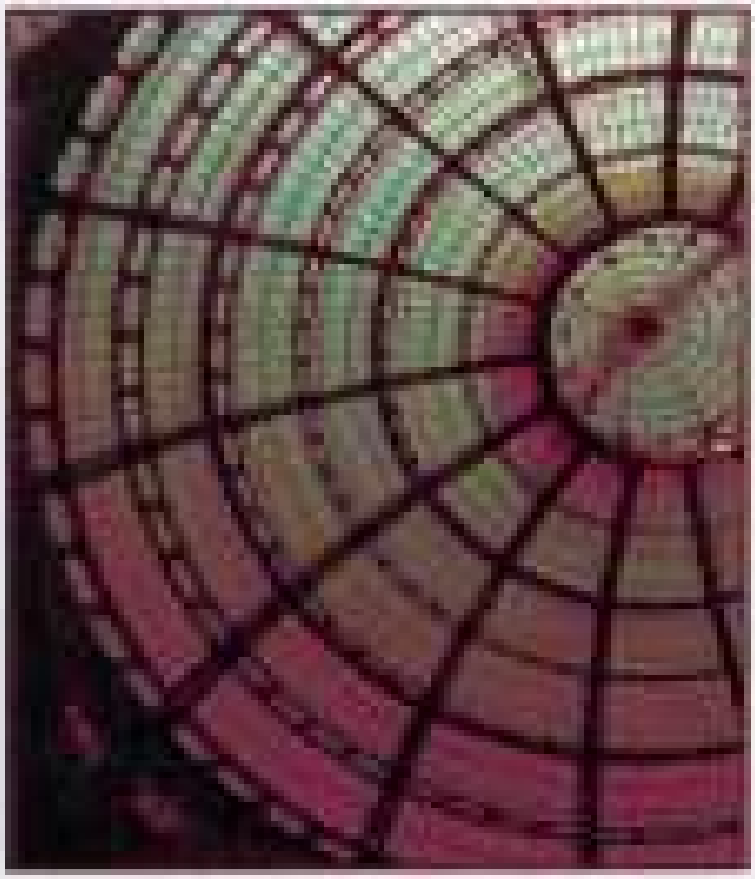
Системное программирование в среде Windows

Третье издание



- Хорошая отправная точка для эффективного освоения главных API-интерфейсов Windows, включая Win64
- Материал актуален для Windows Server 2003, Windows XP и Microsoft Visual Studio .NET Framework
- Приведено множество примеров, иллюстрирующих инструкции и концепции, а также позволяющих выбрать эффективное решение для повышения производительности работы программ

Джонсон М. Харт

Annotation

Эта книга посвящена вопросам разработки приложений с использованием интерфейса прикладного программирования операционных систем компании Microsoft (Windows 9x, Windows XP, Windows 2000 и Windows Server 2003). Основное внимание уделяется базовым системным службам, включая управление файловой системой, процессами и потоками, взаимодействие между процессами, сетевое программирование и синхронизацию. Рассматривается методика переноса приложений, написанных в среде Win32, в среду Win64. Подробно описываются все аспекты системы безопасности Windows и ее практического применения. Изобилие реальных примеров, доступных также и на Web-сайте книги, существенно упрощает усвоение материала.

Книга ориентирована на разработчиков и программистов, как высокой квалификации, так и начинающих, а также будет полезна для студентов соответствующих специальностей.

- 6
- 7
- 8
- 9
- 10
- 11
- 12
- 13
- 14
- 15
- 16
- 17
- 18
- 19
- 20
- 21
- 22
- 23
- 24
- 25
- 26
- 27
- 28
- 29
- 30
- 31
- 32
- 33
- 34
- 35
- 36

Джонсон М. Харт
Системное программирование в среде Windows
Третье издание

Введение

В этой книге описывается разработка приложений с использованием интерфейса прикладного программирования (Application Programming Interface, API) операционных систем Windows компании Microsoft, причем основное внимание уделяется базовым системным службам, включая управление файловой системой, процессами и потоками, межпроцессное взаимодействие, сетевое программирование и синхронизацию. Пользовательские интерфейсы, внутренние функции Windows и драйверы ввода/вывода в данной книге не рассматриваются, хотя сами по себе эти темы не менее важны и представляют не меньший интерес. Для примеров преимущественно выбирались реалистичные сценарии, и поэтому многие из них вполне могут служить в качестве основы для построения реальных приложений.

Win32/Win64 API, или обобщенно Windows API, поддерживаются семейством 32– и 64-разрядных операционных систем компании Microsoft, в которое в настоящее время входят Windows XP, Windows 2000 и Windows Server 2003. К числу ранних представителей этого семейства относятся операционные системы Windows NT, Windows Me, Windows 98 и Windows 95; в настоящее время эти системы считаются устаревшими, однако многие из приведенных в книге примеров программ способны выполняться и под их управлением. Вопросы перехода от платформы Win32 к развивающейся платформе Win64 обсуждаются по мере необходимости. Win64, поддерживаемый в качестве 64-разрядного интерфейса в некоторых версиях Windows Server 2003 и Windows XP, почти идентичен Win32.

Не вызывает сомнений, что Windows API является важнейшим фактором, который оказывает влияние на весь процесс разработки приложений, и во многих случаях вытесняет поддерживаемый операционными системами UNIX и Linux POSIX API, поскольку считается более предпочтительным или, по крайней мере, предоставляющим те же возможности для приложений, ориентированных на настольные и серверные системы. Поэтому многие опытные программисты заинтересованы в скорейшем изучении Windows API, и данная книга призвана содействовать этому.

Прежде всего, необходимо рассказать вам о том, что представляет собой Windows API, и показать, как им пользоваться в реальных ситуациях, причем этот рассказ должен быть как можно более кратким и не перегруженным излишними деталями. Поэтому данная книга предназначена не для использования в качестве справочного руководства, а для ознакомления с основными свойствами наиболее важных функций и демонстрации возможностей их применения в ситуациях практического программирования. Вооружившись этими знаниями, читатель сможет воспользоваться обширной справочной документацией, предоставляемой компанией Microsoft, для самостоятельного углубленного изучения отдельных вопросов, расширенных возможностей и менее приметных функций в соответствии с возникшими потребностями или заинтересованностью. Лично мне при таком подходе изучение Windows API далось легко, а разработка Windows-программ доставила огромное удовольствие, хотя и без неприятных минут также не обошлось. Мои порывы энтузиазма легко просматриваются в некоторых местах книги, что, собственно, и неудивительно. Впрочем, это вовсе не свидетельствует о том, что я безоговорочно соглашусь с превосходством Windows API над API других операционных систем (ОС), но относительно того, что у него есть масса положительных качеств, вряд ли кто-либо станет возражать.

Авторы многих книг, посвященных Windows, значительное внимание уделяют объяснению того, что представляют собой процессы, виртуальная память, межпроцессное взаимодействие, вытесняющий планировщик, но при этом не показывают, как все это используется в реальных

ситуациях. Программистам, имеющим опыт работы с системами UNIX, Linux, IBM MVS, Open VMS и некоторыми другими ОС эти понятия уже знакомы, и они заинтересованы лишь в том, чтобы как можно быстрее перейти к изучению того, как эти возможности реализованы в Windows. К тому же, в большинстве книг по Windows важное место отводится методам программирования на основе пользовательского интерфейса. С целью концентрации внимания лишь на самых главных базовых возможностях, предоставляемых системой, в данной книге тема пользовательского интерфейса не затрагивается, и мы ограничиваемся обсуждением лишь простого консольного символьного ввода/вывода.

В соответствии с принятой в данной книге точке зрения Windows — это всего лишь API операционной системы, предоставляющий набор вполне понятных средств. Потребность в ускоренном изучении Windows испытывают многие программисты, независимо от уровня их опыта, и без знания Windows немыслимо обсуждение таких, например, тем, как модель компонентного объекта (Component Object Model, COM), разработанная компанией Microsoft. В некоторых отношениях системы Windows превосходят остальные системы, в других — отстают от них или находятся примерно на том же уровне. Задача данной книги состоит в том, чтобы продемонстрировать, как эффективнее всего использовать эти возможности в реальных ситуациях для разработки полезных, высококачественных и высокопроизводительных приложений.

Потенциальная аудитория

• Все, кто хочет быстро научиться разрабатывать приложения, независимо от уровня подготовки.

• Программисты и специалисты по разработке программного обеспечения, перед которыми стоит задача переноса существующих приложений, написанных, в частности, для UNIX, на любую из платформ Windows. В книге демонстрируются сравнительные возможности функций и моделей программирования, связанных с использованием Windows, UNIX и стандартной библиотеки C. Каждая из обычных функциональных возможностей UNIX, включая управление процессами, синхронизацию, файловые системы и межпроцессное взаимодействие, рассматривается в терминах Windows.

• Читатели, приступающие к разработке новых проектов, которые не ограничены в своих действиях необходимостью переноса имеющихся программных кодов на другие платформы. В книге охвачены многие аспекты проектирования и реализации программ и продемонстрированы способы использования функций Windows для создания полезных приложений и решения обычных задач программирования.

• Программисты, использующие COM и .NET Framework, которые найдут здесь массу полезной информации, облегчающей изучение принципов работы динамически подключаемых библиотек (dynamic link libraries, DLL), моделей потоков и способов их применения, интерфейсов и синхронизации.

• Студенты, изучающие компьютерные дисциплины на старших курсах вузов или занятые подготовкой дипломных работ, связанных с системным программированием или разработкой приложений. Книга будет полезна также тем, кто изучает многопоточное программирование или сталкивается с необходимостью создания сетевых приложений. Ее также можно использовать в качестве полезного дополнения к таким, например, источникам, как книга У. Ричарда Стивенса (W. Richard Stevens) *Advanced Programming in the UNIX Environment* (см. библиографию), что позволит студентам сравнить возможности Windows и UNIX. Эта книга будет хорошим подспорьем и для студентов, проходящих курс ОС, поскольку в ней показано, какими именно средствами обеспечивается базовая функциональность ОС, представляющих интерес в коммерческом отношении.

Единственным допущением, которое неявно присутствует во всем вышесказанном, является предположение о том, что читатели имеют опыт программирования на языке C.

Изменения в третьем издании

Наряду со значительным обновлением и реорганизацией по сравнению с первыми двумя изданиями, в третьем издании добавлен обширный объем нового материала. Это издание призвано решать следующие задачи:

• Охватить новые возможности, появившиеся в Windows XP, Windows 2000 и Windows Server 2003, а также рассмотреть вопросы перехода к платформе Win64.

• Исключить материал, учитывающий специфику ОС Windows 95, Windows 98 и Windows ME (семейство "Windows 9x"), как устаревший, поскольку на поставляемых в настоящее время персональных системах устанавливается Windows XP, и ограничения, свойственные Windows 9x, уже потеряли свою актуальность.[1] В примерах программ без каких бы то ни было оговорок используются средства, которые входят лишь в текущие версии Windows, хотя в результате этого в Windows 9x некоторые программы работать не будут.

• Предоставить более полное освещение темы потоков и синхронизации, включая связанные с этим аспекты производительности, масштабируемости и надежности. Глава 9, равно как и некоторые из примеров в главе 10, являются новыми.

• Подчеркнуть все возрастающее влияние Windows 2000 и Windows Server 2003 и входящих в их состав новых средств на возможности высокопроизводительных, масштабируемых, многопоточных серверных приложений.

• Исследовать зависимость производительности программ от принципов их построения, обратив особое внимание на многопоточные программы с синхронизацией и на особенности эксплуатации этих программ в условиях симметричных многопроцессорных (Symmetrical Multiprocessor, SMP) систем.

• Учесть замечания читателей и студентов, касающиеся исправления недочетов и улучшения стиля изложения, а также все их советы и пожелания, как важные, так и самые незначительные.

Как организована эта книга

Расположение глав соответствует их тематической направленности, и поэтому сначала рассматриваются средства, необходимые для работы исключительно однопоточных приложений, затем средства, используемые для управления процессами и потоками, и лишь после этого обсуждаются вопросы сетевого программирования в среде с многопоточной поддержкой. Благодаря такой организации книги читателю будет легко следить за логикой изложения, последовательно переходя от файловых систем к управлению памятью и отображению файлов, далее — к процессам, потокам и синхронизации, а затем — к межпроцессному и сетевому взаимодействию и вопросам защиты приложений. Кроме того, такая организация позволяет естественным образом достраивать примеры по мере их усложнения, во многом подобно тому, как действует разработчик, когда сначала создает простой прототип, а затем постепенно вводит в него дополнительные возможности. Рассмотрение вопросов повышенной сложности, таких как асинхронный ввод/вывод и проблемы защиты, перенесено в самый конец книги.

В пределах каждой главы, после краткого обсуждения отдельной функциональности, например, управления процессами или отображения файлов, подробно рассматриваются наиболее важные из соответствующих функций Windows и их взаимосвязь между собой. Изложение сопровождается иллюстративными примерами. В основной текст включены лишь наиболее существенные части листингов программ; полные тексты программ, а также необходимые включаемые файлы, вспомогательные функции и прочий код приведены в приложении A или доступны на Web-сайте книги (http://www.awprofessional.com/titles/0321256190). Во всех случаях, когда какие-либо возможности поддерживаются только текущими версиями Windows (XP, 2000 и Server 2003) и не поддерживаются предыдущими, например, Windows 9x и Windows NT, в которых не реализованы многие из усовершенствованных возможностей, делаются отдельные оговорки. В каждой главе приводится список дополнительной рекомендованной литературы и предлагается несколько упражнений. Многие упражнения акцентируют внимание на вопросах, которые имеют важное значение и представляют определенный интерес, но не были отражены в основном тексте, тогда как другие упражнения заставляют читателя глубже разобраться в темах более сложного или специального характера.

В главе 1 предлагается высокоуровневое введение в семейство ОС Windows и Windows API. Используемая в качестве примера простая программа демонстрирует основные элементы стиля программирования Windows и создает фундамент для реализации усовершенствованных возможностей Windows. Win64 и проблемы межплатформенной миграции программ предварительно рассматриваются в главе 1, более полно изучаются в главе 16, и обсуждаются по мере изложения в остальной части книги там, где это оказывается необходимым.

В главах 2 и 3 рассматриваются файловые системы, операции консольного ввода/вывода, блокирование файлов и управление каталогами. В главе 2 также рассказывается о кодировке Unicode — расширенном символьном наборе, который используется в Windows. Соответствующие иллюстративные примеры охватывают последовательный и прямой доступ к содержащимся в файле данным, обход дерева каталогов и архивирование файлов. Глава 3 заканчивается обсуждением программного управления реестром, имеющего много общего с управлением файлами и каталогами.

Глава 4 знакомит читателя с обработкой исключений в Windows, включая структурную обработку исключений (Structured Exception Handling, SEH), которая будет широко

использоваться на протяжении всей книги. Во многих книгах изучение SEH откладывается до последних глав, однако мы, ознакомившись с этим средством уже на начальной стадии, получим возможность сразу же пользоваться им, что значительно упростит для нас некоторые задачи программирования и позволит повысить качество программ. Кроме того, здесь описано также одно из новейших средств — векторная обработка исключений.

В главе 5 рассмотрены вопросы управления памятью в Windows и показано, каким образом отображение файлов используется не только для упрощения программирования, но и для повышения производительности программ. В этой же главе рассмотрена организация библиотек DLL.

В главе 6 приводятся начальные сведения о процессах, управлении процессами и простых методах синхронизации в Windows. Далее в главе 7 эти понятия используются для описания управления потоками. Примеры в каждой из глав иллюстрируют многочисленные преимущества, включая упрощение программ и повышение их производительности, которые обеспечивает использование потоков и процессов.

Главы 8, 9 и 10 предлагают углубленный анализ одного из наиболее мощных средств Windows — синхронизации потоков. Синхронизация — сложная тема, и поэтому в данных главах содержатся многочисленные примеры и описания вполне понятных моделей, которые должны помочь читателю в полной мере воспользоваться преимуществами потоков для повышения эффективности программирования и производительности программ и вместе с тем обойти множество подводных камней. В эти главы включен новый материал, охватывающий вопросы повышения производительности и масштабируемости, что приобретает особое значение при создании серверных приложений, включая те из них, которые предположительно должны выполняться на SMP-системах.

Главы 11 и 12 посвящены межпроцессным и межпоточным взаимодействиям, а также сетевому программированию. В главе 11 главное внимание уделено средствам, являющимся частью Windows, а именно, анонимным каналам, именованным каналам и почтовым ящикам. В главе 12 обсуждаются сокеты Windows (Windows Sockets), обеспечивающие возможность взаимодействия с системами, не входящими в семейство Windows, посредством стандартных протоколов, главным образом, TCP/IP. И хотя интерфейс Windows Sockets, строго говоря, не является частью Windows API, он способен обеспечивать связь и взаимодействие через сети и Internet, так что предмет рассмотрения данной главы согласуется с остальной частью книги. На примере многопоточной клиент-серверной системы проиллюстрировано, каким образом можно обеспечить межпроцессное взаимодействие наряду с потоками.

В главе 13 показано, каким образом Windows позволяет превращать серверные приложения, аналогичные созданным в главах 11 и 12, в службы Windows (Windows Services), которыми можно управлять как фоновыми серверами. Преобразование сервера в службу требует внесения лишь незначительных изменений в программу.

В главе 14 показано, как осуществлять операции асинхронного ввода/вывода с использованием перекрывающегося ввода/вывода, а также событий и процедур завершения. Тех же результатов можно добиться и с помощью потоков, поэтому приводятся примеры, позволяющие сравнить различные решения с точки зрения простоты и производительности соответствующих программ. В то же время, создание масштабируемых серверов с многопоточной поддержкой требует привлечения портов завершения, использование которых иллюстрируется на примере серверов, созданных в предыдущих главах. Описаны также таймеры ожидания, рассмотрение которых требует привлечения понятий, введенных ранее в этой главе.

Глава 15 посвящена вопросам безопасности объектов Windows, а в качестве примера рассмотрена эмуляция системы защиты файлов в стиле UNIX, в которой для выполнения тех или

иных операций с файлом требуется наличие соответствующих полномочий. Дополнительные примеры иллюстрируют, какими средствами обеспечивается защита процессов, потоков и именованных каналов. Дополнения, обеспечивающие безопасность, могут быть введены после этого в ранее рассмотренные примеры.

Глава 16 завершает изложение основного материала рассмотрением вопросов программирования для Win64, а также обеспечения программной совместимости с этой платформой. После этого один из рассмотренных ранее примеров преобразуется к виду, допускающему перенос программы на платформу Win64.

Основной материал книги дополнен тремя приложениями. В приложении А содержатся описания программ, доступных на Web-сайте книги, и рекомендации по их использованию. Приложение Б содержит несколько таблиц, в которых функции Windows сравниваются с аналогичными им функциями, предоставляемыми системой UNIX и стандартной библиотекой С. В приложении В сравнивается производительность альтернативных вариантов реализации некоторых из примеров, приведенных в основном тексте, что позволяет читателю составить собственное представление о сравнительных достоинствах и недостатках средств Windows, как базовых, так и усовершенствованных, и средств, предоставляемых библиотекой С.

Сопоставление с UNIX и библиотекой C

По ходу изложения материала мы сопоставляем стилевые и функциональные особенности средств Windows и аналогичных им средств, входящих в UNIX (Linux) и стандартную библиотеку ANSI C. Как уже отмечалось, в приложении Б приведены таблицы, содержащие полный перечень сопоставимых функций. Включение этой информации мы сочли целесообразным, поскольку многие читатели знакомы с UNIX, и результаты сравнения обеих систем между собой, несомненно, будут представлять для них интерес. Те же, кто не знаком с системой UNIX, могут смело пропустить соответствующие разделы, которые, чтобы их было легче отличить, набраны мелким шрифтом и выделены отступами.

Примеры

При подготовке примеров автор руководствовался следующими соображениями:

• Примеры должны предоставлять образцы обычного, наиболее характерного и практически полезного применения функций Windows.

• Они должны соответствовать реальным ситуациям из сферы программирования, с которыми приходится сталкиваться в процессе разработки программного обеспечения, оказания консультаций и обучения. Некоторые из моих клиентов и слушателей использовали коды примеров при построении собственных систем. При оказании консультаций мне часто попадаются коды программ, аналогичные тем, которые включены в эту книгу, а в нескольких случаях ко мне приходили даже с кодами, непосредственно взятыми из первого или второго изданий. (Кстати, вы также можете использовать примеры из книги в своей работе, а если включите в документацию еще и благодарность в мой адрес, то я буду только рад.) Нередко эти коды встречались мне и в виде отдельных частей объектов COM или C++. Примеры, с учетом ограничений на время их подготовки и допустимый объем кода, приближены к "реальной жизни" и решают "реальные" задачи.

• Примеры должны подчеркивать фактическое поведение и взаимодействие функций, которые не всегда совпадают с тем, чего можно было бы ожидать после прочтения документации. В этой книге и текст, и примеры фокусируют внимание не на самих функциях, а на том, как они взаимодействуют между собой.

• Программные коды примеров должны строиться по принципу их постепенного наращивания и расширения для добавления новых функциональных возможностей в предыдущее решение простым и понятным способом, а также демонстрировать альтернативные методики реализации.

• Многие из примеров в нескольких первых главах реализуют такие команды UNIX, как ls, touch, chmod и sort, и тем самым представляют функции Windows в знакомом для части читателей контексте, одновременно создавая полезный набор вспомогательных функций. [2] Кроме того, наличие разных вариантов реализации одной и той же команды упрощает оценку преимуществ в отношении производительности, достигаемых за счет использования усовершенствованных средств Windows. Соответствующие результаты тестирования приведены в приложении B.

Примеры, приводимые в начале книги, отличаются, как правило, небольшой длиной программ, однако по мере усложнения материала в последующих главах размеры иллюстративного программного кода в необходимых случаях существенно возрастают.

В упражнениях, представленных в конце каждой главы, читателю предлагается разработать альтернативные варианты решений, самостоятельно исследовать рекомендуемые темы или ознакомиться с дополнительными функциональными возможностями, которые, несмотря на существующий к ним интерес, не могут быть подробно рассмотрены в данной книге. Некоторые из упражнений весьма просты, в то время как с другими у вас могут возникнуть затруднения. Нередко вашему вниманию предлагаются также явно неудачные решения, ибо выявление и устранение ошибок предоставит вам прекрасные возможности для оттачивания своего мастерства.

Все примеры отлажены и протестированы в средах операционных систем Windows XP, Windows 2000 и Windows Server 2003.– В необходимых случаях тестирование проводилось под управлением операционных систем Windows 9x и Windows NT. И хотя для разработки программ в основном использовались однопроцессорные системы на базе процессоров Intel, большинство

программ тестировались также на многопроцессорных системах. При тестировании приложений с клиент-серверной архитектурой использовались одновременно несколько клиентов, взаимодействующих с сервером. Тем не менее, никогда нельзя с полной уверенностью заявлять о корректности или завершенности программ и их пригодности для тех или иных целей. Несомненно, даже простейшие примеры могут иметь недостатки и при определенных обстоятельствах вообще не работать — такова участь почти любого программного обеспечения. Поэтому автор будет искренне благодарен всем, кто пришлет сообщения о любых дефектах, обнаруженных в программах, а еще лучше — об ошибках.

Web-сайты, посвященные этой книге

На Web-сайте книги (http://www.awprofessional.com/titles/0321256190) находится загружаемый файл, содержащий весь программный код и проекты для всех примеров, которые приведены в книге, решения ряда упражнений, альтернативные варианты реализации некоторых примеров, указания, а также результаты тестовых оценок производительности. Эта информация по мере надобности периодически обновляется с целью включения в нее нового материала и внесения необходимых исправлений.

На моем персональном Web-сайте (http://www.world.std.com/~jmhart/windows.htm) вы найдете список опечаток, обнаруженных в книге, а также дополнительные примеры, письма читателей и дополнительные объяснения, не считая многого другого. Сюда же включены слайды PowerPoint, которые могут быть использованы в некоммерческих учебных целях. Этими слайдами уже воспользовались слушатели многих профессиональных курсов, но они вполне пригодны также для использования в колледжах.

По мере выявления недостатков и ошибок и получения откликов читателей этот материал будет периодически обновляться. В случае возникновения каких-либо затруднений при работе с программами или любым другим материалом, содержащимся в книге, посетите сначала указанные сайты, где вам, возможно, удастся найти необходимые объяснения или получить информацию об обнаруженных к этому времени ошибках. Если же подобная попытка получения ответа на интересующий вас вопрос окажется безрезультатной, обращайтесь непосредственно ко мне по следующему адресу электронной почты: jmhart@world.std.com.

Благодарности

Во время подготовки третьего издания множество людей оказывали мне действенную помощь, делились советами или просто поддерживали добрым словом, а читатели подсказали целый ряд ценных идей и замечаний. На Web-сайте автора выражена горячая признательность всем тем, чьи советы и замечания нашли свое отражение в третьем издании книги, тогда как в первых двух изданиях содержатся благодарности в адрес тех, кто еще раньше дал нам ценные советы. Кроме того, прекрасный подробный анализ содержания книги был дан в недавних рецензиях Вагифа Абилова (Vagif Abilov), Билла Дрейпера (Bill Draper), Хорста Д. Клаузена (Horst D. Clausen), Майкла Девидсона (Michael Davidson), Дэниела Джанга (Daniel Jiang), Эрика Ландеса (Eric Landes), Клауса Х. Пробста (Klaus H. Probst) и Дугласа Рейли (Douglas Reilly), которые отнеслись к этой работе с гораздо большим вниманием, чем того требовали бы одни формальные обязанности; их советы и рекомендации заслуживают самой глубокой благодарности, и мне лишь остается надеяться, что с не меньшим вниманием и я отнесся к результатам их труда. Отдельной благодарности заслуживают мои друзья из ArrAy Inc.; у них я многому научился.

Анни Х. Смит (Anne H. Smith), выполнявшая верстку, приложила все свое мастерство, настойчивость и терпение, готовя книгу к публикации; без ее вклада выход книги в свет был бы просто невозможен. Элисса Армер (Elissa Armour), готовившая макеты для первых двух изданий, тем самым заложила фундамент и для настоящего Издания, сделав этот переход как нельзя более гладким.

Криста Мидоубрук (Chrysta Meadowbrooke), редактор рукописи, значительно улучшила точность, ясность и связность изложения материала. Ее внимательное отношение к содержанию книги, острые вопросы и проницательность помогли глубже осветить целый ряд вопросов.

Сотрудники издательства Addison-Wesley Professional проявили такой профессионализм и знание дела, что работать с ними было сплошным удовольствием. Стефани Накиб (Stephane Nakib), редактор, и Карен Гетман (Karen Gettman), главный редактор, работали над проектом с самого начала, когда надо, торопили меня, устраняли все помехи в работе и следили за тем, чтобы я ни на йоту не отклонялся от рабочего графика. Эбони Хейт (Ebony Haight), помощник редактора, осуществлял общее руководство всеми этапами работы, а производственная группа Джона Фуллера (John Fuller) и Патрика Кэш-Петерсона (Patrick Cash-Peterson), координатора производства, заставили считать, что с производственным процессом не могут быть связаны никакие сложности.

Эта книга посвящается горячо любимым Бобу (Bob) и Элизабет (Elizabeth).

Джонсон (Джон) М. Харт

(Johnson (John) M. Hart)

jmhart@world.std.com

Август 2004 года.

От издательства

Вы, читатель этой книги, и есть главный ее критик и комментатор. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересно услышать и любые другие замечания, которые вам хотелось бы высказать в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо, либо просто посетить наш Web-сервер и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится или нет вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Посылая письмо или сообщение, не забудьте указать название книги и ее авторов, а также ваш обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию последующих книг.

Наши координаты:
E-mail: info@williamspublishing.com
WWW: http://www.williamspublishing.com
Информация для писем из: России: 115419, Москва, а/я 783 Украины: 03150, Киев, а/я 152

ГЛАВА 1
Знакомство с Win32 и Win64

В этой главе вы познакомитесь с семейством операционных систем (ОС) Microsoft Windows и интерфейсом прикладного программирования (Application Programming Interface, API), который используется всеми членами этого семейства. Здесь также кратко описывается новейший 64-разрядный API Win64 и достаточно подробно обсуждается проблема переносимости программного обеспечения между Win32 и Win64. Для удобства изложения мы будем ссылаться, главным образом, просто на Windows и Windows API. Как правило, раздельные ссылки на Win32 и Win64 будут делаться лишь в тех случаях, когда различия между этими интерфейсами будут иметь существенное значение. Сориентироваться в том, что именно автор имеет в виду, когда говорит о Windows, — операционную систему или интерфейсе для разработки программ, — читателю поможет контекст.

Подобно API любой другой ОС, Windows API также располагает собственным набором соглашений и приемов программирования, укладывающихся в рамки философии Windows. Со стилем программирования Windows вы ознакомитесь на примере обычного копирования файлов, однако тот же стиль используется при управлении файлами, процессами, памятью, а также такими более развитыми средствами, как синхронизация потоков. Для упомянутого примера будет приведен также код, в котором используется стандартная библиотека С, что облегчит вам сравнение стиля программирования, принятого в Windows, с более распространенными стилями.

Мы начнем с общего обзора основных средств, предоставляемых любой современной операционной системой, чтобы понять, как эти средства используются в Windows.

Основные возможности операционных систем

Windows обеспечивает доступность базовых средств ОС в столь непохожих друг на друга системах, как мобильные телефоны, карманные устройства, переносные компьютеры и серверы масштаба предприятия. Возможности ОС можно охарактеризовать, рассмотрев наиболее важные ресурсы, которыми управляют современные операционные системы.

• **Память.** ОС управляет сплошным, или плоским (flat), виртуальным адресным пространством большого объема, перемещая данные между физической памятью и диском или иным накопительным устройством прозрачным для пользователя образом.

• **Файловые системы.** ОС управляет пространством именованных файлов, предоставляя возможности прямого и последовательного доступа к файлам, а также средства управления файлами и каталогами. Используемые в большинстве систем пространства имен являются иерархическими.

• **Именование и расположение ресурсов.** Файлы могут иметь длинные, описательные имена, причем принятая схема именования распространяется также на такие объекты, как устройства, а также объекты синхронизации или межпроцессного взаимодействия. Размещение именованных объектов и управление доступом к ним также являются прерогативой ОС.

• **Многозадачность.** ОС должна располагать средствами управления процессами, потоками и другими единицами, способными независимо выполняться в асинхронном режиме. Задачи могут планироваться и вытесняться в соответствии с динамически определяемыми приоритетами.

• **Взаимодействие и синхронизация.** ОС управляет обменом информацией между задачами и их синхронизацией в изолированных системах, а также взаимодействием сетевых систем между собой и сетью Internet.

• **Безопасность и защита.** ОС должна предоставлять гибкие механизмы защиты ресурсов от несанкционированного или непреднамеренного доступа и нанесения ущерба системе.

Microsoft Windows Win 32/Win64 API обеспечивает поддержку не только этих, но и множества других средств ОС, и делает их доступными в ряде версий Windows, некоторые из которых постепенно выходит из употребления, а некоторые поддерживает лишь то или иное подмножество полного API.

Эволюция Windows

Windows API поддерживается несколькими версиями Windows. Существование ряда различных версий Windows может вносить некоторую неразбериху, однако с точки зрения программиста все они аналогичны друг другу. В частности, все версии поддерживают подмножества *идентичных* Windows API. Программы, разработанные для одной системы, без особых трудностей будут выполняются и в любой другой, и при этом обеспечивается переносимость исходных, а в большинстве случаев — и бинарных кодов.

Каждая новая версия в определенной степени расширяла функциональные возможности API, хотя уже с самого начала API в целом отличался замечательной стабильностью. Ниже перечислены главные факторы, которые определяют эволюционное развитие Windows:

• **Масштабируемость.** Новые версии способны выполняться на более широком спектре систем, включая серверы масштаба предприятия, использующие оперативную память большого объема и запоминающие устройства повышенной емкости.

• **Интеграция.** Каждый новый выпуск системы интегрирует в себя элементы дополнительных технологий, такие, например, как использование мультимедийных данных или подключение к беспроводным сетям, Web-службы или самонастраивающиеся (plug-and-play) устройства. В целом, рассмотрение этих технологий выходит за рамки данной книги.

• **Простота использования.** Элементы улучшения внешнего вида графического рабочего стола или средства, упрощающие пользование системой, без труда обнаруживаются в каждом новом выпуске системы.

• **Совершенствование API.** За время своего существования API был дополнен новыми замечательными возможностями. Именно API является центральной темой данной книги.

Версии Windows

ОС Windows, в виде развивающейся последовательности версий, используется, начиная с 1993 года. Во время написания данной книги на Web-сайте компании Microsoft в качестве основных фигурировали следующие версии:

• **Windows XP,** включая выпуски Home, Professional и ряд других, которая ориентирована на индивидуальных пользователей. Большинство коммерческих PC, поступающих на рынок на сегодняшний день, включая лэптопы и ноутбуки, поставляются с уже установленной Windows XP соответствующего типа. В рамках данной книги различия между версиями, как правило, существенного значения не имеют.

• **Windows Server 2003,** которая выпускается в виде продуктов Small Business Server, Storage Server 2003, а также некоторых других, и ориентирована на управление приложениями для предприятий и серверными приложениями. В системах, работающих под управлением Windows Server 2003, часто применяется симметричная многопроцессорная обработка (Symmetric Multiprocessing, SMP), характеризующаяся использованием одновременно нескольких независимых процессоров. Новые 64-разрядные приложения, требующие Win64, появляются преимущественно на системах Windows Server 2003.

• **Windows 2000,** которая доступна в виде выпусков Professional и нескольких разновидностей выпусков Server и по-прежнему широко используется как в персональных, так и в серверных системах. Со временем Windows XP и будущие версии Windows вытеснят Windows 2000, продажа которой уже прекращена.

• **Windows Embedded, Windows CE и Windows Mobile,** которые представляют собой специализированные версии Windows, ориентированы на использование в малых системах, таких, например, как ручные (palmtop) и встроенные (embedded) устройства обработки данных, и предоставляют широкие подмножества возможностей Windows.

Устаревшие предыдущие версии Windows

В настоящее время продолжает использоваться ряд предыдущих версий Windows, которые считаются устаревшими и более не предлагаются или не поддерживаются компанией Microsoft, однако многие, хотя и не все, из обсуждаемых в данной книге примеров будут выполняться и под управлением этих систем. Перечень таких систем, постепенно выходящих из употребления, приводится ниже.

• **Windows NT** 3.5, 3.5.1 и 4.0, предшествовавшие современным версиям NT и восходящие своими корнями к 1993 году, из которых наибольшей популярностью пользовалась версия Windows NT 4.0, модернизированная с помощью пакета обновлений Service Pack 3 (SP3). Первоначально системы NT предназначались для профессиональных пользователей и серверов, в то время как версиями, распространяемыми для персональных и офисных нужд, служили версии Windows 9x (см. ниже). Первоначально Windows 2000 называлась Windows NT Version 5.0, в связи с чем многие пользователи, имея в виду системы Windows 2000, Windows Server 2003 или Windows XP, все еще употребляют названия Windows NT или Windows NT5. Несмотря на важные исключения, что особенно касается последних глав, под управлением NT Version 4.0 будут корректно, хотя и не всегда оптимально, работать почти все программы из числа тех, которые представлены в данной книге.

• **Windows 95, 98 и Windows ME** (или просто — **Windows 9x,** поскольку различия между указанными системами вряд ли могут считаться существенными), которые предназначались

главным образом для настольных (desktop) и переносных (laptop) компьютеров и не включали в себя, помимо прочего, средств безопасности Windows NT. В настоящее время эти системы вытесняются системой Windows XP, которая объединяет предоставляемые ими средства со средствами Windows NT. Многие примеры программ, хотя и не все, особенно из числа тех, которые приводятся в начальных главах книги, будут корректно выполняться и под управлением Windows 9x.

Возвращаясь назад в прошлое, можно отметить, что до появления Windows 95 доминировала 16-разрядная ОС Windows 3.1, графический пользовательский интерфейс (Graphical User Interface, GUI) которой можно считать предшественником современных Windows GUI. Вместе с тем, многие важнейшие возможности ОС, такие как истинная многозадачность, управление памятью и средства защиты, она не поддерживала.

Если обратиться к еще более отдаленному прошлому, к концу восьмидесятых годов прошлого столетия, то в качестве исходной "IBM PC-системы можно указать DOS. В DOS имелся всего-навсего простейший интерфейс командной строки, но DOS-команды используются и по сей день. В действительности, большинство из программ, рассматриваемых в данной книге в качестве примеров, написаны в виде консольных приложений и поэтому могут запускаться из командной строки, а для тестирования производительности даже специально предусмотрены пакетные файлы DOS.

Windows NT 5.x

По отношению к системам Windows 2000, Windows XP и Windows Server 2003 используют собирательное название Windows NT Version 5.x или просто NT5. В каждой из трех указанных систем эксплуатируется пятая версия ядра Windows NT, хотя младший номер версии ("x" в "5.x") может быть различным. Так, в Windows XP используется ядро версии NT 5.1.

Несмотря на то что многие программы будут выполняться и под управлением предыдущих версий Windows, мы, как правило, предполагаем применение версии NT5, некоторые возможности которой отсутствуют в предыдущих версиях. Поскольку любая современная Windows-система обладает возможностями NT5, такое предположение не чревато никакими осложнениями, но зато снимает любые ограничения на использование усовершенствованных возможностей ОС. Вместе с тем, на многих устаревших системах все еще могут оставаться установленными ранние версии Windows NT или Windows 9x, и поэтому программы примеров сразу же после запуска проверяют номер версии Windows и в необходимых случаях прекращают свою работу с выводом сообщения об ошибке.

В документации по Microsoft API приводятся требования к номеру версии, сформулированные в терминах NT, Windows (что в данном контексте относится к версиям 9x) и CE, и ряд других требований. В случае любого рода сомнений относительно возможности использования тех или иных функций API в конкретной версии Windows следует обратиться к документации.

Другие интерфейсы программирования для Windows

Windows (под этим термином, если отдельно не оговаривается иное, мы подразумеваем API Win32 и Win64, а также NT5) способна обеспечивать также поддержку среды других "подсистем", хотя этой возможностью пользуются редко и прямого отношения к тематике данной книги она не имеет. Ядро ОС NT имеет действительно надежную защиту от воздействия

со стороны приложений. Windows является для него лишь одной из нескольких возможных сред. Так, предоставляемый компанией Microsoft набор инструментальных средств Windows Resource Kit включает подсистему POSIX, но кроме того существуют подсистемы POSIX, предлагаемые в рамках проекта разработки программного обеспечения с открытым исходным кодом.

Поддержка процессоров

Хотя это не и не входит в сферу непосредственных интересов разработчика приложений, вам следует знать, что Windows поддерживает самые различные базовые процессоры и архитектуры систем, для чего предусмотрен уровень аппаратных абстракций (Hardware Abstraction Layer, HAL), который обеспечивает переносимость программ на системы с другой архитектурой процессора.

Windows выполняется преимущественно на процессорах семейства Intel x86, новейшими представителями которого являются процессоры Pentium и Xeon, а одним из предыдущих — Intel 486. Широкое распространение получили совместимые с ними процессоры компании Advanced Micro Devices (AMD). Кроме того, Windows изначально проектировалась таким образом, чтобы обеспечивалась независимость от типа процессора. Что немаловажно, Windows Server 2003 поддерживается процессором Intel Itanium, новая 64-разрядная архитектура которого самым радикальным образом отличается от классической архитектуры процессоров семейства x86.

Другими примерами независимости Windows от архитектуры процессоров, относящимися как к прошлому, так и к настоящему, могут служить следующие:

• Windows CE способна выполняться на целом ряде процессоров, не принадлежащих семейству x86.

• Windows NT первоначально поддерживалась Alpha-процессорами компании Digital Equipment Corporation (которая была приобретена сначала компанией Compaq, а затем компанией Hewlett Packard).

• 64-разрядные процессоры компании AMD Athlon 64 и Opteron (AMD64) обеспечивают 64-разрядное расширение архитектуры x86, отражающее иной подход по сравнению с тем, который используется в архитектуре Itanium.

• Недавно объявленные компанией Intel 32/64-разрядные процессоры будут представлять собой 64-разрядные расширения процессоров x86.

Воздействие Windows на ситуацию на рынке

Тот факт, что система Windows способна обеспечивать важнейшие функциональные возможности на нескольких платформах, вряд ли можно считать уникальным. В конце концов, этим могут похвастаться многие коммерческие ОС, не говоря уже об ОС с открытым исходным кодом, а UNIX[3] и Linux уже в течение длительного времени эксплуатируются на самых разнообразных системах. Между тем, использование Windows и разработка приложений для Windows сулят значительные преимущества как в коммерческом, так и в техническом отношении.

• Windows занимает на рынке, особенно на рынке настольных систем, господствующее положение, которое прочно удерживается ею в течение многих лет.[4] Поэтому перед приложениями Windows открыт огромный целевой рынок, емкость которого исчисляется десятками миллионов систем, вследствие чего остальные настольные системы, включая UNIX, Linux и Macintosh, играют на рынке значительно меньшую роль.

• Приложения Windows могут использовать GUI, знакомый десяткам миллионов пользователей, а для многих приложений предусмотрена возможность их адаптации, или "локализации", в соответствии с региональными требованиями, предъявляемыми к языку и внешнему виду интерфейса.

• Windows поддерживает SMP-системы. Сфера применения Windows не ограничивается настольными системами и охватывает как серверы масштаба подразделения и предприятия, так и высокопроизводительные рабочие станции.[5]

• Windows (правда, это не относится к Windows 9x и Windows CE) сертифицирована Управлением национальной безопасности (National Security Agency, NSA) как система, обеспечивающая уровень безопасности C2.

• Применение большинства других ОС, отличных от UNIX, Linux и Windows, ограничивается только системами, предоставляемыми единственным поставщиком.

• Операционные системы семейства Windows предлагают ряд возможностей, которые в стандартной системе UNIX отсутствуют, хотя и могут быть доступными в некоторых реализациях. В качестве примера можно привести систему безопасности уровня C2, а также службы NT Services.

Windows предоставляет функциональность современных ОС, а диапазон систем, которые могут работать под ее управлением, простирается от текстовых процессоров и почтовых программ до интегрированных систем предприятия и серверов крупных баз данных. На принятие решений относительно способа разработки Windows-приложений оказывают влияние как соображения технического порядка, так и корпоративные требования.

Windows, стандарты и открытые системы

Эта книга посвящена разработке приложений с использованием Windows API. Вполне естественно, что у программистов, воспитанных на UNIX и открытых системах, могут возникнуть следующие вопросы: "Является ли Windows открытой системой?", "Представляет ли собой Windows промышленный стандарт?", "Не является ли Windows всего лишь очередным патентованным API?" Ответы на эти вопросы во многом зависят от того, что именно понимается под определениями *открытая* (open), *промышленный стандарт* (industry standard) или *патентованный* (proprietary), а также от того, какие преимущества ожидаются от использования открытых систем.

Windows API полностью отличается от API стандарта POSIX, поддерживаемого системами Linux и UNIX. Windows не подчиняется стандарту X/Open, как не подчиняется и никакому другому открытому промышленному стандарту из тех, которые были предложены соответствующими органами стандартизации или промышленными консорциумами.

Windows контролируется единственным поставщиком. Хотя Microsoft и заявляет о своей готовности приспосабливаться к требованиям отрасли и учитывать их, в этих вопросах сама же она является арбитром и исполнителем в одном лице. Отсюда следует, что, помимо других преимуществ, пользователи Windows получают многие из выгод, которые обычно предлагают открытые стандарты.

• Унифицированные реализации быстрее достигают рынка.

• Отсутствуют какие-либо неожиданные фирменные "улучшения" или "расширения", с которыми потом приходится бороться программисту, хотя небольшие различия, существующие между различными платформами Windows, все же приходится учитывать.

• Вся совокупность полноценных ОС-продуктов, предлагающих все необходимые возможности, определена и реализована одним и тем же поставщиком. Разработчикам приложений остается решать только высокоуровневые задачи.

• Базовая аппаратная платформа является открытой. Разработчики могут выбирать любого из многочисленных поставщиков платформ по своему усмотрению.

Жаркие споры относительно того, к добру ли такая ситуация для пользователей и компьютерной индустрии в целом, или она только вредит общему делу, еще не закончились. Мы не будем пытаться участвовать в этом споре; задача данной книги состоит лишь в том, чтобы помочь разработчикам приложений как можно скорее приступить к работе в Windows.

В действительности системы Windows поддерживают многие важные стандарты. Так, Windows поддерживает стандартные библиотеки C и C+ и целый ряд открытых стандартов межплатформенного взаимодействия. В качестве примера можно привести сокеты Windows (Windows Sockets), предоставляющие стандартный интерфейс сетевого программирования, который обеспечивает возможность использования TCP/IP и других сетевых протоколов и тем самым открывает возможности доступа в Internet и взаимодействия с системами, не принадлежащими семейству Windows. То же самое остается справедливым и по отношению к протоколу удаленного вызова процедур (Remote Procedure Calls, RPC).[6] Системы самой различной природы могут связываться с высокоуровневыми системами управления базами данных (СУБД) при помощи языка структурированных запросов (SQL). Наконец, в общий круг предложений Windows входит поддержка Internet, обеспечиваемая Web-серверами и серверам иного рода. Windows поддерживает такие ключевые стандарты, как TCP/IP, а на активно действующем рынке поставщиков решений Windows вам предлагают приобрести за разумную плату множество других ценных дополнительных продуктов, в том числе клиенты и серверы X

Window.

Резюмируя, можно утверждать, что Windows поддерживает наиболее важные из стандартов межплатформенного взаимодействия, и хотя основной API является собственностью компании, он доступен по умеренной цене для широкого ряда систем.

Библиотеки совместимости

Несмотря на наличие библиотек совместимости (compatibility libraries), ими пользуются очень редко. Существуют две возможности.

• В системах на основе UNIX, Linux, Macintosh и некоторых других может быть развернута одна из библиотек совместимости Windows, например, эмулятор Windows с открытым исходным кодом Wine, что обеспечивает переносимость исходного кода из Windows.

• За счет использования программного обеспечения с открытым исходным кодом и набора инструментальных средств Windows Resource Kit компании Microsoft поверх подсистемы Windows может быть развернута библиотека совместимости POSIX. Весьма ограниченная по своим возможностям библиотека совместимости входит в состав среды визуальной разработки при ложений Microsoft Visual C++.

Таким образом, имеется, пусть даже и редко используемая, возможность выбора одного API и развертывания разработанных с его помощью переносимых приложений на системах Windows, POSIX и даже Macintosh.

Принципы, лежащие в основе Windows

Полезно никогда не забывать о некоторых базовых принципах Windows. В Windows API имеется множество как самых незаметных, так и значительных отличий от других API, таких как POSIX API, с которым знакомы программисты, работающие в UNIX и Linux. И хотя с применением Windows не связаны какие-либо специфические трудности в работе, она потребует от вас внесения некоторых изменений в привычные стиль и методику программирования.

Ниже описаны некоторые из важнейших характеристик Windows, с которыми вы ближе познакомитесь по мере дальнейшего изложения материала.

Многие системные ресурсы Windows представляются в виде *объектов ядра* (kernel objects), для идентификации и обращения к которым используются *дескрипторы* (handles). По смыслу эти дескрипторы аналогичны дескрипторам (descriptors) файлов и идентификаторам (ID) процессов в UNIX.[7]

• Любые манипуляции с объектами ядра осуществляются только с использованием Windows API. "Лазеек" для обхода этого правила нет. Подобная организация работы согласуется с принципами абстрагирования данных, используемыми в объектно-ориентированном программировании, хотя сама система Windows объектно-ориентированной не является.

• К объектам относятся файлы, процессы, потоки, каналы межпроцессного взаимодействия, объекты отображения файлов, события и многое другое. Объекты имеют атрибуты защиты.

• Windows — богатый возможностями и гибкий интерфейс. Во-первых, одни и те же или аналогичные задачи могут решаться с помощью сразу нескольких функций; так, имеются вспомогательные функции (convenience functions), полученные объединением часто встречающихся последовательностей функциональных вызовов в одну функцию (к числу подобных функций принадлежит и функция CopyFile, используемая в одном из примеров далее в этой главе). Во-вторых, функции часто имеют многочисленные параметры и флаги, многие из которых обычно игнорируются. Данная книга не претендует на роль энциклопедического справочника, и основное внимание в ней концентрируется лишь на наиболее важных функциях и параметрах.

• Windows предлагает многочисленные механизмы синхронизации и взаимодействия, обеспечивающие удовлетворение самых разнообразных запросов.

• Базовой единицей выполнения в Windows является поток (thread). В одном процессе (process) могут выполняться один или несколько потоков.

• Для функций Windows используются длинные описательные имена. Приведенные ниже в качестве примера имена функций иллюстрируют не только соглашения об использовании имен, но и многоликость функций Windows:

```
WaitForSingleObject
WaitForSingleObjectEx
WaitForMultipleObjects
WaitNamedPipe
```

Существует также несколько соглашений, регулирующих порядок использования имен типов:

• Имена предопределенных типов данных, необходимых API, также являются описательными, и в них должны использоваться прописные буквы.

К числу наиболее распространенных относятся следующие типы данных:

BOOL (определен как 32-битовый объект, предназначенный для хранения одного логического значения)

HANDLE
DWORD (вездесущее 32-битовое целое без знака)
LPTSTR (указатель на строку, состоящую из 8– или 16-битовых символов)
LPSECURITY_ATTRIBUTES

С другими многочисленными типами данных вы будете знакомиться по мере изложения материала.

• В именах предопределенных типов указателей операция * не используется, и они отражают дополнительные отличия между указателями различного типа, как, например, в случае типов LPTSTR (определен как TCHAR *) и LPCTSTR (определен как const TCHAR *). *Примечание.* Тип TCHAR может обозначать как обычный символьный тип char, так и двухбайтовый тип wchar_t.

• В отношении использования имен переменных, — по крайней мере, в прототипах функций, — также имеются определенные соглашения. Так, имя lpszFileName соответствует "длинному указателю на строку, завершающуюся нулевым символом", которая содержит имя файла. Этот пример иллюстрирует применение так называемой "венгерской нотации", которой мы в данной книге, как правило, не стремимся придерживаться. Точно так же, dwAccess — двойное слово (32 бита), содержащее флаги прав доступа к файлу, где "dw" означает "double word" — "двойное слово".

Примечание

Будет очень полезно, если вы просмотрите системные заголовочные (включаемые) файлы, в которых содержатся определения функций, констант, флагов, кодов ошибок и тому подобное. Многие из представляющих для нас интерес файлов, аналогичных тем, которые предложены ниже в качестве примера, являются частью среды Microsoft Visual C++ и обычно устанавливаются в каталоге Program Files\Microsoft Visual Studio.NET\Vc7\PlatformSDK\Include (или Program Files\Microsoft Visual Studio\VC98\Include в случае VC++ 6.0):

WINDOWS.H (файл, обеспечивающий включение всех остальных заголовочных файлов)
WINNT.H
WINBASE.H

Наконец, несмотря на то что оригинальный API Win32 с самого начала разрабатывался как совершенно независимый интерфейс, он проектировался с учетом обеспечения обратной совместимости с API Winl6, входившим в состав Windows 3.1. Это привело к некоторым досадным с точки зрения программиста последствиям:

• В названиях типов встречаются элементы анахронизма, как, например, в случае типов LPTSTR и LPDWORD, ссылающихся на "длинный указатель", который является простым 32– или 64-битовым указателем. Необходимость в указателях какого-либо иного типа отсутствует. Иногда составляющая "длинный" опускается, и тогда, например, типы LPVOID и PVOID являются эквивалентными.[8]

• В имена некоторых символических констант, например WIN32_FIND_DATA, входит компонент "WIN32", хотя те же константы используются и в Win64.

• Несмотря на то что упомянутая проблема обратной совместимости в настоящее время потеряла свою актуальность, она оставила после себя множество 16-разрядных функций, ни одна из которых в этой книге не используется, хотя и могло бы показаться, что эти функции играют весьма важную роль. В качестве примера можно привести функцию OpenFile, которая,

судя по ее названию, нужна для открытия файлов, тогда как в действительно сти для открытия существующих файлов всегда следует пользоваться только функцией CreateFile.

Подготовка к работе с Win64

Интерфейс Win64, который во время написания данной книги поддерживался Windows XP и Windows Server 2003 на процессорах семейства AMD64 (Opteron и Athlon 64) компании AMD и процессорах семейства Itanium (ранее известных под кодовыми названиями Merced, McKinley, Madison и IA-64) компании Intel, будет играть все более важную роль при создании крупных приложений. Существенные отличия между Win32 и Win64 обусловлены различиями в размере указателей (64 бита в Win64) и объеме доступного виртуального адресного пространства.

Проблемы переноса приложений на платформу Win64 обсуждаются по мере изложения материала на протяжении всей книги, а программы организованы таким образом, чтобы создание их в виде приложений Win64 обеспечивалось простым указанием соответствующих параметров на стадии компиляции. В находящихся на Web-хосте книги проектах с программами примеров в необходимых случаях предусмотрен вывод сообщений, предупреждающих о возникновении проблем при переходе к 64 разрядам, но большинство ситуаций (хотя и не полностью все), которые могли бы приводить к генерации таких сообщений, из программного кода исключены.

С точки зрения программиста основные отличия при переходе к Win64 обусловлены размерами указателей и необходимостью помнить о том, что длины указателя и целочисленной переменной (LONG, DWORD и так далее) не обязательно должны совпадать. С этой целью определены, например, типы DWORD32 и DWORD64, позволяющие явно управлять размером переменных. Два других типа, POINTER_32 и POINTER_64, позволяют управлять размером указателей.

Как вы сами убедитесь, приложив лишь самые незначительные усилия, можно добиться того, чтобы программы работали как в Win32, так и в Win64, и поэтому мы будем часто ссылаться на API просто как на Windows или, иногда, Win32. Дополнительная информация относительно Win64 содержится в главе 16, где, в частности, обсуждаются вопросы совместимости исходных и двоичных кодов.

> Программисты, работающие с UNIX и Linux, столкнутся в Windows с рядом интересных особенностей. Так, в Windows дескрипторы HANDLE являются "непрозрачными". Они не представляют собой ряд последовательно возрастающих целых чисел. В то же время, например, в UNIX дескрипторы файлов 0, 1 и 2 имеют специальное назначение, что должно обязательно учитываться при написании программ. Ничего подобного в Windows вы не обнаружите.
>
> Многие из различий, например грань между идентификаторами процессов и дескрипторами файлов, в Windows оказываются стертыми. В Windows объекты обеих типов описываются дескрипторами типа HANDLE. Во многих важных функциях могут наравне использоваться дескрипторы файлов, процессов, событий, каналов и других объектов.
>
> Программистам, которые, работая в UNIX, привыкли к коротким именам функций и параметров и использовали преимущественно строчные буквы, придется приспосабливаться к более пространному стилю Windows. Стиль Windows близок к стилю интерфейса компании Hewlett Packard (ранее — DEC и Compaq); программистам, работающим с OpenVMS, многое покажется знакомым. Указанное сходство между OpenVMS и Windows частично объясняется тем, что Дэвид Катлер (David Cutler), создатель первоначальной архитектуры VMS, предполагал, что она

должна играть ту же роль, что и NT или Windows.

Радикальные отличия касаются такого хорошо знакомого нам всем понятия, как процессы. В Windows процессы не обладают свойствами наследования, хотя и могут быть организованы в виде объектов заданий.

В завершение следует отметить, что в текстовых файлах Windows конец строки отмечается последовательностью управляющих символов CR-LF, а не LF, как в это принято в UNIX.

О целесообразности привлечения функций стандартной библиотеки C для обработки файлов

Несмотря на всю уникальность возможностей Windows, старый добрый язык C и его стандартная библиотека ANSI C по-прежнему могут с успехом использоваться при решении большинства задач, связанных с обработкой файлов. Кроме того, библиотека C (указание на ее соответствие стандарту ANSI C мы будем часто опускать) содержит большое число очень нужных функций, аналогов которых среди системных вызовов нет. К их числу относятся, например, функции, описанные в заголовочных файлах <string.h>, <stdlib.h> и <signal.h>, а также функции форматированного и символьного ввода/вывода. В то же время, имеются и такие функции, как fopen и fread, описанные в заголовочном файле <stdio.h>, для которых находятся близко соответствующие им системные вызовы.

В каких же случаях при обработке файлов можно обойтись библиотекой C, а в каких необходимо использовать системные вызовы Windows? Тот же вопрос можно задать и в отношении использования потоков (streams) ввода/вывода C++ или системы ввода/вывода, которая предоставляется платформой .NET. Простых ответов на эти вопросы не существует, но если во главу угла поставить переносимость программ на платформы, отличные от Windows, то в тех случаях, когда приложению требуется только обработка файлов, а не, например, управление процессами или другие специфические возможности Windows, предпочтение следует отдавать библиотеке C и потокам ввода/вывода C++. Вместе с тем, многими программистами ранее уже делались попытки выработать рекомендации относительно адекватности использования библиотеки C в тех или иных случаях, и эти же рекомендации должны быть применимы и в отношении Windows. Кроме того, с учетом возможностей расширения функциональности, а также повышения производительности и гибкости программ, обеспечиваемые Windows, нередко оказывается более удобным или даже необходимым не ограничиваться библиотекой C, в чем вы постепенно станете убеждаться уже начиная с главы 3. К числу возможностей Windows, не поддерживаемых библиотекой C, относятся блокирование и отображение файлов (необходимое для разделения общих областей памяти), асинхронный ввод/вывод, произвольный доступ к файлам чрезвычайно крупных размеров (4 Гбайт и выше) и взаимодействие между процессами.

В случае простых программ вам будет вполне достаточно использовать функции библиотеки C, предназначенные для работы с файлами. Воспользовавшись библиотекой C, можно написать переносимое приложение даже без изучения Windows, однако возможности выбора при этом будут ограниченными. Так, в главе 5 для повышения производительности программы и упрощения программирования применено отображение файлов, однако библиотека C такие возможности не предоставляет.

Что требуется для работы с данной книгой

Ниже перечислено все то, что необходимо вам для создания и выполнения примеров, приведенных в этой и последующих главах книги.

Разумеется, прежде всего, вам потребуется весь ваш опыт в области разработки приложений; предполагается также, что язык C вам знаком. Однако прежде, чем браться за решение упражнений и разбор примеров, вы должны убедиться в том, что располагаете всем необходимым аппаратным и программным обеспечением, перечень которого приводится ниже.

• Система с установленной ОС Windows.

• Компилятор C и любая подходящая среда разработки приложений, например, Microsoft Visual Studio .NET или Microsoft Visual C++ версии 6.0. Имеются также системы разработки приложений от других поставщиков, и хотя примеры из книги нами на них не тестировались, из поступивших от нескольких читателей писем нам стало известно, что примеры, пусть даже после внесения в них незначительных изменений, в некоторых случаях успешно выполнялись даже при использовании других систем. Кроме того, в приложении A содержится информация, касающаяся использования инструментальных средств с открытым исходным кодом. *Примечание.* Наше внимание будет сосредоточено на разработке консольных приложений Windows, и поэтому возможности Microsoft Visual Studio .NET будут задействованы далеко не в полной мере.

• Достаточный для разработки программ объем ОЗУ и наличие свободного места на жестком диске. Практически любая коммерчески доступная система предоставит вам достаточный объем памяти, место на диске и процессорную мощность, которых хватит для запуска примеров и среды разработки приложений, однако предварительно необходимо проверить, какие именно требования к ресурсам предъявляет эта среда.[9]

• Привод компакт-диска, системного или сетевого, для установки среды разработки приложений.

• Оперативная документация наподобие той, которая поставляется вместе с Microsoft Visual C++. Желательно, чтобы вы установили эту документацию на своем жестком диске, поскольку к ней будет требоваться частый доступ. Дополнительную информацию вы всегда сможете получить на Web-сайте компании Microsoft.

Пример: простое последовательное копирование файла

В следующих разделах приведены примеры коротких программ, реализующих простое последовательное копирование содержимого файла тремя различными способами:

1. С использованием библиотеки С.
2. С использованием Windows.
3. С использованием вспомогательной функции Windows — CopyFile.

Кроме того, что эти примеры дают возможность сопоставить между собой различные модели программирования, они также демонстрируют возможности и ограничения, присущие библиотеке С и Windows. Альтернативные варианты реализации усилят программу, увеличивая ее производительность и повышая гибкость.

Последовательная обработка файлов является простейшей, наиболее распространенной и самой важной из возможностей, обеспечиваемых любой операционной системой, и почти в каждой большой программе хотя бы несколько файлов обязательно подвергаются этому виду обработки. Поэтому простая программа обработки файлов предоставляет прекрасную возможность ознакомиться с Windows и принятыми в ней соглашениями.

Копирование файлов, нередко осуществляемое совместно с обновлением их содержимого, и слияние отсортированных файлов являются распространенными формами последовательной обработки файлов. Примерами других приложений, осуществляющих последовательный доступ к файлам, могут служить компиляторы и инструментальные средства, предназначенные для обработки текста.

Несмотря на концептуальную простоту последовательной обработки файлов, эффективная реализация этого процесса, обеспечивающая оптимальную скорость его выполнения, может оказаться нелегкой задачей. Для этого может потребоваться использование перекрывающегося ввода/вывода, отображения файлов, потоков и других дополнительных методов.

Само по себе копирование файлов не представляет особого интереса, однако сравнение программ не только позволит вам быстро оценить, чем отличаются друг от друга различные системы, но и послужит хорошим предлогом для знакомства с Windows. В последующих примерах реализуется ограниченный вариант одной из команд UNIX — cp, осуществляющей копирование одного файла в другой и требующей задания имен файлов в командной строке. В приведенных программах организована лишь простейшая проверка ошибок, которые могут возникать на стадии выполнения, а существующие файлы просто перезаписываются. Эти и другие недостатки будут учтены в последующих Windows-реализациях этой и других программ.
Примечание. Реализация программы для UNIX находится на Web-сайте книги.

Копирование файлов с использованием стандартной библиотеки С

Как видно из текста программы 1.1, стандартная библиотека С поддерживает объекты потоков ввода/вывода FILE, которые напоминают, несмотря на меньшую общность, объекты Windows HANDLE, представленные в программе 1.2.

Программа 1.1. cpC: копирование файлов с использованием библиотеки С

```
    /* Глава 1. Базовая программа копирования файлов cp. Реализация, использующая
библиотеку С. */
    /* cp файл1 файл2: Копировать файл1 в файл2. */
```

```
#include <stdio.h>
#include <errno.h>
#define BUF_SIZE 256

int main(int argc, char *argv[]) {
 FILE *in_file, *out_file;
 char rec [BUF_SIZE];
 size_t bytes_in, bytes_out;
 if (argc != 3) {
  printf("Использование: cpC файл1 файл2\n");
  return 1;
 }
 in_file = fopen(argv [1], "rb");
 if (in_file == NULL) {
  perror(argv[1]);
  return 2;
 }
 out_file = fopen(argv [2], "wb");
 if (out_file == NULL) {
  perror(argv [2]);
  return 3;
 }
 /* Обработать входной файл по одной записи за один раз. */
 while ((bytes_in = fread(rec, 1, BUF_SIZE, in_file)) > 0) {
  bytes_out = fwrite(rec, 1, bytes_in, out_file);
  if (bytes_out != bytes_in) {
   perror("Неустранимая ошибка записи.");
   return 4;
  }
 }
 fclose (in_file);
 fclose (out_file);
 return 0;
}
```

Этот простой пример может служить наглядной иллюстрацией ряда общепринятых допущений и соглашений программирования, которые не всегда применяются в Windows.

1. Объекты открытых файлов идентифицируются указателями на структуры FILE (в UNIX используются целочисленные дескрипторы файлов). Указателю NULL соответствует несуществующий объект. По сути, указатели являются разновидностью дескрипторов объектов открытых файлов.

2. В вызове функции fopen указывается, каким образом должен обрабатываться файл — как текстовый или как двоичный. В текстовых файлах содержатся специфические для каждой системы последовательности символов, используемых, например, для обозначения конца строки. Во многих системах, включая Windows, в процессе выполнения операций ввода/вывода каждая из таких последовательностей автоматически преобразуется в нулевой символ, который интерпретируется в языке C как метка конца строки, и наоборот. В нашем примере оба файла открываются как двоичные.

3. Диагностика ошибок реализуется с помощью функции perror, которая, в свою очередь, получает информацию относительно природы сбоя, возникающего при вызове функции fopen, из глобальной переменной errno. Вместо этого можно было бы воспользоваться функцией ferror, возвращающей код ошибки, ассоциированный не с системой, а с объектом FILE.

4. Функции fread и fwrite возвращают количество обработанных байтов непосредственно, а не через аргумент, что оказывает существенное влияние на логику организации программы. Неотрицательное возвращаемое значение говорит об успешном выполнении операции чтения,

тогда как нулевое — о попытке чтения метки конца файла.

5. Функция fclose может применяться лишь к объектам типа FILE (аналогичное утверждение справедливо и в отношении дескрипторов файлов UNIX).

6. Операции ввода/вывода осуществляются в синхронном режиме, то есть прежде чем программа сможет выполняться дальше, она должна дождаться завершения операции ввода/вывода.

7. Для вывода сообщений об ошибках удобно использовать входящую в библиотеку C функцию ввода/вывода printf, которая даже будет использована в первом примере Windows-программы.

Преимуществом реализации, использующей библиотеку C, является ее переносимость на платформы UNIX, Windows, а также другие системы, которые поддерживают стандарт ANSI C. Кроме того, как показано в приложении B, в том, что касается производительности, вариант, использующий функции ввода/вывода библиотеки C, ничуть не уступает другим вариантам реализации. Тем не менее, в этом случае программы вынуждены ограничиваться синхронными операциями ввода/вывода, хотя влияние этого ограничения будет несколько ослаблено использованием потоков Windows (начиная с главы 7).

Как и их эквиваленты в UNIX, программы, основанные на функциях для работы с файлами, входящих в библиотеку C, способны выполнять операции произвольного доступа к файлам (с использованием функции fseek или, в случае текстовых файлов, функций fsetpos и fgetpos), но это является уже потолком сложности для функций ввода/вывода стандартной библиотеки C, выше которого они подняться не могут. Вместе с тем, Visual C++ предоставляет нестандартные расширения, способные, например, поддерживать блокирование файлов. Наконец, библиотека C не позволяет управлять средствами защиты файлов.

Резюмируя, можно сделать вывод, что если простой синхронный файловый или консольный ввод/вывод — это все, что вам надо, то для написания переносимых программ, которые будут выполняться под управлением Windows, следует использовать библиотеку C.

Копирование файлов с использованием Windows

В программе 1.2 решается та же задача копирования файлов, но делается это с помощью Windows API, а базовые приемы, стиль и соглашения, иллюстрируемые этой программой, будут использоваться на протяжении всей этой книги.

Программа 1.2. cpW: копирование файлов с использованием Windows, первая реализация

```
    /* Глава 1. Базовая программа копирования файлов ср. Реализация, использующая
Windows. */
    /* cpW файл1 файл2: Копировать файл1 в файл2. */
    #include <windows.h>
    #include <stdio.h>
    #define BUF_SIZE 256

    int main (int argc, LPTSTR argv []) {
     HANDLE hIn, hOut;
     DWORD nIn, nOut;
     CHAR Buffer [BUF_SIZE];
     if (argc != 3) {
      printf ("Использование: cpW файл1 файл2\n");
```

```
      return 1;
     }
     hIn = CreateFile(argv [1], GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
     if (hIn == INVALID_HANDLE_VALUE) {
      printf("Невозможно открыть входной файл. Ошибка: %x\n", GetLastError());
      return 2;
     }
       hOut   =   CreateFile(argv[2],   GENERIC_WRITE,   0,   NULL,   CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
     if (hOut == INVALID_HANDLE_VALUE) {
      printf("Невозможно открыть выходной файл. Ошибка: %x\n", GetLastError());
      return 3;
     }
     while (ReadFile(hIn, Buffer, BUF_SIZE, &nIn, NULL) && nIn > 0) {
      WriteFile(hOut, Buffer, nIn, &nOut, NULL);
      if (nIn != nOut) {
       printf ("Неустранимая ошибка записи: %x\n", GetLastError());
       return 4;
      }
     }
     CloseHandle(hIn);
     CloseHandle(hOut);
     return 0;
    }
```

Этот простой пример иллюстрирует некоторые особенности программирования в среде Windows, к подробному рассмотрению которых мы приступим в главе 2.

1. В программу всегда включается файл <windows.h>, в котором содержатся все необходимые определения функций и типов данных Windows.[10]

2. Все объекты Windows идентифицируются переменными типа Handle, причем для большинства объектов можно использовать одну и ту же общую функцию CloseHandle.

3. Рекомендуется закрывать все ранее открытые дескрипторы, если в необходимость в них отпала, чтобы освободить ресурсы. В то же время, при завершении процессов относящиеся к ним дескрипторы автоматически закрываются ОС, и если не остается ни одного дескриптора, ссылающегося на какой-либо объект, то ОС уничтожает этот объект и освобождает соответствующие ресурсы. *(Примечание.* Как правило, файлы подобным способом не уничтожаются.)

4. Windows определяет многочисленные символические константы и флаги. Обычно они имеют длинные имена, нередко поясняющие назначение данного объекта. В качестве типичного примера можно привести имена INVALID_HANDLE_VALUE и GENERIC_READ.

5. Функции ReadFile и WriteFile возвращают булевские значения, а не количества обработанных байтов, для передачи которых используются аргументы функций. Это определенным образом изменяет логику организации работы циклов.[11] Нулевое значение счетчика байтов указывает на попытку чтения метки конца файла и не считается ошибкой.

6. Функция GetLastError позволяет получать в любой точке программы коды системных ошибок, представляемые значениями типа DWORD. В программе 1.2 показано, как организовать вывод генерируемых Windows текстовых сообщений об ошибках.

7. Windows NT обладает более мощной системой защиты файлов, описанной в главе 15. В данном примере защита выходного файла не обеспечивается.

8. Такие функции, как CreateFile, обладают богатым набором дополнительных параметров, но в данном примере использованы значения по умолчанию.

Копирование файлов с использованием вспомогательной функции Windows

Для повышения удобства работы в Windows предусмотрено множество вспомогательных функций (convenience functions), которые, объединяя в себе несколько других функций, обеспечивают выполнение часто встречающихся задач программирования. В некоторых случаях использование этих функций может приводить к повышению производительности (см. приложение В). Например, благодаря применению функции CopyFile значительно упрощается программа копирования файлов (программа 1.3). Помимо всего прочего, это избавляет нас от необходимости заботиться о буфере, размер которого в двух предыдущих программах произвольно устанавливался равным 256.

Программа1.3.cpCF: копирование файлов с использованием вспомогательной функции Windows

```
    /* Глава 1. Базовая программа копирования файлов ср. Реализация, в которой для
повышения удобства использования и производительности программы используется
функция Windows CopyFile. */
    /* cpCF файл1 файл2: Копировать файл1 в файл2. */
    #include <windows.h>
    #include <stdio.h>

    int main (int argc, LPTSTR argv []) {
     if (argc != 3) {
      printf ("Использование: cpCF файл1 файл2\n");
      return 1;
     }
     if (!CopyFile(argv[1], argv[2], FALSE)) {
      printf("Ошибка при выполнении функции CopyFile: %x\n", GetLastError());
      return 2;
     }
     return 0;
    }
```

Резюме

Ознакомительные примеры, в качестве которых были использованы три простые программы копирования файлов, демонстрируют многие из отличий, существующих между программами, в которых применяется с одной стороны библиотека C, а с другой — Windows. Отличия в производительности различных вариантов реализации анализируются в приложении B. Примеры, в которых используется Windows, наглядно демонстрируют стиль программирования и некоторые соглашения, принятые в Windows, но дают лишь отдаленное представление о тех функциональных возможностях, которые Windows предлагает программистам.

Целевыми платформами для данной книги и содержащихся в ней примеров являются системы NT5 (Windows XP, 2000 и Server 2003). Тем не менее, большая часть материала книги применима также к ранним версиям NT и системам Windows 9x (95, 98 и Me).

В следующих главах

Главы 2 и 3 посвящены гораздо более пристальному рассмотрению функций ввода/вывода и файловой системы. Они включают в себя такие темы, как консольный ввод/вывод, обработка символов ASCII и Unicode, работа с файлами и каталогами, а также программирование реестра. В указанных главах разрабатываются базовые методики и закладывается фундамент для остальной части книги.

Дополнительная литература

Полная информация о рекомендуемых ниже книгах приведена в библиографическом списке в конце книги.

Win32

Двумя доступными в настоящее время книгами, в которых вопросы программирования для Windows рассматриваются с всех возможных точек зрения, являются [5] и [31]. В то же время, существует множество других книг, которые не обновлялись и не отражают прогресс, достигнутый с момента выхода Windows 95 или Windows NT.

По каждой функции Microsoft Visual C++ имеется оперативная гипертекстовая справочная документация, но ту же информацию можно получить, посетив домашнюю страницу компании Microsoft — http://www.microsoft.com, где вы найдете целый ряд ссылок на технические статьи, посвященные различным аспектам Windows. Начните с раздела MSDN (Microsoft Developer's Network) и произведите поиск по любой интересующей вас теме. Вы обнаружите огромное разнообразие официальной документации, описаний продуктов, примеров программного кода, а также другую полезную информацию.

Win64

Win64 обсуждается в нескольких книгах, но обширный материал по этой теме можно найти

на домашней странице компании Microsoft.

Архитектура Windows NT и история ее развития

Читателям, которые хотят больше узнать о целях проектирования Windows NT или понять основные принципы, лежащие в основе ее архитектуры, будет полезна книга [38]. В этой книге рассматриваются объекты, процессы, потоки, виртуальная память, ядро и подсистемы ввода/вывода. Вместе с тем, собственно функции API, а также Windows 9x и CE в ней не обсуждаются. Рекомендуем время от времени заглядывать в упомянутую книгу для получения дополнительной информации. Кроме того, обратитесь к ранее вышедшим книгам [9] и [37], в которых содержится важный ретроспективный анализ эволюции NT.

UNIX

В книге [40], написанной ныне покойным Уильямом Ричардом Стивенсом (W. Richard Stevens), UNIX обсуждается во многом в тех же терминах, которые в настоящей книге используются для обсуждения Windows. Книга Стивенса по-прежнему остается стандартным справочником по средствам UNIX, но в ней не рассматриваются потоки. Стандартизация UNIX претерпела изменения, однако в книге Стивенса содержатся удобные рабочие определения всего того, что предлагается в UNIX, а также в Linux. В этой книге сопоставлены возможности функций файлового ввода/вывода библиотеки C и функций ввода/вывода системы UNIX, что имеет отношение и к Windows.

Если вас интересуют сравнительные характеристики ОС и более глубокое обсуждение UNIX, обратитесь к книге [29] и ее русскоязычному изданию [49], которая помимо того, что является весьма полезной, еще и увлекательно написана, хотя некоторым читателям позиция автора может показаться несколько предвзятой.

Программирование с использованием Windows GUI

Пользовательский интерфейс в настоящей книге не рассматривается. В случае необходимости можете обратиться к [30] или [25].

Теория операционных систем

Существует масса хороших учебников по общей теории ОС. Одной из наиболее популярных является книга [35].

Стандартная библиотека ANSI C

Исчерпывающим руководством по этой теме служит книга [27]. Для получения беглого обзора можно обратиться к книге [20] или к ее русскоязычному изданию [48], которая содержит полное описание библиотеки и остается классическим учебником по языку программирования

C. Эти книги помогут вам принять решение относительно того, достаточно ли возможностей библиотеки C для решения стоящих перед вами задач обработки файлов.

Windows CE

Тем, кто хочет применить материал настоящей книги к Windows CE, можно порекомендовать книгу [23].

Эмуляция Windows в UNIX

Для получения необходимой информации по этому вопросу и загрузки пакета с открытым исходным кодом Wine, позволяющего эмулировать Windows API поверх UNIX и X, посетите сайт http://www.winehq.com.

Упражнения

1.1. Скомпилируйте, скомпонуйте и выполните каждую из трех программ, предназначенных для копирования файлов. К числу других возможных вариантов реализации относится использование библиотек совместимости с UNIX, включая библиотеку Microsoft Visual C++ (программа, использующая эту библиотеку, доступна на Web-сайте книги). *Примечание.* На Web-сайте книги на ходятся исходные коды всех программ. Краткие рекомендации относительно порядка использования этих кодов в средах Microsoft Visual Studio .NET и Microsoft Visual C++ 6.0 вы найдете в приложении А.

1.2. Ознакомьтесь с одной из сред разработки приложений, например, Microsoft Visual Studio .NET или Microsoft Visual C++. В частности, научитесь создавать в выбранной среде консольные приложения. Для проведения самостоятельных экспериментов с использованием рассмотренных в данной главе программ пользуйтесь отладчиком. Инструкции относительно того, как следует приступать к работе, содержатся в приложении А, а обширную дополнительную информацию вы найдете на Web-сайте компании Microsoft и в документации к используемой вами среде разработки приложений.

1.3. В Windows в качестве метки конца строки используется последовательность символов "возврат каретки-перевод строки" (CR-LF). Определите, как изменится поведение программы 1.1, если входной файл открывать в двоичном режиме, а выходной — в текстовом, или наоборот. Как это будет проявляться в системе UNIX и в других системах?

1.4. Выполните для каждой из программ хронометраж при копировании файлов большого размера. Получите соответствующие данные для как можно большего числа различных вариантов и сравните полученные результаты между собой. Вряд ли следует подчеркивать, что быстродействие программ зависит от множества факторов, однако, в предположении, что все остальные параметры системы остаются неизменными, сопоставление результатов, полученных с использованием различных вариантов реализации программы, может представлять определенную ценность. *Совет.* Для облегчения анализа результатов расположите их в виде таблицы. Программа, обеспечивающая количественный контроль длительности временных промежутков, приведе на в главе 6, а некоторые экспериментальные результаты представлены в приложении В.

ГЛАВА 2
Использование файловой системы и функций символьного ввода/вывода Windows

Нередко самыми первыми средствами операционной системы (ОС), с которыми разработчик сталкивается в любой системе, являются файловая система и простой терминальный ввод/вывод. Ранние ОС для PC, такие как MS-DOS, не могли дать ничего больше, кроме возможностей работы с файлами и терминального (или *консольного)* ввода/вывода, но эти же ресурсы и сейчас занимают центральное место почти в любой ОС.

Файлы играют очень важную роль в организации долговременного хранения данных и программ и обеспечивают простейшую форму межпрограммного взаимодействия. Помимо этого, многие аспекты файловых систем оказываются применимыми также к взаимодействию между процессами и сетям.

На примере программ копирования файлов, которые рассматривались в главе 1, вы уже познакомились с четырьмя важными функциями, обеспечивающими последовательную обработку файлов:

```
CreateFile
ReadFile
WriteFile
CloseHandle
```

В данной главе не только подробно описываются эти и родственные им функции, но и обсуждаются функции, предназначенные для обработки символов и обеспечения консольного ввода/вывода. Сначала будут кратко охарактеризованы существующие типы файловых систем и их основные свойства. Далее будет показано, каким образом введение расширенной формы символов в кодировке Unicode помогает приложениям справиться с проблемой поддержки национальных языков. Главу завершает введение в управление файлами и каталогами в Windows.

Файловые системы Windows

Windows поддерживает на непосредственно подключенных устройствах файловые системы четырех типов, но только первый из них будет иметь для нас существенное значение на протяжении всей книги, поскольку именно полнофункциональная файловая система этого типа рекомендуется компанией Microsoft для использования в качестве основной:

1. Файловая система *NT* (NTFS) — современная файловая система, которая поддерживает длинные имена файлов, а также безопасность, устойчивость к сбоям, шифрование, сжатие, расширенные атрибуты, и позволяет работать с очень большими файлами и объемами данных. Заметьте, что на гибких дисках (флоппи-дисках, или дискетах) система NTFS использоваться не может; не поддерживается она и системами Windows 9x.

2. Файловые системы *FAT* и *FAT32* (от *File Allocation Table* — таблица размещения файлов) происходят от 16-разрядной файловой системы (FAT16), первоначально использовавшейся в MS-DOS и Windows 3.1. FAT32 впервые была введена в Windows 98 для поддержки жестких дисков большого объема и других усовершенствованных возможностей; далее под термином FAT мы будем подразумевать любую из вышеуказанных версий. FAT является единственно доступной файловой системой для дисков (но не компакт-дисков), работающих под управлением Windows 9x, а также гибких дисков. Разновидностью FAT является TFAT — ориентированная на поддержку механизма транзакций версия, используемая в Windows CE. Постепенно FAT выходит из употребления и в большинстве случаев ее можно встретить лишь на устаревших системах, особенно тех, обновление которых после первоначальной установки на них Windows 9x выполнялось без преобразования типа существующей файловой системы.

3. Файловая система *компакт-дисков* (CDFS), как говорит само ее название, предназначена для доступа к информации, записанной на компакт-дисках. CDFS удовлетворяет требованиям стандарта ISO 9660.

4. *Универсальный дисковый формат* (Universal Disk Format, UDF) поддерживает диски DVD и, в конечном итоге, должен полностью вытеснить систему CDFS. Поддержка UDF в Windows XP поддерживает как чтение, так и запись файлов, тогда как в Windows 2000 для UDF обеспечивается только запись.

Windows поддерживает, причем как на стороне клиента, так и на стороне сервера, такие распределенные файловые системы, как Networked File System (Сетевая файловая система), или NFS, и Common Internet File System (Общая межсетевая файловая система), или CIFS; на серверах обычно используют NTFS. В Windows 2000 и Windows Server 2003 обеспечивается широкая поддержка сетевых хранилищ данных (Storage Area Networks, SAN) и таких развивающихся технологий хранения данных, как IP-хранилища. Кроме того, Windows дает возможность разрабатывать пользовательские файловые системы, которые поддерживают тот же API доступа к файлам, что и API, рассматриваемый в этой и следующей главах.

Доступ к файловым системам любого типа осуществляется одинаковым образом, иногда с некоторыми ограничениями. Например, поддержка защиты файлов обеспечивается только в NTFS. В необходимых случаях мы будем обращать ваше внимание на особенности, присущие только NTFS, но в этой книге, как правило, будет предполагаться использование именно этой системы.

Формат файловой системы (FAT, NTFS или пользовательской), будь то для диска в целом, или для его разделов, определяется во время разбивки диска на разделы.

Правила именования файлов

Windows поддерживает обычную иерархическую систему имен файлов, соглашения которой, однако, несколько отличаются от соглашений, привычных для пользователей UNIX, и основаны на следующих правилах:

• Полное имя файла на диске, содержащее путь доступа к нему, начинается с указания буквенного имени диска, например, A: или C:. Обычно буквы A: и B: относятся к флоппи-дисководам, а C:, D: и так далее — к жестким дискам и приводам компакт-дисков. Последующие буквы алфавита, например, H: или K:, обычно соответствуют сетевым дискам. *Примечание.* Буквенные обозначения дисков не поддерживаются в Windows CE.

• Существует и другой возможный вариант задания полного пути доступа — использование универсальной кодировки имен (Universal Naming Code, UNC), в соответствии с которой указание пути начинается с глобального корневого каталога, обозначаемого двумя символами обратной косой черты (\\), с последующим указанием имени сервера и *имени разделяемого ресурса* (share name) для определения местоположения ресурса на файловом сервере сети. Таким образом, первая часть полного пути доступа в данном случае будет иметь вид: \\servername\sharename.

• При указании полного пути доступа в качестве *разделителя* обычно используется символ обратной косой черты (\), но в параметрах API для этой цели можно воспользоваться также символом прямой косой черты (/), как это принято в C.

• В именах каталогов и файлов не должны встречаться символы ASCII, численные значения которых попадают в интервал 1-31, а также любой из перечисленных ниже символов:

```
< > : " | ? * \ /
```

В именах разрешается использовать пробелы. В то же время, если имена файлов, содержащие пробелы, указываются в командной строке, то каждое такое имя следует заключать в кавычки, чтобы его нельзя было интерпретировать как два разных имени, относящихся к двум отдельным файлам.

• Строчные и прописные буквы в именах каталогов и файлов не различаются, то есть имена *не чувствительны к регистру* (case-insensitive), но в то же время они *запоминают регистр* (case-retaining); другими словами, если файл был создан с именем MyFile, то это же имя будет использоваться и при его отображении, хотя, например, для доступа к файлу может быть использовано также имя myFILE.

• Длина имени каталога и файла не должна превышать 255 символов, а длина полного пути доступа ограничивается значением параметра MAX_PATH (текущим значением которого является 256).

• Для отделения имени файла от расширения используется символ точки (.), причем расширения имен (как правило, два или три символа, находящиеся справа от самой последней точки, входящей в имя файла) обозначают предположительные типы файлов в соответствии с определенными соглашениями. Так, можно ожидать, что файл atou.EXE — это исполняемый файл, а файл atou.C — файл с исходным текстом программы на языке C. Допускается использование в именах файлов нескольких символов точки.

• Одиночный символ точки (.) и два символа точки (..), используемые в качестве имен каталогов, обозначают, соответственно, текущий каталог и его родительский каталог.

После этого вступления мы можем продолжить изучение функций Windows, начатое в главе 1.

Операции открытия, чтения, записи и закрытияфайлов

Первой функцией Windows, которую мы подробно опишем, является функция CreateFile, используемая как для создания новых, так и для открытия существующих файлов. Для этой функции, как и для всех остальных, сначала приводится прототип, а затем обсуждаются соответствующие параметры и порядок работы с ней.

Создание и открытие файла

Поскольку данная функция является первой из функций Windows, к изучению которых мы приступаем, ее описание будет несколько более подробным по сравнению с остальными; для других функций часто будут приводиться лишь краткие описания. Вместе с тем, даже в случае функции CreateFile будут описаны далеко не все из возможных многочисленных значений ее параметров, однако необходимые дополнительные сведения вы всегда сможете найти в оперативной справочной системе.

Простейшее использование функции CreateFile иллюстрирует приведенный в главе 1 пример ознакомительной Windows-программы (программа 1.2), содержащей два вызова функций, в которых для параметров dwShareMode, lpSecurityAttributes и hTemplateFile были использованы значения по умолчанию. Параметр dwAccess может принимать значения GENERIC_READ и GENERIC_WRITE.

```
HANDLE CreateFile(LPCTSTR lpName, DWORD dwAccess, DWORD dwShareMode,
LPSECURITY_ATTRIBUTES lpSecurityAttributes, DWORD dwCreate, DWORD
dwAttrsAndFlags, HANDLE hTemplateFile)
```

Возвращаемое значение: в случае успешного выполнения — дескриптор открытого файла (типа HANDLE), иначе — INVALID_HANDLE_VALUE.

Параметры

Имена параметров иллюстрируют некоторые соглашения Windows. Префикс dw используется в именах параметров типа DWORD (32-битовые целые без знака), в которых могут храниться флаги или числовые значения, например счетчики, тогда как префикс lpsz (длинный указатель на строку, завершающуюся нулем), или в упрощенной форме — lp, используется для строк, содержащих пути доступа, либо иных строковых значений, хотя документация Microsoft в этом отношении не всегда последовательна. В некоторых случаях для правильного определения типа данных вам придется обратиться к здравому смыслу или внимательно прочесть документацию.

lpName — указатель на строку с завершающим нулевым символом, содержащую имя файла, канала или любого другого именованного объекта, который необходимо открыть или создать. Допустимое количество символов при указании путей доступа обычно ограничивается значением MAX_PATH (260), однако в Windows NT это ограничение можно обойти, поместив перед именем префикс \\?\, что обеспечивает возможность использования очень длинных имен (с числом символов вплоть до 32 К). Сам префикс в имя не входит. О типе данных LPCTSTR говорится в одном из последующих разделов, а пока вам будет достаточно знать, что он относится к строковым данным.

dwAccess — определяет тип доступа к файлу — чтение или запись, что соответственно

указывается флагами GENERIC_READ и GENERIC_WRITE. Ввиду отсутствия флаговых значений READ и WRITE использование префикса GENERIC_ может показаться излишним, однако он необходим для совместимости с именами макросов, определенных в заголовочном файле Windows WINNT.H. Вы еще неоднократно столкнетесь с именами, которые кажутся длиннее, чем необходимо.

Указанные значения можно объединять операцией поразрядного "или" (|), и тогда для получения доступа к файлу как по чтению, так и по записи, следует воспользоваться таким выражением:

```
GENERIC_READ | GENERIC_WRITE
```

dwShareMode — может объединять с помощью операции поразрядного "или" следующие значения:

• 0 — запрещает разделение (совместное использование) файла. Более того, открытие второго дескриптора для данного файла запрещено даже в рамках одного и того же вызывающего процесса.

• FILE_SHARE_READ — другим процессам, включая и тот, который осуществил данный вызов функции, разрешается открывать этот файл для параллельного доступа по чтению.

• FILE_SHARE_WRITE — разрешает параллельную запись в файл.

Используя блокирование файла или иные механизмы, программист должен самостоятельно позаботиться об обработке ситуаций, в которых осуществляются одновременно несколько попыток записи в одно и то же место в файле. Более подробно этот вопрос рассматривается в главе 3.

lpSecurityAttributes — указывает на структуру SECURITY_ATTRIBUTES. На первых порах при вызовах функции CreateFile и всех остальных функций вам будет достаточно использовать значение NULL; вопросы безопасности файловой системы рассматриваются в главе 15.

dwCreate — конкретизирует запрашиваемую операцию: создать новый файл, перезаписать существующий файл и тому подобное. Может принимать одно из приведенных ниже значений, которые могут объединяться при помощи операции поразрядного "или" языка C.

• CREATE_NEW — создать новый файл; если указанный файл уже существует, выполнение функции завершается неудачей.

• CREATE_ALWAYS — создать новый файл; если указанный файл уже существует, функция перезапишет его.

• OPEN_EXISTING — открыть файл; если указанный файл не существует, выполнение функции завершается неудачей.

• OPEN_ALWAYS — открыть файл; если указанный файл не существует, функция создаст его.

• TRUNCATE_EXISTING — открыть файл; размер файла будет установлен равным нулю. Уровень доступа к файлу, установленный параметром dwAccess, должен быть не ниже GENERIC_WRITE. Если указанный файл существует, его содержимое будет уничтожено. В отличие от случая CREATENEW выполнение функции будет успешным даже в тех случаях, когда указанный файл не существует.

dwAttrsAndFlags — позволяет указать атрибуты файла и флаги. Всего имеется 16 флагов и атрибутов. Атрибуты являются характеристиками файла, а не открытого дескриптора, и игнорируются, если открывается существующий файл. Некоторые из наиболее важных флаговых значений приводятся ниже.

• FILE_ATTRIBUTE_NORMAL — этот атрибут можно использовать лишь при условии, что одновременно с ним не устанавливаются никакие другие атрибуты (тогда как для всех остальных флагов одновременная установка допускается).

• FILE_ATTRIBUTE_READONLY — этот атрибут запрещает приложениям осуществлять запись в данный файл или удалять его.

• FILE_FLAG_DELETE_ON_CLOSE — этот флаг полезно применять в случае временных файлов. Файл будет удален сразу же после закрытия последнего из его открытых дескрипторов.

• FILE_FLAG_OVERLAPPED — этот флаг играет важную роль при выполнении операций асинхронного ввода/вывода, описанных в главе 14.

Кроме того, существует несколько дополнительных флагов, позволяющих уточнить способ обработки файла и облегчить реализации Windows оптимизацию производительности и обеспечение целостности файлов.

• FILE_FLAG_WRITE_THROUGH — устанавливает режим сквозной записи промежуточных данных непосредственно в файл на диске, минуя кэш.

• FILE_FLAG_NO_BUFFERING — устанавливает режим отсутствия промежуточі ной буферизации или кэширования, при котором обмен данными происходит непосредственно с буферами данных программы, указанными при вызове функций ReadFile или WriteFile (описаны далее). Соответственно требуется, чтобы начала программных буферов совпадали с границами секторов, а их размеры были кратными размеру сектора тома. Чтобы определить размер сектора при указании этого флага, вы можете воспользоваться функцией GetDiskFreeSpace.

• FILE_FLAG_RANDOM_ACCESS — предполагается открытие файла для произвольного доступа; Windows будет пытаться оптимизировать кэширование файла применительно к этому виду доступа.

• FILE_FLAG_SEQUENTIAL_SCAN — предполагается открытие файла для последовательного доступа; Windows будет пытаться оптимизировать кэширование файла применительно к этому виду доступа. Оба последних режима реализуются системой лишь по мере возможностей.

hTemplateFile — дескриптор с правами доступа GENERIC_READ к шаблону файла, предоставляющему расширенные атрибуты, которые будут применены к создаваемому файлу вместо атрибутов, указанных в параметре dwAttrsAndFlags. Обычно значение этого параметра устанавливается равным NULL. При открытии существующего файла параметр hTemplateFile игнорируется. Этот параметр используется в тех случаях, когда требуется, чтобы атрибуты вновь создаваемого файла совпадали с атрибутами уже существующего файла.

Оба вызова функции CreateFile в программе 1.2 максимально упрощены за счет использования для параметров значений по умолчанию, и, тем не менее, они вполне справляются со своими задачами. В обоих случаях было бы целесообразно использовать флаг FILE_FLAG_SEQUENTIAL_SCAN. (Эта возможность исследуется в упражнении 2.3, а соответствующие результаты тестирования производительности приведены в приложении В.)

Заметьте, что для данного файла могут быть одновременно открыты несколько дескрипторов, если только это разрешается атрибутами совместного доступа и защиты файла. Открытые дескрипторы могут принадлежать одному и тому же или различным процессам. (Управление процессами описано в главе 6).

В Windows Server 2003 предоставляется функция ReOpenFile, которая возвращает новый дескриптор с иными флагами, правами доступа и прочим, нежели те, которые были указаны при первоначальном открытии файла, если только это не приводит к возникновению конфликта между новыми и прежними правами доступа.

Закрытие файла

Для закрытия объектов любого типа, объявления недействительными их дескрипторов и

освобождения системных ресурсов почти во всех случаях используется одна и та же универсальная функция. Исключения из этого правила будут оговариваться отдельно. Закрытие дескриптора сопровождается уменьшением на единицу счетчика ссылок на объект, что делает возможным удаление таких не хранимых постоянно (nonpersistent) объектов, как временные файлы или события. При выходе из программы система автоматически закрывает все открытые дескрипторы, однако лучше все же, чтобы программа самостоятельно закрывала свои дескрипторы перед тем, как завершить работу.

Попытки закрытия недействительных дескрипторов или повторного закрытия одного и того же дескриптора приводят к исключениям (исключения и обработка исключений обсуждаются в главе 4). Не только излишне, но и не следует закрывать дескрипторы стандартных устройств, которые обсуждаются в разделе "Стандартные устройства и консольный ввод/вывод" далее в этой главе.

```
BOOL CloseHandle(HANDLE hObject)
```

Возвращаемое значение: в случае успешного выполнения функции — TRUE, иначе — FALSE.

Функции UNIX, сопоставимые с рассмотренными выше, отличаются от них в нескольких отношениях. Функция (системный вызов) UNIX open возвращает целочисленный дескриптор (descriptor) файла, а не дескриптор типа HANDLE, причем для указания всех параметров доступа, разделения и создания файлов, а также атрибутов и флагов используется единственный целочисленный параметр oflag. Возможные варианты выбора, доступные в обеих системах, перекрываются, однако набор опций, предлагаемый Windows, отличается большим разнообразием.

В UNIX отсутствует параметр, эквивалентный параметру dwShareMode. Файлы UNIX всегда являются разделяемыми.

В обеих системах при создании файла используется информация, касающаяся его защиты. В UNIX для задания хорошо известных разрешений на доступ к файлу для владельца, членов группы и прочих пользователей используется аргумент mode.

Функция close, хотя ее и можно сопоставить с функцией CloseHandle, отличается от последней меньшей универсальностью.

