *OffsetHigfr* и *hEvent* структуры OVERLAPPED должны быть инициализированы. В этом примере я установил их в 0 за исключением *Offset*, которому присвоено значение 345, чтобы *ReadFile* начала чтение файла с 346-го байта,
- Значение, возвращаемое функцией *ReadFile*, записывается в переменную *bReadDone*. Проверив ее, можно узнать, выполнен ли запрос ввода-вывода синхронно.
- Если запрос ввода-вывода не был выполнен синхронно, я пытаюсь выяснить, не возникла ли ошибка, либо запрос был выполнен асинхронно. Это делается путем сравнения результата *GetLastError* с ERROR_IO_PENDING.
- Чтобы поток дождался результатов исполнения запроса, я вызываю *WaitForSingleObject* и передаю ей описатель объекта ядра «устройство». Как сказано в главе 9, вызов этой функции приостанавливает поток до освобождения объекта ядра. Драйвер устройства освобождает этот объект ядра после завершения ввода-вывода. Когда *VMtForSingleObject* возвращает управление, ввод-вывод уже завершен, и я устанавливаю *bReadDone* в TRUE.
- Когда данные будут прочитаны, можно проверить данные в *bBuffer*, код ошибки и число переданных байтов, соответственно, в полях *Internal* и *InternalHigh* структуры OVERLAPPED.
- Если возникла ошибка, ее код и дополнительные сведения можно получить, проверив *dwError*.

Освобождение объекта ядра «событие»

Этот метод получения уведомлений о завершении ввода-вывода очень прост и напоминает предыдущий. Однако на практике он не очень удобен, так как плохо работает, когда запросов ввода-вывода много. Предположим, что ваша программа пытается выполнить сразу несколько асинхронных операций (например, записать и прочитать 10 байт) над одним и тем же файлом. Вот пример такой программы:

```
HANDLE hFile = CreateFile(…, FILE_FLAG_0VERLAPPED, …);

BYTE bReadBuffer[10];
OVERLAPPED oRead = { 0 };
oRead.Offset = 0;
ReadFile(hFile, bReadBuffer, 10, NULL, &oRead);

BYTE bWriteBuffer[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
OVERLAPPED oWrite = { 0 };
oWrite.Offset =10;
WriteFile(hFile, bWriteBuffer, _countof(bWriteBuffer), NULL, &oWrite);
…
WaitForSingle0bject(hFile, INFINITE);

// Неизвестно, какая операция завершилась: чтение, запись или обе операции?
```

Синхронизация потоков путем ожидания устройства невозможно, поскольку объект «устройство» освободиться по завершении любой из операций. Если вызвать функцию *WaitForSingleObject* и передать ей описатель устройства, сложно будет сказать, что стало причиной возврата управления этой функции: завершение чтения, записи или обеих операций. Ясно, что должен быть более эффективный способ координации одновременных запросов ввода-вывода, исполняемых асинхронно, и он, к счастью, существует.

Последний элемент структуры OVERLAPPED, *hEvent*, идентифицирует объект ядра «событие». Вы должны создать этот объект вызовом *CreateEvent*. По завершении асинхронного запроса ввода-вывода драйвер устройства проверяет, не установлен ли элемент *hEvent* структуры OVERLAPPED в NULL и, если нет, освобождает содержащийся в нем объект-событие, вызывая *SetEvent*. Кроме того, драйвер, как обычно, освобождает объект «устройство». Если проверка завершения операций на устройстве выполняется с помощью событий, следует ожидать освобождения не объекта «устройство», а соответствующего объекта «событие».

> **Примечание.** Можно немного повысить производительность, приказав Windows не освобождать объект «файл» сразу после завершения операции. Для это вызовите функцию *SetFileCompletionNotificationModes* следующим образом:

```
BOOL SetFileCompletionNotificationMode8(HANDLE hFile, UCHAR uFlags);
```

Параметр *hFile* содержит описатель файла, параметр *uFlags* определяет, что должна сделать Windows после завершения операции ввода-вывода. Если передать в этом параметре флаг FILE_SKIP_SET_EVENT_ON_HANDLE, Windows не освободит объект «файл» по завершении операции с файлом. К сожалению, флаг FILE_SKIP_SET_EVENT_ON_HANDLE назван очень неудачно, ему куда больше подошло бы имя вроде FILE_SKIP_SIGNAL.

Чтобы одновременно выполнить несколько асинхронных запросов ввода-вывода на устройстве, следует создать отдельный объект «событие» для каждого запрос, инициализировать элемент hEvent структуры OVERLAPPED у каждого запроса и вызвать *ReadFile* или *WriteFile*. Достигнув момента, когда продолжение работы невозможно без результатов запроса ввода-вывода, просто вызовите функцию *WaitForMultipleObjects*, передав ей описатели событий, связанные со структурами OVERLAPPED запросов, исполнения которых необходимо дождаться. Такой алгоритм позволяет просто и надежно выполнять одновременные асинхронные запросы ввода-вывода с использованием одного и того же объекта «устройство». Следующий пример иллюстрирует этот подход:

```
HANDLE hFile = CreateFile(…, FILE_FLAG_OVERLAPPED, …);

BYTE bReadBuffer[10];
OVERLAPPED oRead = { 0 };
oRead.Offset = 0;
oRead.hEvent = CreateEvent(…);
ReadFile(hFile, bReadBuffer, 10, NULL, &oRead);

BYTE bWriteBuffer[10] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
OVERLAPPED oWrite = { 0 };
oWrite.Offset = 10;
oWrite.hEvent = CreateEvent(…);
WriteFile(hFile, bWriteBuffer, .qountof(bWriteBuffer), NULL, AoWrite);
…

HANDLE h[2];
h[0] = oRead.hEvent;
h[1] = oWrite.hEvent;
DWORD dw = WaitForMultiple0bjects(2, h, FALSE, INFINITE);
switch (dw - WAIT_0BJECT_0) {
    case 0:    // чтение завершено
       break;

    case 1:    // запись завершена
       break;
}
```

В реальных приложениях такой код не используется, но зато он хорошо иллюстрирует мою мысль. Как правило, в реальных приложениях ожидание завершения запроса ввода-вывода реализовано в виде цикла. После завершения запроса поток выполняет нужное действие, ставит в очередь следующий асинхронный запрос ввода-вывода и снова крутится в цикле, пока не будет обработан следующий запрос.

GetOverlappedResult

Как я уже говорил, изначально Майкрософт не собиралась документировать поля *Internal* и *InternalHigh* структуры OVERLAPPED. Следовательно, нужно было как-то иначе сообщить разработчику, сколько байтов было передано в ходе операции ввода-вывода и передать ему код ошибки. Для этого Майкрософт предусмотрела функцию *GetOverlappedResult*:

```
BOOL GetOverlappedResult(
    HANDLE      hFile,
    OVERLAPPED* pOverlapped,
    PDWORD      pdwNumBytes,
    BOOL        bWalt);
```

Теперь поля *Internal* и *InternalHigh* задокументированы, поэтому проку от *GetOverlappedResult* немного. Однако во время знакомства с асинхронным вводом-выводом я решил выяснить внутреннее устройство этой функции, чтобы глубже изучить концепции, лежащие в основе этого механизма. Вот как выглядит внутренняя реализация функции *GetOverlappedResult*.

```
BOOL GetOverlappedResult(
    HANDLE hFile,
    OVERLAPPED* po,
    PDWORD pdwNumBytes,
    BOOL bWait) {

    if (po->Internal == STATUS_PENDING) {
        DWORD dwWaitRet = WAIT_TIME0UT;
        if (bWait) {
            // Ожидаем завершения ввода-вывода
                dwWaitRet = WaitForSingleObject(
                (po->hEvent != NULL) ? po->hEvent : hFile, INFINITE);
        }

        if (dwWaitRet == WAIT_TIMEOUT) {
            // Ввод-вывод не завершен и его ожидание не планируется
            SetLastError(ERR0R_I0_INC0MPLETE);
            return(FALSE);
        }
```

```
        if (dwWaitRet != WAIT_OBJECT_0) {
            // ошибка при вызове WaitForSingleObject
            return(FALSE);
        }
    }
    // Ввод-вывод завершен, возвращаем число переданных байтов
    pdwNumBytes = po->InternalHigh;

    if (SUCCEEDED(po->Internal)) {
        return(TRUE);    // Ввод-вывод выполнен без ошибок
    }

    // Устанавливаем последнюю ошибку согласно ошибке ввода-вывода
    SetLastError(po->Internal);
    return(FALSE);
}
```

Ввод-вывод с оповещением

Третий метод получения уведомлений о завершении ввода-вывода называется *ввод-вывод с оповещением* (alertable I/O). Майкрософт продвигала его как самый лучший механизм для создания высокопроизводительных масштабируемых приложений. Однако, начав использовать ввод-вывод с оповещением, разработчики вскоре поняли, что возможности этого механизма не оправдывают их ожиданий.

Мне пришлось довольно много применять ввод-вывод с оповещением, поэтому скажу вам сразу: этот механизм ужасен и лучше всячески избегать его. Однако для обеспечения его работоспособности Майкрософт добавила в свои операционные системы чрезвычайно полезную инфраструктуру с ценными возможностями. Так что, читая этот раздел, не старайтесь досконально разобраться в аспектах, связанных с вводом-выводом, а обращайте больше внимания на вспомогательную инфраструктуру.