Функции библиотеки C, описанные в заголовочном файле <stdio.h>, используют объекты FILE, которые можно поставить в соответствие дескрипторам (дисковые файлы, терминалы, ленточные устройства и тому подобные), связанным с потоками. Параметр mode функции fopen позволяет указать, должны ли содержащиеся в файле данные обрабатываться как двоичные или как текстовые. Имеются также опции открытия файла в режиме "только чтение", обновления файла, присоединения к другому файлу и так далее. Функция freopen обеспечивает возможность повторного использования объектов FILE без их предварительного закрытия. Средства для задания параметров защиты стандартной библиотекой C не предоставляются.

Для закрытия объектов типа FILE предназначена функция fclose. Имена большинства функций стандартной библиотеки C, предназначенных для работы с объектами FILE, снабжены префиксом "f".

Чтение файла

```
BOOL ReadFile(HANDLE hFile, LPVOID lpBuffer, DWORD
nNumberOfBytesToRead, LPDWORD lpNumberOfBytesRead, LPOVERLAPPED
lpOverlapped)
```

Возвращаемое значение: в случае успешного выполнения (которое считается таковым, даже если не был считан ни один байт из-за попытки чтения с выходом за пределы файла) — TRUE, иначе — FALSE.

Вплоть до главы 14 мы будем предполагать, что дескрипторы файлов создаются *без* указания флага перекрывающегося ввода/вывода FILE_FLAG_OVERLAPPED в параметре dwAttrsAndFlags. В этом случае функция ReadFile начинает чтение с текущей позиции указателя файла, и указатель файла сдвигается на число считанных байтов.

Если значения дескриптора файла или иных параметров, используемых при вызове функции, оказались недействительными, возникает ошибка, и функция возвращает значение FALSE. Попытка выполнения чтения в ситуациях, когда указатель файла позиционирован в конце файла, не приводит к ошибке; вместо этого количество считанных байтов (*lpNumberOfBytesRead) устанавливается равным 0.

Параметры

Описательные имена переменных и естественный порядок расположения параметров во многом говорят сами за себя. Тем не менее, ниже приводятся некоторые краткие пояснения.

hFile — дескриптор считываемого файла, который должен быть создан с правами доступа GENERIC_READ. lpBuffer является указателем на буфер в памяти, куда помещаются считываемые данные. nNumberOfBytesToRead — количество байт, которые должны быть считаны из файла.

lpNumberOfBytesRead — указатель на переменную, предназначенную для хранения числа байт, которые были фактически считаны в результате вызова функции ReadFile. Этот параметр может принимать нулевое значение, если перед выполнением чтения указатель файла был позиционирован в конце файла или если во время чтения возникли ошибки, а также после чтения из именованного канала, работающего в режиме обмена сообщениями (глава 11), если переданное сообщение имеет нулевую длину.

lpOverlapped — указатель на структуру OVERLAPPED (главы 3 и 14). На данном этапе просто устанавливайте значение этого параметра равным NULL.

Запись в файл

```
BOOL WriteFile(HANDLE hFile, LPCVOID lpBuffer, DWORD nNumberOfBytesToWrite, LPDWORD lpNumberOfBytesWritten, LPOVERLAPPED lpOverlapped)
```

Возвращаемое значение: в случае успешного выполнения — TRUE, иначе — FALSE.

Все параметры этой функции вам уже знакомы. Заметьте, что успешное выполнение записи еще не говорит о том, что данные действительно оказались записанными на диск, если только при создании файла с помощью функции CreateFile не был использован флаг FILE_FLAG_WRITE_THROUGH. Если во время вызова функции указатель файла был позиционирован в конце файла, Windows увеличит длину существующего файла.

Функции ReadFileGather и WriteFileGather позволяют выполнять операции чтения и записи с использованием набора буферов различного размера.

Сопоставимыми функциями UNIX являются функции read и write, которым программист в качестве параметров должен предоставлять дескриптор файла, буфер и счетчик байтов. Возвращаемые значения этих функций указывают на количество фактически переданных байтов. Возврат функцией read значения 0 означает чтение конца файла, а значения –1 — возникновение ошибки. В противоположность этому в Windows для подсчета количества переданных байтов используется отдельный счетчик, а на успех или неудачу выполнения функции указывает возвращаемое ею булевское значение.

В обеих системах функции имеют сходное назначение и могут выполнять соответствующие операции с использованием файлов, терминалов, ленточных устройств, каналов и так далее.

Входящие в состав стандартной библиотеки C функции read и fwrite, выполняющие операции ввода/вывода в двоичном режиме, вместо счетчика одиночных байтов, как в UNIX и Windows, используют размер объекта и счетчик объектов. Преждевременное прекращение передачи данных может быть вызвано как достижением конца файла, так и возникновением ошибки; точная причина устанавливается с использованием функций feof или ferror. Библиотека предоставляет полный набор функций, ориентированных на работу с текстовыми файлами, таких как fgets или fputs, для которых в каждой из рассматриваемых ОС аналоги вне библиотеки C отсутствуют.

Вступление: стандартные символы и символы Unicode

Прежде чем двигаться дальше, необходимо кратко объяснить, как Windows обрабатывает символы и различает 8-битовые, 16-битовые и обобщенные символы. Эта тема весьма обширна и выходит за рамки данной книги, поэтому мы не будем выделять ее обсуждение в отдельную главу и ограничимся приведением лишь самых необходимых сведений в минимальном объеме.

Windows поддерживает стандартные 8-битовые символы (типы char или CHAR) и (исключая Windows 9x) 16-битовые символы расширенной формы (тип WCHAR, определенный в библиотеке C как wchar_t). В документации Microsoft 8-битовый набор фигурирует как символьный набор ASCII, хотя фактически он является символьным набором Latin-1, однако в целях удобства изложения название "ASCII" будет использоваться и в нашем обсуждении. Обеспечиваемая Windows с использованием кодировки Unicode UTF-16 поддержка обобщенных символов расширенной формы позволяет представлять в стандарте Unicode символы и буквы, встречающиеся во всех основных языках, включая английский, французский, испанский, немецкий, русский, японский и китайский.

Ниже описаны шаги, которые обычно предпринимаются при написании обобщенных (generic) Windows-приложений, то есть приложений, предусматривающих использование как символов Unicode (UTF-16, а не, например, UCS-4), так и 8-битовых ASCII-символов.

1. Определите все символы и строки с использованием обобщенных типов TCHAR, LPTSTR и LPCTSTR.

2. Чтобы иметь возможность работать с символами в расширенной форме Unicode (wchar_t в ANSI C), включите во все модули исходного кода определения #define UNICODE и #define _UNICODE; если этого не сделать, то тип TCHAR будет эквивалентен типу CHAR (char в ANSI C). Это определение должно помещаться перед директивой #include <windows.h>, и его часто задают в командной строке компилятора. Первая из указанных переменных препроцессора управляет определениями функций Windows, вторая — библиотекой C.

3. Размеры буферов для хранения символов, указываемые, например, при вызове функций ReadFile, могут определяться с использованием функции sizeof(TCHAR).

4. Используйте входящие в состав библиотеки C функции ввода/вывода обобщенных символов и строк, описанные в файле <tchar.h>. В качестве наиболее характерных из доступных функций можно назвать такие функции, как _fgettc, _itot (вместо itoa), _stprintf (вместо sprintf), _tstcpy (вместо strcpy), _ttoi, _totupper, _totlower и _tprintf.[12] Полный и исчерпывающий список таких функций можно найти в оперативной справочной системе. Все перечисленные определения зависят от определения символьной константы _UNICODE. Описанная коллекция функций не является полной. Примером функции, для которой еще не реализован аналог, позволяющий работать с символами расширенной формы, может служить функция memchr. Новые версии предоставляются по мере возникновения необходимости в них.

5. Строковые константы могут принимать одну из трех допустимых форм. Эти же соглашения следует применять и к одиночным символам. Первые две формы предоставляются стандартом ANSI C, третья — макрос _T (эквиваленты — TEXT и _TEXT) — поставляется вместе с компилятором Microsoft C.

```
"В этой строке используются 8-битовые символы"
L"В этой строке используются 16-битовые символы"
_T("В этой строке используются обобщенные символы")
```

6. Чтобы получить доступ к необходимым определениям текстовых макросов и обобщенным функциям библиотеки C, в модуль следует включить заголовочный файл <tchar.h>, объявление которого должно предшествовать объявлению файла <windows.h>.

16-битовые символы Unicode (кодировка UTF-16) используются в Windows повсеместно; для внутреннего представления имен файлов и путей доступа в файловой системе NTFS также используется Unicode. Если определена символьная константа _UNICODE, то все вызовы функций Windows требуют использования строк, состоящих из расширенных символов; в противном случае строки 8-битовых символов преобразуются в расширенные строки. В случае программ, которые должны выполняться под управлением систем Windows 9x, не являющихся Unicode-системами, определять символические константы UNICODE и _UNICODE *не следует.* В средах NT или CE решение об использовании указанных определений вы принимаете по своему усмотрению, если только для программы не должна быть одновременно сохранена возможность выполнения под управлением Windows 9x.

Во всех последующих примерах вместо обычного типа char для символов и символьных строк будет использоваться тип TCHAR, если только по каким-то вполне обоснованным причинам не возникнет необходимости в обработке отдельных 8-битовых символов. Точно так же, тип LPTSTR соответствует указателю на обобщенную строковую переменную, а тип LPCTSTR — указателю на обобщенную строковую константу. В результате принятия этих мер программа может стать более громоздкой, однако лишь своевременный учет различных возможных вариантов обеспечивает гибкость, необходимую для разработки и тестирования приложений, допускающих как кодировку Unicode, так и 8-битовую кодировку символов, что позволит легко преобразовать программу к использованию символов Unicode, если впоследствии в этом возникнет необходимость. Более того, предоставление возможности выбора между обеими разновидностями кодировок соответствует общепринятой, если не универсальной, практике, которая сложилась к настоящему времени.

Немалую пользу может принести просмотр системных заголовочных файлов, изучив которые вы поймете, как определяются тип TCHAR и интерфейсы системных функций и как они зависят от того, определены или не определены символьные константы UNICODE и _UNICODE. Соответствующие строки обычно выглядят так:

```
#ifdef UNICODE
#define TCHAR WCHAR
#else
#define TCHAR CHAR
#endif
```

Альтернативные функции для работы с обобщенными строками

В тех случаях, когда при сравнении строк необходим учет специфики языковых и региональных, или *локальных,* особенностей на стадии выполнения, или же когда требуется сравнивать не *строки,* а *слова,*[13] то вместо функций _tcscmp и _tcscmpi вам могут понадобиться функции lstrcmp и lstrcmpi. Сравнение строк осуществляется путем простого сравнения числовых значений символов, тогда как при сравнении слов принимаются во внимание специфические для конкретного языка особенности словообразования. Если применить указанные два метода сравнения к таким парам строк, как *coop/co-op* и *were/we're,* то они приведут к противоположным результатам.

В Windows также существует группа функций, предназначенных для работы с символами и строками в представлении Unicode. Эти функции обеспечивают прозрачную обработку региональных особенностей. Типичными функциями этой группы являются функция CharUpper, которую можно применять как к строкам, так и к отдельным символам, и функция IsCharAlphaNumeric. К числу других функций для работы со строками принадлежат функция CompareString (учитывающая особенности локализации) и функция MultiByteToWideChar.

Многобайтовые символы Windows 3.1 и 9x расширяют наборы 8-битовых символов, позволяя применять для представления наборов символов, используемых в языках дальневосточных стран, сдвоенные байты. Чтобы продемонстрировать использование функций обоих типов, будут рассмотрены примеры программ, в которых используются как обобщенные функции библиотеки C (_tprintf и подобные ей), так и функции Windows (CharUpper и подобные ей). Примеры в последующих главах в основном опираются на обобщенную библиотеку C.

Обобщенная функция Main

Обозначение C-функции main с ее списком аргументов (argv[]) следует заменить макросом _tmain. В зависимости от определения символической константы _UNICODE макрос разворачивается либо до main, либо до wmain. _tmain определяется в заголовочном файле <tchar.h>, который следует включать после файла <windows.h>. Тогда типичный заголовок основной программы будет иметь следующий вид:

```
#include <windows.h>
#include <tchar.h>
int _tmain(int argc, LPTSTR argv[]) {
 …
}
```

В Microsoft C функция _tmain поддерживает дополнительный третий параметр, используемый для строк окружения. Такое нестандартное расширение является обычным в UNIX.

Определения функций

В качестве примера рассмотрим функцию CreateFile. Если символьная переменная UNICODE определена, то эта функция определяется как CreateFileA, а если не определена — то как CreateFileW. Строковые параметры в объявлениях также описываются как строки 8-битовых символов или символов в расширенной форме. Следовательно, если в исходном коде присутствуют такие, например, ошибки, как использование неподходящих параметров в функции CreateFile, то в сообщениях компилятора об этих ошибках будут указываться функции CreateFileA или CreateFileW.

Стратегии использования символов Unicode

Приступая к работе над проектом в Windows, либо для разработки нового программного кода, либо для переноса существующего, программист, в зависимости от требований проекта, может выбрать одну из четырех стратегий.

1 . **Только 8-битовые символы.** Игнорируйте Unicode и продолжайте использовать для таких функций, как printf, atoi и strcmp, типы данных char (или CHAR) и стандартную библиотеку C.

2. **8-битовые символы, но с возможностью использования символов Unicode.** Следуйте ранее данным рекомендациям в отношении обобщенных приложений, но не определяйте константы UNICODE и _UNICODE директивами препроцессора. В приведенных в данной книге примерах программ используется именно эта стратегия.

3. **Только символы Unicode.** Следуйте рекомендациям в отношении обобщенных приложений, но при этом определите директивами препроцессора обе константы UNICODE и _UNICODE. Другой возможный вариант состоит в том, чтобы использовать исключительно расширенную форму символов и функций для работы с символами. Результирующие программы не смогут правильно выполняться под управлением Windows 9x.

4. **Символы Unicode и 8-битовые символы.** Программа ориентируется на работу как с символами Unicode, так и с ASCII-символами, причем решение относительно того, какие участки программного кода должны работать, принимается программой на стадии выполнения с использованием переключателей времени выполнения или других возможных средств.

Как уже отмечалось ранее, несмотря на то что написание обобщенного кода требует дополнительных усилий, а результирующая программа становится менее удобочитаемой, эта мера позволяет программисту добиться максимальной гибкости приложения.

Параметры локализации могут устанавливаться во время выполнения программы. В программе 2.2 показано, как определить язык, который должен использоваться в сообщениях об ошибках.

> Стандарт локализации приложений POSIX XPG4, предоставляемый многими поставщиками UNIX, существенно отличается от стандарта Unicode. Помимо всего прочего, символы в этом стандарте могут представляться 4, 3 или 1 байтами в зависимости от контекста, особенностей локализации и так далее.
>
> Microsoft C реализует функции стандартной библиотеки C, среди которых имеются также версии, рассчитанные на работу с символами в расширенной форме. Так, заголовочный файл <wchar.h> содержит описание функции _tsetlocale. В Windows NT используются символы Unicode, тогда как в Windows 9x используются те же многобайтовые символы (смесь 8– и 16-битовых символов), что и в Windows 3.1.

Стандартные устройства и консольный ввод/вывод

Как и в UNIX, в Windows предусмотрены три стандартных устройства, предназначенные, соответственно, для ввода данных (input), вывода данных (output) и вывода сообщений об ошибках (error). В UNIX для этих устройств используются известные системе значения дескрипторов файлов (0, 1 и 2), однако в Windows доступ к стандартным устройствам осуществляется с помощью дескрипторов типа HANDLE, для получения которых предоставляется специальная функция.

```
HANDLE GetStdHandle(DWORD nStdHandle)
```

Возвращаемое значение: в случае успешного выполнения — действительный дескриптор, иначе — значение INVALID_HANDLE_VALUE.

Параметры

Параметр nStdHandle должен принимать одно из следующих значений:

- `STD_INPUT_HANDLE`
- `STD_OUTPUT_HANDLE`
- `STD_ERROR_HANDLE`

В качестве стандартных устройств обычно назначаются консоль и клавиатура. Стандартный ввод/вывод можно перенаправлять на другие устройства.

Вызов функции GetStdHandle не приводит к созданию новых или дублированию существующих дескрипторов стандартных устройств. Последовательные вызовы с указанием в качестве аргумента одного и того же устройства будут возвращать одно и то же значение дескриптора. Закрытие дескриптора стандартного устройства делает это устройство недоступным для дальнейшего использования. По этой причине в примерах ниже мы будем часто открывать дескриптор стандартного устройства, но не закрывать его.

```
BOOL SetStdHandle(DWORD nStdHandle, HANDLE hHandle)
```

Возвращаемое значение: в случае успешного выполнения — TRUE, иначе — FALSE.

Параметры

Допустимые значения параметра nStdHandle функции SetStdHandle являются теми же, что и в случае функции GetStdHandle. Параметр hHandle указывает открытый файл, который назначается в качестве стандартного устройства.

Одним из методов перенаправления стандартного ввода/вывода является последовательный вызов функций SetStdHandle и GetStdHandle. Полученный в результате этого дескриптор используется в последующих операциях ввода/ вывода.

Для указания путей доступа к консольному вводу (клавиатуре) и консольному выводу предусмотрены два зарезервированных имени: "CONIN$" и "CONOUT$". Роль стандартных устройств ввода, вывода и вывода ошибок первоначально отводится консоли. Однако консоль можно использовать даже после того, как операции ввода/вывода, требующие применения стандартных устройств, будут перенаправлены; для этого требуется лишь открыть дескрипторы для файлов "CONIN$" и "CONOUT$", вызвав функцию CreateFile.

В UNIX стандартный ввод/вывод может быть перенаправлен одним из трех способов (см. [40], стр. 61—64).

Первый метод является косвенным и основывается на том, что функция dup возвращает дескриптор файла с наименьшим доступным номером. Предположим, вы хотите переназначить стандартный ввод (файловый дескриптор 0) открытому файлу, описанному как fd_redirect. Тогда можно записать следующий код:

```
close (STDIN_FILENO);
dup (fd_redirect);
```

Во втором методе используется функция dup2, а третий метод предполагает вызов замысловатой перегруженной функции fcntl с использованием в качестве параметра значения F_DUPFD.

Операции консольного ввода/вывода могут выполняться с помощью функций ReadFile и WriteFile, но проще использовать предназначенные специально для этого функции консоли ReadConsole и WriteConsole. Основное преимущество этих функций заключается в том, что они манипулируют не байтами, а обобщенными символами (TCHAR), и, кроме того, обрабатывают символы в соответствии с текущими режимами консоли, которые устанавливаются функцией SetConsoleMode.

```
BOOL SetConsoleMode(HANDLE hConsoleHandle, DWORD fdevMode)
```

Возвращаемое значение: тогда, и только тогда, когда функция завершается успешно — TRUE, иначе — FALSE.

Параметры

hConsoleHandle — дескриптор буфера ввода консоли или буфера дисплея, который должен быть создан с правами доступа GENERIC_WRITE, даже если устройство предназначено только для ввода информации.

Параметр fdevMode задает способ обработки символов. В имени каждого из его флагов содержится компонент, указывающий, к чему относится данный флаг— к вводу (input) или выводу (output). Ниже перечислены пять обычно используемых флагов, причем все они устанавливаются по умолчанию.

• ENABLE_LINE_INPUT — возврат из функции чтения (ReadConsole) происходит только после считывания символа возврата каретки.

• ENABLE_ECHO_INPUT — эхо-отображение вводимых символов на экране.

• ENABLE_PROCESSED_INPUT — установка этого флага приводит к обработке системой управляющих символов возврата на одну позицию, возврата каретки и перехода на новую строку.

• ENABLE_PROCESSED_OUTPUT — установка этого флага приводит к обработке системой управляющих символов возврата на одну позицию, табуляции, подачи звукового сигнала, возврата каретки и перехода на новую строку.

• ENABLE_WRAP_AT_EOL_OUTPUT — переход на следующую строку экрана как при обычном выводе символов, так и при их эхо-отображении в процессе ввода.

В случае неудачного завершения функции SetConsoleMode текущий режим остается неизменным, и функция возвращает значение FALSE. Как обычно, для получения номера ошибки следует воспользоваться функцией GetLastError.

Функции ReadConsole и WriteConsole аналогичны функциям ReadFile и WriteFile.

```
BOOL ReadConsole(HANDLE hConsoleInput, LPVOID lpBuffer, DWORD
cchToRead, LPDWORD lpcchRead, LPVOID lpReserved)
```

Возвращаемое значение: тогда, и только тогда, когда функция завершается успешно — TRUE, иначе — FALSE.

Параметры у этой функции почти те же, что и у функции ReadFile. Значения обоих параметров, связанных с количеством подлежащих считыванию (cchToRead) и фактически считанных (lpcchRead) символов, выражаются в терминах обобщенных символов, а не байтов, а значение параметра lpReserved должно быть равным NULL. Как и во всех остальных подобных случаях, никогда не используйте для собственных нужд зарезервированные поля, аналогичные lpReserved, которые встречаются в некоторых функциях. Параметры функции WriteConsole имеют тот же смысл и не нуждаются в дополнительных пояснениях. В очередном примере будет проиллюстрировано применение функций Read-Console и WriteConsole, и, кроме того, будет показано, как использовать возможности управления режимом консоли.

Любому процессу в каждый момент времени может быть назначена только одна консоль. Приложениям того типа, с которым мы имели дело до сих пор, консоль передается обычно на стадии инициализации. Однако в целом ряде других случаев, например, в случае серверных или GUI-приложений, у вас может возникнуть необходимость в получении отдельной консоли, на которую можно было бы выводить информацию о состоянии программы или отладочную информацию. Для этих целей можно воспользоваться двумя простыми функциями, не имеющими параметров.

```
BOOL FreeConsole(VOID)
BOOL AllocConsole(VOID)
```

Функция FreeConsole отключает процесс от его консоли, тогда как функция AllocConsole создает новую консоль, ассоциированную с дескрипторами стандартного ввода информации, стандартного вывода информации и стандартного вывода сообщений об ошибках, принадлежащими данному процессу. Если консоль у процесса уже имеется, функция AllocConsole завершится с ошибкой; чтобы избежать этого, следует предварительно вызывать функцию FreeConsole.

Примечание

GUI-приложения Windows не имеют консоли по умолчанию и должны получить ее, прежде чем смогут воспользоваться функциями WriteConsole или printf для вывода на консоль. Процессы на стороне сервера также могут не иметь консоли. О том, как создать процесс без консоли, рассказано в главе 6.

Имеется также множество других функций консольного ввода/вывода, предназначенных для установки позиции курсора, а также задания атрибутов выводимых символов (например, цвета) и так далее. Принятый в данной книге подход состоит в том, чтобы использовать лишь те функции, которые необходимы для создания примеров работоспособных программ, поэтому углубляться больше, чем это необходимо, в пользовательские интерфейсы мы не будем. После того как вы разберете примеры, для вас не составит большого труда изучить дополнительные функции, воспользовавшись справочными материалами.

Исторически сложилось так, что ОС Windows ориентирована на использование терминалов или консолей в меньшей степени, чем UNIX, и не полностью

воспроизводит функциональные средства UNIX, поддерживающие работу с терминалами. В книге [40] одна из глав посвящена рассмотрению обеспечиваемых UNIX возможностей терминального ввода/вывода (глава 11), а другая — псевдотерминалам (глава 19).

Разумеется, работа в Windows почти всегда ведется с использованием мощных графических интерфейсов, поддерживающих мышь и ввод с клавиатуры. Несмотря на то что рассмотрение GUI выходит за рамки данной книги, все, что мы здесь обсуждаем, будет работать и в GUI-приложениях.

Пример: вывод на консоль сообщений и подсказок для пользователя

Функция ConsolePrompt, входящая в программу 2.1, является полезной утилитой, которая выводит на консоль заданное сообщение и возвращает ответ пользователя на него. Данная утилита предусматривает возможность подавления эхо-отображения ответной информации, полученной от пользователя. В указанной функции используются функции консольного ввода/вывода и обобщенные символы. Двумя другими точками входа в этом модуле являются функции Print-Strings и PrintMsg; эти функции допускают использование любого дескриптора, однако обычно они применяются совместно с дескрипторами устройств стандартного вывода информации и стандартного вывода сообщений об ошибках. В первой функции разрешается использовать список аргументов переменной длины, тогда как во второй в качестве аргумента можно задавать только одну строку, что в некоторых случаях может оказаться удобнее. Для обработки списка аргументов переменной длины функция PrintStrings использует функции va_start, va_arg и va_end стандартной библиотеки C.

Описанные функции, а также функции из обобщенной библиотеки C будут привлекаться для использования в приводимых в данной книге примерах программ при всякой удобной возможности.

> **Примечание**
>
> Коды программ, находящиеся на Web-сайте книги, содержат подробные комментарии и тщательно документированы, тогда как в самой книге большинство комментариев с целью сокращения места были опущены, и основное внимание в ней уделяется использованию Windows.

Следует также отметить, что в примере вводится заголовочный файл Envirmnt.h (его код приведен в приложении A и предоставлен на Web-сайте книги), который должен использоваться совместно со всеми приводимыми в книге программами. Этот файл содержит определения символических констант UNICODE и _UNICODE (сами определения "закомментированы"; при компоновке приложений, предназначенных для работы с символами стандарта Unicode, символы комментариев следует удалить), а также необходимых макропеременных, учитывающих особенности окружения. В заголовочных файлах, находящихся на Web-сайте, определены также дополнительные модификаторы, которые обеспечивают импортирование и экспортирование имен функций, а также гарантируют соблюдение соответствующих соглашений о вызове функций.

Программа 2.1. PrintMsg: вспомогательные функции вывода на консоль сообщений и ожидания ответа от пользователя

```
    /* PrintMsg.c: ConsolePrompt, PrintStrings, PrintMsg */
    #include "Envirmnt.h" /* В этом файле устанавливаются директивы #define и
#undef для UNICODE. */
    #include <windows.h>
    #include <stdarg.h>

    BOOL PrintStrings (HANDLE hOut, ...)
```

```
    /* Запись сообщений в буфер экрана консоли. */
    {
     DWORD MsgLen, Count;
     LPCTSTR pMsg;
     va_list pMsgList; /* Строка текущего сообщения. */
     va_start (pMsgList, hOut); /* Начать обработку сообщений. */
     while ((pMsg = va_arg(pMsgList, LPCTSTR)) != NULL) {
      MsgLen = _tcslen(pMsg);
       /* Функция WriteConsole может применяться только с дескриптором буфера экрана
консоли. */
      if (!WriteConsole(hOut, pMsg, MsgLen, &Count, NULL)
            /* Функция WriteFile вызывается только в случае неудачного завершения
функции WriteConsole. */
          && !WriteFile(hOut, pMsg, MsgLen * sizeof (TCHAR), &Count, NULL)) return
FALSE;
     }
     va_end(pMsgList);
     return TRUE;
    }


    BOOL PrintMsg(HANDLE hOut, LPCTSTR pMsg)
    /* Версия PrintStrings для вывода одиночного сообщения. */
    {
     return PrintStrings(hOut, pMsg, NULL);
    }


    BOOL ConsolePrompt(LPCTSTR pPromptMsg, LPTSTR pResponse, DWORD MaxTchar, BOOL
Echo)
    /* Вывести на консоль подсказку для пользователя и получить от него ответ. */
    {
     HANDLE hStdIn, hStdOut;
     DWORD TcharIn, EchoFlag;
     BOOL Success;
      hStdIn = CreateFile(_T("CONIN$"), GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
      hStdOut = CreateFile(_T("CONOUT$"), GENERIC_WRITE, 0, NULL, OPEN_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
     EchoFlag = Echo ? ENABLE_ECHO_INPUT : 0;
       Success = SetConsoleMode(hStdIn, ENABLE_LINE_INPUT | EchoFlag |
ENABLE_PROCESSED_INPUT) &&
                          SetConsoleMode (hStdOut, ENABLE_WRAP_AT_EOL_OUTPUT |
ENABLE_PROCESSED_OUTPUT) &&
               PrintStrings (hStdOut, pPromptMsg, NULL) &&
               ReadConsole (hStdIn, pResponse, MaxTchar, &TcharIn, NULL);
     if (Success) pResponse [TcharIn - 2] = '\0';
     CloseHandle (hStdIn);
     CloseHandle (hStdOut);
     return Success;
    }
```

Обратите внимание, что при вычислении возвращаемого функцией значения булевской переменной Success, которое служит индикатором успешности выполнения, в программе, с выгодой для логики ее работы, используется тот факт, что стандартом ANSI C гарантируется так называемое "сокращенное" вычисление логических выражений в направлении слева направо; поэтому, как только при вычислении части выражения, расположенной слева от любой из операций логического "и" (&&), в качестве результата будет получено значение FALSE, остальная часть выражения, расположенная справа от данной операции, вычисляться не будет, поскольку результат вычисления всего выражения в целом оказывается предопределенным.

Данный стиль написания программ может показаться чересчур компактным, однако он обладает тем преимуществом, что позволяет организовать логически стройную и понятную последовательность системных вызовов, не загромождая программу многочисленными операторами условных переходов. Для получения более подробной информации о возможных ошибках можно воспользоваться функцией GetLastError. Распространенный в Windows возврат функциями логических значений поощряет подобную практику.

В данной функции сообщения об ошибках не выводятся; их вывод, если это будет необходимо, можно предусмотреть в вызывающей программе.

В программном коде используется тот документированный факт, что при использовании функции WriteConsole вместе с дескриптором, который не является дескриптором консоли, ее выполнение будет завершено с ошибкой. В связи с этим предварительный запрос свойств дескриптора не является обязательным. Функция воспользуется консольным режимом лишь в том случае, если указанный в ее вызове дескриптор действительно связан с консолью.

Кроме того, функция ReadConsole возвращает управляющие символы возврата каретки и перехода на новую строку, что диктует необходимость вставки дополнительных нулевых символов после символов возврата каретки в соответствующих местах.

Пример: обработка ошибок

В программе 1.2 было продемонстрировано использование лишь самых примитивных средств обработки ошибок, а именно, получение номера ошибки в переменной типа DWORD с помощью функции GetLastError. Вызов функции, а не просто получение глобального номера ошибки, как это делается при помощи функции UNIX errno, гарантирует уникальную идентификацию системных ошибок для каждого из потоков (глава 7), использующих разделяемую область хранения данных.

Функция FormatMessage превращает простой номер сообщения в описательное сообщение, представляющее собой фразу на английском или любом другом из множества возможных языков, и возвращает размер сообщения.

В программе 2.2 представлена полезная универсальная функция ReportError, предназначенная для обработки ошибок и по своим возможностям аналогичная входящей в состав библиотеки C функции perror, а также описанным в [40] функциям err_sys и err_ret. Функция ReportError передает в выходной буфер сообщение в виде, определяемом первым аргументом, и либо прекращает выполнение с кодом выхода по ошибке, либо осуществляет обычный возврат управления, в зависимости от значения второго аргумента. Третий аргумент определяет, должны ли отображаться системные сообщения об ошибках.

Обратите внимание на аргументы функции FormatMessage. В качестве одного из параметров используется значение, возвращаемое функцией GetLastError, а на необходимость генерации сообщения системой указывает флаг. Сгенерированное сообщение сохраняется в буфере, выделяемом функцией, а соответствующий адрес возвращается в параметре. Имеются также другие параметры, для которых указаны значения по умолчанию. Язык сообщений может быть задан как во время компиляции, так и во время выполнения. В этой книге функция Format-Message далее нигде не используется, поэтому никаких дополнительных пояснений относительно нее в тексте не дается.

Функция ReportError упрощает обработку ошибок, и будет использоваться почти во всех последующих примерах. В главе 4 она будет модифицирована для генерации исключений.

В программе 2.2 вводится заголовочный файл EvryThng.h. Как следует из самого его названия, этот файл включает в себя файлы <windows.h>, Envirmnt.h и все остальные заголовочные файлы, которые были явно указаны в программе 2.1. Кроме того, в нем описаны такие обычно используемые функции, как PrintMsg, PrintStrings и ReportError. Во всех остальных примерах будет использоваться только этот заголовочный файл, листинг которого приведен в приложении A.

Обратите внимание на вызов функции HeapFree, находящийся почти в конце программы. Об этой функции будет рассказано в главе 5.

Программа 2.2. Функция Report Error, предназначенная для вывода сообщений об ошибках при выполнении системных вызовов

```
#include "EvryThng.h"

VOID ReportError(LPCTSTR UserMessage, DWORD ExitCode, BOOL PrintErrorMsg)
/* Универсальная функция для вывода сообщений о системных ошибках. */
{
 DWORD eMsgLen, LastErr = GetLastError();
```

```
    LPTSTR lpvSysMsg;
    HANDLE hStdErr = GetStdHandle(STD_ERROR_HANDLE);
    PrintMsg(hStdErr, UserMessage);
    if (PrintErrorMsg) {
              eMsgLen     =     FormatMessage(FORMAT_MESSAGE_ALLOCATE_BUFFER     |
FORMAT_MESSAGE_FROM_SYSTEM,      NULL,      LastErr,      MAKELANGID(LANG_NEUTRAL,
SUBLANG_DEFAULT), (LPTSTR)&lpvSysMsg, 0, NULL);
     PrintStrings (hStdErr, _T("\n"), lpvSysMsg, _T("\n"), NULL);
     /* Освободить блок памяти, содержащий сообщение об ошибке. */
     HeapFree(GetProcessHeap(), 0, lpvSysMsg); /* См. гл. 5. */
    }
    if (ExitCode > 0) ExitProcess (ExitCode);
    else return;
   }
```

Пример: копирование нескольких файлов на стандартное устройство вывода

В программе 2.3 иллюстрируется использование стандартных устройств ввода/вывода, а также демонстрируется, как улучшить контроль ошибок и усовершенствовать взаимодействие с пользователем. Эта программа представляет собой вариант ограниченной реализации команды UNIX cat, которая копирует один или несколько заданных файлов (или содержимое буфера стандартного устройства ввода, если файлы не указаны) на стандартное устройства вывода.

Программа 2.3 включает полную обработку ошибок. В большинстве других примеров проверка ошибок опущена или сведена к минимуму, но полностью включена в завершенные документированные варианты программ, находящиеся на Web-сайте. Обратите внимание на функцию Options (ее листинг приведен в приложении А), вызываемую в начале программы. Эта функция, которая включена в состав программ, находящихся на Web-сайте, и используется на протяжении всей книги, просматривает параметры в командной строке и возвращает индекс массива argv, соответствующий имени первого файла. Функция Options аналогична функции getopt, которая используется во многих программах в UNIX.

Программа 2.3. cat: вывод нескольких файлов на стандартное устройство вывода

```
    /* Глава 2. cat. */
    /* cat [параметры] [файлы] Допускается только параметр -s, предназначенный для подавления вывода сообщений об ошибках в случае, если один из указанных файлов не существует. */
    #include "EvryThng.h"
    #define BUF_SIZE 0x200

    static VOID CatFile(HANDLE, HANDLE);
    int _tmain(int argc, LPTSTR argv[]) {
     HANDLE hInFile, hStdIn = GetStdHandle(STD_INPUT_HANDLE);
     HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
     BOOL DashS;
     int iArg, iFirstFile;
     /* Переменная DashS будет установлена только в случае задания параметра "-s" в командной строке. */
     /* iFirstFile — индекс первого входного файла в списке argv[]. */
     iFirstFile = Options(argc, argv, _T("s"), &DashS, NULL);
     if (iFirstFile == argc) { /*Отсутствие входных файлов в списке аргументов.*/
      /* Использовать стандартное устройство ввода. */
      CatFile(hStdIn, hStdOut);
      return 0;
     }
     /* Обработать каждый входной файл. */
     for (iArg = iFirstFile; iArg < argc; iArg++) {
       hInFile = CreateFile(argv [iArg], GENERIC_READ, 0, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
      if (hInFile == INVALID_HANDLE_VALUE && !DashS) ReportError (_T("Cat: ошибка при открытии файла"), 1, TRUE);
      CatFile (hInFile, hStdOut);
      CloseHandle (hInFile);
     }
     return 0;
    }
```

```
    /* Функция, выполняющая всю работу:
    /* читает входные данные и копирует их на стандартное устройства вывода. */
    static VOID CatFile(HANDLE hInFile, HANDLE hOutFile) {
     DWORD nIn, nOut;
     BYTE Buffer [BUF_SIZE];
      while (ReadFile(hInFile, Buffer, BUF_SIZE, &nIn, NULL) && (nIn != 0) &&
WriteFile(hOutFile, Buffer, nIn, &nOut, NULL));
     return;
    }
```

Пример: преобразование символов из ASCII в Unicode

Программа 2.4 достраивает программу 1.3, в которой использовалась вспомогательная функция CopyFile. С копированием файлов вы уже знакомы, поэтому в данном примере эта операция дополняется преобразованием файла к кодировке Unicode в предположении, что первоначальной кодировкой символов является ASCII, хотя проверка этого предположения не производится. В программе предусмотрены некоторые возможности вывода сообщений об ошибках и параметр, позволяющий подавить замену существующего файла; завершающий вызов функции CopyFile заменен в программе вызовом новой функции, которая Выполняет преобразование символьных строк файла из кодировки ASCII в кодировку Unicode.

В данной программе основное внимание уделяется обеспечению возможности успешного завершения преобразования. Фактическое выполнение преобразования сосредоточено в единственной функции, вызываемой в самом конце программы. Этот фрагмент, как и аналогичный ему фрагмент предыдущей программы, послужит нам шаблоном и будет вновь использоваться в последующих программах без повторения его исходного кода.

Обратите внимание на вызов функции _taccess, проверяющей существование файла. Эта функция является обобщенной версией функции access, которая имеется в библиотеке UNIX, но не входит в состав стандартной библиотеки C. Ее определение содержится в файле <io.h>. Если говорить точнее, функция _taccess осуществляет проверку прав доступа к файлу в соответствии с режимом, установленным значением второго параметра. Значение 0 задает проверку существования файла, 2 — проверку наличия разрешения на запись в файл, 4 — проверку наличия разрешения на чтение из файла, 6 — проверку наличия разрешения как на чтение из файла, так и на запись в файл (эти значения не связаны напрямую с такими параметрами доступа, используемыми в Windows, как GENERIC_READ). Альтернативой проверке существования файла могло бы быть открытие дескриптора при помощи функции CreateFile и его последующее закрытие после проверки действительности дескриптора.

Программа 2.4. atou: преобразование файла с выводом сообщений об ошибках

```
/* Глава 2. atou - копирование файлов с преобразованием из ASCII в Unicode. */
#include "EvryThng.h"

BOOL Asc2Un(LPCTSTR, LPCTSTR, BOOL);
int _tmain(int argc, LPTSTR argv[]) {
 DWORD LocFileIn, LocFileOut;
 BOOL DashI = FALSE;
 TCHAR YNResp[3] = _T("y");
 /* Получить параметры командной строки и индекс входного файла. */
 LocFileIn = Options(argc, argv, _T("i"), &DashI, NULL);
 LocFileOut = LocFileIn + 1;
 if (DashI) { /* Существует ли выходной файл? */
   /* Обобщенная версия функции access, осуществляющая проверку существования
файла. */
  if (_taccess(argv[LocFileOut], 0) == 0) {
   _tprintf(_T("Перезаписать существующий файл? [y/n]"));
   _tscanf(_T ("%s"), &YNResp);
      if (lstrcmp(CharLower(YNResp), YES) != 0) ReportError(_T("Отказ от
перезаписи"), 4, FALSE);
  }
 }
```

```
     /* Эта функция построена на основе функции CopyFile. */
     Asc2Un(argv[LocFileIn], argv [LocFileOut], FALSE);
     return 0;
    }
```

Программа 2.5 — это вызываемая в программе 2.4 функция Asc2Un, осуществляющая преобразование кодировки символов.

Программа 2.5. Функция Asc2Un

```
    #include "EvryThng.h"
    #define BUF_SIZE 256

    BOOL Asc2Un(LPCTSTR fIn, LPCTSTR fOut, BOOL bFailIfExists)
    /* Функция копирования файлов с преобразованием из ASCII в Unicode. Функция
построена на основе функции CopyFile. */
    {
     HANDLE hIn, hOut;
     DWORD dwOut, nIn, nOut, iCopy;
     CHAR aBuffer[BUF_SIZE];
     WCHAR uBuffer [BUF_SIZE];
     BOOL WriteOK = TRUE;
        hIn    =    CreateFile(fin,    GENERIC_READ,    0,    NULL,    OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
     /* Определить поведение функции CreateFile, если выходной файл уже существует.
*/
     dwOut = bFailIfExists ? CREATE_NEW : CREATE_ALWAYS;
     hOut = CreateFile(fOut, GENERIC_WRITE, 0, NULL, dwOut, FILE_ATTRIBUTE_NORMAL,
NULL);
     while (ReadFile(hIn, aBuffer, BUF_SIZE, &nIn, NULL)  && nIn > 0 && WriteOK) {
      for (iCopy = 0; iCopy < nIn; iCopy++)
       /* Преобразовать каждый символ. */
       uBuffer[iCopy] = (WCHAR)aBuffer [iCopy];
      WriteOK = WriteFile(hOut, uBuffer, 2 * nIn, &nOut, NULL);
     }
     CloseHandle(hIn);
     CloseHandle(hOut);
     return WriteOK;
    }
```

Производительность программы

Как показано в приложении B, производительность программы преобразования файлов можно повысить, предоставив буфер большего размера и задав флаг FILE_FLAG_SEQUENTIAL_SCAN при вызове функции CreateFile. В приложении B также сравниваются показатели производительности программы для файловых систем NTFS и распределенных файловых систем.

Управление файлами и каталогами

В этом разделе вводятся основные функции, предназначенные для управления файлами и каталогами.

Управление файлами

Для управления файлами Windows предоставляет целый ряд функций, работа с которыми обычно не представляет сложности. Ниже описаны функции, с помощью которых можно удалять, копировать и переименовывать файлы. Существует также функция, предназначенная для создания имен временных файлов.

При удалении файла достаточно указать его имя. Вспомните, что все полные имена файлов начинаются с буквы диска или имени сервера. Открытый файл, вообще говоря, удалить невозможно (это допускается в Windows 9x и UNIX); попытка выполнения подобной операции закончится неудачей. У такого ограничения есть свои положительные стороны, поскольку оно предотвращает случайное удаление открытых файлов.

```
BOOL DeleteFile(LPCTSTR lpFileName)
```

Чтобы скопировать файл целиком, достаточно использовать одну функцию.

```
BOOL CopyFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName, BOOL fFailIfExists)
```

Функция CopyFile копирует существующий файл с заданным именем и присваивает копии указанное новое имя. В случае существования файла с таким же именем он будет заменен новым файлом только в том случае, если значением параметра fFailIfExists является FALSE.

Под управлением NT5 можно создать жесткую ссылку (hard link) для двух файлов, аналогичную жестким ссылкам в UNIX, используя для этого функцию CreateHardLink. Жесткие ссылки делают возможным существование файла под двумя различными именами. Заметьте, что в подобных случаях файл как таковой существует в единственном числе, и поэтому можно произвольно использовать любое из его имен, независимо от того, какое из них было использовано для открытия файла.

```
BOOL CreateHardLink(LPCTSTR lpFileName, LPCTSTR lpExistingFileName, BOOL lpSecurityAttributes)
```

Два первых аргумента имеют тот же смысл, что и в функции CopyFile, хотя и расположены в обратном порядке. Оба имени файла, новое и существующее, должны относиться к одному и тому же тому файловой системы, но могут соответствовать различным каталогам. Атрибуты защиты файла, если таковые имеются, применимы и к новому имени файла.

Если заглянуть в документацию Microsoft, то можно увидеть, что в структуре BY_HANDLE_FILE_INFO имеется поле "количество ссылок", и именно этот счетчик используется для определения того, может или не может быть удален данный файл. Функция DeleteFile удаляет имя из каталога файловой системы, но сам файл не может быть удален до тех пор, пока значение счетчика "количество ссылок" не станет равным 0.

Гибких ссылок (soft link) в Windows не существует, хотя оболочки Windows (но не сама

Windows), руководствующиеся при определении местоположения файла его содержимым, поддерживают ярлыки (shortcuts). Ярлыки предоставляют средства, подобные гибким ссылкам, но воспользоваться ими могут только пользователи оболочки.

Доступны две функции, позволяющие переименовывать, или "перемещать", файл. Эти же функции применимы и к каталогам. (Функции DeleteFile и CopyFile могут применяться только к файлам.)

```
BOOL MoveFile(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName)
BOOL MoveFileEx(LPCTSTR lpExistingFileName, LPCTSTR lpNewFileName,
DWORD dwFlags)
```

Если новый файл уже существует, функция MoveFile завершается с ошибкой; в этом случае следует использовать функцию MoveFileEx. Заметьте, что суффикс "Ex" обычно применяется для обозначения усовершенствованных версий функций, обладающих расширенными (extended) функциональными возможностями.

Параметры

lpExistingFileName — указатель на строку, содержащую имя существующего файла *или* каталога.

lpNewFileName — указатель на строку, содержащую имя нового файла *или* каталога, которые, в случае функции MoveFile, до ее вызова существовать не должны. Новый файл может принадлежать другой файловой системе или находиться на другом диске, но новые каталоги обязательно должны находиться на одном и том же диске. Если значение этого параметра положить равным NULL, то существующий файл будет удален.

dwFlags — позволяет задавать следующие опции:

• MOVEFILE_REPLACE_EXISTING — разрешает замену существующего файла.

• MOVEFILE_WRITETHROUGH — используйте этот флаг, если необходимо, чтобы функция выполнила возврат лишь после того, как файл будет фактически перемещен на диске.

• MOVEFILE_COPY_ALLOWED — если новый файл находится на другом томе, перемещение осуществляется путем последовательного выполнения функций CopyFile и DeleteFile.

• MOVEFILE_DELAY_UNTIL_REBOOT — установка этого флага, использование которого является прерогативой администратора системы и который не может применяться совместно с флагом MOVEFILE_COPY_ALLOWED, приводит к тому, что фактическое перемещение файла будет осуществлено только после перезагрузки системы.

С перемещением (переименованием) файлов связаны некоторые ограничения.

• В Windows 9x функция MoveFileEx не реализована; вместо нее вы должны использовать последовательные вызовы функций CopyFile и DeleteFile. Это делает возможным одновременное существование двух экземпляров файла, что порождает определенные проблемы в случае дисков, близких к заполнению, или файлов большого размера. При этом временные атрибуты файлов изменяются иначе, нежели при истинном перемещении.

• Использование групповых символов в именах файлов или каталогов запрещено. Необходимо указывать фактические имена.

> Полные имена файлов в UNIX не включают имен дисков и серверов; корневой системный каталог обозначается обратной косой чертой. В то же время, функции для

работы с файлами, входящие в библиотеку Microsoft C, поддерживают имена дисков, как того требуют соглашения Windows относительно именования файлов.

В UNIX отсутствует функция непосредственного копирования файлов. Вместо этого, чтобы выполнить команду cp, вы должны написать небольшую программу или использовать системный вызов system ().

В UNIX эквивалентом функции DeleteFile служит функция unlink, которая, к тому же, может удалять и каталоги.

В библиотеку C входят функции rename и remove, однако функция remove не позволяет присваивать файлу имя уже существующего файла или присваивать каталогу имя существующего непустого каталога. Новое имя может совпадать с именем существующего каталога только в том случае, если этот каталог пустой.

Управление каталогами

Создание и удаление каталогов осуществляется при помощи двух простых функций.

```
    BOOL    CreateDirectory(LPCTSTR    lpPathName,    LPSECURITY_ATTRIBUTES
lpSecurityAttributes)
    BOOL RemoveDirectory(LPCTSTR lpPathName)
```

lpPathName является указателем на завершающуюся нулевым символом строку, которая содержит путь к создаваемому или удаляемому каталогу. Как и в случае других функций, на данном этапе атрибуты защиты файла должны полагаться равными NULL; вопросы безопасности файлов и объектов рассматриваются в главе 15. Удалить можно только пустой каталог.

Как и в UNIX, у каждого процесса имеется текущий, или рабочий, каталог. Кроме того, для каждого диска поддерживается свой рабочий каталог. Программист может как устанавливать рабочий каталог, так и получать информацию о том, какой каталог в данный момент является текущим. Первая функция предназначена для установки каталогов.

```
    BOOL SetCurrentDirectory(LPCTSTR lpPathName)
```

lpPathName определяет путь к новому текущему каталогу. Это может быть относительный путь или абсолютный полный путь, в начале которого указаны либо буква диска и двоеточие (например, D:), либо имя UNC (например, \\ACCTG_SERVER\PUBLIC).

Если в качестве пути к каталогу указывается только имя диска (например, A: или C:), то рабочим каталогом становится рабочий каталог данного диска. Например, если рабочие каталоги устанавливались в последовательности:

```
C:\MSDEV
INCLUDE
A:\MEMOS\TODO
C:
```

то результирующим рабочим каталогом будет:

```
C:\MSDEV\INCLUDE
```

Следующая функция возвращает абсолютный полный путь к текущему каталогу, помещая его в буфер, предоставляемый программистом:

```
    DWORD GetCurrentDirectory(DWORD cchCurDir, LPTSTR lpCurDir)
```

Возвращаемое значение: длина строки, содержащей путь доступа к текущему каталогу, или требуемый размер буфера, если буфер не достаточно велик; в случае

ошибки — нуль.

cchCurDir — размер буфера, содержащего имя каталога, который определяется количеством символов (а не байт). Размер буфера должен рассчитываться с учетом завершающего нулевого символа строки. lpCurDir является указателем на буфер, предназначенный для получения строки, содержащей путь.

Заметьте, что в случае если размер буфера оказался недостаточным для того, чтобы в нем уместилась вся строка пути, функция возвратит значение, указывающее на требуемый размер буфера. Поэтому при тестировании успешности выполнения функции следует проверять два условия: равно ли возвращаемое значение нулю и не превышает ли оно значение, заданное аргументом cchCurDir.

Подобный метод возврата строк и их длины широко распространен в Windows и требует внимательной обработки результатов. Программа 2.6 иллюстрирует типичный фрагмент кода, реализующего эту логику. Аналогичная логика реализуется и в других примерах. Вместе с тем, указанный метод применяется не всегда. Некоторые функции возвращают булевские значения, а параметр размера в них используется дважды: перед вызовом функции его значение устанавливается равным размеру буфера, а затем изменяется функцией. В качестве одного из многих возможных примеров можно привести функцию LookupAccountName, с которой вы встретитесь в главе 15.

Альтернативный подход, демонстрируемый в программе 15.4 функцией GetFileSecurity, заключается в выделении буферной памяти в промежутке между двумя вызовами функций. Первый вызов обеспечивает получение длины строки, на основании чего и выделяется память, тогда как второй — получение самой строки. Самым простым подходом в данном случае является выделение памяти для строки, насчитывающей MAX_PATH символов.

Пример: печать текущего каталога

Программа 2.6 реализует очередную версию команды UNIX pwd. Размер буфера определяется значением параметра MAX_PATH, однако проверка ошибок все равно предусмотрена, чтобы проиллюстрировать работу функции GetCurrent-Directory.

Программа 2.6. pwd: печать текущего каталога

```
/* Глава 2. pwd - вывод на печать содержимого рабочего каталога. */
#include "EvryThng.h"
#define DIRNAME_LEN MAX_PATH + 2

int _tmain(int argc, LPTSTR argv[]) {
 TCHAR pwdBuffer [DIRNAME_LEN];
 DWORD LenCurDir;
 LenCurDir = GetCurrentDirectory(DIRNAME_LEN, pwdBuffer);
 if (LenCurDir == 0) ReportError(_T("Не удается получить путь."), 1, TRUE);
  if  (LenCurDir  >  DIRNAME_LEN)  ReportError(_T("Слишком  длинный  путь."),  2,
FALSE);
 PrintMsg(GetStdHandle(STD_OUTPUT_HANDLE), pwdBuffer);
 return 0;
}
```

Резюме

Наряду с функциями, предназначенными для обработки символов, в Windows поддерживается полный набор функций, обеспечивающих управление файлами и каталогами. Кроме того, вы можете создавать переносимые, обобщенные приложения, которые могут быть рассчитаны на работу как с символами ASCII, так и с символами Unicode.

Функции Windows во многом напоминают их аналоги в UNIX и библиотеке C, хотя различия между ними также очевидны. В приложении Б представлена таблица, в которой сведены функции Windows, UNIX и библиотеки C и показано, в чем они соответствуют друг другу, а в чем заметно отличаются.

В следующих главах

Нашим следующим шагом будет обсуждение в главе 3 прямого доступа к файлам и использования таких атрибутов файлов и каталогов, как размер файла и метки времени. Кроме того, в главе 3 показано, как управлять каталогами, а в завершение главы обсуждается работа с API реестра, аналогичного API управления каталогами.

Дополнительная литература

Организация хранения данных в Windows и NTFS

В книге [22] содержится исчерпывающее обсуждение всего спектра возможных вариантов организации хранения данных в Windows как на непосредственно подключенных, так и на сетевых устройствах. Наряду с внутренними деталями реализации описаны все последние достижения и успехи в данной области, а также прогресс в отношении повышения быстродействия устройств хранения данных.

Книга [10] — это небольшая монография, в которой описаны цели и особенности реализации NTFS. Содержащаяся в ней информация пригодится вам как для этой, так и для следующей глав.

Unicode

В книге [19] показано, как использовать Unicode на практике. Изложение сопровождается различными рекомендациями, а также рассмотрением международных стандартов и вопросов программирования, связанных с учетом региональных особенностей.

На домашней странице компании Microsoft вы найдете несколько полезных статей о стандарте Unicode. Основной является статья "Unicode Support in Win32" ("Поддержка Unicode в Win32"), отталкиваясь от которой вы, используя средства поиска, сможете отыскать все остальные.

UNIX

В главах 3 и 4 книги [40] рассматриваются файлы и каталоги UNIX, а в главе 11 — терминальный ввод/вывод.

Для быстрого ознакомления с командами UNIX можете обратиться к книге [15].

Упражнения

2.1. Напишите небольшую программу для тестирования обобщенных версий функций printf и scanf.

2.2. Модифицируйте функцию CatFile в программе 2.3 таким образом, чтобы при связывании дескриптора стандартного вывода с консолью в ней использовалась не функция WriteFile, а функция WriteConsole.

2.3. Параметры вызова функции CreateFile позволяют задавать различные характеристики способа доступа к файлу, что может быть использовано для повышения производительности программ. В качестве примера можно привести параметр FILE_FLAG_SEQUENTIAL_SCAN. Используйте этот флаг в программе 2.5 и выясните, приведет ли это к улучшению показателей производительности при работе с файлами большого размера. Результаты для нескольких систем приведены в приложении B. Исследуйте также влияние флага FILE_FLAG_NO_BUFFERING.

2.4. Исследуйте, насколько ощутимы различия в производительности для файловых систем FAT и NTFS при использовании функции atou в случае преобразования файлов большого размера.

2.5. Выполните программу 2.4 с использованием и без использования определения символической константы UNICODE. Как это влияет на результаты, если таковое влияние вообще наблюдается? Если имеется такая возможность, выясните, способны ли программы правильно выполняться в системах Windows 9x.

2.6. Сопоставьте информацию, предоставляемую функциями perror (библиотека C) и ReportError в случае таких распространенных ошибок, как попытка открытия несуществующего файла.

2.7. Протестируйте подавление функцией ConsolePrompt (программа 2.1) эхо-отображения клавиатурного ввода, используя ее для вывода запроса на ввод и подтверждение пароля пользователем.

2.8. Выясните, что происходит, когда для вывода на консоль используются смешанные вызовы функций обобщенной библиотеки C и функций Windows WriteFile и WriteConsole. Дайте происходящему свое объяснение.

2.9. Напишите программу сортировки массива строк Unicode. Изучите различия между случаями сортировки слов и строк с помощью функций lstrcmp и _tcscmp. Приводит ли использование функции lstrlen к получению иных результатов по сравнению с функцией _tcslen? Вам могут пригодиться содержащиеся в оперативной справочной системе Microsoft замечания в описании функции CompareString.

2.10. Расширьте реализацию функции Options таким образом, чтобы она выводила сообщение об ошибке, если в командной строке указаны опции, которые отсутствуют в списке разрешенных опций, заданных в параметре OptionString данной функции.

2.11. В приложении B приводятся данные о показателях производительности для копирования файлов и их преобразования при помощи функции atou с использованием различных вариантов реализации программы и файловых систем. Исследуйте с помощью тестовых программ показатели производительности на доступных вам системах. Кроме того, если возможно, исследуйте показатели производительности для сетевых файловых систем, SAN и так далее, чтобы выяснить, каким образом проявляются различия в организации хранения данных при осуществлении последовательного доступа к файлам.

ГЛАВА 3
Усовершенствованные средства для работы с файлами и каталогами и знакомство с реестром

Файловые системы обеспечивают не только простую последовательную обработку файлов; кроме этого, они должны предоставлять возможности прямого доступа к файлам и блокирования файлов, а также предлагать средства для управления каталогами и атрибутами файлов. В данной главе, которая начинается с обсуждения прямого доступа к файлам, требуемого для обслуживания баз данных, обработки файлов и решения целого ряда других задач, демонстрируются методы непосредственного доступа к данным, находящимся в произвольном месте файла, которые обеспечиваются файловыми указателями. Для этого, в частности, нам надо будет обсудить использование 64-битовых указателей Windows, поскольку файловая система NTFS способна поддерживать файлы гигантских размеров.

Далее будут рассмотрены методы просмотра каталогов, рассказано о том, что такое атрибуты файлов, такие, например, как метки времени, атрибуты прав доступа или размер файла, и показано, как управлять атрибутами и интерпретировать их. Наконец, вы ознакомитесь с тем, как использовать блокирование файлов с целью предотвращения попыток изменения их содержимого одновременно несколькими процессами.

Завершает данную главу рассмотрение реестра Windows — централизованной базы данных, хранящей информация о конфигурации системы, которой могут пользоваться как приложения, так и сама операционная система. Приведенный в конце главы пример программы показывает, что функции, с помощью которых осуществляется доступ к реестру, и структура соответствующих программ напоминают те, которые применяются для управления файлами и каталогами, что и послужило причиной включения этой темы в данную главу.

64-битовая файловая система

Win32 и Win64, работающие с NTFS, поддерживают 64-битовую адресацию в файлах, и поэтому допустимыми являются файлы размером до 2*64* байт.

В 32-разрядных файловых системах, характеризующихся наличием 2*32* –байтового предела, допустимый размер файлов ограничивается величиной 4 Гбайт (4x10*9* байт). Для некоторых приложений, включая крупные базы данных и мультимедийные системы, это ограничение носит серьезный характер, что вынуждает современные ОС обеспечивать поддержку файлов гораздо больших размеров. Файлы, размеры которых превышают 4 Гбайт, иногда называют *гигантскими* (huge).

Вполне очевидно, что многим приложения гигантские файлы никогда не понадобятся, так что большинству программистов на протяжении ближайших нескольких лет возможностей 32-битовой файловой адресации будет вполне достаточно. Однако, с учетом темпов технической модернизации и увеличения емкости дисков[14], улучшения их стоимостных показателей и повышения уровня требований со стороны приложений, целесообразно уже с самого начала работы над новым проектом предусмотреть возможность использования 64-битовых адресов.

Несмотря на возможность использования 64-битовой адресации файлов и поддержку гигантских файлов, интерфейс Win32, в силу его привязки к 32-битовой адресации памяти, о чем говорится в главе 5, остается API 32-битовой ОС, так что для работы с 64-битовыми адресами памяти нам потребуется интерфейс Win64.

Указатели файлов

В Windows аналогично тому, как это предусмотрено в UNIX, библиотеке C и почти любой другой ОС, для каждого дескриптора открытого файла поддерживается *указатель файла* (file pointer), отмечающий позицию текущего байта в данном файле. Именно эта позиция служит отправной точкой для последующей передачи данных при выполнении очередной операции WriteFile или ReadFile, что сопровождается увеличением значения указателя файла на соответствующее количество переданных байт. При открытии файла путем вызова функции CreateFile указатель файла принимает нулевое значение, отмечающее начало файла, которое изменяется по мере чтения или записи каждого очередного байта. Ключевую роль в обеспечении возможности прямого доступа к данным, хранящимся в файле, играет функция SetFilePointer, позволяющая устанавливать значения указателя файла.

Функция SetFilePointer является первой из функций, на примере которых мы познакомимся с обработкой 64-битовых указателей файлов NTFS. Методы, основанные на этой функции, не всегда удобны в применении, и поэтому функцию SetFilePointer проще всего использовать в случае небольших файлов.

```
DWORD SetFilePointer(HANDLE hFile, LONG lDistanceToMove, PLONG
lpDistanceToMoveHigh, DWORD dwMoveMethod)
```

Возвращаемое значение: младшее двойное слово (DWORD, беззнаковое) нового значения указателя файла. Старшая часть значения этого указателя помещается в двойное слово, на которое указывает указатель lpDistanceToMoveHigh (если он отличен от NULL). В случае неудачного завершения функция возвращает значение 0xFFFFFFFF.

Параметры

hFile — дескриптор файла, который должен быть создан с правами доступа по чтению или по записи (или с правами доступа одновременно обоих типов).

lDistanceToMove — 32-битовое число типа LONG *со знаком,* указывающее величину смещения, на которое должен быть перемещен указатель файла, или число типа LONG *без знака,* указывающее позицию, в которую должен быть перемещен указатель файла, в зависимости от значения параметра dwMoveMethod.

lpDistanceToMoveHigh — указатель на старшую часть 64-битового смещения, на которое должен быть перемещен указатель файла. Если значение этого параметра задано равным NULL, то функция может применяться только к файлам, размер которых не превышает $2^{32}–2$ (в байтах). Этот же параметр используется для получения старшей части возвращаемого функцией значения указателя файла.[15] Младшую часть указателя файла возвращает сама функция.

dwMoveMethod — этот параметр устанавливает один из трех возможных режимов перемещения указателя файла.

• FILE_BEGIN — указатель файла позиционируется относительно начала файла, причем параметр DistanceToMove интерпретируется как беззнаковое число.

• FILE_CURRENT — указатель файла перемещается в сторону больших или меньших значений относительно текущей позиции, причем параметр DistanceToMove интерпретируется как число со знаком. Положительным значениям соответствует перемещение указателя файла в сторону больших значений.

• FILE_END — указатель файла перемещается в сторону больших или меньших значений

относительно позиции конца файла.

Эту функцию можно использовать для получения размера файла, задав нулевое смещение указателя от позиции конца файла.

Описанный метод представления 64-битовых указателей файлов становится причиной некоторых затруднений, поскольку возвращенное функцией значение может представлять как действительную позицию указателя файла, так и код ошибки. Рассмотрите, например, случай, когда фактической позиции указателя соответствует значение $2^{32}-1$ (то есть, 0xFFFFFFFF), а при вызове функции указывается ненулевое значение старшей части перемещения указателя файла. Чтобы определить, представляет ли значение, возвращенное функцией SetFile-Pointer, действительную позицию указателя файла или же код ошибки, следует вызвать функцию GetLastError, возвращаемым значением которой в случае неудачного завершения не может быть NO_ERROR. Из этих рассуждений становится ясно, почему размеры файлов не могут превышать значения $2^{32}-2$, если при вызове функции SetFilePointer старшая часть указателя файла опускается.

Дополнительную неразбериху привносит тот факт, что старшая и младшая компоненты указателя файла отделены друг от друга и обрабатываются по-разному. Младшая часть определяется через передачу параметра по значению и равна возвращаемому значению функции, тогда как для старшей части применяется передача параметра по ссылке, и этот параметр используется как в качестве входного, так и выходного.

К счастью, 32-битовой адресации вам будет вполне достаточно для большинства задач программирования. Тем не менее, приведенные в книге примеры программ рассчитаны на далекую перспективу и используют, "как и положено", 64-битовую арифметику.

64-битовая арифметика

Арифметика 64-битовых указателей файлов не так уж сложна, и для ее реализации в примерах программ используется принятый в Microsoft C 64-битовый тип данных LARGE_INTEGER, объединяющий в одном типе данных union величину типа LONGLONG (носящую название QuadPart) и две 32-битовые величины (LowPart типа DWORD и, HighPart типа LONG). Тип данных LONGLONG поддерживает все арифметические операции. Существует также соответствующий тип данных без знака ULONGLONG.

> Аналогами функции SetFilePointer являются функции lseek (UNIX) и fseek (библиотека C). В обеих упомянутых системах выполнение операций чтения или записи также сопровождается перемещением указателя файла.

Указание позиции файла с помощью структуры OVERLAPPED

Для указания позиции в файле Windows предоставляет еще один способ, не требующий использования функции SetFilePointer. Вспомните, что последним параметром в обеих функциях ReadFile и WriteFile является адрес структуры перекрытия OVERLAPPED, который в предыдущих примерах всегда полагался равным NULL. В структуру перекрытия входят элементы Offset и OffsetHigh. Устанавливая соответствующие значения элементов структуры OVERLAPPED, вы можете добиться того, чтобы выполнение операций ввода/вывода начиналось с указанной позиции. В отличие от указателя файла, значение которого изменяется, соответствуя позиции, следующей за последним переданным байтом, значения элементов структуры OVERLAPPED остаются неизменными. Элементом этой структуры является также

дескриптор hEvent, значение которого должно устанавливаться равным NULL.

Примечание

Под управлением Windows 9x описанный метод работать не будет, поскольку в этом случае указатель структуры OVERLAPPED при обработке файлов должен устанавливаться равным NULL.

Предостережение

Хотя в рассмотренном примере и используется структура OVERLAPPED, здесь не идет речь о перекрывающемся вводе/выводе, который обсуждается в главе 14.

Использование структуры OVERLAPPED оказывается особенно удобным в тех случаях, когда требуется обновить запись в файле, что иллюстрирует приведенный ниже фрагмент программного кода; в противном случае вы должны были бы перед каждым вызовом функций ReadFile и WriteFile отдельно вызывать функцию SetFilePointer. Последним из пяти полей структуры OVERLAPPED является поле hEvent, как это видно из оператора инициализации. Для хранения вычисленного значения позиции в файле используется переменная FilePos типа LARGE_INTEGER.

```
    OVERLAPPED ov = { 0, 0, 0, 0, NULL };
    RECORD r; /* Хотя определение этой структуры не приведено, в ней имеется поле
RefCount. */
    LONGLONG n;
    LARGE_INTEGER FilePos;
    DWORD nRead, nWrite;
    …
    /* Обновить счетчик, чтобы он соответствовал n-й записи. */
    FilePos.QuadPart = n * sizeof(RECORD);
    ov.Offset = FilePos.LowPart;
    ov.OffsetHigh = FilePos.HighPart;
    ReadFile(hFile, r, sizeof(RECORD), &nRead, &ov);
    r.RefCount++; /* Обновить запись. */
    WriteFile(hFile, r, sizeof(RECORD), &nWrite, &ov);
```

Если дескриптор файла был создан за счет вызова функции CreateFile с установленным флагом FILE_FLAG_NO_BUFFERING, то как смещение позиции в файле, так и размер записи (количество байт) должны быть кратными размеру сектора диска. Соответствующую информацию относительно физического диска, включая информацию о размере сектора, возвращает функция GetDiskFreeSpace.

Структуры OVERLAPPED будут вновь использованы далее в этой главе для указания областей блокирования файлов и в главе 14 для выполнения операций асинхронного ввода/вывода и прямого доступа к файлам.

Определение размера файла

Размер файла можно получить, используя значение указателя файла, возвращаемое функцией SetFilePointer, если при вызове этой функции задать количество байтов, на которое должен быть перемещен указатель файла, равным 0. Для этой же цели можно воспользоваться также функцией GetFileSize.

```
DWORD GetFileSize(HANDLE hFile, LPDWORD lpFileSizeHigh)
```

Возвращаемое значение: младшая компонента размера файла. Значение 0xFFFFFFFF указывает на возможную ошибку; для проверки наличия ошибок следует использовать функцию GetLastError.

Обратите внимание, что для возвращения размера файла используется, по сути, тот же способ, что и для возвращения фактического указателя файла функцией SetFilePointer.

Функции GetFileSize и GetFileSizeEx (возвращающая 64-битовое значение размера файла в одном элементе данных) требуют указания дескриптора, открытого для файла. Для определения размера файла можно применять также имя файла. Функция GetCompressedFileSize возвращает размер сжатого файла, тогда как функция FindFirstFile, которая обсуждается в разделе "Атрибуты файлов и управление каталогами" далее в этой главе, предоставляет точный размер именованного файла.

Установка размера файла, инициализация файла и разреженные файлы

Функция SetEndOfFile позволяет переустановить размер файла, используя текущее значение указателя файла для определения его размера. Возможно как расширение, так и усечение файла. В случае расширения файла содержимое области расширения не определено. Файл будет фактически потреблять выделенные квоты дискового и пользовательского пространств, если только не является разреженным. Файлы можно сжимать с целью уменьшения объема занимаемого ими пространства. Этот вопрос исследуется в упражнении 3.1.

Функция SetEndOfFile устанавливает физический конец файла. Прежде чем выполнять эту операцию, на которую может уйти довольно длительное время, необходимое для записи данных файл с целью его заполнения, можно установить также логический конец файла, используя для этого функцию SetValidFileData. Эта функция определяет ту часть файла, которая, в соответствии с вашими предположениями, в настоящий момент содержит достоверные данные, благодаря чему вы сможете сэкономить время при установке физического конца файла. Часть файла, заключенная между его логическим и физическим концами, называется *хвостовиком* (tail) и может быть сокращена путем записи оставшихся данных после логического конца файла или в результате дополнительного вызова функции SetValidFileData.

В случае разреженных файлов (sparse files), появившихся в Windows 2000, дисковое пространство расходуется лишь по мере записи данных. Администратор может назначать, какие файлы, каталоги или тома должны быть разреженными. Кроме того, можно назначить существующий файл в качестве разреженного с помощью функции DeviceIoControl, если установить при ее вызове флаг FSCTL_SET_SPARSE. Ситуацию, в которой удобно использовать разреженные файлы, иллюстрирует программа 3.1. К разреженным файлам функция SetValidFileData неприменима.

Файлы FAT нулями автоматически *не* инициализируются. Согласно документации Microsoft содержимое вновь созданных файлов не определено, что подтверждается

экспериментами. Поэтому, если для корректной работы требуется инициализация файлов, приложения должны это делать самостоятельно путем вызова функции WriteFile. Файлы NTFS будут инициализированы, поскольку уровень безопасности C2, обеспечиваемый Windows, требует, чтобы чтение содержимого удаленных файлов было невозможным.

Обратите внимание, что кроме функции SetEndOfFile существуют и другие способы расширения размера файла. Так, можно расширить файл, используя ряд последовательных операций записи, хотя при этом существует риск увеличения степени фрагментации файла; размещение на диске файлов в виде непрерывных блоков большого размера функция SetEndOfFile отдает на откуп операционной системе.

Пример: обновление записей, находящихся в произвольном месте файла

Программа RecordAccess (программа 3.1) обеспечивает поддержку файлов фиксированного размера, состоящих из записей фиксированного размера. В заголовке файла хранится количество непустых записей, содержащихся в файле, а также емкость файла. Пользователю предоставляется возможность выполнять в интерактивном режиме чтение, запись (обновление) и удаление записей, каждая из которых содержит метки времени, текстовую строку и счетчик, показывающий, сколько раз запись изменялась. В качестве несложного и реалистичного расширения возможностей программы можно было бы добавить в структуру записи ключ и определять местоположение записей в файле путем применения хэш-функции к значениям ключа.

Программа демонстрирует позиционирование указателя файла перед заданной записью, а также выполнение 64-битовых арифметических операций с использованием данных типа LARGE_INTEGER Microsoft C. Чтобы проиллюстрировать логику работы указателей файла, в программу включен код, проверяющий наличие ошибок. Программа в целом иллюстрирует применение файловых указателей и множественных структур OVERLAPPED, а также обновление файлов с использованием 64-битовых файловых указателей.

Общее количество записей в файле указывается в командной строке; при большом количестве записей размеры создаваемых файлов могут быть очень большими и даже гигантскими, поскольку длина одной записи составляет примерно 300 байт. После выполнения нескольких экспериментов вы убедитесь, что большие файлы должны быть разреженным; в противном случае необходимо размещать и инициализировать на диске весь файл целиком, в результате чего может существенно увеличиться время обработки файла и занимаемое им место на диске. Хотя в листинге программы 3.1 это и не отражено, в программе предусмотрен участок кода, обеспечивающий создание разреженных файлов, если в этом возникает необходимость; в некоторых системах, например Windows XP Home, этот код правильно работать не сможет.

На Web-сайте книги предоставляются три дополнительные программы, родственные этой: tail.c — другой пример реализации произвольного доступа к файлу, getn.c — упрощенная версия программы RecordAccess, обеспечивающая лишь чтение записей, и atouMT (включена в программы для главы 14, находящиеся на Web-сайте, однако не включена в программы, приведенные в книге), также иллюстрирующая прямой доступ к файлам.

Программа 3.1. RecordAccess

```
    /* Глава 3. RecordAccess. */
    /* Использование: RecordAccess имя файла [количество записей]
    Количество записей (nrec) можно не указывать, если файл с указанным именем уже существует. Если количество записей (nrec) задано, создается файл с указанным именем (если файл с таким именем существует, он уничтожается). При большом количестве записей (nrec) файлы рекомендуется создавать как разреженные. */
    /* Программа иллюстрирует:
    1. Произвольный доступ к файлам.
    2. Арифметику данных типа LARGE_INTEGER и использование 64-битовых указателей файла.
    3. Обновление записей на месте.
    4. Запись в файл нулей во время инициализации (требует использования файловой системы NTFS).
```

```
   */

   #include "EvryThng.h"
   #define STRING_SIZE 256
   typedef struct _RECORD { /* Структура записи в файле */
    DWORD ReferenceCount; /* 0 означает пустую запись. */
    SYSTEMTIME RecordCreationTime;
    SYSTEMTIME RecordLastReferenceTime;
    SYSTEMTIME RecordUpdateTime;
    TCHAR DataString[STRING_SIZE];
   } RECORD;
   typedef struct _HEADER { /* Дескриптор заголовка файла */
    DWORD NumRecords;
    DWORD NumNonEmptyRecords;
   } HEADER;

   int _tmain(int argc, LPTSTR argv[]) {
    HANDLE hFile;
    LARGE_INTEGER CurPtr;
    DWORD FPos, OpenOption, nXfer, RecNo;
    RECORD Record;
    TCHAR String[STRING_SIZE], Command, Extra;
    OVERLAPPED ov = {0, 0, 0, 0, NULL}, ovZero = {0, 0, 0, 0, NULL};
    HEADER Header = {0, 0};
    SYSTEMTIME CurrentTime;
    BOOLEAN HeaderChange, RecordChange;
    OpenOption = (argc == 2) ? OPEN_EXISTING : CREATE_ALWAYS;
    hFile = CreateFile(argv[1], GENERIC_READ | GENERIC_WRITE, 0, NULL, OpenOption,
FILE_ATTRIBUTE_NORMAL, NULL);
     if (argc >= 3) { /* Записать заголовок и заранее установить размер нового
файла */
      Header.NumRecords = atoi(argv[2]);
      WriteFile(hFile, &Header, sizeof(Header), &nXfer, &ovZero);
      CurPtr.QuadPart = sizeof(RECORD)*atoi(argv[2])+sizeof(HEADER);
      FPos = SetFilePointer(hFile, CurPtr.LowPart, &CurPtr.HighPart, FILE_BEGIN);
       if (FPos == 0xFFFFFFFF && GetLastError() != NO_ERROR) ReportError(_T("Ошибка
указателя."), 4, TRUE);
      SetEndOfFile(hFile);
     }
      /* Считать заголовок файла: определить количество записей и количество
непустых записей. */
     ReadFile(hFile, &Header, sizeof(HEADER), &nXfer, &ovZero);
     /* Предложить пользователю считать или записать запись с определенным номером.
*/
     while(TRUE) {
      HeaderChange = FALSE;
      RecordChange = FALSE;
      _tprintf(_T("Введите r(ead)/w(rite)/d(elete)/q Запись#\n"));
      _tscanf(_T("%c" "%d" "%c"), &Command, &RecNo, &Extra );
      if (Command == 'q') break;
      CurPtr.QuadPart = RecNo * sizeof(RECORD) + sizeof(HEADER);
      ov.Offset = CurPtr.LowPart;
      ov.OffsetHigh = CurPtr.HighPart;
      ReadFile(hFile, &Record, sizeof(RECORD), &nXfer, &ov);
      GetSystemTime(&CurrentTime); /* Обновить поля даты и времени в записи. */
      Record.RecordLastRefernceTime = CurrentTime;
      if (Command == 'r' || Command == 'd') { /*Вывести содержимое записи.*/
       if (Record.ReferenceCount == 0) {
        _tprintf(_T("Запись номер %d - пустая.\n"), RecNo);
```

```
        continue;
       } else {
            _tprintf(_T("Запись номер %d. Значение счетчика: %d \n"), RecNo,
Record.ReferenceCount);
        _tprintf(_T("Данные: %s\n"), Record.DataString);
        /* Упражнение: вывести метки времени. См. следующий пример. */
        RecordChange = TRUE;
       }
       if (Command == 'd') { /* Удалить запись. */
        Record.ReferenceCount = 0;
        Header.NumNonEmptyRecords--;
        HeaderChange = TRUE;
        RecordChange = TRUE;
       }
      } else if (Command == 'w') { /* Записать данные. Впервые? */
       _tprintf(_T("Введите новую строку для записи.\n"));
       _getts(String);
       if (Record.ReferenceCount == 0) {
        Record.RecordCreationTime = CurrentTime;
        Header.NumNonEmptyRecords++;
        HeaderChange = TRUE;
       }
       Record.RecordUpdateTime = CurrentTime;
       Record.ReferenceCount++;
       _tcsncpy(Record.DataString, String, STRING_SIZE-1);
       RecordChange = TRUE;
      } else {
       _tprintf(_T("Допустимые команды: r, w и d. Повторите ввод.\n"));
      }
      /* Обновить запись на месте, если ее содержимое изменилось. */
      if (RecordChange) WriteFile(hFile, &Record, sizeof(RECORD), &nXfer, &ov);
      /* При необходимости обновить количество непустых записей. */
      if (HeaderChange) WriteFile(hFile, &Header, sizeof(Header), &nXfer, &ovZero);
     }
        _tprintf(_T("Вычисленное количество непустых записей: %d\n"),
Header.NumNonEmptyRecords);
     CloseHandle(hFile);
     return 0;
    }
```

Атрибуты файлов и управление каталогами

Существует возможность просмотра указанного каталога с целью поиска файлов и других каталогов, имена которых соответствуют заданному шаблону, одновременно с получением атрибутов файлов. Для выполнения поиска требуется дескриптор поиска (search handle), получаемый с помощью функции FindFirstFile. Для нахождения файлов, имена которых удовлетворяют заданным условиям, используется функция FindNextFile, а для прекращения поиска — функция FindClose.

```
HANDLE FindFirstFile(LPCTSTR lpFileName, LPWIN32_FIND_DATA lpffd)
```

Возвращаемое значение: дескриптор поиска. Значение INVALID_HANDLE_VALUE указывает на неудачное завершение функции.

В процессе поиска имен, соответствующих искомому, функция FindFirstFile проверяет имена не только файлов, но и подкаталогов. Возвращенное функцией значение дескриптора типа HANDLE используется для продолжения поиска.

Параметры

lpFileName — указатель на строку, содержащую имя каталога или полное имя файла, при указании которых можно использовать метасимволы (? и *). Если необходимо осуществить поиск конкретного файла, метасимволы опускаются.

lpffd — указатель на структуру WIN32_FIND_DATA, которая принимает информацию о первом найденном файле или каталоге, который удовлетворяет критерию поиска, если таковой был найден.

Структура WIN32_FIND_DATA определяется следующим образом:

```
typedef struct_WIN32_FIND_DATA {
 DWORD dwFileAttributes;
 FILETIME ftCreationTime;
 FILETIME ftLastAccessTime;
 FILETIME ftLastWriteTime;
 DWORD nFileSizeHigh;
 DWORD nFileSizeLow;
 DWORD dwReserved0;
 DWORD dwReserved1;
 TCHAR cFileName[MAX_PATH];
 TCHAR cAlternateFileName[14];
 } WIN32_FIND_DATA;
```

Параметр dwFileAttributes можно тестировать на присутствие значений, описанных при рассмотрении функции CreateFile, а также некоторых других значений, например, FILE_ATTRIBUTE_SPARSE_FILE или FILE_ATTRIBUTE_ENCRYTPED, которые не устанавливаются функцией CreateFile. Описание меток времени трех типов (время создания, время последнего обращения и время последнего изменения) приведено в одном из следующих разделов. Названия полей размера файла (nFileSizeHigh и nFileSizeLow) говорят сами за себя. cFileName — это не полное имя файла, содержащее путь доступа, а само имя файла. cAlternateFileName — имя файла в формате DOS 8.3 (включая точку); эта информация редко

используется и может понадобиться лишь для того, чтобы определить, каким будет имя файла в файловой системе FAT16.

Во многих случаях требуется просматривать каталог с целью поиска файлов, имена которых соответствуют некоторому шаблону, содержащему метасимволы ? и *. Для этого следует использовать дескриптор поиска, полученный из функции FindFirstFile, в котором содержится информация об искомом имени, и вызвать функцию FindNextFile.

```
BOOL FindNextFile(HANDLE hFindFile, LPWIN32_FIND_DATA lpffd)
```

Функция FindNextFile возвращает значение FALSE, если аргументы недействительны или если не удается найти файл, удовлетворяющий критерию поиска, причем последнему случаю соответствует возвращаемое значение функции GetLastError, равное ERROR_NO_MORE_FILES.

После того как поиск завершен, дескриптор поиска должен быть закрыт. Функцию CloseHandle для этой цели использовать нельзя. Это редкий пример нарушения правила, согласно которому функция CloseHandle применима к дескрипторам любого типа; в данном случае закрытие дескриптора поиска подобным способом приведет к генерации исключения. Вместо этого необходимо использовать следующую функцию:

```
BOOL FindClose(HANDLE hFindFile)
```

Функция GetFileInformationByHandle позволяет получить информацию о конкретном файле, на который указывает открытый дескриптор файла. Она также возвращает поле nNumberOfLinks, в котором содержится количество жестких ссылок на файл, установленных функцией CreateHardLink.

Описанный метод расширения метасимволов необходим даже в программах, запускаемых на выполнение из командной строки DOS, поскольку оболочка DOS не расширяет метасимволы.

Полные имена файлов

Полное имя файла можно получить, используя функцию GetFullPathName. Функция GetShortPathName возвращает имя файла в формате DOS 8.3, в предположении, что данный том поддерживает короткие имена файлов.

В NT 5.1 была введена функция SetFileShortName, позволяющая изменить существующее сокращенное имя файла или каталога. Иногда это оказывается удобным, поскольку интерпретация сокращенных имен файлов часто вызывает затруднения.

Другие методы определения атрибутов файлов и каталогов

Функции FindFirstFile и FindNextFile позволяют получить следующую информацию, связанную с атрибутами файла: флаги атрибутов, метки времени трех типов и размер файла. Существуют также другие аналогичные функции, одна из которых предназначена для задания атрибутов, причем эти функции могут работать непосредственно с открытыми дескрипторами файлов, не требуя просмотра каталогов или указания имен файлов. Три из этих функций, а именно, GetFileSize, GetFileSizeEx и SetEndOfFile, были описаны ранее в этой главе.

Для получения других атрибутов используются отдельные функции. Например, чтобы получить метки времени открытого файла, следует вызвать функцию GetFileTime.

```
BOOL GetFileTime(HANDLE hFile, LPFILETIME lpftCreation, LPFILETIME
lpftLastAccess, LPFILETIME lpftLastWrite)
```

Указанные здесь и в структуре WIN32_FIND_DATA метки времени представляют собой 64-битовые целые числа без знака, которые выражают величину временного интервала, вычисленную относительно условного начала отсчета (1 января 1601 года) и преобразованную во время UTC (Universal Coordinated Time — всеобщее скоординированное время)[16], в 100-наносекундных единицах времени (10^7 единиц в 1 секунде). Для работы с этими временными параметрами предусмотрено несколько удобных функций.

• Функция FileTimeToSystemTime (здесь не описывается; см. справочную систему Windows и программу 3.2) разбивает метки времени файла на отдельные блоки, соответствующие естественным единицам измерения, от годов до секунд и миллисекунд. Эти блоки удобно, например, использовать при выводе временных атрибутов файлов на экран или принтер.

• Функция SystemTimeToFileTime обращает этот процесс, преобразуя время, выраженное в естественных единицах, в метки времени файла.

• Функция CompareFileTime сравнивает метки времени двух файлов и в случае успешного завершения возвращает значение, зависящее от того, меньше (-1), равно (0) или больше (+1) значение метки времени первого файла по сравнению со значением метки времени второго файла.

• Для изменения меток времени служит функция SetFileTime; метки времени, не подлежащие изменению, при вызове функции указываются равными 0. NTFS поддерживает все три типа меток времени файлов, но FAT дает точные результаты только для меток времени последнего обращения.

• Функции FileTimeToLocalFileTime и LocalFileTimeToFileTime преобразуют значения меток времени, соответственно, от всеобщего скоординированного времени UTC к местному времени и наоборот.

Функция GetFileType, которая здесь подробно не описывается, позволяет различать файлы трех типов: дисковые, символьные (к ним, по сути, относятся такие устройства, как принтеры и консоли) и каналы (см. главу 11). Как и в большинстве других случаев, файл, характеристику которого необходимо определить, задается дескриптором.

Функция GetFileAttributes принимает в качестве аргумента имя файла или каталога, а всю информацию об атрибутах передает через свое возвращаемое значение dwFileAttributes.

```
DWORD GetFileAttributes(LPCTSTR lpFileName)
```

Возвращаемое значение: в случае успешного завершения — атрибуты файла, иначе — 0xFFFFFFFF.

Для определения атрибутов можно воспользоваться логическим сравнением возвращаемого значения функции с соответствующими масками значений атрибутов. Некоторые атрибуты, например атрибут временного файла, изначально устанавливаются функцией CreateFile. В качестве примера можно привести следующие атрибуты:

• FILE_ATTRIBUTE_DIRECTORY
• FILE_ATTRIBUTE_NORMAL
• FILE_ATTRIBUTE_READONLY
• FILE_ATTRIBUTE_TEMPORARY

Для изменения атрибутов именованных файлов служит функция SetFileAttributes.

В UNIX трем вышеописанным функциям Find соответствуют функции opendir,

readdir и closedir. Функция stat предоставляет размер файла и значения меток времени, а также информацию о его индивидуальном или групповом владельце, необходимую для защиты файлов в UNIX. Разновидностями этой функции являются функции fstat и lstat. Эти функции позволяют также получать информацию о типе файла. Метки времени файла в UNIX устанавливаются с помощью функции utime. Эквивалента атрибута временного файла в UNIX не существует.

Именование временных файлов

Следующая функция создает имена для временных файлов. Файл может находиться в любом заданном каталоге, и его имя должно быть уникальным.

Функция GetTempFileName предоставляет уникальное имя файла с расширением .tmp, используя указанный путь доступа, и при необходимости создает файл. Эта функция широко используется в ряде следующих примеров (программа 6.1, программа 7.1 и другие).

```
UINT GetTempFileName(LPCTSTR lpPathName, LPCTSTR lpPrefixString, UINT uUnique, LPTSTR lpTempFileName)
```

Возвращаемое значение: уникальное числовое значение, используемое для создания имени файла. Этим значением будет значение параметра uUnique, если при вызове функции оно было задано ненулевым. В случае неудачного завершения функции возвращаемое значение равно нулю.

Параметры

lpPathName — каталог, в котором размещается временный файл. Типичным значением этого параметра является строка ".", указывающая на текущий каталог. В других случаях можно воспользоваться функцией Windows GetTempPath, которая предоставляет имя каталога, используемого для хранения временных файлов, но нами здесь не рассматривается.

lpPrefixString — префикс, используемый в имени временного файла. Допускаются лишь 8-битовые символы ASCII. Значение параметра uUnique обычно устанавливается равным нулю, чтобы функция самостоятельно сгенерировала уникальный четырехразрядный префикс и использовала его в имени создаваемого файла. При ненулевом значении этого параметра файл не создается, так что это необходимо сделать отдельно при помощи функции CreateFile, возможно — с использованием флага FILE_FLAG_DELETE_ON_CLOSE.

lpTempFileName — указатель на буфер, предназначенный для хранения имени временного файла. Размер буфера, выраженный в байтах, должен быть не менее MAXPATH. Результирующее полное имя файла получается объединением строк, соответствующих пути доступа к файлу, префикса, четырехразрядного шестнадцатеричного числа и суффикса .tmp.

Точки монтирования

NT 5.0 разрешает монтирование (или подключение) одной файловой системы в точке монтирования, находящейся в другой файловой системе. Обычно управление точками монтирования является прерогативой администратора системы, но эти же задачи можно решать и программным путем.

Функция SetVolumeMountPoint монтирует диск (второй аргумент) в точке монтирования, указанной первым аргументом. Например, вызов

```
SetVolumeMountPoint("C:\\mycd\\, "D:\\");
```

монтирует диск D: (которому в персональных системах часто соответствует привод компакт-диска) в каталоге mycd (точка монтирования), находящемуся на диске C:. Обратите внимание на то, что обозначения всех путей доступа заканчиваются символами обратной косой черты. Тогда после применения этой функции пути доступа C:\mycd\memos\book.doc будет соответствовать путь доступа D:\memos\book.doc.

Одну и ту же точку монтирования можно использовать для подключения нескольких файловых систем. Для размонтирования файловых систем служит функция DeleteMountPoint.

Функция GetVolumePathName возвращает корневую точку монтирования абсолютного или относительного пути доступа или имени файла. В свою очередь, функция GetVolumeNameForVolumeMountPoint предоставляет имя тома, например, C:\, соответствующего точке монтирования.

Пример: вывод списка атрибутов файла

Настало время увидеть функции управления файлами и каталогами в действии. Программа 3.2 представляет собой ограниченную версию команды UNIX ls, предназначенной для вывода содержимого каталогов, которая позволяет вывести дату и время последнего изменения файла и размер файла, хотя данная версия отображает лишь младшую часть размера файла.

Программа просматривает каталог для поиска файлов, соответствующих шаблону поиска. Для каждого найденного файла программа отображает имя файла и, если был задан параметр –1, то и его атрибуты. Данная программа иллюстрирует принцип построения многих, хотя и далеко не всех, функций Windows, предназначенных для работы с каталогами.

Значительная часть кода программы 3.2 отвечает за обход дерева каталогов. Заметьте, что каждый каталог проходится дважды: при первом проходе обрабатываются файлы, а при втором — подкаталоги, чем обеспечивается поддержка параметра рекурсивного обхода каталогов (-R).

В том виде, как она представлена ниже, программа 3.2 будет корректно выполняться в том случае, если при ее вызове используются относительные полные имена файлов, например:

```
lsW -R include\*.h
```

Вместе с тем, в результате указания абсолютного полного имени файла, например:

```
lsW -R C:\Projects\ls\Debug\*.obj
```

правильная работа программы будет нарушена, поскольку в ней самым существенным образом используется привязка каталогов к текущему каталогу. Завершенное решение (доступное на Web-сайте) анализирует абсолютные полные пути доступа к файлам и поэтому обеспечивает правильное выполнение программы и для второй команды.

Программа 3.2. lsw: вывод списка файлов и обход дерева каталогов

```
/* Глава 3. lsW — команда вывода списка файлов */
/* lsW [параметры] [файлы] */
#include "EvryThng.h"
BOOL TraverseDirectory(LPCTSTR, DWORD, LPBOOL);
DWORD FileType(LPWIN32_FIND_DATA);
BOOL ProcessItem(LPWIN32_FIND_DATA, DWORD, LPBOOL);

int _tmain(int argc, LPTSTR argv[]) {
 BOOL Flags [MAX_OPTIONS], ok = TRUE;
 TCHAR PathName [MAX_PATH +1], CurrPath [MAX_PATH + 1];
 LPTSTR pSlash, pFileName;
 int i, FileIndex;
 FileIndex = Options(argc, argv, _T("R1"), &Flags[0], &Flags[1], NULL);
 I* "Разобрать" шаблон поиска на "родительскую часть" и имя файла. */
 GetCurrentDirectory(MAX_PATH, CurrPath); /* Сохранить текущий путь доступа. */
  if (argc < FileIndex +1) /* Путь доступа не указан. Использовать текущий
каталог. */
  ok = TraverseDirectory(_T("*"), MAX_OPTIONS, Flags);
 else for (i = FileIndex; i < argc; i++) {
  /* Обработать все пути, указанные в командной строке. */
  ok = TraverseDirectory(pFileName, MAX_OPTIONS, Flags) && ok;
  SetCurrentDirectory(CurrPath);
  /* Восстановить каталог. */
 }
 return ok ? 0 : 1;
}
```

```
    static BOOL TraverseDirectory(LPCTSTR PathName, DWORD NumFlags, LPBOOL Flags)
    /* Обход дерева каталогов; выполнить функцию ProcessItem для каждого случая
совпадения. */
    /* PathName: относительное или абсолютное имя просматриваемого каталога.*/
    {
     HANDLE SearchHandle;
     WIN32_FIND_DATA FindData;
     BOOL Recursive = Flags[0];
     DWORD FType, iPass;
     TCHAR CurrPath[MAX_PATH + 1];
     GetCurrentDirectory(MAX_PATH, CurrPath);
     for (iPass = 1; iPass <= 2; iPass++) {
      /* Проход 1: вывод списка файлов. */
      /* Проход 2: обход дерева каталогов (если задана опция -R). */
      SearchHandle = FindFirstFile(PathName, &FindData);
      do {
       FType = FileType(&FindData);
       /* Файл или каталог? */
       if (iPass == 1) /* Вывести имя и атрибуты файла. */
        ProcessItem(&FindData, MAX_OPTIONS, Flags);
       if (FType == TYPE_DIR && iPass == 2 && Recursive) {
        /* Обработать подкаталог. */
        _tprintf(_T ("\n%s\\%s:"), CurrPath, FindData.cFileName);
        /* Подготовка к обходу каталога. */
        SetCurrentDirectory(FindData.cFileName);
        TraverseDirectory(_T("*"), NumFlags, Flags);
        /* Рекурсивный вызов. */
        SetCurrentDirectory(_T(".."));
       }
      } while (FindNextFile(SearchHandle, &FindData));
      FindClose (SearchHandle);
     }
     return TRUE;
    }

    static BOOL ProcessItem(LPWIN32_FIND_DATA pFileData, DWORD NumFlags, LPBOOL
Flags)
    /* Выводит список атрибутов файла или каталога. */
    {
     const TCHAR FileTypeChar[] = {' ', 'd'};
     DWORD FType = FileType(pFileData);
     BOOL Long = Flags[1];
     SYSTEMTIME LastWrite;
     if (FType != TYPE_FILE && FType != TYPE_DIR) return FALSE;
     _tprintf(_T ("\n"));
     if (Long) { /* Указан ли в командной строке параметр "-1"? */
      _tprintf(_T("%c"), FileTypeChar[FType - 1]);
      _tprintf(_T("%10d"), pFileData->nFileSizeLow);
      FileTimeToSystemTime(&(pFileData->ftLastWriteTime), &LastWrite);
           _tprintf(_T("    %02d/%02d/%04d    %02d:%02d:%02d"),    LastWrite.wMonth,
LastWrite.wDay,      LastWrite.wYear,      LastWrite.wHour,      LastWrite.wMinute,
LastWrite.wSecond);
     }
     _tprintf(_T(" %s"), pFileData->cFileName);
     return TRUE;
    }

    static DWORD FileType(LPWIN32_FIND_DATA pFileData)
```

```
    /* Поддерживаемые типы файлов - TYPE_FILE: файл; TYPE_DIR: каталог; TYPE_DOT:
каталоги . или .. */
    {
     BOOL IsDir;
     DWORD FType;
     FType = TYPE_FILE;
     IsDir = (pFileData->dwFileAttributes & FILE_ATTRIBUTE_DIRECTORY) != 0;
       if   (IsDir)   if   (lstrcmp(pFileData->cFileName,   _T("."))   ==   0   ||
lstrcmp(pFileData->cFileName, _T("..")) == 0) FType = TYPE_DOT;
     else FType = TYPE_DIR;
     return FType;
    }
```

Пример: установка меток времени файла

Программа 3.3 реализует UNIX-команду touch, предназначенную для изменения кода защиты файлов и обновления меток времени до текущих значений системного времени. В упражнении 3.11 от вас требуется расширить возможности функции touch таким образом, чтобы новые значения меток времени можно было указывать в параметрах командной строки.