Вместе с потоком система всегда создает очередь, называемая *очередью вызовов асинхронных процедур* (asynchronous procedure call, APC) или APC-очередью потока. При генерации запроса ввода-вывода можно приказать драйверу устройства добавить элемент в APC-очередь вызывающего потока. Чтобы уведомления о завершении ввода-вывода помещались в APC-очередь потока, следует вызвать функцию *ReadFileEx* или *WriteFileEx*:

```
BOOL ReadFileEx(
    HANDLE       hFile,
    PVOID        pvBuffer,
    DWORD        nNumBytesToRead,
    OVERLAPPED*  pOverlapped,
    LPOVERLAPPED_C0MPLETI0N_R0UTINE pfnCompletionRoutine);
```

```
BOOL WriteFileEx(
    HANDLE       hFile,
    CONST VOID   *pvBuffer,
    DWORD        nNumBytesToWrite,
    OVERLAPPED*  pOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE pfnCompletionRoutine);
```

Подобно *ReadFile* и *WriteFile*, функции *ReadFileEx* и *WriteFileEx* отправляют запросы ввода-вывода драйверам устройств и немедленно возвращают управление. Функции *ReadFileEx* и *WriteFileEx* по набору параметров не отличаются от *ReadFile* с *WriteFile* за исключением двух моментов. Во-первых, **Ex*-функции не принимают указатель на DWORD-значение, в которое записывается число переданных байтов, получить эту информацию можно лишь с помощью функции обратного вызова. Во-вторых, **Ex*-функциям необходимо передавать адрес функции обратного вызова, называемой также *процедурой завершения* (completion routine). Прототип этой функции должен иметь следующий вид:

```
VOID WINAPI CompletlonRoutine(
    DWORD        dwError,
    DWORD        dwNumBytes,
    OVERLAPPED* po);
```

Генерируя асинхронный запрос ввода-вывода, функции *ReadFileEx* или *WriteFileEx* передают адрес функции обратного вызова драйверу устройства. Завершив обработку этого запроса, драйвер добавляет в APC-очередь вызывающего потока новый элемент. Этот элемент содержит адреса функции обратного вызова (процедуры завершения) и структуры OVERLAPPED, с использованием которой был инициирован запрос ввода-вывода.

Примечание Между прочим, при завершении ввода-вывода с оповещением, драйвер не пытается освободить объект-событие. В действительности устройство даже не ссылается на элемент *hEvent* структуры OVERLAPPED. Поэтому при желании можете использовать элемент *hEvent* для собственных нужд.

Когда поток находится в *тревожном состоянии* (alertable state), о котором будет рассказано чуть ниже, система анализирует его APC-очередь и для каждого ее элемента вызывает соответствующую процедуру завершения, передавая ей код ошибки ввода-вывода, число переданных байтов и адрес структуры OVERLAPPED. Заметьте, что код ошибки и число переданных байтов также хранятся в полях *Internal* и *InternalHigh* структуры OVERLAPPED (как сказано выше, причина этого — в первоначальном нежелании Майкрософт документировать эти поля).

Чуть позже мы вернемся к процедурам завершения, а пока рассмотрим, как система обрабатывает асинхронные запросы ввода-вывода. Следующий код ставит в очередь три асинхронных операции:

```
hFile = CreateFile(…, FILE_FLAG_OVERLAPPED, …);

ReadFileEx(hFile, …);                          // первый вызов ReadFileEx
WriteFileEx(hFile, …);                         // первый вызов WriteFileEx
ReadFileEx(hFile, …);                          // второй вызов ReadFileEx

SomeFunc();
```

Исполнение вызова *SomeFunc* занимает некоторое время, поскольку при этом система должна выполнить три операции. Пока поток исполняет функцию *SomeFunc*, драйвер устройства добавляет в его APC-очередь элементы, соответствующие завершенным операциям ввода-вывода. Вот как может выглядеть APC-очередь:

```
first WriteFileEx completed
second ReadFileEx completed
first ReadFileEx completed
```

Система автоматически поддерживает APC-очередь. Из этого примера также видно, что система может исполнять поставленные в очередь запросы в любом порядке: запросы, добавленные первыми, могут быть исполнены в последнюю очередь, и наоборот. Каждый элемент APC-очереди потока содержит адрес и параметры функции обратного вызова.

Выполненные запросы ввода-вывода просто добавляются в APC-очередь потока, то есть процедура завершения вызывается не сразу, поскольку поток может быть занят исполнение какой-нибудь важной операции, прервать которую невозможно. Для обработки элементов APC-очереди поток должен перевести себя в тревожное состояние. Это всего лишь означает, что исполнение потока достигло точки, в которой оно может быть прервано. Windows поддерживает шесть функций, способных перевести поток в тревожное состояние:

```
DWORD SleepEx(
    DWORD  dwMilliseconds,
    BOOL   bAlertable);

DWORD WaitForSingleObjectEx(
    HANDLE hObject,
    DWORD  dwMilliseconds,
    BOOL   bAlertable);

DWORD WaitForMultipleObjectsEx(
    DWORD  cObjects,
    CONST HANDLE* phObjects,
    BOOL   bWaitAll,
    DWORD  dwMilliseconds,
    BOOL   bAlertable);
```

```
BOOL SignalObjectAndWait(
    HANDLE hObjectToSignal,
    HANDLE hObjectToWaitOn,
    DWORD  dwMilliseconds,
    BOOL   bAlertable);

BOOL GetQueuedCompletionStatusEx(
    HANDLE hCompPort,
    LPOVERLAPPED_ENTRY pCompPo rtEnt ries,
    ULONG  ulCount,
    PULONG pulNumEntriesRemoved,
    DWORD  dwMilliseconds,
    BOOL   bAlertable);

DWORD MsgWaitForMultipleObjectsEx(
    DWORD  nCount,
    CONST HANDLE* pHandles,
    DWORD  dwMilliseconds,
    DWORD  dwWakeMask,
    DWORD  dwFlags);
```

Последним аргументом первых пяти функций является булево значение, которое указывает, должен ли поток перейти в тревожное состояние. Функции *MsgWaitForMultipleObjectsEx* следует передавать флаг MWMO_ALERTABLE, чтобы перевести поток в тревожное состояние. Знакомые с функциями *Sleep*, *WaitForSingleObject* и *WaitForMultipleObjects* не удивятся, узнав, что функции без суффикса *Ex* вызывают расширенные версии этих функций (с суффиксом *Ex* в именах), всегда передавая FALSE в параметре *bAlertable.*

После вызова одной из вышеперечисленных функций и перехода потока в тревожное состояние система прежде всего проверяет APC-очередь этого потока. Если в ней есть хотя бы один элемент, поток не засыпает. Вместо этого система извлекает элемент из APC-очереди и поток вызывает процедуру завершения, передавая ей код ошибки завершенной операции ввода-вывода, число переданных байтов и адрес структуры OVERLAPPED. Кода эта процедура возвращает управление, система ищет следующий элемент в APC-очереди. Если в очереди еще есть элементы, они обрабатываются, в противном случае функция, вызов которой перевел поток в тревожное состояние, возвращает управление. Учтите, что если APC-очередь потока окажется непустой в момент вызова функций, которые переводят поток в тревожное состояние, поток так никогда и не «заснет»!

Эти функции приостановят исполнение потока, только если в момент их вызова APC-очередь этого потока окажется пустой. Спящий (приостановленный) поток пробуждается при освобождении рбъекта (или объектов) ядра, которых он ждал, либо при добавлении элемента в его APC-очередь.

Поскольку при этом поток находится в тревожном состоянии, при появлении в APC-очереди новых элементов система тут же пробуждает поток и обрабатывает их, вызывая соответствующие процедуры завершения. Эти функции немедленно возвращают управление, так что поток не засыпает в ожидании освобождения необходимого ему объекта ядра.

Результат вызова вышеописанных функций можно узнать по значению, которое они возвращают. Если такая функция (либо *GetLastError*) вернула WAIT_IO_COMPLETION, знайте, что поток занят обработкой своей APC-очереди. Если причина завершения другая, поток пробудился, потому что истек его период ожидания, освободился нужный этому потоку объект ядра, либо другой поток отказался от мьютекса.

Плюсы и минусы ввода-вывода с оповещением

Итак, мы разобрались, как работает ввод-вывод с оповещением, а теперь я расскажу, почему я так не люблю этот механизм.

- Функции обратного вызова

  Для работы ввода-вывода с оповещением необходимо создавать функции обратного вызова, что сильно затрудняет написание кода. Как правило, контекстной информации, предоставляемой этими функциями, слишком мало для эффективной диагностики, поэтом в итоге все равно приходится хранить много сведений в глобальных переменных. К счастью, эти глобальные переменные не приходится синхронизировать, поскольку функцию для перехода в тревожное состояние и функцию обратного вызова исполняет один и тот же поток. Поскольку один поток не может исполнить сразу две функции, эти переменные вне опасности.

- Сложности с масштабируемостью

  Самая большая проблема ввода-вывода с оповещением состоит в следующем: уведомление о завершении запроса ввода-вывода должен обработать тот же самый поток, который инициировал этот запрос. Если поток генерирует несколько запросов, тот же самый поток должен обработать все уведомления о завершении этих запросов, даже если все остальные потоки в это время бездействуют. Поскольку балансировка нагрузки в этом случае не поддерживается, использующие этот механизм приложения плохо масштабируются.

Обе проблемы весьма серьезны, поэтому я настоятельно рекомендую избегать применения ввода-вывода с оповещением на устройствах. Думаю, вы уже догадываетесь, что механизм, о котором пойдет речь в следующем разделе, — порты завершения ввода-вывода — решает обе эти проблемы. Но я обещал рассказать о достоинствах инфраструктуры для ввода-вывода с оповещением, поэтому сначала я выполню свое обещание.

Windows поддерживает функцию, которая позволяет вручную добавить элемент в APC-очередь потока:

```
DWORD QueueUserAPC(
    PAPCFUNC  pfnAPC,
    HANDLE    hThread,
    ULONG_PTR dwData);
```

Первый ее параметр — указатель на APC-функцию с прототипом следующего вида:

```
VOID WINAPI APCFunc(ULONG_PTR dwParam);
```

Второй параметр — описатель потока, в очередь которого нужно добавить элемент. Это может быть любой из потоков в системе. Если *hThread* идентифицирует поток, принадлежащий другому процессу, в *pfnAPC* дожжен быть адрес функции в адресном пространстве соответствующего процесса. Последний параметр функции *QueueUserAPC*, *dwData*, — значение, которое передается функции обратного вызова.

Хотя прототип *QueueUserAPC* объявлен так, что эта функция должна возвращать DWORD, на самом деле она возвращает значение типа BOOL, свидетельствующее об успешном либо неудачном завершении вызова. G помощью функции *QueueUserAPC* позволяет наладить чрезвычайно эффективное взаимодействие между потоками, даже принадлежащими разным процессам. К сожалению, так допускается передавать одно-единственное значение.

QueueUserAPC также позволяет принудительно вывести поток из ожидания. Предположим, ваш поток вызвал *WaitForSingleObject* и заснул в ожидании освобождения объекта ядра. В это время пользователь отдал команду на завершение приложения. Вы знаете, что поток должен корректно уничтожить себя, но как заставить пробудить спящий поток, чтобы тот совершил «самоубийство»? Ответ — с помощью функции *QueueUserAPC*.

Следующий код демонстрирует, как принудительно пробудить поток, чтобы тот корректно завершил свою работу. Главная функция потока порождает новый поток и передает ему описатель какого-нибудь объекта ядра. Пока вторичный поток работает, работает и первичный поток. Далее вторичный поток (исполняющий функцию *ThreadFunc*) вызывает функцию *WaitForSingleObjectEx*, которая приостанавливает поток и переводит его в тревожное состояние. А теперь предположим, что пользователь приказывает первичного потоку завершить приложение. Естественно, первичный поток может завершиться и тогда система просто «убьет» весь процесс. Однако это не очень «чистое» решение, и зачастую требуется прервать отдельную операцию, сохранив процесс.

Итак, первичный поток вызывает функцию *QueueUserAPC*, добавляющую элемент в APC-очередь потока. Поскольку вторичный поток находится в тревожном состоянии, он просыпается и опустошает свою APC-очередь вызовом функции *APCFunc*. Эта функция не делает абсолютно ничего и просто возвращает управление. Поскольку APC-очередь теперь пуста, вызов *WaitForSingleObjectEx* возвращает управление потоку с кодом WAIT_IO_

COMPLETION. Функция *ThreadFunc* ждет именно этого значения, поскольку оно говорит о том, что поток был разбужен для корректного завершения.

```
// это функция обратного вызова APC, которая ничего не делает
VOID WINAPI APCFunc(ULONG_PTR dwParam) {
    // здесь нет никаких действий
}

UINT WINAPI ThreadFunc(PVOID pvParam) {
    HANDLE hEvent = (HANDLE) pvParam;    // описатель передается потоку

    // тревожное ожидание, из которого поток нужно вывести для корректного заверше-
       ния
    DWORD dw = WaitForSingleObjectEx(hEvent, INFINITE, TRUE);
    if (dw == WAIT_OBJECT_0) {
        // объект освободился
    }
    if (dw == WAIT_I0_C0MPLETI0N) {
        // вызов QueueUserAPC принудительно вывел поток из ожидания
        return(0);                          // поток завершается корректно
    }
    …
    return(0);
}

void main() {
    HANDLE hEvent = CreateEvent(…);
    HANDLE hThread = (HANDLE) _beginthreadex(NULL, 0,
        ThreadFunc, (PVOID) hEvent, 0, NULL);
    …

    // корректное принудительное завершение вторичного потока
    QueueUserAPC(APCFunc, hThread, NULL);
    WaitForSingleObject(hThread, INFINITE);
    CloseHandle(hTh read);
    CloseHandle(hEvent);
}
```

Некоторые могут подумать, что проблему можно решить, заменив *WaitForSingleObjeciEx* на *WaitForMultipleObjects* и создав объект ядра «событие», освободив которое, можно заставить вторичный поток корректно завершиться. В моем простом примере это сработает, но если вторичный поток вызовет *WaitForMultipleObjects* так, чтобы ждать освобождения всех объектов, использованием *QueueUserAPC* будет единственным решением для принудительного вывода потока из ожидания.

Порты завершения ввода-вывода

Windows создавалась как защищенная и надежная операционная система, в которой смогут работать приложения, обслуживающие тысячи пользователей. Традиционно разработчикам были доступны две модели построения приложений-служб:

- Модель на основе последовательной обработки
  Используется единственный поток, ожидающий поступления запросов от клиентов (обычно по сети). Получив запрос, поток пробуждается и обрабатывает его.
- Модель на основе параллельной обработки
  В этой модели используется несколько потоков. Один из потоков ожидает запросы от клиентов. Получив запрос, он порождает новый поток, обрабатывающий принятый запрос. В это время первый поток крутится в цикле в ожидании следующих клиентских запросов. Закончив обработку запроса, второй поток завершается.

Проблема с моделью, основанной на последовательной обработке, состоит в том, что она плохо справляется с обработкой множества одновременных запросов. Клиентские запросы, поступающие одновременно, могут быть обработаны только по очереди, друг за другом. Службы, спроектированные с использованием этой модели, неспособны использовать преимущества многопроцессорных компьютеров. Ясно, что модель, основанная на последовательной обработке, годится только для простейших серверных приложений, которым приходится обрабатывать мало клиентских запросов, которые не создают большой нагрузки. Типичным примером сервера, обрабатывающего запросы последовательно, может служить сервер Ping.

Ограничения последовательной модели обусловили чрезвычайно высокую популярность модели, основанной на параллельной обработке. Эта модель предусматривает создание отдельного потока для обработки каждого из клиентских запросов. Ее преимуществом является очень низкая загруженность потока, принимающего входящие запросы: большую часть времени он просто спит. При поступлении клиентского запроса этот поток пробуждается, порождает новый поток для обработки запроса и возвращается к ожиданию следующих запросов. Это означает, что клиентские запросы обрабатываются независимо друг от друга. Кроме того, поскольку для каждого запроса создается отдельный поток, такие серверные приложения прекрасно масштабируются и эффективно работают на многопроцессорных машинах. Таким образом, установка дополнительных процессоров позволяет повысить производительность серверных приложений, построенных с использованием параллельной модели.

Для Windows были разработаны службы, использовавшие параллельную обработку. Однако их производительность оказалась меньше желаемой. В частности, разработчики Windows заметили, что при одновременной обработке множества запросов в системе появляется множество потоков.

Поскольку все эти потоки являются готовыми к выполнению (т.е. они не приостановлены и не ожидают какого-либо события), стало ясно, что ядро Windows тратит слишком много времени на переключение контекста, поэтому потокам достается слишком мало процессорного времени для выполнения полезной работы. Чтобы сделать среду Windows более приспособленной для серверных приложений, Майкрософт должна была решить эту проблему. В результате появился объект ядра «порт завершения ввода-вывода».

Создание портов завершения ввода-вывода

Принцип портов завершения ввода-вывода заключается в ограничении числа одновременно исполняемых потоков. Так, невозможно создать 500 готовых к выполнению потоков для обработки 500 запросов. В этой связи возникает вопрос: а каково оптимальное число одновременно работающих потоков? Если вдуматься, вряд ли есть смысл иметь больше одного готового к выполнению потока в расчете на один процессор. Ведь как только число потоков становится больше числа процессоров, системе приходится тратить драгоценное процессорное время на переключение контекста потоков — в этом состоит один из недостатков модели, основанной на параллельной обработке.

К недостаткам этой модели относится и необходимость создания нового потока для каждого клиентского запроса. Создание потока обходится намного дешевле, чем целого процесса с его виртуальным адресным пространством, но оно, все же, далеко не бесплатно. Можно повысить производительность службы, если во время ее инициализации создать пул потоков, который будет доступен, пока служба работает. Порты завершения ввода-вывода предназначены для использования вместе с пулами потоков.