Программа 3.3. touch: установка меток даты и времени файла

```
    /* Глава 3. команда touch. */
    /* touch [параметры] [файлы] */
    #include "EvryThng.h"

    int _tmain(int argc, LPTSTR argv[]) {
     SYSTEMTIME SysTime;
     FILETIME NewFileTime;
     LPFILETIME pAccessTime = NULL, pModifyTime = NULL;
     HANDLE hFile;
     BOOL Flags[MAX_OPTIONS], SetAccessTime, SetModTime, CreateNew;
     DWORD CreateFlag;
     int i, FileIndex;
      FileIndex = Options(argc, argv, _T("amc"), &Flags[0], &Flags[1], &Flags[2],
NULL);
     SetAccessTime = !Flags[0];
     SetModTime = !Flags[1];
     CreateNew = !Flags[2];
     CreateFlag = CreateNew ? OPEN_ALWAYS : OPEN_EXISTING;
     for (i = FileIndex; i < argc; i++) {
        hFile = CreateFile(argv[i], GENERIC_READ | GENERIC_WRITE, 0, NULL,
CreateFlag, FILE_ATTRIBUTE_NORMAL, NULL);
      GetSystemTime(&SysTime);
      SystemTimeToFileTime(&SysTime, &NewFileTime);
      if (SetAccessTime) pAccessTime = &NewFileTime;
      if (SetModTime) pModifyTime = &NewFileTime;
      SetFileTime(hFile, NULL, pAccessTime, pModifyTime);
      CloseHandle(hFile);
     }
     return 0;
    }
```

Стратегии обработки файлов

Уже на ранних стадиях любого проекта разработки приложения или подготовки его к переносу на другую платформу приходится принимать решение относительно того, должна ли осуществляться обработка файлов с использованием функций библиотеки C или функций Windows. Характер этого решения не относится к категории "или-или", поскольку при соблюдении определенных мер предосторожности смешанное применение функций возможно даже по отношению к одному и тому же файлу.

Библиотека C обладает рядом явных преимуществ, среди которых можно выделить следующие:

• Полученный программный код легко переносится на другие системы.

• Наличие удобных функций для работы с символами и строками, не имеющих прямых аналогов среди функций Windows, упрощает обработку строк.

• Функции библиотеки C обычно проще в использовании по сравнению с функциями Windows.

• Функции, ориентированные на обработку символьных строк и потоков, легко преобразовать к форме, допускающей указание обобщенных символов при их вызове, хотя преимущества переносимости при этом будут утеряны.

• Как показано в главе 7, функции библиотеки C способны работать и в средах с многопоточной поддержкой.

Тем не менее, использование библиотеки C налагает некоторые ограничения. В пользу этого утверждения можно привести перечисленные ниже соображения:

• Средства библиотеки C не обеспечивают управление каталогами и обход дерева каталогов и в большинстве случаев не позволяют получать или устанавливать атрибутов файлов.

• В функции fseek, входящей в библиотеку C, используются 32-битовые указатели файла, и поэтому, несмотря на возможность последовательного считывания гигантских файлов, установка произвольной позиции в таком файле, как это требуется, например, в программе 3.1, оказывается невозможной.

• Библиотека C не предоставляет такие развитые возможности, как защита файлов, отображение файлов, блокирование файлов, асинхронный ввод/вывод и взаимодействие между процессами. Вместе с тем, как показано в приложении B, использование некоторых из этих возможностей в ряде случаев может обеспечивать существенное улучшение показателей производительности программ.

Альтернативным вариантом является перенос существующего UNIX-кода с привлечением библиотеки совместимости (compatibility library). Microsoft C предоставляет ограниченную библиотеку совместимости, включающую многие, хотя и далеко не все, функции UNIX. К числу функций UNIX, входящих в состав библиотеки Microsoft, относятся функции ввода/вывода, однако большинство функций управления процессами, не говоря о многих других функциях, в ней отсутствуют. В именах функций-аналогов присутствует префикс в виде символа подчеркивания, например, _read, _write, _stat и так далее.

Решения относительно смешанного использования функций библиотеки C, библиотеки совместимости и Win32/64 API должны приниматься на основании требований проекта. Многие из преимуществ функций Windows будут продемонстрированы в следующих главах, а для ознакомления с количественными данными, характеризующими производительность, которые пригодятся вам в тех случаях, когда этот фактор становится решающим, вы можете обратиться к приложению B.

Блокирование файлов

В системах, допускающих одновременное выполнение нескольких процессов, особую актуальность приобретает проблема координации и синхронизации доступа к разделяемым (совместно используемым) объектам, например файлам.

В Windows имеется возможность блокировать файлы (целиком или частично) таким образом, что никакой другой процесс (выполняющаяся программа) не сможет получить доступ к заблокированному участку файла. Блокирование файла может оставлять другим приложениям возможность доступа только для чтения (разделяемый доступ) или же закрывать им доступ к файлу как для записи, так и для чтения (монопольный доступ). Что немаловажно, владельцем блокировки является блокирующий процесс. Любая попытка получения доступа к части файла (с помощью функций ReadFile или WriteFile) в нарушение существующей блокировки закончится неудачей, поскольку блокировки носят обязательный характер на уровне процесса. Любая попытка получения несовместимой блокировки также завершится неудачей, даже если процесс уже владеет данной блокировкой. Блокирование файлов является ограниченной разновидностью синхронизации параллельно выполняющихся процессов и потоков; обсуждение синхронизации с использованием гораздо более общей терминологии начнется в главе 8.

Для блокирования файлов предусмотрены две функции. Более общей из них является функция LockFileEx, менее общей — LockFile, которую можно использовать и в Windows 9x.

Функция LockFileEx относится к классу функций расширенного (extended) ввода/вывода, поэтому для указания 64-битовой позиции в файле и границ области файла, подлежащей блокированию, необходимо использовать структуру OVERLAPPED, которая ранее уже применялась при указании позиции в файле для функций ReadFile и WriteFile.

```
BOOL LockFileEx(HANDLE hFile, DWORD dwFlags, DWORD dwReserved, DWORD nNumberOfBytesToLockLow, DWORD nNumberOfBytesToLockHigh, LPOVERLAPPED lpOverlapped)
```

Функция LockFileEx блокирует участок открытого файла либо для разделяемого доступа (разрешающего доступ одновременно нескольким приложениям в режиме чтения), либо для монопольного доступа (разрешающего доступ только одному приложению в режиме чтения/записи).

Параметры

hFile — дескриптор открытого файла. Дескриптор должен быть создан либо с правами доступа GENERIC_READ, либо с правами доступа GENERIC_READ и GENERIC_WRITE.

dwFlags — определяет вид блокировки файла, а также режим ожидания доступности затребованной блокировки. Этот параметр определяется комбинацией следующих значений:

LOCKFILE_EXCLUSIVE_LOCK — запрос монопольной блокировки в режиме чтения/записи. Если это значение не задано, запрашивается разделяемая блокировка (только чтение).

LOCKFILE_FAIL_IMMEDIATELY — задает режим немедленного возврата функции с возвращаемым значением равным FALSE, если приобрести блокировку не удается. Если это значение не задано, функция переходит в режим ожидания.

dwReserved — значение этого параметра должно устанавливаться равным 0. Следующие два параметра определяют соответственно младшие и старшие 32-битовые значения размера блокируемого участка файла (в байтах).

lpOverlapped — указатель на структуру данных OVERLAPPED, содержащую информацию о начале блокируемого участка. В этой структуре необходимо устанавливать значения трех элементов (остальные элементы игнорируются), первые два из которых определяют смещение начала блокируемого участка от начала файла.

• DWORD Offset (используется именно такое имя параметра, а не OffsetLow).

• DWORD OffsetHigh.

• HANDLE hEvent должен задаваться равным 0.

Чтобы разблокировать файл, следует вызвать функцию UnlockFileEx, все параметры которой, за исключением dwFlags, совпадают с параметрами предыдущей функции:

```
BOOL UnlockFileEx(HANDLE hFile, DWORD dwReserved, DWORD nNumberOfBytesToLockLow, DWORD nNumberOfBytesToLockHigh, LPOVERLAPPED lpOverlapped)
```

Используя блокирование файлов, вы должны принимать во внимание следующие обстоятельства:

• Границы области разблокирования должны в точности совпадать с границами ранее заблокированной области. Не допускается, например, объединение двух ранее заблокированных областей или разблокирование части заблокированной области. Любая попытка разблокирования области, не совпадающей в точности с одной из существующих заблокированных областей, будет неудачной. В этом случае функция вернет значение FALSE, а в выведенном системой сообщении об ошибке будет указано, что данная область блокирования не существует.

• Вновь создаваемая и существующие области блокирования в файле не могут перекрываться, если это приводит к возникновению конфликтной ситуации.

• Возможно блокирование участка, границы которого выходят за пределы файла. Такая операция может оказаться полезной в случае расширения файла процессом или потоком.

• Блокировки не наследуются вновь создаваемыми процессами.

Логику процедуры блокирования, когда вся область *или* только некоторая ее часть уже содержат заблокированные участки, иллюстрирует табл. 3.1.

Таблица 3.1. Логика предоставления блокировки

	Тип запрашиваемой блокировки	
Существующая блокировка	**Разделяемая блокировка**	**Монопольная блокировка**
Отсутствует	Предоставляется	Предоставляется
Разделяемая блокировка (одна или несколько)	Предоставляется	Отказ
Монопольная блокировка	Отказ	Отказ

Логику предоставления возможности выполнения операций чтения/записи во всей или части области файла, содержащей участки с одной или несколькими блокировками, владельцами которых являются другие процессы, иллюстрирует табл. 3.2.

Таблица 3.2. Блокировки и выполнение операций ввода/вывода

	Операция ввода/вывода	
Существующая блокировка	**Чтение**	**Запись**
Отсутствует	Успешно выполняется	Успешно выполняется

Разделяемая блокировка (одна или несколько)	Выполняется. Вызывающий процесс не обязан быть владельцем блокировки данной области файла.	Не выполняется
Монопольная блокировка	Выполняется, если вызывающий процесс является владельцем блокировки, в противном случае — неудачное завершение.	Выполняется, если вызывающий процесс является владельцем блокировки, в противном случае — неудачное завершение.

Обычно операции чтения и записи выполняются путем вызова функций Read-File и WriteFile или их расширенных версий ReadFileEx и WriteFileEx. Для диагностики ошибок, возникающих в процессе выполнения операций ввода/вывода, следует вызывать функцию GetLastError.

Одна из разновидностей операций ввода/вывода с участием файлов предполагает использование отображения файлов, которое обсуждается в главе 5. Обнаружение конфликтов блокировки на этапе обращения к памяти не производится; такая проверка осуществляется во время вызова функции MapViewOfFile. Указанная функция делает часть файла доступной для процесса, вследствие чего проверка наличия блокировок на этом этапе является необходимой.

Разновидностью функции LockFileEx с ограниченной сферой применимости является функция LockFile, вызов которой, скорее, лишь уведомляет о намерении осуществить блокировку. Эту функцию можно использовать в системах Windows 9x, которые не поддерживают функцию LockFileEx. Функция LockFile предоставляет блокирующему процессу только монопольный доступ, а возврат из функции происходит сразу же. Таким образом, функция LockFile не блокируется. Проверить, предоставлена блокировка или нет, можно путем тестирования возвращаемого функцией значения.

Снятие блокировок

Каждый успешный вызов функции LockFileEx должен сопровождаться последующим вызовом функции UnlockFileEx (то же самое касается и пары функций LockFile и UnlockFile). Если программа не позаботится о снятии блокировки или будет удерживать ее в течение большего, чем это необходимо, времени, другие программы либо вовсе не смогут работать, либо будут вынуждены простаивать. Поэтому уже на стадии проектирования и реализации программ необходимо очень тщательно следить за тем, чтобы снятие блокировки осуществлялось сразу же после того, как необходимость в ней отпала, а логика работы программ не позволяла оставлять невыполненными необходимые операции разблокирования файлов.

Одним из способов, гарантирующих своевременное разблокирования файлов, является использование дескрипторов завершения (termination handlers), которые описаны в главе 4.

Следствия принятой логики блокирования файлов

Несмотря на всю естественность логики блокирования файлов, представленной в таблицах 3.1 и 3.2, последствия ее применения могут оказаться для вас неожиданными и вызвать на первый взгляд необъяснимые изменения в поведении программы. Некоторые возможные примеры этого приводятся ниже.

• Предположим, что процессы A и B периодически приобретают разделяемые блокировки

файла, а процесс С блокируется при попытке получения монопольной блокировки того же файла после того, как процесс А стал владельцем собственной разделяемой блокировки. В этих условиях процесс В может получить свою разделяемую блокировку, но процесс С будет оставаться блокированным даже после того, как процесс А снимет свою блокировку файла. Процесс С будет оставаться блокированным до тех пор, пока все процессы не снимут свои блокировки, даже если они были получены уже тогда, когда процесс С пребывал в блокированном состоянии. Согласно этому сценарию процесс С может оставаться блокированным сколь угодно долго, тогда как другие процессы сохраняют возможность управления своими разделяемыми блокировками.

• Предположим, что процесс А стал владельцем разделяемой блокировки файла, а процесс В пытается осуществить считывание файла без предварительного приобретения разделяемой блокировки. В этой ситуации чтение может быть успешно осуществлено даже несмотря на то, что процесс, выполняющий чтение, не владеет ни одной блокировкой данного файла, поскольку операция чтения не вступает в конфликт с существующей разделяемой блокировкой.

• Все, о чем говорилось выше, относится не только к блокировке файла в целом, но и к блокировке отдельного его участка.

• Процессы чтения и записи вполне могут успешно завершить часть своего запроса, прежде чем возникнет конфликт с существующей блокировкой. В этом случае функции чтения и записи возвратят значения FALSE, а значение счетчика переданных байтов окажется меньше затребованного.

Использование блокирования файлов

Рассмотрение примеров блокирования файлов мы отложим до главы 6, в которой обсуждается управление процессами. В программах 4.2, 6.4, 6.5 и 6.6 блокирование файлов используется для обеспечения того, чтобы в каждый момент времени изменять файл мог только один процесс.

В UNIX блокирование файлов является *уведомляющим* (advisory); выполнение процесса ввода/вывода может продолжаться даже в том случае, если попытка получения блокировки оказалась неудачной (логика, отраженная в табл. 3.1, действует и в этом случае). Это обеспечивает в UNIX возможность блокирования файлов взаимодействующими процессами, но любой другой процесс может нарушить описанный протокол.

Для получения уведомляющей блокировки используются параметры, указываемые при вызове функции fcntl. Допустимыми командами (второй параметр) являются F_SETLK, F_SETLKW и F_GETLK. Информация о типе блокировки (F_RDLCK, F_WRLCK или F_UNLCK) и блокируемой области содержится в дополнительной структуре данных.

Помимо этого, в некоторых UNIX-системах доступна *обязательная* (mandatory) блокировка, обеспечиваемая путем определения групповых полномочий для файла с помощью команды chmode.

Блокирование файлов в UNIX имеет много особенностей. Например, блокировки наследуются при выполнении вызова функции exec.

Блокирование файлов библиотекой С не поддерживается, но в Visual C++ обеспечивается поддержка нестандартных расширений механизма блокирования.

Реестр

Реестр — это централизованная иерархическая база данных, хранящая информацию о параметрах конфигурации операционной системы и установленных приложений. Доступ к реестру осуществляется через *разделы,* или *ключи, реестра* (registry keys), играющие ту же роль, что и каталоги в файловой системе. Раздел может содержать подразделы или пары "имя-значение", в которых между именем и значением существует примерно та же взаимосвязь, что и между именами файлов и их содержимым.

Пользователь или системный администратор может просматривать и изменять содержимое реестра, пользуясь редактором реестра, для запуска которого необходимо выполнить команду REGEDIT. Реестром можно управлять также из программ, используя функции API реестра, описанные в данном разделе.

> **Примечание**
>
> Программирование реестра обсуждается в данной главе по той причине, что решаемая при этом задача весьма напоминает обработку файлов, а также потому, что оно играет важную роль в некоторых, хотя и не во всех, приложениях. Соответствующий пример будет получен путем несложного изменения программы lsW. Вместе с тем, данный раздел вполне мог бы стать небольшой отдельной главой. Поэтому читатели, для которых программирование реестра не представляет непосредственного интереса, могут пропустить этот раздел, чтобы вернуться к нему впоследствии, если это окажется необходимым.

В парах "имя-значение" реестра хранится следующая информация:

• Номер версии операционной системы, номер сборки и информация о зарегистрированном пользователе.

• Аналогичная информация обо всех приложения, которые были надлежащим образом установлены в системе.

• Информация о типе процессоров в системе и их количестве, системной памяти и тому подобное.

• Специфическая для каждого отдельного пользователя системы информация, включая данные относительно основного каталога пользователя и предпочтительных пользовательских настройках приложений.

• Информация, необходимая для системы безопасности, включая имена учетных записей пользователей.

• Информация об установленных службах (глава 13).

• Список соответствий между расширениями имен файлов и ассоциированными с ними исполняемыми программами. Именно эти соответствия используются системой после того, как пользователь щелкнет на пиктограмме какого-либо файла. Например, щелчок на файле с расширением .doc может приводить к запуску текстового редактора Microsoft Word.

• Отображения сетевых адресов на имена, используемые локальным компьютером.

В операционной системе UNIX аналогичная информация хранится в каталоге /etc и файлах, находящихся в основном каталоге пользователя. В Windows 3.1 для этих целей использовались .INI-файлы. Реестр обеспечивает единообразное централизованное хранение всей информации подобного рода. Кроме того, используя средства защиты, описанные в главе 15, можно обеспечить безопасность реестра.

API управления реестром описывается ниже, однако подробное рассмотрение содержимого и смысла различных записей, образующих реестр, выходит за рамки данной книги. Тем не менее, общее представление о структуре и содержимом этого хранилища данных можно получить на рис. 3.1, на котором изображен типичный вид окна открытого редактора реестра.

Рис. 3.1. Окно редактора реестра

Справа на этом рисунке можно видеть специфическая информация, относящаяся к установленному на данном локальном компьютере процессору. В нижней левой части рисунка показаны различные разделы, содержащие информацию об установленном в локальной системе программном обеспечении. Обратите внимание, что каждый ключ обязательно имеет значение по умолчанию, которое указывается в списке самым первым, предшествуя любым другим парам "имя-значение".

Рассмотрение принципов реализации реестра, включая организацию хранения и извлечения хранящихся в реестре данных, выходит за рамки данной книги; для более глубокого изучения этих вопросов обратитесь к списку дополнительной литературы, приведенному в конце главы.

Ключи реестра

На рис. 3.1 показана аналогия между разделами реестра и каталогами файловой системы. Каждый раздел может содержать другие разделы или последовательности пар "имя-значение". В то время как доступ к файловой системе реализуется посредством указания путей доступа, доступ к реестру осуществляется через его разделы. Существует несколько предопределенных разделов, которые играют роль точек входа в реестр.

• HKEY_LOCAL_MACHINE. В этом разделе хранится информация об оборудовании локального компьютера и установленном на нем программном обеспечении. Информация об установленном программном обеспечении обычно создается в подразделах (subkeys) в виде: SOFTWARE\НазваниеКомпании\НазваниеПродукта\Версия.

• HKEY_USERS. В этом разделе хранится информация о настройке пользовательских конфигураций.

• HKEY_CURRENT_CONFIG. В этом разделе хранятся текущие настройки таких параметров, как разрешение дисплея или гарнитура шрифта.

• HKEY_CLASSES_ROOT. В этом разделе содержатся подчиненные записи, устанавливающие соответствие между именами файлов и классами, а также приложениями, используемыми оболочкой для доступа к объектам, имена которых имеют определенные

расширения. В этот раздел также входят все подразделы, необходимые для функционирования модели компонентных объектов (Component Object Model — COM), разработанной компанией Microsoft.

• HKEY_CURRENT_USER. В этом разделе хранится информация, определяемая пользователем, в том числе информация о переменных среды, принтерах и предпочтительных для вошедшего в систему пользователя конфигурационных параметрах приложений.

Управление системным реестром

Функции управления реестром позволяют запрашивать и изменять данные, относящиеся к парам "имя-значение", а также создавать новые подразделы и новые пары "имя-значение". Как для указания существующих разделов, так и для создания новых используются дескрипторы типа HKEY.[17] Нужные значения необходимо вводить; тип значения можно выбрать из нескольких готовых вариантов, соответствующих, например, строкам, двойным словам или расширяемым (expandable) строкам, параметры которых могут быть заменены переменными окружения.

Управление подразделами реестра

Первая из рассматриваемых нами функций, RegOpenKeyEx, предназначена для открытия подразделов системного реестра. Начав с одного из предопределенных зарезервированных дескрипторов, вы можете получить дескриптор любого из его подразделов, совершая обход дерева разделов.

```
LONG RegOpenKeyEx(HKEY hKey, LPCTSTR lpSubKey, DWORD ulOptions, REGSAM
samDesired, PHKEY phkResult)
```

Параметры

hKey — указатель на текущий открытый раздел реестра или значение дескриптора одного из предопределенных зарезервированных разделов. pHkResult — указатель на переменную типа HKEY, получающую значение дескриптора вновь открываемого раздела.

lpSubKey — указатель на строку с именем подраздела. Именем подраздела может быть путь, например: Microsoft\WindowsNT\CurrentVersion. Значению NULL соответствует открытие новой копии раздела для hKey. Значение параметра ulOptions должно быть равным 0.

samDesired — маска доступа, описывающая уровень защиты нового раздела. К числу возможных значений относятся значения KEY_ALL_ACCESS, KEY_WRITE, KEY_QUERY_VALUE и KEY_ENUMARATE_SUBKEYS.

В случае успешного завершения функции возвращается значение ERROR_SUCCESS. Возврат любого другого значения указывает на ошибку. Для закрытия дескриптора открытого раздела используется функция RegCloseKey, которая в качестве своего единственного параметра принимает дескриптор.

Для получения имен подразделов любого заданного раздела следует воспользоваться функцией RegEnumKeyEx.

Для получения пар "имя-значение" используются две взаимно дополняющих функции: RegEnumValue и RegQueryValueEx. [18] Функция RegSetValueEx сохраняет данные различного типа в поле значения открытого раздела реестра. Описания перечисленных функций, применение которых будет проиллюстрировано примером, содержатся в этом и следующем разделах книги.

Функция RegEnumKeyEx перечисляет подразделы открытого раздела системного реестра во многом аналогично тому, как функции FindFirstFile и FindNextFile перечисляют содержимое каталогов. Эта функция извлекает имя подраздела, строку с именем класса подраздела, а также дату и время последнего изменения.

```
LONG RegEnumValue(HKEY hKey, DWORD dwIndex, LPTSTR lpValueName, LPDWORD
lpcbValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD
lpcbData)
```

Значение параметра dwIndex должно устанавливаться равным 0 при первом вызове функции и увеличиваться на единицу при каждом последующем вызове. Название раздела и его размер, а также строка с именем класса и ее размер, возвращаются обычным способом. В случае успешного завершения функция возвращает значение ERROR_SUCCESS, иначе — код ошибки.

Можно также создавать новые разделы, используя для этого функцию RegCreateKeyEx. Разделам системного реестра можно присваивать атрибуты защиты точно так же, как каталогам и файлам (глава 15).

```
LONG RegCreateKeyEx(HKEY hKey, LPCTSTR lpSubKey, DWORD Reserved, LPTSTR
lpClass, DWORD dwOptions, REGSAM samDesired, LPSECURITY_ATTRIBUTES
lpSecurityAttributes, PHKEY phkResult, LPDWORD lpdwDisposition)
```

Параметры

lpSubKey — указатель на строку, содержащую имя нового подраздела, создаваемого в разделе, на который указывает дескриптор hKey.

lpClass — указатель на строку, содержащую имя класса, или объектный тип, раздела, описывающее данные, представляемые разделом. Одними из многочисленных возможных значений являются REG_SZ (строка, завершающаяся нулевым символом) и REG_DWORD (двойное слово).

Параметр dwOptions может иметь значение 0 или одно из двух взаимоисключающих значений — REG_OPTION_VOLATILE или REG_OPTION_NON_VOLATILE. Постоянно хранимая (nonvolatile) информация системного реестра сохраняется в файле на диске и не теряется после перезапуска системы. При этом временные (volatile) разделы системного реестра, хранящиеся в оперативной памяти, не будут восстановлены.

Параметр samDesired имеет тот же смысл, что и в случае функции RegOpenKeyEx.

Параметр lpSecurityAttributes может принимать значение NULL или указывать атрибуты защиты. Опции прав доступа к разделу могут выбираться из того же набора значений, что и в случае параметра samDesired.

lpdwDisposition — указатель на переменную типа DWORD, значение которой говорит о том, существовал ли раздел ранее (REG_OPENED_EXISTING_KEY) или он только создается (REG_CREATED_NEW_KEY).

Для удаления раздела используется функция RegDeleteKey. Двумя ее параметрами являются дескриптор открытого раздела и имя подраздела.

Управление значениями

Для перечисления значений параметров открытого раздела реестра используется функция RegEnumValue. Значение параметра dwIndex должно устанавливаться равным 0 при первом вызове функции и увеличиваться на единицу при каждом последующем вызове. После возврата из функции вы получаете строку, содержащую имя перечисляемого параметра, а также размер данных. Кроме того, вы получаете значение перечисляемого параметра и его тип.

```
LONG RegEnumValue(HKEY hKey, DWORD dwIndex, LPTSTR lpValueName, LPDWORD
lpcbValueName, LPDWORD lpReserved, LPDWORD lpType, LPBYTE lpData, LPDWORD
lpcbData)
```

Фактическое значение параметра возвращается в буфере, на который указывает указатель lpData. Размер результата содержится в переменной, на которую указывает указатель lpcbData. Тип данных, содержащийся в переменной, на которую указывает указатель lpType, может быть самым различным, включая REG_BINARY, REG_DWORD, REG_SZ (строка) и REG_EXPAND_SZ (расширяемая строка с параметрами, которые заменяются переменными окружения). Полный список типов данных системного реестра можно найти в оперативной справочной системе.

Чтобы определить, все ли параметры перечислены, следует проверить возвращаемое значение функции. После успешного нахождения действительного параметра оно должно быть равным ERROR_SUCCESS.

Функция RegQueryValueEx ведет себя аналогичным образом, за исключением того, что требует указания имени перечисляемого параметра, а не его индекса. Эту функцию можно использовать в тех случаях, когда известны имена параметров. Если же имена параметров неизвестны, следует использовать функцию RegEnumValueEx.

Для установки значения параметра в открытом разделе служит функция RegSetValueEx, которой необходимо предоставить имя параметра, тип значения и фактические данные, образующие значение.

```
LONG RegSetValueEx(HKEY hKey, LPCTSTR lpValueName, DWORD Reserved,
DWORD dwType, CONST BYTE * lpData, CONST cbData)
```

Наконец, для удаления именованных значений используется функция RegDeleteValue.

Пример: вывод списка разделов и содержимого реестра

Программа lsReq (программа 3.4), является видоизменением lsW (программа 3.2, предназначенная для вывода списка файлов и каталогов) и обрабатывает не каталоги и файлы, а разделы и пары "имя-значение" системного реестра.

Программа 3.4. lsReq: вывод списка разделов и содержимого системного реестра

```
    /* Глава 3. lsReg: Команда вывода содержимого реестра. Адаптированная версия программы 3.2. */
    /* lsReg [параметры] подраздел */
    #include "EvryThng.h"
    BOOL TraverseRegistry(HKEY, LPTSTR, LPTSTR, LPBOOL);
    BOOL DisplayPair(LPTSTR, DWORD, LPBYTE, DWORD, LPBOOL);
    BOOL DisplaySubKey (LPTSTR, LPTSTR, PFILETIME, LPBOOL);

    int _tmain(int argc, LPTSTR argv[]) {
     BOOL Flags[2], ok = TRUE;
     TCHAR KeyName[MAX_PATH + 1];
     LPTSTR pScan;
     DWORD i, KeyIndex;
     HKEY hKey, hNextKey;
     /* Таблица предопределенных имен и дескрипторов разделов. */
     LPTSTR PreDefKeyNames[] = {
      _T("HKEY_LOCAL_MACHINE"), _T("HKEY_CLASSES_ROOT"),
      _T("HKEY CURRENT USER"), _T ("HKEY CURRENT CONFIG"), NULL
     };
     HKEY PreDefKeys[] = {
      HKEY_LOCAL_MACHINE, HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_CURRENT_CONFIG
     };
     KeyIndex = Options(argc, argv, _T("Rl"), &Flags[0], &Flags[1], NULL);
     /* "Разобрать" шаблон поиска на "раздел" и "подраздел". */
     /* Воссоздать раздел. */
     pScan = argv[KeyIndex];
      for (i = 0; *pScan != _T('\\') && *pScan != _T('\0'); pScan++, i++) KeyName[i] = *pScan;
     KeyName[i] = _T('\0');
     if (*pScan == _T('\\')) pScan++;
     /* Преобразовать предопределенное имя раздела в соответствующий HKEY.*/
      for (i = 0; PreDefKeyNames [i] != NULL && _tcscmp(PreDefKeyNames[i], KeyName) != 0; i++);
     hKey = PreDefKeys[i];
     RegOpenKeyEx(hKey, pScan, 0, KEY_READ, &hNextKey);
     hKey = hNextKey;
     ok = TraverseRegistry(hKey, argv[KeyIndex], NULL, Flags);
     return ok ? 0 : 1;
    }

    BOOL TraverseRegistry(HKEY hKey, LPTSTR FullKeyName, LPTSTR SubKey, LPBOOL Flags)
    /*Совершить обход разделов и подразделов реестра, если задан параметр -R.*/
    {
     HKEY hSubK;
     BOOL Recursive = Flags[0];
     LONG Result;
```

```
    DWORD ValType, Index, NumSubKs, SubKNameLen, ValNameLen, ValLen;
    DWORD MaxSubKLen, NumVals, MaxValNameLen, MaxValLen;
    FILETIME LastWriteTime;
    LPTSTR SubKName, ValName;
    LPBYTE Val;
    TCHAR FullSubKName[MAX_PATH + 1];
    /* Открыть дескриптор раздела. */
    RegOpenKeyEx(hKey, SubKey, 0, KEY_READ, &hSubK);
     /* Определить максимальный размер информации относительно раздела и
распределить память. */
     RegQueryInfoKey(hSubK, NULL, NULL, NULL, &NumSubKs, &MaxSubKLen, NULL,
&NumVals, &MaxValNameLen, &MaxValLen, NULL, &LastWriteTime);
    SubKName = malloc (MaxSubKLen+1); /* Размер без учета завершающего нулевого
символа. */
    ValName = malloc(MaxValNameLen+1); /* Учесть нулевой символ. */
    Val = malloc(MaxValLen); /* Размер в байтах. */
    /* Первый проход: пары "имя-значение". */
    for (Index = 0; Index < NumVals; Index++) {
     ValNameLen = MaxValNameLen + 1; /* Устанавливается каждый раз! */
     ValLen = MaxValLen + 1;
       RegEnumValue(hSubK, Index, ValName, &ValNameLen, NULL, &ValType, Val,
&ValLen);
     DisplayPair(ValName, ValType, Val, ValLen, Flags);
    }
    /* Второй проход: подразделы. */
    for (Index = 0; Index < NumSubKs; Index++) {
     SubKNameLen = MaxSubKLen + 1;
       RegEnumKeyEx(hSubK, Index, SubKName, &SubKNameLen, NULL, NULL, NULL,
&LastWriteTime);
     DisplaySubKey(FullKName, SubKName, &LastWriteTime, Flags);
     if (Recursive) {
      _stprintf(FullSubKName, _T("%s\\%s"), FullKName, SubKName);
      TraverseRegistry(hSubK, FullSubKName, SubKName, Flags);
     }
    }
    _tprintf(_T("\n"));
    free(SubKName);
    free(ValName);
    free(Val);
    RegCloseKey(hSubK);
    return TRUE;
   }

   BOOL DisplayPair(LPTSTR ValueName, DWORD ValueType, LPBYTE Value, DWORD
ValueLen, LPBOOL Flags)
   /* Функция, отображающая пары "имя-значение". */
   {
    LPBYTE pV = Value;
    DWORD i;
    _tprintf(_T("\nValue: %s = "), ValueName);
    switch (ValueType) {
    case REG_FULL_RESOURCE_DESCRIPTOR: /* 9: описание оборудования. */
    case REG_BINARY: /* 3: Любые двоичные данные. */
     for (i = 0; i < ValueLen; i++, pV++) _tprintf (_T (" %x"), *pV);
     break;
    case REG_DWORD: /* 4: 32-битовое число. */
     _tprintf(_T ("%x"), (DWORD)*Value);
     break;
    case REG_MULTI_SZ: /*7: массив строк, завершающихся нулевым символом.*/
    case REG_SZ: /* 1: строка, завершающаяся нулевым символом. */
```

```
      _tprintf(_T("%s"), (LPTSTR)Value);
      break;
     /* … Несколько других типов … */
     }
     return TRUE;
    }

    BOOL DisplaySubKey(LPTSTR KeyName, LPTSTR SubKeyName, PFILETIME pLastWrite,
LPBOOL Flags) {
     BOOL Long = Flags[1];
     SYSTEMTIME SysLastWrite;
     _tprintf(_T("\nSubkey: %s"), KeyName);
     if (_tcslen(SubKeyName) > 0) _tprintf (_T ("\\%s "), SubKeyName);
     if (Long) {
      FileTimeToSystemTime(pLastWrite, &SysLastWrite);
           _tprintf(_T("%02d/%02d/%04d    %02d:%02d:%02d"),    SysLastWrite.wMonth,
SysLastWrite.wDay,  SysLastWrite.wYear,  SysLastWrite.wHour,  SysLastWrite.wMinute,
SysLastWrite.wSecond);
     }
     return TRUE;
    }
```

Резюме

В главах 2 и 3 описаны все наиболее важные базовые функции, необходимые для работы с файлами, каталогами и консольным вводом/выводом. Использование этих функций для построения типичных приложений иллюстрировали многочисленные примеры. Как показывает последний из примеров, между управлением системным реестром и управлением файловой системой имеется много общего.

В последующих главах будут рассмотрены такие усовершенствованные методы ввода/вывода, как асинхронные операции ввода/вывода и отображение файлов. Этих средств будет достаточно для того, чтобы воспроизвести в Windows почти любой из обычных видов обработки файлов, доступных при использовании UNIX или библиотечных функций C.

В приложении Б приведены сравнительные таблицы функций Windows, UNIX и библиотеки C, в которых наглядно показано, в чем указанные группы функций соответствуют друг другу, а в чем заметно отличаются.

В следующих главах

Глава 4 рассказывает о том, как упростить обработку ошибок и исключений, и распространяет применение функции ReportError на случаи обработки любых исключительных ситуаций.

Дополнительная литература

Для получения более подробной информации относительно программирования и использования системного реестра Windows, обратитесь к книге [17].

Упражнения

3.1. Используя функции GetDiskFreeSpace и GetDiskFreeSpaceEx, определите, насколько разреженным оказывается файловое пространство, распределяемое различными версиями операционной системы Windows. Например, создайте новый файл, установите для указателя файла большое значение, задайте размер файла и исследуйте наличие свободного пространства на жестком диске при помощи функции GetDiskFreeSpace. Эту же функцию Windows можно использовать для определения того, чтобы определить, каким образом сконфигурирован диск в терминах секторов и кластеров. Определите, инициализируется ли выделенное для вновь созданного файла дисковое пространство. Решение в виде исходного текста функции FreeSpace.c доступно на Web-сайте книги. Сравните результаты, полученные для столь различных систем, как Windows NT и Windows 9x. Представляет интерес также исследование вопроса о том, как сделать файл разреженным.

3.2. Что произойдет, если длину файла задать такой, чтобы его размер превышал объем диска? Обеспечивает ли Windows изящный выход из функции в случае ее неудачного завершения?

3.3. Измените предоставляемую на Web-сайте программу tail.c таким образом, чтобы в ней можно было обойтись без применения функции SetFilePointer; воспользуйтесь для этого структурой OVERLAPPED.

3.4. Исследуйте значение поля "количество ссылок" (nNumberOfLinks), полученное с использованием функции GetFileInformationByHandle. Всегда ли оно равно 1? Различаются ли ответы на этот вопрос для файловых систем NTFS и FAT? Не включают ли значения счетчиков ссылок жесткие ссылки и ссылки из родительских каталогов и подкаталогов, как это имеет место в UNIX? Открывает ли Windows каталог как файл для получения дескриптора, прежде чем использовать эту функцию? Что можно сказать о ярлыках, поддерживаемых пользовательским интерфейсом?

3.5. В программе 3.2 поиск текущего и родительских каталогов осуществляется с использованием имен "." и "..". Что произойдет в случае, если файлы с такими именами действительно существуют? Могут ли файлы иметь такие имена?

3.6. Значения какого времени выводятся в программе 3.2 — местного или UCT? При необходимости измените программу таким образом, чтобы выводимые значения соответствовали местному времени.

3.7. Усовершенствуйте программу 3.2 таким образом, чтобы в выводимый список включались также текущий (".") и родительский ("..") каталоги (завершенная программа находится на Web-сайте). Кроме того, добавьте опции, позволяющие наряду с датой и временем последнего изменения отображать дату и время создания файла, а также дату и время последнего доступа к нему.

3.8. Напишите программу, которая реализует команду rm, позволяющую удалять файлы, изменив для этого функцию ProcessItem в программе 3.2. Решение доступно на Web-сайте.

3.9. Усовершенствуйте команду cp из главы 2, предназначенную для копирования файлов, таким образом, чтобы она позволяла копировать файлы в указанный каталог. Дополнительно предусмотрите опцию рекурсивного копирования файлов (параметр –r) и опцию сохранения вместе с копией также времени последнего изменения файла (параметр –p). Для реализации опции рекурсивного копирования файлов вам потребуется создать новые каталоги.

3.10. Напишите программу mv, которая реализует одноименную команду UNIX, позволяющую переместить целиком любой каталог. При этом имеет существенное значение,

осуществляется ли перемещение файла или каталога на другой диск или они остаются на прежнем диске. В случае смены диска используйте операцию копирования файлов, в противном случае используйте команды MoveFile или MoveFileEx.

3.11. Усовершенствуйте программу 3.3 (touch) таким образом, чтобы новое время создания файла можно было указывать в командной строке. Команда UNIX допускает (по выбору) указание метки времени после обычных параметров, но перед именами файлов. Метки даты и времени имеют формат MMddhhmm [yy], где MM — месяцы, dd — дни, hh — часы, mm — минуты, yy — года. Двух цифр для обозначения года нам будет недостаточно, поэтому предусмотрите для указания года четыре разряда.

3.12. Программа 3.1 рассчитана на работу с большими файловыми системами NTFS. Если на вашем жестком диске имеется достаточно много свободного места, протестируйте работу этой программы на файлах гигантских размеров (свыше 4 Гбайт). Проверьте, насколько корректно работает 64-битовая арифметика. Выполнять это упражнение на сетевом диске без предварительного разрешения администратора сети не рекомендуется. Завершив работу над этим упражнением, не забудьте удалить тестовый файл.

3.13. Напишите программу, которая блокирует заданный файл и удерживает его в блокированном состоянии в течение длительного времени (вероятно, вызахотите воспользоваться функцией Sleep). Воспользовавшись любым текстовым редактором, попытайтесь получить доступ к файлу (используйте текстовый файл) в период действия блокировки. Что при этом происходит? Заблокирован ли файл должным образом? Вы также можете написать программу, предлагающую пользователю задать блокировку для тестового файла. Чтобы проверить, срабатывает ли блокировка описанным образом, запустите на выполнение два экземпляра программы в разных окнах. Решение этого упражнения содержится в файле TestLock.c, находящемся на Web-сайте.

3.14. Исследуйте представление временных характеристик файла Windows в формате данных FILETIME. В этом формате используются 64-битовые счетчики, выражающие в 100-наносекундных единицах длительность истекшего периода времени, отсчитываемого от 1 января 1601 года. Когда исчерпаются показания этого счетчика? Какова максимально допустимая дата для временных характеристик файлов UNIX?

3.15. Напишите интерактивную утилиту, в которой пользователю предлагается ввести имя раздела реестра и имя значения реестра. Отобразите текущее значение и предложите пользователю указать новое.

3.16. В этой главе, как и в большинстве других глав книги, описываются наиболее важные функции. Однако во многих случаях вам могут оказаться полезными и другие функции. На страницах оперативного справочного руководства для каждой функции приведены ссылки на родственные функции. Ознакомьтесь с некоторыми из них, такими, например, как FindFirstFileEx, ReplaceFile, SearchPath или WriteFileGather. Некоторые функции доступны не во всех версиях NT5.

ГЛАВА 4
Обработка исключений

Основное внимание в данной главе сфокусировано на структурной обработке исключений (Structured Exception Handling, SEH), но наряду с этим обсуждены также обработчики управляющих сигналов консоли и векторная обработка исключений (Vectored Exception Handling, VEH).

SEH предоставляет механизм обеспечения надежности программ, благодаря которому приложения получают возможность реагировать на такие непредсказуемые события, как исключения адресации, арифметические сбои и системные ошибки. Использование SEH позволяет программам осуществлять корректный выход из любой точки программного блока и автоматически выполнять предусмотренную программистом обработку ошибок для восстановления своей работоспособности. SEH гарантирует своевременное освобождение ресурсов и выполнение любых других операций очистки, прежде чем блок, поток или процесс закончат работу либо под управлением программы, либо в ответ на возникновение исключительной ситуации. Кроме того, SEH легко добавляется в существующие программные коды, во многих случаях обеспечивая упрощение логики работы программы.

Мы используем SEH в приведенных ниже примерах программ и расширим посредством этого механизма возможности функции обработки ошибок ReportError, которая была введена в главе 2. Обычно сфера применимости SEH ограничивается программами, написанными на языке на C. Вместе с тем, представленные ниже возможности SEH воспроизводятся в C++, C# и других языках программирования с использованием весьма похожих механизмов.

В настоящей главе описаны также обработчики управляющих сигналов консоли, благодаря которым программы могут воспринимать внешние сигналы, вырабатываемые, например, при нажатии сочетания клавиш <Ctrl+C>, выходе пользователя из системы или завершении работы системы. Кроме того, использование подобных сигналов обеспечивает реализацию ограниченных форм межпроцессного взаимодействия.

Глава завершается рассмотрением векторной обработки исключений, которая потребует от вас использования операционных систем Windows XP или Windows Server 2003. Благодаря VEH пользователь получает возможность определить функции, которые должны вызываться сразу же после возникновения исключения, не дожидаясь активизации SEH.

Исключения и обработчики исключений

В отсутствие обработки исключений возникновение любой нестандартной ситуации, например, попытки разыменования нулевого указателя или деления на ноль, приведет к немедленному прекращению выполнения программы. В качестве примера, иллюстрирующего проблемы, которые могут при этом возникать, можно назвать создаваемые программой временные файлы, подлежащие удалению до того, как программа завершит свою работу. SEH предоставляет возможность определить блок программного кода, или *обработчик исключений* (exception handler), который в случае возникновения исключения удалит временные файлы.

Поддержка SEH обеспечивается за счет совместного использования функций Windows, средств поддержки языков программирования, предоставляемых компилятором, и средств поддержки времени выполнения. Какой именно язык программирования поддерживается, зависит от конкретной системы; наши примеры ориентированы на Microsoft C.

Блоки try и except

Все начинается с выяснения того, в каких именно блоках программного кода вы намерены контролировать возникновение нестандартных ситуаций, после чего этим блокам должны быть предоставлены обработчики исключений в соответствии с приведенным ниже описанием. Можно контролировать как функцию в целом, так и ее отдельные программные блоки или подфункции, предусмотрев для них независимые обработчики исключений.

Ниже перечислены характерные признаки участков программного кода, для которых целесообразно предусматривать отдельные обработчики исключений.

• Возможность возникновения регистрируемых ошибок, включая ошибки системных вызовов, в условиях, когда необходимо организовать устранение последствий ошибки, а не предоставлять программе возможность прекращения выполнения.

• Интенсивное использование указателей, повышающее вероятность попыток разыменования указателей, инициализация которых не была выполнена должным образом.

• Интенсивное использование данных в виде массивов, что может сопровождаться выходом значений индексов элементов массива за границы допустимого диапазона.

• В программе выполняются арифметические операции с участием вещественных чисел (чисел с плавающей точкой), и существует риск того, что могут возникать исключения, связанные с попытками деления на ноль, потерей точности при вычислениях и переполнением.

• Наличие вызовов функций, которые могут генерировать исключения либо программным путем, либо в силу того, что их работоспособность не была достаточно тщательно проверена.

Если при изучении примеров, приведенных в этой главе или книге в целом, вы решите отслеживать исключения, которые могут возникать на том или ином участке программы, создайте для него блоки try и except, как показано ниже:

```
__try {
 /* Блок контролируемого кода */
} __ except(выражение_фильтра) {
 /* Блок обработки исключений */
}
```

Имейте в виду, что __try и __except — это ключевые слова, распознаваемые компилятором.

Блоки try являются частью обычного кода приложения. Если на данном участке кода возникает исключение, ОС передает управление обработчику исключений, который представляет собой блок программного кода, следующий за ключевым словом_ except. Характер

последующих действий определяется значением параметра выражение_фильтра.

Обратите внимание, что исключение может возникнуть также в пределах блока, находящегося внутри try-блока; в этом случае средства поддержки времени исполнения "разворачивают" стек, чтобы отыскать в нем информацию об обработчике исключений, после чего передают управление этому обработчику. То же самое происходит и в тех случаях, когда исключения возникают внутри функций, вызванных в пределах try-блока.

На рис. 4.1 показано, как располагается в стеке информация об обработчике исключений во время возникновения исключения. Как только обработчик исключений завершит свою работу, управление передается оператору, который следует за блоком except, если только в самом обработчике исключений не предусмотрены иные операторы ветвления, изменяющие ход выполнения программы.

Выражения фильтров и их значения

Параметр выражение_фильтра в операторе except вычисляется сразу же после того, как возникает исключение. В качестве выражения может выступать литеральная константа, вызов *функции фильтра* (filter function) или условное выражение. В любом случае выражение должно возвращать одно из следующих трех значений:

1. EXCEPTION_EXECUTE_HANDLER — система выполняет операторы блока обработки исключений, как показано на рис. 4.1 (см. программу 4.1). Это соответствует обычному случаю.
2. EXCEPTION_CONTINUE_SEARCH — система игнорирует данный обработчик исключений и пытается найти обработчик исключений в охватывающем блоке, продолжая этот процесс аналогичным образом до тех пор, пока не будет найден обработчик исключений.
3. EXCEPTION_CONTINUE_EXECUTION — система немедленно возвращает управление в точку, в которой возникло исключение. В случае некоторых исключений дальнейшее выполнение программы невозможно, но если такие попытки делаются, то генерируется повторное исключение.

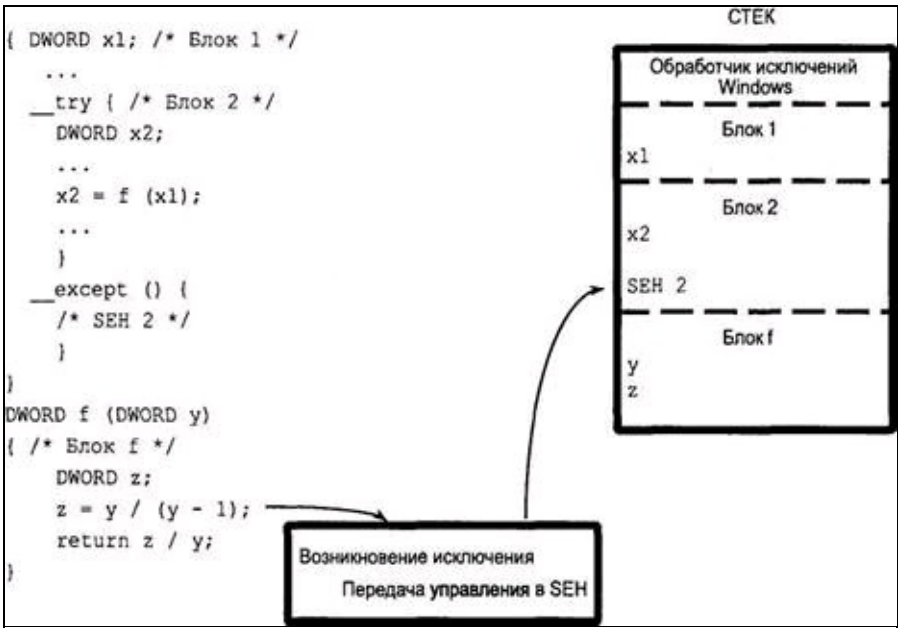


Рис. 4.1. SEH, блоки и функции

Ниже приведен простой пример, в котором обработчик исключений используется для удаления временного файла в тех случаях, когда исключение возникает в теле цикла. Заметьте, что ключевое слово __try может быть применено к любому блоку, включая блоки, связанные с операторами while, if или любым другим оператором ветвления. В данном примере

возникновение любого исключения приводит к удалению временного файла и закрытию дескриптора, после чего выполнение цикла возобновляется.

```
    GetTempFileName(TempFile, …);
    while (…) __try {
     hFile = CreateFile(TempFile, …, OPEN_ALWAYS, …);
     SetFilePointer(hFile, 0, NULL, FILE_END);
     WriteFile(hFile, …);
      i = *p; /* В этом месте программы возможно возникновение исключения адресации.
*/
     CloseHandle (hFile);
     …
    } __except (EXCEPTION_EXECUTE_HANDLER) {
     CloseHandle(hFile);
     DeleteFile(TempFile);
     /* Переход к выполнению очередной итерации цикла. */
    }
    /* Сюда передается управление после нормального завершения цикла.
       Каждый раз при возникновении исключения дескриптор временного
файла  закрывается, а сам файл удаляется. */
```

Ниже описана логика приведенного выше фрагмента кода.

• На каждой итерации цикла в конце файла добавляются новые данные.

• В случае возникновения исключения во время выполнения итерации цикла все данные, накопленные во временном файле, будут уничтожены, и если еще остались невыполненные итерации, то во временном файле начнут накапливаться новые данные.

• В случае возникновения исключения на последней итерации файл прекращает существование. В любом случае файл будет содержать все данные, сгенерированные после предыдущего исключения.

• В примере отмечена лишь одна точка программы, в которой возможно возникновение исключения, хотя исключения могут возникнуть в любой точке тела цикла.

• Чтобы гарантировать закрытие дескриптора файла, это делается как при выходе из цикла, так и перед началом очередной итерации цикла.

Коды исключений

Для точной идентификации типа возникшего исключения блок исключения или выражение фильтра могут использовать следующую функцию:

```
DWORD GetExceptionCode(VOID)
```

Код исключения должен быть получен сразу же после возникновения исключения. Поэтому функция фильтра не может просто вызвать функцию GetExceptionCode (это ограничение налагается компилятором). Обычный способ решения этой проблемы состоит в том, чтобы осуществить этот вызов в выражении фильтра, как показано в следующем примере, в котором код исключения является аргументом функции фильтра, предоставляемой пользователем:

```
    __except(MyFilter(GetExceptionCode())) {
    }
```

В данном случае значение выражения фильтра, которое должно быть одним из трех указанных ранее значений, определяется и возвращается функцией фильтра. В свою очередь, для определения возвращаемого этой функцией значения используется код исключения; например, можно сделать так, чтобы фильтр передавал обработку исключений, возникающих при выполнении операций с плавающей точкой (FP-исключений, от *FloatingPoint* — плавающая

точка), внешнему обработчику (возвращая значение EXCEPTION_CONTINUE_SEARCH), а обработку нарушений доступа к памяти — текущему обработчику (возвращая значение EXCEPTION_EXECUTE_HANDLER).

Число возможных кодов исключений, возвращаемых функцией GetExceptionCode, очень велико, однако их можно разделить на несколько категорий.

• Выполнение программой некорректных действий, например:

EXCEPTION_ACCESS_VIOLATION — попытка чтения или записи по адресу виртуальной памяти, к которой процесс не имеет доступа.

EXCEPTION_DATATYPE_MISALIGNMENT — многие процессоры, например, требуют чтобы данные типа DWORD выравнивались по четырехбайтовым границам.

EXCEPTION_NONCONTINUABLE_EXECUTION — значением выражения фильтра было EXCEPTION_CONTINUE_EXECUTION, но выполнения программы после возникновения исключения не может быть продолжено.

• Исключения, сгенерированные функциями распределения памяти HeapAlloc и HeapCreate, если они используют флаг HEAP_GENERATE_EXCEPTIONS (см. главу 5). Соответствующими значениями кода исключения являются STATUS_NO_MEMORY или EXCEPTION_ACCESS_VIOLATION.

• Коды определенных пользователем исключений, генерируемых путем вызова функции RaiseException, о чем говорится в подразделе "Исключения, генерируемые приложением".

• Коды различных арифметических исключений (особенно FP-исключений), например, EXCEPTION_INT_DIVIDE_BY_ZERO или EXCEPTION_FLT_OVERFLOW.

• Исключения, используемые отладчиками, например, EXCEPTION_BREAKPOINT или EXCEPTION_SINGLE_STEP.

Вам пригодится также функция GetExceptionInformation, которая может быть вызвана только из выражения фильтра и возвращает дополнительную информацию, включая информацию, специфическую для используемого процессора.

```
LPEXCEPTION_POINTERS GetExceptionINFORMATION(VOID)
```

Вся информация, как относящаяся, так и не относящаяся к процессору, содержится в структуре EXCEPTION_POINTERS, состоящей из двух других структур.

```
typedef struct _EXCEPTION_POINTERS {
 PEXCEPTION_RECORD ExceptionRecord;
 PCONTEXT ContextRecord;
} EXCEPTION POINTERS;
```

В структуру EXCEPTION_RECORD входит элемент ExceptionCode, набор возможных значений которого совпадает с набором значений, возвращаемых функцией GetExceptionCode. Элемент ExceptionFlags структуры EXCEPTION_RECORD может принимать значения 0 или EXCEPTION_NONCONTINUABLE, причем последнее значение указывает функции фильтра на то, что она не должна предпринимать попыток продолжения выполнения. К числу других элементов данных этой структуры относятся адрес виртуальной памяти ExceptionAddress и массив параметров ExceptionInformation. В случае исключения EXCEPTION_ACCESS_VIOLATION значение первого элемента этого массива указывает на то, какая именно из операций пыталась получить доступ по недоступному адресу — записи (1) или чтения (0). Второй элемент содержит адрес виртуальный памяти.

Во втором элементе структуры EXCEPTION_POINTERS, а именно, элементе ContextRecord,

содержится информация, относящаяся к процессору. Для каждого типа процессоров предусмотрены свои структуры, определения которых содержатся в файле <winnt.h>.

Резюме: последовательность обработки исключений

На рис. 4.2 в схематическом виде представлена последовательность событий, происходящих после возникновении исключения. Слева приведен программный код, а обведенные кружками цифры справа обозначают операции, выполняемые языковыми средствами поддержки времени выполнения. Отдельные элементы приведенной схемы имеют следующий смысл:

1. Возникло исключение; в данном случае это деление на ноль.
2. Управление передается обработчику исключений, в котором вычисляется выражение фильтра. Сначала вызывается функция GetExceptionCode, а затем ее возвращаемое значение используется в качестве аргумента функции Filter.
3. Функция фильтра выполняет действия, определяемые значением кода исключения.
4. В данном случае значением кода исключения является EXCEPTION_INT_DIVIDE_BY_ZERO.
5. Функция фильтра устанавливает, что должен быть выполнен код обработчика исключений, и поэтому возвращает значение EXCEPTION_EXECUTE_HANDLER.
6. Выполняется код обработчика исключений, связанного с оператором _except.
7. Управление передается за пределы блоков try и except.

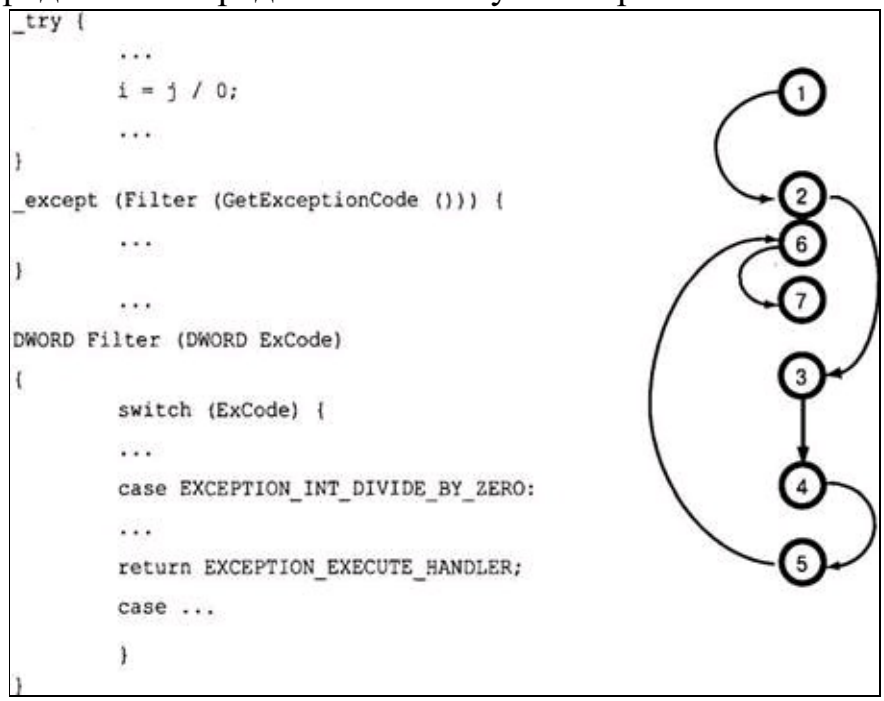


Рис. 4.2. Последовательность операций при обработке исключений

Исключения, возникающие при выполнении операций над числами с плавающей точкой

Существует семь различных кодов исключений, которые могут возникать при выполнении операций с использованием данных вещественного типа. Первоначально эти исключения отключены и не могут возникать до тех пор, пока с помощью функции _controlfp для них не будет предварительно задана специальная маска, не зависящая от типа процессора. Предусмотрены отдельные исключения для ситуаций антипереполнения, переполнения, деления на ноль, неточного результата и так далее, что иллюстрируется приведенным ниже фрагментом кода. Для активизации исключений определенного типа следует *отключить* соответствующий бит маски.

```
DWORD _controlfp(DWORD new, DWORD mask)
```

Фактическое значение маски определяется ее текущим значением (current_mask) и двумя аргументами следующим образом:

```
(current_mask & ~mask) | (new & mask)
```

Данная функция устанавливает лишь те из битов, указанных в аргументе new, которые разрешены аргументом mask. Биты, не активизированные аргументом mask, не изменяются. Маска FP-исключений управляет также точностью, округлением и обработкой значений, соответствующих бесконечности, поэтому при активизации перечисленных исключений необходимо тщательно следить за тем, чтобы случайно не изменить эти установки.

Возвращаемым значением является фактическое значение маски. Так, при нулевых значениях обоих аргументов возвращаемым значением будет текущее значение маски (current_mask), что может быть использовано для восстановления маски, если впоследствии в этом возникнет необходимость. С другой стороны, если задать аргумент mask равным 0xFFFFFFFF, то регистр установится в new, что, например, может быть использовано для восстановления прежнего значения маски.

Обычно для того, чтобы разрешить исключения, связанные с выполнением операций над числами с плавающей точкой, в качестве аргумента mask используют константу MCW_EM, как продемонстрировано в следующем примере. Также заметьте, что при обработке FP-исключения оно должно быть сброшено путем использования функции _clearfp.

```
#include <float.h>
DWORD FPOld, FPNew; /* Старое и новое значения маски. */
…
FPOld = _controlfp(0, 0); /* Сохранить старую маску. */
/* Указать в качестве разрешенных шесть типов исключений. */
FPNew = FPOld & ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT | EM_ZERODIVIDE |
EM_DENORMAL | EM_INVALID);
/* Установить новую управляющую маску. Параметр MCW_EM объединяет шесть
исключений, указанных в предыдущем операторе. */
_controlfp(FPNew, MCW_EM);
while(…) __try { /* Выполнить вычисления над числами с плавающей точкой. */
 … /* На этом участке кода может возникнуть FP-исключение. */
} __except(EXCEPTION_EXECUTE_HANDLER) {
 … /* Обработать FP-исключение. */
 _clearfp(); /* Сбросить исключение. */
 _controlfp(FPOld, 0xFFFFFFFF); /* Восстановить маску. */
}
```

В этом примере разрешены все возможные FP-исключения, кроме одного —

EXCEPTION_FLT_STACK_CHECK, которое соответствует переполнению стека при выполнении операций над числами с плавающей точкой. Можно поступить и по-другому, разрешая отдельные исключения путем использования только выбранных масок исключений, например EM_OVERFLOW. Аналогичный код используется в программе 4.3 в контексте примера программного кода большего объема.

Ошибки и исключения

Под ошибками понимаются исключительные ситуации, которые время от времени могут возникать в известных местах программы. Так, обнаружение ошибок, возникающих во время выполнения системных вызовов, и немедленный вывод сообщений о них должны предусматриваться логикой работы самой программы. Поэтому программисты, как правило, явно включают в программный код участки, ответственные, например, за тестирование успешности завершения операции чтения данных из файла. В главе 2 для диагностики ошибок и принятия соответствующих мер была разработана функция ReportError.

С другой стороны, исключения могут возникать практически в любом месте программы, и поэтому организация явной проверки всех исключений невозможна или практически нецелесообразна. Примерами подобных ситуаций могут служить попытки деления на ноль или обращения к недоступным областям памяти.

Вместе с тем, указанные различия между ошибками и исключениями являются довольно условными. Windows позволяет управлять генерацией исключений, возникающих в случае нехватки памяти при ее распределении с использованием функций HeapAlloc и HeapCreate. Этот процесс описан в главе 5. Помимо этого, программы могут генерировать собственные исключения с кодами, определяемыми программистом, используя для этого функцию RaiseException, о чем далее будет говориться.

Обработчики исключений обеспечивают удобный механизм выхода из внутренних блоков или функций под управлением программы без использования операторов перехода goto или longjmp. Такая возможность оказывается особенно полезной, если блок получил доступ к таким, например, ресурсам, как открытые файлы, память или объекты синхронизации, поскольку обработчик может взять на себя задачу освобождения этих ресурсов. Возможно также продолжение работы программы после выполнения кода обработчика исключений, а не ее обязательное завершение. Кроме того, после выхода из блока программа может восстанавливать прежнее состояние системы, например маску FP-исключений. Именно в этом ключе обработчики используются во многих наших примерах.

Исключения, генерируемые приложением

Существует возможность формирования исключений в любой точке программы в процессе ее выполнения с помощью функции RaiseException. Это позволяет программе обнаруживать и обрабатывать возникающие ошибки как исключения.

```
VOID RaiseException(DWORD dwExceptionCode, DWORD dwExceptionFlags, DWORD cArguments, CONST DWORD *lpArguments)
```

Параметры

dwExceptionCode — код исключения, определяемый пользователем. Бит 28 использовать нельзя, так как он зарезервирован системой. Для кода ошибки отводятся биты 27—0 (то есть все слово, кроме самого старшего шестнадцатеричного разряда). Бит 29 должен быть установлен, чтобы показать, что данное исключение имеет "пользовательскую" природу (а не относится к числу тех, которые предусмотрела Microsoft). В битах 31—30 содержится код серьезности ошибки, принимающий приведенные ниже значения, в которых результирующая старшая шестнадцатеричная цифра кода исключения представлена с установленным битом 29.

• 0 — успешное выполнение (старшая шестнадцатеричная цифра кода исключения равна 2).
• 1 — информационный код (старшая шестнадцатеричная цифра кода исключения равна 6).
• 2 — предупреждение (старшая шестнадцатеричная цифра кода исключения равна A).
• 3 — ошибка (старшая шестнадцатеричная цифра кода исключения равна E).

dwExceptionFlags — обычно устанавливается равным 0, тогда как установка значения EXCEPTION_NONCONTINUABLE будет указывать на то, что выражение фильтра не должно возвращать значение EXCEPTION_CONTINUE_EXECUTION; при попытке это сделать будет немедленно сгенерировано исключение EXCEPTION_NONCONTINUABLE_EXCEPTION.

lpArguments — этот указатель, если он не равен NULL, указывает на массив размера cArguments (третий параметр), содержащий 32-битовые значения, которые должны быть переданы выражению фильтра. Максимально возможное число этих значений ограничивается значением EXCEPTION_MAXIMUM_PARAMETERS, которое в настоящее время установлено равным 15. Для доступа к этой структуре следует использовать функцию GetExceptionInformation.

Заметьте, что невозможно сгенерировать исключение в другом процессе. В то же время, при весьма ограниченных условиях для этой цели могут быть использованы обработчики управляющих сигналов консоли, о чем говорится в конце этой главы и в главе 6.

Пример: обработка ошибок как исключений

В предыдущих примерах для обработки ошибок при выполнении системных вызовов и других ошибок используется функция ReportError. Эта функция прекращает выполнение процесса, если программист указал, что данная ошибка является критической. Вместе с тем, такой подход препятствует нормальному выходу из программы и не обеспечивает возможность продолжения работы программы после устранения последствий ошибки. Так, после отказа от задачи, которая привела к возникновению сбоя, может потребоваться уничтожение временных файлов, созданных в процессе работы программы, или переход программы к выполнению других задач. Функции ReportError присущи и другие ограничения, перечень которых приводится ниже.

• Даже в тех случаях, когда было бы достаточно прекратить выполнения только одного потока, критическая ошибка приводит к остановке всего процесса (главу 7).

• Вместо завершения процесса может оказаться желательным продолжение выполнения программы.

• Во многих случаях становится невозможным освобождение ресурсов синхронизации (глава 8), например мьютексов.

При прекращении выполнения процесса (но не потоки) открытые дескрипторы будут закрываться, однако при этом необходимо учитывать другие отрицательные факторы.

Решение заключается в написании новой функции — ReportException. Если ошибка не является критической, эта функция вызывает функцию ReportError (разработанную в главе 2), которая выводит сообщение об ошибке. В случае же возникновения критической ошибки будет сгенерировано исключение. Система будет использовать обработчик исключений из вызывающего try-блока, и поэтому в действительности характер исключения может быть некритическим, если обработчик предоставляет программе возможность восстановиться после сбоя. По существу, функция ReportException дополняет обычные программные методы защиты от ошибок, ранее ограниченные функцией ReportError. В случае обнаружения ошибки обработчик позволяет программе продолжить свою работу после выполнения необходимых восстановительных действий. Эти возможности иллюстрирует программа 4.2.

Функция ReportException представлена в программе 4.1. Необходимые определения и заголовочные файлы не указаны, поскольку эта функция находится в том же модуле исходного кода, что и функция ReportError.

Программа 4.1. ReportException: функция вывода сообщений об исключениях

```
    /* Расширение функции ReportError для генерации формируемого приложением кода
исключения вместо прекращения выполнения процесса. */
    VOID ReportException(LPCTSTR UserMessage, DWORD ExceptionCode)
    /* Вывести сообщение о некритической ошибке. */
    {
     ReportError(UserMessage, 0, TRUE);
     /* Если ошибка критическая, сгенерировать исключение. */
      if  (ExceptionCode  !=  0)  RaiseException((0x0FFFFFFF  &  ExceptionCode)  |
0xE0000000, 0, 0, NULL);
     return;
    }
```

Функция ReportException используется в нескольких последующих примерах.

Модель сигналов, используемая в UNIX, значительно отличается от SEH. Сигналы

могут быть пропущены или игнорированы, и логика их работы иная. Тем не менее, у этих моделей имеются и общие черты.

Значительная часть поддержки обработки сигналов в UNIX обеспечивается библиотекой C, ограниченная версия которой доступна также под управлением Windows. Во многих случаях в программах Windows вместо сигналов можно воспользоваться обработчиками управляющих сигналов консоли, описанными в конце данной главы.

Некоторые сигналы соответствуют исключениямWindows.

Перечень в некоторой мере ограниченных соответствий "сигнал-исключение" представлен ниже:

• SIGILL — EXCEPTION_PRIV_INSTRUCTION

• SIGSEGV — EXCEPTION_ACCESS_VIOLATION

• SIGFPE — семь различных исключений, связанных с выполнением операций над числами с плавающей точкой, например EXCEPTION_FLT_DIVIDE_BY_ZERO

• SIGUSR1 и SIGUSR2 — исключения, определяемые приложением

Функции RaiseException соответствует функция библиотеки C raise.

В Windows сигналы SIGILL, SIGSEGV и SIGFPE не генерируются, хотя функция raise может генерировать один из них. Сигнал SIGINT в Windows не поддерживается.

Функция UNIX kill (kill не входит в состав стандартной библиотеки C), которая посылает сигнал другому процессу, может быть сопоставлена функции Windows GenerateConsoleCtrlEvent (глава 6). Для ограниченного варианта SIGKILL в Windows имеются аналоги в виде функций TerminateProcess и TerminateThread, с помощью которых один процесс (или поток) может уничтожить другой, хотя при использовании этих функций необходимо соблюдать осторожность (см. главы 6 и 7).

Обработчики завершения

Обработчики завершения служат в основном тем же целям, что и обработчики исключений, но выполняются, когда поток покидает блок в результате нормального выполнения программы, а также когда возникает исключение. С другой стороны, обработчик завершения не может распознавать исключения.

Обработчик завершения строится с использования ключевого слова __finally в операторе try…finally. Структура этого оператора аналогична структуре оператора try…finally, но в ней отсутствует выражение фильтра. Как и обработчики исключений, обработчики завершения предоставляют удобные возможности для закрытия дескрипторов, освобождения ресурсов, восстановления масок и выполнения иных действий, направленных на восстановление известного состояния системы после выхода из блока. Например, программа может выполнять операторы return внутри блока, оставляя всю работу по "уборке мусора" обработчику завершения. Благодаря этому отпадает необходимость во включении кода очистки в код самого блока или переходе к коду очистки при помощи оператора goto.

```
__try {
 /* Блок кода. */
} _finally {
 /* Обработчик завершения (блок finally). */
}
```

Выход из try-блока

Обработчик завершения выполняется всякий раз, когда в соответствии с логикой программы осуществляется выход из try-блока по одной из следующих причин:

• Достижение конца try-блока и "проваливание" в обработчик завершения.

• Выполнение одного из следующих операторов таким образом, что происходит выход за пределы блока:

```
return
break
goto[19]
longjmp
continue
__leave[20]
```

• Исключение.

Аварийное завершение

Любое завершение выполнения программы по причинам, отличным от достижения конца try-блока и "проваливания вниз" или выполнения оператора __leave, считается аварийным завершением. Результатом выполнения оператора __leave является переход в конец блока __try и передача управления вниз по тексту программы, что намного эффективнее простого использования оператора goto, поскольку не требует разворачивания стека. Для определения того, каким образом завершилось выполнение try-блока, в обработчике завершения используется следующая функция:

```
BOOL AbnormalTermination(VOID)
```

При аварийном завершении выполнения блока эта функция возвращает значение TRUE, при нормальном — FALSE.

Примечание

Завершение будет считаться аварийным, даже если, например, последним оператором try-блока был оператор return.

Выполнение обработчика завершения и выход из него

Обработчик завершения, или блок __finally, выполняется в контексте блока или функции, работу которых он отслеживает. Управление может переходить от оператора завершения к следующему оператору. Существует и другая возможность, когда обработчик завершения выполняет оператор передачи управления (return, break, continue, goto, longjmp или __leave). Еще одной возможностью является выход из обработчика по причине возникновения исключения.

Сочетание блоков finally и except

Один try-блок может иметь только один блок finally или только один блок except, но не может иметь оба указанных блока одновременно. Поэтому нижеприведенный код вызовет появление ошибок на стадии компиляции.

```
__try {
 /* Блок контролируемого кода. */
} __except (filter_expression) {
 /* Блок обработчика исключений. */
} __finally {
 /* Так делать нельзя! Это приведет к ошибке на стадии компиляции. */
}
```

Вместе с тем, допускается вложение одного блока в другой, что используется довольно часто. Нижеприведенный код является вполне работоспособным и обеспечивает гарантированное удаление временных файлов при выходе из цикла под управлением программы или в результате возникновения исключения. Эта методика оказывается удобной и в тех случаях, когда требуется обеспечить гарантированную отмену блокирования файлов, что будет использовано в программе 4.2. Кроме того, в коде имеется внутренний блок try…except, размещенный в том месте программы, где выполняются вычисления, в которых участвуют вещественные числа.

```
__try { /* Внешний блок try-except. */
 while (…) __try { /* Внутренний блок try-finally. */
  hFile = CreateFile(TempFile, …);
  if(…) __try { /* Внутренний блок try-except. */
   /* Разрешить FP-исключения. Выполнить вычисления. */
   …
  } __except(EXCEPTION_EXECUTE_HANDLER) {
   … /* Обработать FP-исключение. */
   _clearfp();
  }
  … /* Обработка исключений, не являющихся FP-исключениями. /*
 } __finally { /* Конец цикла while. */
  /* Выполняется на КАЖДОЙ итерации цикла. */
  CloseHandle(hFile);
  DeleteFile(TempFile);
 }
} __except (filter-expression) {
```

```
 /* Обработчик исключений. */
}
```

Глобальное и локальное разворачивание стека

Исключения и аварийные завершения вызывают *глобальное разворачивание стека* (global stack unwind) в поиске обработчика, как было показано на рис. 4.1. Предположим, например, что в отслеживаемом блоке примера, приведенного в конце предыдущего раздела, исключение возникает прежде, чем активизируются FP-исключения. Тогда перед обработчиком исключения в стеке могут находиться многочисленные обработчики завершения.

Вспомните, что структура стека является динамической, как показано на рис. 4.1, и что в стеке, наряду с другими данными, хранятся данные обработчиков исключений и завершения. Фактическое содержимое стека в любой момент времени зависит от следующих факторов:

• *Статической* структуры программных блоков.

• *Динамической* структуры программы, отражаемой в последовательности открытых вызовов функций.

Обработчики завершения: завершение процессов и потоков

Обработчики завершения не выполняются, если выполнение процесса или потока было прекращено независимо от того, было ли это инициировано самим процессом путем использования функций ExitProcess или ExitThread, или вызвано извне, например, инициировано вызовом функций TerminateProcess или TerminateThread из другого места в программе. Поэтому ни одна из этих функций не должна вызываться процессом или потоком внутри блоков try… except или try…finally.

Обратите также внимание, что выполнение функции библиотеки C exit или возврат из функции main приводят к выходу из процесса.

SEH и обработка исключений в C++

При обработке исключений в C++ используются ключевые слова catch и throw, а сам механизм исключений реализован с использованием SEH. Тем не менее, обработка исключений в C++ и SEH — это разные вещи. Их совместное применение требует внимательного обращения, поскольку обработчики исключений, написанные пользователем и сгенерированные C++, могут взаимодействовать между собой и приводить к нежелательным последствиям. Например, находящийся в стеке обработчик __except может перехватить исключение C++, в результате чего данное исключение так и не дойдет до обработчика C++.

Возможно и обратное, когда, например, обработчик C++ перехватит SEH-исключение, сгенерированное функцией RaiseException. Документация Microsoft рекомендует полностью отказаться от использования обработчиков Windows в программах на C++ и ограничиться применением в них только обработчиков исключений C++.

Кроме того, обработчики исключений или завершения Windows не осуществляют вызов деструкторов, что в ряде случаев необходимо для уничтожения экземпляров объектов C++.

Пример: использование обработчиков завершения для повышения качества программ

Обработчики исключений и завершения позволяют повысить надежность программ как за счет упрощения процедуры восстановления программы после возникновения ошибок и исключений, так и за счет гарантированного освобождения ресурсов и отмены блокирования файлов в критических ситуациях.

В программе toupper (программа 4.2) эти моменты иллюстрируются с привлечением идей, почерпнутых в программном коде предшествующих примеров. toupper обрабатывает несколько файлов, имена которых указываются в командной строке, переписывая их с преобразованием всех букв в верхний регистр. Имена преобразованных файлов получаются путем добавления префикса UC_ к исходным именам, и согласно "спецификации" программы запись поверх существующих файлов не производится. Преобразование файлов осуществляется в памяти машины, поэтому для каждого файла выделяется большая буферная область (достаточная для размещения всего файла). Кроме того, чтобы исключить любую возможность изменения файлов другими процессами, а также для того, чтобы вновь создаваемые выходные файлы строго соответствовали преобразованным входным файлам, оба вида файлов блокируются во время обработки. Понятно, что на каждой стадии обработки существует вероятность возникновения самых различных сбойных ситуаций, но в программе должна быть предусмотрена защита от подобных ошибок, и она должна располагать средствами, позволяющими ей восстановить свое нормальное состояние и попытаться обработать все остальные файлы, имена которых были указаны в командной строке. Программа 4.2 решает все эти задачи, обеспечивая разблокирование файлов во всех необходимых случаях без применения громоздкой логики операторов ветвления, к которым пришлось бы прибегнуть, если бы не были использованы средства SEH. Более подробные комментарии к программе содержатся в программном коде, находящемся на Web-сайте книги.

Программа 4.2. toupper: обработка файлов с восстановлением нормального состояния программы после сбоев

```
    /* Глава 4. Команда toupper. */
    /* Преобразование содержимое одного и более файлов с заменой всех букв на прописные. Имя выходного файла получается из имени входного файла добавлением к нему префикса UC_. */
    #include "EvryThng.h"

    int _tmain(DWORD argc, LPTSTR argv[]) {
     HANDLE hIn = INVALID_HANDLE_VALUE, hOut = INVALID_HANDLE_VALUE;
     DWORD FileSize, nXfer, iFile, j;
     CHAR OutFileName [256] = "", *pBuffer = NULL;
     OVERLAPPED ov = {0, 0, 0, 0, NULL}; /* Используется для блокирования файлов. */
     if (argc <= 1) ReportError(_T("Использование: toupper файлы"), 1, FALSE);
     /* Обработать все файлы, указанные в командной строке. */
     for (iFile = 1; iFile < argc; iFile++) __try { /* Блок исключений. */
        /* Все дескрипторы файлов недействительны, pBuffer == NULL, а файл OutFileName пуст. Выполнение этих условий обеспечивается обработчиками. */
      _stprintf(OutFileName, "UC_%s", argv[iFile]);
      __try { /* Внутренний блок try-finally. */
```

```
      /* Ошибка на любом шаге сгенерирует исключение, и следующий */
      /* файл будет обрабатываться только после "уборки мусора". */
      /* Объем работы по очистке зависит от того, в каком месте */
      /* программы возникла ошибка. */
       /* Создать выходной файл (завершается с ошибкой, если файл уже существует).
*/
        hIn = CreateFile(argv[iFile], GENERIC_READ, 0, NULL, OPEN_EXISTING, 0,
NULL);
      if (hIn == INVALID_HANDLE_VALUE) ReportException(argv[iFile], 1);
      FileSize = GetFileSize(hIn, NULL);
        hOut = CreateFile(OutFileName, GENERIC_READ | GENERIC_WRITE, 0, NULL,
CREATE_NEW, 0, NULL);
      if (hOut == INVALID_HANDLE_VALUE) ReportException(OutFileName, 1);
      /* Распределить память под содержимое файла. */
      pBuffer = malloc(FileSize);
       if (pBuffer == NULL) ReportException(_T("Ошибка при распределении памяти"),
1);
      /* Блокировать оба файла для обеспечения целостности копии. */
        if (!LockFileEx(hIn, LOCKFILE_FAIL_IMMEDIATELY, 0, FileSize, 0, &ov)
ReportException(_T("Ошибка при блокировании входного файла"), 1);
       if (!LockFileEx(hOut, LOCKFILE_EXCLUSIVE_LOCK | LOCKFILE_FAIL_IMMEDIATELY,
0, FileSize, 0, &ov) ReportException(_T("Ошибка при блокировании выходного файла
"), 1);
      /* Считать данные, преобразовать их и записать в выходной файл. */
      /* Освободить ресурсы при завершении обработки или возникновении */
      /* ошибки; обработать следующий файл. */
              if (!ReadFile(hIn, pBuffer, FileSize, &nXfer, NULL))
ReportException(_T("Ошибка при чтении файла"), 1);
      for (j = 0; j < FileSize; j++) /* Преобразовать данные. */
       if (isalpha(pBuffer [j])) pBuffer[j] = toupper(pBuffer [j]);
              if(WriteFile(hOut, pBuffer, FileSize, &nXfer, NULL))
ReportException(T("Ошибка при записи в файл"), 1);
     } __finally {
      /*Освобождение блокировок, закрытие дескрипторов файлов,*/
      /*освобождение памяти и повторная инициализация */
      /*дескрипторов и указателя. */
      if (pBuffer != NULL) free (pBuffer);
      pBuffer = NULL;
      if (hIn != INVALID_HANDLE_VALUE) {
       UnlockFileEx(hIn, 0, FileSize, 0, &ov);
       CloseHandle(hIn);
       hIn = INVALID_HANDLE_VALUE;
      }
      if (hOut != INVALID_HANDLE_VALUE) {
       UnlockFileEx(hOut, 0, FileSize, 0, &ov);
       CloseHandle(hOut);
       hOut = INVALID_HANDLE_VALUE;
      }
      _tcscpy(OutFileName, _T(""));
     }
    }
    /* Конец основного цикла обработки файлов и блока try. */
    /* Обработчик исключений для тела цикла. */
    __except(EXCEPTION_EXECUTE_HANDLER) {
     _tprintf(_T("Ошибка при обработке файла %s\n"), argv[iFile]);
     DeleteFile (OutFileName);
    }
    _tprintf(_T("Обработаны все файлы, кроме указанных выше \n"));
    return 0;
   }
```

Пример: использование функции фильтра

Программа 4.3 представляет собой каркас программы, иллюстрирующей обработку исключений и завершения выполнения, в которой используется функция фильтра. Программа предлагает пользователю указать тип исключения, после чего продолжает работу для генерации исключения. Функция фильтра обрабатывает различные типы исключений по-разному; выбор вариантов, предусмотренных в программе, был совершенно произвольным и определялся исключительно целями демонстрации. В частности, программа обнаруживает попытки обращения к недоступным областям памяти, предоставляя адреса виртуальной памяти, по которым производилось такое обращение.

Блок __finally восстанавливает состояние маски FP-исключений. Совершенно очевидно, что восстановление состояния маски в данном случае, когда процесс уже должен завершаться, особого значения не имеет, но эта методика пригодится нам впоследствии, когда мы будем использовать ее на стадии завершения выполнения потока. Вообще говоря, процесс должен восстанавливать и системные ресурсы, например, удалять временные файлы, освобождать ресурсы синхронизации (глава 8) и отменять блокирование файлов (главы 3 и 6). Функция фильтра представлена в программе 4.4.

Данный пример не иллюстрирует обработку исключений, которые могут возникать при распределении памяти; эти исключения мы начнем интенсивно использовать в главе 5.

Программа 4.3. Exception: обработка исключений и завершения выполнения

```
    #include "EvryThng.h"
    #include <float.h>

    DWORD Filter(LPEXCEPTION_POINTERS, LPDWORD);
    double x = 1.0, y = 0.0;

    int _tmain(int argc, LPTSTR argv[]) {
     DWORD ECatgry, i = 0, ix, iy = 0;
     LPDWORD pNull = NULL;
     BOOL Done = FALSE;
     DWORD FPOld, FPNew;
     FPOld = _controlfp(0, 0); /* Сохранить старую управляющую маску. */
     /* Разрешить FP-исключения. */
      FPNew = FPOld & ~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT | EM_ZERODIVIDE |
EM_DENORMAL | EM_INVALID);
     _controlfp(FPNew, MCW_EM);
     while (!Done) _try { /* Блок try-finally. */
      _tprintf(_T("Введите тип исключения: "));
      _tprintf(_T(" 1: Mem, 2: Int, 3: Flt 4: User 5: __leave "));
      _tscanf(_T("%d"), &i);
      __try { /* Блок try-except. */
       switch (i) {
       case 1: /* Исключение при обращении к памяти. */
        ix = *pNull;
        *pNull = 5;
        break;
         case 2: /* Исключение при выполнении арифметических операций над целыми
числами. */
        ix = ix / iy;
```

```
        break;
       case 3: /* FP-исключение. */
        x = x / y;
        _tprintf(_T("x = %20e\n"), x);
        break;
       case 4: /* Пользовательское исключение. */
        ReportException(_T("Пользовательское исключение"), 1);
        break;
       case 5: /* Использовать оператор _leave для завершения выполнения.*/
        __leave;
       default:
        Done = TRUE;
       }
      } /* Конец внутреннего блока __try. */
      __except(Filter(GetExceptionInformation(), &ECatgry)) {
       switch(ECatgry) {
       case 0:
        _tprintf(_T("Неизвестное исключение\n"));
        break;
       case 1:
        _tprintf(_T("Исключение при обращении к памяти\n"));
        continue;
       case 2:
         _tprintf(_T("Исключение при выполнении арифметических операций над целыми
числами \n"));
        break;
       case 3:
        _tprintf(_T("FP-исключение\n"));
        _clearfp();
        break;
       case 10:
        _tprintf(_T("Пользовательское исключение\n"));
        break;
       default:
        _tprintf(_T("Неизвестное исключение\n"));
        break;
       } /* Конец оператора switch. */
       _tprintf(_T("Конец обработчика\n"));
      }
      /* Конец блока try-except. */
     } /* Конец цикла while - ниже находится обработчик завершения. */
     __finally { /* Это часть цикла while. */
      _tprintf(_T("Аварийное завершение?: %d\n"),
      AbnormalTermination());
     }
     _controlfp(FPOld, 0xFFFFFFFF); /* Восстановить старую FP-маску.*/
     return 0;
    }
```

Программа 4.4 представляет функцию фильтра, используемую в программе 4.3. Эта функция просто проверяет и классифицирует различные возможные значения кодов исключений. В программном коде, размещенном на Web-сайте книги, проверяется каждое из возможных значений, в то время как приведенная ниже функция осуществляет проверку лишь тех из них, которые нужны для тестовой программы.

Программа 4.4. Функция Filter

```
    static DWORD Filter(LPEXCEPTION_POINTERS pExP, LPDWORD ECatgry)
    /* Классификация исключений и выбор соответствующего действия. */
    {
     DWORD ExCode, ReadWrite, VirtAddr;
     ExCode = pExP->ExceptionRecord->ExceptionCode;
     _tprintf(_T("Filter. ExCode:. %x\n"), ExCode);
     if ((0x20000000 & ExCode) != 0) { /* Пользовательское исключение. */
      *ECatgry =10;
      return EXCEPTION_EXECUTE_HANDLER;
     }
     switch (ExCode) {
     case EXCEPTION_ACCESS_VIOLATION:
      ReadWrite = /* Операция чтения или записи? */
       pExP->ExceptionRecord->ExceptionInformation[0];
      VirtAddr = /* Адрес сбоя в виртуальный памяти. */
       pExP->ExceptionRecord->ExceptionInformation [1];
       _tprintf(_T("Нарушение доступа. Чтение/запись: %d. Адрес: %x\n"), ReadWrite,
VirtAddr);
      *ECatgry = 1;
      return EXCEPTION_EXECUTE_HANDLER;
     case EXCEPTION_INT_DIVIDE_BY_ZERO:
     case EXCEPTION_INT_OVERFLOW:
      *ECatgry = 2;
      return EXCEPTION_EXECUTE_HANDLER;
     case EXCEPTION_FLT_DIVIDE_BY_ZERO:
     case EXCEPTION_FLT_OVERFLOW:
      _tprintf(_T("FP-исключение — слишком большое значение.\n"));
      *ECatgry = 3;
      _clearfp();
      return (DWORD)EXCEPTION_EXECUTE_HANDLER;
     default:
      *ECatgry = 0;
      return EXCEPTION_CONTINUE_SEARCH;
     }
    }
```

Обработчики управляющих сигналов консоли

Обработчики исключений могут реагировать на самые разнообразные события, но они не в состоянии обнаруживать такие ситуации, как выход пользователя из системы или нажатие комбинации клавиш <Ctrl+C> на клавиатуре с целью прекращения выполнения программы. Для обработки таких событий требуются обработчики управляющих сигналов консоли.

Функция SetConsoleCtrlHandler позволяет одной или нескольким указанным функциям выполняться в ответ на получение сигналов Ctrl-c, Ctrl-break или одного из трех других сигналов, связанных с консолью. Функция GenerateConsoleCtrlEvent, описанная в главе 6, также генерирует эти сигналы, а, кроме того, все эти сигналы могут посылаться другим процессам, совместно использующим ту же консоль. Обработчиками сигналов являются указанные пользователем функции, которые возвращают булевские значения и принимают единственный аргумент типа DWORD, идентифицирующий фактический сигнал.

С одним сигналом могут быть ассоциированы несколько обработчиков, причем обработчики можно добавлять и удалять. Функция, которая используется для добавления и удаления обработчиков, имеет следующий вид:

```
BOOL SetConsoleCtrlHandler(PHANDLER_ROUTINE HandlerRoutine, BOOL Add)
```

Значению флага Add, равному TRUE, соответствует добавление процедуры обработчика, в противном случае происходит удаление процедуры из списка процедур обработки управляющих сигналов консоли. Заметьте, что тип сигнала при вызове функции не конкретизируется. Тестирование с целью проверки того, какой именно сигнал получен, должен выполнять сам обработчик.

Процедура обработчика возвращает булевское значение и принимает единственный параметр типа DWORD, идентифицирующий фактический сигнал. Использованное в объявлении имя обработчика (HandlerRoutine) является заменителем, и программист может выбирать его по своему усмотрению.

Ниже приводятся дополнительные полезные сведения, касающиеся использования обработчиков управляющих сигналов консоли.

• Если значение параметра HandlerRoutine равно NULL, а параметра Add — TRUE, то сигналы Ctrl-c будут игнорироваться.

• Если при вызове функции SetConsoleMode был задан параметр ENABLE_PROCESSED_INPUT (глава 2), то комбинация <Ctrl+C> будет обрабатываться не как сигнал, а как клавиатурный ввод.

• Процедура обработчика фактически выполняется как независимый поток (см. главу 7) внутри процесса. При этом выполнение основной программы, как показано в следующем примере, не приостанавливается.

• Формирование исключения в обработчике *не вызовет* исключения в потоки, выполнение которого было прервано, поскольку исключения применяются только к потокам, а не к процессу в целом. Если вы хотите организовать связь с прерванным потоком, используйте переменную, как в следующем примере, или метод синхронизации (глава 8).

Между исключениями и сигналами существует важное отличие. Сигналы применяются к процессу в целом, тогда как исключения — только к потоку, выполняющему код, в котором возникло исключение.

```
BOOL HandlerRoutine(DWORD dwCtrlType)
```

dwCtrlType идентифицирует фактический сигнал (или *событие)* и может принимать одно из следующих пяти значений:

1. CTRL_C_EVENT указывает на то, что комбинация <Ctrl+C> должна восприниматься как клавиатурный ввод.
2. CTRL_CLOSE_EVENT указывает на закрытие окна консоли.
3. CTRL_BREAK_EVENT указывает на сигнал Ctrl-break.
4. CTRL_LOGOFF_EVENT указывает на выход пользователя из системы.
5. CTRL_SHUTDOWN_EVENT указывает на завершение работы системы.

Обработчик сигналов может выполнять операции по "уборке мусора" точно так же, как это делают обработчики исключений и завершения. В случае успешной обработки сигнала обработчик должен вернуть значение TRUE. Если обработчик возвращает значение FALSE, выполняется следующая функция обработчика из числа тех, что указаны в списке. Обработчики сигналов выполняются в порядке, обратном порядку их установки, так что первым будет выполняться самый последний из установленных обработчиков, а системный обработчик будет выполняться самым последним.

Пример: обработчик управляющих сигналов консоли

В программе 4.5 организован бесконечный цикл, в котором каждые 5 секунд вызывается функция Beep, подающая звуковой сигнал. Пользователь может завершить выполнение программы, нажав комбинацию клавиш <Ctrl+C> или закрыв консоль. Процедура обработчика выводит на экран сообщение, выжидает 10 секунд, после чего, казалось бы, выполнение программы должно завершиться с возвратом значения TRUE. Однако в действительности основная программа обнаруживает флаг Exit и останавливает процесс. Это демонстрирует параллельную природу выполнения процедуры обработчика; заметьте, что объем выходной информации обработчика сигналов зависит от временных характеристик сигнала. Обработчики управляющих сигналов консоли будут использоваться также в примерах, приводимых в следующих главах.

Обратите внимание на использование макроса WINAPI; он применяется к пользовательским функциям, передаваемым в качестве аргументов функциям Windows, чтобы гарантировать выполнение соответствующих соглашений о вызовах. Этот макрос определен в заголовочном файле Microsoft C WTYPES.H.

Программа 4.5. Ctrlc: программа обработки сигналов

```
    /* Глава 4. Ctrlc.c */
    /* Перехватчик событий консоли. */
    #include "EvryThng.h"

    static BOOL WINAPI Handler(DWORD CtrlEvent); /* См. WTYPES.H. */
    volatile static BOOL Exit = FALSE;

    int _tmain(int argc, LPTSTR argv[])
     /* Периодическая подача звукового сигнала до поступления сигнала о прекращении
выполнения. */
    {
     /* Добавить обработчик событий. */
       if   (!SetConsoleCtrlHandler(Handler,   TRUE))   ReportError(_T("Ошибка   при
установке обработчика событий."), 1, TRUE);
     while (!Exit) {
      Sleep (5000); /* Подача звукового сигнала каждые 5 секунд. */
      Beep(1000 /* Частота. */, 250 /* Длительность. */);
     }
     _tprintf(_T("Прекращение выполнения программы по требованию.\n"));
     return 0;
    }

    BOOL WINAPI Handler (DWORD CtrlEvent) {
     Exit = TRUE;
     switch (CntrlEvent) {
      /*  Увидите  ли  вы  второе  сообщения  обработчика,  зависит  от  соотношения
временных параметров. */
     case CTRL_C_EVENT:
         _tprintf(_T("Получен   сигнал   Ctrl-c.   Выход   из   обработчика   через   10
секунд.\n"));
      Sleep(4000); /* Уменьшите это значение, чтобы получить другой эффект. */
      _tprintf(_T("Выход из обработчика через 6 секунд.\n"));
      Sleep(6000); /* Попробуйте уменьшить и это значение. */
```

```
      return TRUE; /* TRUE указывает на успешную обработку сигнала. */
     case CTRL_CLOSE_EVENT:
      _tprintf(_T("Выход из обработчика через 10 секунд.\n"));
      Sleep(4000);
      _tprintf(_T ("Выход из обработчика через 6 секунд.\n"));
      Sleep (6000); /* Попробуйте уменьшить и это значение. */
       return TRUE; /* Попробуйте возвратить FALSE. Приводит ли это к изменению
поведения программы? */
     default:
        _tprintf(_T("Событие: %d. Выход из обработчика через 10 секунд.\n"),
CntrlEvent);
      Sleep(4000);
      _tprintf(_T("Выход из обработчика через 6 секунд.\n"));
      Sleep(6000);
      return TRUE;
     }
    }
```

Векторная обработка исключений

Функции обработки исключений можно непосредственно связывать с исключениями, точно так же, как обработчики управляющих сигналов консоли можно связывать с управляющими событиями консоли. В этом случае, если возникает исключение, то первыми, еще до того, как система начнет разворачивать стек в поиске структурных обработчиков исключений, будут вызываться *векторные обработчики исключений* (vectored exception handlers). При этом никакие ключевые слова, аналогичные __try или __catch, не требуются. Такая возможность предоставляется только в Windows XP и Windows Server 2003.