Порт завершения ввода-вывода является, наверное, самым сложным объектом ядра. Чтобы создать его, следует вызвать функцию *CreateIoCompletionPort*.

```
HANDLE CreateIoCompletionPort(
    HANDLE      hFile,
    HANDLE      hExistingCompletionPort,
    UL0NG_PTR CompletionKey,
    DWORD       dwNumbe rOfConcu r rentTh reads);
```

Эта функция выполняет две операции: создает порт завершения ввода-вывода и связывает с ним устройство. По-моему, эта функция слишком сложна и лучше бы Майкрософт разбила ее на две функции. Когда я работают с портами завершения ввода-вывода, я разделяю эти операции путем создания пары крошечных функций, абстрагирующих вызовы *CreateIoCompletionPort*. Первая функция, *CreateNewCompletionPort,* выглядит так:

```
HANDLE CreateNewCompletionPort(DWORD dwNumberOfConcurrentThreads) {
```

```
    return(CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0,
        dwNumberOfConcurrentThreads));
}
```

Она принимает единственный аргумент, *dwNumberOfCmmnentThreads*, и вызывает Windows-функцию *CreateIoCompletionPort* с параметрами, из которых первые три «зашиты» в коде, а четвертый является значением *dwNumberOfConcurrentThreads*. Как видите, первые три параметра *CreateIoCompletionPort* используются только для связывания устройства с портом завершения ввода-вывода (об этом — чуть ниже). Чтобы просто создать порт завершения ввода-вывода, я передаю функции *CreateIoCompletionPort* в первых трёх параметрах значения INVALID_HANDLE_VALUE, NULL и 0, соответственно.

Параметр *dwNumberOfConcurrentThreads* определяет максимальное число планируемых потоков для данного порта завершения ввода-вывода. Если установить этот параметр в 0, то по умолчанию число готовых к выполнению потоков для этого порта будет равно числу процессоров, установленных в системе. Как правило, это то, что надо, чтобы свести к минимуму лишние переключения контекста. Имеет смысл увеличить это значение, если обработка клиентских запросов требует интенсивных вычислений и редко приводит к блокированию потоков, но лучше этого не делать. Попробуйте поэкспериментировать с параметром *dwNumberOfConcurrentThreads,* чтобы узнать, как его значение влияет на производительность вашего приложения на имеющемся оборудовании.

Вероятно, вы заметили, что *CreateIoCompletionPort* — одна из немногих Windows-функций для создания объектов ядра, но не принимающих адрес структуры SECURITY_ATTRIBUTES. Дело в том, что порты ввода-вывода предназначены для использования в пределах одного процесса, почему — станет ясно, когда я объясню, как их используют.

Связывание устройства с портом завершения ввода-вывода

При создании порта завершения ввода-вывода ядро в действительности создает целых пять различных структур данных, как показано на рис. 10-1; продолжая чтение раздела, держите этот рисунок перед глазами.

Первая структура данных представляет собой список устройств, связанных с данным портом. Для связывания устройства с портом ввода-вывода необходимо вызвать функцию *CreateIoCompletionPort.* Для этого я тоже написал собственную функцию:

```
BOOL AssociateDeviceWithCompletionPort(
    HANDLE hCompletionPort, HANDLE hDevice, DWORD dwCompletionKey) {

    HANDLE h = CreateIoCompletionPort(hDevice, hCompletionPort,
        dwCompletionKey, 0);
    return(h == hCompletionPort);
}
```

Функция *AssociateDeviceWithCompletionPort* добавляет элемент в список устройств существующего порта завершения ввода-вывода. Она принимает описатель существующего порта (его возвращает вызов *CreateNewCompletionPort*), описатель устройства (файла, сокета, почтового ящика, канал и пр.) и ключ завершения (это значение, которое имеет смысл для разработчика, системе же безразлично, что передается в этом параметре). Каждый раз при связывании устройства с портом система добавляет эти сведения в список устройств порта завершения ввода-вывода.

> **Примечание.** Функция *CreateIoCompletionPort* слишком сложна, поэтому я рекомендую логически разделить две ее возможности. Однако у столь сложной функции есть одно преимущество: она позволяет в один прием создать порт завершения ввода-вывода и связать с ним устройство. Например, следующий код открывает файл и создает связанный с ним порт завершения ввода-вывода. У всех адресованных этому файлу запросов ввода-вывода будет ключ завершения CK_FILE, при этом возможно одновременное исполнение до двух потоков.

```
#define CK_FILE    1
HANDLE hFile = CreateFile(…);
HANDLE hCompletionPort = CreateIoCompletionPort(hFile, NULL, CK_FILE, 2);
```

Вторая структура данных — очередь завершения ввода-вывода. После завершения асинхронного запроса ввода-вывода на устройстве система проверяет, не связано ли это устройство с портом завершения ввода-вывода. Если это так, система добавляет в очередь завершения ввода-вывода этого порта элемент, представляющий обработанный запрос. В каждом элементе этой очереди содержатся сведения о числе переданных байтов, значение ключа завершения, заданное при связывании устройства с портом, указатель на структуру OVERLAPPED запроса и код ошибки. Об удалении элементов из этой очереди я расскажу чуть позже.

> **Примечание.** Возможна генерация запросов ввода-вывода на устройстве без постановки элементов в очередь порта завершения ввода-вывода. Обычно это не требуется, но иногда удобно отказаться от уведомлений, например при пересылке данных через сокеты.

Чтобы сгенерировать запрос ввода-вывода без постановки элемента в очередь завершения, необходимо записать в элемент *hEvent* структуры OVERLAPPED допустимый описатель события, а затем обработать побитовой операцией OR этот описатель и 1.

```
Overlapped.hEvent * CreateEvent(NULL, TRUE, FALSE, NULL);
Overlapped.hEvent = (HANDLE) ((DWORD_PTR) Overlapped.hEvent | 1);
ReadFile(…, &Overlapped);
```

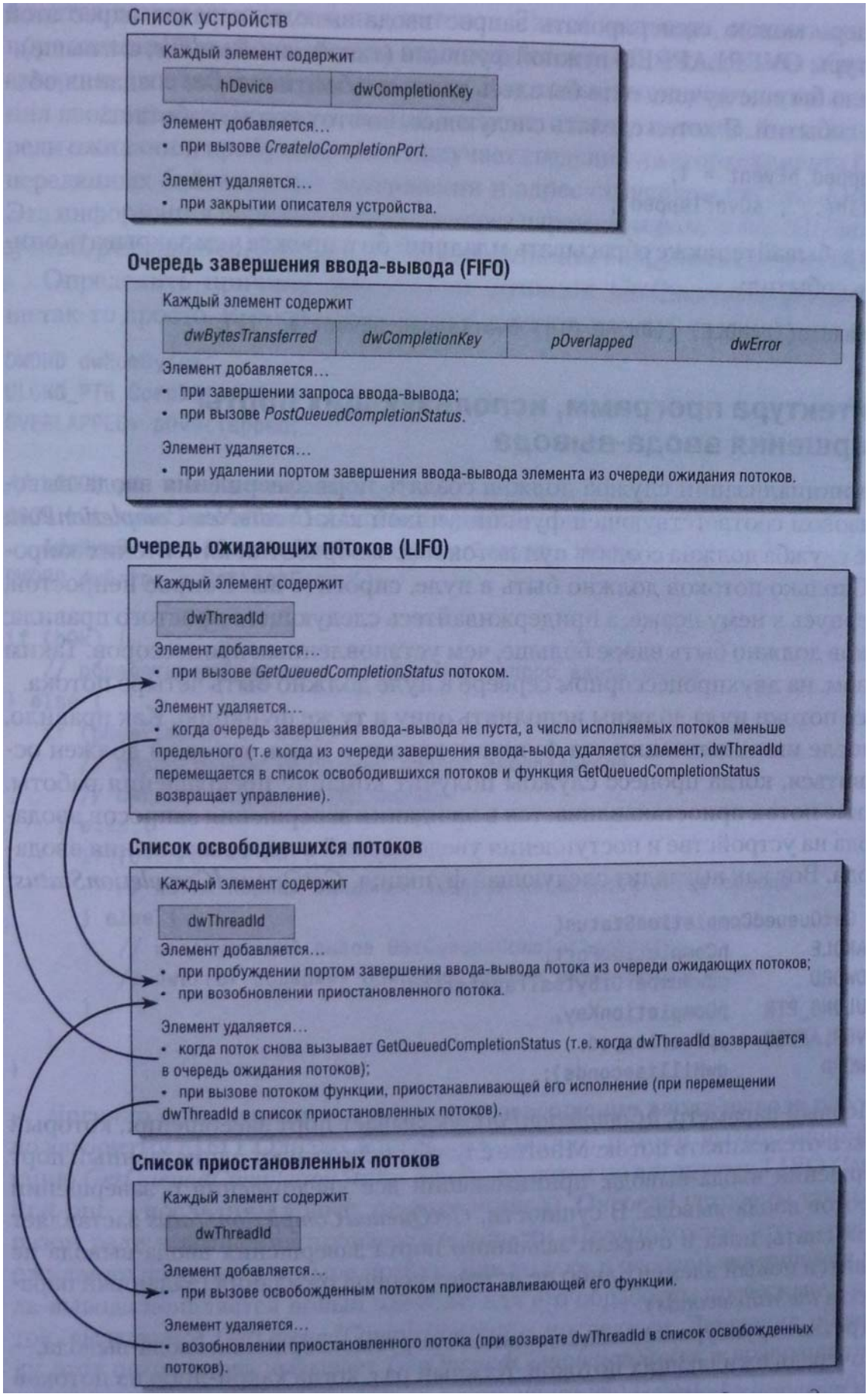

Рис. 10-1. *Внутреннее устройство порта завершения ввода-вывода*

Теперь можно сгенерировать запрос ввода-вывода, передав адрес этой структуры OVERLAPPED нужной функции (такой, как *ReadFile;* см. выше).

Было бы еще лучше, если бы здесь удалось обойтись и без создания объектов-событий. Я хотел сделать следующее, но этот код не работает:

```
Overlapped.hEvent = 1;
ReadFile(…, &Overlapped);
```

Не забывайте также сбрасывать младший бит, прежде чем закрывать описатель события:

```
CloseHandle((HANDLE) ((DWORD.PTR) Overlapped.hEvent & "1));
```

Архитектура программ, использующих порты завершения ввода-вывода

При инициализации служба должна создать порт завершения ввода-вывода вызовом соответствующей функции, такой как *CreateNewCompletionPort*. Далее служба должна создать пул потоков для обработки клиентских запросов. Сколько потоков должно быть в пуле, спросите вы. Вопрос непростой, и я вернусь к нему позже, а придерживайтесь следующего простого правила: потоков должно быть вдвое больше, чем установленных процессоров. Таким образом, на двухпроцессорном сервере в пуле должно быть четыре потока.

Все потоки пула должны исполнять одну и ту же функцию. Как правило, эта после инициализации эта функция входит в цикл, который должен остановиться, когда процесс службы получит команду прекращения работы. В цикле поток приостанавливается в ожидании завершения запросов ввода-вывода на устройстве и поступления уведомлений в порт завершения ввода-вывода. Вот как выглядит следующая функция, *GetQueuedCompletionStatus*:

```
BOOL GetQueuedCompletionStatus(
    HANDLE          hCompletionPort,
    PDWORD          pdwNumberOfBytesTransferred,
    PULONG_PTR      pCompletionKey,
    OVERLAPPED**    ppOverlapped,
    DWORD           dwMilliseconds);
```

Первый параметр, *hCompletionPort*, указывает порт завершения, который должен отслеживать поток. Многие службы используют единственный порт завершения ввода-вывода, принимающий все уведомления о завершении запросов ввода-вывода. В сущности, *GetQueuedCompletionStatus* заставляет поток спать, пока в очереди заданного порта завершения ввода-вывода не появится новый элемент либо не истечет период ожидания (заданный параметром *dwMilliseconds*).

Третья структура данных, связанная с портом завершения ввода-вывода, — это очередь ожидающих потоков. Каждый раз» когда какой-либо из потоков пула вызывает функцию *GetQueuedCompletionStatus*, его идентификатор за-

носится в очередь ожидающих потоков. Так объект ядра, представляющий порт завершения ввода-вывода отслеживает потоки, ожидающие обработки завершенных запросов ввода-вывода. Как только в очереди порта завершения ввода-вывода появится новый элемент, один из потоков, стоящих в очереди ожидания, пробуждается и получает сведения из этого элемента (число переданных байтов, ключ завершения и адрес структуры OVERLAPPED). Эта информация передается потоку через параметры *pdwNumberOfBytesTransferred*, *pCompletionKey* и *ppOverlapped* функции *GetQueuedCompletionStatus*.

Определить причину завершения функции *GetQueuedCompletionStatus* не так-то просто. Вот как это делается правильно:

```
DWORD dwNumBytes;
ULONG_PTR CompletionKey;
OVERLAPPED* pOverlapped;

// hIOCP уже инициализирована
BOOL bOk = GetQueuedCompletionStatus(hIOCP,
    &dwNumBytes, &CompletionKey, &pOverlapped, 1000);
DWORD dwError = GetLastError();

if (bOk) {
    // обрабатываем успешно завершенный запрос ввода-вывода
} else {
    if (pOverlapped != NULL) {
        // обрабатываем неудачный запрос ввода-вывода
        // dwError содержит код ошибки
    } else {
        if (dwError == WAJT_TIME0UT) {
            // истек период ожидания завершения запроса ввода-вывода
        } else {
            // недопустимый вызов GetQueuedCompletionStatus
            // dwError содержит описание причин неудачного вызова
        }
    }
}
```

Логично предположить, что очередь завершения ввода-вывода работает по алгоритму FIFO. Однако, вопреки ожиданиям, потоки, вызывающие функцию *GetQueuedCompletionStatus*, пробуждаются по алгоритму LIFO (last-in first-out — последним вошел, первым вышел). Очереди устроены таким образом ради повышения производительности. Предположим, что в очереди ожидания находится четыре потока. Как только в очереди завершения ввода-вывода появляется новый элемент, для его обработки пробуждается поток, вызвавший *GetQueuedCompletionStatus* последним. Завершив обработку, этот поток снова вызывает *GetQueuedCompletionStatus* и возвращается в очередь ожидания. Если в очереди завершения ввода-вывода появится еще один элемент, для его обработки проснется тот же самый поток.

Таким образом, если запросы ввода-вывода обрабатываются так медленно, что для их обработки достаточно единственного потока, система будет пробуждать один и тот же поток, а остальные три так и не проснутся. Благодаря использованию алгоритма LIFO, страницы памяти, занятые ресурсами (такими как стек) потоков, не получающих процессорное время, могут выкачиваться из оперативной памяти в страничный файл на жестком диске и сбрасываться из кэша процессора. Так что существование множества потоков, ожидающих поступления уведомлений в порт завершения ввода-вывода, не так уж сильно снижает быстродействие. Если ожидающих потоков много, а обработанных запросов ввода-вывода мало, большинство ресурсов простаивающих потоков, скорее всего, будет выгружено в страничный файл.

Если ожидается, что серверное приложение будет постоянно получать множество запросов ввода-вывода, в Windows Vista не обязательно плодить потоки, которые в основном ждут прихода уведомлений в порт завершения ввода-вывода и тратят системные ресурсы на переключения контекста. Вместо этого можно обрабатывать запросы группам, вызывая следующую функцию:

```
BOOL GetQueuedCompletionStatusEx(
    HANDLE hCompletionPort,
    LPOVERLAPPED_ENTRY pCompletionPortEntries,
    ULONG ulCount,
    PULONG pulNumEntriesRemoved,
    DWORD dwMilliseconds,
    BOOL bAlertable);
```

Первый параметр, *hCompletionPort*, указывает порт завершения ввода-вывода, который поток должен отслеживать. При вызове этой функции из очереди указанного порта извлекаются элементы, затем их содержимое копируется в массив, содержащийся в параметре *pCompletionPortEntries*. Параметр *ulCount* указывает число элементов этого массива, а в long-значение, указанное параметром *pulNumEntriesRemoved*, записывается, сколько запросов извлечено из очереди завершения ввода-вывода.

Элемент массива *pCompletionPortEntries* — это структура **OVERLAPPED_ENTRY**, в которой хранятся данные, составляющие элемент очереди завершения ввода-вывода: ключ завершения, адрес структуры OVERLAPPED, код ошибки запроса ввода-вывода и число переданных байтов.

```
typedef struct _OVERLAPPED_ENTRY {
    ULONG_PTR lpCompletionKey;
    LPOVERLAPPED lpOverlapped;
    ULONG_PTR Internal;
    DWORD dwNumberOfBytesTransferred;
} OVERLAPPED_ENTRY, *LPOVERLAPPED_ENTRY;
```

Поле *Internal* незадокументировано, поэтому не следует его использовать.

Последний параметр, *bAlertable,* устанавливается в FALSE. Функция ожидает постановки в очередь порта завершенных запросов ввода-вывода до истечения периода ожидания (заданного параметром *dwMilliseconds).* Если параметр *bAlertable* установлен в TRUE и в очереди нет завершенных запросов, поток переходит в тревожное состояние (об этом см. выше).

> **Примечание.** При генерации асинхронного запроса ввода-вывода на устройстве, связанном с портом завершения, Windows заносит результаты запроса в очередь этого порта. Windows поступает так, даже если асинхронный запрос был выполнен синхронно, дабы не нарушать логическое единство модели программирования. Однако за это приходится расплачиваться небольшим снижением производительности, поскольку информация о выполненном запросе должна быть передана в порт завершения ввода-вывода, а потому получена оттуда потоком.
>
> Можно немного повысить производительность, запретив Windows ставить в очередь порта завершения асинхронные запросы, выполненные синхронно. Для этого следует вызвать функцию *SetFileCompletionNotificationModes*, передав ей флаг FILE_SKIP_COMPLETION_PORT_ON_SUCCESS (см. выше). Разработчики, для которых производительность особенно важна, могут воспользоваться функцией *SetFileloOverlappedRange* (о ней можно прочитать в документации Platform SDK).