Работа с векторными обработчиками исключений (Vectored Exception Handling, VEH) напоминает работу с обработчиками управляющих сигналов консоли, хотя детали и отличаются. Для добавления, или регистрации, обработчика служит функция AddVectoredExceptionHandler.

```
PVOID AddVectoredExceptionHandler(ULONG FirstHandler,
PVECTORED_EXCEPTION_HANDLER VectoredHandler)
```

Обработчики можно связывать в цепочки, поэтому первый параметр First-Handler указывает, что при возникновении исключения обработчик должен вызываться либо первым (ненулевое значение), либо последним (нулевое значение). Последующие вызовы функции AddVectoredExceptionHandler могут изменить этот порядок. Например, если добавляются два обработчика, причем для каждого из них задаются нулевые значения параметра FirstHandler, то они будут вызываться в том порядке, в котором добавлялись.

Функция RemoveVectoredExceptionHandler, прекращающая регистрацию векторного обработчика исключений, требует задания единственного параметра, адреса обработчика, и в случае успешного выполнения возвращает ненулевое значение.

Функция AddVectoredExceptionHandler в случае успешного выполнения возвращает адрес обработчика исключений, т.е. VectoredHandler. Возвращаемое значение NULL указывает на неудачное завершение выполнения функции.

VectorHandler — это указатель на функцию обработчика, которая имеет следующий прототип:

```
LONG WINAPI VectoredHandler(PEXCEPTION_POINTERS ExceptionInfo)
```

PEXCEPTION_POINTERS — адрес структуры EXCEPTION_POINTERS, которая содержит как информацию, зависящую от типа процессора, так и информацию общего характера. Это та же структура, которую возвращает функция GetExceptionInformation и которая уже использовалась нами в программе 4.4.

От функции VEH-обработчика требуется, чтобы она выполнялась быстро и никогда не получала доступа к объектам синхронизации, таким как мьютекс (см. главу 8). В большинстве случаев VEH-обработчики просто обращаются к структуре исключения, выполняют некоторую минимальную обработку (например, устанавливают флаг) и осуществляют возврат. Возможны два возвращаемых значения, с которыми мы уже встречались при обсуждении SEH-обработчиков.

1. EXCEPTION_CONTINUE_EXECUTION — обработчики далее не выполняются, обработка средствами SEH не производится, и управление передается в ту точку программы, в которой возникло исключение. Как и в случае SEH, это оказывается возможным не всегда.

2. EXCEPTION_CONTINUE_SEARCH — выполняется следующий VEH-обработчик, если

таковой имеется. Если обработчиков больше нет, разворачивается стек для поиска SEH-обработчиков.

В упражнении 4.9 вам предлагается добавить VEH в программы 4.3 и 4.4.

Резюме

Структурная обработка исключений в Windows предоставляет в распоряжение разработчиков механизм повышения надежности, благодаря которому С-программы могут адекватно реагировать на ошибки и исключения и восстанавливаться после возникновения сбоев в процессе выполнения. Методы обработки исключений отличаются высокой эффективностью, и их применение делает структуру программ более понятной, что облегчает их сопровождение и улучшает их качественные характеристики. В большинстве других языков и ОС также реализованы аналогичные подходы, однако решение Windows обеспечивает возможность точного анализа природы возникающих исключений.

Обработчики управляющих сигналов консоли позволяют реагировать на внешние события, наступление которых не сопровождается генерацией исключений. Векторная обработка исключений является новейшим средством, обеспечивающим выполнение соответствующих функций еще до того, как начнется выполнение SEH-процедур. Механизм VEH аналогичен обычному механизму векторных прерываний.

В следующих главах

Функция ReportException, а также обработчики исключений и завершения будут неоднократно использоваться в последующих примерах, когда в этом возникнет необходимость. Глава 5 посвящена вопросам управления памятью, а в приведенных в ней в качестве примера программах для обнаружения ошибок, которые могут возникать в процессе распределения памяти, используется механизм SEH.

Упражнения

4.1. Расширьте возможности программы 4.2 путем предоставления при каждом вызове функции ReportException достаточно большого объема информации, чтобы обработчик исключений в своих сообщениях указывал точную природу возникающих ошибок и удалял выходные файлы, если их содержимое оказывается незначащим.

4.2. Расширьте возможности программы 4.3 за счет генерации таких исключений, связанных с нарушениями доступа к памяти, как выход индекса за пределы допустимого диапазона, а также исключений, обусловленных сбоями при выполнении арифметических операций, и других FP-исключений, не предусмотренных в программе 4.3.

4.3. Дополните программу 4.3 таким образом, чтобы она выводила на печать фактическое значение FP-маски после разрешения исключений. Все ли исключения оказались действительно разрешенными? Объясните результаты.

4.4. Какие значения вы в действительности получаете после возникновения таких FP-исключений, как деление на ноль? Можете ли вы установить результат в функции фильтра, как это пытается делать программа 4.3?

4.5. Что произойдет при выполнении программы 4.3, если не сбросить FP-исключение? Объясните результат. *Подсказка.* Запросите дополнительное исключение после возникновения FP-исключения.

4.6. Расширьте возможности программы 4.5 таким образом, чтобы процедура обработчика формировала исключение, а не возврат из функции. Объясните полученные результаты.

4.7. Расширьте возможности программы 4.5 таким образом, чтобы она могла обрабатывать сигналы, указывающие на выход пользователя из системы и завершение работы системы.

4.8. Экспериментальным путем убедитесь в том, что процедура обработчика в программе 4.5 выполняется параллельно с основной программой.

4.9. Усовершенствуйте программы 4.3 и 4.4. В частности, организуйте обработку арифметических и FP-исключений до активизации SEH.

ГЛАВА 5
Управление памятью, отображение файлов и библиотеки DLL

Управление динамической памятью в той или иной форме требуется в большинстве программ. Необходимость в этом возникает всякий раз, когда требуется создавать структуры данных, размер которых не может быть определен заранее на стадии создания программы. Типичными примерами динамических структур данных могут служить деревья поиска, таблицы имен и связанные списки.

В Windows предусмотрены гибкие механизмы управления динамической памятью программы. Кроме того, Windows предоставляет средства отображения файлов, которые позволяют ассоциировать файл непосредственно с виртуальным адресным пространством процесса, благодаря чему ОС может управлять любыми перемещениями данных между файлом и памятью, так что программисту вообще не приходится иметь дело с функциями ReadFile, WriteFile, SetFilePointer и другими функциями ввода/вывода.

В случае использования отображения файлов программе удобно сохранять внутренние динамические структуры данных в виде постоянно существующих файлов, а все алгоритмы обработки применять к создаваемой в памяти копии файла. Более того, отображение файлов может значительно ускорить последовательную обработку файлов и предоставляет механизм, обеспечивающий совместное использование областей памяти одновременно несколькими процессами.

Важным специальным случаем отображения файлов и разделения памяти являются динамически компонуемые библиотеки (dynamic linked libraries, DLL), обеспечивающие возможность отображения файлов (обычно, когда они используются только для чтения) на адресное пространство процесса для их выполнения.

В этой главе описывается система управления памятью и функции отображения файлов Windows, что иллюстрируется целым рядом примеров их использования, а также обсуждаются явно и неявно связанные библиотеки DLL.

Архитектура системы управления памятью в Win32 и Win64

Win32 (в данном случае различия между Win32 и Win64 становятся существенными) — это API 32-разрядных ОС семейства Windows. "32-разрядность" проявляет себя при адресации памяти тем, что указатели (LPSTR, LPDWORD и так далее) являются 4-байтовыми (32-битовыми) объектами. Win64 API предоставляет виртуальное адресное пространство гораздо большего объема, и 64-битовые указатели являются естественным результатом эволюции Win32. Тем не менее, о переносимости приложений на платформу Win64 необходимо заботиться отдельно. Настоящее обсуждение будет относиться только к Win32; вопросы миграции приложений на платформу Win64 обсуждаются в главе 16, где также приводятся ссылки на соответствующие источники информации.

Далее, в рамках Win32 у каждого процесса имеется собственное виртуальное адресное пространство объемом 4 Гбайт (*232* байт). Разумеется, объем виртуального адресного пространства в Win64 гораздо больше. По крайней мере, половину указанного пространства (2-3 Гбайт; расширение до 3 Гбайт должно производиться во время загрузки) Win32 делает доступной для процесса. Оставшаяся часть виртуального адресного пространства выделяется для совместно используемых данных и кода, системного кода, драйверов и так далее.

Хотя детали описанного распределения памяти и заслуживают интереса, здесь они обсуждаться не будут; прикладные программы используют абстрактные модели памяти, предоставляемые API. С точки зрения программиста ОС просто предоставляет адресное пространство большого объема для размещения кода, данных и других ресурсов. В этой главе мы сосредоточим свое внимание на использовании средств управления памятью в Windows, не заботясь о том, как все это реализуется в ОС. Тем не менее, ниже приводится соответствующий краткий обзор.

Обзор методов управления памятью

Обо всех деталях отображения виртуальных адресов на физические адреса (virtual to physical memory mapping), механизмах страничной подкачки (page swapping) и замещения страниц по запросу (demand paging) и прочих моментах заботится ОС. Эти вопросы подробно обсуждаются в документации по ОС, а также в книге Соломона (Solomon) и Руссиновича (Russinovich) *Inside Windows2000.* Краткое изложение наиболее существенных сведений приводится ниже:

• Система может располагать сравнительно небольшим объемом физической памяти; на практике для всех систем, кроме Windows XP, необходимый минимум составляет 128 Мбайт, однако в типичных случаях доступные объемы физической памяти оказываются намного большими.[21]

• Каждый отдельный процесс — а таких процессов, как пользовательских, так и системных, может выполняться одновременно несколько — имеет собственное виртуальное адресное пространство, объем которого может значительно превосходить объем доступного физического адресного пространства. Например, емкость виртуального адресного пространства объемом 1 Гбайт, относящегося к одному процессу, в восемь раз превышает емкость физической памяти объемом 128 Мбайт, и таких процессов может быть множество.

• ОС преобразует виртуальные адреса в физические адреса.

• Для большинства виртуальных страниц в физической памяти места не хватит, поэтому ОС имеет возможность реагировать на страничные ошибки (page faults), возникающие при попытках обращения к страницам, которые отсутствуют в памяти, и загружать данные с жесткого диска —

из системного файла подкачки (swap file) или из обычного файла. Будучи прозрачными для программиста, страничные ошибки отрицательно влияют на производительность, поэтому программы должны проектироваться таким образом, чтобы вероятность возникновения подобных ошибок была сведена к минимуму. Более подробное освещение этой темы, рассмотрение которой выходит за рамки данной книги, вы найдете в справочной документации по ОС.

На рис. 5.1 проиллюстрировано расположение уровней API управления памятью Windows поверх диспетчера виртуальной памяти (Virtual Memory Manager, VMM). API виртуальной памяти Windows (VirtualAlloc, VirtualFree, Virtual-Lock, VirtualUnlock и так далее) работает с целыми страницами. API кучи Windows управляет блоками памяти, размер которых определяется пользователем.

Мы не будем останавливаться на топологии адресного пространства виртуальной памяти, поскольку она не имеет непосредственного отношения к API, различна в Windows 9x и Windows NT и в будущем может измениться. Соответствующая информация содержится в документации Microsoft.

Тем не менее, многим программистам хотелось бы знать больше о своей среде разработки. Начните исследование структуры памяти в вашей системе с вызова следующей функции:

```
VOID GetSystemInfo(LPSYSTEM_INFO lpSystemInfo)
```

Параметром этой функции служит адрес структуры PSYSTEM_INFO, в которой содержится информация относительно размера системной страницы, а также адресах физической памяти, доступных для приложений.

Куча

Windows поддерживает пулы памяти, называемые *кучами* (heaps). Процесс может иметь несколько куч, которые используются для распределения памяти.

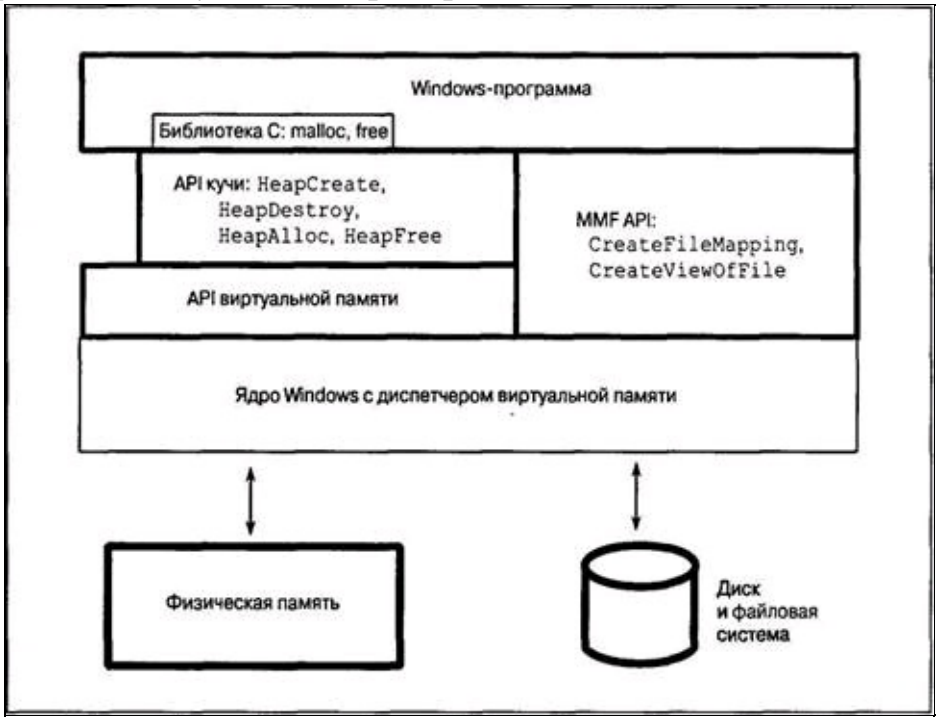


Рис. 5.1. Архитектура системы управления памятью Windows

Во многих случаях одной кучи вполне достаточно, но в силу ряда причин, о которых будет сказано ниже, иногда целесообразно иметь в своем распоряжении несколько куч. Если одной кучи вам хватает, можно обойтись использованием функций управления памятью, предоставляемых библиотекой C (malloc, free, calloc, realloc).

Кучи являются объектами Windows и, следовательно, имеют дескрипторы. Дескриптор кучи используется при распределении памяти. У каждого процесса имеется куча, заданная по умолчанию, которую использует функция malloc и для получения дескриптора которой используется следующая функция:

```
HANDLE GetProcessHeap(VOID)
```

Возвращаемое значение: дескриптор кучи процесса; в случае неуспешного завершения — NULL.

Заметьте, что для индикации неудачного завершения функции используется возвращаемое значение NULL, а не INVALID_HANDLE_VALUE, как в случае функции CreateFile.

Программа также может создать несколько различных куч. Иногда для размещения в памяти отдельных структур данных оказывается удобным, чтобы для каждой из них была предусмотрена своя куча. Использование независимых куч обеспечивает ряд преимуществ.

• **Отсутствие взаимной дискриминации между потоками.** Ни один из потоков не сможет получить больше памяти, чем распределено для ее кучи. В частности, так называемая утечка памяти (memory leak), возникающая в тех случаях, когда программа "забывает" своевременно освободить память, занятую элементами данных, необходимости в которых больше нет, будет влиять лишь на один поток процесса.[22]

• **Повышение производительности.** Предоставление собственной кучи каждого потока

уменьшает состязательность между ними, в результате чего общая производительность программы может значительно повыситься. См. главу 9.

• **Эффективность размещения данных.** Размещение элементов данных фиксированного размера в небольшой куче может оказаться гораздо более эффективным, чем размещение множества элементов самых различных размеров в одной большой куче. При этом также уменьшается фрагментация памяти. Кроме того, предоставление каждого потока собственной кучи существенно упрощает синхронизацию потоков, что приводит к дополнительному повышению производительности.

• **Эффективность освобождения памяти.** Области памяти, распределенные для кучи в целом и всех структур данных, которые она содержит, могут быть освобождены с помощью единственного вызова функции. Этот вызов также устранит отрицательные последствия утечки памяти, связанной с данной кучей.

• **Эффективность локализации обращений к памяти.** Сохранение структуры данных в небольшой куче гарантирует, что для всех элементов данных потребуется сравнительно небольшое количество страниц, а это может уменьшить вероятность возникновения ошибок страниц в процессе обработки элементов структур данных.

Ценность указанных преимуществ может варьироваться в зависимости от приложения, и многие программисты ограничиваются использованием только кучи процесса, для управления которой используют функции библиотеки C. Однако такой выбор лишает программу возможности воспользоваться способностью функций управления памятью Windows генерировать исключения (обсуждается при рассмотрении функций). В любом случае для создания и уничтожения куч применяются две функции, описания которых приводятся ниже.[23]

Начальный размер кучи, устанавливаемый параметром dwInitialSize (который может быть нулевым), всегда округляется до величины, кратной размеру страницы, и определяет объем физической памяти (в *файле подкачки),* который *передается* (commit) в распоряжение кучи (для последующего распределения памяти по запросам) первоначально, а не в ответ на запросы распределения (allocation) памяти из кучи. Когда программа исчерпывает первоначальный размер кучи, куче автоматически передаются дополнительные страницы памяти вплоть до пор, пока она не достигнет установленного для нее максимального размера. Поскольку файл подкачки является ограниченным ресурсом, рекомендуется откладывать передачу памяти куче на более поздний срок, если только заранее не известно, какой размер кучи потребуется. Максимально допустимый размер кучи при ее увеличении в результате динамического расширения определяется значением параметра dwMaximumSize (если оно ненулевое). Рост куч процессов, заданных по умолчанию, также осуществляется динамическим путем.

```
HANDLE HeapCreate(DWORD flOptions, SIZE_T dwInitialSize, SIZE_T
dwMaximumSize)
```

Возвращаемое значение: дескриптор кучи; в случае неудачного завершения — NULL.

Типом данных обоих упомянутых полей, связанных с размерами кучи, является не DWORD, а SIZE_T. Тип данных SIZE_T определяется как 32– или 64-битовое целое число без знака, в зависимости от флагов компилятора (_WIN32 или _WIN64). Этот тип данных был введен специально для того, чтобы обеспечить возможность миграции приложений Win64 (см. главу 16), и охватывает весь диапазон 32– и 64-битовых указателей. Вариантом этого типа данных для чисел со знаком является тип SSIZE_T).

flOptions — этот параметр может объединять следующие два флага:

• HEAP_GENERATE_EXCEPTIONS: в случае ошибки при распределении памяти вместо

возврата значения NULL генерируется исключение, которое должно быть обработано средствами SEH (см. главу 4). Если установлен этот флаг, то такие исключения при сбоях будет возбуждаться не самой функцией HeapCreate, а такими функциями, как HeapAlloc, к рассмотрению которых мы вскоре перейдем.

• HEAP_NO SERIALIZE: при определенных обстоятельствах, о которых сказано ниже, установка этого флага может привести к незначительному повышению производительности.

Существуют другие важные моменты, связанные с параметром dwMaximumSize.

• Если параметр dwMaximumSize имеет ненулевое значение, то виртуальное адресное пространство резервируется в соответствии с этим значением, даже если первоначально не все оно передается в распоряжение кучи. Это значение определяет максимальный размер кучи, о котором в этом случае говорят как о *нерастущем* (nongrowable). Данный параметр ограничивает размер кучи, чтобы, например, обеспечить отсутствие дискриминации между потоками, о чем говорилось выше.

• Если же значение dwMaximumSize равно 0, то куча может *расти* (grow), превышая предел, установленный начальным размером, и в этом случае максимальный размер кучи ограничивается лишь объемом доступного виртуального адресного пространства, не распределенного в данный момент для других куч и файла подкачки.

Заметьте, что кучи не имеют атрибутов защиты, поскольку доступ к ним извне процесса невозможен. В то же время, для объектов отображения файлов, описанных далее в этой главе, защита предусмотрена (глава 15), так как они могут совместно использоваться несколькими процессами.

Для уничтожения объекта кучи используется функция HeapDestroy. Она также может служить примером исключения из общих правил, в данном случае — правила, согласно которому для удаления ненужных дескрипторов любого типа используется функция CloseHandle.

```
BOOL HeapDestroy(HANDLE hHeap)
```

Параметр hHeap должен указывать на кучу, созданную посредством вызова функции HeapCreate. Будьте внимательны и следите за тем, чтобы случайно не уничтожить кучу процесса, заданную по умолчанию (дескриптор которой получают с помощью функции GetProcessHeap). В результате уничтожения кучи освобождается область виртуального адресного пространства и физическая область сохранения файла подкачки. Разумеется, правильно спроектированная программа должна уничтожать кучи, необходимости в которых больше нет.

Помимо всего прочего, уничтожение кучи позволяет быстро освободить память, занимаемую структурами данных, избавляя вас от необходимости отдельного уничтожения каждой из структур, однако экземпляры объектов C++ уничтожены не будут, поскольку их деструкторы при этом не вызываются. Применение операции уничтожения кучи имеет следующие положительные стороны:

1. Отпадает необходимость в написании программного кода, обеспечивающего обход структур данных.

2. Отпадает необходимость в освобождении памяти, занимаемой каждым из элементов, по отдельности.

3. Система не затрачивает время на обслуживание кучи, поскольку отмена распределения памяти для всех элементов структуры данных осуществляется посредством единственного вызова функции.

Функции библиотеки C используют только одну кучу. В силу этого иметь дело с чем-либо, напоминающим дескрипторы куч Windows, в данном случае не приходится.

В UNIX адресное пространство процесса может быть увеличено с помощью функции sbrk, однако эта функция не является диспетчером памяти общего назначения.

При неудачных попытках распределения памяти в UNIX сигналы не генерируются, поэтому в программах должна быть предусмотрена явная проверка значений возвращаемых указателей.

Управление памятью кучи

Для получения блока памяти из кучи следует указать дескриптор области памяти кучи, размер блока и некоторые флаги.

```
LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes)
```

Возвращаемое значение: в случае успешного выполнения — указатель на распределенный блок памяти, иначе — NULL (если только не была указана генерация исключения).

Параметры

hHeap — дескриптор кучи, из которой должен быть распределен блок памяти. Этот дескриптор должен быть предоставлен либо функцией GetProcessHeap, либо функцией HeapCreate.

dwFlags — может объединять следующие флаги:

• HEAP_GENERATE_EXCEPTIONS и HEAP_NO_SERIALIZE: эти флаги имеют тот же смысл, что и в случае функции HeapCreate. Первый флаг игнорируется, если он был установлен функцией кучи HeapCreate, но активизирует исключения для каждого отдельного вызова функции HeapAlloc, даже если функцией HeapCreate флаг HEAP_GENERATE_EXCEPTIONS и не был задан. При распределении памяти из кучи процесса второй флаг использовать не следует.

• HEAP_ZERO_MEMORY: этот флаг указывает, что распределенная память будет инициализирована значениями 0; если этот флаг не установлен, содержимое памяти является неопределенным.

dwBytes — размер блока памяти, который должен быть распределен. Для нерастущих куч значение этого параметра не должно превышать 0x7FFF8 (приблизительно 0,5 Мбайт).

Примечание

Как только функция HeapAlloc вернула указатель, вы можете использовать его самым обычным способом; ссылаться после этого на его кучу нет никакой необходимости. Заметьте, что тип данных LPVOID может представлять либо 32-битовый, либо 64-битовый указатель.

Для освобождения блока памяти, распределенного из кучи достаточно вызвать следующую функцию:

```
BOOL HeapFree(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem)
```

dwFlags — значениями этого параметра должны быть 0 или HEAP_NO_SERIALIZE. Значением параметра lpMem должно быть значение, возвращенное функциями HeapAlloc или HeapReAlloc (описана ниже), а дескриптор hHeap должен быть дескриптором кучи, которой принадлежит освобождаемый блок памяти, указываемый lpMem.

Для повторного распределения блоков памяти с целью изменения их размера используется следующая функция:

```
LPVOID HeapReAlloc(HANDLE hHeap, DWORD dwFlags, LPVOID lpMem, SIZE_T dwBytes)
```

Возвращаемое значение: в случае успешного выполнения — указатель на перераспределенный блок памяти; в противном случае функция возвращает NULL или вызывает исключение.

Параметры

• HEAP_GENERATE_EXCEPTIONS и HEAP_NO_SERIALIZE: это те же флаги, которые были описаны при рассмотрении функции HeapAlloc.

• HEAP_ZERO_MEMORY: нулями инициализируется лишь вновь распределенная память (когда значение параметра dwBytes превышает первоначальный размер блока). Содержимое исходного блока не изменяется.

• HEAP_REALLOC_IN_PLACE_ONLY: установка этого флага запрещает перемещение блока при перераспределении памяти. Если вы увеличиваете размер блока, адреса добавляемой памяти будут располагаться непосредственно вслед за адресами памяти, занимаемой существующим блоком.

lpMem — указывает на блок памяти, перераспределяемый из кучи hHeap.

dwBytes — размер нового блока памяти, который может быть как меньше, так и больше размера существующего блока.

Обычно возвращенный указатель имеет то же значение, что и указатель lpMem. В то же время, если блок перемещается (чтобы такое перемещение было разрешено, следует при вызове функции опустить флаг HEAP_REALLOC_IN_PLACE_ONLY), то возвращенное значение будет другим. Следите за своевременным изменением любых ссылок на блок. Независимо от того, перемещается блок или не перемещается, содержащиеся в нем данные остаются неизменными; в то же время, при уменьшении блока часть данных может теряться.

Размер распределенного блока памяти можно определить, вызвав функцию HeapSize (эту функцию следовало бы назвать BlockSize, поскольку о размере кучи она ничего не сообщает), используя в качестве параметров дескриптор кучи и указатель на блок.

```
DWORD HeapSize(HANDLE hHeap, DWORD dwFlags, LPCVOID lpMem)
```

Возвращаемое значение: в случае успешного выполнения — размер блока; иначе — ноль.

Флаг HEAP_NO_SERIALIZE

При вызове функций HeapCreate, HeapAlloc и HeapReAlloc можно указывать флаг HEAP_NO_SERIALIZE. Использование этого флага иногда обеспечивает незначительный выигрыш в производительности, поскольку во время обращения функции к куче взаимоисключающая блокировка к потокам в этом случае применяться не будет. Результаты простых тестов, в которых не делалось ничего, кроме распределения блоков памяти, показали повышение производительности примерно на 16 процентов. Этот флаг без какого бы то ни было риска можно использовать в следующих ситуациях:

• Программа не использует потоки (глава 7), или, точнее, процесс (глава 6) имеет только один поток. В данной главе этот флаг используется во всех примерах.

• Каждый поток имеет собственную кучу или набор куч, и никакой другой поток не имеет доступа к этой куче.

• Программа располагает собственным механизмом взаимоисключающей блокировки, который предотвращает одновременный доступ к куче сразу нескольких потоков, использующих функции HeapAlloc и HeapAlloc. Для этой цели также могут применяться функции HeapLock и HeapUnlock.

Флаг HEAP_GENERATE_EXCEPTIONS

Разрешение исключений вместо возврата значений NULL в случае сбоев при распределении памяти позволяет избавиться от утомительной необходимости тестирования результатов каждой попытки такого распределения. К тому же, обработчики исключений или завершения могут производить очистку памяти, которая к этому моменту была частично распределена. Эта методика применена в нескольких примерах.

Возможны два кода исключения:

1. STATUS_NO_MEMORY: это значение указывает на то, что системе не удалось создать блок запрошенного объема. Причинами этого могут быть фрагментация памяти, достижение нерастущей кучей максимально допустимого размера или исчерпание всей доступной памяти растущими кучами.

2. STATUS_ACCESS_VIOLATION: это значение указывает на повреждение кучи.

Одной из возможных причин этого может быть выполнение программой записи в память с выходом за границы распределенного блока.

Другие функции кучи

Функция HeapCompact пытается уплотнить, или *дефрагментировать,* смежные блоки в куче. Функция HeapValidate пытается обнаруживать повреждения кучи. Функция HeapWalk перечисляет блоки в куче, а функция GetProcessHeaps получает все действительные дескрипторы куч.

Функции HeapLock и HeapUnlock позволяют потоки сериализовать доступ к куче, о чем говорится в главе 8.

Имейте в виду, что эти функции не работают под управлением Windows 9x или Windows CE. Кроме того, имеются некоторые вышедшие из употребления функции, которые использовались ранее для совместимости с 16-битовыми системами. Мы упомянули об этих функциях лишь для того, чтобы лишний раз подчеркнуть тот факт, что многие функции продолжают поддерживаться, хотя никакой необходимости в них больше нет.

Резюме: управление кучами

Обычная процедура использования куч не представляет никаких сложностей:

1. Получите дескриптор кучи, воспользовавшись одной из функций CreateHeap или GetProcessHeap.

2. Распределите блоки из кучи, используя функцию HeapAlloc.

3. В случае необходимости освободите все или только некоторые блоки при помощи функции HeapFree.

4. Уничтожьте кучу и закройте ее дескриптор при помощи функции HeapDestroy.

Этот процесс иллюстрируют рис. 5.2 и программа 5.2.

В отсутствие необходимости создания отдельных куч или генерации исключений программисты, которые привыкли использовать функции управления памятью из библиотеки C, могут использовать их и далее. При этом, если речь идет о куче процесса, функция malloc эквивалентна функции HeapAlloc, функция realloc — функции HeapReAlloc, а функция free — функции HeapFree. Функция calloc распределяет память и инициализирует объекты, и ее поведение легко эмулируется функцией HeapAlloc. Эквивалент функции HeapSize в библиотеке C отсутствует.

Пример: сортировка файлов с использованием бинарного дерева поиска

Распространенными динамическими структурами данных, требующими управления памятью, являются деревья поиска. Деревья поиска предоставляют удобный способ сопровождения коллекций записей, дополнительным преимуществом которого является возможность применения чрезвычайно эффективных алгоритмов обхода узлов.

Программа sortBT (программа 5.1) реализует ограниченную версию UNIX-команды sort за счет создания бинарного дерева поиска с использованием двух куч. Ключи размещаются в *куче узлов* (node heap), представляющей дерево поиска. Каждый узел содержит левый и правый указатели, ключ и указатель на запись в *куче данных* (data heap). Заметьте, что куча узлов состоит из блоков фиксированного размера, тогда как куча данных содержит строки переменной длины. Наконец, отсортированный файл выводится путем обхода дерева.

В данном примере для использования в качестве ключа произвольно выбраны первые 8 байтов строки, а не целая строка. В двух других вариантах реализации сортировки, приведенных в настоящей главе (программы 5.4 и 5.5), выполняется сортировка индексированных файлов, а показатели производительности всех трех программ сравниваются в приложении B.

Последовательность операций по созданию куч и размещению блоков в памяти представлена на рис. 5.2. Программный код, приведенный справа, является *псевдокодом,* который отражает лишь наиболее существенные вызовы функций и аргументы. В виртуальном адресном пространстве, схематически изображенном слева, выделена память для трех куч, в каждой из которых имеются распределенные блоки. Программа 5.1 незначительно отличается от рисунка в том, что на рисунке, в отличие от программы, корень дерева размещен в куче процесса.

Примечание

Фактическое расположение куч и блоков в пределах куч зависит от варианта реализации Windows, а также от предыстории использования памяти процессом, включая рост кучи сверх ее начального размера. Кроме того, после увеличения размера растущей кучи с выходом за границы начальной области она может уже не занимать непрерывное адресное пространство. Наиболее оптимальная практика программирования состоит в том, чтобы не делать относительно фактической топологии распределения памяти никаких предположений; просто используйте функции управления памятью так, как это определяют правила работы с ними.

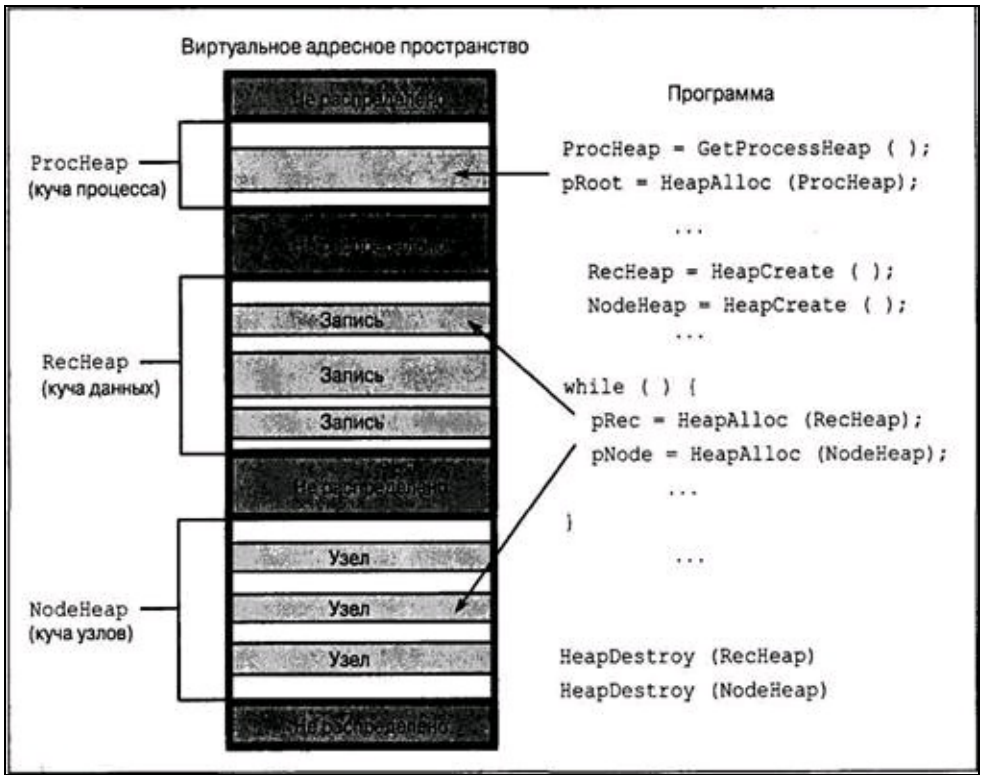


Рис. 5.2. Управление памятью при наличии нескольких куч

Программа 5.1 иллюстрирует некоторые методики, которые упрощают программу, но были бы невозможны при использовании одной только библиотеки C или же только кучи процесса.

• Элементы узлов имеют фиксированный размер и размещаются в собственной куче, тогда как элементы данных переменной длины размещаются в отдельной куче.

• Готовясь к сортировке очередного файла, программа уничтожает две кучи, а не освобождает память, занимаемую отдельными элементами.

• Ошибки при распределении памяти обрабатываются как исключения, вследствие чего отпадает необходимость в тестировании возвращаемых значений функциями для отслеживания нулевых указателей.

Если используется Windows, то сфера применимости таких программ, как программа 5.1, ограничивается файлами небольшого размера, поскольку в виртуальной памяти должны находиться целиком весь файл и копии ключей. Абсолютный верхний предел размера файла определяется объемом доступного виртуального адресного пространства (максимум 3 Гбайт); фактически достижимый предел оказывается еще меньшим. В случае Win64 ограничения подобного рода практически отсутствуют.

В программе 5.1 вызываются некоторые функции управления деревом: FillTree, InsertTree, Scan и TreeCompare. Все они представлены в программе 5.2.

В этой программе используются исключения кучи. Можно было бы поступить иначе, отказавшись от использования флага HEAP_GENERATE_EXCEPTIONS и отслеживая ошибки, возникающие при распределении памяти, явным образом.

Программа 5.1. sortBT: сортировка с использованием бинарного дерева поиска

```
/* Глава 5. Команда sortBT. Версия, использующая бинарное дерево поиска.*/
#include "EvryThng.h"
#define KEY_SIZE 8
```

```
typedef struct _TreeNode {/* Описание структуры узла. */
 struct _TreeNode *Left, *Right;
 TCHAR Key[KEY_SIZE];
 LPTSTR pData;
} TREENODE, *LPTNODE, **LPPTNODE;
#define NODE_SIZE sizeof(TREENODE)
#define NODE_HEAP_ISIZE 0x8000
#define DATA_HEAP_ISIZE 0x8000
#define MAX_DATA_LEN 0x1000
#define TKEY_SIZE KEY_SIZE * sizeof(TCHAR)

LPTNODE FillTree(HANDLE, HANDLE, HANDLE);
BOOL Scan(LPTNODE);
int KeyCompare (LPCTSTR, LPCTSTR); iFile;

BOOL InsertTree (LPPTNODE, LPTNODE);

int _tmain(int argc, LPTSTR argv[]) {
 HANDLE hIn, hNode = NULL, hData = NULL;
 LPTNODE pRoot;
 CHAR ErrorMessage[256];
 int iFirstFile = Options(argc, argv, _T("n"), &NoPrint, NULL);
 /* Обработать все файлы, указанные в командной строке. */
 for (iFile = iFirstFile; iFile < argc; iFile++) __try {
  /* Открыть входной файл. */
  hIn = CreateFile(argv[iFile], GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
  if (hIn == INVALID_HANDLE_VALUE) RaiseException(0, 0, 0, NULL);
  __try { /* Распределить две кучи. */
        hNode   =   HeapCreate(HEAP_GENERATE_EXCEPTIONS   |   HEAP_NO_SERIALIZE,
NODE_HEAP_ISIZE, 0);
        hData   =   HeapCreate(HEAP_GENERATE_EXCEPTIONS   |   HEAP_NO_SERIALIZE,
DATA_HEAP_ISIZE, 0);
    /* Обработать входной файл, создавая дерево. */
    pRoot = FillTree(hIn, hNode, hData);
    /* Отобразить дерево в порядке следования ключей. */
    _tprintf(_T("Сортируемый файл: %s\n"), argv [iFile]);
    Scan(pRoot);
  } _ finally { /* Кучи и дескрипторы файлов всегда закрываются.
    /* Уничтожить обе кучи и структуры данных. */
    if (hNode !=NULL) HeapDestroy (hNode);
    if (hNode != NULL) HeapDestroy (hData);
    hNode = NULL;
    hData = NULL;
    if (hIn != INVALID_HANDLE_VALUE) CloseHandle (hIn);
  }
 } /* Конец основного цикла обработки файлов и try-блока. */
 __except(EXCEPTION_EXECUTE_HANDLER) {
   _stprintf(ErrorMessage, _T("\n%s %s"), _T("sortBT, ошибка при обработке
файла:"), argv [iFile]);
  ReportError(ErrorMessage, 0, TRUE);
 }
 return 0;
}
```

В программе 5.2 представлены функции, которые фактически реализуют алгоритмы поиска с использованием бинарного дерева. Первая из этих функций, FillTree, распределяет память в обеих кучах. Вторая функция, KeyCompare, используется также в нескольких других программах в данной главе. Заметьте, что функции FillTree и KeyCompare используют обработчики завершения и исключений программы 5.1, которая вызывает эти функции. Таким образом,

ошибки распределения памяти будут обрабатываться основной программой, которая после этого продолжит свое выполнение, переходя к обработке следующего файла.

Программа 5.2. FillTree и другие функции управления деревом поиска

```
    LPTNODE FillTree(HANDLE hIn, HANDLE hNode, HANDLE hData)
    /* Заполнение дерева записями из входного файла. Используется обработчик
исключений вызывающей программы. */
    {
     LPTNODE pRoot = NULL, pNode;
     DWORD nRead, i;
     BOOL AtCR;
     TCHAR DataHold [MAX_DATA_LEN] ;
     LPTSTR pString;
     while (TRUE) {
      /* Разместить и инициализировать новый узел дерева. */
      pNode = HeapAlloc(hNode, HEAP_ZERO_MEMORY, NODE_SIZE);
      /* Считать ключ из следующей записи файла. */
        if  (!ReadFile(hIn,  pNode->Key,  TKEY_SIZE,  &nRead,  NULL)  ||  nRead  !=
TKEY_SIZE) return pRoot;
      AtCR = FALSE; /* Считать данные до конца строки. */
      for (i = 0; i < MAX_DATA_LEN; i++) {
       ReadFile(hIn, &DataHold [i], TSIZE, &nRead, NULL);
       if (AtCR && DataHold [i] == LF) break;
       AtCR = (DataHold [i] == CR);
      }
      DataHold[i - 1] = '\0';
      /* Объединить ключ и данные — вставить в дерево. */
       pString  =  HeapAlloc(hData,  HEAP_ZERO_MEMORY,  (SIZE_T)(KEY_SIZE  +  _tcslen
(DataHold) + 1) * TSIZE);
      memcpy(pString, pNode->Key, TKEY_SIZE);
      pString [KEY_SIZE] = '\0';
      _tcscat (pString, DataHold);
      pNode->pData = pString;
      InsertTree(&pRoot, pNode);
     } /* Конец цикла while (TRUE). */
     return NULL; /* Ошибка */
    }

    BOOL InsertTree(LPPTNODE ppRoot, LPTNODE pNode)
    /* Добавить в дерево одиночный узел, содержащий данные. */
    {
     if (*ppRoot == NULL) {
      *ppRoot = pNode;
      return TRUE;
     }
     /* Обратите внимание на рекурсивные вызовы InsertTree. */
     if (KeyCompare(pNode->Key, (*ppRoot)->Key) < 0) InsertTree(&((*ppRoot)->Left),
pNode);
      else InsertTree(&((*ppRoot)->Right), pNode);
    }

    static int KeyCompare(LPCTSTR pKey1, LPCTSTR pKey2)
    /* Сравнить две записи, состоящие из обобщенных символов. */
    {
     return _tcsncmp(pKey1, pKey2, KEY_SIZE);
    }
```

```
static BOOL Scan(LPTNODE pNode)
/* Рекурсивный просмотр и отображение содержимого бинарного дерева. */
{
 if (pNode == NULL) return TRUE;
 Scan(pNode->Left);
 _tprintf(_T ("%s\n"), pNode->pData);
 Scan(pNode->Right);
 return TRUE;
}
```

Примечание

Очевидно, что данную реализацию дерева поиска нельзя назвать самой эффективной, поскольку дереву поиска ничто не мешает стать несбалансированным. Разумеется, о балансировке дерева поиска следовало бы позаботиться, однако на организацию управления памятью в программе это никак не повлияет.

Отображение файлов

Динамическая память, распределенная в кучах, должна физически размещаться в файле подкачки. Управление перемещением страниц между физической памятью и файлом подкачки, а также отображением файла подкачки на виртуальное адресное пространство процесса осуществляется средствами ОС, ответственными за управление памятью. По завершении выполнения процесса физическое пространство в этом файле освобождается.

Те же функциональные возможности Windows, которые обеспечивают отображение файла подкачки, позволяют отображать и обычные файлы. Отображение файлов дает следующие преимущества:

• Отпадает необходимость в выполнении операций непосредственного файлового ввода/вывода (чтения и записи).

• Структуры данных, созданные в памяти, будут сохраняться в файле для последующего использования этой же или другими программами. Необходимо тщательно следить за правильностью использования указателей, что иллюстрируется в программе 5.5.

• Становится возможным применение удобных и эффективных алгоритмов, ориентированных на работу с файлами "в памяти" (in-memory files) (сортировка, деревья поиска, обработка строк и тому подобное), которые позволяют обрабатывать хранящиеся в файлах данные даже в тех случаях, когда размеры файлов значительно превышают доступный объем физической памяти. При больших размерах файлов особенности организации страничного обмена могут оказывать заметное влияние на производительность.

• В некоторых случаях значительно повышается эффективность обработки файлов.

• Исчезает необходимость в управлении буферами и манипулировании содержащимися в них данными файлов. Всю эту тяжелую работу выполняет ОС, причем делает она это в высшей степени эффективно и надежно.

• Обеспечивается возможность разделения памяти несколькими параллельно выполняющимися процессами (глава 6) за счет отображения на их виртуальные адресные пространства одного и того же обычного файла или файла подкачки (разделение памяти несколькими процессами является одной из основных причин использования объекта отображения файла подкачки).

• Отпадает необходимость в расходовании излишнего пространства файла подкачки.

ОС сама использует методы отображения файлов для реализации DLL, а также для загрузки и выполнения исполняемых (.EXE) файлов. Библиотеки DLL описаны в конце настоящей главы.

Объекты отображения файлов

Сначала необходимо создать для открытого файла *объект отображения файла* (file mapping object), у которого имеется дескриптор, а затем отобразить этот файл или только некоторую его часть на виртуальное адресное пространство процесса. Объектам отображения можно присваивать имена, по которым к ним смогут обращаться другие процессы, разделяющие память совместно с данным процессом. Кроме того, объекты отображения файлов имеют параметры размера и атрибуты защиты.

```
HANDLE CreateFileMapping(HANDLE hFile, LPSECURITY_ATTRIBUTES lpsa,
DWORD dwProtect, DWORD dwMaximumSizeHigh, DWORD dwMaximumSizeLow, LPCTSTR
lpMapName)
```

Возвращаемое значение: в случае успешного выполнения — дескриптор объекта

отображения файла, иначе — NULL.

Параметры

hFile — дескриптор открытого файла, атрибуты защиты которого совместимы с флагами защиты, указанными параметром dwProtect. Значение этого дескриптора (тип данных HANDLE), равное 0xFFFFFFFF (его эквивалент — символическая константа INVALID_HANDLE_VALUE), соответствует системному файлу подкачки, и его можно использовать для организации разделения памяти несколькими процессами без создания отдельного файла.

LPSECURITY_ATTRIBUTES — позволяет указать атрибуты защиты объекта отображения.

dwProtect — с помощью флагов, которые приводятся ниже, определяет возможности доступа к представлению файла при его отображении. Помимо упомянутых флагов предусмотрены дополнительные флаги, имеющие специальное назначение. Так, флаг SEC_IMAGE указывает на то, что открытый файл, на основе которого создается объект отображения, является исполняемым загрузочным модулем; для получения более подробной информации обратитесь к оперативной справочной документации.

• PAGE_READONLY: страницы в указанной области отображения доступны программе только для чтения; программа не может осуществлять в них запись или запускать на выполнение. Файл с дескриптором hFile должен быть открыт с правами доступа GENERIC_READ.

• PAGE_READWRITE: предоставляет полный доступ к объекту, если файл с дескриптором hFile был открыт с правами доступа GENERIC_READ и GENERIC_WRITE.

• PAGE_WRITECOPY: при изменении отображения файла его приватная (для данного процесса) копия записывается в файл подкачки, а не в исходный файл. Отладчики могут использовать этот флаг для установки точек прерывания в разделяемом коде.

dwMaximumSizeHigh и dwMaximumSizeLow — соответственно, старшая и младшая 32-битовые части значения максимального размера объекта отображения файла. Если оба эти параметры равны 0, используется текущий размер файла; в случае работы с файлом подкачки указание размера является обязательным. Если предполагается, что впоследствии файл может увеличиться, укажите его предполагаемый конечный размер, и, если это необходимо, этот размер будет сразу же установлен для файла. Не пытайтесь отображать область файла, лежащую за пределами указанного размера, поскольку размер объекта отображения расти не может.

lpMapName — указатель на строку, содержащую имя объекта отображения, которое другие процессы могут использовать для разделения объекта; имя объекта чувствительно к регистру. Если не предполагается разделение памяти, используйте для этого параметра значение NULL.

На возникновение ошибок указывает возвращение функцией значения NULL (а не INVALID_HANDLE_VALUE).

Дескриптор объекта отображения файла можно получить, указав имя существующего объекта отображения. Это имя должно совпадать с тем, которое было задано во время создания открываемого объекта отображения с помощью функции CreateFileMapping. Два процесса могут разделять память, разделяя отображение файла. При этом первый процесс создает именованный объект отображения, а второй открывает этот объект, используя его имя. Если объекта отображения с указанным именем не существует, попытка его открытия будет неудачной.

```
    HANDLE OpenFileMapping(DWORD dwDesiredAccess, BOOL bInheritHandle,
LPCTSTR lpMapName)
```

Возвращаемое значение: в случае успешного выполнения — дескриптор объекта отображения файла, иначе — NULL.

Параметр dwDesiredAccess использует тот же набор флагов, что и параметр dwProtect в функции CreateFileMapping. Указатель lpMapName должен указывать на строку с именем, совпадающим с тем, которое было задано при вызове функции CreateFileMapping. Дескриптор наследования (bInheritTable) рассматривается в главе 6.

Как несложно догадаться, для закрытия дескрипторов объектов отображения используется функция CloseHandle.

Отображение файла на адресное пространство процесса

Следующим шагом является распределение виртуального адресного пространства и отображение на него файла с использованием объекта отображения. С точки зрения программиста этот процесс распределения памяти аналогичен тому, который обсуждался при рассмотрении функции HeapAlloc, хотя и делает это намного грубее, оперируя более крупными блоками. В результате этого распределения возвращается указатель на распределенный блок, или представление файла (file view); различие состоит в том, что этот распределенный блок является отображением пользовательского файла, а не файла подкачки. Объект отображения файла играет ту же роль, что и куча в случае использования функции HeapAlloc.

```
LPVOID MapViewOfFile(HANDLE hMapObject, DWORD dwAccess, DWORD dwOffsetHigh, DWORD dwOffsetLow, SIZE_T cbMap)
```

Возвращаемое значение: В случае успешного выполнения — начальный адрес блока (представления файла), иначе — NULL.

Параметры

hMapObject — дескриптор объекта отображения файла, возвращенный функцией CreateFileMapping или OpenFileMapping.

dwAccess — этот параметр должен быть совместимым с разрешенными типами доступа к объекту отображения. Тремя возможными флаговыми значениями являются FILE_MAP_WRITE, FILE_MAP_READ и FILE_MAP_ALL_ACCESS. (Последний флаг является результатом применения поразрядной операции "или" к двум предыдущим флагам).

dwOffsetHigh и dwOffsetLow — соответственно, старшая и младшая 32-битовые части смещения начала отображаемого участка в файле. Значение этого начального адреса должно быть кратным 64 Кбайт. Чтобы начало отображаемого участка совпадало с началом файла, оба параметра следует задать равными 0.

cbMap — размер отображаемого участка файла в байтах. Если значение этого параметра установлено равным 0, то отображаться будет весь файл, существующий в момент вызова функции MapViewOfFile.

Функция MapViewOfFileEx аналогична функции MapViewOfFile, но дополнительно позволяет указать при вызове начальный адрес памяти для отображенного представления. Например, в качестве такого адреса может быть указан адрес массива в пространстве данных программы. В Windows, если затребованная область памяти уже используется для отображения, выполнение этой функции завершится с ошибкой.

Точно так же как память, распределенная из кучи, должна освобождаться при помощи функции HeapFree, необходимо отменять и отображение представления файла, которое больше не используется.

```
BOOL UnmapViewOfFile(LPVOID lpBaseAddress)
```

Взаимосвязь между адресным пространством процесса и отображаемым файлом проиллюстрирована на рис. 5.3.

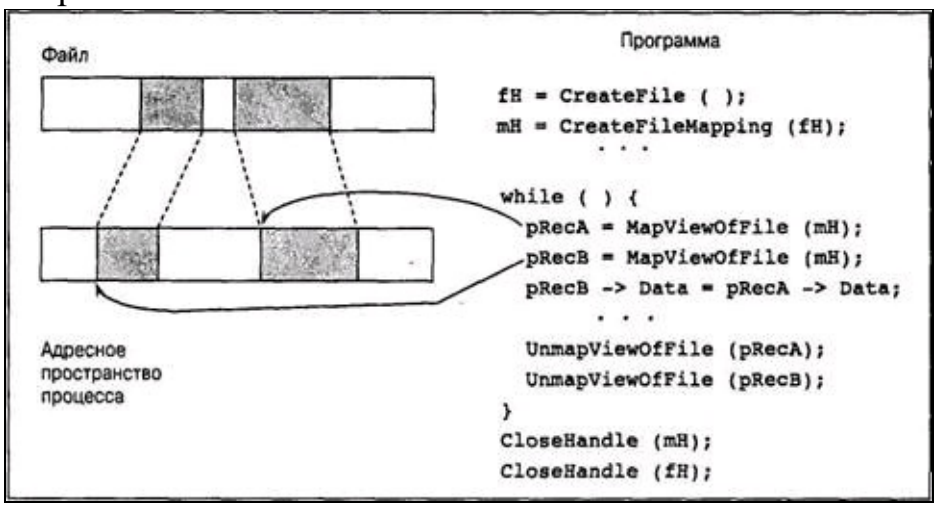


Рис. 5.3. Отображение представления файла на адресное пространство процесса

Вызов функции FlushViewOfFile вынуждает систему записать измененные страницы на диск. Как правило, процесс, получающий доступ к файлу через его отображение в памяти, и процесс, получающий доступ к файлу посредством обычных файловых операций ввода/вывода, будут "видеть" разные представления файла. Не решает эту проблему и выполнение файловых операций ввода/вывода без буферизации, так как представление отображаемого файла в памяти не записывается немедленно на диск.

В силу этого идея получения доступа к отображаемому файлу с помощью функций ReadFile и WriteFile не сулит ничего хорошего, поскольку согласованность данных при этом не гарантируется. С другой стороны, представления файла для процессов, получающих совместный доступ к нему через разделяемую память, будут согласованными. Если один процесс изменяет какой-либо участок памяти в области отображения файла, то другой процесс при получении доступа к соответствующему участку в своей области отображения файла получит измененные данные. Этот механизм проиллюстрирован на рис. 5.4, из которого следует, что согласованность отображенных представлений файла в двух процессах (PA и PB) действительно обеспечивается, поскольку виртуальным адресам данных в обоих процессах, несмотря на то, что эти адреса различны, соответствуют одни и те же участки физической памяти; Естественным образом связанная с этим тема синхронизации процессов обсуждается в главах 8—10.[24]

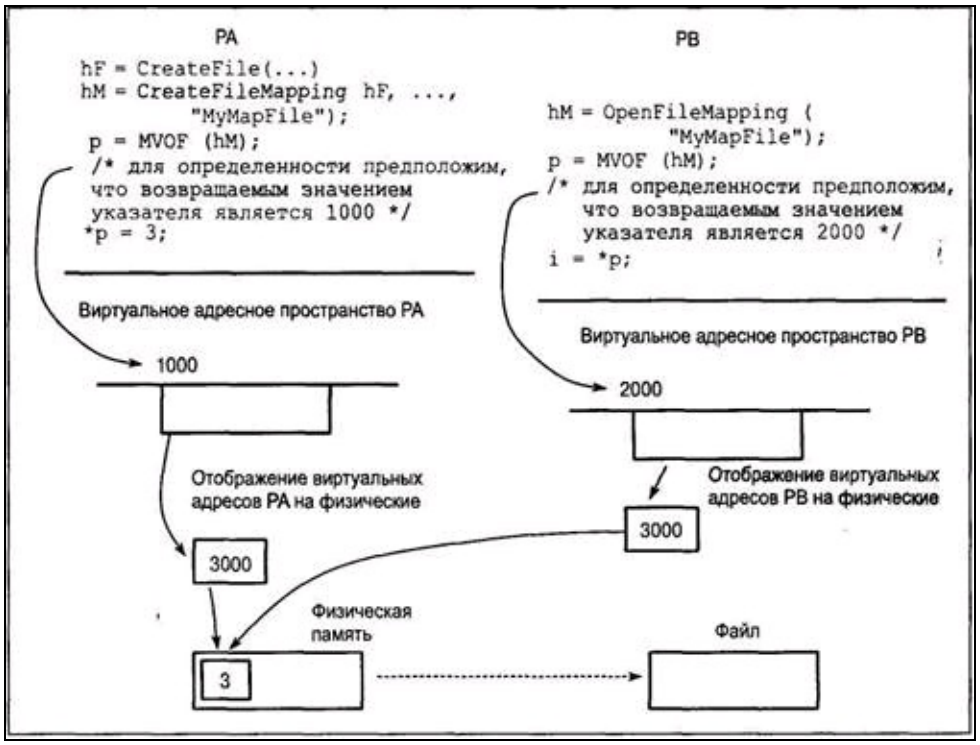


Рис. 5.4. Разделяемая память

В UNIX (выпуски SVR4 и 4.3+BSD) поддерживается функция mmap, аналогичная функции MapViewOfFile. В ее параметрах указывается та же информация, за исключением того, что объект отображения отсутствует.

Эквивалентом функции UnMapViewOfFile является функция munmap.

Для функций CreateFileMapping и OpenFileMapping эквиваленты отсутствуют. Любой обычный файл может непосредственно отображаться. В UNIX отображение файлов для разделения памяти не используется, и для этих целей предусмотрены специальные функции API, а именно, shmctl, shmat и shmdt.

Ограничения метода отображения файлов

Как уже отмечалось ранее, отображение файлов является весьма мощным и полезным средством. Существующая в Windows диспропорция между 64-битовой файловой системой и 32-битовой адресацией снижает ценность обеспечиваемых этим средством преимуществ; Win64 свободен от этих ограничений.

Основная проблема заключается в том, что при больших размерах файлов (в данном случае, свыше 2—3 Гбайт) отображение всего файла на пространство виртуальной памяти становится невозможным. Более того, не будут доступны даже все 3 Гбайт, поскольку часть виртуального адресного пространства распределяется для других целей, а суммарный объем доступных смежных блоков будет гораздо меньше теоретического максимума. Win64 в значительной степени снимает это ограничение.

При работе с большими файлами, для которых объект отображения не может быть создан целиком, необходимо предусматривать отдельный программный код, осуществляющий отображение и отмену отображения соответствующих участков файла по мере необходимости. По сложности реализации такая методика сопоставима с организацией управления буферами в памяти, хотя необходимость в выполнении явных операций чтения и записи в данном случае отсутствует.

Двумя другими существенными недостатками метода отображения файлов являются следующие:

• Размер объекта отображения не может увеличиваться. Создавая объект отображения, вы должны заранее определить, какой максимальный размер вам может понадобиться, но во многих случаях сделать это трудно или вообще невозможно.

• Невозможно распределить память в пределах области, занимаемой представлением объекта отображения, не создав для этого собственные функции управления памятью. Было бы очень удобно, если бы существовал способ задавать объект отображения файла и указатель, возвращаемый функцией MapViewOfFile, с последующим получением дескриптора кучи.

Резюме: отображение файлов

Ниже приведена стандартная последовательность действий, которые необходимо выполнять, если используется отображение файлов:

1. Откройте файл. Убедитесь в том, что он имеет права доступа GENERIC_READ.

2. В случае создания нового файла укажите его размер, используя для этого либо функцию CreateFileMapping (шаг 3), либо функцию SetFilePointer с последующим вызовом функции SetEndOfFile.

3. Отобразите файл при помощи функций CreateFileMapping или OpenFileMapping.

4. Создайте одно или несколько представлений объекта отображения файла с помощью функции MapViewOfFile.

5. Осуществляйте доступ к файлу через обращения к памяти. Для перехода к другим участкам отображаемого файла используйте функции UnmapViewOfFile и MapViewOfFile.

6. Завершив работу, вызовите последовательно функции UnmapViewOfFile, CloseHandle для закрытия дескриптора объекта отображения и CloseHandle для закрытия дескриптора файла.

Пример: последовательная обработка файлов с использованием метода отображения

Программа atou (программа 2.4) иллюстрирует последовательную обработку файлов на примере преобразования ASCII-файлов к кодировке Unicode, приводящего к удвоению размера файла. Этот случай является идеальным для применения отображения файлов, поскольку наиболее естественным способом указанного преобразования является посимвольная обработка данных без обращения к операциям файлового ввода/вывода. Программа 5.3 сначала просто отображает входной и выходной файлы в память, предварительно вычисляя размер выходного файла путем удвоения размера входного файла, а затем осуществляет требуемое посимвольное преобразование.

Этот пример отчетливо демонстрирует, как сложность процесса отображения файлов, выполнение которого необходимо для инициализации программы, компенсируется простотой результирующей обработки. Принимая во внимание, насколько просто выполняются обычные операции файлового ввода/вывода, применение более сложного метода могло бы показаться излишним, однако это с лихвой окупается выигрышем в производительности. В приложении В показано, что версия, использующая отображение файлов, в файловых системах NTFS работает значительно быстрее по сравнению с версиями, использующими обычные способы доступа к файлам, так что некоторое усложнение программы себя полностью оправдывает. Дополнительные результаты анализа производительности приведены на Web-сайте книги, поэтому ниже мы ограничимся лишь краткими выводами.

• Повышение производительности программ за счет использования отображения файлов наблюдается только в случае Windows NT и файловых систем NTFS.

• По сравнению с наилучшими из методик последовательной обработки файлов обеспечивается, по крайней мере, трехкратное повышение производительности .

• При работе с файлами большого размера преимущества в отношении производительности теряются. В нашем примере обычный последовательный просмотр файлов оказывается более предпочтительным, так как размер входного файла составляет около одной трети объема физической памяти. Снижение производительности метода отображения файлов в данном случае объясняется тем, что для входного файла требуется одна треть памяти, а для выходного файла, размер которого в два раза больше, — оставшиеся две трети, что заставляет нас сбрасывать отдельные части выходного файла на диск. Таким образом, в системе с объемом оперативной памяти 192 Мбайт ухудшение производительности метода отображения файлов будет наступать после достижения входными файлами размера 60 Мбайт. В большинстве случаев приходится иметь дело с файлами меньшего размера, в результате чего применение метода отображения файлов становится целесообразным.

В программе 5.3 представлена лишь функция Asc2UnMM. Основная программа совпадает с той, которая приведена в программе 2.4.

Программа 5.3. Asc2UnMM: преобразование файла с использованием метода отображения файлов

```
/* Глава 5. Asc2UnMM.c: Реализация, использующая отображение файлов. */
#include "EvryThng.h"
```

```
BOOL Asc2Un(LPCTSTR fin, LPCTSTR fOut, BOOL bFaillfExists) {
 HANDLE hIn, hOut, hInMap, hOutMap;
 LPSTR pIn, pInFile;
 LPWSTR pOut, pOutFile;
 DWORD FsLow, dwOut;
 /* Открыть и отобразить входной и выходной файлы. */
    hIn    =    CreateFile(fIn,    GENERIC_READ,    0,    NULL,    OPEN_EXISTING,
FILE_ATTRIBUTE_NORMAL, NULL);
 hInMap = CreateFileMapping(hIn, NULL, PAGE_READONLY, 0, 0, NULL);
 pInFile = MapViewOfFile(hInMap, FILE_MAP_READ, 0, 0, 0);
 dwOut = bFailIfExists ? CREATE NEW : CREATE ALWAYS;
  hOut  =  CreateFile(fOut,  GENERIC_READ  |  GENERIC_WRITE,  0,  NULL,  dwOut,
FILE_ATTRIBUTE_NORMAL, NULL);
 FsLow = GetFileSize (hIn, NULL); /* Установить размер отображения. */
 hOutMap = CreateFileMapping(hOut, NULL, PAGE_READWRITE, 0, 2* FsLow, NULL);
 pOutFile = MapViewOfFile(hOutMap, FILE_MAP_WRITE, 0, 0, (SIZE_T)(2 * FsLow));
 /* Преобразовать данные отображенного файла из ASCII в Unicode. */
 pIn = pInFile;
 pOut = pOutFile;
 while (pIn < pInFile + FsLow) {
  *pOut = (WCHAR) *pIn;
  pIn++;
  pOut++;
 }
 UnmapViewOfFile(pOutFile);
 UnmapViewOfFile(pInFile);
 CloseHandle(hOutMap);
 CloseHandle(hInMap);
 CloseHandle(hIn);
 CloseHandle(hOut);
 return TRUE;
}
```

Пример: сортировка отображенных файлов

Дополнительным преимуществом метода отображения файлов является то, что он допускает применение обычных алгоритмов обработки файлов в памяти компьютера. Так, сортировку данных в памяти осуществить гораздо легче, чем сортировку записей в файле.

Программа 5.4 предназначена для сортировки файлов с записями фиксированной длины. Данная программа, sortFL, аналогична программе 5.1 в том отношении, что предполагает наличие 8-байтового ключа сортировки в начале записи, но ограничивается записями фиксированной длины. В программе 5.5 этот недостаток будет устранен за счет некоторого усложнения программы.

Сортировку выполняет описанная в файле <stdlib.h> функция qsort, входящая в состав библиотеки C. Заметьте, что эта функции требует от программиста предоставления функции, осуществляющей сравнение записей, в качестве которой нами будет использована функция KeyCompare из программы 5.2.

Структура программы достаточно проста. Сначала на основе временной копии входного файла создается объект отображения файла, затем создается единое представление объекта отображения файла в памяти, и, наконец, вызывается функция qsort. При этом какие-либо операции файлового ввода/вывода отсутствуют. Отсортированный файл направляется далее на стандартный вывод, причем в конце отображения файла добавляется нулевой символ.

Программа 5.4. sortFL: сортировка файла с использованием его отображения в памяти

```
/* Глава 5. sortFL. Сортировка файлов. Записи имеют фиксированную длину.*/
/* Использование: sortFL файл */
#include "EvryThng.h"

typedef struct _RECORD {
 TCHAR Key[KEY_SIZE];
 TCHAR Data[DATALEN];
} RECORD;

#define RECSIZE sizeof(RECORD)

int _tmain(int argc, LPTSTR argv[]) {
 HANDLE hFile = INVALID_HANDLE_VALUE, hMap = NULL;
 LPVOID pFile = NULL;
 DWORD FsLow, Result = 2;
 TCHAR TempFile[MAX_PATH];
 LPTSTR pTFile;
  /* Создать имя временного файла, предназначенного для хранения копии
сортируемого файла, которая и подвергается сортировке. */
 /* Можно действовать и по-другому, оставив файл в качестве постоянно хранимой
сортируемой версии. */
 _stprintf(TempFile, _T("%s%s"), argv[1], _T(".tmp"));
 CopyFile(argv[1], TempFile, TRUE);
 Result = 1; /* Временный файл является вновь созданным и должен быть удален.
*/
 /* Отобразить временный файл и выполнить его сортировку в памяти. */
  hFile = CreateFile(TempFile, GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, 0, NULL);
 FsLow = GetFileSize(hFile, NULL);
```

```
    hMap = CreateFileMapping(hFile, NULL, PAGE_READWRITE, 0, FsLow + TSIZE, NULL);
    pFile = MapViewOfFile(hMap, FILE_MAP_ALL_ACCESS, 0, 0 /* FsLow + TSIZE */, 0);
    qsort(pFile, FsLow / RECSIZE, RECSIZE, KeyCompare);
    /* KeyCompare - как в программе 5.2. */
    /* Отобразить отсортированный файл. */
    pTFile = (LPTSTR)pFile;
    pTFile[FsLow/TSIZE] = '\0';
    _tprintf(_T("%s"), pFile);
    UnmapViewOfFile(pFile);
    CloseHandle(hMap);
    CloseHandle(hFile);
    DeleteFile(TempFile);
    return 0;
}
```

Описанный вариант реализации довольно прост, однако возможен и другой вариант, не требующий использования отображения файлов. Для этого достаточно распределить память, считать весь файл, выполнить его сортировку в памяти и записать на диск. По своей эффективности это решение, которое приведено на Web-сайте книги, не уступает программе 5.4, а нередко и превосходит ее, как показано в приложении В.

Базовые указатели

Как показали предыдущие примеры, во многих случаях метод отображения файлов является весьма удобным. Однако предположим, что в программе создается структура данных с указателями, ссылающимися на область отображения файла, и ожидается, что впоследствии к этому файлу будет производиться обращение. В этом случае указатели оказываются установленными относительно виртуального адреса, возвращенного функцией MapViewOfFile, и не будут иметь смысла при использовании представления объекта отображения в следующий раз. Решение состоит в том, чтобы использовать базовые указатели (based pointers), являющиеся фактически смещениями относительно другого указателя. Соответствующий синтаксис Microsoft C, доступный в Visual C++ и некоторых других системах, выглядит следующим образом:

```
тип _based (база) объявление
```

Ниже показаны два примера таких указателей.

```
LPTSTR pInFile = NULL;
DWORD _based (pInFile) *pSize;
TCHAR _based (pInFile) *pIn;
```

Обратите внимание на тот факт, что синтаксис требует использования символа *, хотя такая практика противоречит соглашениям Windows.

Пример: использование базовых указателей

Рассмотренные выше примеры относились к сортировке файлов в различных ситуациях. Вместе с тем, должно быть очевидным, что наша цель состояла не в обсуждении методик сортировки, а в демонстрации применения различных методов управления памятью. В программе 5.1 используется бинарное дерево поиска, которое уничтожается при переходе к сортировке очередного файла, тогда как в программе 5.4 сортируется массив фиксированных записей, отображенный в памяти компьютера. В приложении В представлены показатели производительности для различных вариантов реализации, включая и тот, который реализует программа 5.5.

Предположим, что необходимо обеспечить сопровождение постоянно существующего индексного файла, предоставляющего отсортированный ключ исходного файла. Могло бы показаться, что очевидным решением является отображение в памяти файла, содержащего постоянно хранимые индексы в виде дерева поиска, или его формы с отсортированным ключом. Однако это решение страдает серьезным недостатком. Все указатели дерева, сохраняемые в файле, являются заданными относительно адреса, возвращенного функцией MapViewOfFile. Когда программа будет запущена в следующий раз и создаст отображение файла, эти указатели окажутся бесполезными.

Программа 5.5, которая должна применяться совместно с программой 5.6, решает эту проблему, которая проявляется всякий раз, когда отображаются структуры данных, использующие указатели. В предлагаемом решении используется ключевое слово _based, предоставляемое Microsoft C. Альтернативным вариантом было бы отображение файла в массив и обеспечение доступа к записям в представлении объекта отображения файла с помощью индекса.

Программа написана в виде еще одной версии команды sort, которой в данном случае присвоено имя sortMM. Данная версия программы отличается следующими особенностями, заслуживающими внимания:

• Записи могут иметь переменную длину.

• Программа использует первое поле в качестве ключа, но определяет его длину.

• Строятся два представления файла. Одно из них представляет исходный файл, а второе — файл, содержащий отсортированные ключи. Второй файл является *индексным файлом* (index file), каждая из записей которого содержит ключ и указатель (базовый адрес), относящийся к исходному файлу. Для сортировки индексного файла, во многом по аналогии с программой 5.4, применяется функция qsort.

• Индексный файл сохраняется и впоследствии может быть использован, причем предусмотрена возможность (параметр командной строки –I) отказаться от сортировки и использовать существующий индексный файл. Кроме того, индексный файл может быть использован для быстрого поиска ключей путем проведения бинарного поиска (возможно, с использованием входящей в библиотеку С функции bsearch) в индексном файле.

Взаимосвязь между индексным файлом и сортируемым файлом иллюстрирует рис. 5.5. Главной программой является программа 5.5, которая обеспечивает создание представлений файлов в памяти компьютера, осуществляет сортировку индексного файла и отображает результаты. Эта программа вызывает функцию CreateIndexFile, представленную программой 5.6.

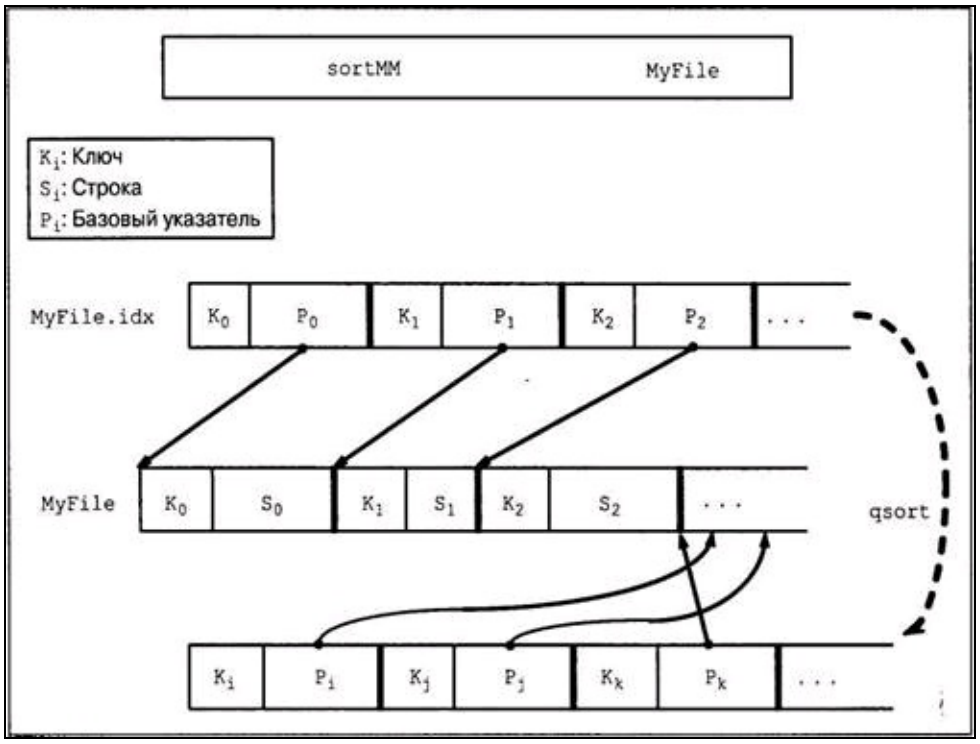


Рис. 5.5. Сортировка с использованием отображения индексного файла

Программа 5.5. sortMM: использование базовых указателей в индексном файле

```
/* Глава 5. Команда sortMM.
 Сортировка отображенного в памяти файла - только один файл. Опции:
 -r Сортировать в обратном порядке.
 -I Использовать индексный файл для получения отсортированного файла. */
#include "EvryThng.h"

int KeyCompare(LPCTSTR , LPCTSTR);
DWORD CreateIndexFile (DWORD, LPCTSTR, LPTSTR);
DWORD KStart, KSize; /* Начальная позиция и размер ключа (TCHAR) . */
BOOL Revrs;

int _tmain(int argc, LPTSTR argv []) {
 HANDLE hInFile, hInMap; /* Дескрипторы входного файла. */
 HANDLE hXFile, hXMap; /* Дескрипторы индексного файла. */
 HANDLE hStdOut = GetStdHandle(STD_OUTPUT_HANDLE);
 BOOL IdxExists;
 DWORD FsIn, FsX, RSize, iKey, nWrite, *pSizes;
 LPTSTR pInFile = NULL;
 LPBYTE pXFile = NULL, pX;
 TCHAR _based(pInFile) *pIn;
 TCHAR IdxFlNam [MAX_PATH], ChNewLine = TNEWLINE;
 int FlIdx = Options(argc, argv, _T("rI"), &Revrs, &IdxExists, NULL);
 /* Шаг 1: открыть и отобразить входной файл. */
  hInFile = CreateFile(argv [FlIdx], GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, 0, NULL);
 hInMap = CreateFileMapping(hInFile, NULL, PAGE_READWRITE, 0, 0, NULL);
 pInFile = MapViewOfFile(hInMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);
 FsIn = GetFileSize(hInFile, NULL);
 /* Шаги 2 и 3: создать имя индексного файла. */
 _stprintf(IdxFlNam, _T("%s%s"), argv[FlIdx], _T(".idx"));
```

```
    if (!IdxExists) RSize = CreateIndexFile(FsIn, IdxFlNam, pInFile);
    /* Шаг 4: отобразить индексный файл. */
     hXFile  =  CreateFile(IdxFlNam,  GENERIC_READ  |  GENERIC_WRITE,  0,  NULL,
OPEN_EXISTING, 0, NULL);
    hXMap = CreateFileMapping(hXFile, NULL, PAGE_READWRITE, 0, 0, NULL);
    pXFile = MapViewOfFile(hXMap, FILE_MAP_ALL_ACCESS, 0, 0, 0);
    FsX = GetFileSize(hXFile, NULL);
    pSizes = (LPDWORD)pXFile; /* Поля размера в .idx-файле. */
    KSize = *pSizes; /* Размер ключа */
    KStart = *(pSizes + 1); /* Начальная позиция ключа в записи. */
    FsX -= 2 * sizeof(DWORD);
    /* Шаг 5: сортировать индексный файл при помощи qsort. */
     if  (!IdxExists)  qsort(pXFile  +  2  *  sizeof(DWORD),  FsX  /  RSize,  RSize,
KeyCompare);
    /* Шаг 6: отобразить входной файл в отсортированном виде. */
    pX = pXFile + 2 * sizeof(DWORD) + RSize - sizeof(LPTSTR);
    for (iKey = 0; iKey < FsX / RSize; iKey++) {
     WriteFile(hStdOut, &ChNewLine, TSIZE, &nWrite, NULL);
     /* Приведение типа pX, если это необходимо! */
     pIn = (TCHAR _based (pInFile)*) *(LPDWORD)pX;
     while ((*pIn != CR || * (pIn + 1) != LF) && (DWORD) pIn < FsIn) {
      WriteFile(hStdOut, pIn, TSIZE, &nWrite, NULL); pIn++;
     }
     pX += RSize;
    }
    UnmapViewOfFile(pInFile);
    CloseHandle(hInMap);
    CloseHandle(hInFile);
    UnmapViewOfFile(pXFile);
    CloseHandle(hXMap);
    CloseHandle(hXFile);
    return 0;
   }
```

Программа 5.6 представляет собой функцию CreateIndexFile, с помощью которой создается индексный файл. Сначала она просматривает входной файл для определения размера ключа по первой записи. После этого она должна просматривать входной файл для нахождения границ каждой из записей переменной длины для организации структуры, представленной на рис. 5.5.

Программа 5.6. sortMM: создание индексного файла

```
   DWORD CreateIndexFile(DWORD FsIn, LPCTSTR IdxFlNam, LPTSTR pInFile) {
    HANDLE hXFile;
    TCHAR _based (pInFile) *pInScan = 0;
    DWORD nWrite;
    /* Шаг 2a: создать индексный файл. Не отображать его на данной стадии. */
     hXFile = CreateFile(IdxFlNam, GENERIC_READ | GENERIC_WRITE, FILE_SHARE_READ,
NULL, CREATE_ALWAYS, 0, NULL);
    /* Шаг 2b: получить первый ключ и определить его размер и начальную позицию.
Пропустить пробел и получить длину ключа. */
    KStart = (DWORD) pInScan;
    while (*pInScan != TSPACE && *pInScan != TAB) pInScan++; /* Найти поле первого
ключа. */
    KSize = ((DWORD)pInScan - KStart) / TSIZE;
     /* Шаг 3: просмотреть весь файл, записывая ключи и указатели записей в
индексный файл. */
    WriteFile(hXFile, &KSize, sizeof(DWORD) , &nWrite, NULL);
    WriteFile(hXFile, &KStart, sizeof(DWORD), &nWrite, NULL);
```

```
     pInScan = 0;
     while ((DWORD)pInScan < FsIn) {
      WriteFile(hXFile, pInScan + KStart, KSize * TSIZE, &nWrite, NULL);
      WriteFile(hXFile, &pInScan, sizeof(LPTSTR), &nWrite, NULL);
      while ((DWORD)pInScan < FsIn && ((*pInScan != CR) || (*(pInScan + 1) != LF)))
{
       pInScan++; /* Пропустить до конца строки. */
      }
      pInScan += 2; /* Пропустить CR, LF. */
     }
     CloseHandle(hXFile);
     /* Размер отдельной записи. */
     return KSize * TSIZE + sizeof(LPTSTR);
    }
```

Динамически компонуемые библиотеки

Как вы имели возможность убедиться, средства управления памятью и отображения файлов оказываются важными и полезными для широкого класса программ. Системы управления памятью используются также самими ОС, и наиболее важной и заслуживающей внимания сферой применения отображения файлов являются библиотеки DLL. DLL широко используются приложениями Windows, являясь существенным элементом таких высокоуровневых технологий, как COM, а многие компоненты программного обеспечения поставляются в виде DLL.

Нашим первым шагом будет рассмотрение различных методов построения библиотек наиболее часто используемых функций.

Статические и динамические библиотеки

Самый непосредственный способ построения любой программы — это объединение исходных кодов всех функций, их компиляция и компоновка всех необходимых элементов в один исполняемый модуль. Чтобы упростить процесс сборки, такие функции общего назначения, как ReportError, можно поместить в библиотеку. Этот подход применялся во всех представленных до сих пор примерах программ, хотя и касался всего лишь нескольких функций, большинство из которых предназначались для вывода сообщений об ошибках.

Эта монолитная, одномодульная модель отличается простотой, однако обладает и рядом недостатков.

• Исполняемый модуль может разрастаться до больших размеров, занимая большие объемы дискового пространства и физической памяти во время выполнения и требуя дополнительных усилий для организации управления модулем и передачи его пользователям.

• При каждом обновлении потребуется повторная сборка всей программы, даже если необходимые изменения незначительны или носят локальный характер.

• Каждый исполняемый модуль тех программ в системе, которые используют эти функции, будет иметь свои экземпляры функций, версии которых могут различаться. Подобная схема компоновки приводит к тому, что при одновременном выполнении нескольких таких программ будет напрасно расходоваться дисковое пространство и, что намного существеннее, физическая память.

• Для достижения наилучшей производительности в различных средах может потребоваться использование различных версий программы, в которых применяются различные методики. Так, функция Asc2Un в программе 2.4 (atou) и программе 5.3 (Asc2UnMM) реализована по-разному. Единственный способ выполнения программ, имеющих несколько различных реализаций, — это заранее принять решение относительно того, какую из двух версий запускать, исходя из свойств окружения.

Библиотеки DLL обеспечивают возможность элегантного решения этих и других проблем.

• Библиотечные функции не связываются во время компоновки. Вместо этого их связывание осуществляется во время загрузки программы (*неявное связывание*) или во время ее выполнения *(явное связывание)*. Это позволяет существенно уменьшить размер модуля программы, поскольку библиотечные функции в него не включаются.

• DLL могут использоваться для создания *общих библиотек* (shared libraries). Одну и ту же библиотеку DLL могут совместно использовать несколько одновременно выполняющихся программ, но в память будет загружена только одна ее копия. Все программы отображают код DLL на адресные пространства своих процессов, хотя каждый поток будет иметь собственный

экземпляр неразделяемого хранилища в стеке. Например, функция ReportError использовалась почти в каждом из приведенных ранее примеров программ, тогда как для всех программ было бы вполне достаточно ее единственной DLL-реализации.

• Новые версии программ или другие возможные варианты их реализации могут поддерживаться путем простого предоставления новой версии DLL, а все программы, использующие эту библиотеку, могут выполняться как новая версия без внесения каких бы то ни было дополнительных изменений.

• В случае явного связывания решение о том, какую версию библиотеки использовать, программа может принимать во время выполнения. Разные библиотеки могут быть альтернативными вариантами реализации одной и той же функции или решать совершенно иные задачи, как если бы они были независимыми программами. Библиотека выполняется в том же процессе и том же потоке, что и вызывающая программа.

Библиотеки DLL, иногда в ограниченном виде, используются практически в любой ОС. Так, в UNIX аналогичные библиотеки фигурируют под названием "разделяемых библиотек" (shared libraries). В Windows библиотеки DLL используются, помимо прочего, для создания интерфейсов ОС. Весь Windows API поддерживается одной DLL, которая для предоставления дополнительных услуг вызывает ядро Windows.

Один код DLL может совместно использоваться несколькими процессами Windows, но после его вызова он выполняется как часть вызывающего процесса или потока, Поэтому библиотека может использовать ресурсы вызывающего процесса, например дескрипторы файлов, и стек потока. Следовательно, DLL должны создаваться таким образом, чтобы обеспечивалась безопасная многопоточная поддержка (thread safety). (Более подробная информация относительно DLL и безопасной многопоточной поддержки содержится в главах 8, 9 и 10. Методы создания DLL, предоставляющих многопоточную поддержку, иллюстрируются программами 12.4 и 12.5.) Кроме того, DLL могут экспортировать переменные, а также точки входа функций.

Неявное связывание

Неявное связывание, или *связывание во время загрузки* (load-time linking) является простейшей из двух методик связывания. Порядок действий в случае использования Microsoft C++ следующий:

1. После того как собраны все необходимые для новой DLL функции, осуществляется сборка DLL, а не, например, консольного приложения.

2. В процессе сборки создается библиотечный .LIB-файл, играющий роль *заглушки* (stub) для фактического кода. Этот файл должен помещаться в каталог библиотек общего пользования, указанный в проекте.

3. В процессе сборки создается также .DLL-файл, содержащий исполняемый модуль. В типичных случаях этот файл размещается в том же каталоге, что и приложение, которое будет его использовать, и приложение загружает DLL в процессе своей инициализации. Вторым возможным местом расположения DLL является рабочий каталог, а далее ОС будет осуществлять поиск .DLL-файла в системном каталоге, каталоге Windows, а также в путях доступа, указанных в переменной окружения PATH.

4. В исходном коде DLL следует предусмотреть экспортирование интерфейсов функций, о чем рассказано ниже.

Самое значительное изменение, которое требуется внести в функцию, прежде чем ее можно будет поместить в DLL, — это объявить ее экспортируемой (UNIX и некоторые другие системы не требуют явного выполнения этого шага). Это достигается либо за счет использования .DEF-файла, либо, что проще и возможно в Microsoft C, за счет использования в объявлениях модификатора _declspec (dllexport) следующим образом:

```
_declspec(dllexport) DWORD MyFunction (…);
```

Далее в процессе сборки создаются .DLL-файл и .LIB-файл. .LIB-файл — это библиотека-заглушка, которая должна быть скомпонована с вызывающей программой для разрешения внешних ссылок и создания актуальных связей с . DLL-файлом во время загрузки.

Вызывающая, или *клиентская,* программа должна объявить о том, что функцию следует импортировать, используя для этого модификатор _declspec (dllexport). Стандартный метод заключается в создании включаемого файла, использующего переменную препроцессора, имя которой формируется на основе имени проекта Microsoft Visual C++, набранного в верхнем регистре и дополненного суффиксом _EXPORTS.

Вам также может потребоваться еще одно объявление. Если вызывающая (клиентская) программа написана на C++, то для нее будет определена переменная препроцессора __cplusplus, и вы должны будете указать на необходимость использования системы соглашений о вызовах, принятой в C, с помощью следующего выражения:

```
extern "C"
```

Если, например, в качестве части сборки DLL в проекте MyLibrary определена функция MyLibrary, то содержимое заголовочного файла должно быть таким:

```
#if defined(MYLIBRARY_EXPORTS)
#define LIBSPEC _declspec(dllexport)
#elif defined(__cplusplus)
#define LIBSPEC extern "C" _declspec(dllimport)
#else
#define LIBSPEC _declspec(dllimport)
#endif
LIBSPEC DWORD MyFunction (…);
```

Visual C++ автоматически определяет MYLIBRARY_EXPORTS при вызове компилятора, если проект предназначен для создания DLL MyLibrary. Клиентский проект, который использует DLL, переменную MYLIBRARYEXPORTS не определяет, и поэтому имя функции импортируется из библиотеки.

При построении вызывающей программы укажите соответствующий .DLL-файл. Когда будете запускать вызывающую программу на выполнение, убедитесь в наличии доступа к этому файлу; часто это обеспечивается размещением .DLL-файла в одном каталоге с исполняемым файлом. Как ранее уже отмечалось, существует ряд правил поиска DLL, определяющих последовательность просмотра каталогов, в которых Windows будет осуществлять поиск указанного .DLL-файла, *а также* других DLL и исполняемых файлов, необходимых указанному файлу, прекращая этот поиск, как только будет найден первый подходящий экземпляр. Ниже приведена *стандартная последовательность просмотра каталогов при поиске,* используемая как в случае явного, так и в случае неявного связывания.

• Каталог, в котором находится загружаемое приложение.

• Текущий каталог, если он отличен от каталога, содержащего исполняемый модуль.

• Системный каталог Windows. Вы можете определить этот путь, вызвав функцию GetSystemDirectory; таковым обычно является каталог c:\WINDOWS\SYSTEM32.

• Системный каталог 16-разрядной Windows, который отсутствует в системах Windows 9x. Функция, позволяющая получить путь доступа к этому каталогу, отсутствует, и для наших целей он оказывается ненужным.

• Каталог Windows (используйте функцию GetWindowsDirectory).

• Каталоги, перечисленные в переменной окружения PATH, которые будут просматриваться в той последовательности, в какой они указаны.

Заметьте, что этот стандартный порядок просмотра каталогов при поиске можно изменить, о чем говорится в разделе "Явное связывание". Для получения более подробной информации относительно стратегии поиска посетите Web-сайт по адресу http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dllproc/base/loadlibrary.asp, а также ознакомьтесь с описанием функции SetDllDirectory, введенной в Windows NT 5.1 (то есть Windows XP). Изменить стратегию поиска позволяет также функция LoadLibraryEx, описанная в следующем разделе.

Применение стандартной стратегии поиска иллюстрируется в проекте Utilities, доступном на Web-сайте книги, а такие вспомогательные функции, как ReportError, используются почти в каждом примере проектов.

Возможно также экспортирование и импортирование переменных, а также точек входа функций, хотя эти возможности в примерах не иллюстрируются.

Явное связывание

Явное связывание, или *связывание во время выполнения* (run-time linking), требует, чтобы в программе содержались конкретные указания относительно того, когда именно необходимо загрузить или освободить библиотеку DLL. Далее программа получает адрес запрошенной точки входа и использует этот адрес в качестве указателя при вызове функции. В вызывающей программе функция не объявляется; вместо этого в качестве указателя на функцию объявляется переменная. Поэтому во время компоновки программы присутствие библиотеки не является обязательным. Для выполнения необходимых операций требуются три функции: LoadLibrary (или LoadLibraryEx), GetProcAddress и FreeLibrary. На 16-битовое происхождение определений функций указывает присутствие в них дальних (far) указателей и дескрипторов различных типов.

Для загрузки библиотеки служат две функции: LoadLibrary и LoadLibraryEx.

```
HINSTANCE LoadLibrary(LPCTSTR lpLibFileName)

HINSTANCE LoadLibraryEx(LPCTSTR lpLibFileName, HANDLE hFile, DWORD
dwFlags)
```

В обоих случаях значением возвращаемого дескриптора (типа HINSTANCE, а не HANDLE) в случае ошибки будет NULL. Суффикс .DLL в имени файла указывать не обязательно. С помощью функций LoadLibrary можно загружать также .EXE-файлы. При указании путей доступа должны использоваться символы обратной косой черты (\); символы прямой косой черты (/) в данном случае работать не будут.

Поскольку библиотеки DLL являются совместно используемым ресурсом, системой поддерживается счетчик ссылок на каждую DLL (который увеличивается на единицу при каждом вызове любой из указанных выше функций загрузки), так что повторное отображение фактического файла библиотеки не требуется. Функция LoadLibrary завершится с ошибкой даже в случае успешного нахождения .DLL-файла, если данная библиотека DLL неявно связана с

другой DLL, найти которую программе не удалось.

Функция LoadLibraryEx аналогична функции LoadLibrary, однако имеет несколько флагов, которые оказываются полезными для указания альтернативных путей поиска и загрузки библиотек в виде файла данных. Параметр hFile зарезервирован для использования в будущем. Параметр dwFlags позволяет определять различные варианты поведения системы путем указания одного из трех значений:

1. LOAD_WITH_ALTERED_SEARCH_PATH: отменяет ранее описанный стандартный порядок просмотра каталогов при поиске, изменяя лишь первый из шагов стратегии поиска. Вместо каталога, из которого загружалось приложение, используется путь поиска, указанный в имени lpLibFileName.

2. LOAD_LIBRARY_AS_DATAFILE: файл воспринимается как файл данных и не требует выполнения каких-либо действий по его подготовке к запуску, на пример вызова функции DllMain (см. раздел "Точки входа библиотеки DLL" далее в этой главе).

3. DONT_RESOLVE_DLL_REFERENCE: функция DllMain для инициализаций процессов и потоков не вызывается; загрузка дополнительных модулей, на которые имеются ссылки в указанной DLL, также не производится.

Закончив работать с экземпляром DLL — возможно, с намерением загрузить другую ее версию — вы должны освободить дескриптор библиотеки, тем самым освобождая ресурсы, в том числе распределенное для библиотеки виртуальное адресное пространство. Однако DLL продолжает оставаться загруженной, если счетчик ссылок указывает на то, что она все еще используется другими процессами.

```
BOOL FreeLibrary(HINSTANCE hLibModule)
```

После загрузки библиотеки, но до ее освобождения, вы можете получить адрес любой точки входа, используя функцию GetProcAddress.

```
FARPROC GetProcAddress(HMODULE hModule, LPCSTR lpProcName)
```

Параметр hModule, несмотря на другой тип имени (HINSTANCE определен как HMODULE), является экземпляром (instance) библиотеки, получаемым посредством вызова функции LoadLibrary или GetModuleHandle (см. следующий абзац). lpProcName — указатель на строку, содержащую имя точки входа; это имя не может задаваться в кодировке Unicode. В случае неуспешного выполнения функция возвращает значение NULL. Слово FARPROC, означающее "длинный указатель на процедуру", является анахронизмом.

Имя файла, связанного с дескриптором hHandle, можно получить с помощью функции GetModuleFileName. Возможно и обратное: для заданного имени файла (.DLL или .EXE) функция GetModuleHandle в случае успешного выполнения возвратит дескриптор, связанный с этим файлом, если текущий процесс загрузил его.

В следующем примере показано, как использовать адрес точки входа для вызова функции.

Пример: явное связывание функци и преобразования файлов

Программа 2.4, предназначенная для преобразования кодировки текстовых файлов из ASCII в Unicode, вызывает функцию Asc2Un (программа 2.5), выполняющую обработку файла с использованием операций файлового ввода/вывода. Программа 5.3 (Asc2UnMM) представляет альтернативную функцию, которая для выполнения той же операции использует отображение файлов. Обстоятельства, при которых функция Asc2UnMM обеспечивает выигрыш в скорости выполнения преобразования, ранее уже обсуждались; в основном они сводятся к тому, что файловой системой должна быть NTFS, а размер файла не должен быть слишком большим.

Программа 5.7 является модифицированным вариантом вызывающей программы, обеспечивающим возможность принятия решения относительно того, какой вариант реализации функции преобразования должен быть загружен, во время выполнения. Программа загружает DLL, получает адрес точки входа Asc2Un и вызывает функцию. В данном случае существует только одна точка входа, но реализовать вариант с несколькими точками входа не составляет особого труда. Основная программа является, по существу, той же, что и прежде, за исключением того, что библиотека DLL, которую необходимо использовать, указывается в виде параметра командной строки. В упражнении 5.9 вам предлагается написать вариант программы, в котором нужная DLL определяется на основе свойств системы и файла. Обратите внимание на то, каким образом осуществляется приведение типа адреса FARPROC к типу соответствующей функции с использованием необходимого в этом случае, но довольно сложного, синтаксиса C.

Программа 5.7. atouEL: преобразование файлов с использованием явного связывания

```
/* Глава 5. Версия atou, использующая явное связывание. */
#include "EvryThng.h"

int _tmain(int argc, LPTSTR argv[]) {
 /* Объявить переменную Asc2Un как функцию. */
 BOOL (*Asc2Un)(LPCTSTR, LPCTSTR, BOOL);
 DWORD LocFileIn, LocFileOut, LocDLL, DashI;
 HINSTANCE hDLL;
 FARPROC pA2U;
 LocFileIn = Options(argc, argv, _T("i"), &DashI, NULL);
 LocFileOut = LocFileIn + 1;
 LocDLL = LocFileOut + 1;
 /* Проверить существование файла, а также опущен ли параметр DashI. */
 /* Загрузить функцию преобразования из ASCII в Unicode. */
 hDLL = LoadLibrary(argv[LocDLL]);
 if (hDLL == NULL) ReportError(_T("Не удается загрузить DLL."), 1, TRUE);
 /* Получить адрес точки входа. */
 pA2U = GetProcAddress(hDLL, "Asc2Un");
 if (pA2U == NULL) ReportError(_T("Не найдена точка входа."), 2, TRUE);
 /* Привести тип указателя. Здесь можно использовать typedef. */
 Asc2Un = (BOOL(*)(LPCTSTR, LPCTSTR, BOOL))pA2U;
 /* Вызвать функцию. */
 Asc2Un(argv[LocFileIn], argv[LocFileOut], FALSE);
 FreeLibrary(hDLL);
 return 0;
}
```

Создание библиотек DLL на основе функции Asc2Un

Программа тестировалась с двумя функциями преобразования файлов, которые должны были создаваться в виде библиотек DLL, имеющих различные имена, но идентичные точки входа. В данном случае существует только одна точка входа. Единственным существенным изменением в исходном коде является добавление модификатора _declspec(dllexport) для экспортирования функции.

Точки входа библиотеки DLL

Для каждой создаваемой DLL вы можете указать точку входа запуска библиотеки, которая обычно автоматически вызывается при каждом подключении или отключении процесса. В то же время, в функции LoadLibraryEx предусмотрена опция, позволяющая подавить вызов точки входа. В случае неявно связываемых (связываемых во время выполнения) библиотек DLL подключение и отключение процесса происходит, соответственно, при его запуске и завершении. В случае же явно связываемых DLL это осуществляется при вызове функций LoadLibrary, LoadLibraryEx и FreeLibrary.

Кроме того, точка входа вызывается всякий раз, когда процесс создает новый поток (глава 7) или прекращает его выполнение.

Точкой входа с именем DllMain, прототип которой приводится ниже, мы воспользуемся в полной мере только в главе 12 (программа 12.4), где она предоставит потокам удобный способ управления ресурсами и так называемыми локальными областями хранения потоков (Thread Local Storage, SLT) в DLL с многопоточной поддержкой.

```
BOOL DllMain(HINSTANCE hDll, DWORD Reason, LPVOID Reserved)
```

Параметр hDll является дескриптором экземпляра DLL, возвращенным функцией LoadLibrary. Значение NULL параметра Reserved указывает на то, что подключение процесса к библиотеке произошло в результате вызова функции Load-Library; иные значения этого параметра свидетельствуют о подключении к библиотеке в результате неявного связывания во время загрузки. Подобным образом, к значению NULL параметра Reserved приводит и отключение процесса от библиотеки в результате вызова функции FreeLibrary.

Параметр Reason может иметь одно из четырех значений: DLL_PROCESS_ATTACH, DLL_THREAD_ATTACH, DLL_THREAD_DETACH и DLL_PROCESS_DETACH. Функции точки входа DLL обычно используют операторы switch и в качестве индикатора успешного выполнения возвращают значение TRUE.

Система сериализует вызовы DllMain таким образом, что в каждый момент времени выполнять ее может только один поток (к подробному обсуждению потоков мы приступим в главе 7). Эта сериализация весьма существенна, поскольку операции инициализации, которые должна выполнять DllMain, не должны прерываться до их завершения. По этой же причине внутри точки входа не рекомендуется использовать блокирующие вызовы функций, например, функций ввода/вывода или функций ожидания (см. главу 8), поскольку они будут препятствовать запуску точки входа другими потоками. В частности, не следует вызывать внутри точки входа DLL функции LoadLibrary и LoadLibraryEx, поскольку это будет порождать дополнительные вызовы точек входа DLL.

Функция DisableThreadLibraryCalls отменяет отправку указанному экземпляру DLL уведомлений о подключении и отключении потоков. Запрет отправки уведомлений может пригодиться в тех случаях, когда потоки не нуждаются в каких-либо уникальных ресурсах во время инициализации.

Управление версиями DLL

При использовании DLL обычно проявляются трудности, обусловленные обновлением библиотек за счет введения новых символов и добавления новых средств. Основное преимущество DLL заключается в том, что несколько приложений могут совместно использовать одну и ту же библиотеку, находящуюся в памяти. Вместе с тем, это порождает целый ряд осложнений, связанных с совместимостью версий, что иллюстрируется приведенными ниже примерами.

• В результате добавления новых функций в случае неявного связывания могут стать недействительными смещения, определенные для приложений во время компоновки с .lib-файлами. От этой проблемы можно избавиться, применив явное связывание.

• Поведение новых версий функций может быть иным, в результате чего существующие приложения могут испытывать проблемы, если не будут своевременно обновлены.

• Для приложений, использующих обновленную функциональность DLL, возможны случая связывания с прежними версиями DLL.

Проблемы совместимости различных версий DLL, носящие жаргонное название "кошмара DLL", не являются столь острыми, если в одном каталоге поддерживать только одну версию DLL. Однако предоставить отдельный каталог для каждой из различных версий вовсе не так просто, как может показаться. Существует несколько других вариантов решения этой проблемы.

• Можно использовать номер версии DLL в именах .DLL– и .LIB-файлов, обычно в виде суффикса. Так, чтобы соответствовать номеру версии, используемой в данной книге, в примерах, приведенных на Web-сайте книги, и во всех проектах используются файлы Utility_3_0.LIB и Utility_3_0.DLL. Применяя явное или неявное связывание, приложения могут формулировать свои требования к версиям и получать доступ к файлам с различными именами. Такое решение характерно для UNIX-приложений.

• Компания Microsoft ввела понятие параллельных DLL (side-by-side DLL), или сборок (assemblies) и компонентов (components). При таком подходе в приложение необходимо включать объявление на языке XML, в котором определяются требования к DLL. Рассмотрение этой темы выходит за рамки данной книги, однако дополнительную информацию вы можете получить на Web-сайте компании Microsoft, в разделе, посвященном вопросам разработки приложений.

• Платформа .NET Framework предоставляет дополнительные средства поддержки выполнения приложений в условиях сосуществования различных версий DLL.

В примерах проектов, используемых в данной книге, используется первый из отмеченных подходов, предусматривающий включение номеров версий в имена файлов. С целью предоставления дополнительной поддержки, обеспечивающей возможность получения приложениями информации о DLL, во всех DLL реализована функция DllGetVersion. Кроме того, Microsoft предоставляет эту косвенно вызываемую функцию в качестве стандартного средства получения информации о версии в динамическом режиме. Указанная функция имеет следующий прототип:

```
HRESULT CALLBACK DllGetVersion(DLLVERSIONINFO *pdvi )
```

Информация о DLL возвращается в структуре DLLVERSIONINFO, в которой имеются поля типа DWORD для параметров cbSize (размер структуры), dwMajorVersion, dwMinorVersion, dwBuildNumber и dwPlatformID. В последнем поле, dwPlatformID, может быть установлено значение DLLVER_PLATFORM_NT, если библиотека не выполняется под управлением Windows

9x, или DLLVER_PLATFORM_WINDOWS, если это ограничение отсутствует. В поле cbSize должно находиться значение sizeof (DLLVERSIONINFO). В случае успешного выполнения функция возвращает значение NOERROR. Функция DllGetVersion реализована в проекте Utility_3_0.

Резюме

Система управления памятью Windows предоставляет следующие возможности:

• Использование средств Windows, осуществляющих управление кучей, а также обработчиков исключений для обнаружения и обработки ошибок, возникающих при распределении памяти, значительно упрощает логическую организацию.

• Использование нескольких независимых куч обладает рядом преимуществ по сравнению с распределением памяти из одной кучи.

• Методы отображения файлов, доступные в UNIX, но не предоставляемые библиотекой C, обеспечивают обработку файлов в памяти, что было проиллюстрировано несколькими примерами. Отображение файлов в памяти осуществляется независимо от управления кучей и упрощает решение многих задач программирования. Преимущества использования отображения файлов подтверждаются данными о достигаемом за счет этого повышении производительности, приведенными в приложении B.

• DLL являются важным специальным случаем отображения файлов и могут загружаться либо явным, либо неявным образом. DLL, предназначенные для использования многими приложениями, должны предоставлять информацию о версии библиотеки.

В следующих главах

Мы завершили обзор задач, решаемых в рамках одного процесса. Далее мы переходим к изучению методов параллельной обработки, сначала на уровне процессов (глава 6), а затем — потоков (глава 7). В последующих главах показано, как организовать синхронизацию и взаимодействие параллельно выполняющихся операций по обработке данных.

Дополнительная литература

Отображение файлов, виртуальная память и ошибки страниц

Описание этих важных понятий содержится в книге [38], а их углубленное обсуждение вы можете найти в документации, поставляемой вместе с большинством ОС.

Структуры данных и алгоритмы

Деревьям поиска и алгоритмам сортировки посвящено множество работ, включая [39] и [34].

Использование явного связывания

DLL и явное связывание имеют фундаментальное значение для использования модели COM, которая широко применяется при разработке программного обеспечения Windows. Важность функций LoadLibrary и GetProcAddress продемонстрирована в главе 1 книги [3].

Упражнения

5.1. Спроектируйте и проведите эксперименты для оценки выигрыша в производительности, достигаемого за счет использования флага HEAP_NO_SERIALIZE при вызове функций HeapCreate и HeapAlloc. Как зависит этот показатель от размера кучи и размера блока? Заметна ли разница в результатах для различных версий Windows? На Web-сайте книги находится программа HeapNoSr.c, которая поможет вам приступить к выполнению этого и следующего упражнений.

5.2. Измените тестовую программу из предыдущего упражнения таким образом, чтобы она позволяла определить, генерирует ли функция malloc исключения или возвращает нулевой указатель в случае нехватки памяти. Является ли обнаруженное поведение функции корректным? Сравните также производительность, обеспечиваемую функцией malloc, с результатами предыдущего упражнения.

5.3. Доля накладных издержек при распределении памяти из кучи колеблется в зависимости от используемой версии Windows, что особенно заметно в случае выходящих из употребления версий Windows 9x. Спроектируйте и проведите эксперимент для определения количества блоков памяти фиксированного размера, которые каждая из систем предоставляет в одной куче. Используя SEH для определения того момента, когда распределенными оказываются все блоки, вы значительно упростите программу. Подобным образом ведет себя программа clear.c, находящаяся на Web-сайте книги, если игнорировать часть ее кода, ответственную за явное тестирование ОС. Между прочим, эта программа используется в некоторых тестах по измерению временных характеристик для гарантии того, что данные, полученные в процессе выполнении предыдущего теста, не остались в памяти.

5.4. Путем изменения программы sortFL (программа 5.4) создайте программу sortHP, распределяющую память для буфера, размер которого достаточно велик, чтобы в нем уместился весь файл, и выполните считывание файла в этот буфер. Отображение файла использовать не следует. Сравните производительность обеих программ.

5.5. В программе 5.5 применены указатели типа _base, специфические для Microsoft C. Если ваш компилятор не поддерживает это средство (но в любом случае — просто в качестве упражнения) переделайте программу 5.5, используя для генерации значений базового указателя макрос, массив или иной механизм.

5.6. Напишите программу поиска записей по указанному ключу в файле, проиндексированном с применением программы 5.5. Для этой цели удобно воспользоваться функцией bsearch, входящей в состав библиотеки C.

5.7. Реализуйте программу tail из главы 3, используя отображение файлов.

5.8. Поместите вспомогательные функции ReportError, PrintStrings, PrintMsg и ConsolePrompt в DLL и перекомпонуйте некоторые из программ, с которыми мы работали раньше. Проделайте то же самое с функциями Options и GetArgs, предназначенными, соответственно, для обработки параметров командной строки и аргументов. Важно, чтобы как вспомогательная DLL, так и вызывающая программа использовали также и библиотеку C в виде DLL. Например, в Visual C++ и Visual Studio 6.0 выберите, начав со строки главного меню, следующие команды: Project (Проект), Settings (Параметры), вкладку C/C++, Category (Code Generation) (Категория (Генерация кода)), Use Run-Time Library (Multithreaded DLL) (Использовать библиотеку времени выполнения (многопоточная DLL)). Заметьте, что библиотеки DLL, вообще говоря, должны обеспечивать многопоточную поддержку, поскольку они будут использоваться потоками нескольких процессов. Пример возможного решения содержится в проекте Utilities_3_0, доступном на Web-

сайте книги.

5.9. Измените программу 5.7 таким образом, чтобы решение относительно того, какую DLL следует использовать, базировалось на размере файла и конфигурации системы. .LIB-файл здесь не требуется, поэтому продумайте, как отменить его генерацию. Для определения типа файловой системы используйте функцию GetVolumeInformation.

5.10. Создайте дополнительные DLL для функции преобразования из предыдущего упражнения, каждая версия которых использует иную методику обработки файлов, и расширьте вызывающую программу таким образом, чтобы она сама решала, когда и какую версию использовать.

ГЛАВА 6
Управление процессами

Процесс (process) представляет собой объект, обладающий собственным независимым виртуальным адресным пространством, в котором могут размещаться код и данные, защищенные от других процессов. В свою очередь, внутри каждого процесса могут независимо выполняться одна или несколько *потоков* (threads). Поток, выполняющийся внутри процесса, может сама создавать новые потоки и новые независимые процессы, а также управлять взаимодействием объектов между собой и их синхронизацией.

Создавая процессы и управляя ими, приложения могут организовывать параллельное выполнение нескольких задач, обеспечивающих обработку файлов, проведение вычислений или связь с другими системами в сети. Допускается даже использование нескольких процессоров с целью ускорения обработки данных.

В этой главе объясняются основы управления процессами и вводятся; простейшие операции синхронизации, которые будут использоваться на протяжении оставшейся части книги.

Процессы и потоки Windows

Внутри каждого процесса могут выполняться одна или несколько потоков, и именно поток является базовой единицей выполнения в Windows. Выполнение потоков планируется системой на основе обычных факторов: наличие таких ресурсов, как CPU и физическая память, приоритеты, равнодоступность ресурсов и так далее. Начиная с версии NT4, в Windows поддерживается симметричная многопроцессорная обработка (Symmetric Multiprocessing, SMP), позволяющая распределять выполнение потоков между отдельными процессорами, установленными в системе.

С точки зрения программиста каждому процессу принадлежат ресурсы, представленные следующими компонентами:

• Одна или несколько потоков.

• Виртуальное адресное пространство, отличное от адресных пространств других процессов, если не считать областей памяти, распределенных явным образом для совместного использования (разделения) несколькими процессами. Заметьте, что разделяемые отображенные файлы совместно используют физическую память, тогда как разделяющие их процессы используют различные виртуальные адресные пространства.

• Один или несколько сегментов кода, включая код DLL.

• Один или несколько сегментов данных, содержащих глобальные переменные.

• Строки, содержащие информацию об окружении, например, информацию о текущем пути доступа к файлам.

• Куча процесса.

• Различного рода ресурсы, например, дескрипторы открытых файлов и другие кучи.

Поток разделяет вместе с процессом код, глобальные переменные, строки окружения и другие ресурсы. Каждый поток планируется независимо от других и располагает следующими элементами:

• Стек, используемый для вызова процедур, прерываний и обработчиков исключений, а также хранения автоматических переменных.

• Локальные области хранения потока (Thread Local Storage, SLT) — массивы указателей, используя которые каждый поток может создавать собственную уникальную информационную среду.

• Аргумент в стеке, получаемый от создающего потока, который обычно является уникальным для каждого потока.

• Структура контекста, поддерживаемая ядром системы и содержащая значения машинных регистров.

На рис. 6.1 показан процесс с несколькими потоками. Рисунок является схематическим, поэтому на нем не указаны фактические адреса памяти и не соблюдены масштабы.

В данной главе показано, как работать с процессами, состоящими из единственного потока. О том, как использовать несколько потоков, рассказывается в главе 7.

Примечание

Рисунок 6.1 является высокоуровневым с точки зрения программиста представлением процесса. В действительности эта картина должна быть дополнена множеством технических деталей и особенностями реализации. Более подробную информацию заинтересованные читатели могут найти в книге Соломона (Solomon) и Руссиновича (Russinovich) *Inside Windows 2000.*

Процессы UNIX сопоставимы с процессами Windows, имеющими единственный поток.

Реализации UNIX недавно пополнились потоками POSIX Pthreads, которые в настоящее время используются почти повсеместно. В [40] потоки не обсуждаются; все рассмотрение основано на процессах.

Наверное, можно было бы даже не напоминать о том, что понятие потоков не является новым, и их различные реализации предлагаются поставщиками уже на протяжении целого ряда лет. Однако потоки Pthreads являются самым распространенным стандартом, в то время как коммерческие реализации потоков являются устаревшими.

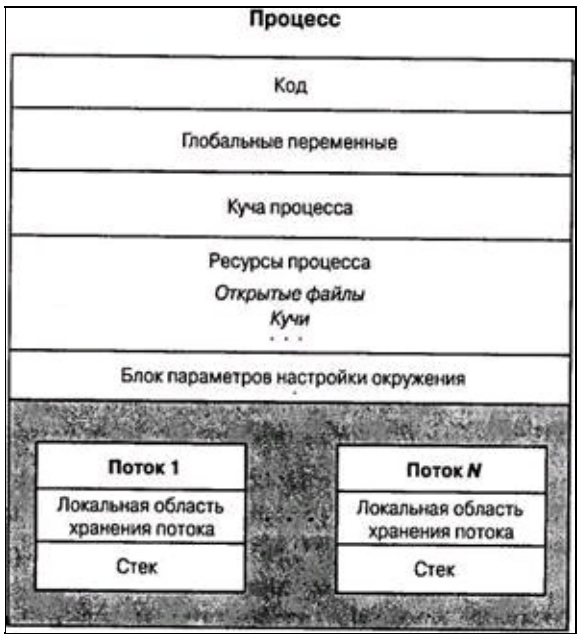


Рис. 6.1. Процесс и его потоки

Создание процесса

Одной из важнейших функций Windows, обеспечивающих управление процессами, является функция CreateProcess, которая создает новый процесс с единственным потоком. При вызове этой функции требуется указать имя файла исполняемой программы.

Обычно принято говорить о *процессах-предках,* или *родительских процессах* (parent processes), и *процессах-потомках,* или *дочерних процессах* (child processes), однако между процессами Windows эти отношения фактически не поддерживаются. Использование данной терминология является просто удобным способом выражения того факта, что один процесс порождается другим.

Гибкие и мощные возможности функции CreateProcess обеспечиваются ее десятью параметрами. На первых порах для упрощения работы целесообразно использовать значения параметров, заданные по умолчанию. Точно так же, как и в случае функции CreateFile, имеет смысл подробно рассмотреть каждый из параметров функции CreateProcess. Благодаря этому изучить другие аналогичные функции вам будет гораздо легче.

Прежде всего, заметьте, что возвращаемое значение функции не является дескриптором типа HANDLE; вместо этого функция возвращает два отдельных дескриптора, по одному для процесса и потока, передавая их в структуре, которая указывается при вызове функции. Эти дескрипторы относятся к создаваемому функцией CreateProcess новому процессу и его *основного* (primary) потока. Во избежание утечки ресурсов в процессе работы с примерами программ тщательно следите за своевременным закрытием обоих дескрипторов, когда они вам больше не нужны; забывчивость в отношении закрытия дескрипторов потоков является одной из самых распространенных ошибок. Закрытие дескриптора потока не приводит к прекращению ее выполнения; функция CloseHandle лишь удаляет ссылку на поток внутри процесса, вызвавшего функцию CreateProcess.

```
BOOL CreateProcess(lpApplicationName, LPTSTR lpCommandLine,
LPSECURITY_ATTRIBUTES lpsaProcess, LPSECURITY_ATTRIBUTES lpsaThread, BOOL
bInheritHandles, DWORD dwCreationFlags, LPVOID lpEnvironment, LPCTSTR
lpCurDir, LPSTARTUPINFO lpStartupInfo, LPPROCESS_INFORMATION lpProcInfo)
```

Возвращаемое значение: в случае успешного создания процесса и потока — TRUE, иначе — FALSE.

Параметры

Некоторые параметры потребуют дальнейшего подробного обсуждения в следующих разделах, тогда как смысл многих других станет для вас более понятным при рассмотрении примеров программ.

lpApplicationName и lpCommandLine (последний указатель имеет тип LPTSTR, а не LPCTSTR) — используются вместе для указания исполняемой программы и аргументов командной строки, о чем говорится в следующем разделе.

lpsaProcess и lpsaThread — указатели на структуры атрибутов защиты процесса и потока. Значениям NULL соответствует использование атрибутов защиты, заданных по умолчанию, и именно эти значения будут использоваться нами вплоть до главы 15, посвященной рассмотрению средств безопасности Windows.

bInheritHandles — показывает, наследует ли новый процесс наследуемые открытые

дескрипторы (файлов, отображений файлов и так далее) из вызывающего процесса. Наследуемые дескрипторы имеют те же атрибуты, что и исходные, и их обсуждение будет продолжено в одном из следующих разделов.

dwCreationFlags — может объединять в себе несколько флаговых значений, включая следующие:

• CREATE_SUSPENDED — указывает на то, что основной поток будет создан в приостановленном состоянии и начнет выполняться лишь после вызова функция ResumeThread.

• DETACHED_PROCESS и CREATE_NEW_CONSOLE — взаимоисключающие значения, которые не должны устанавливаться оба одновременно. Первый флаг означает создание нового процесса, у которого консоль отсутствует, а второй — процесса, у которого имеется собственная консоль. Если ни один из этих флагов не указан, то новый процесс наследует консоль родительского процесса.

• Create_New_Process_Group — указывает на то, что создаваемый процесс является корневым для новой группы процессов. Если все процессы, принадлежащие данной группе, разделяют общую консоль, то все они будут получать управляющие сигналы консоли (Ctrl-C или Ctrl-break). Обработчики управляющих сигналов консоли описывались в главе 4, а их применение было продемонстрировано в программе 4.5. Упомянутые группы процессов в некотором отношении аналогичны группам процессов UNIX и рассматриваются далее в этой главе.

Некоторые из флагов управляют приоритетами потоков нового процесса. О возможных значениях этих флагов более подробно говорится в главе 7. Пока же нам будет достаточно использовать приоритет родительского процесса (этот режим устанавливается по умолчанию) или указывать значение NORMAL_PRIORITY_CLASS.

lpEnvironment — указывает на блок параметров настройки окружения нового процесса. Если задано значение NULL, то новый процесс будет использовать значения параметров окружения родительского процесса. Блок параметров содержит строки, в которых заданы пары "имя-значение", определяющие, например, пути доступа к файлам.

lpCurDir — указатель на строку, содержащую путь к текущему каталогу нового процесса. Если задано значение NULL, то в качестве текущего каталога будет использоваться рабочий каталог родительского процесса.

lpStartupInfo — указатель на структуру, которая описывает внешний вид основного окна и содержит дескрипторы стандартных устройств нового процесса. Используйте соответствующую информацию из родительского процесса, которую можно получить при помощи функции GetStartupInfo. Можно поступить и по-другому, обнулив структуру STARTUPINFO перед вызовом функции CreateProcess. Для указания стандартных устройств ввода, вывода информации и вывода сообщений об ошибках следует определить значения полей дескрипторов стандартных устройств (hStdInput, hStdOutput и hStdError) в структуре STARTUPINFO. Чтобы эти значения не игнорировались, следует задать для другого элемента этой же структуры, а именно, элемента dwFlags, значение STARTF_USESTDHANDLES и определить все дескрипторы, которые потребуются дочернему процессу. Убедитесь в том, что эти дескрипторы являются наследуемыми и что при вызове функции CreateProcess значение параметра bInheritHandles установлено равным TRUE. Более подробная информация по этому вопросу, сопровождаемая соответствующим примером, приводится в разделе "Наследуемые дескрипторы".

lpProInfo — указатель на структуру, в которую будут помещены возвращаемые функцией значения дескрипторов и глобальных идентификаторов процесса и потока. Структура PROCESS_INFORMATION, о которой идет речь, имеет следующий вид:

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

Зачем процессам и потокам нужны еще и дескрипторы, если они снабжаются глобальными идентификаторами (ID)? Глобальные идентификаторы остаются уникальными для данного объекта на протяжении всего времени его существования и во всех процессах, тогда дескрипторов процесса может быть несколько и каждый из которых может характеризоваться собственным набором атрибутов, например определенными разрешениями доступа. В силу указанных причин одним функциям управления процессами требуется предоставлять идентификаторы процессов, а другим — дескрипторы. Кроме того, необходимость в дескрипторах процессов возникает при использовании универсальных функций, которые требуют указания дескрипторов. В качестве примера можно привести функции ожидания, обсуждаемые далее в этой главе, которые обеспечивают отслеживание переходов объектов различного типа, в том числе и процессов, указываемых с помощью дескрипторов, в определенные состояния. Точно так же, как и дескрипторы файлов, дескрипторы процессов и потоков должны закрываться сразу же после того, как необходимость в них отпала.

Примечание

Новый процесс получает информацию об окружении, рабочем каталоге и иную информацию в результате вызова функции CreateProcess. По завершении этого вызова любые изменения характеристик родительского процесса никак не отразятся на дочернем процессе. Так, после вызова функции CreateProcess рабочий каталог родительского процесса может измениться, но на дочерний процесс это не окажет никакого влияния, если только он сам не сменит рабочий каталог. Оба процесса полностью независимы друг от друга.

Модели процесса в UNIX и Windows значительно отличаются друг от друга. Прежде всего, в Windows отсутствует эквивалент UNIX-функции fork, создающей копию родительского процесса, включая его пространство данных, кучу и стек. В Windows трудно добиться точной эмуляции fork, но как ни расценивать последствия этого ограничения, остается фактом, что проблемы с использованием функции fork существуют и в многопоточных системах UNIX, поскольку любые попытки создания точной реплики многопоточной системы с копиями всех потоков и объектов синхронизации, особенно в случае SMP-систем, приводят к возникновению множества трудностей. Поэтому в действительности функция fork вообще плохо подходит для многопоточных систем.

В то же время, функция CreateProcess аналогична обычной для UNIX цепочке последовательных вызовов функций fork и execl (или одной из пяти остальных функций exec). В отличие от Windows пути доступа в UNIX определяются исключительно переменной среды PATH.

Как ранее уже отмечалось, отношения "предок-потомок" между процессами в Windows не поддерживаются. Так, выполнение дочернего процесса будет продолжаться даже после того, как завершится родительский процесс. Кроме того, в Windows отсутствуют группы процессов. Существует, однако, ограниченная форма

группы процессов, в которой все процессы получают управляющие события консоли.

Процессы Windows идентифицируются как дескрипторами, так и идентификаторами процессов, тогда как в UNIX дескрипторы процессов отсутствуют.

Указание исполняемого модуля и командной строки

Для указания имени файла исполняемого модуля используются как параметр lpApplicationName, так и параметр lpCommandLine. При этом действуют следующие правила:

• Указатель lpApplicationName, если его значение не равно NULL, указывает на строку, содержащую имя файла исполняемого модуля. Если имя модуля содержит пробелы, его следует заключить в кавычки. Более детальное описание приводится ниже.

• Если же значение указателя lpApplicationName равно NULL, то имя модуля определяется первой из лексем, переданных параметром lpCommandLine.

Обычно задается только параметр lpCommandLine, в то время как параметр lpApplicationName полагается равным NULL. Тем не менее, ниже приведены более подробные правила, которые определяют порядок использования этих двух параметров.

• Параметр lpApplicationName, если он не равен NULL, определяет исполняемый модуль. В строке, на которую указывает этот указатель, задайте полный путь доступа и имя файла или же ограничьтесь только именем файла, и тогда будут использоваться текущие диск и каталог; дополнительный поиск при этом производиться не будет. В имя файла включите расширение, например, .EXE или .BAT.

• Если значение параметра lpApplicationName равно NULL, то именем исполняемого модуля является первая из разделенных пробельными символами лексем, переданных параметром lpCommandLine. Если имя полный путь доступа не указан, то поиск файла осуществляется в следующем порядке:

1. Каталог модуля текущего процесса.
2. Текущий каталог.
3. Системный каталог Windows, информацию о котором можно получить с помощью функции GetSystemDirectory.
4. Каталог Windows, возвращаемый функцией GetWindowsDirectory.
5. Каталоги, перечисленные в переменной окружения PATH.

Новый процесс может получить командную строку посредством обычного argv-механизма или путем вызова функции GetCommandLine для получения командной строки в виде одиночной строки символов.

Заметьте, что командная строка не является строковой константой. Это согласуется с тем, что параметры argv главной программы не являются константами. Программа может модифицировать свои аргументы, хотя для внесения любых изменений рекомендуется использовать копию строки аргументов.

Вовсе не обязательно, чтобы новый процесс создавался с тем же определением UNICODE, что и родительский процесс. Возможны любые комбинации. Использование _tmain, как обсуждалось в главе 2, облегчает разработку программного кода, который сможет работать как с символами Unicode, так и с символами ASCII.

Наследуемые дескрипторы

Часто бывает так, что дочернему процессу требуется доступ к объекту, к которому можно

обратиться через дескриптор, определенный в родительском процессе, и если этот дескриптор — наследуемый, то дочерний процесс может получить копию открытого дескриптора родительского процесса. Часто именно так обеспечивается возможность использования дескрипторов стандартного ввода и вывода дочерним процессом. Преобразование дескриптора в наследуемый, чтобы дочерний процесс мог получить и использовать его копию, требует выполнения нескольких шагов.

Флаг bInheritHandles, который можно указать при вызове функции CreateProcess, определяет, будет ли дочерний процесс наследовать копии наследуемых дескрипторов открытых файлов, процессов и так далее. Этот флаг можно рассматривать как главный переключатель, действующий в отношении всех дескрипторов.

Кроме того, чтобы сделать наследуемым любой отдельный дескриптор, также требуется предпринимать специальные действия, поскольку дескрипторы не становятся таковыми по умолчанию. Создать наследуемый дескриптор можно либо путем использования структуры SECURITY_ATTRIBUTES в момент создания дескриптора, либо путем копирования существующего дескриптора.

В структуре SECURITY_ATTRIBUTES присутствует флаг bInheritHandle, значение которого должно быть установлено равным TRUE. Не забывайте также о том, что элемент nLength должен инициализироваться следующим значением:

```
sizeof(SECURITY_ATTRIBUTES)
```

Приведенный ниже фрагмент кода иллюстрирует создание наследуемых файловых или иных дескрипторов в типичных случаях. В этом примере дескриптор защиты в структуре атрибутов защиты установлен в NULL; подробнее об использовании дескрипторов защиты говорится в главе 15.

```
HANDLE h1, h2, h3;
SECURITY_ATTRIBUTES sa = { sizeof(SECURITY_ATTRIBUTES), NULL, TRUE };
…
h1 = CreateFile(…, &sa, …); /* Наследуемый. */
h2 = CreateFile(…, NULL, …); /* Ненаследуемый. */
h3 = CreateFile(…, &sa, …); /* Наследуемый. Возможно повторное использование структуры sa. */
```

Однако дочернему процессу значение наследуемого дескриптора пока еще не известно, и поэтому родительский процесс должен передать это значение дочернему процессу либо через механизм межпроцессного взаимодействия (Interprocess Communication, IPC), либо путем назначения дескриптора стандартному устройству ввода/вывода в структуре STARTUPINFO, как это делается в первом из примеров, приведенных в данной главе (программа 6.1), а также в ряде примеров в остальной части книги. Обычно последний метод является более предпочтительным, так как он позволяет перенаправить ввод/вывод стандартным способом без внесения каких-либо изменений в дочернюю программу.

В случае дескрипторов, которые не являются дескрипторами файлов или не используются для перенаправления ввода/вывода, применим другой способ, в соответствии с которым дескриптор преобразуется в текстовый формат и помещается в командную строку или переменную окружения. Такой подход можно использовать лишь в том случае, если дескриптор является наследуемым, поскольку и родительский, и дочерний процессы используют для идентификации дескриптора одно и то же значение. Один из способов реализации этого подхода предлагается в упражнении 6.2, а соответствующее решение приводится на Web-сайте книги.

Унаследованные дескрипторы представляют собой отдельные экземпляры. Поэтому родительский и дочерний процессы могут получить доступ к одному и тому же файлу, используя различные указатели файлов. Более того, каждый из обоих процессов может и должен

самостоятельно закрывать принадлежащий ему дескриптор.

На рис. 6.2 показан пример двух процессов с двумя различными таблицами дескрипторов, в которых с одним и тем же файлом или иным объектом связаны два различных дескриптора. Процесс 1 является родительским, процесс 2 — дочерним. Если принадлежащий дочернему процессу дескриптор был унаследован им, как это имеет место в случае дескрипторов 1 и 3, то значения дескрипторов в обоих процессах будут одинаковыми.

Однако подобные дескрипторы могут иметь и различные значения. Так, на файл D указывают два дескриптора, причем процесс 2 получил дескриптор за счет вызова функции CreateFile, а не путем наследования. Наконец, возможны ситуации, когда один из процессов имеет дескриптор объекта, а второй — не имеет, что наблюдается для файлов B и E. Так происходит в тех случаях, когда дескриптор создается дочерним процессом или дублируется из одного процесса в другой, о чем говорится в разделе "Дублирование дескрипторов".

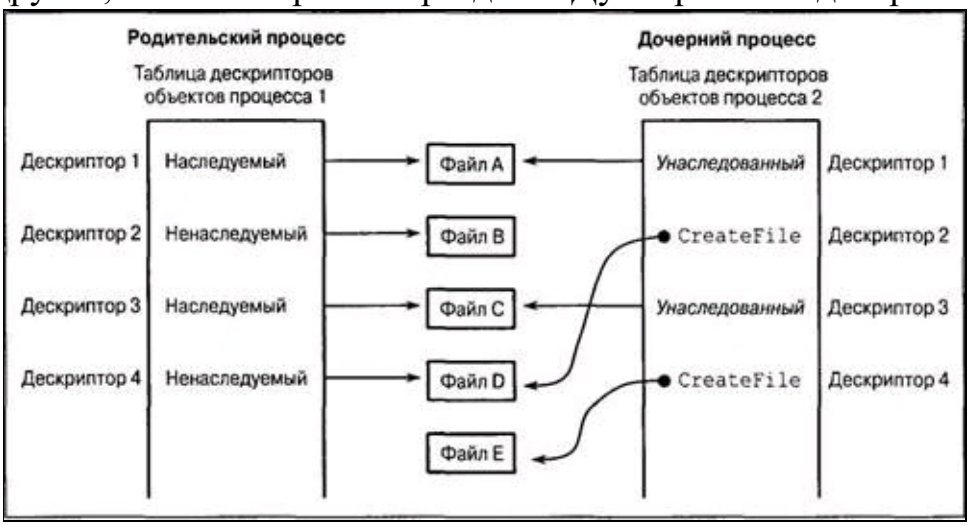


Рис. 6.2. Таблицы дескрипторов объектов для двух процессов

Счетчики дескрипторов процессов

Распространенной ошибкой программистов является пренебрежение закрытием дескрипторов после того, как необходимость в них отпала; это может стать причиной утечки ресурсов, что, в свою очередь, может приводить к снижению производительности или сбоям в программе и даже влиять на другие процессы. В версии NT 5.1 добавлена новая функция, позволяющая определить количество открытых дескрипторов, принадлежащих указанному процессу. Таким способом вы можете контролировать как собственный, так и другие процессы.

Приведенное ниже определение упомянутой функции не нуждается в отдельных пояснениях:

```
BOOL GetProcessHandleCount( HANDLE hProcess, PDWORD pdwHandleCount)
```

Идентификаторы процессов

Процесс может получить идентификатор и дескриптор нового дочернего процесса из структуры PROCESS_INFORMATION. Разумеется, закрытие дескриптора дочернего процесса не приводит к уничтожению самого процесса; становится невозможным лишь доступ к нему со стороны родительского процесса. Для получения идентификационной информации о текущем процессе служат две функции.

```
HANDLE GetCurrentProcess(VOID)
DWORD GetCurrentProcessId(VOID)
```

В действительности функция GetCurrentProcess возвращает *псевдодескриптор* (pseudohandle), который не является наследуемым. Это значение может использоваться вызывающим процессом всякий раз, когда ему требуется его собственный дескриптор. Реальный дескриптор процесса создается на основе идентификатора (ID) процесса, включая и тот, который возвращается функцией GetCurrentProcessID, путем использования функции OpenProcess. Как и в случае любого разделяемого объекта, при отсутствии надлежащих разрешений доступа попытка открытия объекта процесса окажется неуспешной.

```
HANDLE OpenProcess(DWORD dwDesiredAccess, BOOL bInheritHandle, DWORD dwProcessId)
```

Возвращаемое значение: в случае успешного завершения — дескриптор процесса, иначе — NULL.

Параметры

dwDesiredAccess — определяет права доступа к процессу. Некоторые из возможных значений этого параметра перечислены ниже.

• SYNCHRONIZE — разрешается использование дескриптора процесса в функциях ожидания завершения процесса, которые описываются далее в этой главе.

• PROCESS_ALL_ACCESS — устанавливаются все флаги доступа к процессу.

• PROCESS_TERMINATE — делает возможным завершение процесса с использованием функции TerminateProcess.

• PROCESS_QUERY_INFORMATION — разрешает использование дескриптора процесса в функциях GetExitCodeProcess и GetPriorityClass для получения информации о процессе.

bInheritHandle — позволяет указать, является ли новый дескриптор наследуемым. Параметр dwProcessID является идентификатором процесса, запрашивающего дескриптор.

Наконец, выполняющийся процесс может определить полный путь доступа к файлу исполняемого модуля, который использовался для его запуска, с помощью функций GetModuleFileName или GetModuleFileNameEx, при вызове которых значение параметра hModule должно устанавливаться равным NULL. При вызове этой функции из DLL будет возвращено имя файла DLL, а не .EXE-файла, который использует эту библиотеку DLL.

Дублирование дескрипторов

Родительскому и дочернему процессам может требоваться различный доступ к объекту, идентифицируемому дескриптором, который наследует дочерний процесс. Кроме того, процессу вместо псевдодескриптора, получаемого с помощью функции GetModuleFileName или GetModuleFileNameEx, может потребоваться реальный, наследуемый дескриптор, который мог бы использоваться дочерним процессом. Родительский процесс может обеспечить это, создав копию дескриптора с желаемыми разрешениями доступа и свойствами наследования. Функция, позволяющая создавать копии дескрипторов, имеет следующий вид:

```
BOOL DuplicateHandle(HANDLE hSourceProcessHandle, HANDLE hSourceHandle,
HANDLE hTargetProcessHandle, LPHANDLE lphTargetHandle, DWORD
dwDesiredAccess, BOOL bInheritHandle, DWORD dwOptions)
```

По завершении выполнения функции указатель lphTargetHandle будет указывать на копию исходного дескриптора, hSourceHandle. hSourceHandle является дескриптором дублируемого объекта в процессе, указанном дескриптором hSourceProcessHandle, и должен иметь права доступа PROCESS_DUP_HANDLE; если указанного дескриптора в исходном процессе не существует, функция DuplicateHandle завершается ошибкой. Новый дескриптор, на который указывает указатель lphTargetHandle, является действительным в целевом процессе, hTargetProcessHandle. Обратите внимание на то, что в нашем рассмотрении фигурировали три процесса, включая вызывающий. Часто в роли вызывающего процесса выступает целевой или исходный процесс, и тогда соответствующий дескриптор получают с помощью функции GetCurrentProcess. Заметьте также, что процесс может создать дескриптор в другом процессе; если вы это делаете, то вам потребуется механизм, с помощью которого можно было бы передать в другой процесс идентификационные данные нового дескриптора.

Функция DuplicateHandle может применяться к дескрипторам любого типа.

Если действие параметра dwDesiredAccess не отменяется флагом DUPLICATE_SAME_ACCESS параметра dwOptions, то у него может быть много возможных значений (для получения более подробных сведений обратитесь к библиотеке MSDN оперативного справочного руководства).

Параметр dwOptions может содержать любую комбинацию указанных ниже двух флагов.

- DUPLICATE_CLOSE_SOURCE — вызывает закрытие исходного дескриптора.
- DUPLICATE_SAME_ACCESS — вынуждает игнорировать параметр dwDesiredAccess.

Напоминание

Ядро Windows поддерживает счетчики ссылок для всех объектов; этот счетчик представляет количество различных дескрипторов, ссылающихся на данный объект. В то же время, приложения не имеют доступа к этому счетчику. Любой объект не может быть уничтожен до тех пор, пока не будет закрыт его последний дескриптор, а счетчик ссылок не примет нулевое значение. Унаследованные и продублированные дескрипторы считаются отличными от исходных и также учитываются в счетчике ссылок. Наследуемые дескрипторы используются в программе 6.1 далее в этой главе. В то же время, дескрипторы, переданные из одного процесса в другой посредством той или иной формы механизма IPC, не считаются независимыми, и поэтому если один процесс закрывает такой дескриптор, то другие процессы использовать его не могут. Подобной методикой пользуются редко, однако в упражнении 6.2 вам предлагается

передать значение унаследованного дескриптора из одного процесса в другой, используя механизм IPC.

Далее вы узнаете о том, как определить, завершено ли выполнение процесса.

Завершение и прекращение выполнения процесса

После того как процесс завершил свою работу, он, или, точнее, выполняющийся в этом процессе поток, может вызвать функцию ExitProcess, указав в качестве параметра кодом завершения (exit code):

```
VOID ExitProcess(UINT uExitCode)
```

Эта функция не осуществляет возврата. Вместо этого она завершает вызывающий процесс и все его потоки. Обработчики завершения игнорируются, но делаются все необходимые вызовы точек входа DllMain (см. главу 5) с кодом отключения от библиотеки. Код завершения связывается с процессом. Выполнение оператора return в основной программе с использованием кода возврата равносильно вызову функции ExitProcess, в котором этот код возврата указан в качестве кода завершения.

Другой процесс может определить код завершения, вызвав функцию GetExitCodeProcess:

```
BOOL GetExitCodeProcess(HANDLE hProcess, LPDWORD lpExitCode)
```

Процесс, идентифицируемый дескриптором hProcess, должен обладать правами доступа PROCESS_QUERY_INFORMATION (см. описание функции OpenProcess, которая нами уже обсуждалась). lpExitCode указывает на переменную типа DWORD, которая принимает значение кода завершения. Одним из ее возможных значений является STILL_ACTIVE, означающее, что данный процесс еще не завершился.

Наконец, один процесс может прекратить выполнение другого процесса, если у дескриптора завершаемого процесса имеются права доступа PROCESS_TERMINATE. При вызове функции завершения процесса указывается код завершения.

```
BOOL TerminateProcess(HANDLE hProcess, UINT uExitCode)
```

Предостережение

Прежде чем завершить выполнение процесса, убедитесь в том, что все ресурсы, которые он мог разделять с другими процессами, освобождены. В частности, должны быть корректно обработаны ресурсы синхронизации, о которых говорится в главе 8 (мьютексы, семафоры и события). В этом отношении могут оказаться полезными SEH (глава 4), а вызов функции ExitProcess может быть осуществлен из обработчика. В то же время, при вызове функции ExitProcess обработчики __finally и __except *не* выполняются, поэтому в идее завершения выполнения изнутри программы нет ничего хорошего. Особенно рискованно применять функцию TerminateProcess, поскольку у завершаемого процесса в этом случае отсутствует возможность выполнить свои SEH или вызвать функции DllMain связанных с ним библиотек DLL. Ограниченной альтернативой являются обработчики управляющих сигналов консоли, обеспечивающие возможность передачи сигнала одним процессом другому, который после этого может корректно организовать свое завершение.

Программа 6.3 иллюстрирует применение методики, обеспечивающей взаимодействие между процессами. В этом примере один процесс посылает другому процессу запрос завершения выполнения, получив который второй процесс сможет аккуратно завершить свою

работу.

Процессы UNIX имеют свои идентификаторы, pid, которые сопоставимы с идентификаторами процессов Windows. Функция getpid аналогична функции GetCurrentProcessID, но эквивалентов функциям getppid и getgpid в Windows не находится ввиду отсутствия предков процессов и групп процессов.

И, наоборот, в UNIX отсутствуют дескрипторы процессов, и поэтому в ней нет функций, которые можно было бы сравнить с функциями GetCurrentProcess или OpenProcess.

В UNIX допускается использование дескрипторов (descriptors) открытых файлов после вызова функции exec, если для дескриптора файла не был установлен флаг close-on-exec. Это правило применимо только к дескрипторам файлов, которые, в силу вышесказанного, можно сравнить с наследуемыми дескрипторами (handles) файлов Windows.

Функция UNIX exit, которая фактически является функцией библиотеки C, аналогична функции ExitProcess; чтобы прекратить выполнение другого процесса ему следует послать сигнал SIGKILL.

Ожидание завершения процесса

Простейшим, но наряду с этим и обладающим наиболее ограниченными возможностями, методом синхронизации с другим процессом является ожидание его завершения. Представленные ниже стандартные функции ожидания Windows обладают рядом интересных свойств.

• Функции ожидания могут работать с самыми различными типами объектов; дескрипторы процессов являются лишь самым первым из рассматриваемых нами примеров применения этих функций.

• Эти функции могут ожидать завершения одного процесса, первого из нескольких указанных процессов или всех процессов, образующих группу.

• Существует возможность устанавливать конечный интервал ожидания (time-out).

Обе рассмотренных ниже функции ожидают перехода объекта синхронизации в *сигнальное* состояние. Например, система переводит процесс в сигнальное состояние, когда он завершается или его выполнение прекращается извне. Функциями ожидания, которые мы будем впоследствии неоднократно использовать, являются следующие функции:

```
DWORD WaitForSingleObject(HANDLE hObject, DWORD dwMilliseconds)

DWORD WaitForMultipleObjects(DWORD nCount, CONST HANDLE *lpHandles,
BOOL fWaitAll, DWORD dwMilliseconds)
```

Возвращаемое значение: указывает причину завершения ожидания или, в случае ошибки, равно 0xFFFFFFFF (для получения более подробной информации используйте функцию GetLastError).

В аргументах этих функций указывается либо дескриптор одиночного процесса (hObject), либо дескрипторы ряда отдельных объектов, хранящиеся в массиве, на который указывает указатель lpHandles. Значение параметра nCount, определяющего размер массива, не должно превышать значение MAXIMUM_WAIT_OBJECTS (определено равным 64 в файле WINNT.H).

dwMilliseconds — число миллисекунд интервала ожидания. Если значение этого параметра равно 0, то возврат из функции осуществляется сразу же после проверки состояния указанного объекта, что позволяет программе опрашивать процессы для определения их состояния завершения. Если же значение этого параметра равно INFINITE, то ожидание длится до тех пор, пока ожидаемый процесс не завершится.

fWaitAll — параметр второй функции, указывающий (если его значение равно TRUE) на необходимость ожидания завершения всех процессов, а не только одного.

Возможными возвращаемыми значениями этой функции в случае ее успешного завершения являются следующие:

• WAIT_OBJECT_0 — означает, что указанный объект перешел в сигнальное состояние (в случае функции WaitForSingleObject) или что одновременно все nCount объектов перешли в сигнальное состояние (в специальном случае функции WaitForMultipleObject, когда значение параметра fWaitAll равно TRUE).

• WAIT_OBJECT_0+n, где $0 \leq n < nCount$ — вычтите значение WAIT_OBJECT_0 из возвращенного значения, чтобы определить, выполнение какого именно процесса завершилось, когда ожидается завершение выполнения любого из группы процессов. Если в сигнальное состояние перешли несколько объектов, то возвращается наименьшее из возможных значений. WAIT_ABANDONED является возможным базовым значением в случае использования

дескрипторов мьютексов; см. главу 8.

• WAIT_TIMEOUT — указывает на то, что в течение отведенного периода ожидания сигнализируемый объект (объекты) не смогли удовлетворить условию ожидания.

• WAIT_FAILED — означает неудачное завершение функции, вызванное, например, тем, что у дескриптора отсутствовали права доступа SYNCHRONIZE.

• WAIT_ABANDONED_0 — это значение невозможно в случае процессов и рассматривается в главе 8 при рассмотрении мьютексов.

Для определения кода завершения процесса используется функция GetExitCodeProcess, описанная в предыдущем разделе.

Блоки и строки окружения

Схема, представленная на рис. 6.1, включает блок окружения процесса. Блок окружения (environment block) процесса содержит последовательность строк вида:

```
Имя = Значение
```

Каждая строка окружения (environment string), будучи символьной строкой, заканчивается нулевым символом, а весь блок строк в целом завершается дополнительным нулевым символом. Одним из примеров широко используемых переменных среды является переменная PATH.

Чтобы передать информацию об окружении из родительского процесса в дочерний, параметр lpEnvironment при вызове функции CreateProcess следует установить равным NULL. В свою очередь, любой процесс может запросить или изменить свои переменные окружения или добавить новые в блок окружения.

Для получения, а также создания новых или изменения существующих переменных окружения используются следующие функции:

```
    DWORD GetEnvironmentVariable(LPCTSTR lpName, LPTSTR lpValue, DWORD
cchValue)
    BOOL SetEnvironmentVariable(LPCTSTR lpName, LPCTSTR lpValue)
```

lpName — указатель на строку, содержащую имя переменной окружения. После определения переменной окружения она добавляется в блок окружения при условии, что такая переменная ранее не существовала, а определяемое значение не равно NULL. Если же определяемое значение равно NULL, то переменная удаляется из блока. Строка значения не может содержать символы "=".

В случае успешного завершения функция GetEnvironmentVariable возвращает длину строки значения переменной окружения, иначе — 0. Если размер буфера lpValue, указанный значением параметра cchValue, оказался недостаточно большим, то возвращаемое значение равно количеству символов, которое фактически требуется для сохранения значения переменной. Вспомните, что аналогичный механизм используется и в функции GetCurrentDirectory (глава 2).

Защита процесса

Обычно функция CreateProcess предоставляет права доступа к процессу на уровне PROCESS_ALL_ACCESS. Однако имеется возможность определения детализированных прав доступа, из которых в качестве примера можно назвать права доступа PROCESS_QUERY_INFORMATION, CREATE_PROCESS, PROCESS_TERMINATE, PROCESS_SET_INFORMATION, DUPLICATE_HANDLE и CREATETHREAD. В частности, с учетом возможных рисков, которые могут подстерегать вас в случае принудительного завершения выполняющихся процессов, на что мы уже неоднократно обращали ваше внимание, может оказаться полезным ограничить предоставление прав доступа к процессам на уровне PROCESS_TERMINATE для родительского процесса. Подробнее об атрибутах защиты процессов и других объектов говорится в главе 15.

> В UNIX для ожидания завершения процессов используются функции wait и waitpid, однако отсутствует понятие интервала ожидания, хотя функция waitpid может опрашивать процессы (существует возможность ее вызова без блокировки). Эти функции способны ожидать лишь завершения дочерних процессов, и эквивалентных

им функций, применимых к ряду процессов, не существует, хотя и возможно ожидание завершения всех процессов, относящихся к одной группе. Кроме того, имеется одно незначительное отличие, заключающееся в том, что функции wait и waitpid возвращают код завершения сами, в результате чего отпадает необходимость в вызове отдельной функции, эквивалентной функции GetExitCodeProcess.

Строки окружения, аналогичные строкам окружения Windows, поддерживаются и в UNIX. Функция getenv (входящая в библиотеку C) имеет те же самые функциональные возможности, что и функция GetEnvironmentVariable, но программист сам должен заботиться о необходимом размере буфера. Функции putenv, setenv и unsetenv обеспечивают различные способы добавления, изменения и удаления переменных окружения и их значений, предлагая функциональность, аналогичную функциональности SetEnvironmentVariable.

Пример: параллельный поиск указанного текстового шаблона

Настало время посмотреть на процессы Windows в действии. Приведенная ниже в качестве примера программа grepMP создает процессы для поиска указанного текстового шаблона в файлах, по одному процессу на каждый файл. Эта программа моделирует UNIX-утилиту grep, хотя используемая нами методика применима к любой программе, которая полагается на стандартный вывод. Рассматривайте программу поиска как "черный ящик" и считайте, что она является просто исполняемой программой, выполнение которой должно контролироваться родительским процессом.

Командная строка программы имеет следующий вид:

```
grepMP шаблон F1 F2 … FN
```

Программа 6.1 выполняет следующие виды обработки:

• Для поиска указанного шаблона в каждом из входных файлов, от F1 до FN, используется отдельный процесс, запускающий один и тот же исполняе мый модуль. Для каждого процесса программа создает командную строку такого вида: grep *шаблон* FK.

• Полю hStdOut структуры STARTUPINFO нового процесса присваивается значение дескриптора временного файла, который определяется как наследуемый.

• Программа организует ожидание завершения всех процессов поиска, используя для этого функцию WaitForMultipleObjects.

• По завершении всех процессов поиска осуществляется поочередный вывод результатов (временных файлов). Вывод временного файла осуществляет процесс, выполняющий утилиту cat (программа 2.3).

• Возможности функции WaitForMultipleObjects ограничиваются лишь максимально допустимым количеством дескрипторов, которое устанавливается значением MAXIMUM_WAIT_OBJECTS (64), поэтому она вызывается многократно.

• Для определения успешности попытки нахождения данным процессом заданного шаблона программа использует код завершения процесса grep.

Порядок обработки файлов программой 6.1 иллюстрируется на рис. 6.3.

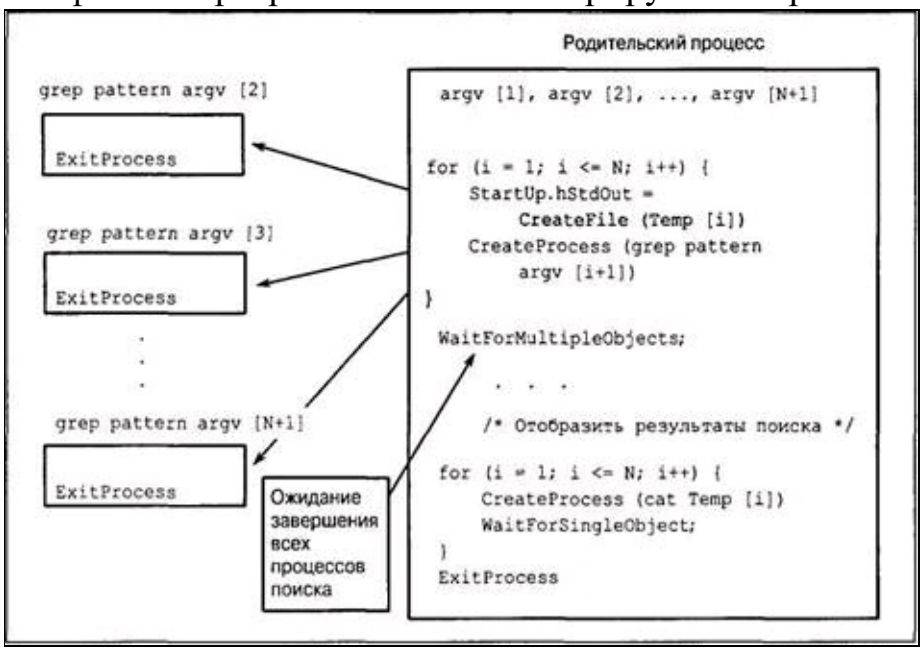


Рис. 6.3. Поиск текстового шаблона в файлах с использованием нескольких процессов

Программа 6.1. grepMP: выполнение параллельного поиска текстового шаблона

```
    /* Глава 6. grepMP. */
    /* Версия команды grep, использующая несколько процессов. */
    #include "EvryThng.h"

    int _tmain(DWORD argc, LPTSTR argv[])
    /* Для выполнения поиска в каждом из файлов, указанных в командной строке,
создается отдельный процесс. Каждому процессу предоставляется временный файл в
текущем каталоге, в котором сохраняются результаты. */
    {
     HANDLE hTempFile;
      SECURITY_ATTRIBUTES StdOutSA = /* Атрибуты защиты для наследуемого
дескриптора. */
      {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
     TCHAR CommandLine[MAX_PATH + 100];
     STARTUPINFO StartUpSearch, Startup;
     PROCESS_INFORMATION ProcessInfo;
     DWORD iProc, ExCode;
     HANDLE *hProc; /* Указатель на массив дескрипторов процессов. */
     typedef struct {TCHAR TempFile[MAX_PATH];} PROCFILE;
     PROCFILE *ProcFile; /* Указатель на массив имен временных файлов. */
     GetStartupInfo(&StartUpSearch);
     GetStartupInfo(&StartUp);
     ProcFile = malloc((argc - 2) * sizeof(PROCFILE));
     hProc = malloc((argc - 2) * sizeof(HANDLE));
     /* Создать для каждого файла отдельный процесс "grep". */
     for (iProc = 0; iProc < argc - 2; iProc++) {
      _stprintf(CommandLine, _T("%s%s %s"), _T("grep "), argv[1], argv[iProc + 2]);
       GetTempFileName(_T("."), _T("gtm"), 0, ProcFile[iProc].TempFile); /* Для
хранения результатов поиска.*/
      hTempFile = /* Этот дескриптор является наследуемым */
          CreateFile(ProcFile[iProc].TempFile, GENERIC_WRITE, FILE_SHARE_READ |
FILE_SHARE_WRITE, &StdOutSA, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
      StartUpSearch.dwFlags = STARTF_USESTDHANDLES;
      StartUpSearch.hStdOutput = hTempFile;
      StartUpSearch.hStdError = hTempFile;
      StartUpSearch.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
      /* Создать процесс для выполнения командной строки. */
        CreateProcess(NULL, CommandLine, NULL, NULL, TRUE, 0, NULL, NULL,
&StartUpSearch, &ProcessInfo);
      /* Закрыть ненужные дескрипторы. */
      CloseHandle(hTempFile);
      CloseHandle(ProcessInfo.hThread);
      hProc[iProc] = ProcessInfo.hProcess;
     }
     /* Выполнить все процессы и дождаться завершения каждого из них. */
      for (iProc = 0; iProc < argc - 2; iProc += MAXIMUM_WAIT_OBJECTS)
WaitForMultipleObjects( /* Разрешить использование достаточно большого количества
процессов */
      min(MAXIMUM_WAIT_OBJECTS, argc - 2 - iProc), &hProc [iProc], TRUE, INFINITE);
      /* Переслать результирующие файлы на стандартный вывод с использованием
утилиты cat */
     for (iProc = 0; iProc < argc - 2; iProc++) {
      if (GetExitCodeProcess(hProc[iProc], &ExCode) && ExCode==0) {
       /* Обнаружен шаблон — Вывести результаты. */
       if (argc > 3) _tprintf(_T("%s:\n"), argv [iProc + 2]);
       fflush(stdout); /* Использование стандартного вывода несколькими процессами.
```

```
*/
       _stprintf(CommandLine, _T("%s%s"), _T("cat "), ProcFile[iProc].TempFile);
        CreateProcess(NULL, CommandLine, NULL, NULL, TRUE, 0, NULL, NULL, &StartUp,
&ProcessInfo);
       WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
       CloseHandle(ProcessInfo.hProcess);
       CloseHandle(ProcessInfo.hThread);
      }
      CloseHandle(hProc [iProc]);
      DeleteFile(ProcFile[iProc].TempFile);
     }
     free(ProcFile);
     free(hProc);
     return 0;
    }
```

Процессы в многопроцессорной среде

В программе 6.1 процессы и их основные (и только эти) потоки выполняются практически полностью независимо друг от друга. Единственная зависимость между ними проявляется лишь в конце выполнения родительского процесса, поскольку он ожидает завершения выполнения каждого из них, чтобы перейти к последовательной обработке выходных файлов. Поэтому в SMP-системах планировщик Windows может и будет обеспечивать параллельное выполнение потоков процесса на нескольких независимых процессорах. В результате этого производительность, если оценивать ее по времени выполнения всей программы, значительно повышается, причем для этого с вашей стороны не требуется предпринимать никаких действий.

Типичные результаты тестирования производительности приведены в приложении В. Ввиду выполнения программой ряда вспомогательных операций, а также необходимости последовательного вывода результатов, зависимость производительности от количества процессоров не является линейной. Тем не менее, улучшение производительности налицо, и это автоматически обеспечивается организацией программы, которая предусматривает передачу выполнения независимых вычислительных задач независимым процессам.

Вместе с тем, существует возможность привязки процессов к определенным процессорам, что позволяет всегда быть уверенным в том, что другие процессоры остаются свободными и их можно использовать для решения каких-либо иных, критических задач. Это достигается за счет применения маски родства процессора (processor affinity mask) (см. главу 9) в объекте задачи. Объекты задач (job objects) описываются в одном из следующих разделов настоящей главы.

Наконец, внутри процесса можно создавать независимые потоки, и для этих потоков также будет спланировано выполнение с использованием отдельных процессоров SMP для каждого из них. Связь между использованием потоков и показателями производительности обсуждается в главе 7.

Временные характеристики процесса

Воспользовавшись функцией GetProcessTimes, которая в Windows 9x отсутствует, можно получить различные временные характеристики процесса, а именно: истекшее время (elapsed time), время, затраченное ядром (kernel time), и пользовательское время (user time).

```
BOOL GetProcessTimes(HANDLE hProcess, LPFILETIME lpCreationTime,
LPFILETIME lpExitTime, LPFILETIME lpKernelTime, LPFILETIME lpUserTime)
```

Дескриптор процесса может ссылаться как на процесс, который продолжает выполняться, так и на процесс, выполнение которого прекратилось. Вычитая время создания процесса (creation time) из времени завершения процесса (exit time), мы получаем истекшее время, как показано в следующем примере. Тип данных FILETIME является 64-битовым; для вычисления указанной разности объедините переменную этого типа с переменной тип LARGE_INTEGER в структуру типа union. Ранее преобразование и отображение отметок времени файлов было продемонстрировано в главе 3 на примере программы lsw.

Функция GetThreadTimes аналогична только что описанной, но требует указания дескриптора потока в качестве параметра. Управлению потоками посвящена глава 7.

Пример: временные характеристики процессов

Наш следующий пример (программа 6.2) представляет собой команду timep (от *time print* — вывод временных параметров), аналогичную UNIX-команде time (поскольку команда time поддерживается процессором командной строки, мы должны использовать для нашей команды другое имя). Программа позволяет вывести все три временные характеристики, однако в Windows 9x будет доступно лишь истекшее время процесса.

Одним из возможных применений этой команды является сравнительный анализ времени выполнения и эффективности различных версий функций копирования и преобразования файлов из ASCII в Unicode, реализованных в предыдущих главах.

В данной программе используется функция Windows GetCommandLine, которая возвращает целую командную строку, а не отдельные строки из массива argv.

Кроме того, программа использует вспомогательную функцию SkipArg, которая просматривает командную строку и устанавливает в ней указатель в позицию, непосредственно следующую за именем исполняемого файла. Листинг функции SkipArg приведен в приложении А.

Для определения версии ОС в программе 6.2 используется функция GetVer-sionEx. В операционных системах Windows 9x и Windows CE доступным будет лишь истекшее время процесса. Программный код для этих систем представлен с той целью, чтобы показать, что в некоторых случаях работоспособность программ, по крайней мере — с частичным сохранением их функциональности, удается обеспечивать для целого диапазона различных версий Windows.

Программа 6.2. timep: временные характеристики процессов

```
    /* Глава 6. timep. */
    #include "EvryThng.h"

    int _tmain(int argc, LPTSTR argv[]) {
     STARTUPINFO Startup;
     PROCESS_INFORMATION ProcInfo;
     union { /* Эта структура используется для выполнения арифметических операций с
участием временных параметров. */
      LONGLONG li;
      FILETIME ft;
     } CreateTime, ExitTime, ElapsedTime;
     FILETIME KernelTime, UserTime;
     SYSTEMTIME ElTiSys, KeTiSys, UsTiSys, StartTimeSys, ExitTimeSys;
     LPTSTR targv = SkipArg(GetCommandLine());
     OSVERSIONINFO OSVer;
     BOOL IsNT;
     HANDLE hProc;
     OSVer.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
     GetVersionEx(&OSVer);
     IsNT = (OSVer.dwPlatformId == VER_PLATFORM_WIN32_NT);
     /* NT (все версии) возвращает VER_PLATFORM_WIN32_NT. */
     GetStartupInfo(&StartUp);
     GetSystemTime(&StartTimeSys);
     /* Выполнить командную строку; дождаться завершения процесса. */
      CreateProcess (NULL, targv, NULL, NULL, TRUE, NORMAL_PRIORITY_CLASS, NULL,
NULL, &StartUp, &ProcInfo);
     /* Убедиться в наличии ВСЕХ НЕОБХОДИМЫХ прав доступа к процессу. */
```

```
      DuplicateHandle(GetCurrentProcess(), ProcInfo.hProcess, GetCurrentProcess(),
&hProc, PROCESS_QUERY_INFORMATION | SYNCHRONIZE, FALSE, 0);
     WaitForSingleObject(hProc, INFINITE);
     GetSystemTime (&ExitTimeSys);
      if (IsNT) { /* Windows NT. Для процесса вычисляется истекшее время, время
выполнения в режиме ядра и время выполнения в пользовательском режиме. */
      GetProcessTimes(hProc, &CreateTime.ft, &ExitTime.ft, &KernelTime, &UserTime);
      ElapsedTime.li = ExitTime.li - CreateTime.li;
      FileTimeToSystemTime(&ElapsedTime.ft, &ElTiSys);
      FileTimeToSystemTime(&KernelTime, &KeTiSys);
      FileTimeToSystemTime(&UserTime, &UsTiSys);
          _tprintf(_T("Истекшее   время:   %02d:%02d:%02d:%03d\n"),   ElTiSys.wHour,
ElTiSys.wMinute, ElTiSys.wSecond, ElTiSys.wMilliseconds);
       _tprintf(_T("Пользовательское время: %02d:%02d:%02d:%03d\n"), UsTiSys.wHour,
UsTiSys.wMinute, UsTiSys.wSecond, UsTiSys.wMilliseconds);
          _tprintf(_T("Системное   время:   %02d:%02d:%02d:%03d\n"),   KeTiSys.wHour,
KeTiSys.wMinute, KeTiSys.wSecond, KeTiSys.wMilliseconds);
     } else {
      /* Windows 9x и CE. Вычисляется лишь истекшее время. */
      …
     }
     CloseHandle(ProcInfo.hThread);
     CloseHandle(ProcInfo.hProcess);
     CloseHandle(hProc);
     return 0;
    }
```

Использование команды timep

Теперь мы можем воспользоваться командой timep для анализа производительности различных вариантов программ копирования файлов и их преобразования из ASCII в Unicode, таких, например, как утилиты atou (программа 2.4) и sortMP (программа 5.5). Некоторые из полученных результатов и краткий их анализ представлены в приложении B.

Обратите внимание, что для таких программ, как grepMP, тестирование предоставляет системное и пользовательское время только для родительских процессов. Объекты задач, описанные в конце настоящей главы, позволяют собрать информацию, касающуюся группы процессов. Как показано в приложении B, в случае SMP-систем производительность может повышаться за счет того, что отдельные процессы, вернее, потоки, выполняются на различных процессорах. Выигрыш в производительности возможен и в тех случаях, когда файлы располагаются на различных физических дисках.

Генерация управляющих событий консоли

Прерывание выполнения процесса извне может порождать проблемы, поскольку это лишает процесс возможности произвести необходимую завершающую обработку данных и очистку ресурсов. Воспользоваться SEH в данном случае нельзя ввиду того, что не существует общего метода, который позволял бы одному процессу возбуждать исключения в другом [25]. В то же время, с учетом некоторых ограничений, механизм управляющих событий консоли делает возможной передачу одним процессом другому управляющих сигналов, или событий, консоли. В программе 4.5 было продемонстрировано, как установить обработчик для перехвата сигналов и организовать генерацию исключений этим обработчиком. В указанном примере сигнал генерировался по приказу пользователя средствами пользовательского интерфейса.

Таким образом, вполне можно добиться того, чтобы один процесс генерировал сигнал, соответствующий определенному событию, в другом указанном процессе или группе процессов. Вспомните флаг CREATE_NEW_PROCESS_GROUP функции CreateProcess. Если этот флаг установлен, то идентификатор нового процесса идентифицирует группу процессов и является корневым (root) процессом данной группы. Все новые процессы, создаваемые данным родительским процессом, будут автоматически попадать в эту группу до тех пор, пока при вызове функции CreateProcess не будет использован флаг CREATE_NEW_PROCESS_GROUP. Сгруппированные процессы аналогичны группам процессов в UNIX.

Процесс может генерировать события CTRL_C_EVENT или CTRL_BREAK_EVENT в указанной группе процессов, идентифицируя ее с помощью идентификатора корневого процесса. Консоль целевых процессов должна совпадать с консолью процесса, генерирующего событие. В частности, вызывающий процесс не может быть создан с использованием собственной консоли (посредством флагов CREATE_NEW_CONSOLE или DETACHED_PROCESS).

```
BOOL GenerateConsoleCtrlEvent(DWORD dwCtrlEvent, DWORD dwProcessGroup)
```

Тогда значением первого параметра должно быть либо CTRL_C_EVENT, либо CTRL_BREAK_EVENT. Второй параметр идентифицирует группу процессов.

Пример: простое управление задачами

Оболочки UNIX предоставляют команды, позволяющие выполнять процессы в фоновом режиме и получать их текущее состояние. В этом разделе разрабатывается простой "процессор задач" ("job shell") с аналогичным набором команд, перечень которых приводится ниже.

• jobbg — использует остальную часть командной строки в качестве командной строки для нового процесса, или *задачи* (job), однако возврат из команды осуществляется немедленно, без ожидания завершения нового процесса. По желанию пользователя новый процесс может либо получить собственную консоль, либо выполняться как *отсоединенный* (detached) процесс, то есть как процесс, связь с которым не поддерживается. Этот подход аналогичен запуску команд UNIX с указанием опции & в конце команды.

• jobs — выводит список текущих активных задач, снабжая каждую из задач порядковым номером и идентификатором процесса. Эта команда аналогична одноименной команде UNIX.

• kill — прекращает выполнение задачи. В данной реализации используется функция TerminateProcess, которая, как ранее уже отмечалось, не обеспечивает корректного завершения задачи, сопровождающегося "уборкой мусора". Доступна также опция, позволяющая передавать управляющие сигналы консоли.

Создать дополнительные команды, позволяющие приостанавливать существующие задачи или переводить их в фоновый режим, вам будет несложно.

Поскольку выполнение оболочки, которая поддерживает список задач, может быть прекращено, она использует специфический для каждого пользователя разделяемый файл, в котором содержатся идентификаторы процессов, команды и другая необходимая информация. Благодаря этому перезапуск оболочки никак не отразится на списке задач. В одном из упражнений вам предлагается применять для хранения этой информации не временный файл, а реестр.

Реализация программы наталкивается на определенные проблемы, связанные с параллельным выполнением задач. Некоторые процессы, запущенные из командных строк различных оболочек, могут одновременно пытаться управлять задачами. Чтобы справиться с этим, функции управления задачами используют блокировки (глава 3) в файле списка задач, в результате чего пользователь может активизировать управление задачами из различных оболочек или процессов.

В полном варианте программы, находящемся на Web-сайте книги, содержится ряд дополнительных возможностей, не представленных в приводимых листингах, например, возможность получения входных данных для командной строки из файла. Программа JobDhell послужит основой для создания более общего "процессора служб" ("service processor") в главе 13 (программа 13.3). Службы NT являются фоновыми процессами, обычно — серверами, управление которыми осуществляется командами запуска, остановки, приостановки, а также другими командами.

Создание фоновых задач

Программа 6.3 реализует процессор задач, в котором пользователю предлагается ввести одну из трех возможных команд для их дальнейшего выполнения программой. В этой программе используется набор функций управления задачами, представленный программами 6.4, 6.5 и 6.6.

Программа 6.3. JobShell: создание, вывод списка и прекращение выполнения фоновых задач

```
/* Глава 6. */
/* JobShell.c - команды управления задачами:
   jobbg — Выполнить задачу в фоновом режиме.
   jobs — Вывести список всех фоновых задач.
   kill — Прекратить выполнение указанной задачи из семейства задач.
   Существует опция, позволяющая генерировать управляющие сигналы консоли. */
#include "EvryThng.h"
#include "JobMgt.h"

int _tmain(int argc, LPTSTR argv[]) {
 BOOL Exit = FALSE;
 TCHAR Command[MAX_COMMAND_LINE + 10], *pc;
 DWORD i, LocArgc; /* Локальный параметр argc. */
 TCHAR argstr[MAX_ARG][MAX_COMMAND_LINE];
 LPTSTR pArgs[MAX_ARG];
 for (i = 0; i < MAX_ARG; i++) pArgs[i] = argstr[i];
 /* Вывести подсказку пользователю, считать команду и выполнить ее. */
 _tprintf(_T("Управление задачами Windows\n"));
 while (!Exit) {
  _tprintf(_T("%s"), _T("JM$"));
  _fgetts(Command, MAX_COMMAND_LINE, stdin);
  pc = strchr(Command, '\n');
  *pc = '\0';
   /* Выполнить синтаксический разбор входных данных с целью получения командной
строки для новой задачи. */
   GetArgs(Command, &LocArgc, pArgs); /* См. Приложение A. */
   CharLower(argstr[0]);
   if(_tcscmp(argstr[0], _T("jobbg")) == 0) {
    Jobbg(LocArgc, pArgs, Command);
   } else if(_tcscmp(argstr[0], _T("jobs")) == 0) {
    Jobs(LocArgc, pArgs, Command);
   } else if(_tcscmp(argstr[0], _T("kill")) == 0) {
    Kill(LocArgc, pArgs, Command);
   } else if(_tcscmp(argstr[0], _T("quit")) == 0) {
    Exit = TRUE;
   } else _tprintf(_T("Такой команды не существует. Повторите ввод\n"));
 }
 return 0;
}

/* jobbg [параметры] командная строка [Параметры являются взаимоисключающими]
 -c: Предоставить консоль новому процессу.
 -d: Отсоединить новый процесс без предоставления ему консоли.
 Если параметры не заданы, процесс разделяет консоль с jobbg. */
int Jobbg(int argc, LPTSTR argv[], LPTSTR Command) {
 DWORD fCreate;
 LONG JobNo;
 BOOL Flags[2];
 STARTUPINFO Startup;
 PROCESS_INFORMATION ProcessInfo;
 LPTSTR targv = SkipArg(Command);
 GetStartupInfo(&StartUp);
 Options(argc, argv, _T("cd"), &Flags[0], &Flags[1], NULL);
 /* Пропустить также поле параметра, если он присутствует. */
 if (argv[1][0] == '-') targv = SkipArg(targv);
 fCreate = Flags[0] ? CREATE_NEW_CONSOLE : Flags [1] ? DETACHED_PROCESS : 0;
  /* Создать приостановленную задачу/поток. Возобновить выполнение после ввода
```

```
номера задачи. */
     CreateProcess(NULL, targv, NULL, NULL, TRUE, fCreate | CREATE_SUSPENDED |
CREATE_NEW_PROCESS_GROUP, NULL, NULL, &StartUp, &ProcessInfo);
     /* Создать номер задачи и ввести ID и дескриптор процесса в "базу данных"
задачи. */
     JobNo = GetJobNumber(&ProcessInfo, targv); /* См. "JobMgt.h" */
     if (JobNo >= 0) ResumeThread(ProcessInfo.hThread);
     else {
      TerminateProcess(ProcessInfo.hProcess, 3);
      CloseHandle(ProcessInfo.hProcess);
      ReportError(_T("Ошибка: Не хватает места в списке задач."), 0, FALSE);
      return 5;
     }
     CloseHandle(ProcessInfo.hThread);
     CloseHandle(ProcessInfo.hProcess);
     _tprintf(_T(" [%d] %d\n"), JobNo, ProcessInfo.dwProcessId);
     return 0;
    }

    /* jobs: вывод списка всех выполняющихся и остановленных задач. */
    int Jobs(int argc, LPTSTR argv[], LPTSTR Command) {
     if (!DisplayJobs ()) return 1; /*См. описание функций управления задачами*/
     return 0;
    }

    /* kill [параметры] Номер задачи (JobNumber)
     -b: Генерировать Ctrl-Break.
     -c: Генерировать Ctrl-C.
     В противном случае прекратить выполнение процесса. */
    int Kill(int argc, LPTSTR argv[], LPTSTR Command) {
     DWORD ProcessId, JobNumber, iJobNo;
     HANDLE hProcess;
     BOOL CntrlC, CntrlB, Killed;
     iJobNo = Options(argc, argv, _T("bc"), &CntrlB, &CntrlC, NULL);
     /* Найти ID процесса, связанного с данной задачей. */
     JobNumber = _ttoi(argv [iJobNo]);
      ProcessId = FindProcessId(JobNumber); /* См. описание функций управления
задачами. */
     hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, ProcessId);
     if (hProcess == NULL) { /* ID процесса может не использоваться. */
      ReportError(_T("Выполнение процесса уже прекращено.\n"), 0, FALSE);
      return 2;
     }
     if (CntrlB) GenerateConsoleCtrlEvent(CTRL_BREAK_EVENT, ProcessId);
     else if (CntrlC) GenerateConsoleCtrlEvent(CTRL_C_EVENT, ProcessId);
     else TerminateProcess(hProcess, JM_EXIT_CODE);
     WaitForSingleObject(hProcess, 5000);
     CloseHandle(hProcess);
     _tprintf(T("Задача [%d] прекращена или приостановлена \n"), JobNumber);
     return 0;
    }
```

Обратите внимание на то, как команда jobbg создает процесс в приостановленном состоянии, а затем вызывает функцию управления задачами Get JobNumber (программа 6.4) для получения номера задачи, а также регистрации задачи и процесса, который с ней связан. Если в силу каких-либо причин задача не может быть зарегистрирована, выполнение данного процесса немедленно прекращается. Обычно' генерируется корректный номер задачи, после чего выполнение основного потока возобновляется, и он может продолжать выполнение.

Получение номера задачи

Следующие три программы представляют три отдельные функции управления задачами. Все эти функции включены в единый файл JobMgt.c, содержащий все исходные тексты.

Первая из них, программа 6.4, представляет функцию Get JobNumber. Обратите внимание на использование блокирования файлов, а также обработчиков завершения, осуществляющих разблокирование файлов. Эта методика обеспечивает защиту от исключений и непреднамеренного обхода вызова функции разблокирования файлов. Переходы такого рода могут быть случайно вставлены в процессе сопровождения кода, даже если исходная программа корректна. Обратите также внимание на блокирование попыток записи за пределами конца файла в тех случаях, когда файл должен быть расширен за счет добавления новой записи.

Программа 6.4. JobMgt: создание информации о новой задаче

```
    /* Вспомогательная функция управления задачами. */
    #include "EvryThng.h"
    #include "JobMgt.h" /* Листинг приведен в приложении А. */
    void GetJobMgtFileName (LPTSTR);

    LONG GetJobNumber(PROCESS_INFORMATION *pProcessInfo, LPCTSTR Command)
    /* Создать номер задачи для нового процесса и ввести информацию о новом
процессе в базу данных задачи. */
    {
     HANDLE hJobData, hProcess;
     JM_JOB JobRecord;
     DWORD JobNumber = 0, nXfer, ExitCode, FsLow, FsHigh;
     TCHAR JobMgtFileName[MAX_PATH];
     OVERLAPPED RegionStart;
     if (!GetJobMgtFileName(JobMgtFileName)) return -1;
     /* Предоставление результата в виде строки "\tmp\UserName.JobMgt" */
      hJobData = CreateFile(JobMgtFileName, GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE, NULL, OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL,
NULL);
     if (hJobData == INVALID_HANDLE_VALUE) return -1;
      /* Блокировать весь файл плюс одну возможную запись для получения
исключительного доступа. */
     RegionStart.Offset = 0;
     RegionStart.OffsetHigh = 0;
     RegionStart.hEvent = (HANDLE)0;
     FsLow = GetFileSize(hJobData, &FsHigh);
      LockFileEx(hJobData, LOCKFILE_EXCLUSIVE_LOCK, 0, FsLow + SJM_JOB, 0,
&RegionStart);
     __try {
      /* Чтение записи для нахождения пустого сегмента. */
      while(ReadFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL) && (nXfer > 0)) {
       if (JobRecord.ProcessId == 0) break;
       hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, JobRecord.ProcessId);
       if (hProcess == NULL) break;
        if (GetExitCodeProcess(hProcess, &ExitCode) && (ExitCode != STILL_ACTIVE))
break;
       JobNumber++;
      }
       /* Либо найден пустой сегмент, либо мы находимся в конце файла и должны
создать новый сегмент. */
```

```
      if (nXfer != 0) /* Не конец файла. Резервировать. */
       SetFilePointer(hJobData, -(LONG)SJM_JOB, NULL, FILE_CURRENT);
      JobRecord.ProcessId = pProcessInfo->dwProcessId;
      _tcsnccpy(JobRecord.CommandLine, Command, MAX_PATH);
      WriteFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL);
     } /* Конец try-блока. */
     __finally {
      UnlockFileEx(hJobData, 0, FsLow + SJM_JOB, 0, &RegionStart);
      CloseHandle(hJobData);
     }
     return JobNumber + 1;
    }
```

Вывод списка фоновых задач

Программа 6.5 реализует функцию управления задачами DisplayJobs.

Программа 6.5. JobMgt: отображение списка активных задач

```
    BOOL DisplayJobs(void)
    /* Просмотреть файл базы данных, сообщить статус задачи. */
    {
     HANDLE hJobData, hProcess;
     JM_JOB JobRecord;
     DWORD JobNumber = 0, nXfer, ExitCode, FsLow, FsHigh;
     TCHAR JobMgtFileName[MAX_PATH];
     OVERLAPPED RegionStart;
     GetJobMgtFileName(JobMgtFileName);
       hJobData   =   CreateFile(JobMgtFileName,   GENERIC_READ   |   GENERIC_WRITE,
FILE_SHARE_READ  |  FILE_SHARE_WRITE,  NULL,  OPEN_EXISTING,  FILE_ATTRIBUTE_NORMAL,
NULL);
     RegionStart.Offset = 0;
     RegionStart.OffsetHigh = 0;
     RegionStart.hEvent = (HANDLE)0;
     FsLow = GetFileSize(hJobData, &FsHigh);
     LockFileEx(hJobData, LOCKFILE_EXCLUSIVE_LOCK, 0, FsLow, FsHigh, &RegionStart);
     __try {
      while(ReadFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL) && (nXfer > 0)) {
       JobNumber++;
       if (JobRecord.ProcessId == 0) continue;
       hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, JobRecord.ProcessId);
       if (hProcess != NULL) GetExitCodeProcess(hProcess, &ExitCode);
       _tprintf(_T(" [%d] "), JobNumber);
       if (hProcess == NULL) _tprintf(_T(" Готово"));
       else if (ExitCode != STILL_ACTIVE) _tprintf(_T("+ Готово"));
       else _tprintf(_T(" "));
       _tprintf(_T(" %s\n"), JobRecord.CommandLine);
       /* Удалить процессы, которые в системе уже не присутствуют. */
       if (hProcess == NULL) {
        /* Зарезервировать одну запись. */
        SetFilePointer(hJobData, -(LONG)nXfer, NULL, FILE_CURRENT);
        JobRecord.ProcessId = 0;
        WriteFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL);
       }
      } /* Конец цикла while. */
     } /* Конец __try-блока. */
```

```
    __finally {
     UnlockFileEx(hJobData, 0, FsLow, FsHigh, &RegionStart);
     CloseHandle(hJobData);
    }
    return TRUE;
   }
```

Поиск задачи в файле списка задач

Программа 6.6 представляет последнюю функцию управления задачами, FindProcessID, которая получает идентификатор процесса, соответствующего задаче с указанным номером. В свою очередь, идентификатор процесса может использоваться вызывающей программой для получения дескриптора и другой информации о состоянии процесса.

Программа 6.6. JobMgt: получение идентификатора процесса по номеру задачи

```
   DWORD FindProcessId(DWORD JobNumber)
   /* Получить ID процесса для задачи с указанным номером. */
   {
    HANDLE hJobData;
    JM_JOB JobRecord;
    DWORD nXfer;
    TCHAR JobMgtFileName[MAX_PATH];
    OVERLAPPED RegionStart;
    /* Открыть файл управления задачами. */
    GetJobMgtFileName(JobMgtFileName);
      hJobData   =   CreateFile(JobMgtFileName,   GENERIC_READ,   FILE_SHARE_READ   |
FILE_SHARE_WRITE, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hJobData == INVALID_HANDLE_VALUE) return 0;
    /* Перейти к позиции записи, соответствующей указанному номеру задачи.
       * В полной версии программы обеспечивается принадлежность номера задачи
(JobNumber) допустимому диапазону значений. */
    SetFilePointer(hJobData, SJM_JOB * (JobNumber - 1), NULL, FILE_BEGIN);
    /* Блокировка и чтение записи. */
    RegionStart.Offset = SJM_JOB * (JobNumber - 1);
    RegionStart.OffsetHigh =0; /* Предполагаем, что файл "короткий". */
    RegionStart.hEvent = (HANDLE)0;
    LockFileEx(hJobData, 0, 0, SJM_JOB,0, &RegionStart);
    ReadFile(hJobData, &JobRecord, SJM_JOB, &nXfer, NULL);
    UnlockFileEx(hJobData, 0, SJM_JOB, 0, &RegionStart);
    CloseHandle(hJobData);
    return JobRecord.ProcessId;
   }
```

Объекты задач

Процессы можно объединять в объекты задач (job objects), что позволяет управлять процессами как группой, устанавливать лимиты ресурсов для всех процессов, входящих в объект задачи, и вести учетную информацию. Объекты задач были впервые введены в Windows 2000 и теперь поддерживаются во всех системах NT5.

Первым шагом является создание пустого объекта задачи с помощью функции CreateObject, которая принимает два аргумента, имя и атрибуты защиты, и возвращает дескриптор объекта задачи. Существует также функция OpenJobObject, которую можно применять к именованным объектам задач. Для уничтожения объектов используется функция CloseHandle.

Функция AssignProcessToJobObject просто добавляет процесс с указанным дескриптором в объект задачи; она принимает только два параметра. Процесс может принадлежать только одной задаче, поэтому в тех случаях, когда процесс, связанный с указанным дескриптором, уже является элементом какого-либо задания, функция AssignProcessToJobObject завершается с ошибкой. Добавляемый в задачу процесс наследует значения всех ограничений, связанных с задачей, и добавляет в задачу свою учетную информацию, например использованное процессорное время.

По умолчанию новый дочерний процесс, созданный функцией CreateProcess, также принадлежит задаче, если только в аргументе dwCreationFlags при вызове функции CreateProcess не был задан флаг CREATE_BREAKWAY_FROM_JOB. В предусмотренном по умолчанию случае попытки назначения дочернего процесса задаче при помощи функции AssignProcessToJobObject приводят к ее сбойному завершению.

Наконец, для установления управляющих лимитов процессов, входящих в задачу, используется функция SetInformationJobObject.

```
BOOL SetInformationJobObject(HANDLE hJob, JOBOBJECTINFOCLASS
JobObjectInformationClass, LPVOID lpJobObjectInformation, DWORD
cbJobObjectInformationLength)
```

• hJob — дескриптор существующего объекта задачи.

• JobObjectInformationClass — указывает информационный класс устанавливаемых ограничений. Всего существует пять возможных значений; одним из них является значение JobObjectBasicLimitInformation, используемое для указания такой информации, как ограничения общего времени и времени, приходящегося на один процесс, ограничения размеров рабочего набора (working set)[26], ограничения на количество активных процессов, приоритет и родство процессоров (в SMP-системах родственными называются процессоры, которые могут использоваться потоками в процессах задач).

• lpJobObjectInformation — указывает на фактическую информацию, необходимую для предыдущего параметра. Для каждого информационного класса существует своя структура.

• JOBOBJECT_BASIC_ACCOUNTING_INFORMATION — позволяет получить суммарные временные характеристики (пользовательское, системное и истекшее время) процессов, входящих в задачу.

• Значением последнего параметра является размер предыдущей структуры.

Функция QueryJobInformationObject позволяет получить значения текущих ограничений. Другие информационные классы устанавливают ограничения в отношении пользовательского интерфейса, портов завершения ввода/вывода (см. главу 14), атрибутов защиты, а также завершения задачи.

Резюме

Windows предоставляет простой механизм управления процессами и синхронизацией их выполнения. Приведенные примеры продемонстрировали способы управления параллельным выполнением нескольких процессов, а также получения информации о временных характеристиках каждого процесса. Отношения "предок-потомок" между процессами в Windows не поддерживаются, так что в необходимых случаях управление этой информацией возлагается на программиста.

В следующих главах

В следующей главе описываются потоки, являющиеся независимыми единицами выполнения внутри процесса. В некоторых отношениях управление потоками аналогично управлению процессами; все, что связано с кодами завершения, прекращением выполнения и ожиданием завершения, применимо и к потокам. Чтобы продемонстрировать эту аналогию, самый первый из рассматриваемых в главе 7 примеров является переделанным вариантом программы grepMP (программа 6.1), который приспособлен для работы с потоками.

Глава 8 ознакомит вас с методами синхронизации, которые могут быть использованы для координации выполнения потоков, принадлежащих одному и тому же или различным процессам.

Упражнения

6.1. Расширьте возможности программы 6.1 (grepMP) таким образом, чтобы она принимала также параметры командной строки, а не только текстовый шаблон.

6.2. Вместо того чтобы передавать дочернему процессу имя временного файла, как это делается в программе 6.1, преобразуйте наследуемый дескриптор файла к типу DWORD (для типа HANDLE требуется 4 байта), а затем в строку символов. Передайте эту строку дочернему процессу в командной строке. В свою очередь, дочерний процесс должен осуществить обратное преобразование строки символов в значение дескриптора файла, который будет использован для вывода. Эту методику иллюстрируют программы catHA.c и grepHA.c, доступные на Web-сайте книги.

6.3. Программа 6.1 ожидает завершения всех процессов и лишь после этого выводит результаты. При этом возможность определения того, в каком именно порядке завершились процессы внутри программы, отсутствует. Модифицируйте программу таким образом, чтобы она определяла очередность завершения процессов. *Подсказка.* Измените вызов функции WaitForMultipleObjects таким образом, чтобы возврат из нее осуществлялся после завершения каждого отдельного процесса. Другой возможностью является сортировка времени завершения процессов.

6.4. В программе 6.1 временные файлы должны удаляться явным образом. Возможно ли использование флага FILE_FLAG_DELETE_ON_CLOSE при создании временных файлов таким образом, чтобы избавиться от необходимости удаления указанных файлов?

6.5. Определите, заметны ли какие-либо преимущества программы grepMP в отношении производительности (по сравнению с простой последовательной обработкой) в случае SMP-систем, если такая возможность у вас имеется, или при размещении файлов на отдельных или сетевых дисках. Частичные результаты соответствующих тестов приведены в приложении B.

6.6. Можете ли вы предложить способ, возможно, связанный с использованием объектов задач, для определения времени, затраченного на выполнение операций в пользовательском режиме и в режиме ядра? Использование объектов задач может потребовать внесения изменений в программу grepMP.

6.7. Улучшите функцию grepMP (программа 6.5) таким образом, чтобы она сообщала код завершения для каждой завершенной задачи. Кроме того, организуйте вывод временных характеристик (истекшего времени, времени работы в режиме ядра и времени работы в пользовательском режиме) суммарно для всех процессов.

6.8. У функций управления задачами есть один трудно устранимый недостаток. Предположим, что задача уничтожена и что главная программа повторно использует идентификатор процесса данного задания еще до того, как этот идентификатор будет удален из файла управления задачами. Вместе с тем, ранее этот идентификатор мог быть использован функцией OpenProcess для создания дескриптора какого-либо процесса, хотя теперь этот же идентификатор ссылается на совершенно другой процесс. Чтобы устранить возможность появления проблем подобного рода, требуется создать вспомогательный процесс, в котором будут храниться копии дескрипторов каждого созданного процесса, что позволит избежать повторного использования идентификаторов. Другая возможная методика заключается в сохранении времени запуска процесса в файле управления задачами. Это время должно совпадать со временем запуска процесса, полученного с использованием идентификатора. *Примечание.* Идентификаторы процессов быстро исчерпываются, и поэтому вероятность их повторного использования очень велика. В UNIX для получения идентификаторов новых

процессов применяются последовательно увеличиваемые значения 32-битового счетчика, так что идентификаторы могут повторяться только после исчерпания этих значений, что происходит очень редко. В отличие от этого, в программах Windows никогда нельзя полагаться на то, что идентификатор процесса не будет повторно использован.

6.9. Измените программу JobShell таким образом, чтобы информация сохранялась в реестре, а не во временном файле.

6.10. Измените программу JobShell таким образом, чтобы процессы связывались с объектом задачи. Наложите временные и другого рода ограничения на объекты задач, предоставив пользователю возможность ввода числовых значений некоторых из этих ограничений.

6.11. Улучшите программу JobShell таким образом, чтобы команда jobs обеспечивала подсчет числа дескрипторов, используемых каждой из задач. *Подсказка.* Воспользуйтесь функцией GetProcessHandleCount, для которой требуется NT 5.1.

6.12. Создайте проект Version (находится на Web-сайте), использующий программу verison.c. Попытайтесь произвести пробные запуски этой программы под управлением как можно большего числа различных версий Windows, к которым у вас имеется доступ, включая Windows 9x и NT 4.0, если это возможно. Каковы старшие и младшие номера версий для этих систем, полученные вами, и какую дополнительную информацию о версиях вам удалось получить?

ГЛАВА 7
Потоки и планирование выполнения

Основной единицей выполнения в Windows является поток, и одновременно несколько потоков могут выполняться в рамках одного процесса, разделяя его адресное пространство и другие ресурсы. В главе 6 процессы ограничивались только одним потоком, однако существует много ситуаций, в которых возможность использования более одного потока была бы весьма желательной. Настоящая глава посвящена описанию потоков и иллюстрации областей их применения. Глава 8 продолжает эту тему описанием объектов синхронизации и анализом как положительных, так и отрицательных аспектов использования потоков, в то время как в главе 9 исследуется проблема повышения производительности и компромиссные способы ее решения. В главе 10 описываются методы и модели программирования объектов синхронизации, позволяющие значительно упростить проектирование и разработку надежных многопоточных программ. Эти методы применяются далее во всей оставшейся части книги.

Завершается настоящая глава кратким обсуждением облегченных потоков, посредством которых можно создавать отдельные задачи в контексте потоков. Ввиду того, что облегченные потоки используются довольно редко, можно предположить, что многие читатели предпочтут пропустить этот материал при первом чтении.

Обзор потоков

Поток (thread) — это независимая единица выполнения в контексте процесса. Программист, разрабатывающий многопоточную программу, должен организовать выполнение потоков таким образом, чтобы это позволило упростить программу и воспользоваться предоставляемыми самим хост-компьютером возможностями распараллеливания задач.

При традиционном подходе программы выполняются в виде единственного потока. Несмотря на возможность организации параллельного выполнения нескольких процессов, что было продемонстрировано на ряде примеров в главе 6, и даже их взаимодействия между собой посредством таких механизмов, как разделение памяти или каналы (глава 11), однопоточные процессы имеют ряд недостатков.

• Переключение между выполняющимися процессами потребляет заметную долю временных и других ресурсов ОС, а в случаях, аналогичных многопроцессному поиску (grepMP, программа 6.1), все процессы заняты выполнением одной и той же программы. Организация параллельной обработки файла с помощью потоков в контексте единственного процесса позволяет снизить общие накладные расходы системы.

• Не считая случаев разделения памяти, между процессами существует лишь слабая взаимосвязь, а организация разделения ресурсов, например, открытых файлов, вызывает затруднения.

• С использованием только однопоточных процессов трудно организовать простое и эффективное управление несколькими параллельно выполняющимися задачами, взаимодействующими между собой, в таких, например, случаях, как ожидание и обработка пользовательского ввода, ожидание ввода из файла или сети и выполнение вычислений.