Как порт завершения ввода-вывода управляет пулом потоков

А теперь пришло время поговорить о достоинствах портов завершения ввода-вывода. При создании порта прежде всего необходимо указать предельное число одновременно работающих потоков. Как сказано выше, это число обычно равно числу процессоров компьютера, на котором предполагается использовать эту программу. После постановки в очередь выполненных запросов порт завершения ввода-вывода должен пробудить спящие потоки для их обработки. Однако число разбуженных потоков будет не больше, чем задано при создании порта. То есть, если в очередь будет поставлено четыре выполненных запроса ввода-вывода и четыре потока находятся в состоянии ожидания после вызова *GetQueuedCompletionStatus*, проснутся только два потока из четырех, остальные продолжат ожидание. Обработав выполненный запрос, поток снова вызывает *GetQueuedCompletionStatus*. Если система видит, что в очереди завершения ввода-вывода еще есть элементы, она пробуждает для их обработки те же самые потоки.

Если вдуматься в принцип работы этого механизма, он может показаться бессмысленным: порт завершения ввода-вывода разрешает одновременную работу ограниченного числа потоков, зачем тогда в пуле создаются дополнительные потоки, которые только и делают, что спят? Предположим для примера, что серверное приложение работает на компьютере с двумя про-

цессорами. Эта программа создает порт завершения ввода-вывода, разрешающий одновременную работу максимум двух потоков. Тем не менее, в пуле создается четыре потока (т.е. вдвое больше, чем процессоров). По-видимому, два из них так никогда и не проснутся для обработки данных.

Однако порты завершения ввода-вывода «умнее», чем вы думаете. Пробудив поток, порт завершения ввода-вывода заносит его идентификатор в четвертую структуру данных, составляющую этот порт, — список освобожденных потоков (см. рис. 10-1). Так порт отслеживает потоки, которые он пробуждает для обработки. Если освобожденный поток вызывает функцию, которая переводит его в состояние ожидания, это становится известно порту завершения, который переносит идентификатор потока из списка освобожденных потоков в список приостановленных потоков (это пятая и последняя из структур данных, составляющих порт завершения ввода-вывода).

Порт завершения ввода-вывода старается, чтобы число элементов списка освобожденных потоков было равно максимальному числу одновременно работающих потоков, указанному при создании порта. Если освобожденный поток по какой-либо причине переходит в состояние ожидания, число элементов списка освобожденных потоков уменьшается и порт пробуждает один из ожидающих потоков. Проснувшийся поток покидает список приостановленных и возвращается в список освобожденных потоков. В результате число элементов списка освобожденных потоков может превысить предельное число одновременно работающих потоков.

> **Примечание.** После вызова *GetQueuedCompletionStatus* поток «приписывается» к заданному порту завершения ввода-вывода. Система предполагает, что все приписанные к порту потоки будут обрабатывать его очередь. Однако число потоков пула, пробуждаемых портом для обработки, не больше предела, заданного при создании порта. Разорвать связь между потоком и портом завершения ввода-вывода можно:
> - завершив поток;
> - вызвав в потоке функцию *GetQueuedCompletionStatus*, передав ей описатель другого порта завершения ввода-вывода;
> - разрушив порт, к которому поток приписан в настоящее время.

А теперь подведем итоги. Допустим, что программа работает на двухпроцессорном компьютере, создает порт завершения ввода-вывода, допускающий одновременное пробуждение не более двух потоков, и пул с четырьмя потоками. При постановке в очередь трех выполненных запросов ввода-вывода в целях экономии времени на переключениях контекста порт пробудит только два потока. Если же один из активных потоков сгенерирует асинхронный запрос, вызовет *Sleep*, *WaitForSingleObject*, *WaitForMultipleObjects*, *SignalObjectAndWait* или другую функцию, которая выведет его из числа потоков, готовых к исполнению, порт заметит это и немедленно разбудит третий поток. Так порт завершения ввода-вывода выполняет свою задачу по предотвращении) простоя процессоров.

Рано или поздно первый поток вновь становится готовым к исполнению. При этом число готовых потоков становится больше числа процессоров в системе. Однако порт завершения ввода-вывода знает об этом и не позволит пробудить дополнительные потоки, пока число готовых потоков вновь не станет меньше числа процессоров. Порт завершения ввода-вывода спроектирован так, что число готовых к исполнению потоков недолго превышает предельное и быстро снижается по мере вызова потоками функции *Get Queued Completion Status.* Собственно, поэтому в пуле создается больше потоков, чем разрешено для одновременного исполнения портом завершения ввода-вывода.

Сколько потоков должно быть в пуле?

Сейчас уместно вернуться к вопросу о том, сколько потоков *должно* быть в пуле. При этом нужно учесть два момента. Во-первых, минимальное число потоков желательно создавать при инициализации службы, чтобы свести к минимуму затраты процессорного времени на создание и разрушение потоков. Во-вторых, необходимо ограничить число потоков для экономии системных ресурсов. Ресурсы простаивающих потоков выкачиваются из оперативной памяти в страничный файл, однако следует стремиться свести к минимуму бесполезную трату ресурсов, в том числе места в страничном файле.

Имеет смысл поэкспериментировать с числом потоков. Большинство служб (включая IIS) используют эвристические алгоритмы управления пулом потоков, я рекомендую вам тоже использовать подобные алгоритмы. Например, можно создать следующие переменные для управления пулом потоков:

```
LONG g_nThreadsMin;          // минимальное число потоков в пуле
LONG g_nThreadsMax;          // максимальное число потоков в пуле
LONG g_nThreadsCrnt;         // текущее число потоков в пуле
LONG g_nThreadsBusy;         // число занятых потоков в пуле
```

При инициализации вашего приложения можно создать столько потоков, сколько задано переменной *gjnThreadsMin*, все эти поток должны исполнять одну и ту же функцию. Ниже показан пример псевдокода функции потока из пула.

```
DWORD WINAPI ThreadPoolFunc(PVOID pv) {

    // добавляем поток в пул
    InterlockedIncrement(&g_nThreadsCrnt);
    InterlockedIncrement(&g_nThreadsBusy);

    for (BOOL bStaylnPool = TRUE; bStaylnPool;) {

        // приостанавливаем поток е ожидании выполненных запросов
        InterlockedDecrement(&m_nThreadsBusy);
        BOOL bOk = GetQueuedCompletionStatus(…);
```

```
        DWORD dwIOError = GetLastError();

        // поток занят
        int nThreadsBusy w InterlockedIncrement(&m_nThreadsBusy);

        // нужно ли добавить в пул еще один поток?
        if (nThreadsBusy == m_nThreadsCrnt) {      // все потоки заняты
            if (nThreadsBusy < m_nThreadsMax) {   // в пуле есть место для потока
                if (GetCPUUsage() < 75) {   // процессор загружен меньше, чем на 75X

                    // добавляем поток в пул
                    CloseHandle(chBEGINTHBEADEX(…));
                }
            }
        }

        if (!b0k && (dwIOError == WAIT_TIMEOUT)) {   // срок ожидания потока закон-
            чился
            // Сервер загружен слабо, следовательно этот поток можно завершить,
            // даже если не все запросы ввода-вывода обработана
            bStayInPool = FALSE;
        }

        if (bOk || (po != NULL)) {
            // пробуждаем поток и выполняем обработку
            …

            if (GetCPUUsage() > 90) {    // процессор загружен больше чем на 90X
                if (g_nThreadsCrnt > g_nThreadsMin)) {  // число потоков в пуле
                                                        // больше минимального
                    bStayInPool = FALSE;     // удаляем поток из пула
                }
            }
        }
    }

    // поток удаляется из пула
    InterlockedDecrement(&g_nThreadsBusy);
    InterlockedDecrement(&g_nThreadsCurrrent);
    return(O);
}
```

Этот пример демонстрирует возможности для творчества, которые открываются перед вами при использовании портов ввода-вывода. Функция *GetCPUUsage* не входит в Windows API, поэтому вы должны реализовать ее самостоятельно. Также необходимо следить, чтобы в пуле был хотя бы один поток, иначе клиенты не получат обслуживание. Используйте показанный

выше пример как образец при написании собственных служб. Однако чтобы добиться прироста производительности службы на ваших конкретных задачах, может потребоваться изменить его структуру.

> **Примечание.** Как сказано выше, при завершении потока система автоматически отменяет все необработанные запросы ввода-вывода, сгенерированные этим потоком. Действительно, в прежних версиях Windows (до Vista) поток, сгенерировавший запрос к устройству, связанному с портом завершения ввода-вывода, нельзя было завершить до исполнения этого запроса, иначе система отменяла все запросы этого потока. В Windows Vista поток может сгенерировать запросы ввода-вывода и спокойно завершиться, не дожидаясь его исполнения: его запросы будут исполнены и соответствующие уведомления будут направлены в порт завершения ввода-вывода.

У многих служб имеются утилиты, предоставляющие администраторам возможности по управлению пулом потоков. Они позволяют, например, устанавливать максимальное и минимальное число потоков, предельные значения загруженности процессора, а также максимальное число одновременно работающих потоков для порта завершения ввода-вывода.