• Тесно связанные с выполнением операций ввода/вывода программы, подобные рассмотренной в главе 2 программе преобразования файлов из ASCII в Unicode (atou, программа 2.4), вынуждены ограничиваться простой моделью "чтение-изменение-запись". При обработке последовательностей файлов гораздо эффективнее инициализировать выполнение как можно большего числа операций чтения. Windows NT предлагает дополнительные возможности перекрывающегося асинхронного ввода/вывода (глава 14), однако потоки позволяют добиться того же эффекта с меньшими усилиями.

• В SMP-системах планировщик Windows распределяет выполнение отдельных потоков между различными процессорами, что во многих случаях приводит к повышению производительности.

В этой главе обсуждаются потоки и способы управления ими. Использование потоков рассматривается на примере задач параллельного поиска и многопоточной сортировки содержимого файлов. Эти две задачи позволяют сопоставить применение потоков в операциях ввода/вывода и в операциях, связанных с выполнением интенсивных вычислений. Кроме того, в этой главе представлен общий обзор планирования выполнения процессов и потоков в Windows.

Перспективы и проблемы

Согласно принятой в этой и последующих главах точке зрения использование потоков не только позволяет упростить проектирование и реализацию некоторых программ, но и (при условии соблюдения нескольких элементарных правил и следования определенным моделям программирования) обеспечивает повышение производительности и надежности программ, а также делает более понятной их структуру и облегчает их обслуживание. Функции управления

потоками весьма напоминают функции управления процессами, так что, например, наряду с функцией GetProcessExitCode существует также функция GetThreadExitCode.

Указанная точка зрения не является общепринятой. Многие авторы и разработчики программного обеспечения обращают внимание на всевозможные риски и проблемы, которые возникают в случае использования потоков, и отдают предпочтение использованию нескольких процессов, когда требуется параллелизм операций. К числу проблем упомянутого рода относятся следующие:

• Поскольку потоки разделяют общую память и другие ресурсы, принадлежащие одному процессу, существует вероятность того, что один поток может случайно изменить данные, относящиеся к другому потоку.

• При определенных обстоятельствах вместо улучшения производительности может наблюдаться ее резкое ухудшение.

• Разделение потоками общей памяти и других ресурсов в контексте одного процесса может стать причиной нарушения условий состязаний между процессами и вызывать блокирование некоторых из них.

Некоторых проблем, с которыми действительно приходится сталкиваться, можно избежать, тщательно проектируя и программируя соответствующие задачи, тогда как природа других проблем обусловлена самим параллелизмом, независимо от того, реализуется он путем разбиения процессов на потоки, использованием нескольких процессов или применением специальных методов, например, методов асинхронного ввода/вывода, предоставляемых Windows.

Основные сведения о потоках

В предыдущей главе на рис. 6.1 было показано, каким образом обеспечивается существование потоков в среде процесса. Использование потоков на примере многопоточного сервера, способного обрабатывать запросы одновременно нескольких клиентов, иллюстрирует рис. 7.1; каждому клиенту отведен поток. Эта модель будет реализована в главе 11.

Потоки, принадлежащие одному процессу, разделяют общие данные и код, поэтому очень важно, чтобы каждый поток имел также собственную область памяти, относящуюся только к нему. В Windows удовлетворение этого требования обеспечивается несколькими способами.

• У каждого потока имеется собственный стек, который она использует при вызове функций и обработке некоторых данных.

• При создании потока вызывающий процесс может передать ему аргумент (Arg на рис. 7.1), который обычно является указателем. На практике этот аргумент помещается в стек потока.

• Каждый поток может распределять индексы собственных локальных областей хранения (Thread Local Storage, TLS), а также считывать и устанавливать значения TLS. TLS, описанные далее, предоставляют в распоряжение потоков небольшие массивы данных, и каждый из потоков может обращаться к собственной TLS. Одним из преимуществ TLS является то, что они обеспечивают защиту данных, принадлежащих одному потоку, от воздействия со стороны других потоков.

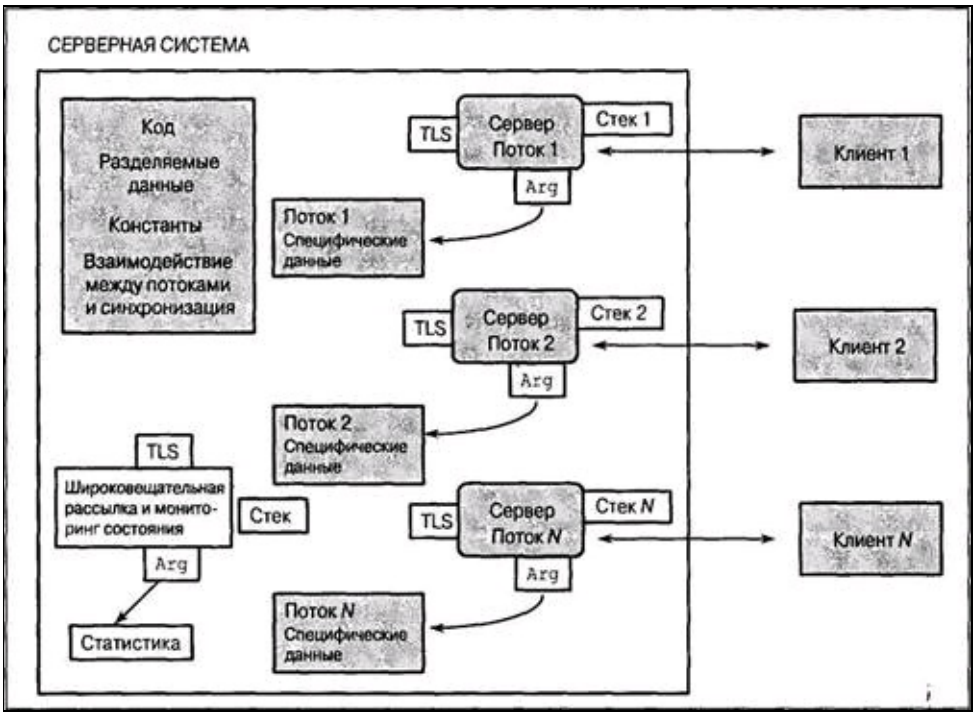


Рис. 7.1. Потоки в среде сервера

Аргумент потока и TLS могут использоваться для указания произвольной структуры данных. Применительно к представленному на рис. 7.1 примеру сервера эта структура может содержать текущий запрос и отклик потока на этот запрос, а также предоставлять рабочую память для других целей.

В случае SMP-систем Windows обеспечивает параллельное выполнение различных потоков, в том числе и принадлежащих одному и тому же процессу, на разных процессорах. Правильное использование этой возможности позволяет повысить производительность, однако, как будет показано в двух следующих главах, в результате непродуманных действий без заранее определенной стратегии использования нескольких процессоров производительность SMP-

систем может даже ухудшиться по сравнению с однопроцессорными системами.

Управление потоками

Вероятно, вы не будете удивлены, узнав о том, что у потоков, как и у любого другого объекта Windows, имеются дескрипторы и что для создания потоков, выполняющихся в адресном пространстве вызывающего процесса, предусмотрен системный вызов CreateThread. Как и в случае процессов, мы будем говорить иногда о "родительских" и "дочерних" потоках, хотя ОС не делает в этом отношении никаких различий. Системный вызов CreateThread предъявляет ряд специфических требований:

• Укажите начальный адрес потока в коде процесса.

• Укажите размер стека, и необходимое пространство стека будет выделено из виртуального адресного пространства процесса. Размер стека по умолчанию равен размеру стека основного потока (обычно 1 Мбайт). Первоначально для стека отводится одна страница (см. главу 5). Новые страницы стека выделяются по мере надобности до тех пор, пока стек не достигнет своего максимального размера, поэтому не сможет больше расти.

• Задайте указатель на аргумент, передаваемый потоку. Этот аргумент может быть чем угодно и должен интерпретироваться самим потоком.

• Функция возвращает значение идентификатора (ID) и дескриптор потока.

В случае ошибки возвращаемое значение равно NULL.

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpsa, DWORD dwStackSize,
LPTHREAD_START_ROUTINE lpStartAddr, LPVOID lpThreadParm, DWORD
dwCreationFlags, LPDWORD lpThreadId)
```

Параметры

lpsa — указатель на уже хорошо знакомую структуру атрибутов защиты.

dwStackSize — размер стека нового потока в байтах. Значению 0 этого параметра соответствует размер стека по умолчанию, равный размеру стека основного потока.

lpStartAddr — указатель на функцию (принадлежащую контексту процесса), которая должна выполняться. Эта функция принимает единственный аргумент в виде указателя и возвращает 32-битовый код завершения. Этот аргумент может интерпретироваться потоком либо как переменная типа DWORD, либо как указатель. Функция потока (ThreadFunc) имеет следующую сигнатуру:

```
DWORD WINAPI ThreadFunc(LPVOID)
```

lpThreadParm — указатель, передаваемый потоку в качестве аргумента, который обычно интерпретируется потоком как указатель на структуру аргумента.

dwCreationFlags — если значение этого параметра установлено равным 0, то поток запускается сразу же после вызова функции CreateThread. Установка значения CREATE_SUSPENDED приведет к запуску потока в приостановленном состоянии, из которого поток может быть переведен в состояние готовности путем вызова функции ResumeThread.

lpThreadId — указатель на переменную типа DWORD, которая получает идентификатор нового потока; в Windows 9x и Windows NT 3.51 значение NULL для этого параметра устанавливать нельзя.

Любой поток процесса может сама завершить свое выполнение, вызвав функцию

ExitThread, однако более обычным способом самостоятельного завершения потока является возврата из функции потока с использованием кода завершения в качестве возвращаемого значения. По завершении выполнения потока память, занимаемая ее стеком, освобождается. В случае если поток был создан в библиотеке DLL, будет вызвана соответствующая точка входа DllMain (глава 4) с указанием флага DLL_THREAD_DETACH в качестве "причины" этого вызова.

```
VOID ExitThread(DWORD dwExitCode)
```

Когда завершается выполнение последнего потока, завершается и выполнение самого процесса.

Выполнение потока также может быть завершено другим потоком с помощью функции TerminateThread, однако освобождения ресурсов потока при этом не происходит, обработчики завершения не выполняются и уведомления библиотекам DLL не посылаются. Лучше всего, когда поток сам завершает свое выполнение; применять для этого функцию TerminateThread крайне нежелательно. Функции TerminateThread присущи те же недостатки, что и функции TerminateProcess.

Поток, выполнение которого было завершено (напомним, что обычно поток должен самостоятельно завершать свое выполнение), продолжает существовать до тех пор, пока посредством функции CloseHandle не будет закрыт ее последний дескриптор. Любой другой поток, возможно и такой, который ожидает завершения другого потока, может получить код завершения потока.

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpExitCode)
```

lpExitCode — будет содержать код завершения потока, указывающий на его состояние. Если поток еще не завершен, значение этой переменной будет равно STILL_ACTIVE.

Идентификация потоков

Функции, используемые для получения идентификаторов (ID) и дескрипторов потоков, напоминают те, которые используются для аналогичных целей в случае процессов.

• GetCurrentThread — возвращает ненаследуемый псевдодескриптор вызывающего потока.

• GetCurrentThreadId — позволяет получить идентификатор потока, а не его дескриптор.

• GetThreadId — позволяет получить идентификатор потока, если известен его дескриптор; эта функция требует использования Windows Server 2003.

• OpenThread — создает дескриптор потока по известному идентификатору.

В программе JobShell (программа 6.3) нам очень пригодилась функция OpenProcess, и функция OpenThread может применяться для аналогичных целей.

Дополнительные функции управления потоками

Несмотря на то что функций управления потоками, которые мы выше обсуждали, вполне достаточно для большинства случаев, в том числе и для примеров, приведенных в этой книге, в Windows XP и Windows Server 2003 были введены две дополнительные функции. Их краткие описания представлены ниже.

1. Функция GetProcessIdOfThread, требующая использования Windows Server 2003,

позволяет получать идентификатор процесса, которому принадлежит поток, по известному дескриптору потока. Вы могли бы задействовать эту функцию в программах, предназначенных для управления потоками, принадлежащими другим процессам, или взаимодействия с такими потоками. Если необходимо получить дескриптор процесса, применяйте для этого функцию OpenProcess.

2. Функция GetThreadIOPendingFlag позволяет определить, имеются ли у потока, на который указывает дескриптор, необслуженные запросы ввода/вывода. Например, поток мог быть заблокирован во время выполнения операции ReadFile. В качестве результата возвращается состояние потока во время выполнения данной функции; фактическое состояние может в любой момент измениться, если целевой поток завершает или начинает выполнение операции. Эта функция требует использования NT 5.1 и поэтому доступна лишь в Windows XP или Windows Server 2003.

Приостановка и возобновление выполнения потока

Для каждого потока поддерживается *счетчик приостановок* (suspend count), и выполнение потока может быть продолжено лишь в том случае, если значение этого счетчика равно 0. Поток может увеличивать или уменьшать значение счетчика приостановок другого потока с помощью функций SuspendThread и Resume-Thread. Вспомните, что поток можно создать в приостановленном состоянии со счетчиком приостановок равным 1.

```
DWORD ResumeThread(HANDLE hThread)
DWORD SuspendThread(HANDLE hThread)
```

В случае успешного выполнения обе функции возвращают предыдущее значение счетчика приостановок, иначе — 0xFFFFFFFF.

Ожидание завершения потока

Поток может дожидаться завершения выполнения другого потока точно так же, как потоки могут дожидаться завершения процесса, что обсуждалось в главе 6. В этом случае при вызове функций ожидания (WaitForSingleObject и WaitForMultipleObjects) вместо дескрипторов процессов следует использовать дескрипторы потоков. Заметьте, что не все дескрипторы в массиве, передаваемом функции WaitForMultipleObjects, должны быть обязательно одного и того же типа; например, в одном вызове могут быть одновременно указаны дескрипторы потоков, процессов и других объектов.

Допустимое количество объектов, одновременно ожидаемых функцией WaitForMultipleObjects, ограничено значением MAXIMUM_WAIT_OBJECTS (64), но при большом количестве потоков можно воспользоваться серией вызовов функций ожидания. Эта техника уже была продемонстрирована в программе 6.1; программы, приведенные в книге, ожидают завершения выполнения одиночных объектов, но на Web-сайте приведены полные решения.

Функция ожидания дожидается, пока объект, указанный дескриптором, не перейдет в *сигнальное* состояние. В случае потоков объект потока переводится в сигнальное состояние при помощи функций ExitThread и TerminateThread, что приводит к освобождению всех других потоков, дожидающихся перехода данного объекта в сигнальное состояние, включая и те потоки, которые могли оставаться в состоянии ожидания и впоследствии, после того, как поток

завершится. Дескриптор потока, перешедший в сигнальное состояние, не выходит из этого состояния. То же самое остается справедливым и по отношению к дескрипторам процессов, но не относится к дескрипторам некоторых других объектов, например, мьютексов и событий (описываются в следующей главе).

Заметьте, что дожидаться перехода в сигнальное состояние одного и того же объекта могут одновременно несколько потоков. Аналогично, функция ExitProcess переводит в сигнальное состояние как сам процесс, так и все его потоки.

Удаленные потоки

Функция CreateRemoteThread позволяет создавать потоки, выполняющиеся в другом процессе. По сравнению с функцией CreateThread в ней имеется один дополнительный параметр для указания дескриптора процесса, а адрес функции, задающий начальный адрес нового потока, должен находиться в адресном пространстве целевого процесса. Использование функции CreateRemoteThread относится к числу интересных, однако рискованных способов непосредственного воздействия одним процессом на другой, и может пригодиться, например, при написании отладчиков.

У функции CreateRemoteThread есть одно очень интересное применение. Вместо того чтобы вызывать функцию TerminateProcess, управляющий процесс может создать поток, выполняющийся в другом процессе, который и организует корректное завершение этого процесса. Однако в главе 10 демонстрируется более безопасный метод, позволяющий одному потоку завершить другой с использованием асинхронного вызова процедур.

> Понятие о потоках твердо упрочилось во многих ОС, и исторически так сложилось, что многие поставщики и пользователи UNIX предоставляли собственные частные варианты их реализации. Были разработаны некоторые библиотеки, обеспечивающие многопоточную поддержку вне ядра. В настоящее время стандартом в этой области являются потоки POSIX Pthreads. Потоки Pthreads включены в частные варианты реализации UNIX и Linux и иногда считаются частью UNIX. Соответствующие системные вызовы отличаются от обычных системных вызовов UNIX наличием в именах префикса pthread. Потоки Pthreads поддерживаются также некоторыми другими системами, отличными от UNIX, такими, например, как Open VMS.
>
> Системный вызов pthread_create эквивалентен вызову CreateThread, а системный вызов pthread_exit — вызову ExitThread. Для организации ожидания одним потоком завершения другого применяется системный вызов pthread_join. Потоки Pthreads предоставляют очень полезную функцию pthread_cancel, гарантирующую, в отличие от функции TerminateThread, выполнение обработчиков завершения и уничтожение ненужных дескрипторов. Возможность уничтожения потоков была бы в Windows крайне желательной, но в главе 10 представлен метод, обеспечивающий получение такого же эффекта.

Использование библиотеки C впотоках

В большинстве программ требуется библиотека C, хотя бы для того, чтобы обеспечить выполнение операций над строками. Исторически так сложилось, что библиотека C была рассчитана на применение в однопоточных процессах, поэтому для хранения промежуточных результатов многие функции используют области глобальной памяти. Подобные библиотеки, в которых отсутствует многопоточная поддержка, не являются безопасными (thread-safe) с точки зрения одновременного выполнения нескольких потоков, поскольку, например, одновременно две независимые потоки могут пытаться получить доступ к библиотеке и изменить данные, содержащиеся в ее глобальной памяти. Принципы проектирования многопоточных программ будут вновь обсуждаться в главе 8, в которой описывается синхронизация объектов Windows.

Пример функции strtok показывает, почему при написании некоторых функций библиотеки C не учитывалась многопоточная поддержка. Функция strtok, просматривающая строку в поиске очередного вхождения определенной лексемы, поддерживает сохранение состояния (persistent state) между последовательными вызовами функции, и это состояние хранится в области статической памяти, совместный доступ к которой имеют все потоки, вызывающие эту функцию.

Microsoft C решает эту проблему, предлагая реализацию библиотеки C под названием LIBCMT.LIB, которая обеспечивает многопоточную поддержку. Однако, это еще не все. Вы не должны использовать функцию CreateThread; для запуска потока и создания специфической для него области рабочей памяти библиотеки LIBCMT.LIB необходимо пользоваться специальной функцией C, а именно, функцией _beginthreadex. Для завершения потока вместо функции ExitThread применяется функция _endthreadex.

> **Примечание**
>
> В качестве упрощенного варианта функции _beginthreadex предусмотрена функция _beginthread, однако *использовать ее не рекомендуется.* Прежде всего, функция _beginthread не имеет ни атрибутов, ни флагов защиты и не возвращает идентификатор потока. Более того, в действительности она закрывает дескриптор потока, который создает, в результате чего возвращенное значение дескриптора может оказаться недействительным на момент его сохранения родительским потоком. Не следует вызывать и функцию _endthread; она не позволяет пользоваться возвращаемым значением.

Аргументы функции _beginthreadex в точности совпадают с аргументами функций Windows, однако типы данных Windows для этой функции не определены, и поэтому тип возвращаемого значения функции _beginthread необходимо привести к типу HANDLE, что позволит избежать появления предупреждающих сообщений. Убедитесь в том, что определение символической константы _MT предшествует любому из включаемых файлов; в примерах программ это определение содержится в файле Envirmnt.h. Больше от вас ничего не требуется. Резюмируя, перечислим действия, которые вы должны выполнить, если имеете дело со средой разработки Visual C++.

• Подключите библиотеку LIBCMT.LIB и откажитесь от использования библиотеки, заданной по умолчанию.

• Включите директиву #define _MT во все исходные файлы, в которых используется библиотека C.

• Добавьте включаемый файл <process.h>, содержащий определения функций _beginthreadex и _endthreadex.

• Создайте потоки с помощью функции _beginthreadex; не применяйте для этой цели функцию CreateThread.

• Завершите потоки посредством функции _endthreadex или просто воспользуйтесь оператором return в конце функции потока.

В приложении А вы найдете указания относительно того, как создавать многопоточные приложения. В частности, можно, и даже рекомендуется, указывать библиотеку и определять константу _MT непосредственно в среде разработки.

Именно так будут построены все наши примеры, и функция CreateThread никогда не будет непосредственно применяться в программах даже в тех случаях, когда библиотека C в функциях потоков не используется.

Библиотеки с многопоточной поддержкой

При проектировании пользовательских библиотек следует уделять самое пристальное внимание тому, чтобы избежать возникновения проблем, связанных с параллельным выполнением нескольких потоков, особенно в тех случаях, когда речь идет о сохранении информации о состоянии процессов. Одна из возможных стратегий демонстрируется в примере в главе 12 (программа 12.4), где библиотека DLL для сохранения информации о состоянии использует отдельный параметр.

Еще один пример в главе 12 (программа 12.5) иллюстрирует альтернативный подход, в котором применяется функция DllMain и TLS, описанные далее в настоящей главе.

Пример: многопоточный поиск контекста

В программе 6.1 (grepMP) для выполнения одновременного поиска текстового шаблона в нескольких файлах использовались процессы. Программа 7.1 (grepMT), которая включает исходный код функции поиска текстового шаблона grep, обеспечивает выполнение поиска несколькими потоками в рамках одного процесса. Код функции поиска основан на вызовах функций файлового ввода/вывода библиотеки C. Основная программа аналогична той, которая предлагалась в варианте реализации, основанном на использовании процессов.

Этот пример также показывает, что применение потоков позволяет выполнять асинхронные операции ввода/вывода даже без привлечения специально для этого предназначенных методов, описанных в главе 14. В данном примере параллельным вводом/выводом с участием нескольких файлов управляет программа, в то время как основной или любого другого потока предоставляется возможность в ожидании завершения ввода/вывода выполнять дополнительную обработку. По мнению автора, способ реализации асинхронного ввода/вывода, обеспечиваемый потоками, является более простым, а сравнительный анализ эффективности различных методов, представленный в главе 14, поможет вам выработать собственное мнение на этот счет.

Мы увидим, однако, что в сочетании с портами завершения ввода/вывода операции асинхронного ввода/вывода становятся очень полезным, а часто и необходимым средством в тех случаях, когда количество потоков очень велико.

В иллюстративных целях в программу grepMT введено дополнительное отличие по сравнению с программой grepMP. В данном случае функция WaiForMultipleObjects ожидает завершения не всех потоков, а только одного. Соответствующая информация выводится без ожидания завершения других потоков. В большинстве случае порядок завершения потоков будет меняться от одного запуска программы к другому. Программу легко видоизменить таким образом, чтобы результаты отображались в порядке указания аргументов в командной строке; для этого будет достаточно сымитировать программу grepMP.

Наконец, обратите внимание на ограничение в 64 потока, обусловленное значением константы MAXIMUM_WAIT_OBJECTS, которая ограничивает количество дескрипторов при вызове функции WaitForMultipleObjects. Если у вас возникнет необходимость в большем количестве потоков, организуйте для функций WaitForSingleObjects или WaitForMultipleObjects соответствующий цикл.

Предостережение

Программа grepMP осуществляет асинхронный ввод/вывод в том смысле, что отдельные потоки выполняют параллельное синхронное чтение различных файлов, которые блокируются до момента завершения операции чтения. Можно также организовать параллельное чтение одного и того же файла, если у него имеются различные дескрипторы (обычно, по одному дескриптору для каждого потока). Эти дескрипторы должны быть сгенерированы функцией CreateFile, а не функцией DuplicateHandle. В главе 14 описывается асинхронный ввод/вывод, осуществляемый как с использованием, так и без использования пользовательских потоков, а в примере, доступном на Web-сайте (программа atouMT, описанная в главе 14), операции ввода/вывода выполняются с использованием нескольких потоков по отношению к одному и тому же файлу.

Программа 7.1. grepMT: многопоточный поиск текстового шаблона

```
    /* Глава 7. grepMT. */
    /* Параллельный поиск текстового шаблона — версия, использующая несколько
потоков. */
    #include "EvryThng.h"

    typedef struct { /* Структура данных потока поиска. */
     int argc;
     TCHAR targv[4][MAX_PATH];
    } GREP_THREAD_ARG;
    typedef GREP_THREAD_ARG *PGR_ARGS;
    static DWORD WINAPI ThGrep(PGR_ARGS pArgs);

    int _tmain(int argc, LPTSTR argv[]) {
     GREP_THREAD_ARG * gArg;
     HANDLE * tHandle;
     DWORD ThdIdxP, ThId, ExitCode;
     TCHAR CmdLine[MAX_COMMAND_LINE];
     int iThrd, ThdCnt;
     STARTUPINFO Startup;
     PROCESS_INFORMATION ProcessInfo;
     GetStartupInfo(&StartUp);
      /* Основной поток: создает отдельные потоки поиска на основе функции "grep"
для каждого файла. */
     tHandle = malloc((argc - 2) * sizeof(HANDLE));
     gArg = malloc((argc - 2) * sizeof(GREP_THREAD_ARG));
     for (iThrd = 0; iThrd < argc - 2; iThrd++) {
      _tcscpy(gArg[iThrd].targv[1], argv[1]); /* Pattern. */
      _tcscpy(gArg[iThrd].targv[2], argv[iThrd + 2]);
      GetTempFileName /* Имя временного файла. */
       (".", "Gre", 0, gArg[iThrd].targv[3]);
      gArg[iThrd].argc = 4;
      /* Создать рабочий поток для выполнения командной строки. */
       tHandle[iThrd] = (HANDLE)_beginthreadex(NULL, 0, ThGrep, &gArg[iThrd], 0,
&ThId);
     }
     /* Перенаправить стандартный вывод для вывода списка файлов. */
     Startup.dwFlags = STARTF_USESTDHANDLES;
     Startup.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
     /* Выполняются все рабочие потоки. Ожидать их завершения. */
     ThdCnt = argc - 2;
     while (ThdCnt > 0) {
      ThdIdxP = WaitForMultipleObjects(ThdCnt, tHandle, FALSE, INFINITE);
      iThrd = (int)ThdIdxP - (int)WAIT_OBJECT_0;
      GetExitCodeThread(tHandle [iThrd], &ExitCode);
      CloseHandle(tHandle [iThrd]);
      if (ExitCode ==0) { /* Шаблон найден. */
       if (argc > 3) {
        /* Вывести имя файла, если имеется несколько файлов. */
        _tprintf(_T("\n**Результаты поиска - файл: %s\n"), gArg[iThrd].targv [2]);
        fflush(stdout);
       }
       /* Использовать программу "cat" для перечисления результирующих файлов. */
       _stprintf(CmdLine, _T("%s%s"), _T("cat "), gArg [iThrd].targv[3]);
         CreateProcess(NULL, CmdLine, NULL, NULL, TRUE, 0, NULL, NULL, &StartUp,
&ProcessInfo);
       WaitForSingleObject(ProcessInfo.hProcess, INFINITE);
       CloseHandle(ProcessInfo.hProcess);
       CloseHandle(ProcessInfo.hThread);
      }
```

```
  DeleteFile(gArg[iThrd].targv[3]);
  /* Скорректировать массивы потоков и имен файлов. */
  tHandle[iThrd] = tHandle[ThdCnt - 1];
  _tcscpy(gArg[iThrd].targv[3], gArg[ThdCnt - 1].targv[3]);
  _tcscpy(gArg[iThrd].targv[2], gArg[ThdCnt - 1].targv[2]);
  ThdCnt--;
 }
}

/* Прототип функции контекстного поиска:
static DWORD WINAPI ThGrep(PGR_ARGS pArgs){ } */
```

Потоки и производительность

Программы grepMP и grepMT по своей структуре и сложности сопоставимы друг с другом, однако, как и следовало ожидать, программа grepMT характеризуется более высокой производительностью, так как переключение между потоками осуществляется ядром намного эффективнее, чем переключение между процессами. В приложении B показано, что эти теоретические ожидания отвечают действительности, и это особенно заметно в тех случаях, когда файлы размещены на различных дисках. Оба варианта реализации способны работать в SMP-системах, существенно улучшая показатели производительности в терминах общего времени выполнения (истекшего времени); потоки, независимо от того, принадлежат ли они одному и тому же или разным процессам, параллельно выполняются на различных процессорах. Измеренное пользовательское время в действительности превышает общее время выполнения, поскольку рассчитывается в виде суммарной величины для всех процессоров.

В то же время, существует весьма распространенное заблуждение, суть которого состоит в том, что отмеченный параллелизм, независимо от того, касается ли он использования нескольких процессов, как в случае grepMP, или же применения нескольких потоков, как в случае grepMT, способен приводить к повышению производительности лишь в случае SMP-систем. Выигрыш в производительности можно получить и при использовании нескольких дисков, а также при любом другом распараллеливании в системе хранения. Во всех подобных случаях операции ввода/вывода с участием нескольких файлов будут осуществляться в параллельном режиме.

Модель "хозяин/рабочий" и другие модели многопоточных приложений

Программа grepMT демонстрирует модель многопоточных приложений, носящую название модели "хозяин/рабочий" ("boss/worker"), а рис. 6.3, после замены в нем термина "процесс" на термин "поток", может служить графической иллюстрацией соответствующих отношений. Главный поток (основной поток в данном случае) поручает выполнение отдельных задач рабочим потокам. Каждый рабочий, поток получает файл, в котором она должна выполнить поиск, а полученные рабочим потоком результаты передаются главному потоку во временном файле.

Существуют многочисленные вариации этой модели, одной из которых является модель рабочей группы (work crew model), в которой рабочие потоки объединяют свои усилия для решения одной задачи, причем каждый отдельный поток выполняет свою небольшую часть работы. Модель рабочей группы используется в нашем следующем примере (рис. 7.2). Рабочие группы даже могут самостоятельно распределять работу между собой без получения каких-либо указаний со стороны главного потока. В многопоточных программах может быть применена практически любая из схем управления, разработанных для коллективов в человеческом обществе.

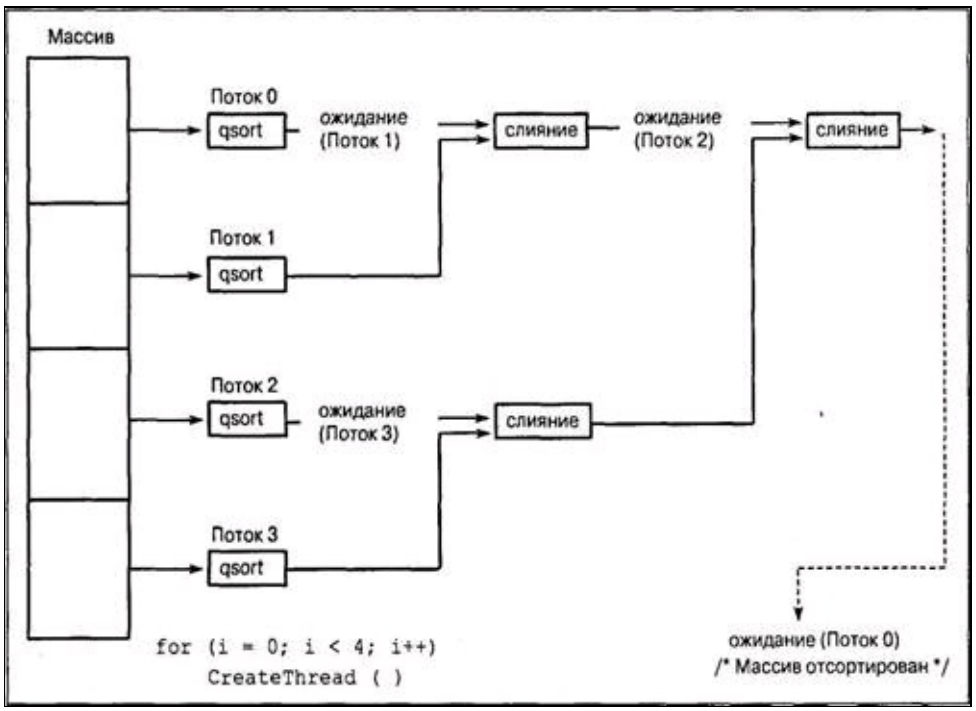


Рис. 7.2. Выполнение сортировки слиянием с использованием нескольких потоков

Двумя другими основными моделями являются модель *"клиент/сервер"* (client/server) (проиллюстрирована на рис. 7.1, а пример ее практической реализации рассматривается в главе 11) и *конвейерная модель* (pipeline model), в которой выполнение задания передается от одного потока к другому (пример многоступенчатого конвейера рассматривается в главе 10 и иллюстрируется на рис. 10.1).

При проектировании многопоточных систем эти модели обладают целым рядом преимуществ, к числу которых можно отнести следующие:

• Большинство проблем многопоточного программирования могут быть разрешены с использованием одной из стандартных моделей, облегчающих проектирование, разработку и отладку программ.

• Применение понятных и проверенных моделей не только позволяет избежать многих ошибок, которые легко допустить при написании многопоточных программ, но и способствует повышению производительности результирующих приложений.

• Эти модели естественным образом соответствуют структуре большинства обычных задач программирования.

• Программистам, сопровождающим программу, будет гораздо легче понять ее устройство, если она будет описана в документации на понятном языке.

• Находить неполадки в незнакомой программе гораздо легче, если ее можно анализировать в терминах моделей. Очень часто главную причину неполадок удается установить на основании видимых нарушений базовых принципов одной из моделей.

• Многие распространенные дефекты программ, например, нарушение условий состязаний задач и их блокирование, также можно описать с использованием простых моделей, к числу которых относятся эффективные методы использования объектов синхронизации, описанные в главах 9 и 10.

Эти классические модели потоков реализованы во многих ОС. В модели компонентных объектов (Component Object Model, COM), широко используемой во многих Windows-системах, применяется другая терминология, и хотя рассмотрение модели COM выходит за рамки данной книги, об этих моделях говорится в конце главы 11, где они сравниваются с другими примерами программ.

Пример: применение принципа "разделяй и властвуй" для решения задачи сортировки слиянием в SMP-системах

Этот пример демонстрирует возможности значительного повышения производительности за счет использования потоков, особенно в случае SMP-систем. Основная идея заключается в разбиении задачи на более мелкие составляющие, распределении выполнения подзадач между отдельными потоками и последующем объединении результатов для получения окончательного решения. Планировщик Windows автоматически назначит потокам отдельные процессоры, в результате чего задачи будут выполняться параллельно, снижая общее время выполнения приложения.

Эта стратегия, которую часто называют стратегией "разделяй и властвуй" (divide and conquer), или *моделью рабочей группы* (work crew model), оказалась весьма полезной и в качестве средства повышения производительности, и в качестве метода проектирования алгоритмов. Одним из примеров ее применения служит программа grepMT (программа 7.1), в которой для каждой файловой операции ввода/вывода и для поиска шаблона создается отдельный поток. Как показано в приложении B, в случае SMP-систем производительность повышается, поскольку планировщик может распределять выполнение потоков между различными процессорами.

Далее мы рассмотрим другой пример, в котором задача сортировки содержимого файла разбивается на ряд подзадач, выполнение которых делегируется отдельным потокам.

Решение задачи сортировки слиянием (merge-sort), в которой сортируемый массив разбивается на несколько массивов меньшего размера, является классическим примером алгоритма, построенного на принципе "разделяй и властвуй". Каждый из массивов небольшого размера сортируется по отдельности, после чего отсортированные массивы попарно объединяются с образованием отсортированных массивов большего размера. Описанное слияние массивов попарно осуществляется вплоть до завершения всего процесса сортировки. В общем случае, сортировка слиянием начинается с массивов размерности 1, которые сами по себе не нуждаются в сортировке. В данном примере сортировка начинается с массивов большей размерности, чтобы на каждый процессор приходилось по одному массиву. Блок-схема используемого алгоритма показана на рис. 7.2.

Детали реализации представлены в программе 7.2. Число задач задается пользователем в командной строке. Временные показатели сортировки приведены в приложении B. В упражнении 7.9 вам предлагается изменить программу sortMT таким образом, чтобы она сначала определяла количество доступных процессоров, используя для этого функцию GetSystemInfo, а затем создавала по одному потоку для каждого процессора.

Заметьте, что эта программа эффективно выполняется в однопроцессорных системах, в которых имеется достаточно большой запас оперативной памяти, и обеспечивает значительное повышение производительности в SMP-системах. *Предостережение.* Представленный алгоритм будет корректно работать лишь при условии, что число записей в сортируемом файле нацело делится на число потоков, а число потоков выражается степенью 2. В упражнении 7.8 упомянутые ограничения снимаются.

Примечание

Изучая работу этой программы, постарайтесь отделить логику управления потоками от логики определения части массива, которую должна сортировать тот или иной поток. Обратите также внимание на использование функции qsort из библиотеки

C, применение которой избавляет нас от необходимости самостоятельно разрабатывать эффективную функцию сортировки.

Программа 7.2. sortMT: сортировка слиянием с использованием нескольких потоков

```
/* Глава 7. SortMT.
   Сортировка файлов с использованием нескольких потоков (рабочая группа).
   sortMT [параметры] число_задач файл */

#include "EvryThng.h"
#define DATALEN 56 /* Данные: 56 байт; ключ: 8 байт. */
#define KEYLEN 8

typedef struct _RECORD {
 CHAR Key[KEYLEN];
 TCHAR Data[DATALEN];
} RECORD;

#define RECSIZE sizeof (RECORD)
typedef RECORD * LPRECORD;

typedef struct _THREADARG { /* Аргумент потока */
 DWORD iTh; /* Номер потока: 0, 1, 2, … */
 LPRECORD LowRec; /* Младшая часть указателя записи */
 LPRECORD HighRec; /* Старшая часть указателя записи */
} THREADARG, *PTHREADARG;

static int KeyCompare(LPCTSTR, LPCTSTR);
static DWORD WINAPI ThSort(PTHREADARG pThArg);
static DWORD nRec; /* Общее число записей, подлежащих сортировке. */
static HANDLE* ThreadHandle;

int _tmain(int argc, LPTSTR argv[]) {
 HANDLE hFile;
 LPRECORD pRecords = NULL;
 DWORD FsLow, nRead, LowRecNo, nRecTh, NPr, ThId, iTh;
 BOOL NoPrint;
 int iFF, iNP;
 PTHREADARG ThArg;
 LPTSTR StringEnd;
 iNP = Options(argc, argv, _T("n"), &NoPrint, NULL);
 iFF = iNP + 1;
 NPr = _ttoi(argv[iNP]); /* Количество потоков. */
  hFile = CreateFile(argv[iFF], GENERIC_READ | GENERIC_WRITE, 0, NULL,
OPEN_EXISTING, 0, NULL);
 FsLow = GetFileSize(hFile, NULL);
 nRec = FsLow / RECSIZE; /* Общее число записей. */
 nRecTh = nRec / NPr; /* Количество записей на один поток. */
 /* Распределить память для аргументов потока и массива дескрипторов и выделить
в памяти место для файла. Считать весь файл. */
 ThArg = malloc(NPr * sizeof(THREADARG));
 /* Аргументы потоков. */
 ThreadHandle = malloc(NPr * sizeof(HANDLE));
 pRecords = malloc(FsLow + sizeof(TCHAR));
 ReadFile(hFile, pRecords, FsLow, &nRead, NULL);
 CloseHandle(hFile);
```

```
    LowRecNo = 0; /* Создать потоки, выполняющие сортировку. */
    for (iTh = 0; iTh < NPr; iTh++) {
     ThArg[iTh].iTh = iTh;
     ThArg[iTh].LowRec = pRecords + LowRecNo;
     ThArg[iTh].HighRec = pRecords + (LowRecNo + nRecTh);
     LowRecNo += nRecTh;
      ThreadHandle[iTh] = (HANDLE)_beginthreadex (NULL, 0, ThSort, &ThArg[iTh],
CREATE_SUSPENDED, &ThId);
    }
    for (iTh = 0; iTh < NPr; iTh++) /* Запустить все потоки сортировки. */
     ResumeThread(ThreadHandle [iTh]);
    WaitForSingleObject(ThreadHandle[0], INFINITE);
    for (iTh = 0; iTh < NPr; iTh++) CloseHandle(ThreadHandle [iTh]);
    StringEnd = (LPTSTR)pRecords + FsLow;
    *StringEnd = '\0';
    if (!NoPrint) printf("\n%s", (LPCTSTR)pRecords);
    free(pRecords);
    free(ThArg);
    free(ThreadHandle);
    return 0;
    } /* Конец tmain. */

   static VOID MergeArrays(LPRECORD, LPRECORD);

   DWORD WINAPI ThSort(PTHREADARG pThArg) {
    DWORD GrpSize = 2, RecsInGrp, MyNumber, TwoToI = 1;
    LPRECORD First;
    MyNumber = pThArg->iTh;
    First = pThArg->LowRec;
    RecsInGrp = pThArg->HighRec - First;
    qsort(First, RecsInGrp, RECSIZE, KeyCompare);
    while ((MyNumber % GrpSize) == 0 && RecsInGrp < nRec) {
     /* Объединить слиянием отсортированные массивы. */
     WaitForSingleObject(ThreadHandle[MyNumber + TwoToI], INFINITE);
     MergeArrays(First, First + RecsInGrp);
     RecsInGrp *= 2;
     GrpSize *= 2;
     TwoToI *= 2;
    }
    _endthreadex(0);
    return 0; /* Подавить вывод предупреждающих сообщений. */
   }

   static VOID MergeArrays(LPRECORD p1, LPRECORD p2) {
    DWORD iRec = 0, nRecs, i1 = 0, i2 = 0;
    LPRECORD pDest, p1Hold, pDestHold;
    nRecs = p2 - p1;
    pDest = pDestHold = malloc(2 * nRecs * RECSIZE);
    p1Hold = p1;
    while (i1 < nRecs && i2 < nRecs) {
     if (KeyCompare((LPCTSTR)p1, (LPCTSTR)p2) <= 0) {
      memcpy(pDest, p1, RECSIZE);
      i1++;
      p1++;
      pDest++;
     } else {
      memcpy(pDest, p2, RECSIZE);
      i2++;
      p2++;
```

```
    pDest++;
   }
  }
  if (i1 >= nRecs) memcpy(pDest, p2, RECSIZE * (nRecs - i2));
  else memcpy(pDest, p1, RECSIZE * (nRecs - i1));
  memcpy(p1Hold, pDestHold, 2 * nRecs * RECSIZE);
  free (pDestHold);
  return;
}
```

Производительность

В приложении В представлены результаты сортировки файлов большого размера, содержащих записи длиной 64 байта, для случаев использования одной, двух и четырех потоков. SMP-системы позволяют получать значительно лучшие результаты. Упомянутый принцип "разделяй и властвуй" обеспечивает нечто большее, чем просто стратегию проектирования алгоритмов; он также служит ключом к использованию потоков и SMP. Результаты для однопроцессорных систем могут быть различными в зависимости от остальных характеристик системы. В системах с ограниченным объемом памяти (то есть объема физической памяти не достаточно для того, чтобы наряду с ОС и другими активными процессами в ней уместился весь файл) использование нескольких потоков увеличивает время сортировки, поскольку потоки состязаются между собой в захвате доступной физической памяти. С другой стороны, если памяти имеется достаточно, то многопоточный вариант может привести к повышению производительности и в случае однопроцессорных систем. Кроме того, как следует из приложения В, получаемые результаты существенно зависят от начального распределения данных.

Локальные области хранения потоков

Потокам могут требоваться собственные, независимо распределяемые и управляемые ими области памяти, защищенные от других потоков того же процесса. Одним из методов создания таких областей является вызов функции CreateThread (или _beginthreadex) с параметром lpvThreadParm, указывающим на структуру данных, уникальную для каждого потока. После этого поток может распределять память для дополнительных структур данных и получать доступ к ним через указатель lpvThreadParm. Эта методика используется в программе 7.1.

Кроме того, Windows предоставляет локальные области хранения потоков (Thread Local Storage, TLS), обеспечивающие каждый из потоков собственным массивом указателей. Организация TLS показана на рис. 7.3.

Индексы (строки) TLS первоначально не распределены, но в любой момент времени можно добавлять новые строки и освобождать существующие, причем минимально возможное число строк для любого процесса определяется значением TLS_MINIMUM_AVAILABLE (равным, по крайней мере, 64). Число столбцов может изменяться по мере создания новых потоков и завершения существующих.

Сначала мы рассмотрим управление индексами TLS. Логическим пространством для этого служит основной поток, но управлять индексами может любой поток.

Функция TlsAlloc возвращает распределенный индекс (> 0) или –1 (0xFFFFFFFF) в случае отсутствия доступных индексов.

```
DWORD TlsAlloc(VOID)
BOOL TlsFree(DWORD dwIndex)
```

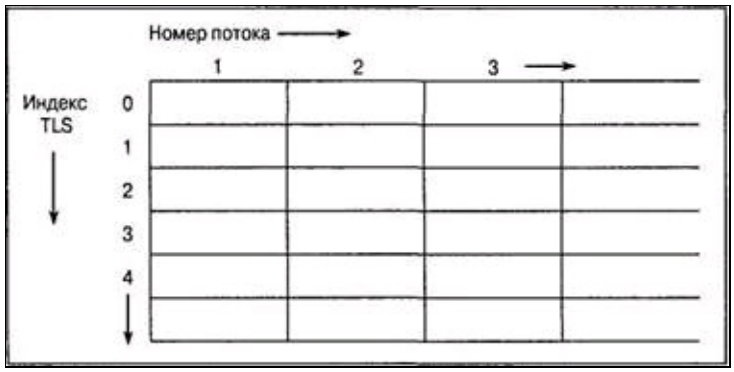


Рис. 7.3. Локальные области хранения потоков в контексте процесса

Каждый отдельный поток может выбирать и устанавливать значения (указатели типа void), связанные с ее областью памяти, используя индексы TLS.

Программист всегда должен убеждаться в том, что параметр индекса TLS является действительным, то есть что он был распределен с помощью функции TlsAlloc, но не был освобожден.

```
LPVOID TlsGetValue(DWORD dwTlsIndex)
BOOL TlsSetValue(DWORD dwTlsIndex, LPVOID lpTlsValue)
```

TLS предоставляют удобный механизм организации памяти, являющейся глобальной в контексте потока, но недоступной другим потокам. Обычные глобальные хранилища разделяются всеми потоками. Несмотря на то что ни один поток не может получить доступа к TLS другого потока, любой поток может уничтожить индекс TLS другого потока, вызвав

функцию TlsFree, так что этой функцией следует пользоваться с осторожностью. TLS часто используются DLL в качестве замены глобальной памяти библиотеки; в результате этого каждый поток получает в свое распоряжение собственную глобальную память. Кроме того, TLS обеспечивают вызывающим программам удобный способ взаимодействия с функциями DLL, и именно этот способ применения TLS является наиболее распространенным. В качестве примера в главе 12 (программа 12.4) TLS используются для создания библиотеки DLL с многопоточной поддержкой; другим важным элементом этого решения являются уведомления DLL о присоединении/отсоединении потоков и процессов путем вызова функции DllMain (глава 5).

Приоритеты процессов и потоков и планирование выполнения

Ядро Windows всегда запускает тот из потоков, готовых к выполнению, который обладает наивысшим приоритетом. Поток не является готовым к выполнению, если он находится в состоянии ожидания, приостановлен или блокирован по той или иной причине.

Потоки получают приоритеты на базе классов приоритета своих процессов. Как обсуждалось в главе 6, первоначально функцией CreateProcess устанавливаются четыре класса приоритета, каждый из которых имеет *базовый приоритет* (base priority):

- IDLE_PRIORITY_CLASS, базовый приоритет 4.
- NORMAL_PRIORITY_CLASS, базовый приоритет 9 или 7.
- HIGH_PRIORITY_CLASS, базовый приоритет 13.
- REALTIME_PRIORITY_CLASS, базовый приоритет 24.

Оба предельных класса используются редко, и в большинстве случаев можно обойтись нормальным (normal) классом. Windows NT (все версии) не является ОС реального времени (real-time), чего нельзя сказать, например, о Windows CE, и в случаях, аналогичных последнему, классом REALTIME_PRIORITY_CLASS следует пользоваться с осторожностью, чтобы не допустить вытеснения других процессов. Нормальный базовый приоритет равен 9, если фокус ввода с клавиатуры находится в окне; в противном случае этот приоритет равен 7.

Один процесс может изменить или установить свой собственный приоритет или приоритет другого процесса, если это разрешено атрибутами защиты.

```
BOOL SetPriorityClass(HANDLE hProcess, DWORD dwPriority)
DWORD GetPriorityClass(HANDLE hProcess)
```

Приоритеты потоков устанавливаются относительно базового приоритета процесса, и во время создания потока его приоритет устанавливается равным приоритету процесса. Приоритеты потоков могут принимать значения в интервале ±2 относительно базового приоритета процесса. Результирующим пяти значениям приоритета присвоены следующие символические имена:

- THREAD_PRIORITY_LOWEST
- THREAD_PRIORITY_BELOW_NORMAL
- THREAD_PRIORITY_NORMAL
- THREAD_PRIORITY_HIGHEST

Для установки и выборки относительного приоритета потока следует использовать эти значения. Обратите внимание на использование целых чисел со знаком вместо чисел типа DWORD.

```
BOOL SetThreadPriority(HANDLE hThread, int nPriority)
int GetThreadPriority(HANDLE hThread)
```

Существуют два дополнительных значения приоритета потоков. Они являются абсолютными, а не относительными, и используются только в специальных случаях.

- THREAD_PRIORITY_IDLE имеет значение 1 (или 16 — для процессов, выполняющихся в режиме реального времени).
- THREAD_PRIORITY_TIME_CRITICAL имеет значение 15 (или 31 — для процессов, выполняющихся в режиме реального времени).

Приоритеты потоков автоматически изменяются при изменении приоритета процесса.

Помимо этого, ОС может регулировать приоритеты потоков динамическим путем на основании поведения потоков. Вы можете активизировать или отключить это средство с помощью функции SetThreadPriorityBoost.

Предостережение относительно использования приоритетов потоков и процессов

Высокими приоритетами потоков и высокими классами приоритета процессов необходимо пользоваться с осторожностью. Следует решительно избегать использования приоритетов реального времени для обычных пользовательских процессов; эти приоритеты должны использоваться лишь в тех случаях, когда приложения действительно являются приложениями реального времени. Нарушение этого правила чревато тем, что пользовательские потоки будут тормозить выполнение потоков операционной системы.

Кроме того, приводимые в последующих главах высказывания относительно корректности многопоточных программ справедливы лишь при условии соблюдения принципа *равноправия* (fairness) потоков. Равноправие потоков означает, что все они, в конечном счете, будут выполняться. Если не соблюдать этот принцип, то потоки с более низким приоритетом смогут удерживать ресурсы, необходимые потокам, имеющим более высокий приоритет. При описании недостатков планирования, осуществляемого с нарушением принципа равноправия, используют термины зависание потоков (thread starvation) и инверсия приоритетов (priority inversion).

Состояния потоков

На рис. 7.4, взятом из [9] (см. также [38], версию, обновленную Соломоном (Solomon) и Руссиновичем (Russinovych)), представлена схема планирования потоков и показаны их возможные состояния. Кроме того, этот рисунок иллюстрирует результаты работы программы. Такие диаграммы состояния являются общими для всех многозадачных ОС и помогают выяснить, каким образом планируется выполнение потоков и как они переходят из одного состояния в другое.

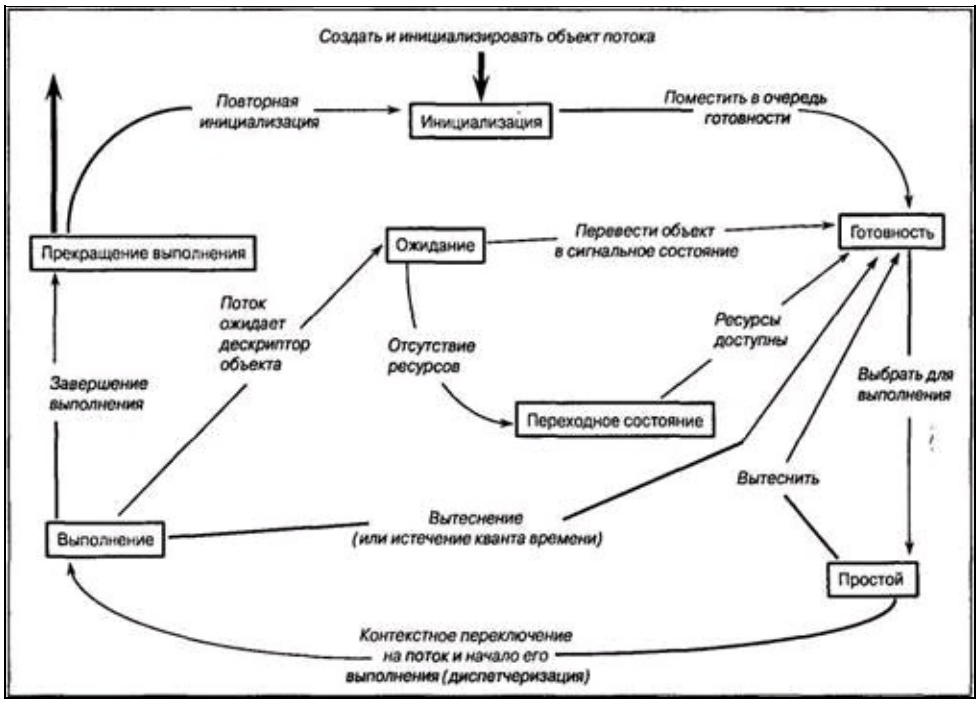


Рис. 7.4. Состояния потоков и переходы между состояниями (Источник: *Inside Windows NT,* Copyright © 1993, by Helen Custer. Copyright Microsoft Press. Воспроизводится с разрешения Microsoft Press. Все права сохранены.)

Ниже приводится краткая сводка основных положений. Для получения более подробной информации по этому вопросу обратитесь в [38] или к руководству по ОС.

• Поток находится в *состоянии выполнения* (running state), если она фактически выполняется процессором. В SMP-системах в состоянии выполнения могут находиться одновременно несколько потоков.

• Планировщик переводит поток в *состояние ожидания* (wait state), если он выполняет функцию ожидания несигнализирующих объектов, например, потоков или процессов, или перехода в сигнальное состояние объектов синхронизации, о чем говорится в главе 8. Операции ввода/вывода также будут ожидать завершения передачи дисковых или иных данных, но ожидание может быть вызвано и другими многочисленными функциями. О потоках, находящихся в состоянии ожидания, нередко говорят как о *блокированных* (blocked) или *спящих* (sleeping).

• Поток находится в *состоянии готовности* (ready state), если она может выполняться. Планировщик в любой момент может перевести такой поток в состояние выполнения. Когда процессор станет доступным, планировщик запустит тот из потоков, находящихся в состоянии готовности, который обладает наивысшим приоритетом, а при наличии нескольких потоков с одинаковым высшим приоритетом запущен будет та, который пребывал в состоянии готовности дольше всех. При этом поток проходит через *состояние простоя* (standby state), или *резервное состояние.*

• Обычно, в соответствии с приведенным описанием, планировщик помещает поток, находящийся в состоянии готовности, на любой доступный процессор. Программист может указать *маску родства процессоров* (processor affinity mask) для потока (см. главу 9), предоставляя потоку процессоры, на которых он может выполняться. Используя этот способ, программист может распределять процессоры между потоками. Соответствующими функциями являются SetProcessorAffinityMask и GetProcessorAffinityMask. Функция SetThreadIdealProcessor позволяет указать предпочтительный процессор, подлежащий использованию планировщиком при первой же возможности.

• После истечения кванта времени, отведенного выполняющемуся потоку, планировщик без ожидания переводит его в состояние готовности. В результате выполнения функции Sleep(0) поток также будет переведен из состояния выполнения в состояние готовности.

• Планировщик переводит ожидающий поток в состояние готовности сразу же, как только соответствующие дескрипторы оказываются в сигнальном состоянии, хотя при этом поток фактически проходит через промежуточное *переходное состояние* (transition state). В подобных случаях принято говорить о том, что поток *пробуждается* (wakes).

• Не существует способа, позволяющего программе определить состояние другого потока (разумеется, если поток выполняется, то он находится в состоянии выполнения, и поэтому ему нет никакого смысла определять свое состояние). Даже если бы такой способ и существовал, то состояние потока может измениться еще до того, как опрашивающий поток успеет предпринять какие-либо действия в ответ на полученную информацию.

• Поток, независимо от его состояния, может быть приостановлен (suspended), и приостановленный поток не будет запущен, даже если он находится в состоянии готовности. В случае приостановки выполняющегося потока, независимо от того, по собственной ли инициативе или по инициативе потока, выполняющегося на другом процессоре, он переводится в состояние готовности.

• Поток переходит в *состояние завершения* (terminated state) тогда, когда его выполнение завершается, и остается в этом состоянии до тех пор, пока остается открытым хотя бы один из ее дескрипторов. Это позволяет другим потокам запрашивать состояние данного потока и его код завершения.

Возможные ловушки и распространенные ошибки

Существует ряд факторов, о которых следует всегда помнить при разработке многопоточных программ. Пренебрежение некоторыми базовыми принципами может привести к появлению серьезных дефектов в программе, и лучше заранее стремиться к тому, чтобы не допустить ошибок, чем впоследствии затрачивать время на тестирование и отладку программ.

Существенно то, что потоки выполняются в асинхронном режиме. По отношению к ним не действует никакая система упорядочения, если только вы не создали ее явно. Именно асинхронное поведение потоков делает их столь полезными, однако при отсутствии должного внимания можно столкнуться с серьезными трудностями.

Часть соответствующих рекомендаций представлена ниже, а остальные будут даваться по мере изложения материала в последующих главах.

• Не делайте никаких предположений относительно очередности выполнения родительских и дочерних потоков. Вполне возможно, что дочерний поток будет выполняться вплоть до своего завершения, прежде чем родительский поток вернется из функции CreateThread, или наоборот, дочерний поток может вообще не выполняться в течение длительного периода времени. В случае же SMP-систем возможно параллельное выполнение родительского и одного или нескольких дочерних потоков.

• Убедитесь в том, что до вызова функции CreateThread были завершены все действия по инициализации данных, необходимые для правильной работы дочернего потока, либо приостановите поток или же воспользуйтесь любой другой подходящей методикой. Несвоевременная инициализация данных, требуемых дочерним потоком, может создать "условия состязаний" ("race conditions"), суть которых заключается в том, что родительский поток "состязается" с дочерним, чтобы инициализировать данные до того, как они начнут использоваться дочерним потоком.

• Проследите за тем, чтобы каждый отдельный поток имел собственную структуру данных, переданную ему через параметр функции потока. Не делайте никаких предположений относительно очередности завершения дочерних потоков (иначе можете столкнуться с другой разновидностью "проблемы состязаний").

• Выполнение любого потока может быть прервано в любой момент, и точно так же выполнение любого потока в любой момент может быть возобновлено.

• Не пользуйтесь приоритетами потоков в качестве замены явной синхронизации.

• Никогда не прибегайте к аргументации наподобие "вряд ли это может произойти" при анализе корректности программы. Если что-то *может* произойти, оно *обязательно* произойдет, причем тогда, когда вы меньше всего этого ожидаете.

• В еще большей степени, чем в случае однопоточных программ, справедливо утверждение о том, что, хотя тестирование и необходимо, но его одного еще не достаточно для проверки корректности многопоточной программы. Довольно часто программы способны успешно пройти через самые различные тесты, несмотря на наличие в них дефектов. Ничто не может заменить тщательно выполненных проектирования, реализации и анализа кода.

• Поведение многопоточных программ может заметно меняться в зависимости от быстродействия и количества процессоров, версии ОС и множества других факторов. Тестирование программы в самых различных системах помогает выявлению всевозможных дефектов, но предыдущее предостережение остается в силе.

• Убедитесь в том, что предусмотрели для потоков достаточно большой объем стека, хотя заданного по умолчанию стека размером 1 Мбайт в большинстве случаев вам должно хватить.

• Потоки следует использовать только тогда, когда это действительно необходимо. Таким образом, если по самой своей природе некоторые операции допускают параллелизм, то каждое такое действие может быть представлено потоком. В то же время, если операциям присуща очередность, то потоки лишь усложнят программу, а связанный с ними расход системных ресурсов может привести к снижению производительности.

• К счастью, чаще всего корректно работают самые простые программы и те, отличительной чертой которых является элегантность. При малейшей возможности избегайте усложнения программ.

Ожидание в течение конечного интервала времени

Наконец, рассмотрим функцию Sleep, позволяющую потоку отказаться от процессора и перейти из состояния выполнения в состояние ожидания, которое будет длиться в течение заданного промежутка времени. Например, выполнение задачи потоком может продолжаться в течение некоторого периода времени, после чего поток приостанавливается. По истечении периода ожидания планировщик вновь переводит поток в состояние готовности. Именно эта техника применена в одной из программ в главе 11 (программа 11.4).

```
VOID Sleep(DWORD dwMilliseconds)
```

Длительность интервала ожидания указывается в миллисекундах, и одним из ее возможных значений является INFINITE, что соответствует бесконечному периоду ожидания, при котором выполнение приостанавливается на неопределенное время. Значению 0 соответствует отказ потока от оставшейся части отведенного ей временного промежутка; в этом случае ядро переводит поток из состояния выполнения в состояние готовности, как показано на рис. 7.4.

Функция SwitchToThread предоставляет потоку еще один способ уступить процессор другому потоку из числа тех, которые находятся в состоянии готовности, если таковые имеются.

> UNIX-функция sleep аналогична функции Sleep, но длительность периода ожидания измеряется в секундах. Чтобы получить миллисекундное разрешение, используйте функции select или poll без дескрипторов файлов.

Облегченные потоки

Примечание

Облегченные потоки относятся к специальной тематике. Ознакомьтесь с комментарием, включенным в конце первого абзаца приведенного ниже списка, и решите для себя, стоит ли вам читать данный раздел.

Облегченные потоки (fibers), как говорит само их название, являются элементами потока. Точнее, облегченный поток — это единица выполнения в контексте потока, планируемая приложением, а не ядром. В потоке могут быть запланированы несколько облегченных потоков, и облегченные потоки сами определяют, какой из них должен выполняться следующим. Облегченные потоки имеют независимые стеки, но во всем остальном выполняются исключительно в контексте потока, имея, например, доступ к TLS потока и любому мьютексу[27], владельцем которого является данный поток. Более того, вся работа с облегченными потоками осуществляется вне ядра исключительно в пользовательском пространстве. Между потоками и облегченными потоками существуют многочисленные отличия.

Облегченные потоки могут использоваться в нескольких целях.

• Следует отметить тот немаловажный факт, что во многих приложениях, особенно в приложениях UNIX, использующих патентованные реализации потоков, которые в настоящее время можно, как правило, считать устаревшими, предусмотрено планирование собственных потоков. Использование облегченных потоков упрощает перенос таких приложений в среду Windows. *Поскольку для большинства читателей этот вопрос не является актуальным, они, вероятно, предпочтут пропустить данный раздел.*

• Потоки вовсе не обязательно должны блокироваться в ожидании блокировки файла, мьютекса, именованного входного канала или иных ресурсов. Вместо этого один облегченный поток может опрашивать ресурсы и, если эти ресурсы остаются недоступными, передавать управление другому указанному облегченному потоку.

• Облегченные потоки действуют в контексте потока и имеют доступ к ресурсам потока и процесса. В отличии от потоков вытесняющее планирование к облегченным потокам не применяется. В действительности планировщику Windows об облегченных потоках ничего не известно; управление такими потоками осуществляется из DLL облегченных потоков исключительно в пользовательском пространстве.

• Облегченные потоки позволяют реализовать механизм *сопрограмм* (co-routines), посредством которого приложение может переключаться между несколькими взаимосвязанными задачами. Добиться этого с помощью потоков невозможно, поскольку в распоряжении программиста нет средств, обеспечивающих непосредственное управление очередностью выполнения потоков.

• Основные разработчики программного обеспечения, использующие облегченные потоки, представляют этот элемент как фактор повышения производительности. Так, в приложении Oracle Database 10g предусмотрена возможность переключения в "режим облегченных потоков" ("fiber mode") (см. http://download.oracle.com/owsf_2003/40171_colello.ppt; там же находится описание многопоточной модели).

API облегченных потоков представлен шестью функциями. Порядок их использования описывается ниже и иллюстрируется рис. 7.5.

1. Прежде всего, поток должен сделать возможным выполнение облегченного потока,

вызвав функцию ConvertThreadToFiber. В результате этого поток становится облегченным потоком, который может рассматриваться в качестве *основного* (primary). Вызов упомянутой функции обеспечивает получение указателя на данные облегченного потока, который может быть использован во многом так же, как аргумент потока использовался для создания специфических для этого потока данных.

2. Основной или другие облегченные потоки создают дополнительные облегченные потоки с помощью функции CreateFiber. Каждый облегченный поток характеризуется начальным адресом, размером стека и параметром. Каждый новый облегченный поток идентифицируется с помощью адреса, а не дескриптора.

3. Отдельный облегченный поток может получить свои данные, назначенные ему функцией CreateFiber, обратившись к функции GetFiberData.

4. Аналогично, облегченный поток может идентифицировать себя при помощи функции GetCurrentFiber.

5. Выполняющийся облегченный поток может уступить управление другому облегченному потоку, указав его адрес в вызове функции SwitchToFiber. Облегченные потоки должны явно указывать очередной облегченный поток, который должен выполняться в контексте данного потока.

6. Функция DeleteFiber уничтожает существующий облегченный поток и все относящиеся к нему данные.

7. В Windows XP (NT 5.1) наряду с локальными областями хранения облегченных потоков введены новые функции, такие, например, как ConvertFiberToThread (которая освобождает ресурсы, созданные функцией ConvertThreadToFiber).

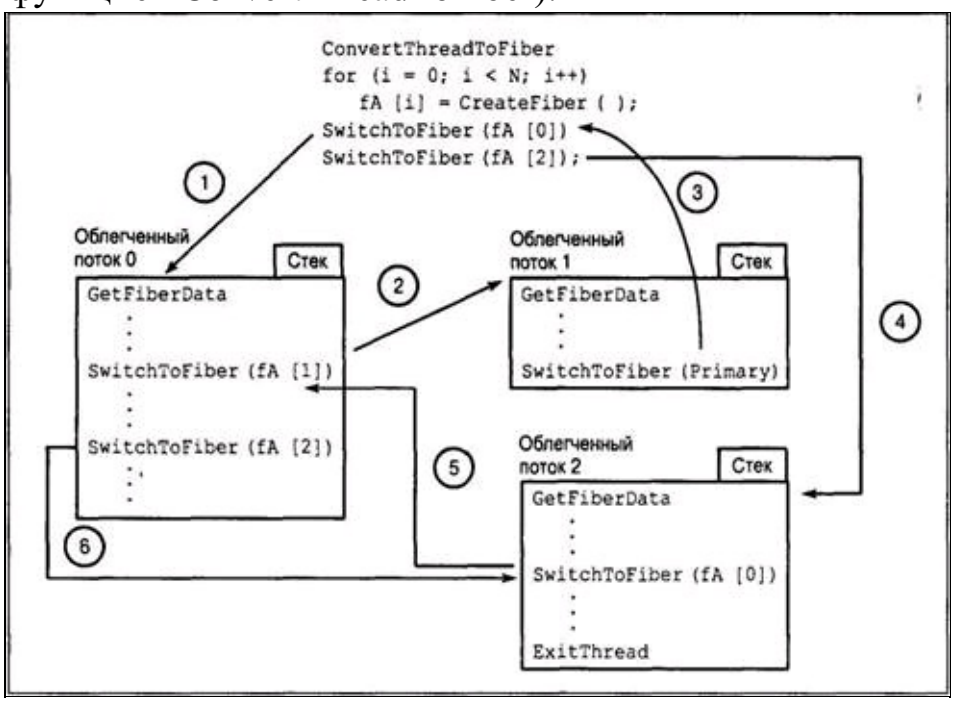


Рис. 7.5. Передача управления между облегченными потоками внутри потока

Схема взаимодействия между облегченными потоками в контексте потока представлена на рис. 7.5. Этот пример иллюстрирует два способа вытеснения одного потока другим.

• **Подчиненное планирование (master-slave scheduling).** Только один, главный (master) облегченный поток, в данном случае — основной, принимает решения относительно того, какой облегченный поток должен выполняться, и этот облегченный поток всегда уступает управление главному облегченный потоке. На рис. 7.5 главным является облегченный поток 1.

• **Равноправное планирование (peer-to-peer scheduling).** Облегченный поток сам

определяет, какой из других облегченных потоков должен выполняться следующим. Определение очередного облегченного потока может базироваться на таких стратегиях, как круговое планирование (round-robin scheduling), приоритетное планирование на основании схемы приоритетов и тому подобное. По принципу равноправного планирования реализуются сопрограммы. На рис. 7.5 такого типа передача управления осуществляется между облегченными потоками 0 и 2.

Резюме

Windows поддерживает потоки, которые планируются независимо друг от друга, но разделяют адресное пространство и ресурсы одного и того же процесса. Потоки дают программисту возможность упростить программу и использовать параллелизм выполнения задач для повышения производительности приложения. Потоки могут обеспечивать выигрыш в производительности даже в однопроцессорных системах.

В следующих главах

Рассмотрение темы синхронизации, которое начинается в главе 8 с описания и сравнительного анализа объектов синхронизации Windows, продолжается в главах 9 и 10 обсуждением более сложных вопросов синхронизации с привлечением многочисленных примеров. В главе 11 реализуется сервер с многопоточной поддержкой; он показан на рис. 7.1.

Дополнительная литература

Windows

Книга [1] полностью посвящена потокам Win32. Также заслуживают внимания книги [26] и [7]. В то же время во многих из этих и других книг не учтены новшества, появившиеся в Windows 2000, XP и Server 2003.

UNIX и Pthreads

В книге [40] применение потоков в UNIX не рассматривается, однако для изучения этой темы можно порекомендовать книгу [6]. В этой книге даются многочисленные рекомендации, касающиеся проектирования и реализации многопоточных программ. Приведенная в ней информация применима в равной степени как к потокам Pthreads, так и к потокам Windows, и многие примеры без труда переносятся в Windows. В ней также хорошо изложены модели "хозяин/рабочий", "клиент/сервер" и конвейерная модель, и представление Бутенхофа (Butenhof) было положено в основу описания указанных моделей в данной главе.

Упражнения

7.1. Реализуйте набор функций, позволяющий приостанавливать и возобновлять выполнение потоков, и, кроме того, получать значение счетчика приостановок потоков.

7.2. Сравните производительность программ параллельного поиска, одна из которых использует потоки (программа 7.1, GrepMT), а другая — процессы (программа 6.1, GrepMP). Сравните полученные результаты с теми, которые приведены в приложении В.

7.3. Проведите дополнительное исследование производительности программы GrepMT для случаев, когда файлы находятся на различных дисках или являются сетевыми файлами. Определите также выигрыш в производительности, если таковой будет наблюдаться, в случае SMP-систем.

7.4. Измените программу 7.1, GrepMT, таким образом, чтобы она выводила результаты в той очередности, в какой файлы указаны в командной строке. Сказываются ли эти изменения каким-либо образом на показателях производительности?

7.5. Дополнительно усовершенствуйте программу 7.1, GrepMT, таким образом, чтобы она выводила время, потребовавшееся для выполнения каждого из потоков. Для этого вам понадобится функция GetThreadTimes, аналогичная функции GetProcessTimes, которая была использована в главе 6. Указанное усовершенствование сможет работать лишь в Windows NT4 и более поздних версиях.

7.6. На Web-сайте книги находится многопоточная программа подсчета слов, wcMT.c, структура которой аналогична структуре программы grepMT.c. Там же находится версия этой программы, wcMTx, в которую были намеренно введены некоторые дефекты. Попытайтесь найти и устранить указанные дефекты, в том числе и синтаксические ошибки, не сверяясь с корректным решением. Кроме того, создайте тестовые примеры, иллюстрирующие эти дефекты, и выполните эксперименты по определению производительно сти, аналогичные тем, которые предлагались для программы grepMT. На Web-сайте находится также однопоточная версия упомянутой программы, wcST.c, которую можно использовать для того, чтобы определить, обеспечивают ли потоки выигрыш в производительности по сравнению с последовательной обработкой.

7.7. На Web-сайте находится программа grepMTx.c, в которой имеются дефекты, связанные с нарушением базовых правил, соблюдение которых необходимо для безопасного выполнения нескольких потоков. Опишите, к чему это приводит, а также найдите и устраните ошибки.

7.8. Для правильного выполнения программы sortMT требуется, чтобы количество записей в массиве нацело делилось на количество потоков, а количество потоков равнялось степени 2. Устраните эти ограничения.

7.9. Усовершенствуйте программу sortMT таким образом, чтобы в случаях, когда количество потоков, заданное в командной строке, равно 0, программа определяла количество процессоров, установленных в локальной системе, с помощью функции GetSystemInfo. Задавая количество потоков равным различным кратным количества процессоров (используя коэффициенты кратности 1, 2, 4 и так далее), определите, изменяется ли при этом производительность.

7.10. Видоизмените программу sortMT таким образом, чтобы рабочие потоки не приостанавливались при их создании. Сказываются ли, и если да, то каким именно образом, возникающие при этом нежелательные условия состязаний на работе программы?

7.11. В программе sortMT, еще до того, как создаются потоки, выполняющие сортировку, осуществляется считывание всего файла основным потоком. Видоизмените программу таким образом, чтобы каждый поток самостоятельно считывал необходимую часть файла.

7.12. Видоизмените одну из приведенных в данной главе программ (grepMT или sortMT) таким образом, чтобы специфическая для потоков информация частично или полностью передавалась через TLS, а не через структуры данных.

7.13. Наблюдается ли выигрыш в производительности в случае предоставления некоторым потокам в программе sortMT более высокого приоритета по сравнению со всеми остальными? Например, может оказаться выгодным предоставить таким потокам, как поток 3 на рис. 7.2, которая занята лишь сортировкой без слияния, более высокий приоритет, чем всем остальным. Объясните результаты.

7.14. В программе sortMT все потоки создаются в приостановленном состоянии с той целью, чтобы избежать создания условий состязаний. Видоизмените эту программу таким образом, чтобы потоки создавались в обратном порядке и в состоянии выполнения. Продолжают ли после этого оставаться какие-либо предпосылки для существования условий состязаний? Сравните показатели производительности измененной и исходной версий программы.

7.15. Алгоритм быстрой сортировки (Quicksort), обычно используемый функцией qsort библиотеки C, как правило, характеризуется высоким быстродействием, но в некоторых случаях может замедляться. В большинстве учебников по алгоритмам рассматривается его версия, которая работает быстрее всего в тех случаях, когда массив отсортирован в обратном порядке, и медленнее всего, когда массив является уже отсортированным. Однако реализация этого алгоритма в библиотеке Microsoft C ведет себя иначе. Определите из кода библиотечной программы, какого типа последовательности элементов массива будут приводить к наилучшим и наихудшим результатам, и исследуйте показатели производительности программы sortMT для этих двух крайних случаев. Как влияет на результаты увеличение или уменьшение количества потоков? *Примечание.* Исходный код библиотеки C мог быть установлен в подкаталоге CRT основного каталога Visual Studio на вашей машине. Ищите функцию qsort.c. Кроме того, вы можете попытаться отыскать эту функцию на установочном компакт-диске.

7.16. На Web-сайте содержится программа sortMTx, в которую были намеренно внесены некоторые дефекты. Продемонстрируйте наличие дефектов на тестовых примерах, а затем объясните и устраните указанные дефекты, не сверяясь с корректными решениями. *Предостережение.* Версии программ, в которых присутствуют дефекты, могут содержать как синтаксические ошибки, так и ошибки в логике организации выполнения потоков.

7.17. Прочитайте статью Джейсона Кларка (Jason Clark) "Waiting for More than 64 Objects" ("Ожидание более чем 64 объектов"), опубликованную в октябрьском номере журнала *Windows Developer's Journal* за 1997 год. Примените описанное в ней решение к программе grepMT. Найти старые выпуски журналов иногда бывает очень трудно, поэтому воспользуйтесь каким-нибудь поисковым механизмом для поиска сразу по нескольким ключевым словам. Мною для этого поиска была использована фраза "wait for multiple objects more than 64", хотя другие варианты могут оказаться более эффективными.

ГЛАВА 8
Синхронизация потоков

Потоки могут упрощать проектирование и реализацию программ и повышать их производительность, но их использование требует принятия мер по защите разделяемых ресурсов от попыток их изменения одновременно несколькими потоками, а также создания таких условий, при которых потоки выполняются лишь в ответ на запрос или тогда, когда это является необходимым. В настоящей главе представлены способы решения этих задач с помощью объектов синхронизации Windows — критических участков кода, мьютексов, семафоров и событий, а также описаны некоторые из проблем, например, взаимоблокировка потоков и возникновение состязаний между ними, которые могут наблюдаться в результате неправильного использования потоков. Объекты синхронизации могут применяться для синхронизации потоков, принадлежащих как одному и тому же, так и различным процессам.

Примеры иллюстрируют объекты синхронизации, а также создают почву для обсуждения как положительных, так и отрицательных аспектов применения тех или иных методов синхронизации на производительность. В последующих главах демонстрируется использование синхронизации для решения дополнительных задач программирования и повышения производительности программ, а также рассказывается о возможных ловушках и применении более развитых средств.

Синхронизация потоков является одной из важнейших и интереснейших тем и играет существенную роль почти в любом многопоточном приложении. *Тем не менее, те из читателей, которые заинтересованы главным образом в межпроцессном взаимодействии, сетевом программировании и построении серверов с многопоточной поддержкой, могут перейти непосредственно к главе 11 и вернуться к изучению глав 8-10 в качестве вспомогательного материала, лишь в том случае, если в этом возникнет необходимость.*

Необходимость в синхронизации потоков

В главе 7 были продемонстрированы методы создания рабочих потоков и управления ими в условиях, когда каждый рабочий поток обращался к собственным ресурсам. В приведенных в главе 7 примерах каждый поток обрабатывает отдельный файл или отдельную область памяти, но даже и в этом случае возникает необходимость в простейшей синхронизации во время создания и завершения потоков. Так, в программе grepMT все рабочие потоки выполняются независимо друг от друга, но главный поток должен ожидать завершения рабочих потоков, прежде чем вывести сгенерированные ими результаты. Заметьте, что главный поток разделяет общую память с рабочими потоками, но структура программы гарантирует, что главный поток не получит доступа к памяти до тех пор, пока рабочий поток не завершит своего выполнения.

Программа sortMT несколько сложнее, поскольку рабочие потоки должны синхронизировать свое выполнение, ожидая завершения смежных потоков, и не могут быть запущены до тех пор, пока главный поток не создаст все рабочие потоки. Как и в случае программы grepMT, синхронизация достигается за счет ожидания завершения одного или нескольких потоков.

Однако во многих случаях требуется, чтобы выполнение двух и более потоков могло координироваться на протяжении всего времени жизни каждой из них. Например, несколько потоков могут обращаться к одной и той же переменной или набору переменных, и тогда возникает вопрос о взаимоисключающем доступе. В других случаях поток не может продолжать выполнение до тех пор, пока другой поток не достигнет определенного этапа выполнения. Каким образом программист может получить уверенность в том, что, например, два или более потоков не попытаются одновременно изменить данные, хранящиеся в глобальной памяти, такие, например, как статистические данные о производительности? Как, далее, программист может добиться того, чтобы поток не предпринимал попыток удаления элемента из очереди, если очередь не содержит хотя (бы одного элемента?

Несколько примеров иллюстрируют ситуации, которые могут приводить к нарушению условий безопасного выполнения нескольких потоков. (Код считается безопасным в этом смысле, если он может выполняться одновременно несколькими потоками без каких-либо нежелательных последствий.) Условия безопасного выполнения потоков обсуждаются далее в этой и последующих главах.

На рис. 8.1 показано, что может случиться, когда две несинхронизированные потоки разделяют общий ресурс, например ячейку памяти. Оба потока увеличивают значение переменной N на единицу, но в силу специфики очередности, в которой могут выполняться потоки, окончательное значение N равно 5, тогда как правильным значением является 6. Заметьте, что представленный здесь частный результат не обладает ни повторяемостью, ни предсказуемостью; другая очередность выполнения потоков могла бы привести к правильному результату. В SMP-системах эта проблема еще более усугубляется.

Критические участки кода

Инкрементирование N при помощи единственного оператора, например, в виде N++, не улучшает ситуацию, поскольку компилятор сгенерирует последовательность из одной или более машинных инструкций, которые вовсе не обязательно должны выполняться *атомарно* (atomically), то есть как одна неделимая единица выполнения.

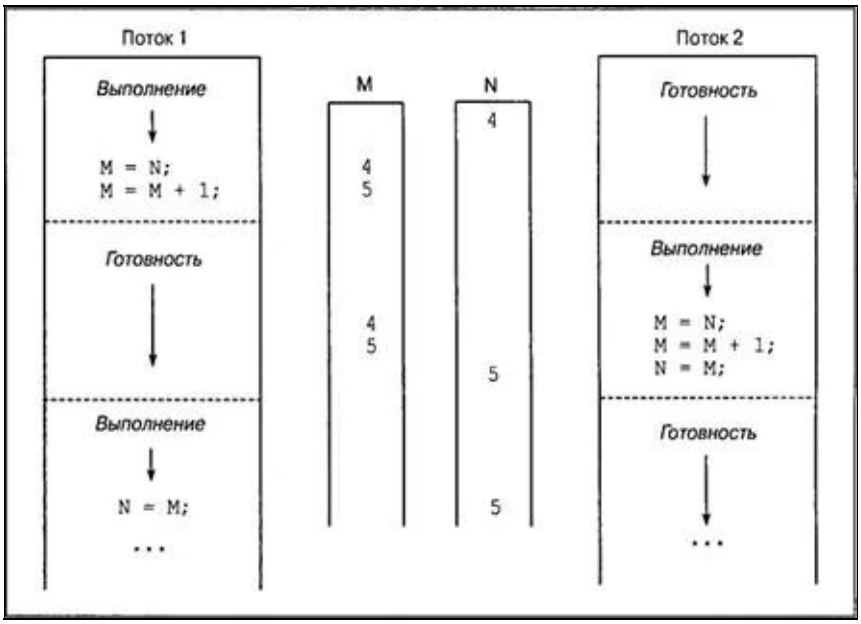


Рис. 8.1. Разделение общей памяти несинхронизированными потоками

Основная проблема состоит в том, что имеется *критический участок кода* (critical section) (в данном примере — код, который увеличивает N на 1), характеризующийся тем, что если один из потоков приступил к его выполнению, то никакой другой поток не должен входить в данный код до тех пор, пока его не покинет первый поток. Проблему критических участков кода можно считать разновидностью проблемы состязаний, поскольку первый поток "состязается" со вторым потоком в том, чтобы завершить выполнения критического участка еще до того, как его начнет выполнять любой другой поток. Таким образом, мы должны так синхронизировать выполнение потоков, чтобы можно было гарантировать, что в каждый момент времени код будет выполняться только одним потоком.

Неудачные пути решения проблемы критических участков кода

К аналогичным непредсказуемым результатам будет приводить и код, в котором предпринимается попытка защитить участок инкрементирования переменной путем опроса состояния флага.

```
while (Flag) Sleep (1000);
Flag = TRUE;
N++;
Flag = FALSE;
```

Даже в этом случае поток может быть вытеснен в процессе выполнения программы от момента тестирования значения флага до момента, когда его значение будет установлено равным TRUE; критический участок кода образуют два оператора, которые не защищены должным образом от параллельного доступа к ним двух и более потоков.

Другая разновидность попытки решения проблемы синхронизации выполнения потоками критического участка кода могла бы состоять в том, чтобы предоставить каждому потоку собственный экземпляр переменной N, например, так, как показано ниже:

```
DWORD WINAPI ThFunc(TH_ARGS pArgs) {
 volatile DWORD N;
 … N++; …
}
```

Однако такой подход ничем не лучше предыдущего, поскольку каждый поток имеет собственный экземпляр переменной в своем стеке, но может, например, требоваться, чтобы N

представляло суммарное число действующих потоков. В то же время, этот тип решения необходим в тех случаях, когда каждый поток должен иметь собственный, независимый от других потоков экземпляр переменной. Эта методика часто встречается в наших примерах.

Заметьте, что проблемы подобного рода не ограничиваются случаем потоков одного процесса. С этими проблемами приходится сталкиваться также в случаях, когда два процесса разделяют общую память или изменяют один и тот же файл.

Класс памяти volatile

Даже если решить проблему синхронизации, все равно остается еще один скрытый дефект. Оптимизирующие компиляторы могут оставлять значение N в регистре, а не заносить его обратно в ячейку памяти, соответствующую переменной N. Попытка решения этой проблемы путем переустановки переключателей опций компилятора окажет отрицательное воздействие на скорость выполнения остальных участков программы. Правильное решение состоит в том, чтобы использовать определенный в стандарте ANSI C спецификатор памяти volatile, который гарантирует, что после изменения значения переменной оно будет сохраняться в памяти, а при необходимости будет всегда извлекаться из памяти. Ключевое слово volatile сообщает компилятору, что значение переменной может быть в любой момент изменено.

Функции взаимоблокировки

Если все, что требуется — это увеличение, уменьшение или обмен значениями переменных, как в нашем первом простом примере, то *функций взаимоблокировки* (interlocked functions) вам будет вполне достаточно. Функции взаимоблокировки проще в использовании, обеспечивают более высокое быстродействие по сравнению с другими возможными методами и не приводят к блокированию потоков. Двумя членами этого семейства функций, которые представляют для нас интерес, являются функции InterlockedIncrement и InterlockedDecrement. Обе функции применяются по отношению к 32-битовым целым числам со знаком.

Эти функции имеют ограниченную область применимости, но будут использоваться нами при любой удобной возможности.

Задача инкрементирования N, представленная на рис. 8.1, может быть реализована посредством единственной строки кода:

```
InterlockedIncrement(&N);
```

N — это целое число типа long со знаком, и функция возвращает его новое значение, несмотря на то что другой поток мог изменить значение N еще до того, как поток, вызвавший функцию InterlockedIncrement, успеет воспользоваться возвращенным значением.

Следует, однако, проявлять осторожность и, например, не вызывать эту функцию два раза подряд, если, значение переменной должно быть увеличено на 2, поскольку поток может быть вытеснен в промежутке между двумя вызовами функции. Вместо этого лучше воспользоваться функцией InterlockedExchangeAdd, описание которой приводится далее в настоящей главе.

Локальная и глобальная память

Суть другого требования, предъявляемого к корректному многопоточному коду, состоит в том, что глобальная память не должна использоваться для локальных целей. Так, применение функции ThFunc, приводившейся ранее в качестве примера, будет необходимым и уместным в

тех случаях, когда поток должен располагать собственным экземпляром N. N может быть использовано для хранения временных результатов или размещения аргумента функции. Если же N размещается в глобальной памяти, то все процессы будут разделять единственный экземпляр N, что может стать причиной некорректного поведения программы, как бы тщательно вы ни планировали синхронизацию доступа к этой переменной. Ниже приводится пример подобного некорректного использования N. N должно быть локальной переменной, размещаемой в стеке функции потока.

```
DWORD N;
DWORD WINAPI ThFunc (TH_ARGS pArgs) {
 …
 N = 2 * pArgs->Count; …
}
```

Резюме: безопасный многопоточный код

Прежде чем мы приступим к рассмотрению объектов синхронизации, ознакомьтесь с пятью начальными рекомендациями, соблюдение которых будет гарантировать корректное выполнение программ в многопоточной среде.

1. Переменные, являющиеся локальными по отношению к потоку, не должны быть статическими, и их следует размещать в стеке потока или же в структуре данных или TLS, непосредственный доступ к которым имеет только отдельный поток.

2. В тех случаях, когда функцию могут вызывать несколько потоков, а какой-либо специфический для состояния потока параметр, например счетчик, должен сохранять свое значение в течение промежутков времени, отделяющих один вызов функции от другого, значение параметра состояния должно храниться в TLS или в структуре данных, выделенной специально для этого потока, например, в структуре данных, передаваемой потоку при его создании. Использовать стек для сохранения постоянно хранимых (persistent) значений не следует. Применение необходимой методики при построении безопасных многопоточных DLL иллюстрируют программы 12.4 и 12.5.

3. Старайтесь не создавать предпосылок для формирования условий состязаний наподобие тех, которые возникли бы в программе 7.2 (sortMT), если бы потоки не создавались в приостановленном состоянии. Если предполагается, что в определенной точке программы должно выполняться некоторое условие, используйте ожидание объекта синхронизации для гарантии того, что, например, дескриптор всегда будет ссылаться на существующий поток.

4. Вообще говоря, потоки не должны изменять окружение процесса, поскольку это окажет воздействие на все потоки. Таким образом, поток не должен определять дескрипторы стандартного ввода и вывода или изменять переменные окружения. Это не касается только основного потока, который может вносить такие изменения до создания других потоков.

5. Переменные, разделяемые всеми потоками, должны быть статическими или храниться в глобальной памяти, объявленной с использованием спецификатора volatile, а также должны быть защищены с использованием описанных ниже механизмов синхронизации.

Объекты синхронизации обсуждаются в следующем разделе. Приведенных в нем объяснений вам будет достаточно для того, чтобы разработать простой пример системы "производитель/потребитель" (producer/consumer).

Объекты синхронизации потоков

До сих пор нами были обсуждены только два механизма, обеспечивающие синхронизацию процессов и потоков друг с другом:

1. Поток, выполняющийся в контексте одного процесса, может дожидаться завершения другого процесса с использованием функции ExitProcess путем применения к дескриптору процесса функций ожидания WaitForSingleObject или WaitForMultipleObject. Тем же способом поток может организовать ожидание завершения (с помощью функции ExitThread или выполнения оператора return) другого потока.

2. Блокировки файлов, предназначенные для частного случая синхронизации доступа к файлам.

Windows предоставляет четыре других объекта, предназначенных для синхронизации потоков и процессов. Три из них — мьютексы, семафоры и события — являются объектами ядра, имеющими дескрипторы. События используются также для других целей, например, для асинхронного ввода/вывода (глава 14).

Мы начнем обсуждение с четвертого объекта, а именно, объекта критического участка кода CRITICAL_SECTION. В силу своей простоты и предоставляемых ими преимуществ в отношении производительности объекты критических участков кода являются предпочтительным механизмом, если их возможностей достаточно для того, чтобы удовлетворить требования программиста.

В то же время, при этом возникают некоторые проблемы, связанные с производительностью, о чем говорится в главе 9.

> **Предостережение**
>
> Неправильное применение объектов критических участков кода порождает определенные риски. Эти риски, такие, например, как риск блокировки, описываются в этой и последующих главах наряду с изложением методик, предназначенных для разработки надежного кода. Однако прежде всего мы приведем некоторые примеры синхронизации в реалистических ситуациях.

Рассмотрение двух других объектов синхронизации — таймеров ожидания и портов завершения ввода/вывода — отложено до главы 14. Эти типы объектов требуют использования методик асинхронного ввода/вывода Windows, которые описываются в указанной главе.

Объекты критических участковкода

Как уже упоминалось ранее, объект критического участка кода — это участок программного кода, который каждый раз должен выполняться только одним потоком; параллельное выполнение этого участка несколькими потоками может приводить к непредсказуемым или неверным результатам.

В качестве простого механизма реализации и применения на практике концепции критических участков кода Windows предоставляет объект CRITICAL_SECTION.

Объекты CRITICAL_SECTION (CS) можно инициализировать и удалять, но они не имеют дескрипторов и не могут совместно использоваться другими процессами. Соответствующие переменные должны объявляться как переменные типа CRITICAL_SECTION. Потоки входят в объекты CS и покидают их, но выполнение кода отдельного объекта CS каждый раз разрешено только одному потоку. Вместе с тем, один и тот же поток может входить в несколько отдельных объектов CS и покидать их, если они расположены в разных местах программы.

Для инициализации и удаления переменной типа CRITICAL_SECTION используются, соответственно, функции InitializeCriticalSection и DeleteCriticalSection:

```
VOID InitializeCriticalSection(LPCRITICAL_SECTION lpCriticalSection)

VOID DeleteCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
```

Функция EnterCriticalSection блокирует поток, если на данном критическом участке кода присутствует другой поток. Ожидающий поток разблокируется после того, как другой поток выполнит функцию LeaveCriticalSection. Говорят, что поток *получил права владения* объектом CS, если произошел возврат из функции EnterCriticalSection, тогда как для уступки прав владения используется функция LeaveCriticalSection. *Всегда следите за своевременной переуступкой прав владения объектами CS; несоблюдение этого правила может привести к тому, что другие потоки будут пребывать в состоянии ожидания в течение неопределенного времени даже после завершения выполнения потока-владельца.*

Мы часто будем говорить о *блокировании* и *разблокировании* объектов CS, а вхождение в CS будет означать то же, что и блокирование CS.

```
VOID EnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
```

Поток, владеющий объектом CS, может повторно войти в этот же CS без его блокирования; это означает, что объекты CRITICAL_SECTION являются *рекурсивными* (recursive). Поддерживается счетчик вхождений в объект CS, и поэтому поток должен покинуть данный CS столько раз, сколько было вхождений в него, чтобы разблокировать этот объект для других потоков. Эта возможность может оказаться полезной для реализации рекурсивных функций и обеспечения безопасного многопоточного выполнения функций общих (разделяемых) библиотек.

Выход из объекта CS, которым данный поток не владеет, может привести к непредсказуемым результатам, включая блокирование самого потока.

Для возврата из функции EnterCriticalSection не существует конечного интервала ожидания; другие потоки будут блокированы на неопределенное время, пока поток, владеющий объектом CS, не покинет его. Однако, используя функцию TryEnterCriticalSection, можно тестировать

(опросить) CS, чтобы проверить, не владеет ли им другой поток.

```
BOOL TryEnterCriticalSection(LPCRITICAL_SECTION lpCriticalSection)
```

Возврат функцией TryEnterCriticalSection значения True означает, что вызывающий поток приобрел права владения критическим участком кода, тогда как возврат значения False говорит о том, что данный критический участок кода уже принадлежит другого потока.

Объекты CRITICAL_SECTION обладают тем преимуществом, что они не являются объектами ядра и поддерживаются в пользовательском пространстве. Обычно, но не всегда, это приводит к дополнительному улучшению показателей производительности. К обсуждению аспектов производительности мы вернемся после того, как ознакомимся с объектами синхронизации, относящимися к ядру.

Настройка спин-счетчика

Обычно, если в результате выполнения функции EnterCriticalSection поток обнаруживает, что объект CS уже принадлежит другому потоку, он входит в ядро и остается блокированным до тех пор, пока не освободится объект CRITICAL_SECTION, что требует определенного времени. Однако в SMP-системах вы можете потребовать, чтобы поток повторил попытку завладеть объектом CS, прежде чем блокироваться, поскольку существует вероятность того, что поток, владеющий CS, выполняется на другом процессоре и в любой момент может освободить CS. Это может оказаться полезным для повышения производительности, если между потоками наблюдается высокая состязательность за право владения единственным объектом CRITICAL_SECTION. Влияние упомянутых факторов на производительность обсуждается далее в этой и последующих главах.

Для настройки счетчика занятости, или спин-счетчика (spin-count), предназначены две функции, одна из которых, SetCriticalSectionSpinCount, обеспечивает динамическую настройку счетчика, а вторая, InitializeCritical-SectionAndSpinCount, выступает в качестве замены функции Initialize-CriticalSection. Настройка спин-счетчика рассматривается в главе 9.

Использование объектов CRITICAL_SECTION для защиты разделяемыхпеременных

Использование объектов CRITICAL_SECTION не вызывает сложностей, и одним из наиболее распространенных способов их применения является обеспечение доступа потоков к разделяемым глобальным переменным. Рассмотрим, например, многопоточный сервер (аналогичный представленному на рис. 7.1), в котором необходимо вести учет следующих статистических данных:

• Общее количество полученных запросов.

• Общее количество отправленных ответов.