Эмуляция выполненных запросов ввода-вывода

Вовсе не обязательно применять порты завершения ввода-вывода исключительно для работы с устройствами. Я уже говорил, что объект ядра «порт завершения ввода-вывода» — прекрасное средство для организации взаимодействия между потоками. Выше я показал функцию *QueueUserAPC*, которая позволяет потоку поставить APC-элемент в очередь другого потока. Порты завершения ввода-вывода поддерживают сходную функцию, *PostQueuedComptetionStatus*:

```
BOOL PostQueuedCompletionStatus(
    HANDLE      hCompletionPort,
    DWORD       dwNumBytes,
    ULONG_PTR   CompletionKey,
    OVERLAPPED* pOverlapped);
```

Эта функция ставит в очередь порта уведомление о завершении ввода-вывода. Первый ее параметр, *hCompletionPort*, определяет порт, в очередь которого следует добавить элемент. Остальные параметры, *dwNumBytes*, *CompletionKey* и *pOverlapped*, задают значения, возвращаемые вызовом *GetQueuedCompletionStatus*. Когда поток извлекает из очереди порта «эмулированный» элемент, функция *GetQueuedCompletionStatus* возвращает TRUE как при успешном исполнении запроса ввода-вывода.

Функция *PostQueuedCompletionStatus* невероятно полезное средство взаимодействия с потоками пула. Например, когда пользователь прекращает работу службы, все ее потоки должны корректно завершаться. Но потоки,

ждущие поступления уведомлений в порт ввода-вывода, не проснутся, пока не поступят новые запросы ввода-вывода. Вызвав *PostQueuedCompletionStatus* для каждого из потоков пула, можно пробудить спящие потоки и, если приложение прекращает работу (узнать об этом поможет функция *GetQueuedCompletionStatus*), заставить их освободить ресурсы и корректно завершиться.

Используйте только что показанный прием завершения работы потоков с осторожностью. Мой код работает, поскольку потоки пула завершаются и больше не вызывают *GetQueuedCompletionStatus*. Но если вам потребуется о чем-то уведомить потоки и заставить их снова вызывать *GetQueuedCompletionStatus*, вы столкнетесь с проблемой, поскольку потоки просыпаются по принципу LIFO. В таком случае придется воспользоваться дополнительными средствами синхронизации потоков, чтобы дать каждому потоку из пула обнаружить эмулированный элемент в очереди порта. Без этого потоки могут обнаруживать одни и те же уведомления по нескольку раз.

> **Примечание.** Если в Windows Vista вызвать функцию *CloseHandle* и передать ей описатель порта завершения ввода-вывода, все потоки, ожидающие после вызова *GetQueuedCompletionStatus*, проснутся и получат FALSE. При этом вызов *GetLastError* вернет ERROR_INVALID_HANDLE. Такое сообщение говорит потокам, что пришло время для их корректного завершения.

Программа-пример FileCopy

Показанная ниже программа FileCopy (10-FileCopy.exe) иллюстрирует использование портов завершения ввода-вывода. Исходный текст этой программы можно найти в каталоге 10-FileCopy внутри архива, доступного на веб-сайте поддержки этой книги. Эта просто копирует заданный пользователем файл в новый файл, FileCopy.cpy. При запуске FileCopy открывается окно, показанное на рис. **10-2.**

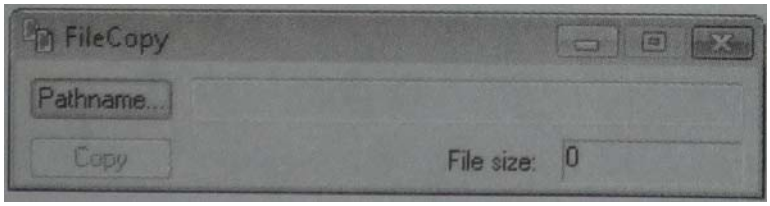

Рис. 10-2. *Окно программы FileCopy*

Пользователь щелкает кнопку и выбирает файл для копирования, после чего обновляются поля Pathname и File Size. По щелчку кнопки Copy программа вызывает функцию *FileCopy*, которая и выполняет основную работу. Познакомимся поближе с этой функцией.

При подготовке к копированию *FileCopy* открывает исходный файл и получает его размер в байтах. Я хотел, чтобы моя программ работала максимально быстро, поэтому файл открывается с флагом FILE_FLAG_NO_ BUFFERING. Так удается получить прямой доступ к файлу без дополни

тельных издержек, связанных с кэшированием, с помощью которого система «ускоряет» доступ к файлам. Естественно, выбрав прямой доступ к файлу, я взял на себя кое-какие дополнительные обязательства. Так, мне придется обращаться к файлу для чтения и записи только с использованием смещений, кратным размеру сектора дискового тома. Я решил передавать данные порциями по 64 Кб (это значение BUFFSIZE). Заметьте также, что исходный файл открывается с флагом FILE_FLAG_OVERLAPPED, что позволяет асинхронно исполнять адресованные ему запросы ввода-вывода.

Аналогичным образом открывается и целевой файл (с указанием флагов FILE_FLAG_NO_BUFFERING и FILE_FLAG_OVERLAPPED). При создании целевого файла я также передаю через параметр *hFileTemplate* функции *CreateFile* описатель исходного файла, чтобы новый файл получил атрибуты исходного.

> **Примечание.** После открытия обоих файлов для целевого файла сразу устанавливается максимальный размер (вызовами *SetFilePointerEx* и SetEndOfFile). Это очень важно, и вот почему: NTFS ставит особый указатель в месте завершения последней операции записи в файл. Попытка прочитать область за пределами этого указателя вернет нули. При записи в эту область система сначала заполнит область между старым и новым положениями указателя нулями, затем запишет в файл ваши данные и переставит указатель в новое положение. Это делается во избежание случайного доступа к старым данным, ранее записанным в этой области, согласно требованиям стандарта безопасности C2. Когда данные записываются рядом с концом файла, хранящегося в NTFS-разделе, система перемещает указатель записи. В этой ситуации NTFS приходится исполнять запросы ввода-вывода синхронно, даже если программа запрашивает асинхронный ввод-вывод. Так что если бы функция FileCopy не устанавливала бы размер целевого файла сразу, все асинхронные запросы исполнялись бы синхронно.

Когда нужные файлы уже открыты и подготовлены к обработке, *FileCopy* создает порт завершения ввода-вывода. Чтобы облегчить работу с портом, я написал маленький C++-класс, СЮСР. По сути, это простейшая оболочка для функций порта завершения ввода-вывода (см. файл IOCP.h и приложение А). Функция *FileCopy* создает порт завершения ввода-вывода путем создания экземпляра класса СЮСР (объекта с именем *iocp)*.

Исходный и целевой файлы связываются с портом завершения ввода-вывода с помощью вызова функции *AssociateDevice* класса *СЮСР*. При связывании с портом каждое устройство получает ключ завершения. Исходному файлу назначается ключ CK_READ, свидетельствующий об успешном завершении чтения, а целевому файлу — CK_WRITE, поскольку в этот файл данные будут записываться. Теперь все готово для инициализации структур OVERLAPPED и буферов запросов ввода-вывода. Функция *FileCopy* устроена так, что число необработанных запросов ввода-вывода не превышает

четырех (согласно значению MAX_PENDING_IO_REQS). В ваших собственных приложениях можно реализовать динамическую подстройку этого значения. В программе *FileCopy* запросы ввода-вывода инкапсулированы в классе *CIOReq*. Это C++-класс, производный от структуры OVERLAPPED и содержащий дополнительную контекстную информацию. Функция *FileCopy* создает массив объектов *CIOReq* и вызывает метод *AllocBuffer*, который связывает буферы с объектами, представляющими запросы ввода-вывода. Буферы создаются с помощью функции *VirtualAlloc,* их размер определяется значением BUFFSIZE. Применение *VirtualAlloc* гарантирует, что размер выделенного блока памяти и его начальный адрес будет кратным размеру сектора тома, что необходимо при использовании флага FILE_FLAG_NO_BUFFERING.

Я начинаю генерацию запросов ввода-вывода с небольшой хитрости, отправляя в порт завершения ввода-вывода четыре уведомления об исполнении запросов с ключом CK_WRITE. В результате ожидающий порт поток считает, что завершилась операция записи, немедленно пробуждается и отправляет запрос на чтение исходного файла — вот тут-то и начинается настоящее копирование файла.

Главный цикл завершается после обработки всех запросов ввода-вывода. Если необработанные запросы еще остались, главный цикл совершает очередной оборот и поток пытается перейти в состояние ожидания порта завершения ввода-вывода, вызывая метод *GetStatus* класса CIOCP (который, в свою очередь, вызывает *GetQueuedCompletionStatus*). Таким образом» поток спит, пока в порт не придет уведомление о завершении запроса ввода-вывода. После возврата управления функцией *GetQueuedCompletionStatus* выполняется проверка ключа завершения (*CompletionKey*). Если CompletionKey = CK_READ, запрос чтения исходного файла выполнен, и я вызываю метод *Write* класса *CIOReq*, генерирующий запрос записи в целевой файл. Если же *CompletionKey* = CK_WRITE, то выполнен запрос записи. Если конец исходного файла еще не достигнут, я вызываю метод *Read* класса *CIOReq*, чтобы продолжить чтение.

Когда необработанные запросы ввода-вывода закончатся, цикл завершится. Далее программа выполняет очистку, закрывая описатели исходного и целевого файлов. Прежде, чем функция *FileCopy* вернет управление, нужно сделать еще кое-что, а именно исправить размер целевого файла, чтобы он стал равным размеру исходного файла. Для этого я еще раз открываю целевой файл, но на этот раз без флага FILE_FLAG_NO_BUFFERING. Это позволит мне оперировать порциями данных любого размера, а не только кратными размеру сектора. Следовательно, я смогу установить для целевого файла размер, в точности соответствующий размеру исходного файла.