• Количество запросов, обрабатываемых в настоящее время всеми потоками сервера.

Поскольку переменные счетчиков являются глобальными переменными процесса, нельзя допустить того, чтобы одновременно два потока изменяли их значения. Один из методов обеспечения этого, базирующийся на применении объектов CRITICAL_SECTION, иллюстрирует схема, показанная ниже на рис. 8.2. Использование объектов CRITICAL_SECTION демонстрируется на примере программы 8.1, представляющей намного более простую систему, чем серверная.

Объекты CS могут привлекаться для решения задач, аналогичных той, которую иллюстрирует рис. 8.1, где два потока увеличивают значение одной и той же переменной. Приведенный ниже фрагмент кода обеспечивает нечто большее, нежели простое увеличение переменной, поскольку для этого достаточно было бы воспользоваться функциями взаимоблокировки. Обратите внимание на спецификатор volatile, предотвращающий размещение текущего значения переменной оптимизирующим компилятором в регистре, а не в ячейке памяти, отведенной для хранения переменной. Кроме того, в этом примере используется промежуточная переменная; этот необязательный элемент снижает эффективность программы, однако позволяет более отчетливо продемонстрировать, каким образом решается задача, иллюстрируемая рис. 8.1.

```
CRITICAL_SECTION cs1;
volatile DWORD N = 0, M;
/* N — глобальная переменная, разделяемая всеми потоками. */
InitializeCriticalSection (&cs1);
…
EnterCriticalSection (&cs1);
if (N < N_MAX) { M = N; M += 1; N = M; }
LeaveCriticalSection (&cs1);
…
DeleteCriticalSection (&cs1);
```

На рис. 8.2 представлена одна из возможных последовательностей выполнения программы для случая, изображенного на рис. 8.1, и продемонстрировано, каким образом объекты CS упрощают решение проблемы синхронизации.

Программа 8.1 демонстрирует, насколько полезными могут быть объекты CS.

Пример: простая система "производитель/потребитель"

Программа 8.1 иллюстрирует, насколько полезными могут быть объекты CS. Кроме того, эта программа демонстрирует, как создаются защищенные структуры данных для хранения состояний объектов, и знакомит с понятием *инварианта* (invariant) — свойства состояния объекта, относительно которого гарантируется (путем соответствующей реализации программы), что оно будет истинным за пределами критического участка кода.

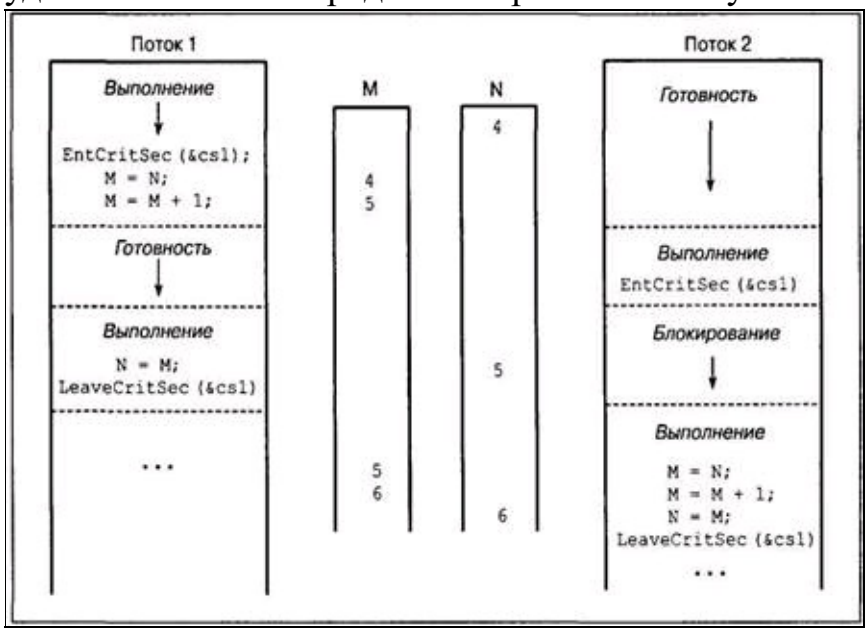


Рис. 8.2. Разделение общей памяти синхронизированными потоками

Описание задачи приводится ниже.

• Имеются два потока, *производитель* (producer) и *потребитель* (consumer), работающие в полностью асинхронном режиме.

• Производитель периодически создает сообщения, содержащие таблицу чисел, например, таблицу биржевых котировок, которая периодически обновляется.

• По требованию пользователя потребитель отображает текущие данные. Требуется, чтобы отображаемые данные представляли собой *самый последний полный набор данных, но никакие данные не должны отображаться дважды*.

• Данные не должны отображаться в те промежутки времени, когда они обновляются производителем; устаревшие данные также не должны отображаться. Обратите внимание на то, что многие сообщения вообще никогда не используются и, таким образом, "теряются". Этот пример является частным случаем конвейерной модели, в которой данные передаются из одного потока в другой.

• В качестве средства контроля целостности данных производитель вычисляет простую контрольную сумму[28] данных таблицы, которая далее сравнивается с аналогичной суммой, вычисленной потребителем, дабы удостовериться в том, что данные не были повреждены при их передаче из одного потока в другой. Данные, полученные при обращении к таблице в моменты ее обновления, будут недействительными; использование объектов CS гарантирует, что этого никогда не произойдет. Инвариантом блока сообщения (message block invariant) является корректность контрольной суммы для содержимого текущего сообщения.

• Обоими потоками поддерживается статистика суммарного количества отправленных, полученных и утерянных сообщений.

Программа 8.1.simplePC: простая система "производитель/потребитель"

```
    /* Глава 8. simplePC.c */
    /* Поддерживает два потока — производителя и потребителя. */
    /* Производитель периодически создает буферные данные с контрольными */
    /* суммами, или "блоки сообщений", отображаемые потребителем по запросу */
    /* пользователя. */

    #include "EvryThng.h"
    #include <time.h>
    #define DATA_SIZE 256

    typedef struct msg_block_tag { /* Блок сообщения. */
     volatile DWORD f_ready, f_stop; /* Флаги готовности и прекращения сообщений.
*/
     volatile DWORD sequence; /* Порядковый номер блока сообщения. */
     volatile DWORD nCons, nLost;
     time_t timestamp;
     CRITICAL_SECTION mguard; /* Структура защиты блока сообщения. */
     DWORD checksum; /* Контрольная сумма содержимого сообщения. */
     DWORD data[DATA_SIZE]; /* Содержимое сообщения. */
    } MSG_BLOCK;

    /* Одиночный блок, подготовленный к заполнению новым сообщением. */
    MSG_BLOCK mblock = { 0, 0, 0, 0, 0 };

    DWORD WINAPI produce(void*);
    DWORD WINAPI consume(void*);
    void MessageFill(MSG_BLOCK*);
    void MessageDisplay(MSG_BLOCK*);

    DWORD _tmain(DWORD argc, LPTSTR argv[]) {
     DWORD Status, ThId;
     HANDLE produce h, consume_h;
     /* Инициализировать критический участок блока сообщения. */
     InitializeCriticalSection (&mblock.mguard);
     /* Создать два потока. */
     produce_h = (HANDLE)_beginthreadex(NULL, 0, produce, NULL, 0, &ThId);
     consume_h = (HANDLE)_beginthreadex (NULL, 0, consume, NULL, 0, &ThId);
     /* Ожидать завершения потоков производителя и потребителя. */
     WaitForSingleObject(consume_h, INFINITE);
     WaitForSingleObject(produce_h, INFINITE);
     DeleteCriticalSection(&mblock.mguard);
     _tprintf(_T("Потоки производителя и потребителя завершили выполнение\n"));
      _tprintf(_T("Отправлено:  %d,  Получено:  %d,  Известные  потери:  %d\n"),
mblock.sequence, mblock.nCons, mblock.nLost);
     return 0;
    }

    DWORD WINAPI produce(void *arg)
    /* Поток производителя — создание новых сообщений через случайные */
    /* интервалы времени. */
    {
     srand((DWORD)time(NULL)); /* Создать начальное число для генератора случайных
чисел. */
     while (!mblock.f_stop) {
      /* Случайная задержка. */
```

```
      Sleep(rand() / 100);
      /* Получить и заполнить буфер. */
      EnterCriticalSection(&mblock.mguard);
      __try {
       if (!mblock.f_stop) {
        mblock.f_ready = 0;
        MessageFill(&mblock);
        mblock.f_ready = 1;
        mblock.sequence++;
       }
      } __finally { LeaveCriticalSection (&mblock.mguard); }
     }
     return 0;
    }

    DWORD WINAPI consume (void *arg) {
     DWORD ShutDown = 0;
     CHAR command, extra;
     /* Принять ОЧЕРЕДНОЕ сообщение по запросу пользователя. */
      while (!ShutDown) { /* Единственный поток, получающий доступ к стандартным
устройствам ввода/вывода. */
      _tprintf(_T("\n**Введите 'c' для приема; 's' для прекращения работы: "));
      _tscanf("%c%c", &command, &extra);
      if (command == 's') {
       EnterCriticalSection(&mblock.mguard);
       ShutDown = mblock.f_stop = 1;
       LeaveCriticalSection(&mblock.mguard);
       } else if (command == 'c') { /* Получить новый буфер для принимаемых
сообщений. */
       EnterCriticalSection(&mblock.mguard);
       __try {
            if (mblock.f_ready == 0) _tprintf(_T("Новые сообщения отсутствуют.
Повторите попытку.\n"));
        else {
         MessageDisplay(&mblock);
         mblock.nCons++;
         mblock.nLost = mblock.sequence - mblock.nCons;
         mblock.f_ready = 0; /* Новые сообщения отсутствуют. */
        }
       } __finally { LeaveCriticalSection (&mblock.mguard); }
      } else {
       tprintf(_T("Такая команда отсутствует. Повторите попытку.\n"));
      }
     }
     return 0;
    }

    void MessageFill(MSG_BLOCK *mblock) {
     /* Заполнить буфер сообщения содержимым, включая контрольную сумму и отметку
времени. */
     DWORD i;
     mblock->checksum = 0;
     for (i = 0; i < DATA_SIZE; i++) {
      mblock->data[i] = rand();
      mblock->checksum ^= mblock->data[i];
     }
     mblock->timestamp = time(NULL);
     return;
    }
```

```
    void MessageDisplay(MSG_BLOCK *mblock) {
     /* Отобразить буфер сообщения, отметку времени и контрольную сумму. */
     DWORD i, tcheck = 0;
     for (i = 0; i < DATA_SIZE; i++) tcheck ^= mblock->data[i];
      _tprintf(_T("\nВремя генерации сообщения № %d: %s"), mblock->sequence,
_tctime(&(mblock->timestamp)));
      _tprintf(_T("Первая и последняя записи: %x %x\n"), mblock->data[0], mblock-
>data[DATA_SIZE - 1]);
     if (tcheck == mblock->checksum) _tprintf(_T("УСПЕШНАЯ ОБРАБОТКА ->Контрольная
сумма совпадает.\n"));
       else tprintf(_T("СБОЙ ->Несовпадение контрольной суммы. Сообщение
запорчено.\n"));
     return;
    }
```

Комментарии к примеру простой системы "производитель/потребитель"

Этот пример иллюстрирует некоторые моменты и соглашения, касающиеся программирования, которые будут важны для нас на протяжении этой и последующих глав.

• Объект CRITICAL_SECTION является частью объекта (блока сообщения), защиту которого он обеспечивает.

• Каждый доступ к сообщению осуществляется на критическом участке кода.

• Типом переменных, доступ к которым осуществляется разными потоками, является volatile.

• Использование обработчиков завершения гарантирует, что объекты CS будут обязательно освобождены. Хотя в данном случае эта методика и не является для нас существенной, она дополнительно гарантирует, что вызов функции LeaveCriticalSection не будет случайно опущен впоследствии при изменении кода программы. Имейте также в виду, что обработчик завершения ограничен использованием средств C, и его не следует использовать совместно с C++.

• Функции MessageFill и MessageDisplay вызываются лишь на критических участках кода и используют для нужд своих вычислений не глобальную, а локальную память. Кстати, обе они будут применяться и в последующих примерах, но их листинги больше приводиться не будут.

• Не существует удобного способа, при помощи которого поток производителя мог бы известить поток потребителя о наличии нового сообщения, и поэтому поток потребителя должен просто ожидать, пока не будет установлен флаг готовности, который используется для индикации появления нового сообщения. Устранить этот недостаток нам помогут объекты событий ядра.

• Одним из инвариантных свойств, которые гарантируются этой программой, является то, что контрольная сумма блока сообщения будет всегда корректной *вне* критических участков кода. Другим инвариантным свойством является следующее:

```
0 <= nLost + nCons <= sequence
```

Об этом важном свойстве далее еще будет идти речь.

• О необходимости прекращения передачи поток производителя узнает лишь после проверки флага, устанавливаемого в блоке сообщения потока потребителя. Поскольку потоки не могут обмениваться между собой никакими сигналами, а вызов функции TerminateThread чреват нежелательными побочными эффектами, эта методика является простейшим способом остановки другого потока. Разумеется, чтобы эта методика была эффективной, работа потоков должна быть скоординированной. В то же время, подобное решение требует, чтобы поток не блокировался, иначе он не сможет тестировать флаг; способы решения проблемы

блокированных потоков обсуждаются в главе 10.

Объекты CRITICAL_SECTION предоставляют в наше распоряжение мощный механизм синхронизации, но, тем не менее, они не в состоянии обеспечить всю полноту необходимых функциональных возможностей. О невозможности отправки сигналов одним потоком другому уже говорилось, кроме того, эти объекты не позволяют воспользоваться конечными интервалами ожидания (time-out). Объекты синхронизации ядра Windows позволяют снизить остроту не только этих, но и других ограничений.

Мьютексы

Объект *взаимного исключения* (mutual exception), или *мьютекс* (mutex), обеспечивает более универсальную функциональность по сравнению с объектом CRITICAL_SECTION. Поскольку мьютексы могут иметь имена и дескрипторы, их можно использовать также для синхронизации потоков, принадлежащих различным процессам. Так, два процесса, разделяющие общую память посредством отображения файлов, могут использовать мьютексы для синхронизации доступа к разделяемым областям памяти.

Объекты мьютексов аналогичны объектам CS, однако, дополнительно к возможности их совместного использования различными процессами, они допускают конечные периоды ожидания, а мьютексы, *покинутые* (abandoned) завершающимся процессом, переходят в сигнальное состояние.[29] Поток приобретает права владения мьютексом (или *блокирует* (block) мьютекс) путем вызова функции ожидания (WaitForSingleObject или WaitForMultipleObjects) по отношению к дескриптору мьютекса и уступает эти права посредством вызова функции ReleaseMutex.

Как всегда, необходимо тщательно следить за тем, чтобы потоки своевременно освобождали ресурсы, в которых они больше не нуждаются. Поток может завладевать одним и тем же ресурсом несколько раз, и при этом не будет блокироваться даже в тех случаях, когда уже владеет данным ресурсом. В конечном счете, поток должен освободить мьютекс столько раз, сколько она его захватывала. Такая возможность рекурсивного захвата ресурсов, существующая и в случае объектов CS, может оказаться полезной для ограничения доступа к рекурсивным функциям, а также в приложениях, реализующих вложенные транзакции (nested transactions).

При работе с мьютексами мы будем пользоваться функциями CreateMutex, ReleaseMutex и OpenMutex.

```
    HANDLE CreateMutex(LPSECURITY_ATTRIBUTES lpsa, BOOL bInitialOwner,
LPCTSTR lpMutexName)
    BOOL ReleaseMutex(HANDLE hMutex)
```

bInitialOwner — если значение этого флага установлено равным True, вызывающий поток немедленно приобретает права владения новым мьютексом. Эта атомарная операция позволяет предотвратить приобретение прав владения мьютексом другими потоками, прежде чем это сделает поток, создающий мьютекс. Как следует из самого его названия (initial owner — исходный владелец), этот флаг не оказывает никакого действия, если мьютекс уже существует.

lpMutexName — указатель на строку, содержащую имя мьютекса; в отличие от файлов имена мьютексов чувствительны к регистру. Если этот параметр равен NULL, то мьютекс создается без имени. События, мьютексы, семафоры, отображения файлов и другие объекты ядра, упоминаемые в данной книге, — все они используют одно и то же пространство имен, отличное от пространства имен файловой системы. Поэтому имена всех объектов синхронизации должны быть различными. Длина указанных имен не может превышать 260 символов.

Возвращаемое значение имеет тип HANDLE; значение NULL указывает на неудачное завершение функции.

Функция OpenMutex открывает существующий именованный мьютекс. Впоследствии эта функция не обсуждается, но используется в некоторых примерах. Эта функция дает возможность потокам, принадлежащим различным процессам, синхронизироваться так, как если бы они принадлежали одному и тому же процессу. Вызову функции OpenMutex в одном процессе

должен предшествовать вызов функции CreateMutex в другом процессе. Для семафоров и событий, как и для отображенных файлов (глава 5), также имеются соответствующие функции Create и Open. При вызове этих функций всегда предполагается, что сначала один процесс, например сервер, вызывает функцию Create для создания именованного объекта, а затем другие процессы вызывают функцию Open, которая завершается неудачей, если именованный объект к этому моменту еще не был создан. Возможен и такой вариант, когда все процессы самостоятельно используют вызов функции Create с одним и тем же именем, если порядок создания объектов не имеет значения.

Функция ReleaseMutex освобождает мьютекс, которым владеет вызывающий поток. Если мьютекс не принадлежит потоку, функция завершается с ошибкой.

```
BOOL ReleaseMutex(HANDLE hMutex)
```

Спецификация POSIX Pthreads поддерживает мьютексы. Имеются следующие основные функции:

- `pthread_mutex_init`
- `pthread_mutex_destroy`
- `pthread_mutex_lock`
- `pthread_mutex_unlock`

Функция pthread_mutex_lock является блокирующей и поэтому эквивалентна функции WaitForSingleObject в случае ее применения к дескриптору мьютекса. Функция pthread_mutex_trylock осуществляет опрос и не является блокирующей, соответствуя функции WaitForSingleObject в случае ее применения с нулевым значением интервала ожидания. Потоки Pthreads не поддерживают конечные интервалы ожидания и не предлагают средств, аналогичных Windows-объектам CRITICAL_SECTION.

Покинутые мьютексы

Мьютекс, владевший которым поток завершился, не освободив его, называют *покинутым* (abandoned), и его дескриптор переходит в сигнальное состояние. На то, что сигнализирующий дескриптор (дескрипторы) представляет покинутый мьютекс (мьютексы), указывает возврат функцией WaitForSingleObject значения WAIT_ABANDONED_0 или использование значения WAIT_ABANDONED_0 в качестве базового значения функцией WaitForMultipleObject.

То, что дескрипторы покинутых мьютексов переходят в сигнальное состояние, является весьма полезным их свойством, недоступным в случае объектов CS. Обнаружение покинутого мьютекса может означать наличие дефекта в коде, организующем работу потоков, поскольку потоки должны программироваться таким образом, чтобы ресурсы всегда освобождались, прежде чем поток завершит свое выполнение. Возможно также, что выполнение данного потока было прервано другим потоком.

Мьютексы, критические участки кода и взаимоблокировки

Несмотря на то что объекты CS и мьютексы обеспечивают решение задач, подобных той, которая иллюстрируется на рис. 8.1, при их использовании следует соблюдать осторожность, иначе можно создать ситуацию *взаимоблокировки* (deadlock), в которой каждый из двух потоков ждет освобождения ресурсов, принадлежащих другому потоку.

Взаимоблокировки являются одним из наиболее распространенных и коварных дефектов

синхронизации и часто возникают, когда должны быть одновременно блокированы (lock) два и более мьютекса. Рассмотрим следующую задачу:

• Имеется два связных списка, список A и список B, каждый из которых содержит идентичные структуры и поддерживается рабочими потоками.

• Для одного класса элементов списка корректность операции зависит от того факта, что данный элемент X находится или отсутствует одновременно в обоих списках. Здесь мы имеем дело с инвариантом, который неформально можно выразить так: "X либо находится в обоих списках, либо не находится ни в одном из них".

• В других ситуациях допускается нахождение элемента только в одном из списков, но не в обоих одновременно. *Мотивация.* Указанными списками могут быть списки сотрудников отделов A и B, когда некоторым сотрудникам разрешена работа одновременно в двух отделах.

• В связи с вышеизложенным для обоих списков требуются различные мьютексы (объекты CS), но при добавлении или удалении общих элементов списков блокироваться должны одновременно оба мьютекса. Использование только одного мьютекса оказало бы отрицательное влияние на производительность, препятствуя независимому параллельному обновлению двух списков, поскольку мьютекс оказался бы "слишком большим".

Ниже приведен пример возможной реализации функций рабочего потока, предназначенных для добавления и удаления общих элементов списков:

```
    static struct {
     /* Инвариант: действительность списка. */
     HANDLE guard; /* Дескриптор мьютекса. */
     struct ListStuff;
     } ListA, ListB;
    …
    DWORD WINAPI AddSharedElement(void *arg) /* Добавляет общий элемент в списки A
и B. */
    { /* Инвариант: новый элемент либо находится в обоих списках, либо не находится
ни в одном из них. */
     WaitForSingleObject(ListA.guard, INFINITE);
     WaitForSingleObject(ListB.guard, INFINITE);
     /* Добавить элемент в оба списка … */
     ReleaseMutex(ListB.guard);
     ReleaseMutex(ListA.guard);
     return 0;
    }
    DWORD WINAPI DeleteSharedElement(void *arg) /* Удаляет общий элемент из списков
A и B. */
    {
     WaitForSingleObject(ListB.guard, INFINITE);
     WaitForSingleObject(ListA.guard, INFINITE);
     /* Удалить элемент из обоих списков … */
     ReleaseMutex(ListB.guard);
     ReleaseMutex(ListA.guard);
     return 0;
    }
```

С учетом ранее данных рекомендаций этот код выглядит вполне корректным. Однако вытеснение потока AddSharedElement сразу же после того, как он блокирует список A, и непосредственно перед тем, как он попытается заблокировать список B, приведет к взаимоблокировке потоков, если поток DeleteSharedElement начнет выполняться до того, как возобновится выполнение потока AddSharedElement. Каждый из потоков владеет мьютексом, который необходим другому потоку, и ни один из потоков не может перейти к вызову функции ReleaseMutex, который разблокировал бы другой поток.

Обратите внимание, что взаимоблокировка по сути дела является еще одной разновидностью состязаний, поскольку каждый из потоков состязается с другим за право первым овладеть всеми своими мьютексами.

Один из способов, позволяющих избежать взаимоблокировки, заключается в применении метода "проб и ошибок", когда поток вызывает функцию WaitForSingleObject с конечным интервалом ожидания, и если оказывается, что мьютекс уже принадлежит другому потоку, то первый поток уступает процессор или "засыпает" на короткое время, а затем вновь повторяет попытку. Намного лучше и эффективнее с самого начала проектировать программу таким образом, чтобы исключить саму возможность возникновения взаимоблокировок, о чем говорится ниже.

Гораздо более простой метод, который описывается почти в любом учебнике по ОС, заключается в предварительном определении "иерархии мьютексов" и программировании потоков таким образом, чтобы захват ими мьютексов осуществлялся в строгом соответствии с заданным иерархическим порядком, а освобождение — в обратном порядке. Эта иерархия может устанавливаться произвольно или естественным образом определяться структурой самой задачи, но в любом случае ее должны придерживаться все потоки. В данном примере лишь требуется, чтобы функция удаления мьютекса поочередно ожидала освобождения списков A и B, и тогда взаимоблокировка потоков никогда не случится, если указанная иерархическая очередность будет соблюдаться всеми потоками в любом месте программы.

Еще одним действенным методом снижения риска взаимоблокировки является размещение двух дескрипторов мьютексов в массиве и использование функции WaitForMultipleObjects с флагом fWaitAll, значение которого установлено равным True, вследствие чего поток в результате выполнения одной атомарной операции будет захватывать либо оба мьютекса, либо ни одного. В случае использования объектов CRITICAL_SECTION описанная методика неприменима.

Сравнительный обзор: мьютексы и объекты CRITICAL_SECTION

Как уже неоднократно упоминалось, мьютексы и объекты CRITICAL_SECTION весьма напоминают друг друга и предназначены для решения одного и того же круга задач. В частности, объекты обоих типов могут принадлежать только одного потока, и если объектом, которым уже владеет какой-либо поток, пытаются завладеть другие потоки, то они будут блокированы до тех пор, пока объект не освободится. Мьютексы могут обеспечивать большую гибкость, однако достигается это лишь за счет ухудшения производительности. В заключение перечислим наиболее важные отличия, существующие между указанными двумя типами объектов синхронизации.

• Мьютексы, покинутые завершающимися потоками, переходят в сигнальное состояние, в результате чего другие потоки не будут блокироваться на неопределенное время.

• Имеется возможность организовать ожидание мьютекса с использованием конечного интервала ожидания, тогда как в случае объектов CS возможен только опрос их состояния.

• Мьютексам можно присваивать имена, и их могут совместно использовать потоки, принадлежащие разным процессам.

• К мьютексам применима функция WaitForMultipleObjects, что не только удобно с точки зрения программирования, но и позволяет избежать взаимоблокировки потоков при надлежащей организации программы.

• Поток, создающий мьютекс, может сразу же указать, что он становится его владельцем. В случае объектов CS за право владения объектом могут состязаться несколько потоков.

• Обычно, хотя и не всегда, использование объектов CS обеспечивает более высокую производительность по сравнению с той, которая достигается при использовании мьютексов. Этот вопрос более подробно обсуждается в главе 9.

Синхронизация куч

В NT для синхронизации доступа к кучам (глава 5) предусмотрены две функции — HeapLock и HeapUnlock. В каждой из этих функций единственным аргументом является дескриптор. Эти функции удобно применять в тех случаях, когда используется флаг HEAP_NO_SERIALIZE, или когда потоку необходимы права исключительного доступа к куче.

Семафоры

Объекты второго из трех упомянутых в начале главы типов объектов синхронизации ядра — *семафоры* (semaphores), поддерживают счетчики, и когда значение этого счетчика больше 0, объект семафора находится в сигнальном состоянии. Если же значение счетчика становится нулевым, объект семафора переходит в несигнальное состояние.

Потоки и процессы организуют ожидание обычным способом, используя для этого одну или несколько функций ожидания. При разблокировании ожидающего потока значение счетчика уменьшается на 1.

К функциям управления семафорами относятся CreateSemaphore, OpenSemaphore и ReleaseSemaphore, причем последняя функция может инкрементировать значение счетчика на 1 и более. Эти функции аналогичны своим эквивалентам, предназначенным для управления мьютексами.

```
HANDLE CreateSemaphore(LPSECURITY_ATTRIBUTES lpsa, LONG lSemInitial,
LONG lSemMax, LPCTSTR lpSemName)
```

Параметр lSemMax, значение которого должно быть равным, по крайней мере, 1, определяет максимально допустимое значение счетчика семафора. Параметр lSemInitial — начальное значение этого счетчика, которое должно удовлетворять следующему условию: 0≤ lSemInitial≤ lSemMax и никогда не должно выходить за пределы указанного диапазона. Возвращение функцией значения NULL указывает на ее неудачное выполнение.

Каждая отдельная операция ожидания может уменьшить значение счетчика только на 1, но с помощью функции ReleaseSemaphore значение его счетчика может быть увеличено до любого значения вплоть до максимально допустимого.

```
BOOL ReleaseSemaphore(HANDLE hSemaphore, LONG cReleaseCount, LPLONG
lpPreviousCount)
```

Обратите внимание на возможность получения предыдущего значения счетчика, определяемого указателем lpPreviousCount, которое он имел до освобождения объекта синхронизации при помощи функции ReleaseSemaphore, но если необходимости в этом нет, то значение упомянутого указателя следует установить равным NULL.

Число, прибавляемое к счетчику семафора (cReleaseCount), должно быть больше 0, но если выполнение функции ReleaseSemaphore приводит к выходу значения счетчика за пределы допустимого диапазона, то она завершается с ошибкой, возвращая значение FALSE, а значение счетчика семафора остается неизменным. Предыдущим значением счетчика следует пользоваться с осторожностью, поскольку оно могло быть изменено другими потоками. Кроме того, невозможно определить, достиг ли счетчик максимально допустимого значения, поскольку не предусмотрено средство, отслеживающее увеличение счетчика в результате его освобождения. Пример использования предыдущего значения счетчика семафора приведен на Web-сайте книги.

Как ни соблазнительно пытаться рассматривать мьютекс как частный случай семафора, значение счетчика которого задано равным 1, это было бы заблуждением ввиду отсутствия понятия прав владения семафором. Семафор может быть освобожден любым потоком, а не только тем, который ожидает. Точно так же, поскольку нельзя говорить о правах владения семафором, отсутствует и понятие покинутого семафора.

Использование семафоров

Классической областью применения семафоров является управление распределением конечных ресурсов, когда значение счетчика семафора ассоциируется с определенным количеством доступных ресурсов, например, количеством сообщений, находящихся в очереди. Тогда максимальное значение счетчика соответствует максимальному размеру очереди. Таким образом, производитель помещает сообщение в буфер и вызывает функцию ReleaseSemaphore, обычно с увеличением значения счетчика на 1 (cReleaseCount). Потоки потребителя будут ожидать перехода семафора в сигнальное состояние, получая сообщения и уменьшая значения счетчика.

Вслед за рассмотрением программы 9.1 обсуждается другой важный случай применения семафоров, когда они используются для ограничения количества рабочих потоков, фактически выполняющихся в любой момент времени, что позволяет снизить состязательность между ними, а в некоторых случаях — повысить производительность. Эта методика, в которой используются дроссели семафоров (semaphore throttles), обсуждается в главе 9.

Опасность возникновения условий состязаний в программе sortMT (программа 7.2) иллюстрирует другое возможное применение семафоров, связанное с управлением точным количеством потоков, которые должны находиться в пробужденном состоянии. Можно создать все потоки, не приостанавливая их. После этого все они сразу же переходят к ожиданию перехода в сигнальное состояние семафора, инициализированного значением 0. Далее, главный поток вместо того, чтобы освобождать потоки, просто вызывает функцию ReleaseCount с увеличением счетчика, например, на 4 (или на любое другое значение, соответствующее количеству потоков), в результате чего возможность выполняться получат четыре потока.

Несмотря на все удобства, которые сулит использование семафоров, они являются в некотором смысле излишними в том смысле, что мьютексы и события (описанные в одном из следующих разделов), при условии их совместного использования, предлагают гораздо более широкие возможности, чем семафоры. Более подробная информация по этому поводу содержится в главе 10.

Ограниченность семафоров

В Windows существуют важные ограничения, касающиеся реализации семафоров. Например, каким образом поток может потребовать, чтобы счетчик семафора уменьшился на 2? Для этого поток мог бы организовать ожидание два раза подряд, как показано ниже, но эта операция не была бы атомарной, поскольку в промежутке между двумя вызовами функции ожидания данный поток может быть вытеснен. В результате этого, как описывается ниже, может наступить взаимоблокировка (deadlock) потоков.

```
    /* hsem – дескриптор семафора. Максимальное значение счетчика семафора равно 2.
*/
    …
    /* Уменьшить значение счетчика семафора на 2. */
    WaitForSingleObject(hSem, INFINITE);
    WaitForSingleObject(hSem, INFINITE);
    …
    /* Увеличить значение счетчика семафора на 2. */
    ReleaseSemaphore(hSem, 2, &PrevCount);
```

Чтобы увидеть, каким образом в подобной ситуации может возникнуть взаимоблокировка, предположим, что максимальное и начальное значения счетчика устанавливаются равными 2 и

что первый из двух потоков завершает первый цикл ожидания, а затем вытесняется. Далее второй поток может завершить первый цикл ожидания и уменьшить значение счетчика до 0. Оба потока окажутся блокированными на неопределенное время, поскольку ни одна из них не сможет выполнить второй цикл ожидания. Такая простая ситуация взаимоблокировки является довольно типичной.

Один из возможных вариантов правильного решения заключается в том, чтобы защитить циклы ожидания при помощи мьютекса или объекта CRITI-CAL_SECTION, как показано в приведенном ниже фрагменте программного кода:

```
/* Уменьшаем значение счетчика семафора на 2. */
EnterCriticalSection(&csSem);
WaitForSingleObject(hSem, INFINITE);
WaitForSingleObject(hSem, INFINITE);
LeaveCriticalSection (&csSem);
…
ReleaseSemaphore(hSem, 2, &PrevCount);
```

Но и эта реализация, в таком общем виде, страдает ограничениями. Предположим, например, что в счетчике семафора остается две единицы, и потоку A необходимы три единицы, а потоку B — только две. Если первой начнет выполняться поток A, то он выполнит два цикла ожидания и блокируется на третьем, продолжая владеть мьютексом. При этом поток B, которому были необходимы только две единицы, по-прежнему будет оставаться блокированным.

Казалось бы, можно воспользоваться функцией WaitForMultipleObjects с использованием одного и того же дескриптора семафора в нескольких элементах массива дескрипторов. Однако такое предложение было бы неудачным по двум причинам. Прежде всего, обнаружив, что два дескриптора указывают на один и тот же объект, функция WaitForMultipleObjects завершится с ошибкой. Более того, даже если значение счетчика семафора будет составлять только 1, сигнализироваться будут все дескрипторы, что противоречит самой исходной цели.

Полное решение проблемы множественных циклов ожидания предлагается в упражнении 10.11.

Проектировать семафоры Windows было бы гораздо удобнее, если бы существовала возможность выполнять множественные циклы ожидания в виде одной атомарной операции (atomic multiple-wait operation).

События

Последним из рассматриваемых нами типов объектов синхронизации ядра являются *события* (events). Объекты события используются для того, чтобы сигнализировать другим потокам о наступлении какого-либо события, например, о появлении нового сообщения.

Важной дополнительной возможностью, обеспечиваемой объектами событий, является то, что переход в сигнальное состояние единственного объекта события способен вывести из состояния ожидания одновременно несколько потоков. Объекты события делятся на сбрасываемые вручную и автоматически сбрасываемые, и это их свойство устанавливается при вызове функции CreateEvent.

• Сбрасываемые вручную события (manual-reset events) могут сигнализировать одновременно всем потокам, ожидающим наступления этого события, и переводятся в несигнальное состояние программно.

• Автоматически сбрасываемые события (auto-reset event) сбрасываются самостоятельно после освобождения одного из ожидающих потоков, тогда как другие ожидающие потоки продолжают ожидать перехода события в сигнальное состояние.

События используют пять новых функций: CreateEvent, OpenEvent, SetEvent, ResetEvent и CreateEvent.

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpsa, BOOL bManualReset, BOOL
bInitialState, LPTCSTR lpEventName)
```

Чтобы создать событие, сбрасываемое вручную, необходимо установить значение параметра bManualReset равным True. Точно так же, чтобы сделать начальное состояние события сигнальным, установите равным True значение параметра bInitialState. Для открытия именованного объекта события используется функция OpenEvent, причем это может сделать и другой процесс.

Для управления объектами событий используются следующие три функции:

```
BOOL SetEvent(HANDLE hEvent)
BOOL ResetEvent(HANDLE hEvent)
BOOL PulseEvent(HANDLE hEvent)
```

Поток может установить событие в сигнальное состояние, используя функцию SetEvent. Если событие является автоматически сбрасываемым, то оно автоматически возвращается в несигнальное состояние уже после освобождения только одного из ожидающих потоков. В отсутствие потоков, ожидающих наступления этого события, оно остается в сигнальном состоянии до тех пор, пока такой поток не появится, после чего этот поток сразу же освобождается. Заметьте, что таким же образом ведет себя семафор, максимальное значение счетчика которого установлено равным 1.

С другой стороны, если событие является сбрасываемым вручную, то оно остается в сигнальном состоянии до тех пор, пока какой-либо поток не вызовет функцию ResetEvent, указав дескриптор этого события в качестве аргумента. В это время все ожидающие потоки освобождаются, но до выполнения такого сброса события другие потоки могут как переходить в состояние его ожидания, так и освобождаться.

Функция PulseEvent освобождает все потоки, ожидающие наступления сбрасываемого вручную события, но после этого событие сразу же автоматически сбрасывается. В случае же

использования автоматически сбрасываемого события функция PulseEvent освобождает только один ожидающий поток, если таковые имеются.

Примечание

Хотя в книгах многих авторов и даже в некоторых документах Microsoft (см. примечания в разделе MSDN, содержащем описание функции PulseEvent) рекомендуется избегать использования функции PulseEvent, лично я считаю эту функцию не только полезной, но и существенно важной, как это следует из обсуждения многочисленных примеров, приведенных в двух следующих главах.

Следует отметить, что функция PulseEvent становится полезной лишь после того, как сбрасываемое вручную событие установлено в сигнальное состояние с помощью функции SetEvent. Будьте внимательны, когда используете функцию WaitForMultipleObjects для ожидания перехода в сигнальное состояние *всех* событий. Ожидающий поток освободится только тогда, когда одновременно все события будут находиться в сигнальном состоянии, и некоторые из событий, находящихся в сигнальном состоянии, могут быть сброшены, прежде чем поток освободится.

В упражнении 8.5 вам предлагается изменить программу sortMT (программа 7.2) за счет использования в ней событий.

Переменные условий (condition variables) Pthreads в некоторой степени сравнимы с событиями, но используются в сочетании с мьютексами. Такой способ их использования в действительности является очень плодотворным и будет описан в главе 10. Для создания и уничтожения переменной условий используются, соответственно, системные вызовы pthread_cond_init и pthread_cond_destroy. Функциями ожидания являются pthread_cond_wait и pthread_cond_timedwait. Системный вызов pthread_cond_signal осуществляет возврат после освобождения одного ожидающего потока аналогично Windows-функции PulseEvent в случае автоматически сбрасываемых событий, тогда как вызов pthread_cond_broadcast сигнализирует всем ожидающим потокам, и поэтому его можно сопоставить функции PulseEvent, применяемой к сбрасываемому вручную событию. Эквивалентов функций PulseEvent и ResetEvent, используемых в случае сбрасываемых вручную событий, не существует.

Обзор: четыре модели использования событий

Комбинирование автоматически сбрасываемых и сбрасываемых вручную событий с функциями SetEvent и PulseEvent приводит к четырем различным способам использования событий. Каждая из четырех комбинаций уникальна и каждая из них оказывается полезной или даже необходимой в той или иной ситуации, так что все они будет соответствующим образом использованы в примерах и упражнениях, приведенных в этой и следующей главах.

Предостережение

Некорректное использование событий может привести к возникновению условий состязаний, взаимоблокировок и других тонких, трудно обнаруживаемых ошибок. В главе 10 описываются методики, применение которых является обязательным почти во всех случаях использования событий, за исключением самых тривиальных.

В табл. 8.1 описаны четыре возможные ситуации.

Таблица 8.1. Сводная таблица свойств событий

	Автоматически сбрасываемые события	**Сбрасываемые вручную события**
SetEvent	Освобождается строго один поток. Если в этот момент ни один из потоков не ожидает наступления события, то поток, который первым перейдет в состояние ожидания следующих событий, будет сразу же освобожден. После этого событие немедленно автоматически сбрасывается.	Освобождаются все потоки, которые в настоящее время ожидают наступления события. Событие остается в сигнальном состоянии до тех пор, пока не будет сброшено каким-либо потоком.
PulseEvent	Освобождается строго один поток, но только в том случае, если имеется поток, ожидающий наступления события.	Освобождаются все потоки, которые в этот момент ожидают наступления события, если таковые имеются, после чего событие сбрасывается и переходит в несигнальное состояние.

Образно говоря, автоматически сбрасываемое событие — это дверь, снабженная пружиной, которая обеспечивает автоматическое закрытие двери, в то время как вручную сбрасываемое событие можно уподобить двери, в которой пружина отсутствует и которая, будучи раз открытой, продолжает оставаться в таком состоянии. Используя эту метафору, можно сказать, что функция PulseEvent открывает дверь и закрывает ее сразу же после того, как через нее проходят одна (автоматически сбрасываемые события) или все (вручную сбрасываемые события) ожидающие потоки. Функция SetEvent открывает дверь и освобождает ее.

Пример: система "производитель/потребитель"

В этом примере возможности программы 8.1 расширяются таким образом, чтобы потребитель мог дожидаться момента, когда появится доступное сообщение. Тем самым устраняется одна из проблем, связанная с тем, что в предыдущем варианте программы потребитель должен был непрерывно повторять попытки получения новых сообщений. Результирующая программа (программа 8.2) называется eventPC.

Заметьте, что в предлагаемом решении вместо объектов CRITICAL_SECTION используются мьютексы; единственной причиной для этого послужило лишь желание проиллюстрировать применение мьютексов. В то же время, использование автоматически сбрасываемого события и функции SetEvent в потоке потребителя является весьма существенным для работы программы, поскольку это гарантирует освобождение только одного потока.

Также обратите внимание на способ связывания мьютекса и события со структурой данных блока сообщения. Мьютекс активизирует критический участок кода для доступа к объекту структуры данных, тогда как событие используется для уведомления о том, что появилось новое сообщение. Обобщая, можно сказать, что мьютекс гарантирует сохранение инвариантов объекта, а событие сигнализирует о нахождении объекта в заданном состоянии. Эта базовая методика широко применяется в последующих главах.

Программа 8.2. eventPC: система "производитель/потребитель", использующая сигналы

```
    /* Глава 8. eventPC.c */
    /* Поддерживает два потока — производителя и потребителя. */
    /* Производитель периодически создает буферные данные с контрольными */
    /* суммами, или "блоки сообщений", сигнализирующие потребителю о готовности*/
    /* сообщения. Поток потребителя отображает информацию в ответ на запрос.*/

    #include "EvryThng.h"
    #include <time.h>
    #define DATA_SIZE 256

    typedef struct msg_block_tag { /* Блок сообщения. */
     volatile DWORD f_ready, f_stop; /* Флаги готовности и прекращения сообщений.
*/
     volatile DWORD sequence; /* Порядковый номер блока сообщения. */
     volatile DWORD nCons, nLost; time_t timestamp;
     HANDLE mguard; /* Мьютекс, защищающий структуру блока сообщения. */
     HANDLE mready; /* Событие "Сообщение готово". */
     DWORD checksum; /* Контрольная сумма сообщения. */
     DWORD data[DATA_SIZE]; /* Содержимое сообщения. */
    } MSG_BLOCK;

    /* … */

    DWORD _tmain(DWORD argc, LPTSTR argv[]) {
     DWORD Status, ThId;
     HANDLE produce_h, consume_h;
      /* Инициализировать мьютекс и событие (автоматически сбрасываемое) в блоке
сообщения. */
     mblock.mguard = CreateMutex(NULL, FALSE, NULL);
     mblock.mready = CreateEvent(NULL, FALSE, FALSE, NULL);
```

```
    /* Создать потоки производителя и потребителя; ожидать их завершения.*/
    /* … Как в программе 9.1 … */
    CloseHandle(mblock.mguard);
    CloseHandle(mblock.mready);
    _tprintf(_T("Потоки производителя и потребителя завершили выполнение\n"));
      _tprintf(_T("Отправлено:  %d,  Получено:  %d,  Известные  потери:  %d\n"),
mblock.sequence, mblock.nCons, mblock.nLost);
    return 0;
   }

   DWORD WINAPI produce(void *arg)
   /* Поток производителя — создание новых сообщений через случайные */
   /* интервалы времени. */
   {
    srand((DWORD)time(NULL)); /* Создать начальное число для генератора случайных
чисел. */
    while(!mblock.f_stop) {
     /* Случайная задержка. */
     Sleep(rand() / 10); /* Длительный период ожидания следующего сообщения. */
     /* Получить и заполнить буфер. */
     WaitForSingleObject(mblock.mguard, INFINITE);
     __try {
      if (!mblock.f_stop) {
       mblock.f_ready = 0;
       MessageFill(&mblock);
       mblock.f_ready = 1;
       mblock.sequence++;
       SetEvent(mblock.mready); /* Сигнал "Сообщение готово". */
      }
     } __finally { ReleaseMutex (mblock.mguard); }
    }
    return 0;
   }

   DWORD WINAPI consume (void *arg) {
    DWORD ShutDown = 0;
    CHAR command, extra;
    /* Принять ОЧЕРЕДНОЕ сообщение по запросу пользователя. */
     while (!ShutDown) { /* Единственный поток, получающий доступ к стандартным
устройствам ввода/вывода. */
     _tprintf(_T("\n** Введите 'c' для приема; 's' для прекращения работы: "));
     _tscanf("%c%c", &command, &extra);
     if (command == 's') {
      WaitForSingleObject(mblock.mguard, INFINITE);
      ShutDown = mblock.f_stop = 1;
      ReleaseMutex(mblock.mguard);
     } else if (command == 'c') {
      /* Получить новый буфер принимаемых сообщений. */
      WaitForSingleObject(mblock.mready, INFINITE);
      WaitForSingleObject(mblock.mguard, INFINITE);
      __try {
       if (!mblock.f_ready) _leave;
       /* Ожидать наступление события, указывающего на готовность сообщения. */
       MessageDisplay(&mblock);
       mblock.nCons++;
       mblock.nLost = mblock.sequence - mblock.nCons;
       mblock.f_ready = 0; /* Новые готовые сообщения отсутствуют. */
      } __finally { ReleaseMutex (mblock.mguard); }
     } else {
```

```
    _tprintf(_T("Недопустимая команда. Повторите попытку.\n"));
   }
  }
  return 0;
}
```

Примечание

Существует вероятность того, что поток потребителя, уведомленный о готовности сообщения, в действительности не успеет обработать текущее сообщение до того, как поток производителя сгенерирует еще одно сообщение до захвата мьютекса потоком потребителя. В результате такого поведения программы поток потребителя может обработать одно и то же сообщение дважды, если бы не проверка, предусмотренная в начале try-блока потребителя. Эта и другие аналогичные проблемы обсуждаются в главе 10.

Обзор: объекты синхронизации Windows

Наиболее важные свойства объектов синхронизации Windows перечислены в табл. 8.2.

Таблица 8.2. Сравнительные характеристики объектов синхронизации Windows

	CRITICAL_SECTION	Мьютекс	Семафор	Событие
Именованный защищаемый объект синхронизации	Нет	Да	Да	Да
Доступность из нескольких процессов	Нет	Да	Да	Да
Синхронизация	Вхождение	Ожидание	Ожидание	Ожидание
Освобождение	Выход	Мьютекс может быть освобожден или оставлен без контроля.	Освобождается любым потоком.	Функции SetEvent, PulseEvent.
Права владения	В каждый момент времени иметь права владельца может только один поток. Владеющий поток может осуществлять вхождение несколько раз, не блокируя свое выполнение.	В каждый момент времени иметь права владельца может только один поток. Владеющий поток может выполнять функцию ожидания несколько раз, не блокируя свое выполнение.	Понятие владения неприменимо. Доступ разрешен одновременно нескольким потокам, число которых ограничено максимальным значением счетчика.	Понятие владения неприменимо. Функции SetEvent и PulseEvent могут быть вызваны любым потоком.
Результат освобождения	Разрешается вхождение одного потока из числа ожидающих.	Вслед за последним освобождением права владения разрешается приобрести одному потоку из числа ожидающих.	Продолжать выполнение могут несколько потоков, число которых определяется текущим значением счетчика.	После вызова функций SetEvent или PulseEvent продолжать выполнение будет один или несколько ожидающих потоков.

Ожидание сообщений и объектов

Функция MsgWaitForMultipleObjects аналогична функции WaitForMultipleObjects. Применяйте ее для того, чтобы разрешить потоку или процессу обработку событий

пользовательского интерфейса, таких как щелчки мышью, во время ожидания перехода объектов синхронизации в сигнальное состояние.

Дополнительные рекомендации относительно использования мьютексов и объектов CRITICAL_SECTION

К этому времени мы успели познакомиться со всеми объектами синхронизации Windows и исследовали их применимость на ряде примеров. Мьютексы и объекты CS рассматривались первыми, а поскольку события мы еще будем интенсивно использовать в следующей главе, то настоящую главу целесообразно завершить рекомендациями относительно применения мьютексов и объектов CS для обеспечения корректности выполнения, удобства сопровождения и повышения производительности программ.

Приведенные ниже утверждения сформулированы, как правило, в терминах мьютексов, однако, если не оговорено иное, все сказанное относится и к объектам CS.

• Если функция WaitForSingleObject, одним из аргументов которой является дескриптор мьютекса, вызывается без использования конечного интервала ожидания, то вызывающий поток может оказаться блокированным на неопределенное время. Ответственность за то, чтобы захваченный (блокированный) мьютекс в конечном счете был освобожден (разблокирован), возлагается на программиста.

• Если поток завершает выполнение или его выполнение прерывается до того, как он покинет (разблокирует) объект CS, то этот объект остается блокированным. Чрезвычайно полезным свойством мьютексов является то, что владеющий ими поток может завершить выполнение, не уступив прав владения мьютексом.

• Не пытайтесь получить доступ к ресурсам, защищаемым мьютексом, если функция WaitForSingleObject вызвана с использованием конечного интервала ожидания.

• Ожидать перехода блокированного мьютекса в сигнальное состояние могут сразу несколько потоков. Когда мьютекс освобождается, то только один из ожидающих потоков получает права владения мьютексом и переводится в состояние готовности планировщиком ОС на основании действующей стратегии приоритетов и планирования. Не следует делать никаких предположений относительно того, что какой-либо поток будет пользоваться приоритетом; как и в любом другом случае, программу следует проектировать таким образом, чтобы приложение работало корректно независимо от того, какой именно из ожидающих потоков получит права владения мьютексом и возобновит выполнение. Те же замечания остаются справедливыми и в отношении потоков, ожидающих наступления события; никогда не следует предполагать, что при переходе объекта события в сигнальное состояние освободится какой-то определенный поток или что потоки будут разблокированы в какой-то определенной очередности.

• К критическому участку кода относятся все операторы, расположенные между точками программы, в которых поток приобретает права владения мьютексом и уступает их. Для определения нескольких критических участков кода может быть использован один и тот же мьютекс. Корректная организация программы предполагает, что критический участок кода, определяемый мьютексом, в каждый момент времени может выполняться только одним потоком.

• Определяемая мьютексами степень детализации программы, или гранулярность мьютексов (mutex granularity), оказывает влияние на производительность и требует серьезного рассмотрения. Размер каждого критического участка кода ни в коем случае не должен превышать необходимой величины, и мьютекс не должен захватываться на более длительный промежуток времени, чем это необходимо. Использование критических участков кода чрезмерно большого размера, захватываемых на длительные промежутки времени, снижает параллелизм и может оказывать отрицательное влияние на производительность.

• Связывайте мьютекс непосредственно с ресурсом, защиту которого он должен обеспечивать, возможно, с использованием структуры данных. (Именно эта методика задействована в программах 8.1 и 8.2.)

• Максимально точно документируйте инвариант, используя для этого словесные описания либо логические, или булевские, выражения. Инвариант— это свойство защищаемого ресурса, сохранение которого неизменным вне критического участка кода вы гарантируете. Форма выражения инвариантов может быть самой различной: "элемент принадлежит обоим спискам или не принадлежит ни одному из них", "контрольная сумма данных в буфере является достоверной", "связанный список является действительным" или "0 <= nLost + nCons <= sequence". Точно сформулированные инварианты могут использоваться совместно с макросом ASSERT при отладке программ, хотя оператор ASSERT должен иметь собственный критический участок кода.

• Убедитесь в том, что каждый критический участок кода имеет только одну точку входа, в которой поток блокирует мьютекс, и только одну точку выхода, в которой поток освобождает мьютекс. Избегайте использования сложных операторов ветвления и таких операторов, как break, return или goto, предоставляющих возможность выхода за пределы критического участка кода. Для защиты от подобных рисков оказываются удобными обработчики завершения.

• Если требуемая логика работы программы приводит к чрезмерному разрастанию критического участка кода (скажем, его размер превышает одну страницу), попробуйте разместить этот код в отдельной функции, чтобы можно было легко понять схему синхронизации. Так, целесообразно выделить в отдельную функцию код, предназначенный для удаления узла из сбалансированного дерева поиска, пока дерево остается блокированным.

Другие функции взаимоблокировки

Ранее уже было продемонстрировано, что функции InterlockedIncrement и InterlockedDecrement могут пригодиться в тех случаях, когда все, что требуется — это выполнение простейших операций над переменными, доступ к которым разделяется несколькими потоками. Используя некоторые другие функции, вы можете выполнять атомарные операции, позволяющие осуществлять сравнение и обмен значениями пар переменных.

Функции взаимоблокировки настолько же полезны, насколько и эффективны; эти функции реализуются в пользовательском пространстве с применением всего лишь нескольких машинных команд.

Функция InterlockedExchange сохраняет значение одной переменной в другой.

```
LONG InterlockedExchange(LPLONG Target, LONG Value)
```

Эта функция возвращает текущее значение переменной, на которую указывает параметр Target, и устанавливает значение этой переменной равным Value. Функция InterlockedExchangeAdd прибавляет второе значение к первому.

```
LONG InterlockedExchangeAdd(PLONG Addend, LONG Increment)
```

Значение Increment прибавляется к значению переменной, на которую указывает параметр Addend, а начальное значение этой переменной возвращается функцией. Данная функция позволяет увеличивать значение переменной на 2 (и более) атомарным образом, чего невозможно добиться последовательными вызовами функции InterlockedIncrement.

Последняя из функций этой группы, которую мы рассмотрим — это функция InterlockedCompareExchange, аналогичная функции InterlockedExchange, если не считать того, что обмен значениями осуществляется лишь в случае равенства сравниваемых значений.

```
PVOID InterlockedCompareExchange(PVOID *Destination, PVOID Exchange,
PVOID Comparand)
```

Эта функция выполняет атомарным образом следующие действия (использование типа данных PVOID для двух последних параметров может казаться вам непонятным):

```
Temp = *Destination;
if (*Destination == Comparand) *Destination = Exchange;
return Temp;
```

Одним из вариантов применения этой функции является управление блокировкой с целью реализации критического участка кода. *Destination является *переменной блокировки* (lock variable), причем значению 1 соответствует разблокированное состояние, а значению 0 — блокированное. Значение Exchange задается равным 0, а Comparand — 1. Вызывающему потоку известно, что она владеет критическим участком, если функция возвращает 1. В противном случае вызывающий поток должен "уснуть", или выполнить ожидание в состоянии занятости ("spin"), то есть совершать в течение короткого промежутка времени цикл, в котором ничего не делается, с той только целью, чтобы выждать некоторое время, а затем вновь повторить попытку. По существу, именно такой цикл и выполняет функция EnterCriticalSection, ожидая перехода в сигнальное состояние объекта CRITICAL_SECTION с ненулевым значением спин-счетчика; для получения более подробной информации по этому вопросу обратитесь к главе 9.

Учет факторов производительности при организации управленияпамятью

Программа 9.1, приведенная в следующей главе, позволяет исследовать различные аспекты производительности в условиях, когда несколько потоков соревнуются между собой за право обладания разделяемыми ресурсами. Аналогичные эффекты будут наблюдаться и в случае, когда потоки привлекаются для управления памятью с использованием функций malloc и free из многопоточной стандартной библиотеки C, поскольку эти функции используют объекты CRITICAL_SECTION для синхронизации доступа к структуре данных кучи (вы можете в этом сами убедиться, просмотрев исходный код библиотеки C). Ниже описаны два возможных способа улучшения производительности.

• Каждый поток, управляющий памятью, может создать дескриптор типа HANDLE для собственной кучи с помощью функции HeapCreate (глава 5). После этого для распределения памяти вместо функций malloc и free можно использовать функции HeapAlloc и HeapFree.

• Значение переменной окружения времени выполнения __MSVCRT_HEAP_SELECT можно установить равным __GLOBAL_HEAP_SELECTED. Это приведет к тому, что функции malloc и free будут использовать для управления памятью схему Windows, которая использует спин-блокировки (spin locks) вместо объектов CS и может быть намного более эффективной. Этот метод был предложен Гербертом Орашем (Gerbert Orasche) в статье "Configuring VC++ Multithreaded Memory Management", опубликованной в майском выпуске журнала *Windows Developer's Journal* за 2000 год, а представленные в этой статье результаты убедительно свидетельствуют о преимуществах данного метода в отношении производительности.

Резюме

Windows поддерживает полный набор операций синхронизации, способных обеспечить безопасную реализацию потоков и процессов. Синхронизация привносит в проектирование и разработку программ массу проблем, требующих самого тщательного рассмотрения, которое могло бы гарантировать не только корректную работу программ, но и их высокую производительность.

В следующих главах

В главе 9 внимание концентрируется на тех аспектах производительности, которые связаны с многопоточным характером приложений и применением в них объектов синхронизации. Сначала анализируются факторы, влияющие на производительность SMP-систем; в некоторых случаях производительность может резко ухудшаться из-за конфликтов за право владения ресурсами, в связи с чем предлагается несколько стратегий, обеспечивающих поддержание эксплуатационных характеристик SMP-систем на высоком уровне. Далее следует сравнительный анализ достоинств и недостатков мьютексов и объектов CRITICAL_SECTION, а затем рассматривается тонкая настройка объектов CRITICAL_SECTION с использованием спин-счетчиков. Завершается глава рекомендациями, в которых суммируются известные методики повышения производительности и заостряется внимание на возможных рисках.

Дополнительная литература

Windows

Вопросы синхронизации важны для любой ОС, и поэтому многие руководства по ОС содержат их подробное обсуждение в рамках более общего контекста.

Ранее уже упоминались другие книги, посвященные синхронизации в Windows. В то же время, при чтении книг по Windows более общего характера следует быть очень внимательными, поскольку в том, что касается потоков и синхронизации, некоторые из них могут попросту дезориентировать, и большинство из них не были обновлены с целью включения в рассмотрение средств NT5, которые мы используем в данной книге. Так, в одной очень популярной книге, получившей положительные отзывы рецензентов, несмотря на большой объем содержащихся в ней словесных описаний, ни слова не говорится о классе памяти volatile, не совсем правильно объяснены четыре модели событий, а в качестве метода, позволяющего изменить значение счетчика семафора более чем на единицу, рекомендуется решение, в котором используются многократные вызовы функций ожидания, что чревато возникновением взаимоблокировок (вспомните обсуждение в разделе, посвященном семафорам).

Для углубленного изучения тематики потоков и синхронизации можно порекомендовать книгу [6], которая будет полезна даже тем, кто программирует исключительно в среде Windows. Приведенные в этой книге обсуждения и описания в равной степени применимы, как правило, и к Windows, а перенос примеров программ послужит вам хорошим упражнением.

Упражнения

8.1. На Web-сайте книги находится версия программы simplePC.c (программа 8.1), содержащая дефекты, которая называется simplePCx.c. Проверьте работу этой программы и опишите симптомы дефектов, если они проявляются. Внесите в программу необходимые исправления, не сверяясь с правильным решением.

8.2. Измените программу simplePC.c таким образом, чтобы промежуток времени между генерацией новых сообщений увеличился. *(Подсказка.* Уберите операцию деления в том месте программы, где вызывается функция sleep.) Убедитесь в правильности логики, определяющей наличие новых сообщений. Кроме того, самостоятельно поэкспериментируйте с программой simplePCx.c, содержащей дефекты.

8.3. Переделайте программу simplePC. c, задействовав в ней мьютексы.

8.4. Переделайте программу sortMT.c (программа 7.2), использовав для синхронизации запуска рабочих потоков не приостановку потоков, а семафор.

8.5. Переделайте программу sortMT.c (программа 7.2), использовав для синхронизации запуска рабочих потоков не приостановку потоков, а события. В рекомендуемом решении используется функция SetEvent и сбрасываемое вручную событие. Другие комбинации не могли бы гарантировать корректную работу программы. Дайте этому свои объяснения.

8.6. Поэкспериментируйте с программой 8.2, используя различные комбинации автоматически и вручную сбрасываемых событий, а также функций SetEvent и PulseEvent (в текущем решении используются функция SetEvent и автоматически сбрасываемое событие). Могут ли считаться корректными альтернативные и исходный варианты реализации с учетом объявленного функционального назначения программы? (См. примечание после программы 8.2.) Объясните результаты и поясните, в чем с функциональной точки зрения состоит полезность альтернативных вариантов реализации. Можете ли вы добиться того, чтобы заработали альтернативные варианты реализации, изменив логику программы?

8.7. Создайте пул рабочих потоков, но организуйте такое управление частотой выполнения рабочих потоков, чтобы на протяжении любого односекундного интервала времени выполняться мог только один поток. Измените программу таким образом, чтобы на протяжении одного интервала могли выполняться два потока, но суммарная частота выполнения потоков соответствовала одному потоку в секунду. *Подсказка.* Рабочие потоки должны ожидать наступления события (события какого типа?) и управляющий поток должен переводить событие в сигнальное состояние (с помощью функции SetEvent или PulseEvent?) каждую секунду.

8.8. *Упражнение повышенной сложности.* Объекты CRITICAL_SECTION предназначены для использования потоками в рамках одного и того же процесса. Что произойдет, если объект CS будет создан в разделяемой отображаемой области памяти? Смогут ли использовать CS оба процесса? Вы можете провести самостоятельный эксперимент, изменив программу таким образом, чтобы производитель и потребитель выполнялись в различных процессах.

ГЛАВА 9
Влияние синхронизации на производительность и рекомендации по ее повышению

В предыдущей главе были введены операции синхронизации, использование которых иллюстрировалось с привлечением нескольких относительно простых примеров. В следующей главе предлагаются более сложные, но вместе с тем более реалистичные и полезные примеры, а также описывается общая модель синхронизации, позволяющая решить многие практические задачи и повысить надежность программ. В данной же небольшой главе анализируется влияние синхронизация на производительность приложений и рассматриваются методы, минимизирующие отрицательные последствия этого влияния.

Несмотря на всю важность синхронизации потоков, применение этого средства сопряжено со значительными рисками снижения производительности, которые ниже частично обсуждаются на примере как однопроцессорных, так и многопроцессорных (SMP) систем. У возможных альтернативных решений имеются собственные достоинства и недостатки. Например, объекты CRITICAL_SECTION (CS) и мьютексы обладают почти одинаковыми свойствами и решают одну и ту же фундаментальную задачу. Вообще говоря, наиболее эффективным механизмом блокирования являются объекты CS, хотя это справедливо не во всех ситуациях. Кроме того, как показано в главе 10, объекты CS менее удобны в работе по сравнению с мьютексами. В некоторых случаях достаточно использовать функции взаимоблокировки потоков, а при тщательном проектировании и реализации приложения иногда можно вообще обойтись без использования объектов синхронизации.

Сначала мы обсудим сравнительные достоинства и недостатки объектов CS и мьютексов, дополнив этот анализ учетом факторов, проявляющихся в SMP-системах. К числу других рассмотренных ниже тем относятся спин-счетчики объектов CS, дросселирование семафоров и родство процессоров. Глава заканчивается сводкой рекомендаций, касающихся оптимизации производительности.

Примечание

В NT 5.0 достигнут значительный прогресс в плане повышения производительности. В ранних версиях NT и в Windows 9x некоторые из отмеченных выше проблем носили гораздо более острый характер.

Влияние синхронизации на производительность

Использование синхронизации в программах может и будет ухудшать их производительность, и в этом отношении следует быть особенно осмотрительным в случае SMP-систем. На первый взгляд, это противоречит здравому смыслу, поскольку от SMP-систем в целом можно было, бы ожидать только повышения производительности, а уж о том, что при переходе к ним быстродействие программ может снижаться, казалось бы, и речи идти не может. Тем не менее, в силу особенностей внутренних механизмов реализации, а также конкуренции между процессорами за право доступа к памяти могут наблюдаться неожиданные эффекты, в том числе и резкое ухудшение производительности программы.

Достоинства и недостатки объектов CRITICAL_SECTION

Прежде всего, мы попытаемся количественно оценить влияние объектов синхронизации на производительность, и сравним между собой объекты CRITICAL_SECTION и мьютексы. В программе statsMX.c (программа 9.1) для синхронизации доступа к специфической для каждого потока структуре данных используется мьютекс. Программа statsCS.c, листинг которой здесь не приводится, но его можно найти на Web-сайте книги, делает точно то же, но с использованием объекта CRITICAL_SECTION, тогда как в программе stats IN. с для этого привлекаются функции взаимоблокировки (interlocked functions). Наконец, в программе statsNS.c, которая также здесь не приводится, синхронизация вообще не используется; оказывается, в данном примере можно вообще обойтись без синхронизации, поскольку каждый рабочий поток обращается к собственной уникальной области памяти. Некоторые предостережения по этому поводу приведены в конце данного раздела. В реальных программах количество рабочих потоков может быть неограниченным, однако для простоты в программе 9.1 обеспечивается поддержка 64 потоков.

Описанная совокупность программ не только позволяет оценить зависимость производительности от выбора конкретного типа объекта синхронизации, но и говорит о следующих вещах:

• При тщательном проектировании программы в некоторых случаях можно вообще обойтись без использования синхронизации.

• В простейших ситуациях, например, когда требуется инкрементировать значение совместно используемой переменной, достаточно использовать функции взаимоблокировки.

• В большинстве случаев использование мьютексов обеспечивают более высокое быстродействие программы по сравнению с использованием объектов CS.

• Обычная методика заключается в определении структуры данных аргумента потока таким образом, чтобы она содержала информацию о состоянии, которая должна поддерживаться потоком, а также указатель на мьютекс или иной объект синхронизации.

Программа 9.1. statsMX: поддержка статистики потоков

```
/* Глава 9. statsMX.c */
/* Простая система "хозяин/рабочий", в которой каждый рабочий поток */
/* информирует главный поток о результатах своей работы для их отображения.*/
/* Версия, использующая мьютекс. */
#include "EvryThng.h"
```

```
#define DELAY_COUNT 20
/* Использование: statsMX nthread ntasks */
/* Запускается "nthread" рабочих потоков, каждой из которых поручается */
/* выполнение "ntasks" единичных рабочих заданий. Каждый поток сохраняет*/
/* информацию о выполненной работе в собственной неразделяемой ячейке */
/* массива, хранящего данные о выполненной потоком работе. */
DWORD WINAPI worker(void *);
typedef struct _THARG {
 int thread_number;
 HANDLE *phMutex;
 unsigned int tasks_to_complete;
 unsigned int *tasks_complete;
} THARG;

int _tmain(DWORD argc, LPTSTR argv[]) {
 INT tstatus, nthread, ithread;
 HANDLE *worker_t, hMutex;
 unsigned int* task_count, tasks_per_thread;
 THARG* thread_arg;
 /* Создать мьютекс. */
 hMutex = CreateMutex(NULL, FALSE, NULL);
 nthread = _ttoi(argv[1]);
 tasks_per_thread = _ttoi(argv[2]);
 worker_t = malloc(nthread * sizeof(HANDLE));
 task_count = calloc(nthread, sizeof(unsigned int));
 thread_arg = calloc(nthread, sizeof(THARG));
 for(ithread = 0; ithread < nthread; ithread++) {
  /* Заполнить данными аргумент потока. */
  thread_arg[ithread].thread_number = ithread;
  thread_arg[ithread].tasks_to_complete = tasks_per_thread;
  thread_arg[ithread].tasks_complete = &task_count[ithread];
  thread_arg[ithread].phMutex = &hMutex;
       worker_t[ithread]   =   (HANDLE)_beginthreadex   (NULL,   0,   worker,
&thread_arg[ithread], 0, &ThId);
 }
 /* Ожидать завершения рабочих потоков. */
 WaitForMultipleObjects(nthread, worker_t, TRUE, INFINITE);
 free(worker_t);
 printf("Выполнение рабочих потоков завершено\n");
 for (ithread = 0; ithread < nthread; ithread++) {
   _tprintf(_T("Количество заданий, выполненных потоком %5d: %6d\n"), ithread,
task_count[ithread]);
 }
 return 0;
 free(task_count);
 free(thread_arg);
}

DWORD WINAPI worker(void *arg) {
 THARG * thread_arg;
 int ithread;
 thread_arg = (THARG*)arg;
 ithread = thread_arg->thread_number;
 while (*thread_arg->tasks_complete < thread_arg->tasks_to_complete) {
  delay_cpu(DELAY_COUNT);
  WaitForSingleObject(*(thread_arg->phMutex), INFINITE);
  (*thread_arg->tasks_complete)++;
  ReleaseMutex(*(thread_arg->phMutex));
 }
```

```
    return 0;
}
```

Для изучения поведения различных вариантов реализации можно воспользоваться программой timep из главы 6 (программа 6.2). Тесты, которые проводились на системах, не загруженных никакими другими задачами, и состояли в выполнении 250 000 единичных рабочих заданий с использованием 1,2,4, 8, 16, 32, 64 и 128 потоков, показали следующие результаты:

• При небольшом количестве потоков (4 и менее) для выполнения каждого из вариантов реализации NS (отсутствие синхронизации), IN (функции взаимоблокировки) и CS (объекты CRITICAL_SECTION) требуется примерно одно и то же время. Вариант CS может оказаться несколько более медленным (10-20 процентов), демонстрируя типичное замедление работы программ, использующих синхронизацию. Вместе с тем, для выполнения варианта MX (мьютексы) требуется в два-три раза больше времени.

• Производительность варианта CS на однопроцессорных системах при использовании 5 и более потоков не всегда изменяется пропорционально количеству потоков. Картина может меняться при переходе от одной NT5-системы к другой, однако, как свидетельствуют данные, для каждой конкретной системы результаты согласуются между собой. В случае некоторых систем истекшее время удваивается при переходе к следующему члену ряда 1, 2, 4 и так далее, соответствующему количеству используемых потоков, но в одном случае (Windows 2000, процессор Pentium с частотой 1 ГГц, портативный компьютер) оно составляло (в секундах) 0.5, 1.0, 2.0, 4.0, 14.9, 16.0, 32.1 и 363.4, а в другом (Windows 2000, процессор Pentium 500 МГц, на стольный компьютер) — 1.2, 2.3, 4.7, 9.3, 42.7, 101.3, 207.8 и 1212.5 секунд. Как правило, резкое изменение поведения происходит тогда, когда количество потоков начинает превышать 4 или 8, но производительность остается приемлемой, пока количество потоков не превышает 128.

• В случае однопроцессорных систем вариант MX уступает варианту CS, причем отношение показателей производительности варьирует в пределах от 2:1 до 10:1 в зависимости от типа системы.

• *В случае SMP-систем производительность может резко ухудшаться* в десятки и сотни раз. Интуитивно кажется, что с увеличением количества процессоров производительность может только повышаться, но в силу механизмов внутренней реализации процессоры конкурируют между собой за право владения блокировками и обращения к памяти, и это объясняет, почему результаты для вариантов MX и CS оказываются практически одинаковыми. В случае объектов CS некоторого улучшения производительности удавалось добиться за счет тонкой настройки спин-счетчиков, о чем говорится в одном из следующих разделов.

• Для ограничения количества готовых к выполнению рабочих потоков без изменения базовой программной модели можно использовать семафоры. Эта методика рассматривается далее в этой главе.

Предупреждение

В массиве task_count намеренно использованы 32-битовые целые числа, чтобы увеличить верхний предел значений счетчика заданий и избежать создания предпосылок для возникновения "разрыва слов" ("word tearing") и "конфликтов строки кэша" ("cache line conflict") в SMP-системах. Два независимых процессора, на которых выполняются смежные рабочие потоки, могут одновременно изменять значения счетчиков смежных заданий путем внесения соответствующих изменений в свои кэши (32-битовые в системах на основе Intel x86). Вместе с тем, реально записываться в память будет только один кэш, что может сделать результаты недействительными. Чтобы избежать возможных рисков, следует позаботиться об отделении рабочих ячеек

каждым из потоков друг от друга и их выравнивании в соответствии с размерами кэшей. В данном примере счетчик заданий может быть сгруппирован с аргументом потока, так что использованию 32-битовых счетчиков ничто не препятствует. Эта тема исследуется в упражнении 9.6.

Модельная программа для исследования факторов производительности

На Web-сайте книги находится проект TimedMutualExclusion, который вы можете использовать для проведения собственных экспериментов с различными моделями "хозяин/рабочий" и характеристиками прикладных приложений. Ниже приводится перечень возможностей этой программы, которыми можно управлять из командной строки.

• Использование объектов CS или мьютексов.

• Глубина, или *рекурсивность,* счетчиков.

• Время удержания блокировки, или *задержка* (delay), которое моделирует объем работы, выполненной в пределах критического участка кода.

• Количество рабочих потоков, ограниченное системными ресурсами.

• Количество *точек "засыпания"* (sleep points), в которых рабочий поток уступает процессор, используя вызов Sleep(0), но продолжает владеть блокировкой. Точки "засыпания" моделируют ожидание рабочим потоком операций ввода/вывода или событий, тогда как задержка моделирует активность ЦП.

• Количество активных потоков, о чем говорится в разделе, посвященном дросселированию семафоров.

Регулируя параметры задержек и точек "засыпания", можно оказывать заметное воздействие на производительность, поскольку от этих параметров зависит доля времени, в течение которого поток владеет блокировкой, не давая выполняться другим потокам.

В листинг программы включены детальные комментарии, объясняющие порядок запуска программы и настройки параметров. В упражнении 9.1 вам предлагается провести самостоятельные эксперименты с использованием как можно большего количества различных систем, к которым у вас имеется доступ. Видоизмененный вариант этой программы под названием MutualExclusionSC поддерживает спин-счетчики, о которых говорится в следующем разделе.

Примечание

Программа TimedMutualExclusion представляет простую модель, способную отражать многие из особенностей рабочих потоков. Во многих случаях ее можно настроить так, чтобы она представляла реальное приложение, и если эта модель позволяет выявить определенные проблемы, связанные с ухудшением производительности, то не исключено, что с аналогичными трудностями вы столкнетесь и в случае реального приложения. С другой стороны, хорошие эксплуатационные показатели модели вовсе не обязательно означают, что такими же качествами будет обладать и реальное приложение, хотя хорошая исходная модель и способна упростить настройку его производительности.

Настройка производительности SMP-систем с помощью спин-счетчиков

Эффективность методики блокирования (вхождение в раздел) и разблокирования (выход из раздела) объекта CRITICAL_SECTION объясняется тем, что тестирование объекта CS выполняется в пользовательском пространстве без использования системных вызовов ядра, как это требуется в случае мьютексов. Снятие блокировки раздела также выполняется полностью в пространстве пользователя, в отличие от функции ReleaseMutex, которая требует использования системного вызова. Объекты CS работают следующим образом:

• Поток, выполняющий функцию EnterCriticalSection (ECS), непрерывно тестирует бит блокировки объекта CS. Если обнаруживается, что бит выключен (объект разблокирован), ECS автоматически устанавливает его, выполняя эту операцию как часть инструкций тестирования, и продолжает дальнейшее выполнение уже без какого-либо ожидания. Поэтому блокирование разблокированного объекта CS осуществляется чрезвычайно эффективным образом и требует, как правило, всего лишь одной или двух машинных команд. Идентификационные данные владеющего потока, а также рекурсивный счетчик поддерживаются структурой данных объекта CS.

• Если обнаруживается, что объект CS блокирован, ECS входит в жесткий цикл (tight loop) на SMP-системах и выполняет многократное тестирование бита блокировки, не уступая процессора (разумеется, поток может быть вытеснен). Количество повторений цикла, после выполнения которых ECS прекращает дальнейшее тестирование, определяется значением спин-счетчика CS. В однопроцессорных системах тестирование прекращается немедленно; спин-счетчики используются лишь в случае SMP-систем.

• Как только ECS прекращает тестирование бита блокировки (в случае однопроцессорных систем это происходит немедленно), она входит в ядро, и поток переводится в состояние ожидания. Следовательно, блокирование объектов CS оказывается эффективным лишь в условиях низкой состязательности между потоками или когда спин-счетчик предоставляет другому процессору время для разблокирования CS.

• Функция LeaveCriticalSectibn реализуется путем выключения бита блокировки после проверки того, что вызывающий поток действительно является владельцем CS. Кроме того, ядро должно быть также уведомлено о том, существуют ли еще другие ожидающие потоки, для чего используется функции ReleaseSemaphore.

Таким образом, в случае однопроцессорных систем объекты CS эффективны тогда, когда высока вероятность их разблокирования, что иллюстрирует вариант CS программы 9.1. Преимущества SMP-систем обусловлены тем фактом, что пока ожидающий поток выполняет цикл ожидания, управляемый спин-счетчиком, объект CS может оказаться разблокированным потоком, выполняющимся на другом процессоре.

Далее будет показано, как устанавливать значения спин-счетчиков и настраивать приложения путем выбора наиболее оптимальных значений. Еще раз подчеркнем, что спин-счетчики оказываются полезными лишь в случае SMP-систем; на однопроцессорных системах они не используются.

Установка значений спин-счетчиков

Спин-счетчики CS могут устанавливаться на стадии инициализации объектов CS или динамическим путем. В первом случае функция InitializeCriticalSection заменяется функцией

InitializeCriticalSectionAndSpinCount, в которой добавлен параметр счетчика. В то же время, способа, позволяющего считать значение спин-счетчика, не существует.

```
    VOID           InitializeCriticalSectionAndSpinCount(LPCRITICAL_SECTION
lpCriticalSection, DWORD dwCount)
```

Значение спин-счетчика можно в любой момент изменить.

```
    VOID  SetCriticalSectionSpinCount(LPCRITICAL_SECTION  lpCriticalSection,
DWORD dwCount)
```

В документации Microsoft говорится о том, что рекомендуемым для управления кучей значением спин-счетчика является 4000. Однако оптимальное значение зависит от свойств приложения, и поэтому спин-счетчики должны настраиваться индивидуально для каждого приложения, выполняющегося в реалистическом SMP-окружении. Наилучшее значение будет различным в зависимости от количества процессоров, характера приложения и тому подобного.

На Web-сайте книги находится программа TimedMutualExclusionSC. Эта программа представляет собой видоизмененный вариант уже знакомой вам программы TimedMutualExclusion, в котором значение спин-счетчика указания в качестве параметра командной строки. Вы можете запустить эту программу на своей машине для приблизительной оценки того, какое значение спин-счетчика будет наиболее приемлемым для выполнения того или иного из вариантов тестовых программ на доступных вам SMP-системах, что и предлагается сделать в упражнении 9.2.

Дросселирование семафора для уменьшения состязательности между потоками

Слишком большое количество потоков, соревнующихся между собой за право владения единственным ресурсом, например, мьютексом или объектом CS, могут стать причиной снижения производительности как в однопроцессорных, так и в многопроцессорных системах. В большинстве случаев негативное влияние некоторых факторов на производительность может быть сведено к минимуму за счет использования спин-счетчиков, тщательного выбора типа объектов синхронизации или перестройки структуры программы с целью увеличения степени детализации блокировок и длительности периодов блокирования.

Если ни один из этих методов не дает желаемого улучшения, то могло бы показаться, что нет иного выхода, кроме как уменьшить количество потоков, но при этом вы будете вынуждены заставлять одиночные потоки мультиплексировать те операции, которые естественнее было бы распределить между несколькими потоками. Выход из этой ситуации обеспечивают семафоры, которые дают возможность сохранить естественную многопоточную модель, но вместе с тем свести к минимуму количество активных потоков, конкурирующих между собой. Такое решение является концептуально простым, и его можно без труда включить в любую существующую прикладную программу, например TimedMutualExclusion. В системе "хозяин/рабочий" решение, носящее название *"дросселя" семафора* (semaphore throttle), использует следующую методику:

• Главный поток создает семафор с небольшим, например 4, максимальным значением параметра, представляющего максимально допустимое количество активных потоков, которое, например, можно принимать равным количеству процессоров, установленных в системе, для обеспечения приемлемой производительности. Начальное значение счетчика семафора также следует установить равным максимальному значению. Это число можно сделать параметром и подбирать его оптимальное значение экспериментальным путем так же, как и в случае спин-счетчиков. Другим возможным значением этого параметра может служить количество процессоров, которое может быть определено во время выполнения программы (см. следующий раздел).

• Каждый рабочий поток ожидает перехода семафора в сигнальное состояние, прежде чем войти в критический участок кода. Ожидание семафора может непосредственно предшествовать ожиданию мьютекса или объекта CS.

• Если максимальное значение счетчика семафора равно 1, то использование мьютекса становится излишним. Подобное решение нередко является наилучшим для SMP-систем.

• Общий уровень состязательности между объектами CS или мьютексами снижается, если при сериализации выполнения потоков лишь небольшое количество потоков ожидают перехода мьютексов или объектов CS в сигнальное состояние.

Счетчик семафора просто представляет число потоков, которые могут быть активными в любой момент времени, и ограничивает количество потоков, соревнующихся между собой за право владения мьютексом, объектом CS или иным ресурсом. Главный поток даже может регулировать, или, как говорят, "дросселировать" (throttle) выполнение рабочих потоков и динамически настраивать работу приложения, ожидая, пока не уменьшится значение счетчика, если он решает, что уровень состязательности слишком высок, или увеличивая значение счетчика с помощью функции ReleaseSemaphore, чтобы дать возможность выполняться большему количеству потоков. Заметьте, однако, что максимальное значение счетчика семафора устанавливается при его создании, и изменить его впоследствии невозможно.

В приведенном ниже фрагменте кода представлен видоизмененный рабочий цикл,

выполняющий две операции с семафором.

```
while (TRUE) { // Рабочий цикл
 WaitForSingleObject(hThrottleSem, INFINITE);
 WaitForSingleObject(hMutex, INFINITE);
 … Критический участок кода …
 ReleaseMutex(hMutex);
 ReleaseSemaphore(hThrottleSem, 1, NULL);
} // Конец рабочего цикла
```

Можно предложить еще одну разновидность программы. Если некоторый рабочий поток потребляет "слишком много" ресурсов, то можно заставить его выжидать некоторое время, пока значение счетчика семафора не уменьшится на несколько единиц. Однако, как уже отмечалось в предыдущей главе, использование двух последовательных циклов ожидания может стать причиной взаимоблокировки (deadlock) потоков. В следующей главе в одном из упражнений (упражнение 10.11) показано, как построить сложный объект семафора, допускающий атомарное выполнение многократных функций ожидания.

В уже упоминавшемся примере программы TimedMutualExclusion добавлен шестой параметр, являющийся начальным значением дроссельного счетчика семафора для количества активных потоков. Вы можете поэкспериментировать со значениями этого счетчика, как предлагается в одном из упражнений. На рис. 9.1 показана зависимость различных временных характеристик для шести потоков, синхронизируемых посредством одного объекта CS, от количества активных потоков, изменяющегося в интервале от 1 до 6. Во всех случаях объем выполняемой работы остается одним и тем же, но истекшее время резко увеличивается, когда количество активных потоков превышает 4.

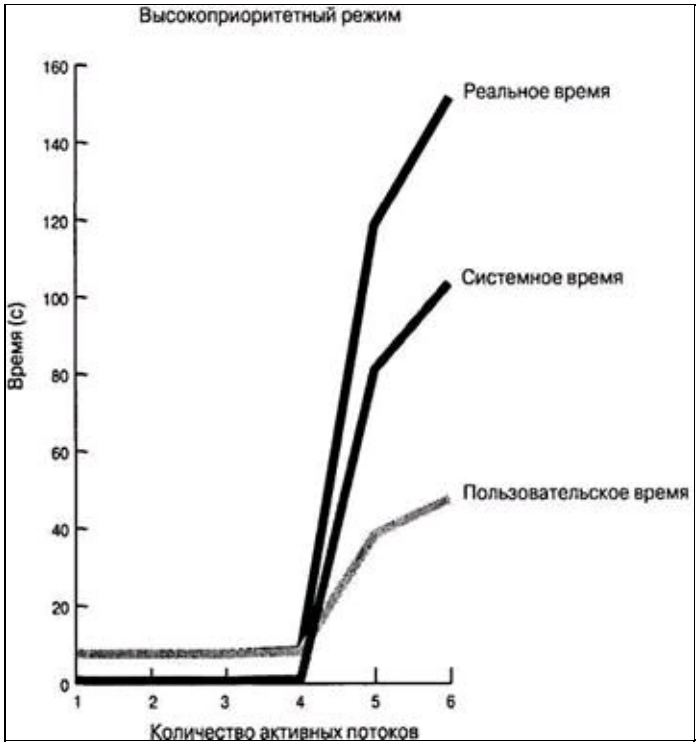


Рис. 9.1. Зависимость производительности от количества потоков

Указанные зависимости получены на устаревших, медленных системах. Для системы Windows 2000 на базе процессора Intel 586 (Pentium), характеризующейся гораздо более высоким быстродействием, соответствующие значения истекшего времени для 1–6 потоков составили (в секундах) 0.8, 0.8, 2.3, 21.2, 28.4 и 29.0, и эти результаты могут быть последовательно воспроизведены. В этом случае ухудшение производительности становилось заметным, начиная уже с 3 активных потоков. *В то же время,* соответствующие временные характеристики,

оцененные с использованием произвольно выбранной совокупности аналогичных систем, оказались примерно постоянными, независимо от количества активных потоков. Результаты некоторых экспериментов дают основания сделать следующие выводы:

• В системе NT5 достигнут значительный прогресс по сравнению с NT4, по следовательно демонстрирующей результаты, аналогичные тем, которые представлены на рис. 9.1.

• Получаемые результаты зависят от того, в каком режиме выполняются операции — приоритетном или фоновом, то есть, находится или не находится фокус на окне приложения, а также от присутствия в системе других выполняемых задач.

• Как правило, мьютексы работают медленнее по сравнению с объектами CS, но в случае NT5 результаты остаются примерно постоянными, независимо от количества активных потоков.

• В SMP-системах наиболее предпочтительным вариантом является дросселирование семафора при значении счетчика равном 1. В этом случае мьютексы становятся ненужными. Так, в случае двухпроцессорной системы Xeon частотой 1.8 ГГц использованные времена для варианта CS при 1, 2 и 4 активных потоках составили 1.8, 33.0 и 31.9 секунды. Соответствующие времена в случае мьютекса составили 34.0, 66.5 и 65.0 секунды.

Резюме. Дросселирование семафоров может обеспечивать хорошую производительность как для приоритетных, так и для фоновых операций даже в случае систем, загруженных выполнением других-задач. Дроссели семафоров могут играть очень важную роль в случае SMP-систем, для которых количество активных потоков должно быть равным 1. В том, что касается производительности, семафоры, по-видимому, более эффективны, чем мьютексы.

Родство процессоров

Во всем предшествующем обсуждении предполагалось, что все процессоры SMP-системы доступны всем потокам, а планирование выполнения потоков и распределение процессоров между ними осуществляет ядро. По своей сути такой простой подход является вполне естественным и согласуется с природой SMP-систем. В то же время, имеется возможность назначать потокам определенные процессоры, задавая так называемое родство процессоров (processor affinity). Родство процессоров можно использовать в нескольких ситуациях.

• Процессор может быть назначен небольшой группе, состоящей из одной и более высокоприоритетных потоков.

• Рабочие потоки, конкурирующие за право владения единственным ресурсом, могут быть распределены для выполнения на одном процессоре, что позволяет избежать затруднений с производительностью в случае SMP-систем, о которых перед этим говорилось.

• Возможен и другой вариант, когда потоки распределяются по доступным процессорам.

• Различным процессорам можно назначать различные классы рабочих потоков.

Маски родства системы, процесса и потока

У каждого процесса имеется собственная маска родства процесса (process affinity mask), представляющая собой битовый вектор. Существует также маска родства системы (system affinity mask).

• Маска родства системы отображает процессоры, сконфигурированные в системе.

• Маска родства процесса отображает процессоры, на которых разрешается выполнение потоков данного процесса.

• Каждый индивидуальный поток имеет маску родства потока (thread affinity mask), которая должна представлять собой подмножество значений маски родства процесса. Первоначально маска родства потока совпадает с маской родства процесса.

Существуют функции для получения и установки масок, хотя системную маску вы можете только считывать (получать), а маски потоков — только устанавливать. Функции установки масок используют дескрипторы потоков и процессов, поэтому процессы или потоки могут устанавливать маску родства друг для друга, если имеются соответствующие права доступа, или для самих себя. Установка маски никак не повлияет на поток, уже выполняющийся на процессоре, использование которого вы пытаетесь исключить данной маской.

Для считывания как системных масок родства, так и масок родства процессов используется одна функция — GetProcessAffinityMask. В однопроцессорных системах, включая Windows 9x, все биты маски должны быть равными 1.

```
BOOL GetProcessAffinityMask(HANDLE hProcess, LPDWORD
lpProcessAffinityMask, LPDWORD lpSystemAffinityMask)
```

Маска родства процесса, которая будет наследоваться любым дочерним процессом, устанавливается при помощи функции SetProcessAffinityMask.

```
BOOL SetProcessAffinityMask(HANDLE hProcess, DWORD
dwProcessAffinityMask)
```

В документации Microsoft говорится, что значение новой маски должно быть *строгим*

подмножеством (proper subset) значений масок, получаемых с помощью функции GetProcessAffinityMask. Как показывает несложный эксперимент, включенный в код программы TimedMutualExclusion, новая маска может быть той же, что и маска системы или предыдущая маска процесса. Однако упомянутое ограничение не может быть справедливым, ибо в таком случае вы были бы лишены возможности восстанавливать маску родства системы до предыдущего значения.

В Windows 9x поддержка SMP, а также функций манипулирования масками процессов не поддерживаются. Новые значения масок влияют на все потоки, принадлежащие данному процессу.

Для установки маски родства потоков применяется аналогичная функция.