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


#include "..\CommonFiles\CmnHdr.h"           // See Appendix A.
#include "..\CommonFiles\IOCompletionPort.h" // See Appendix A.
#include "..\CommonFiles\EnsureCleanup.h"    // See Appendix A.

#include <WindowsX.h>
#include "Resource.h"

// C RunTime Header Files
#include <stdlib.h>
#include <malloc.h>
#include <memory.h>
#include <tchar.h>


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


// Each I/O Request needs an OVERLAPPED structure and a data buffer
class CIOReq : public OVERLAPPED {
public:
   CIOReq() {
      Internal = InternalHigh = 0;
      Offset = OffsetHigh = 0;
      hEvent = NULL;
      m_nBuffSize = 0;
      m_pvData = NULL;
   }

   ~CIOReq() {
      if (m_pvData != NULL)
         VirtualFree(m_pvData, 0, MEM_RELEASE);
   }

   BOOL AllocBuffer(SIZE_T nBuffSize) {
      m_nBuffSize = nBuffSize;
      m_pvData = VirtualAlloc(NULL, m_nBuffSize, MEM_COMMIT, PAGE_READWRITE);
      return(m_pvData != NULL);
   }

   BOOL Read(HANDLE hDevice, PLARGE_INTEGER pliOffset = NULL) {
      if (pliOffset != NULL) {
         Offset     = pliOffset->LowPart;
         OffsetHigh = pliOffset->HighPart;
      }
      return(::ReadFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
   }

   BOOL Write(HANDLE hDevice, PLARGE_INTEGER pliOffset = NULL) {
      if (pliOffset != NULL) {
         Offset     = pliOffset->LowPart;
         OffsetHigh = pliOffset->HighPart;
      }
```

```
      return(::WriteFile(hDevice, m_pvData, m_nBuffSize, NULL, this));
   }

private:
   SIZE_T m_nBuffSize;
   PVOID  m_pvData;
};


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


#define BUFFSIZE              (64 * 1024) // The size of an I/O buffer
#define MAX_PENDING_IO_REQS   4           // The maximum # of I/Os


// The completion key values indicate the type of completed I/O.
#define CK_READ  1
#define CK_WRITE 2


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


BOOL FileCopy(PCTSTR pszFileSrc, PCTSTR pszFileDst) {

   BOOL bOk = FALSE;    // Assume file copy fails
   LARGE_INTEGER liFileSizeSrc = { 0 }, liFileSizeDst;

   try {
      {
      // Open the source file without buffering & get its size
      CEnsureCloseFile hFileSrc = CreateFile(pszFileSrc, GENERIC_READ,
         FILE_SHARE_READ, NULL, OPEN_EXISTING,
         FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, NULL);
      if (hFileSrc.IsInvalid()) goto leave;

      // Get the file's size
      GetFileSizeEx(hFileSrc, &liFileSizeSrc);

      // Nonbuffered I/O requires sector-sized transfers.
      // I'll use buffer-size transfers since it's easier to calculate.
      liFileSizeDst.QuadPart = chROUNDUP(liFileSizeSrc.QuadPart, BUFFSIZE);

      // Open the destination file without buffering & set its size
      CEnsureCloseFile hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
         0, NULL, CREATE_ALWAYS,
```

```
         FILE_FLAG_NO_BUFFERING | FILE_FLAG_OVERLAPPED, hFileSrc);
      if (hFileDst.IsInvalid()) goto leave;

      // File systems extend files synchronously. Extend the destination file
      // now so that I/Os execute asynchronously improving performance.
      SetFilePointerEx(hFileDst, liFileSizeDst, NULL, FILE_BEGIN);
      SetEndOfFile(hFileDst);

      // Create an I/O completion port and associate the files with it.
      CIOCP iocp(0);
      iocp.AssociateDevice(hFileSrc, CK_READ);  // Read from source file
      iocp.AssociateDevice(hFileDst, CK_WRITE); // Write to destination file

      // Initialize record-keeping variables
      CIOReq ior[MAX_PENDING_IO_REQS];
      LARGE_INTEGER liNextReadOffset = { 0 };
      int nReadsInProgress  = 0;
      int nWritesInProgress = 0;

      // Prime the file copy engine by simulating that writes have completed.
      // This causes read operations to be issued.
      for (int nIOReq = 0; nIOReq < _countof(ior); nIOReq++) {

         // Each I/O request requires a data buffer for transfers
         chVERIFY(ior[nIOReq].AllocBuffer(BUFFSIZE));
         nWritesInProgress++;
         iocp.PostStatus(CK_WRITE, 0, &ior[nIOReq]);
      }

      BOOL bResult = FALSE;

      // Loop while outstanding I/O requests still exist
      while ((nReadsInProgress > 0) || (nWritesInProgress > 0)) {

         // Suspend the thread until an I/O completes
         ULONG_PTR CompletionKey;
         DWORD dwNumBytes;
         CIOReq* pior;
         bResult = iocp.GetStatus(&CompletionKey, &dwNumBytes, (OVERLAPPED**)
&pior, INFINITE);

         switch (CompletionKey) {
         case CK_READ:  // Read completed, write to destination
            nReadsInProgress--;
            bResult = pior->Write(hFileDst);  // Write to same offset read from
source
```

```
            nWritesInProgress++;
            break;

         case CK_WRITE: // Write completed, read from source
            nWritesInProgress--;
            if (liNextReadOffset.QuadPart < liFileSizeDst.QuadPart) {
               // Not EOF, read the next block of data from the source file.
               bResult = pior->Read(hFileSrc, &liNextReadOffset);
               nReadsInProgress++;
               liNextReadOffset.QuadPart += BUFFSIZE; // Advance source offset
            }
            break;
         }
      }
      bOk = TRUE;
      }
   leave:;
   }
   catch (...) {
   }

   if (bOk) {
      // The destination file size is a multiple of the page size. Open the
      // file WITH buffering to shrink its size to the source file's size.
      CEnsureCloseFile hFileDst = CreateFile(pszFileDst, GENERIC_WRITE,
         0, NULL, OPEN_EXISTING, 0, NULL);
      if (hFileDst.IsValid()) {

         SetFilePointerEx(hFileDst, liFileSizeSrc, NULL, FILE_BEGIN);
         SetEndOfFile(hFileDst);
      }
   }

   return(bOk);
}


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


BOOL Dlg_OnInitDialog(HWND hWnd, HWND hWndFocus, LPARAM lParam) {

   chSETDLGICONS(hWnd, IDI_FILECOPY);
```

```
   // Disable Copy button since no file is selected yet.
   EnableWindow(GetDlgItem(hWnd, IDOK), FALSE);
   return(TRUE);
}


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


void Dlg_OnCommand(HWND hWnd, int id, HWND hWndCtl, UINT codeNotify) {

   TCHAR szPathname[_MAX_PATH];

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

   case IDOK:
      // Copy the source file to the destination file.
      Static_GetText(GetDlgItem(hWnd, IDC_SRCFILE),
         szPathname, _countof(szPathname));
      SetCursor(LoadCursor(NULL, IDC_WAIT));
      chMB(FileCopy(szPathname, TEXT("FileCopy.cpy"))
         ? "File Copy Successful" : "File Copy Failed");
      break;

   case IDC_PATHNAME:
      OPENFILENAME ofn = { OPENFILENAME_SIZE_VERSION_400 };
      ofn.hwndOwner = hWnd;
      ofn.lpstrFilter = TEXT("*.*\0");
      lstrcpy(szPathname, TEXT("*.*"));
      ofn.lpstrFile = szPathname;
      ofn.nMaxFile = _countof(szPathname);
      ofn.lpstrTitle = TEXT("Select file to copy");
      ofn.Flags = OFN_EXPLORER | OFN_FILEMUSTEXIST;
      BOOL bOk = GetOpenFileName(&ofn);
      if (bOk) {
         // Show user the source file's size
         Static_SetText(GetDlgItem(hWnd, IDC_SRCFILE), szPathname);
         CEnsureCloseFile hFile = CreateFile(szPathname, 0, 0, NULL,
            OPEN_EXISTING, 0, NULL);
         if (hFile.IsValid()) {
            LARGE_INTEGER liFileSize;
            GetFileSizeEx(hFile, &liFileSize);
            // NOTE: Only shows bottom 32 bits of size
            SetDlgItemInt(hWnd, IDC_SRCFILESIZE, liFileSize.LowPart, FALSE);
```

```
         }
      }
      EnableWindow(GetDlgItem(hWnd, IDOK), bOk);
      break;
   }
}


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


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

   switch (uMsg) {
   chHANDLE_DLGMSG(hWnd, WM_INITDIALOG, Dlg_OnInitDialog);
   chHANDLE_DLGMSG(hWnd, WM_COMMAND,    Dlg_OnCommand);
   }
   return(FALSE);
}


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


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

   DialogBox(hInstExe, MAKEINTRESOURCE(IDD_FILECOPY), NULL, Dlg_Proc);
   return(0);
}


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