```
DWORD SetThreadAffinityMask(HANDLE hThread, DWORD dwThreadAffinityMask)
```

Типы возвращаемых значений этих функций не согласуются между собой. Типом возвращаемого значения функции SetThreadAffinityMask является DWORD, а не BOOL, но результат остается одним и тем же (1 — в случае успеха, 0 — в противном случае). Функция SetThreadAffinityMask работает и под управлением Windows 9x, но маска должна быть единичной, что не дает никакого прока. Кроме того, невзирая на документацию, новая маска не обязательно должна быть строгим подмножеством системной маски.

Функция SetThreadIdealProcessor является видоизменением функции SetThreadAffinityMask. Вы указываете предпочтительный ("идеальный") номер процессора (а не маску), и планировщик назначает потоку этот процессор, если такая возможность имеется, или назначит ему другой процессор, если предпочтительный процессор недоступен. Возвращаемым значением функции является номер предыдущего предпочтительного процессора, если таковой был назначен.

Определение количества процессоров в системе

Фактически, на количество процессоров, установленных в системе, указывает маска родства системы; чтобы его определить, вам достаточно подсчитать количество ненулевых битов в маске. Вместе с тем, гораздо проще вызвать функцию GetSystemInfo, возвращающую структуру SYSTEM_INFO, среди полей которой имеются поля, содержащие количество процессоров и активную маску процессоров, которая совпадает с маской системы. Простая программа и проект Version, доступные на Web-сайте книги, отображают эту информацию вместе с версией Windows.

Гиперпотоки и счетчик процессоров

Процессоры Intel Pentium 4 и Xeon поддерживают механизм HyperThreading (гиперпотоки), посредством которого состояния ожидания, возникающие в процессе выполнения потока, используются для выполнения другого потока. Для поддержки этого средства используется второй регистровый файл, что вполне осуществимо, поскольку архитектура процессоров x86 характеризуется сравнительно небольшим количеством регистров. Xeon или любой другой процессор, поддерживающий гиперпоточную обработку, воспринимается функциями GetSystemInfo и GetProcessAffinityMask как одиночный процессор.

Порты завершения ввода/вывода

В главе 14 описываются порты завершения ввода/вывода, которые предоставляют другой механизм, позволяющий избежать состязательности между потоками путем ограничения их количества. Порты завершения ввода/вывода дают возможность небольшому количеству потоков управлять большим количеством параллельно выполняющихся операций ввода/вывода. Отдельные операции ввода/вывода начинают выполняться в асинхронном режиме и, вообще говоря, не завершаются сразу же после того, как осуществляется возврат из функции чтения или записи. В то же время, обработка данных по мере завершения операций, ожидающих выполнения, поручается одной из небольшого числа рабочих потоков. В главе 14 приведен пример сервера, связывающегося с удаленными клиентами (программа 14.4).

Рекомендации по повышению производительности и возможные риски

Многопоточные приложения предоставляют значительные программные преимущества, включая возможность использования более простых моделей программирования и повышение быстродействия программ. Вместе с тем, существует ряд факторов, которые способны оказывать на производительность заметное отрицательное влияние, с трудом поддающееся прогнозированию, причем характер этого влияния может быть различным на различных компьютерах, даже если на них и выполняются одни и те же версии Windows. Некоторые простые рекомендации, суммирующие сведения, изложенные в настоящей главе, помогут вам минимизировать эти риски. Часть этих рекомендаций, равно как и многие из советов по проектированию, отладке и тестированию программ, которые приводятся в следующей главе, в переработанном виде взята из [6].

• Критически относитесь к аргументации предположительного и теоретического характера, касающейся вопросов производительности, которая часто звучит убедительно, но на практике оказывается ошибочной. Проверяйте предположения на простых прототипах программ, таких как TimedMutualExclusion, или проверяйте их действенность на альтернативных вариантах реализации своего приложения.

• Используйте для тестирования производительности приложений как можно более широкий круг систем из числа тех, которые доступны вам. Полезно запускать программу с использованием самых различных конфигураций памяти, типов процессоров, версий Windows и количества процессоров. *Приложение может продемонстрировать очень высокую производительность на одной системе, но крайне низкую на другой;* см. обсуждение программы 9.1.

• Блокирование потребляет значительные системные ресурсы; пользуйтесь этим средством лишь при настоятельной необходимости. Предоставляйте возможность удержания (владения) мьютекса или объекта CS строго в пределах лишь необходимого времени. Варьирование параметров задержки или точек "засыпания" демонстрирует снижение производительности с увеличением длительности периодов блокирования.

• Используйте различные мьютексы для различных ресурсов, чтобы уменьшить степень детализации блокировок настолько, насколько это возможно. В частности, старайтесь не использовать глобальные блокировки.

• Условия высокой состязательности между блокировками затрудняют достижение высокой производительности. Чем выше частота блокирования и разблокирования потоков, тем заметнее снижается производительность. Ухудшение производительности с увеличением количества потоков может быть очень резким, заметно отклоняясь от простой линейной зависимости.

• Объекты CS предоставляют эффективный упрощенный механизм блокирования при небольшом количестве конкурирующих потоков, но в некоторых случаях мьютексы обеспечивают лучшую производительность. При использовании объектов CS в критических по отношению к производительности SMP-приложениях возможно настройка производительности с помощью спин-счетчиков.

• Семафоры могут помочь уменьшить количество конкурирующих активных потоков, не вынуждая вас менять программную модель.

• Переход на SMP-систему может приводить к неожиданному ухудшению производительности в тех случаях, когда производительность, казалось бы, могла только улучшиться. Сохранить приемлемую производительность в подобных ситуациях позволяют

методики, уменьшающие состязательность между потоками и использующие маски родства потоков.

• Заметное влияние на производительность оказывает также выбор модели — сигнальной или широковещательной, о чем более подробно говорится в главе 10.

• Используйте доступные стандартные программы протоколирования, позволяющие оценивать время выполнения различных функций и анализировать факторы, влияющие на производительность, что поможет вам лучше представить себе поведение потоков в вашей программе и определить участки кода, выполнение которых занимает наибольшее время.

Резюме

Применение синхронизации может отрицательно воздействовать на производительность как в однопроцессорных, так и в SMP-системах, причем степень такого влияния иногда может становиться весьма существенной. Добиться хорошей производительности в подобных ситуациях можно путем тщательного проектирования программы и правильного выбора типов объектов синхронизации. В этой главе рассмотрен целый ряд полезных методик и даны рекомендации, которые помогут вам поддерживать производительность своих программ на высоком уровне, а также изучен характер возникающих при этом проблем, которые были проиллюстрированы на примере простой тестовой программы, отражающей наиболее существенные характеристики многих реальных ситуаций.

В следующих главах

В главе 10 рассматриваются более общие способы использования объектов синхронизации Windows и обсуждаются некоторые модели программирования, помогающие обеспечивать корректность программ и удобство их сопровождения, а также повышать их производительность. Также в главе 10 создаются несколько сложных объектов синхронизации, которые оказываются полезными при разрешении ряда важных проблем. В последующих главах демонстрируются различные способы использования потоков и объектов синхронизации, находящие применение в таких, например, приложениях, как серверы. Наряду с этим внимание уделено и некоторым фундаментальным аспектам использования потоков. Например, в главе 12 обсуждаются такие темы, как безопасный многопоточный режим и повторное использование библиотек DLL.

Дополнительная литература

Литературные источники, относящиеся также к данной главе, перечислены в главе 10.

Упражнения

9.1. Поэкспериментируйте с программой statsMX, используя для этого собственную систему, а также как можно большее количество других доступных вам систем, отличающихся друг от друга не только аппаратным обеспечением, но и версиями Windows. Аналогичны ли полученные вами результаты тем, о которых сообщалось в настоящей главе? Что наблюдается в случае SMP-систем?

9.2. Используйте функцию TimedMutualExclusionSC для экспериментальной проверки того, что путем изменения значений спин-счетчиков объектов CRITICAL_SECTION действительно можно улучшить производительность SMP-систем в случае большого количества потоков. Результаты могут меняться от системы к системе, однако практические эксперименты показали, что значения счетчиков, лежащие в интервале от 2000 до 10000, являются оптимальными.

9.3. Используя функцию TimedMutualExclusion, которая находится на Web-сайте книги, проведите эксперименты путем варьирования длительности периодов задержки и количества точек "засыпания" потоков.

9.4. Для ограничения количества выполняющихся потоков в функции TimedMutualExclusion наряду с другими средствами используется дросселирование семафоров. Поэкспериментируйте со значениями счетчиков как на однопроцессорных, так и на SMP-системах. Воспроизводят ли полученные вами результаты те, о которых сообщалось ранее в настоящей главе?

9.5. Воспользуйтесь методикой дросселирования семафоров в программе statsMX (statsCS.c, statsMX.c).

9.6. *Упражнение повышенной сложности.* Все ли из четырех разновидностей программы работают корректно, если не обращать внимания на производительность, на SMP-системах? Исследуйте результаты, получаемые при большом количестве потоков. Запустите программы на SMP-системах, работающих под управлением Windows 2000 или Windows Server 2003. Проявляются ли при этом проблемы "разрыва слов" ("word tearing") и "конфликтов строки кэша" ("cache line conflict"), описанных ранее в настоящей главе, а также в [6]? Для воспроизведения указанных проблем вам может потребоваться использование 16-битовых (тип данных short integer) счетчиков.

9.7. Используйте родство процессора в качестве средства улучшения производительности, внеся необходимые изменения в программы, о которых шла речь в настоящей главе.

9.8. Постарайтесь определить, оказывает ли использование гиперпотоков влияние на производительность приложений. Средства гиперпоточной обработки обеспечиваются, например, процессором Intel Xeon.

ГЛАВА 10
Усовершенствованные методы синхронизации потоков

В предыдущей главе были описаны проблемы производительности, возникающие в Windows, и способы их преодоления в реалистичных ситуациях. В главе 8 обсуждался ряд простых задач, требующих привлечения объектов синхронизации. В настоящей главе на основании идей, изложенные в главах 8 и 9, решаются задачи, которые также встречаются в реальной практике, но отличаются большей сложностью.

Первое, что нам предстоит сделать — это объединить два или более объекта синхронизации вместе с данными для создания сложного объекта синхронизации. Наиболее полезной комбинацией такого рода является модель переменных условий (condition variable model), включающая мьютекс и одно или несколько событий. Указанная модель играет весьма существенную роль в самых различных практических ситуациях, поскольку многие серьезные программные дефекты, обусловленные влиянием состязательности, проявляются именно тогда, когда объекты синхронизации Windows, особенно события, используются программистами неправильно. События имеют сложную природу и ведут себя по-разному в зависимости от того, какой именно из описанных в табл. 8.1 вариантов используется, и поэтому следует придерживаться определенных правил, устанавливаемых хорошо изученными моделями.

В последующих разделах показано, как систематизировать управление запуском и отменой выполнения каждого из совместно работающих потоков при помощи асинхронного вызова процедур (Asynchronous Procedure Calls, APC).

Другие проблемы производительности обсуждаются по мере необходимости.

Модель переменных условий и свойства безопасности

Многопоточные программы намного легче разрабатывать, делать их более понятными и сопровождать, если использовать известные, хорошо разработанные методики и модели. Эти вопросы уже обсуждались в главе 7, в которой для создания полезной концептуальной основы, позволяющей понять принципы работы многопоточных программ, были введены модель "хозяин/рабочий" ("boss/worker") и модель рабочей группы (work crew model). Понятие критических участков кода (critical sections) играет существенную роль при использовании мьютексов, а определение инвариантов (invariants) используемых структур данных также может принести немалую пользу. Наконец, даже для дефектов существуют свои модели, как это было показано на примере взаимной блокировки (deadlock) потоков.

Примечание

Компания Microsoft разработала собственный набор моделей, таких как апартаментная модель (apartment model) или модель свободных потоков (free threading). Эта терминология чаще всего встречается в технологии COM и кратко обсуждается в конце главы 11.

Совместное использование событий и мьютексов

Далее показано, как обеспечить совместное использование мьютексов и событий путем обобщения программы 8.2, представляющей описанную ниже ситуацию, с которой нам еще не раз предстоит столкнуться. *Примечание.* Это обсуждение в равной степени применимо как к мьютексам, так и к объектам CRITICAL_SECTION.

• Как мьютекс, так и событие связываются с блоком сообщений или иной структурой данных.

• Мьютекс определяет критический участок кода для доступа к объекту структуры данных.

• Событие используется для того, чтобы сигнализировать о появлении нового сообщения.

• Обобщая, можно утверждать, что мьютекс обеспечивает соблюдение условий, определяемых инвариантами объекта (то есть поддерживает свойства безопасности), а событие сигнализирует о том, что состояние объекта изменилось (например, сообщение было добавлено в буфер сообщений или удалено из него), и он мог перейти в известное состояние (например, в буфере сообщений присутствует, по крайней мере, одно сообщение).

• Один поток (в программе 8.2 — поток производителя) блокирует структуру данных, изменяет состояние объекта путем создания нового сообщения и применяет функции SetEvent или PulseEvent к событию, связанному с появлением нового сообщения.

• По крайней мере, один поток из числа остальных (в данном примере — поток потребителя) ожидает наступления события, сигнализирующего о том, что объект достиг требуемого состояния. Ожидание должно выполняться за пределами критического участка кода, чтобы поток производителя мог иметь доступ к объекту.

• Кроме того, поток потребителя может блокировать мьютекс, проверить состояние объекта (например, поступило ли в буфер новое сообщение) и отказаться от ожидания события, если объект уже находится в требуемом состоянии.

Модель переменных условий

А теперь давайте объединим все это в едином фрагменте кода, представляющем то, что мы будем называть *моделью переменных условий* (condition variable model, CV model), которая может существовать в виде *сигнальной* (signal) и *широковещательной* (broadcast) моделей. В первых примерах будет использована широковещательная модель. Результат представляет собой программную модель, с которой мы будем работать еще не один раз, и которая может быть использована для решения широкого круга задач синхронизации. Для удобства изложения примеры сформулированы в терминах задачи производителя и потребителя.

Обсуждение может показаться вам несколько абстрактным, однако, поняв суть методики, вы сможете решать многие задачи синхронизации, справиться с которыми без наличия хорошей модели было бы очень трудно.

В упомянутом фрагменте кода имеется несколько ключевых элементов.

• Структура данных типа STATE_TYPE, в которой содержатся все данные, или *переменные состояний* (state variables), такие как сообщения, контрольные суммы и счетчики, используемые в программе 8.2.

• Мьютекс и одно или более событий, связанных с этой структурой данных и обычно входящих в ее состав.

• Одна или несколько булевых функций, предназначенных для вычисления *предикатов переменных условий* (condition variable predicates), представляющих собой условия (состояния), наступления которых может ожидать поток. Например, в качестве предикатов могут использоваться следующие условия: "готово новое сообщение", "имеется свободное место в буфере", "очередь не пуста". Можно связывать с каждым предикатом переменной условия отдельное событие, но возможно также использование одного события для представления изменения состояния или же для представления комбинации нескольких предикатов (получаемой посредством применения операции логического "или"). В последнем случае для определения фактического состояния должны проверяться возвращаемые значения отдельных предикативных функций при блокированном мьютексе. Если предикат (логическое выражение) является простым, необходимость в использовании отдельной функции отпадает.

Эти принципы используются потоками производителя и потребителя в приведенном ниже фрагменте кода, включающем единственное событие и предикат переменной условия (реализованный с помощью функции cvp, которая здесь не представлена). В данном примере принимается, что если поток производителя сигнализирует о достижении требуемого состояния, то должны быть освобождены сразу несколько потоков, откуда следует, что сигнал должен рассылаться всем ожидающим потокам потребителя. Так, сигналом, соответствующим созданию потоком производителя нескольких сообщений, может служить увеличение значения счетчика сообщений. Во многих случаях вам может требоваться освобождение только одного потока, что обсуждается после приведенного ниже фрагмента кода.

Этот код ориентирован на работу под управлением Windows 9x и всех версий Windows NT. Для упрощения решения впоследствии в нем будет использована функция SignalObjectAndWait.

Примечание и предостережение

В данном примере намеренно и вполне осознанно используется функция PulseEvent, хотя некоторые авторы, а кое-где и документация Microsoft (см. замечания в соответствующем разделе MSDN), этого делать не рекомендуют. Причины нашего выбора будут ясны из последующего обсуждения и подкреплены примерами, а читателю предлагается решить (корректно) эту задачу, используя функцию SetEvent.

```
typedef struct _state_t {
```

```
    HANDLE Guard; /* Мьютекс, защищающий объект. */
     HANDLE CvpSet; /* Вручную сбрасываемое событие — выполняется условие,
определяемое предикатом cvp(). */
    … другие переменные условий …
    /* Структура состояния, содержащая счетчики, контрольные суммы и прочее. */
    struct STATE_VAR_TYPE StateVar;
    } STATE_TYPE State;
   …
   /* Инициализировать состояние, создавая мьютекс и событие. */
   …
   /* Поток ПРОИЗВОДИТЕЛЯ, который изменяет состояние. */
   WaitForSingleObject(State.Guard, INFINITE);
   /* Изменить состояние таким образом, чтобы выполнялось условие, */
   /* определяемое предикатом CV. */
   /* Пример: к данному моменту подготовлено одно или несколько сообщений.*/
   State.StateVar.MsgCount += N;
   PulseEvent(State.CvpSet);
   ReleaseMutex(State.Guard);
   /* Конец интересующей нас части кода потока производителя. */
   …
   /* Ожидание определенного состояния функцией потока ПОТРЕБИТЕЛЯ. */
   WaitForSingleObject(State.Guard, INFINITE);
   while (!cvp(&State)) {
    ReleaseMutex(State.Guard);
    WaitForSingleObject(State.CvpSet, TimeOut);
    WaitForSingleObject(State.Guard, INFINITE);
   }
   /* Теперь этот поток владеет мьютексом, и выполняется условие, */
   /* определяемое предикатом cvp(&State). */
   /* Предпринять соответствующее действие, возможно, изменяя состояние.*/
   …
   ReleaseMutex(State.Guard);
   /* Конец интересующей нас части кода потока потребителя. */
```

Комментарии по поводу модели переменных условий

В приведенном выше фрагменте кода очень важная роль принадлежит циклу в той части кода, которая соответствует потребителю. Этот цикл включает три операции: 1) освобождение мьютекса, заблокированного до входа в цикл; 2) ожидание события; 3) повторное блокирование мьютекса. Как будет показано далее, *использование конечного интервала ожидания события является весьма существенным*.

Потоки Pthreads в том виде, в каком они реализованы во многих системах UNIX и других системах, сочетают эти три операции в одной функции — pthread_cond_wait, объединяющей мьютекс и переменную условия (которая аналогична, но не идентична событиям Windows). Именно поэтому и используется термин *модель переменных условий.* Существует также версия этой функции, допускающая использование конечных интервалов ожидания событий.

Что немаловажно, в Pthreads первые две операции (освобождение мьютекса и ожидание события) реализуются посредством вызова одной функции как одна атомарная операция, так что никакой другой поток не сможет вклиниться раньше, чем начнется выполнения вызывающим потоком функции ожидания наступления события (или выполнения условия).

Проектировщики Pthreads сделали мудрый выбор: единственный способ организовать ожидания выполнения условия, определенного для переменной условия, — это использование одной из двух указанных выше функций (с конечным и неопределенным интервалами

ожидания), так что переменная условия должна всегда использоваться вместе с мьютексом. Windows вынуждает вас использовать для этой цели два или три отдельных вызова функций, и вы сами должны проследить за тем, чтобы все было сделано правильно, иначе вам не избежать проблем.

Помимо того, что это упрощает разработку программ и является существенно необходимым в случае использования потоков Pthreads, есть еще одна причина, по которой следует изучать модель CV, заключающаяся в том, что именно эта модель используется рядом сторонних производителей для реализации классов потоков и объектов синхронизации, не зависящих от ОС. Владея изложенным в этой книге материалом, вы сможете очень быстро разобраться в особенностях этих реализаций.

Примечание

В версии Windows NT 4.0 была введена новая функция — SignalObjectAndWait (SOAW), которая выполняет упомянутые два шага атомарным образом. В дальнейших примерах программ предполагается, что эта функция доступна, и она будет использоваться, а это означает, что под управлением Windows 9x такие программы выполняться не смогут. Тем не менее, на стадии ознакомления с моделью CV функция SOAW не применяется, чтобы сделать более понятной мотивировку необходимости ее использования впоследствии, а на Web-сайте книги приведены альтернативные варианты реализации некоторых примеров, в которых вместо мьютексов используются объекты CS. (Функцию SOAW нельзя применять вместе с объектами CS.) О значительных преимуществах функции SignalObjectAndWait в отношении производительности свидетельствуют данные, представленные в приложении В (табл. В.5).

Использование модели переменных условий

Модель переменных условий при правильной ее реализации работает следующим образом:

• Поток производителя блокирует мьютекс, изменяет состояние, применяет к событию функцию PulseEvent, когда это необходимо, и разблокирует мьютекс. Например, функция PulseEvent может вызываться в случае готовности одного или нескольких сообщений.

• Функция PulseEvent должна применяться к событию при блокированном мьютексе, чтобы никакой другой поток не мог изменить объект, что могло бы сделать недействительным условие, определенное предикатом.

• Поток потребителя тестирует предикат переменной условия при блокированном мьютексе. Если условие, выраженное предикатом, выполняется, выполнять функцию ожидания нет никакой необходимости.

• Если же условие, выраженное предикатом, не выполняется, поток потребителя должен разблокировать мьютекс до выполнения ожидания события. Если этого не сделать, то никакой поток вообще не сможет изменить состояние и установить событие.

• Интервал ожидания события должен быть конечным, чтобы обеспечить правильную обработку в том случае, если поток производителя применит к событию функцию PulseEvent в промежутке времени между освобождением мьютекса (шаг 1) и выполнением ожидания события (шаг 2). Таким образом, без использования *конечного* интервала ожидания сигнал мог бы потеряться, что является еще одним примером проявления проблемы состязательности. К потере сигналов могут приводить и асинхронные вызовы процедур, описанные далее в этой

главе. Используемый в приведенном выше фрагменте кода интервал ожидания является настраиваемым параметром. (С комментариями по поводу оптимальных значений этого параметра вы можете ознакомиться, обратившись к приложению В.)

• По завершении ожидания события поток потребителя всегда повторно проверяет выполнение условия, определенного предикатом. Среди прочих других причин, это необходимо делать с учетом того, что интервал ожидания может просто исчерпаться. Кроме того, за это время состояние также могло измениться. Например, поток производителя мог сгенерировать два сообщения, а затем освободить три ожидающих потока потребителя, в результате чего один из потребителей проверит состояние, определит, что сообщения отсутствуют, и продолжит выполнение ожидания. Наконец, повторная проверка предиката необходима для защиты от ложного пробуждения потоков, которое могло бы произойти в результате того, что поток установит событие в сигнальное состояние или применит к нему функцию PulseEvent без предварительного блокирования мьютекса.

• После выхода из цикла поток потребителя всегда сохраняет за собой право владения мьютексом, независимо от того, выполнялось или не выполнялось тело цикла.

Разновидности модели переменных условий

Прежде всего, обратите внимание на то, что в предшествующем фрагменте кода используется сбрасываемое вручную событие и вызывается функция PulseEvent, а не функция SetEvent. Является ли такой выбор корректным и возможен ли иной способ использования события? Ответ на оба эти вопросы является положительным.

Вернувшись к табл. 8.1, можно увидеть, что сбрасываемые вручную события характеризуются освобождением *нескольких потоков.* Это именно так в случае нашего примера, в котором генерируются несколько сообщений и существует несколько потоков потребителя, и все они должны быть оповещены о произошедших изменениях. В то же время, если поток производителя создает всего лишь одно сообщение и имеется несколько потоков потребителя, то событие должно быть автоматически сбрасываемым, а поток производителя должен вызывать функцию SetEvent, чтобы обеспечить освобождение только одного потока. В этом случае мы имеем дело не с сигнальной разновидностью модели CV, а с широковещательной. При этом по-прежнему остается существенным, чтобы освобожденный поток потребителя, который приобретает права владения мьютексом, изменил объект для указания того, что доступные сообщения отсутствуют (то есть, что условие, определяемое предикатом переменной условия, уже не выполняется).

Из четырех возможных комбинаций, указанных в табл. 8.1, для модели переменных условий важны только две. Что касается двух других комбинаций, то в силу конечности интервала ожидания эффект комбинации "автоматически сбрасываемое событие/PulseEvent" будет тем же, что и комбинации "автоматически сбрасываемое событие/SetEvent" (сигнальная модель CV), однако зависимость от длительности интервала ожидания приведет к снижению характеристик реактивности.

Использование же комбинации "вручную сбрасываемое событие/PulseEvent" приведет к появлению ложных сигналов (от которых, правда, можно защититься проверкой предикатов переменных условий), поскольку событие должно быть сброшено каким-либо из потоков, а до сброса события потоки будут состязаться между собой.

Подводя итоги, можно сделать вывод, что комбинация "автоматически сбрасываемое событие/SetEvent" представляет собой сигнальную модель CV, в которой освобождается

единственный из ожидающих потоков, а комбинация "вручную сбрасываемое событие/PulseEvent" — широковещательную модель CV, в которой освобождаются все ожидающие потоки. Для потоков Pthreads существуют те же различия, но использование конечных интервалов ожидания событий для широковещательной модели в данном случае не требуется, тогда как в Windows этот фактор является весьма существенным, поскольку освобождение мьютекса и ожидание события не выполняются атомарно, то есть за одну операцию. В то же время, введение функции SignalObjectAndWait меняет эту ситуацию.

Пример предиката переменной условия

Рассмотрим следующий предикат переменной условия:

```
State.StateVar.Count >= K;
```

В данном случае поток потребителя будет ожидать до тех пор, пока значение счетчика не станет достаточно большим, и поток производителя может увеличивать это значение на произвольную величину. Отсюда, например, становится понятным, как можно реализовать сложные семафоры; вспомните, что обычные семафоры не допускают атомарного выполнения нескольких функций ожидания. В данном же случае поток потребителя может просто уменьшить значение счетчика на К единиц после выхода из цикла, но перед тем, как освободить мьютекс.

Заметьте, что в данном случае подходит широковещательная модель CV, поскольку один поток производителя может увеличить значение счетчика и тем самым разрешить выполнение нескольким, но не всем ожидающим потокам потребителя.

Семафоры и модель переменных условий

В некоторых случаях уместнее использовать не события, а семафоры, преимущество которых заключается в том, что они позволяют указывать точное количество потоков, которые необходимо освободить. Например, если бы было известно, что каждый из потоков потребителя может получить только одно сообщение, то поток производителя мог бы вызвать функцию ReleaseSemaphore, используя в качестве параметра точное количество сгенерированных сообщений. Однако в общем случае потоку производителя ничего не известно о том, каким образом отдельные потоки потребителя изменят структуру переменной состояния, и поэтому модель переменных условий применима для решения более широкого круга задач.

Модель CV обладает достаточно мощными возможностями, которых хватает для реализации семафоров. Как уже отмечалось ранее, в основе этого метода лежит определение предиката, эквивалентного утверждению: "значение счетчика является ненулевым", и создание структуры состояния, содержащей текущее значение счетчика и его максимально допустимое значение. В упражнении 10.11 представлено завершенное решение, позволяющее манипулировать функциями ожидания путем изменения значений счетчика на несколько единиц одной операцией. Создание семафоров для потоков Pthreads не предусмотрено, поскольку переменные условий предоставляют достаточно широкие возможности.

Использование функции SignalObjectAndWait

Цикл, выполняемый потоком потребителя в предшествующем фрагменте кода, играет очень

важную роль в модели CV, поскольку в нем выполняется ожидание изменения состояния, а затем проверяется, является ли состояние именно тем, какое требуется. Последнее условие может не выдерживаться, если событие оказывается слишком *обобщенным,* указывая, например, только на сам факт изменения состояния, а не на характеристики такого изменения. К тому же, другие потоки могут дополнительно изменить состояние, например, очистить буфер сообщений. Упомянутый цикл требовал выполнения двух функций ожидания и одной функции освобождения мьютекса, как показано ниже.

```
while (!cvp(&State)) {
 ReleaseMutex(State.Guard);
 WaitForSingleObject(State.CvpSet, TimeOut);
 WaitForSingleObject(State.Guard, INFINITE);
}
```

Использование конечного интервала ожидания (time-out) при выполнении первой функции ожидания (ожидание события) требуется здесь для того, чтобы избежать потери сигналов или возникновения других вероятных проблем. Этот код будет работать как под управлением Windows 9x, так и под управлением Windows NT 3.5 (еще одна устаревшая версия Windows), а предыдущий фрагмент кода сохранит свою работоспособность и в том случае, если мьютексы заменить объектами CS.

Однако в случае Windows NT 5.x (XP, 2000 и Server 2003) и даже Windows NT 4.0 мы можем использовать функцию SignalObjectAndWait — важный элемент усовершенствования, который избавляет от необходимости применения конечных интервалов ожидания и объединяет освобождение мьютекса и ожидание события. При этом кроме явного упрощения программы, производительность в общем случае повышается, что объясняется устранением системного вызова и отсутствием необходимости в настройке длительности интервала ожидания.

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

Эта функция, при вызове которой используются дескрипторы, указывающие соответственно на мьютекс и событие, упрощает цикл потребителя. Интервал ожидания здесь отсутствует, поскольку вызывающий поток переходит к ожиданию второго дескриптора *сразу же* после того, как первый дескриптор переходит в сигнальное состояние (что в данном случае означает освобождение мьютекса). Перевод объекта в сигнальное состояние и переход к ожиданию осуществляются атомарным образом, то есть за одну операцию, так что никакой другой поток не может сигнализировать о наступлении события в течение промежутка времени между освобождением мьютекса вызывающим потоком и ожидания потоком события, на которое указывает второй дескриптор. Тогда упрощенный цикл потребителя приобретает следующий вид:

```
while (!cvp(&State)) {
 SignalObjectAndWait(State.Guard, State.CvpSet, INFINITE, FALSE);
 WaitForSingleObject (State.Guard, INFINITE);
}
```

Значением последнего аргумента этой функции, bAlertable, в данном случае является FALSE, однако в последующих разделах, посвященных рассмотрению APC, он будет полагаться равным TRUE.

Вообще говоря, оба дескриптора могут указывать на любые подходящие объекты синхронизации. В то же время, использовать объект CRITICAL_SECTION в качестве объекта, сигнальное состояние которого отслеживается, нельзя, поскольку допустимыми являются только объекты ядра.

Функция SignalObjectAndWait применяется во всех примерах программ, представленных как в книге, так и на Web-сайте, хотя на Web-сайте находятся и другие варианты решений, о которых будет говориться в тексте. Если программа должна выполняться под управлением Windows 9x, то следует заменить эту функцию парой функций "сигнал/ожидание", как в первоначально приведенном фрагменте кода, и обязательно использовать конечный интервал ожидания.

В разделе, посвященном APC, представлены различные методы отправки сигналов ожидающим потокам, обеспечивающие получение сигналов только определенными потоками, тогда как в случае событий простых способов, позволяющих контролировать, каким потокам направляются сигналы, не существует.

Пример: объект порогового барьера

Предположим, вам необходимо, чтобы рабочие потоки оставались в состоянии ожидания и не выполнялись до тех пор, пока количество таких потоков не станет достаточным для образования рабочей группы, способной выполнить нужную работу. Как только количество потоков достигает порогового значения, все ожидающие рабочие потоки начинают выполняться, а появляющиеся впоследствии дополнительные рабочие потоки будут выполняться без ожидания. Эту задачу можно решить путем создания сложного объекта порогового барьера (threshold barrier compound object).

В программах 10.1 и 10.2 представлена реализация трех функций, поддерживающих сложный объект барьера. Две из этих функций, CreateThresholdBarrier и CloseThresholdBarrier, управляют переменными THB_HANDLE, аналогичными дескрипторам, которые на протяжении всего времени применялись нами вместе с объектами ядра. Пороговое количество потоков является параметром функции CreateThresholdBarrier.

Программа 10.1 представляет соответствующую часть заголовочного файла, SynchObj.h, тогда как программа 10.2 — реализацию трех упомянутых функций. Обратите внимание, что объект барьера содержит мьютекс, событие, счетчик и пороговое значение. Предикат переменной условия документирован в заголовочном файле, а именно, событие должно устанавливаться только тогда, когда значение счетчика достигает или становится больше порогового значения.

Программа 10.1. SynchObj.h: часть 1 — объявления объекта порогового барьера

```
/* Глава 10. Сложные объекты синхронизации. */
#define CV_TIMEOUT 50 /* Настраиваемый параметр для модели CV. */
/* ОБЪЕКТ ПОРОГОВОГО БАРЬЕРА — ОПРЕДЕЛЕНИЕ ТИПА И ПРОТОТИПЫ ФУНКЦИЙ. */
typedef struct THRESHOLD_BARRIER_TAG { /* Пороговый барьер. */
 HANDLE b_guard; /* Мьютекс для объекта. */
 HANDLE b_broadcast; /* Вручную сбрасываемое событие: b_count >= b_threshold.*/
 volatile DWORD b_destroyed; /* Установить после закрытия. */
 volatile DWORD b_count; /* Количество потоков до достижения барьера. */
 volatile DWORD b_threshold; /* Пороговый барьер. */
} THRESHOLD_BARRIER, *THB_HANDLE;

/* Коды ошибок. */
#define SYNCH_OBJ_NOMEM 1 /* Невозможно выделить ресурсы. */
#define SYNCH_OBJ_BUSY 2 /* Объект используется и не может быть закрыт. */
#define SYNCH_OBJ_INVALID 3 /* Объект более не является действительным. */
DWORD CreateThresholdBarrier(THB_HANDLE *, DWORD /* Порог. */);
DWORD WaitThresholdBarrier(THB_HANDLE);
DWORD CloseThresholdBarrier(THB_HANDLE);
```

Рассмотрим теперь предложенную в программе 10.2 реализацию трех функций. На Web-сайте книги находится тестовая программа testTHB. Обратите внимание на уже знакомый вам цикл проверки переменной условия в функции ожидания WaitThresholdBarrier. Кроме того, эта функция не только ожидает наступления события, но и переводит объект события в сигнальное состояние с помощью функции PulseEvent. Предыдущее обсуждение модели "производитель/потребитель" предполагало использование отдельных функций потоков.

Наконец, в данном случае предикат переменной условия обладает последействием. Как только условие выполнилось, оно будет выполняться и в дальнейшем, что исключает

возможность перевода объекта события в сигнальное состояние более одного раза.

Программа 10.2. ThbObject.c: реализация объекта порогового барьера

```
/* Глава 10. Программа 10.2. */
/* Библиотека сложных объектов синхронизации на основе порогового барьера.*/
#include "EvryThng.h"
#include "synchobj.h"
/*********************************/
/*   ОБЪЕКТЫ ПОРОГОВОГО БАРЬЕРА   */
/*********************************/
DWORD CreateThresholdBarrier(THB_HANDLE *pthb, DWORD b_value) {
 THB_HANDLE hthb;
  /* Инициализация объекта барьера. Вариант программы с полной проверкой ошибок
находится на Web-сайте. */
 hthb = malloc(sizeof(THRESHOLD_BARRIER));
 hthb->b_guard = CreateMutex(NULL, FALSE, NULL);
  hthb->b_broadcast  =  CreateEvent(NULL,  FALSE  /*  Автоматически  сбрасываемое
событие. */, FALSE, NULL);
 hthb->b_threshold = b_value;
 hthb->b_count = 0;
 hthb->b_destroyed = 0;
 *pthb = hthb;
 return 0;
}

DWORD WaitThresholdBarrier(THB_HANDLE thb) {
  /* Ожидать, пока заданное количество потоков не достигнет порога, а затем
установить событие. */
 if (thb->b_destroyed == 1) return SYNCH_OBJ_INVALID;
 WaitForSingleObject(thb->b_guard, INFINITE);
 thb->b_count++; /* Появился новый поток. */
 while (thb->b_count < thb->b_threshold) {
  SignalObjectAndWait(thb->b_guard, thb->b_broadcast, INFINITE, FALSE);
  WaitForSingleObject(thb->b_guard, INFINITE);
 }
 PulseEvent(thb->b_broadcast) ;
 /* Широковещательная модель CV, освобождение всех ожидающих потоков. */
 ReleaseMutex(thb->b_guard);
 return 0;
}

DWORD CloseThresholdBarrier(THB_HANDLE thb) {
 /* Уничтожить мьютекс и событие объекта барьера. */
 /* Убедиться в отсутствии потоков, ожидающих объект. */
 if (thb->b_destroyed == 1) return SYNCH_OBJ_INVALID;
 WaitForSingleObject(thb->b_guard, INFINITE);
 if (thb->b_count < thb->b_threshold) {
  ReleaseMutex(thb->b_guard);
  return SYNCH_OBJ_BUSY;
 }
 ReleaseMutex(thb->b_guard);
 CloseHandle(thb->b_guard);
 CloseHandle(thb->b_broadcast);
 free(thb);
 return 0;
}
```

Комментарии по поводу реализации объекта порогового барьера

Возможности реализованного выше объекта порогового барьера в интересах простоты были намеренно ограничены. Вообще говоря, было бы желательно эмулировать объекты Windows следующим образом:

• Разрешив объектам иметь атрибуты защиты (глава 15).

• Разрешив присвоение имен объектам.

• Допуская наличие у одного объекта нескольких "дескрипторов" и не уничтожая их до тех пор, пока счетчик ссылок не станет равным 0.

• Разрешив совместное использование объекта несколькими процессами.

На Web-сайте доступна полная реализация одного из таких объектов — сложного (multiple-wait) семафора, допускающего изменение счетчика семафора сразу на несколько единиц, которая использует методы, применимые по отношению к любому из объектов, рассматриваемых в данной главе.

Объект очереди

До сих пор мы связывали с каждым мьютексом только одно событие, но в общем случае могут существовать несколько предикатов переменных условий. Например, в случае очереди, действующей по принципу "первым пришел, первым ушел" (first in first out, FIFO), поток, который пытается удалить элемент из очереди, должен дождаться события, указывающего на то, что очередь не является пустой, а поток, помещающий элемент в очередь, должен дождаться события, указывающего на то, что очередь не является заполненной. Решение заключается в предоставлении двух событий — по одному для каждого условия.

В программе 10.3 представлены необходимые объявления объекта очереди и его функций. В объявлениях намеренно применяется стиль, отличающийся от того, который принят в Windows и который мы использовали до сих пор. Эта программа была получена преобразованием ее первоначального варианта, реализованного в UNIX на основе потоков Pthreads, чем и объясняется происхождение использованного нами стиля. Точно так же и вы можете наследовать тот или иной стиль или определить собственный, который соответствует вашему вкусу или принятым в вашей организации требованиям. В упражнении 10.7 вам предлагается преобразовать приведенный стиль к стилю Windows.

Программы 10.4 и 10.5 представляют функции очереди и программу, которая их использует.

Программа 10.3. SynchObj.h: часть 2 — объявления объекта очереди

```
    /* Объявления структуры обычной ограниченной синхронизированной очереди.*/
    /* Очереди закольцованы и реализованы в виде массивов с индексацией */
    /* последнего и первого сообщений. */
    /* Кроме того, каждая очередь содержит защитный мьютекс и */
    /* переменные условий "очередь не пуста" и "очередь не заполнена". */
    /* Наконец, имеется указатель массива сообщений произвольного типа. */
    typedef struct queue_tag { /* Универсальная очередь. */
     HANDLE q_guard; /* Защита блока сообщения. */
     HANDLE q_ne; /* Очередь не пуста. Вручную сбрасываемое событие. (Автоматически
сбрасываемое событие для "сигнальной модели".) */
      HANDLE q_nf; /* Очередь не заполнена. Вручную сбрасываемое событие.
(Автоматически сбрасываемое событие для "сигнальной модели".) */
     volatile DWORD q_size; /* Максимальный размер очереди. */
     volatile DWORD q_first; /* Индекс первого сообщения. */
     volatile DWORD q_last; /* Индекс последнего сообщения. */
      volatile DWORD q_destroyed; /* Получатель сообщений очереди завершил
выполнение. */
     PVOID msg_array; /* Массив q_size сообщений. */
    } queue_t;

    /* Функции управления очередью. */
    DWORD q_initialize(queue_t *, DWORD, DWORD);
    DWORD q_destroy(queue_t *);
    DWORD q_destroyed(queue_t *);
    DWORD q_empty(queue_t *);
    DWORD q_full(queue_t *);
    DWORD q_get(queue_t *, PVOID, DWORD, DWORD);
    DWORD q_put(queue_t *, PVOID, DWORD, DWORD);
    DWORD q_remove(queue_t *, PVOID, DWORD);
    DWORD q_insert(queue_t *, PVOID, DWORD);
```

В программе 10.4 представлены такие функции, как q_initialize и q_get, прототипы которых описаны в конце программы 10.3. Обратите внимание, что функции q_get и q_put обеспечивают синхронизацию доступа, а функции q_remove и q_insert, которые вызываются первыми двумя функциями, сами по себе не являются синхронизированными и могут быть использованы в однонитевых программах. В первых двух функциях предусмотрена возможность использования конечных интервалов ожидания, что требует незначительного расширения модели переменных условий.

q_empty и q_full — две другие важные функции, которые используются для реализации предикатов переменных условий.

Данная реализация использует функцию PulseEvent и вручную сбрасываемые события (широковещательная модель), так что все события уведомляются о том, что очередь не пуста или не заполнена.

Замечательной особенностью этой реализации является симметрия функций q_get и q_put. Обратите внимание хотя бы на то, как в этих функциях используются предикаты пустой и заполненной очередей или события. Подобная простота не только восхитительна сама по себе, но и имеет благоприятные практические последствия, облегчающие написание, понимание и сопровождение программы, и все это было достигнуто за счет использования модели переменных условий.

Наконец, те, кто программирует на C++, легко сообразят, что приведенный код может быть использован для создания класса синхронизированной очереди; именно это вам и предлагается сделать в упражнении 10.8.

Программа 10.4. QueueObj.c: функции управления очередью

```
/* Глава 10. QueueObj.c. */
/* Функции очереди */
#include "EvryThng.h"
#include "SynchObj.h"
/* Функции управления конечной ограниченной очередью. */
DWORD q_get(queue_t *q, PVOID msg, DWORD msize, DWORD MaxWait) {
 if (q_destroyed (q)) return 1;
 WaitForSingleObject(q->q_guard, INFINITE);
 while (q_empty(q)) {
  SignalObjectAndWait(q->q_guard, q->q_ne, INFINITE, FALSE);
  WaitForSingleObject(q->q_guard, INFINITE);
 }
 /* Удалить сообщение из очереди. */
 q_remove(q, msg, msize);
  /* Сигнализировать о том, что очередь не заполнена, поскольку мы удалили
сообщение. */
 PulseEvent(q->q_nf);
 ReleaseMutex(q->q_guard);
 return 0;
}

DWORD q_put(queue_t *q, PVOID msg, DWORD msize, DWORD MaxWait) {
 if (q_destroyed(q)) return 1;
 WaitForSingleObject(q->q_guard, INFINITE);
 while(q_full(q)) {
  SignalObjectAndWait(q->q_guard, q->q_nf, INFINITE, FALSE);
  WaitForSingleObject(q->q_guard, INFINITE);
 }
```

```
    /* Поместить сообщение в очередь. */
    q_insert(q, msg, msize);
    /* Сигнализировать о том, что очередь не пуста; мы вставили сообщение.*/
    PulseEvent (q->q_ne);
    /* Широковещательная модель CV. */
    ReleaseMutex(q->q_guard);
    return 0;
   }

   DWORD q_initialize(queue_t *q, DWORD msize, DWORD nmsgs) {
    /* Инициализация очереди, включая ее мьютекс и события. */
    /* Выделить память для всех сообщений. */
    q->q_first = q->q_last = 0;
    q->q_size = nmsgs;
    q->q_destroyed = 0;
    q->q_guard = CreateMutex(NULL, FALSE, NULL);
    q->q_ne = CreateEvent(NULL, TRUE, FALSE, NULL);
    q->q_nf = CreateEvent(NULL, TRUE, FALSE, NULL);
    if ((q->msg_array = calloc(nmsgs, msize)) == NULL) return 1;
    return 0; /* Ошибки отсутствуют. */
   }

   DWORD q_destroy(queue_t *q) {
    if (q_destroyed(q)) return 1;
    /* Освободить все ресурсы, созданные вызовом q_initialize. */
    WaitForSingleObject(q->q_guard, INFINITE);
    q->q_destroyed = 1;
    free(q->msg_array);
    CloseHandle(q->q_ne);
    CloseHandle(q->q_nf);
    ReleaseMutex(q->q_guard);
    CloseHandle(q->q_guard);
    return 0;
   }

   DWORD q_destroyed(queue_t *q) {
    return (q->q_destroyed);
   }

   DWORD q_empty(queue_t *q) {
    return (q->q_first == q->q_last);
   }

   DWORD q_full(queue_t *q) {
     return ((q->q_last - q->q_first) == 1 || (q->q_first == q->q_size-l && q-
>q_last == 0));
   }

   DWORD q_remove(queue_t *q, PVOID msg, DWORD msize) {
    char *pm;
    pm = (char *)q->msg_array;
    /* Удалить наиболее давнее ("первое") сообщение. */
    memcpy(msg, pm + (q->q_first * msize), msize);
    q->q_first = ((q->q_first + 1) % q->q_size);
    return 0; /* Ошибки отсутствуют. */
   }

   DWORD q_insert(queue_t *q, PVOID msg, DWORD msize) {
```

```
    char *pm;
    pm = (char *)q->msg_array;
    /* Добавить новое ("последнее") сообщение. */
    if (q_full(q)) return 1; /* Ошибка - очередь заполнена. */
    memcpy(pm + (q->q_last * msize), msg, msize);
    q->q_last = ((q->q_last + 1) % q->q_size);
    return 0;
}
```

Комментарии по поводу функций управления очередью с точки зрения производительности

В приложении В представлены данные, характеризующие производительность программы 10.5, в которой используются функции управления очередью. Приведенные ниже замечания по поводу различных факторов, которые могут оказывать влияние на производительность, основываются на этих данных. Программные коды упоминаемых ниже альтернативных вариантов реализации находятся на Web-сайте книги.

• В данной реализации используется широковещательная модель ("вручную сбрасываемое событие/PulseEvent"), обеспечивающая поддержку общего случая, когда один поток может запрашивать или создавать несколько сообщений. Если такая общность не требуется, можно использовать сигнальную модель ("автоматически сбрасываемое событие/SetEvent"), которая, к тому же, обеспечит значительно более высокую производительность, поскольку для тестирования предиката будет освобождаться только один поток. На Web-сайте находится файл QueueObj_Sig.c, содержащий исходный код, в котором вместо широковещательной модели используется сигнальная модель.

• Использование для защиты объекта очереди объекта CRITICAL_SECTION вместо мьютекса также может привести к повышению производительности. Однако в этом случае вместо функции SignalObjectAndWait следует использовать функцию EnterCriticalSection с последующим ожиданием события. Этот альтернативный подход иллюстрируется двумя файлами — QueueObjCS.c и QueueObjCS_Sig.c, находящимися на Web-сайте книги.

• На Web-сайте находятся два других файла с исходными кодами — QueueObj_noSOAW.c и QueueObjSig_noSOAW.c, в которых функция SignalObjectAndWait не используется и которые обеспечивают выполнение программы под управлением Windows 9x.

• Результаты, приведенные в приложении В, свидетельствуют о нелинейном поведении производительности при большом количестве потоков, состязающихся за доступ к очереди. Проекты для каждой из альтернативных стратегий содержатся на Web-сайте книги; эти проекты соответствуют различным вариантам конвейерной системы ThreeStage, описанной в следующих разделах.

•Резюмируя, следует подчеркнуть, что свойства очередей могут быть расширены таким образом, чтобы очередь могла совместно использоваться несколькими процессами и обеспечивать отправку или получение сразу нескольких сообщений за одну операцию. В то же время, некоторого выигрыша в производительности можно добиться за счет использования сигнальной модели, объектов CRITICAL_SECTIONS или функции SignalObjectAndWait. Соответствующие результаты представлены в приложении В.

Пример: использование очередей в многоступенчатом конвейере

Модель "хозяин/рабочий", во всех ее вариациях, является одной из наиболее популярных моделей многопоточного программирования, а программа 8.2 представляет простую модель "производитель/потребитель", являющуюся частным случаем более общей конвейерной модели (pipeline model).

В другом важном частном случае имеется один главный поток, который производит единичные рабочие задания (work units) для ограниченного количества рабочих потоков и помещает их в очередь. Такая методика может оказаться полезной при создании масштабируемого сервера с большим количеством клиентов (число которых может достигать тысячи и более), когда возможность выделения независимого рабочего потока для каждого клиента весьма сомнительна. В главе 14 задача создания масштабируемого сервера обсуждается в контексте портов завершения ввода/вывода.

В конвейерной модели каждый поток или группа потоков определенным образом обрабатывает единичные задания, например, сообщения, и передает их другим потокам для дальнейшей обработки. Аналогом многопоточного конвейера может служить производственная сборочная линия. Идеальным механизмом реализации конвейера являются очереди.

В программе 10.5 (ThreeStage.c) предусмотрено создание нескольких этапов производства и потребления, на каждой из которых поддерживается очередь рабочих заданий, подлежащих обработке. Каждая очередь имеет ограниченную, конечную длину. Всего существует три конвейерных ступени, соединяющих четыре этапа обработки. Программа имеет следующую структуру:

• Производители (producers) периодически создают единичные сообщения, дополненные контрольными суммами, используя для этого ту же функцию, что и в программе 8.2, если не считать того, что в каждом сообщении содержится дополнительное поле адресата, указывающее поток потребителя (consumer), для которой предназначено это сообщение, причем каждый производитель связывается только с одним потребителем. Количество пар "производитель/потребитель" задается в виде параметра командной строки. Далее производитель посылает одиночное сообщение передающему потоку (transmitter), помещая его в очередь передачи сообщений. Если очередь заполнена, производитель ждет, пока ее состояние не изменится.

• Передающий поток объединяет имеющиеся единичные сообщения (но не более пяти за один раз) и создает одно передаваемое сообщение, которое содержит заголовок и ряд единичных сообщений. Затем передающий поток помещает каждое передаваемое сообщение в очередь приема сообщений (receiver), блокируясь, если очередь заполнена. В общем случае передатчик и приемник могут связываться между собой через сетевое соединение. Произвольно выбранное здесь значение коэффициента блокирования (blocking factor), равное 5:1, легко поддается регулировке.

• Принимающий поток обрабатывает единичные сообщения, входящие в состав каждого передаваемого сообщения, и помещает каждое из них в соответствующую очередь потребителя, если она не заполнена.

• Каждый поток потребителя получает одиночные сообщения по мере их поступления и записывает сообщение в файл журнала регистрации.

Блок-схема системы представлена на рис. 10.1. Обратите внимание, что эта система моделирует сетевое соединение, в котором сообщения, относящиеся к различным парам "отправитель/получатель" объединяются и передаются по общему каналу связи.

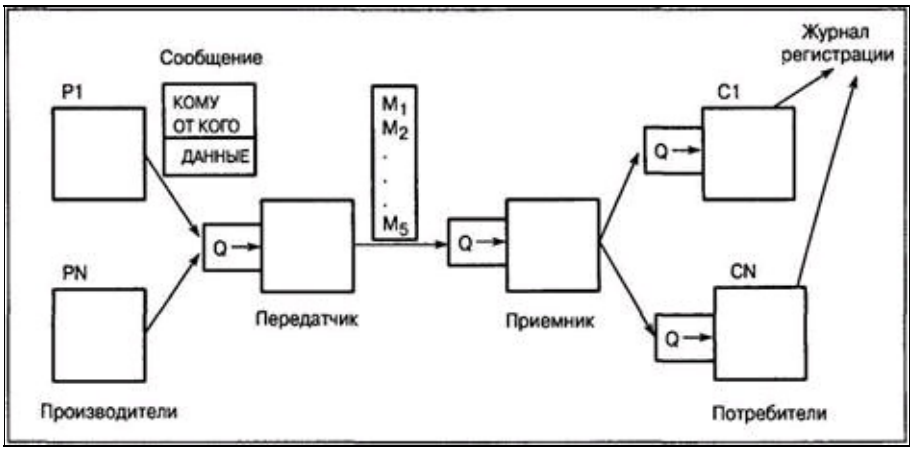


Рис. 10.1. Многоступенчатый конвейер

В программе 10.5 предложен вариант реализации, в котором используются функции очереди из программы 10.4. Функции генерации и отображения сообщений здесь не представлены, но они взяты из программы 8.1. При этом, наряду с контрольными суммами и данными, в блоки сообщений введены поля производителя и адресата.

Программа 10.5. ThreeStage.c: многоступенчатыйконвейер

```
    /* Глава 10. ThreeStage.c */
    /* Трехступенчатая система производитель/потребитель. */
    /* Использование: ThreeStage npc goal. */
    /* Запустить "npc" пар потоков производителя и потребителя. */
    /* Каждый производитель должен сгенерировать в общей сложности */
    /* "goal" сообщений, каждое из которых снабжается меткой, указывающей */
    /* потребителя, для которого оно предназначено. */
    /* Сообщения отправляются "передающему потоку", который, прежде чем */
    /* отправить группу сообщений "принимающему потоку", выполняет некоторую*/
    /*  дополнительную обработку. Наконец, принимающий поток отправляет сообщения
потокам потребителя. */

    #include "EvryThng.h"
    #include "SynchObj.h"
    #include "messages.h"
    #include <time.h>

    #define DELAY_COUNT 1000
    #define MAX_THREADS 1024

    /* Размеры и коэффициенты блокирования очередей. Эти величины являются */
    /* произвольными и могут регулироваться для обеспечения оптимальной */
    /* производительности. Текущие значения не являются сбалансированными. */
    #define TBLOCK_SIZE 5 /*Передающий поток формирует группы из 5 сообщений.*/
    #define TBLOCK_TIMEOUT 50 /*Интервал ожидания сообщений передающим потоком.*/
    #define P2T_QLEN 10 /* Размер очереди "производитель/передающий поток". */
    #define T2R_QLEN 4 /*Размер очереди "передающий поток/принимающий поток".*/
    #define R2C_QLEN 4 /* Размер очереди "принимающий поток/потребитель" -- */
    /* для каждого потребителя существует только одна очередь.*/

    DWORD WINAPI producer(PVOID);
    DWORD WINAPI consumer(PVOID);
    DWORD WINAPI transmitter(PVOID);
    DWORD WINAPI receiver(PVOID);
```

```
    typedef struct _THARG {
     volatile DWORD thread_number;
     volatile DWORD work_goal; /* Используется потоками производителей. */
      volatile DWORD work_done; /* Используется потоками производителей и
потребителей. */ '
     char future[8];
    } THARG;

    /* Сгруппированные сообщения, посылаемые передающим потоком потребителю.*/
    typedef struct t2r_msg_tag {
     volatile DWORD num_msgs; /* Количество содержащихся сообщений. */
     msg_block_t messages[TBLOCK_SIZE];
    } t2r_msg_t;

    queue_t p2tq, t2rq, *r2cq_array;

    static volatile DWORD ShutDown = 0;
    static DWORD EventTimeout = 50;

    DWORD _tmain(DWORD argc, LPTSTR * argv[]) {
     DWORD tstatus, nthread, ithread, goal, thid;
     HANDLE *producer_th, *consumer_th, transmitter_th, receiver_th;
     THARG *producer_arg, *consumer_arg;
     nthread = atoi(argv[1]);
     goal = atoi(argv[2]);
     producer_th = malloc(nthread * sizeof(HANDLE));
     producer_arg = calloc(nthread, sizeof(THARG));
     consumer_th = malloc(nthread * sizeof(HANDLE));
     consumer_arg = calloc(nthread, sizeof(THARG));
     q_initialize(&p2tq, sizeof(msg_block_t), P2T_QLEN);
     q_initialize(&t2rq, sizeof(t2r_msg_t), T2R_QLEN);
       /*   Распределить   ресурсы,   инициализировать   очереди   "принимающий
поток/потребитель" для каждого потребителя. */
     r2cq_array = calloc(nthread, sizeof(queue_t));
     for (ithread = 0; ithread < nthread; ithread++) {
      /* Инициализировать очередь r2c для потока данного потребителя. */
      q_initialize(&r2cq_array[ithread], sizeof(msg_block_t), R2C_QLEN);
      /* Заполнить аргументы потока. */
      consumer_arg[ithread].thread_number = ithread;
      consumer_arg[ithread].work_goal = goal;
      consumer_arg[ithread].work_done = 0;
          consumer_th[ithread]   =   (HANDLE)_beginthreadex(NULL,   0,   consumer,
(PVOID)&consumer_arg[ithread], 0, &thid);
      producer_arg[ithread].thread_number = ithread;
      producer_arg[ithread].work_goal = goal;
      producer_arg[ithread].work_done = 0;
          producer_th[ithread]   =   (HANDLE)_beginthreadex(NULL,   0,   producer,
(PVOID)&producer_arg[ithread], 0, &thid);
     }
      transraitter_th  =  (HANDLE)_beginthreadex(NULL,  0,  transmitter,  NULL,  0,
&thid);
     receiver_th = (HANDLE)_beginthreadex (NULL, 0, receiver, NULL, 0, &thid);
     _tprintf(_T("ХОЗЯИН: Выполняются все потоки\n"));
     /* Ждать завершения потоков производителя. */
     for (ithread = 0; ithread < nthread; ithread++) {
      WaitForSingleObject(producer_th[ithread], INFINITE);
       _tprintf(_T("ХОЗЯИН: производитель %d выработал %d единичных сообщений\n"),
```

```
ithread, producer_arg[ithread].work_done);
     }
     /* Производители завершили работу. */
     _tprintf(_T("ХОЗЯИН: Все потоки производителя выполнили свою работу.\n"));
     /* Ждать завершения потоков потребителя. */
     for (ithread = 0; ithread < nthread; ithread++) {
      WaitForSingleObject(consumer_th[ithread], INFINITE);
        _tprintf(_T("ХОЗЯИН: потребитель %d принял %d одиночных сообщений\n"),
ithread, consumer_arg[ithread].work_done);
     }
     _tprintf(_T("ХОЗЯИН: Все потоки потребителя выполнили свою работу.\n"));
     ShutDown = 1; /* Установить флаг завершения работы. */
      /* Завершить выполнение и перейти в состояние ожидания передающих и
принимающих потоков. */
     /* Эта процедура завершения работает нормально, поскольку и передающий,*/
     /* и принимающий потоки не владеют иными ресурсами, кроме мьютекса, */
      /* которые они могли бы покинуть по завершении выполнения, не уступив прав
владения ими. Можете ли вы улучшить эту процедуру? */
     TerminateThread(transmitter_th, 0);
     TerminateThread(receiver_th, 0);
     WaitForSingleObject(transmitter_th, INFINITE);
     WaitForSingleObject(receiver_th, INFINITE);
     q_destroy(&p2tq);
     q_destroy(&t2rq);
      for (ithread = 0; ithread < nthread; ithread++) q_destroy(&r2cq_array
[ithread]);
     free(r2cq_array);
     free(producer_th);
     free(consumer_th);
     free(producer_arg);
     free(consumer_arg);
     _tprintf(_T("Система завершила работу. Останов системы\n"));
     return 0;
    }

    DWORD WINAPI producer(PVOID arg) {
     THARG * parg;
     DWORD ithread, tstatus;
     msg_block_t msg;
     parg = (THARG *)arg;
     ithread = parg->thread_number;
     while (parg->work_done < parg->work_goal) {
      /* Вырабатывать единичные сообщения, пока их общее количество */
      /* не станет равным "goal". */
      /* Сообщения снабжаются адресами отправителя и адресата, которые в */
      /* нашем примере одинаковы для всех сообщений, но в общем случае */
      /* могут быть различными. */
      delay_cpu(DELAY_COUNT * rand() / RAND_MAX);
      message_fill(&msg, ithread, ithread, parg->work_done);
      /* Поместить сообщение в очередь. */
      tstatus = q_put(&p2tq, &msg, sizeof(msg), INFINITE);
      parg->work_done++;
     }
     return 0;
    }

    DWORD WINAPI transmitter(PVOID arg) {
     /* Получись несколько сообщений от производителя, объединяя их в одно*/
     /* составное сообщение, предназначенное для принимающего потока. */
```

```
 DWORD tstatus, im;
 t2r_msg_t t2r_msg = {0};
 msg_block_t p2t_msg;
 while (!ShutDown) {
  t2r_msg.num_msgs = 0;
  /* Упаковать сообщения для передачи принимающему потоку. */
  for (im = 0; im < TBLOCK_SIZE; im++) {
   tstatus = q_get(&p2tq, &p2t_msg, sizeof(p2t_msg), INFINITE);
   if (tstatus != 0) break;
   memcpy(&t2r_msg.messages[im], &p2t_msg, sizeof(p2t_msg));
   t2r_rasg.num_msgs++;
  }
  tstatus = q_put(&t2rq, &t2r_msg, sizeof(t2r_msg), INFINITE);
  if (tstatus != 0) return tstatus;
 }
 return 0;
}

DWORD WINAPI receiver(PVOID arg) {
 /* Получить составные сообщения от передающего потока; распаковать */
 /* их и передать соответствующему потребителю. */
 DWORD tstatus, im, ic;
 t2r_msg_t t2r_msg;
 msg_block_t r2c_msg;
 while (!ShutDown) {
  tstatus = q_get(&t2rq, &t2r_msg, sizeof(t2r_msg), INFINITE);
  if (tstatus != 0) return tstatus;
  /* Распределить сообщения между соответствующими потребителями. */
  for (im = 0; im < t2r_msg.num_msgs; im++) {
   memcpy(&r2c_msg, &t2r_msg.messages[im], sizeof(r2c_msg));
   ic = r2c_msg.destination; /* Конечный потребитель. */
   tstatus = q_put(&r2cq_array[ic], &r2c_msg, sizeof(r2c_msg), INFINITE);
   if (tstatus != 0) return tstatus;
  }
 }
 return 0;
}

DWORD WINAPI consumer(PVOID arg) {
 THARG * carg;
 DWORD tstatus, ithread;
 msg_block_t msg;
 queue_t *pr2cq;
 carg = (THARG *)arg;
 ithread = carg->thread_number;
 carg = (THARG *)arg;
 pr2cq = &r2cq_array[ithread];
 while (carg->work_done < carg->work_goal) {
  /* Получить и отобразить (необязательно — не показано) сообщения. */
  tstatus = q_get(pr2cq, &msg, sizeof(msg), INFINITE);
  if (tstatus != 0) return tstatus;
  carg->work_done++;
 }
 return 0;
}
```

Комментарии по поводу многоступенчатого конвейера

Данная реализация характеризуется некоторыми особенностями, суть которых частично отражена в комментариях, включенных в листинг программы. На эти же особенности обращают ваше внимание и упражнения 10.6, 10.7 и 10.10.

• Значительные возражения вызывает способ, используемый основным потоком для завершения выполнения передающего и принимающего потоков. Лучшим решением было бы использование конечных интервалов ожидания во внутренних циклах передатчика и приемника и прекращение работы после того, как будет установлен соответствующий глобальный флаг. Другой возможный подход заключается в отмене выполнения потоков, как описано далее в этой главе.

• Обратите внимание на существование симметрии между передающим и принимающим потоками. Как и при реализации очереди, это обстоятельство упрощает проектирование, отладку и сопровождение программы.

• Реализация не сбалансирована в смысле согласования скорости генерации сообщений, емкости конвейера и коэффициента блокирования "передатчик/приемник".

• В данной реализации (программа 10.4) для защиты очередей используются мьютексы. Результаты экспериментов с объектами CRITICAL_SECTION не позволили обнаружить сколько-нибудь заметного ускорения работы программы на однопроцессорной системе (см. приложение В). CS-версия программы, ThreeStageCS, находится на Web-сайте. Аналогичным образом вела себя программа и после того, как в ней была использована функции SignalObjectAndWait.

Асинхронные вызовы процедур

Основное возражение, которое можно предъявить к программе ThreeSage.c (программа 10.5) в ее нынешнем виде, касается прекращения выполнения передающего и принимающего потоков с помощью функции TerminateThread. В комментариях, включенных в код, вам предлагается подумать над более элегантным способом завершения выполнения потоков, который обеспечивал бы корректное прекращение работы программы и освобождение ресурсов.

Другой нерешенной проблемой является отсутствие общего метода (не считая использования функции TerminateThread), который обеспечивал бы отправку сигнала определенному потоку или инициировал его выполнение. События могут посылать сигналы одному потоку, ожидающему наступления автоматически сбрасываемого события, или всем потокам, ожидающим наступления вручную сбрасываемого события, но невозможно добиться того, чтобы сигнал был получен определенным потоком. Используемое до сих пор решение сводилось к тому, что пробуждались все ожидающие потоки, которые самостоятельно определяли, могут ли они теперь продолжить свое выполнение. Иногда привлекается альтернативное решение, суть которого состоит в назначении событий определенным потокам, так что сигнализирующий поток может определять, объект какого события следует перевести в сигнальное состояние одной из функций SetEvent или PulseEvent.

Обе эти проблемы решаются путем использования объектов асинхронного вызова процедур (Asynchronous Procedure Call, APC). События развиваются в следующей последовательности, причем рабочий или целевой потоки должны управляться главным потоком.

• Главный поток указывает APC-функцию данной целевого потока путем помещения объекта APC в очередь APC данного потока. В очередь могут быть помещены несколько APC.

• Целевой поток переходит в состояние дежурного ожидания (alertable wait state), обеспечивающее возможность безопасного выполнения потоком APC. Порядок первых двух шагов безразличен, поэтому о возникновении условий состязательности можно не беспокоиться.

• Указанный поток, находящийся в состоянии ожидания, выполняет все APC, находящиеся в очереди.

• APC могут выполнять любые нужные действия, например, освобождать ресурсы или генерировать исключения. Благодаря этому главный поток может инициировать возбуждение исключения в целевом потоке, хотя само исключение не произойдет до тех пор, пока целевой поток не перейдет в состояние дежурного ожидания.

Выполнение APC является асинхронным в том смысле, что APC может быть помещен в очередь целевого потока в любой момент, но само выполнение является синхронизированным в том смысле, что это может произойти лишь тогда, когда целевой поток входит в состояние дежурного ожидания.

Состояния дежурного ожидания будут вновь обсуждаться в главе 14, посвященной асинхронному вводу/выводу.

Описания необходимых функций и примеры их использования в другом варианте программы ThreeStage приводятся в следующих разделах. Проект для построения новой версии программы (ThreeStageCancel) и соответствующий исходный код (ThreeStageCancel. c) находятся на Web-сайте книги.

Очередизация асинхронных вызовов процедур

Один поток (главный) помещает APC в очередь целевого потока с помощью функции QueueUserAPC:

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

hThread — дескриптор целевого потока. dwData — аргумент, который будет передан функции APC при ее выполнении и может являться кодом завершения или сообщать функции иную информацию.

В основной функции программы ThreeStageCancel.c (сравните с программой 10.5) вызовы TerminateThread заменяются вызовами QueueUserAPC следующим образом:

```
// TerminateThread(transmitter_th, 0) ; заменить на APC
// TerminateThread(receiver_th, 0); заменить на APC
tstatus = QueueUserAPC(ShutDownTransmitter, transmitter_th, 1);
if (tstatus == 0) ReportError(…);
tstatus = QueueUserAPC(ShutDownReceiver, receiver_th, 2);
if (tstatus == 0) ReportError (…);
```

Функция QueueUserAPC в случае успешного ее завершения возвращает ненулевое значение, иначе — нуль. В то же время, функция GetLastError () не возвращает никакого полезного значения, и поэтому при вызове функции ReportError не требуется задавать текст сообщения об ошибке (значением последнего аргумента является FALSE).

pfnAPC — указатель на фактическую функцию, вызываемую целевым потоком, как показывает следующий фрагмент, взятый из программы ThreeStageCancel. c:

```
/* APC для завершения выполнения потребителя. */
void WINAPI ShutDownReceiver(DWORD n) {
 printf("Внутри ShutDownReceiver. %d\n", n);
 /* Освободить все ресурсы (в данном примере отсутствуют). */
 return;
}
```

Функция ShutDownTransmitter аналогична вышеприведенной, отличаясь от нее только текстом сообщения. Сразу трудно понять, каким образом эта функция, которая, казалось бы, не выполняет никаких существенных операций, может инициировать прекращение выполнения целевого принимающего потока. Соответствующие пояснения приводятся далее в этой главе.

APC и упущенные сигналы

APC, выполняемые в режиме ядра (используются в операциях асинхронного ввода/вывода), могут немедленно выводить ожидающий поток из состояния ожидания, что может стать причиной потери сигналов PulseEvent. В связи с этим в документации можно встретить советы, в которых функции PulseEvent рекомендуется не использовать, хотя, как было продемонстрировано в данной главе, они могут и приносить пользу. Применение функции PulseEvent в наших примерах было вполне безопасным, поскольку APC, выполняемые в режиме ядра, в них не используются. Кроме того, применение функции SignalObjectAndWait и тестирование возвращаемого ею значения обеспечивает достаточно надежную защиту от подобных потерь сигналов. Наконец, если вы опасаетесь, что это все-таки может случиться, просто включайте указание конечного интервала ожидания в соответствующие вызовы функций ожидания.

Состояния дежурного ожидания

Во всех предыдущих примерах значение параметра bAlertable, являющегося последним параметром функции SignalObjectAndWait, полагалось равным FALSE. Используя вместо него значение TRUE, мы указываем, что ожидание должно быть, как говорят, *дежурным* (alertable), и тогда после выполнения функции поток перейдет в состояние дежурного ожидания. В этом состоянии поток ведет себя следующим образом:

• Если один или более APC помещаются в очередь потока (указанного в качестве целевого при вызове функции QueueUserAPC) еще до того, как либо объект, указываемый дескриптором hObjectToWaitOn (обычно таким объектом является событие), перейдет в сигнальное состояние, либо истечет интервал ожидания, то все эти потоки выполнятся (при этом не гарантируется какой-то определенный порядок их выполнения), а функция SignalObjectAndWait завершит выполнение, возвращая значение WAIT_IO_COMPLETED.

• Если APC в очередь не помещались, то функция SignalObjectAndWait ведет себя обычным образом, то есть ожидает перехода объекта в сигнальное состояние или истечения интервала ожидания.

Состояния дежурного ожидания будут вновь использоваться нами при выполнении операций асинхронного ввода/вывода (глава 14); именно в связи с этими операциями и получило свое название значение WAIT_IO_COMPLETED. В состояние дежурного ожидания потока можно переводить также с помощью функций WaitForSingleObjectEx, WaitForMultipleObjectsEx и SleepEx, которые оказываются полезными и при выполнении операций асинхронного ввода/вывода.

Теперь можно изменить функции q_get и q_put (см. программу 10.4) таким образом, чтобы завершение работы программы после выполнения APC было корректным, хотя APC-функция и не выполняет никаких иных действий, кроме вывода сообщения и возврата из функции. Все, что в данном случае требуется — это организовать вход в состояние дежурного ожидания и проверить значение, возвращаемое функцией SignalObjectAndWait, как показано в приведенной ниже видоизмененной версии функции q_get (см. файл QueueObjCancel.c, находящийся на Web-сайте).

Программа 10.6. Модифицированная функция q_get, обеспечивающая корректное завершение выполнения потоков

```
    DWORD q_put(queue_t *q, PVOID msg, DWORD msize, DWORD MaxWait) {
     BOOL Cancelled = FALSE;
     if (q_destroyed(q)) return 1;
     WaitForSingleObject(q->q_guard, INFINITE);
     while (q_full(q) && !Cancelled) {
          if   (SignalObjectAndWait(q->q_guard,   q->q_nf,   INFINITE,   TRUE)   ==
WAIT_IO_COMPLETION) {
       Cancelled = TRUE;
       continue;
      }
      WaitForSingleObject(q->q_guard, INFINITE);
     }
     /* Поместить сообщение в очередь. */
     if (!Cancelled) {
      q_remove(q, msg, msize);
       /* Сигнализировать о том, что очередь не заполнена, поскольку мы удалили
```

```
сообщение. */
      PulseEvent(q->q_nf);
      ReleaseMutex(q->q_guard);
     }
     return Cancelled ? WAIT_TIMEOUT : 0;
    }
```

В качестве функции APC могут выступать и функция ShutDownReceiver, и функция ShutDownTransmitter, поскольку приемник и передатчик используют как функцию q_get, так и функцию q_put. Если требуется, чтобы функциям завершения было известно, из какого потока они выполняются, применяйте различные значения для аргументов функций APC, которые передаются третьим аргументом функции QueueUserAPC во фрагменте кода, предшествующем программе 10.6.

Чтобы обеспечить согласованность с предыдущими версиями программы, в качестве кода завершения следует использовать значение WAIT_TIMEOUT.

В качестве альтернативного варианта вместо проверки совпадения возвращаемого значения со значением WAIT_IO_COMPLETION можно предусмотреть генерацию исключения функциями завершения и поместить тело функции q_put в try-блок, дополнив программу обработчиком исключений.

Безопасная отмена выполнения потоков

Обсуждение предыдущего примера продемонстрировало, как безопасно отменить выполнение целевого потока, который использует состояния дежурного ожидания. Несмотря на использование APC, такую отмену выполнения иногда называют *синхронной отменой* (synchronous cancellation), поскольку отмена выполнения, которую инициировал вызов функции QueueUserAPC главным потоком, сможет осуществиться лишь тогда, когда целевой поток достигнет безопасного состояния дежурного ожидания.

Синхронная отмена выполнения требует участия в этом целевого потока, которая время от времени должна предоставлять возможность прекратить ее выполнение. Естественный способ вхождения в состояние дежурного ожидания предоставляют функции ожидания событий, поскольку в процессе прекращения работы системы объект события может не перейти вновь в сигнальное состояние. Ожидание мьютексов также можно выполнять в дежурном режиме, чтобы учесть те случаи, когда поток ожидает ресурса, который, возможно, не будет вновь доступным. Этот метод, например, может использоваться главным потоком для разрушения взаимной блокировки потоков.

Асинхронная отмена выполнения потоков (asynchronous thread cancellation) может применяться в тех случаях, когда сигнал должен посылаться потоку, который выполняет интенсивные вычисления и находится в состоянии ожидания ввода/вывода или событий исключительно редко, если это вообще происходит. Возможность асинхронной отмены выполнения потоков в Windows отсутствует, хотя и существуют методики, использующие зависящий от типа процессора код, которые позволяют прерывать выполнение определенного потока.

Создание переносимых приложений с использованием потоков Pthreads

Потоки Pthreads уже неоднократно упоминались нами в качестве альтернативной модели многопоточного программирования и синхронизации, доступной в UNIX, Linux и других системах, не принадлежащих семейству Windows. Существует библиотека Windows Pthreads с открытым исходным кодом, используя которую можно создавать переносимые многопоточные приложения, способные выполняться на самых различных системах. Более подробное обсуждение этого вопроса вы найдете на Web-сайте книги. Указанная библиотека с открытым исходным кодом применяется в проекте ThreeStagePthreads, в котором также предоставляется соответствующая ссылка на сайт загрузки.

Стеки потоков и допустимые количества потоков

Следует сделать еще два предостережения. Во-первых, подумайте о размере стека, который по умолчанию составляет 1 Мбайт. В большинстве случаев этого будет вполне достаточно, но если существуют какие-либо сомнения на сей счет, оцените максимальный объем стекового пространства, которое требуется для каждого потока с учетом всех библиотечных и рекурсивных функций, которые вызываются потоком. Переполнение стека может привести к порче памяти или вызвать исключение.

Во-вторых, использование большого количества потоков с большими размерами стеков потребует больших объемов виртуальной памяти для их обработки и может оказать отрицательное влияние на процессы страничного обмена и состояние файла подкачки. Так, использовать свыше 1000 потоков в некоторых из примеров, приведенных в этой и последующей главах, было бы неразумно. При размере стека 1 Мбайт на один поток для этого потребовалось бы виртуальное адресное пространство объемом 1 Гбайт. Соответствующие меры предосторожности включают тщательное планирование размеров стеков, использование портов завершения ввода/вывода и мультиплексирование операций в пределах одного потока.

Рекомендации по проектированию, отладке и тестированию программ

Рискуя дать совет, противоречащий высказываниям во многих других книгах и технических статьях, в которых основной упор делается на тестировании и уже затем рассматривается все остальное, лично я порекомендовал бы вам распределить свои усилия таким образом, чтобы нашлось время для ознакомления с особенностями проектирования, реализации и использования известных моделей программирования. Лучший метод отладки заключается, прежде всего, в недопущении ошибок; разумеется, такие советы легче давать, чем им следовать. Тем не менее, когда начинают проявляться программные дефекты — а это происходит всегда – наиболее эффективным методом обнаружения и исправления основных причин дефектов является тщательное исследование кода в сочетании с отладкой.

Не возлагайте слишком большие надежды на тестирование программ, ибо каким бы тщательным и обширным оно ни было, многие серьезные дефекты от вас все равно ускользнут. Тестирование способно лишь выявлять дефекты; оно не может служить доказательством отсутствия дефектов и указывает лишь на их симптомы, а не на причины, которые их породили. Для наглядности могу сослаться на личный опыт, приведя в качестве примера версию программы, включающую функцию ожидания сложного семафора, построенную на основе модели CV, в которой конечный интервал ожидания не использовался. Дефект, который мог приводить к блокированию потока на неопределенное время, никак не проявлял себя на протяжении года, пока, в конечном счете, что-то пошло не так, как надо. Мне удалось обнаружить ошибку путем простого просмотра кода, благодаря пониманию принципов, лежащих в основе работы модели переменных условий.

Не следует переоценивать и помощь, которую вам может оказать отладка, поскольку отладчики изменяют временное поведение программы, маскируя возникновение условий состязательности, то есть именно то, что вы хотели бы исследовать. Так, вряд ли можно ожидать, что с помощью отладчика вам удастся выявить проблемы, обусловленные неправильным выбором типа события (автоматически сбрасываемым или сбрасываемым вручную) или функции (SetEvent или PulseEvent). Всегда тщательно продумывайте, чего именно вы хотите добиться, применяя те или иные средства.

В заключение следует подчеркнуть, что тестирование программ с использованием как можно более широкого круга платформ, включая SMP, составляет важнейшую часть любого проекта, целью которого является разработка многопоточного программного обеспечения.

Как избежать создания некорректного программного кода

Каждая ошибка, не допущенная вами в исходном коде, — это, прежде всего, еще одна сэкономленная ошибка, которую вам не придется отыскивать на стадии отладки программы или в процессе проверки работоспособности ее завершенной версии. Ниже приведены некоторые рекомендации, большинство из которых взяты, хотя и в перефразированном виде, из [6].

• **Не полагайтесь на инерционность потоков.** Потоки асинхронны, но мы, например, часто предполагаем, что после создания родительским потоком одного или нескольких дочерних потоков он продолжает свое выполнение. В основе таких предположений лежит допущение об "инерционных" свойствах родительского потока, благодаря которым он, якобы, будет выполняться вплоть до начала выполнения дочерних потоков. Предположения подобного рода особенно опасны в случае SMP-систем, тем не менее, они могут привести к возникновению проблем и в случае однопроцессорных систем.

• **Никогда не делайте ставок на состязании потоков.** Об очередности выполнения потоков практически никогда нельзя сказать ничего определенного. В программе должно предполагаться, что любой готовый к выполнению поток может быть запущен в любой момент и что любой выполняющийся поток в любой момент может быть вытеснен. Никакого упорядочения выполнения потоков не существует, если только вы специально об этом не позаботились.

• **Не следует путать планирование выполнения с синхронизацией.** Стратегии планирования задач и назначения приоритетов не в состоянии обеспечить нужную синхронизацию. Для этого должны использоваться объекты синхронизации.

• **Состязательность за очередность выполнения будет существовать даже при использовании мьютексов для защиты разделяемых данных.** Наличие защиты данных само по себе не может служить гарантией определенной очередности получения доступа к разделяемым данным различными потоками. Например, если один поток добавляет средства на банковский счет, а другой снимает их со счета, то одна только защита счета с помощью мьютекса не сможет гарантировать, что внесение денег на счет всегда будет осуществляться раньше, чем их снятие. В упражнении 10.14 показано, как организовать управление очередностью выполнения потоков.

• **Необходимость кооперирования потоков во избежание взаимной блокировки.** Чтобы гарантировать невозможность взаимоблокировки потоков, вы должны располагать хорошо продуманной иерархией блокировок, которая использовалась бы всеми потоками.

• **Не допускайте разделения событий предикатами.** В реализациях, использующих переменные условий, каждое событие должно связываться с отдельным предикатом. Кроме того, событие всегда должно использоваться с одним и тем же мьютексом.

• **Остерегайтесь совместного использования стеков и связанного с этим риска порчи памяти.** Никогда не забывайте о том, что при возврате из функции или завершении выполнения потока их локальные области памяти становятся недействительными. Память в области стека потока может использоваться другими потоками, но вы должны быть уверены в том, что первый поток все еще существует.

• **Следите за использованием модификатором класса памяти volatile в необходимых случаях.** Всякий раз, когда возможно изменение разделяемой переменной в одном потоке и обращение к нему в другом, класс памяти, используемой этой переменной, должен быть определен как volatile, что будет гарантировать использование ячеек памяти при сохранении и извлечении этой переменной потоками, а не регистров, специфичных для каждого потока.

Ниже приводятся некоторые дополнительные рекомендации и мнемонические правила, которые вам могут пригодиться.

• **Правильно пользуйтесь моделью переменных условий,** проверяя, чтобы два разных мьютекса не использовались с одним и тем же событием. Хорошо изучите модель переменных условий, на которой основана ваша программа. Прежде чем вызывать функцию ожидания перехода переменной условия в сигнальное состояние, убедитесь в выполнении условия, выраженного предикатом.

• **Старайтесь вникнуть в смысл используемых инвариантов и предикатов переменных условий,** даже если они имеют неформальное выражение. Убедитесь в том, что условие, выраженное предикатом, всегда выполняется за пределами критического участка кода.

• **Стремитесь к упрощению.** Многопоточное программирование уже само по себе является достаточно сложным, чтобы еще дополнительно усложнять его введением трудно воспринимаемых моделей и логики. Если ваша программа становится чрезмерно сложной, постарайтесь оценить, продиктована ли эта сложность действительной необходимостью или же является следствием того, что программа была неудачно спроектирована.

• **Тестируйте программы как на однопроцессорных, так и на многопроцессорных системах, а также на системах с различными тактовыми частотами и другими характеристиками.** Природа некоторых дефектов такова, что на однопроцессорных системах они проявляются лишь в редких случаях или вообще никогда, в то время как на SMP-системах вы их сразу определите, и наоборот. Аналогичным образом, чем шире круг характеристик систем, на которых будет выполняться программа, содержащая дефекты, тем выше вероятность сбоя.

• **Тестирование является необходимой, но не достаточной мерой, которая могла бы гарантировать корректную работу программы.** Известны многочисленные примеры программ, заведомо содержащих дефекты, которые лишь в редких случаях удавалось обнаруживать средствами обычного и даже расширенного тестирования.

• **Смиритесь!** Как бы вы ни старались следовать этим советам, ошибок в своих программах вам не избежать. Это утверждение справедливо даже в случае однопоточных программ, не говоря уже о многопоточных, которые предоставляют нам гораздо больше разнообразных и интереснейших возможностей создавать себе проблемы.

За рамками Windows API

В своем рассмотрении мы намеренно ограничились случаем Windows API. Вместе с тем, Microsoft предоставляет дополнительные средства доступа к таким объектам ядра, как потоки. Так, класс ThreadPool, доступный в C++, C# и других языках программирования, позволяет создавать пулы потоков и очереди задач потоков (для этого служит метод QueueUserWorkItem класса ThreadPool).

Кроме того, Microsoft реализует службу Microsoft Message Queuing (MSMQ), которая предоставляет услуги по передаче сообщений между сетевыми системами. Приведенный в данной главе пример должен был продемонстрировать вам, насколько полезными могут быть универсальные системы очередизации сообщений. MSMQ документирована на Web-сайте компании Microsoft.

Резюме

Разработка многопоточных программ значительно упрощается, если используются хорошо себя зарекомендовавшие, известные модели и методы программирования. В этой главе была продемонстрирована полезность модели переменных условий и решен ряд сравнительно сложных, но важных проблем программирования. APC позволяют одному потоку посылать сигналы другому и вызывать в нем выполнение определенных действий, что позволяет отменять выполнение потоков таким образом, чтобы все потоки в системе имели возможность корректно завершиться.

Сложность синхронизации и управления потоками объясняется тем, что существует множество путей решения одной и той же задачи, и при выборе метода следует учитывать его сложность и характер влияния, которое он может оказать на производительность. Чтобы проиллюстрировать все многообразие доступных возможностей, пример трехступенчатого конвейера был реализован несколькими отличными друг от друга способами.

Тщательное продумывание проекта и путей его реализации является предпочтительным способом улучшения качества программы. Возложение чрезмерных надежд на результаты тестирования и отладку без того, чтобы уделить должное внимания деталям, может привести к возникновению серьезных проблем, обнаружить и устранить которые будет очень трудно.

В следующих главах

В главе 11 показано, как организовать взаимодействие между процессами и потоками, выполняющимися внутри этих процессов, используя именованные каналы (named pipes) и почтовые ящики (mailslots) Windows. В качестве основного примера выбрана клиент-серверная система, в которой для обслуживания запросов клиентов используется пул рабочих потоков. В главе 12 эта же система реализуется с привлечением гнезд Windows Sockets, что делает ее пригодной для использования стандартных протоколов. В обновленной клиент-серверной системе применяются безопасные библиотеки DLL с многопоточной поддержкой, а сервер использует внутрипроцессный сервер (in-process server) на основе DLL.

Дополнительная литература

Источником большей части информации и советов по программированию, приведенных в конце настоящей главы, послужила книга [6]. Из нее же было взято в переработанном виде и решение на базе объекта барьера, использованное в программах 10.1 и 10.2.

В статье Дугласа Шмидта (Douglas Schmidt) и Ирфана Пьярали (Irfan Pyarali) "Strategies for Implementing POSIX Condition Variables in Win32" ("Стратегии реализации переменных условий POSIX в Win32") (доступна по адресу http:// www.es.wustl.edu/~schmidt/win32-cv-1.html), обсуждается ограниченность событий Win32 (Windows) и эмуляция переменных условий, а также дан глубокий анализ и оценка нескольких подходов. Вместе с тем, этот материал был написан еще до появления функции SignalObjectAndWait, и поэтому большое внимание в статье уделяется тому, как избежать потери сигналов. Чтение этой статьи позволит вам по достоинству оценить возможности новых функций. В другой статье тех же авторов (http://www.cs.wustl.edu/~schmidt/win32-cv-2.html) рассматривается создание объектно-ориентированных оболочек вокруг объектов синхронизации Windows, позволяющих добиться независимости интерфейса синхронизации от платформы. Основанная на работе Шмидта и

Пьярали реализация, в которой используются потоки Pthreads, доступна по адресу http://sources.redhat.com/pthreads-win32/.

Упражнения

10.1. Переработайте программу 10.1, исключив из нее функцию SignalObjectAndWait; для тестирования полученного результата воспользуйтесь Windows 9x.

10.2. Модифицируйте программу evenpc (программа 8.2), использовав модель переменных условий и обеспечив возможность существования нескольких потребителей. События какого типа потребуются в данном случае?

10.3. Измените логику работы программы 10.2 таким образом, чтобы объект события переходил в сигнальное состояние только один раз.

10.4. Замените мьютекс в объекте очереди, который используется в программе 10.2, объектом CS. Какое влияние это изменение оказывает на производительность и пропускную способность программы? Решение находится на Web-сайте книги, а соответствующие экспериментальные данные приведены в приложении B.

10.5. Для индикации состояний очереди, в которых она не пуста или не заполнена, в программе 10.4 применяется широковещательная модель CV. Будет ли в данном случае работать сигнальная модель CV? Не является ли она даже более предпочтительной в некоторых отношениях? Соответствующие экспериментальные данные приведены в приложении B.

10.6. Поэкспериментируйте с размерами очереди и величиной коэффициента блокирования "передатчик/приемник" в программе 10.5 для выяснения того, какое влияние оказывают эти факторы на загрузку ЦП, а также производительность и пропускную способность программы.

10.7. Видоизмените программы 10.3–10.5, обеспечив их соответствие принятым в Windows соглашениям о правилах образования имен, которых мы придерживаемся на протяжении всей книги.

10.8. *Для программистов на* C++. Приведенный в программах 10.3 и 10.4 код можно использовать для создания в C++ класса синхронизированной очереди; создайте этот класс и протестируйте его, модифицировав соответствующим образом программу 10.5. Какие из функций должны быть общедоступными, а какие — закрытыми?

10.9. Исследуйте, как изменятся показатели производительности программы 10.5 после замены мьютексов объектами CRITICAL_SECTIONS.

10.10. Улучшите программу 10.5, исключив необходимость прекращения выполнения потоков передатчика и приемника. Потоки должны самостоятельно завершать свое выполнение.

10.11. На web-сайте находится файл multisem.c, который реализует сложный семафор, имитирующий объекты Windows (они имеют имена и атрибуты безопасности, могут разделяться процессами, и для них предусмотрены две модели ожидания), а также файл тестовой программы TestMultiSem.c. Выполните сборку и тестирование этой программы. Как в ней используется модель переменных условий? Повышается ли производительность в результате использования объекта CRITICAL_SECTION? Что здесь выступает в роли инвариантов и предикатов переменных условий?

10.12. Проиллюстрируйте целесообразность рекомендаций, приведенных в конце настоящей главы, ссылаясь на ошибки, с которыми вам пришлось столкнуться, или ошибки, содержащиеся в версии программы с дефектами, представленной на Web-сайте.

10.13. Ознакомьтесь со статьей Шмидта и Пьярали "Strategies for Implementing POSIX Condition Variables in Win32" ("Стратегии реализации переменных условий POSIX в Win32") (см. раздел "Дополнительная литература"). Примените их методы анализа равноправия, корректности, сериализации и других программных факторов к моделям переменных условий (которые в указанной статье называются "идиомами" ("idioms")), фигурирующим в настоящей

главе. Заметьте, что сами переменные условия в настоящей главе не эмулируются; вместо этого эмулируется их использование, тогда как Шмидт и Пьярали эмулируют переменные условий, используемые в произвольном контексте.

10.14. Находящиеся на web-сайте проекты batons и batonsmultipleevents демонстрируют альтернативные варианты решения задачи сериализации выполнения потоков. О предпосылках и предшествующих работах других авторов говорится в комментариях, включенных в код. Во втором решении с каждым потоком связывается уникальное событие, что позволяет отслеживать сигнальные состояния отдельных потоков. Для реализации выбран язык C++, что дало возможность воспользоваться средствами стандартной библиотеки шаблонов C++ (Standard Template Library, STL). Проанализируйте, что имеют общего и чем различаются между собой эти два решения и используйте второе из них в качестве средства ознакомления с библиотекой STL.

ГЛАВА 11
Взаимодействие между процессами

В главе 6 было показано, как создавать процессы и управлять ими, тогда как главы 7—10 были посвящены описанию методов управления потоками, которые выполняются внутри процессов, и объектов, обеспечивающих их синхронизацию. Вместе с тем, если не считать использования разделяемой памяти, мы до сих пор не рассмотрели ни одного из методов взаимодействия между процессами.

Ниже вы ознакомитесь с последовательным межпроцессным взаимодействием (Interprocess Communication, IPC)[30], в котором используются объекты, подобные файлам. Двумя основными механизмами Windows, реализующими IPC, являются анонимные и именованные каналы, доступ к которым осуществляется с помощью уже известных вам функций ReadFile и WriteFile. Простые анонимные каналы являются символьными и работают в полудуплексном режиме. Эти свойства делают их удобными для перенаправления выходных данных одной программы на вход другой, как это обычно делается в UNIX. В первом примере демонстрируется, как реализовать эту возможность.

По сравнению с анонимными каналами возможности именованных каналов гораздо богаче. Они являются дуплексными, ориентированы на обмен сообщениями и обеспечивают взаимодействие через сеть. Кроме того, один именованный канал может иметь несколько открытых дескрипторов. В сочетании с удобными, ориентированными на выполнение транзакций функциями эти возможности делают именованные каналы пригодными для создания клиент-серверных систем. Это демонстрируется во втором из приведенных в настоящей главе примере, представляющем многопоточный клиент-серверный командный процессор, моделируемый в соответствии с рис. 7.1, который привлекался для обсуждения потоков. Каждый из потоков сервера управляет взаимодействием с отдельным клиентом, и для каждой пары "поток/клиент" используется отдельный дескриптор, то есть отдельный экземпляр именованного канала.

Наконец, почтовые ящики обеспечивают широковещательную рассылку сообщений по схеме "один многим", а их использование для расширения возможностей командного процессора демонстрируется в последнем примере.

Анонимные каналы

Анонимные каналы (anonymous channels) Windows обеспечивают однонаправленное (полудуплексное) посимвольное межпроцессное взаимодействие. Каждый канал имеет два дескриптора: дескриптор чтения (read handle) и дескриптор записи (write handle). Функция, с помощью которой создаются анонимные каналы, имеет следующий прототип:

```
    BOOL CreatePipe(PHANDLE phRead, PHANDLE phWrite, LPSECURITY_ATTRIBUTES
lpsa, DWORD cbPipe)
```

Дескрипторы каналов часто бывают наследуемыми; причины этого станут понятными из приведенного ниже примера. Значение параметра cbPipe, указывающее размер канала в байтах, носит рекомендательный характер, причем значению 0 соответствует размер канала по умолчанию.

Чтобы канал можно было использовать для IPC, должен существовать еще один процесс, и для этого процесса требуется один из дескрипторов канала. Предположим, например, что родительскому процессу, вызвавшему функцию CreatePipe, необходимо вывести данные, которые нужны дочернему процессу. Тогда возникает вопрос о том, как передать дочернему процессу дескриптор чтения (phRead). Родительский процесс осуществляет это, устанавливая дескриптор стандартного ввода в структуре STARTUPINFO для дочерней процедуры равным *phRead.

Чтение с использованием дескриптора чтения канала блокируется, если канал пуст. В противном случае в процессе чтения будет воспринято столько байтов, сколько имеется в канале, вплоть до количества, указанного при вызове функции ReadFile. Операция записи в заполненный канал, которая выполняется с использованием буфера в памяти, также будет блокирована.

Наконец, анонимные каналы обеспечивают только однонаправленное взаимодействие. Для двухстороннего взаимодействия необходимы два канала.

Пример: перенаправление ввода/вывода с использованием анонимного канала

В программе 11.1 представлен родительский процесс, который создает два процесса из командной строки и соединяет их каналом. Родительский процесс устанавливает канал и осуществляет перенаправление стандартного ввода/вывода. Обратите внимание на то, каким образом задается свойство наследования дескрипторов анонимного канала и как организуется перенаправление стандартного ввода/вывода на два дочерних процесса; эти методики описаны в главе 6.

Местоположение оператора WriteFile в блоке Program2 на рис. 11.1 справа предполагает, что программа считывает большой объем данных, обрабатывает их, и лишь после этого записывает результаты. Эту запись можно было бы осуществлять и внутри цикла, выводя результаты после каждого считывания.

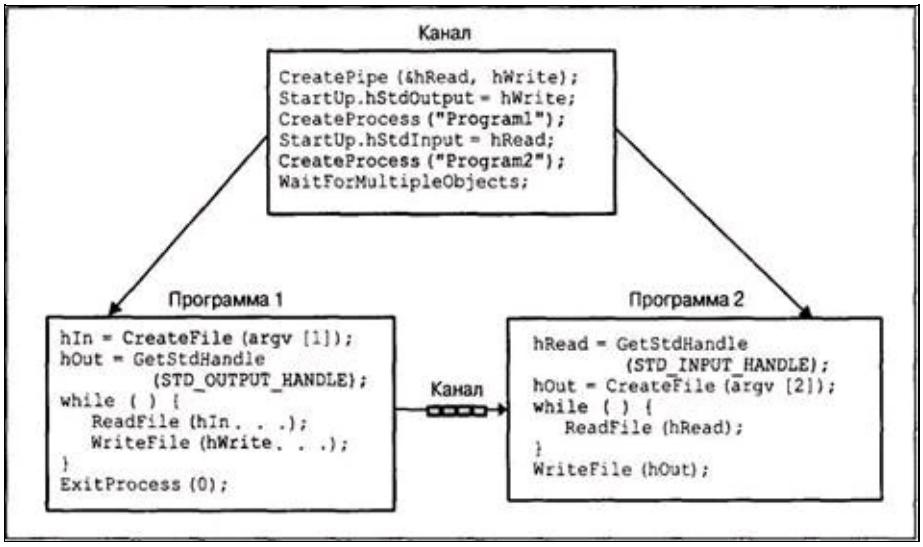


Рис. 11.1. Межпроцессное взаимодействие с использованием анонимного канала

Дескрипторы каналов и потоков должны закрываться при первой же возможности. На рис. 11.1 закрытие дескрипторов не отражено, однако это делается в программе 11.1. Родительский процесс должен закрыть дескриптор устройства стандартного вывода сразу же после создания первого дочернего процесса, чтобы второй процесс мог распознать метку конца файла, когда завершится выполнение первого процесса. В случае существования открытого дескриптора первого процесса второй процесс не смог бы завершиться, поскольку система не обозначила бы конец файла.

В программе 11.1 используется непривычный синтаксис: две команды, разделенные символом =, обозначающим канал. Использование для этой цели символа вертикальной черты (|) привело бы к возникновению конфликта с системным командным процессором. Рисунок 11.1 является схематическим представлением выполнения следующей команды:

```
$ pipe Program1 аргументы = Program2 аргументы
```

При использовании средств командного процессора UNIX или Windows соответствующая команда имела бы следующий вид:

```
$ Program1 аргументы | Program2 аргументы
```

Программа 11.1. pipe: межпроцессное взаимодействие с использованием анонимных каналов

```
    #include "EvryThng.h"
    int _tmain(int argc, LPTSTR argv[])
    /* Соединение двух команд с помощью канала в командной строке: pipe команда1 =
команда2 */
    {
     DWORD i = 0;
     HANDLE hReadPipe, hWritePipe;
     TCHAR Command1[MAX_PATH];
     SECURITY_ATTRIBUTES PipeSA = /* Для наследуемых дескрипторов. */
      {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
     PROCESS_INFORMATION ProcInfo1, ProcInfo2;
     STARTUPINFO StartInfoCh1, StartInfoCh2;
     LPTSTR targv = SkipArg(GetCommandLine());
     GetStartupInfo(&StartInfoCh1);
     GetStartupInfo(&StartInfoCh2);
     /* Найти символ "=", разделяющий две команды. */
     while (*targv != '=' && *targv != '\0') {
      Command1[i] = *targv;
      targv++;
      i++;
     }
     Command1[i] = '\0';
     /* Пропуск до начала второй команды. */
     targv = SkipArg(targv);
     CreatePipe(&hReadPipe, &hWritePipe, &PipeSA, 0);
     /* Перенаправить стандартный вывод и создать первый процесс. */
     StartInfoCh1.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
     StartInfoCh1.hStdError = GetStdHandle(STD_ERROR_HANDLE);
     StartInfoCh1.hStdOutput = hWritePipe;
     StartInfoCh1.dwFlags = STARTF_USESTDHANDLES;
      CreateProcess(NULL,  (LPTSTR)Command1,  NULL,  NULL,  TRUE  /*  Унаследовать
дескрипторы. */, 0, NULL, NULL, &StartInfoCh1, &ProcInfo1);
     CloseHandle(ProcInfo1.hThread);
      /* Закрыть дескриптор записи канала, поскольку он больше не нужен, чтобы
вторая команда могла обнаружить конец файла. */
     CloseHandle(hWritePipe);
     /* Повторить операции (симметричным образом) для второго процесса. */
     StartInfoCh2.hStdInput = hReadPipe;
     StartInfoCh2.hStdOutput = GetStdHandle(STD_OUTPUT_HANDLE);
     StartInfoCh2.hStdError = GetStdHandle(STD_ERROR_HANDLE);
     StartInfoCh2.dwFlags = STARTF_USESTDHANDLES;
      CreateProcess(NULL,  (LPTSTR)targv,  NULL,  NULL,  TRUE,  0,  NULL,  NULL,
&StartInfoCh2, &ProcInfo2);
     CloseHandle(ProcInfo2.hThread);
     CloseHandle(hReadPipe);
     /* Ожидать завершения первого и второго процессов. */
     WaitForSingleObject(ProcInfo1.hProcess, INFINITE);
     CloseHandle(ProcInfo1.hProcess);
     WaitForSingleObject(ProcInfo2.hProcess, INFINITE);
     CloseHandle(ProcInfo2.hProcess);
     return 0;
    }
```

Именованные каналы

Именованные каналы (named pipes) предлагают ряд возможностей, которые делают их полезными в качестве универсального механизма реализации приложений на основе IPC, включая приложения, требующие сетевого доступа к файлам, и клиент-серверные системы[31], хотя для реализации простых вариантов IPC, ориентированных на байтовые потоки, как в предыдущем примере, в котором взаимодействие процессов ограничивается рамками одной системы, анонимных каналов вам будет вполне достаточно. К числу упомянутых возможностей (часть которых обеспечивается дополнительно) относятся следующие:

• Именованные каналы ориентированы на обмен сообщениями, поэтому процесс, выполняющий чтение, может считывать сообщения переменной длины именно в том виде, в каком они были посланы процессом, выполняющим запись.

• Именованные каналы являются двунаправленными, что позволяет осуществлять обмен сообщениями между двумя процессами посредством единственного канала.

• Допускается существование нескольких независимых экземпляров канала, имеющих одинаковые имена. Например, с единственной серверной системой могут связываться одновременно несколько клиентов, использующих каналы с одним и тем же именем. Каждый клиент может иметь собственный экземпляр именованного канала, и сервер может использовать этот же канал для отправки ответа клиенту.

• Каждая из систем, подключенных к сети, может обратиться к каналу, используя его имя. Взаимодействие посредством именованного канала осуществляется одинаковым образом для процессов, выполняющихся как на одной и той же, так и на разных машинах.

• Имеется несколько вспомогательных и связных функций, упрощающих обслуживание взаимодействия "запрос/ответ" и клиент-серверных соединений.

Как правило, именованные каналы являются более предпочтительными по сравнению с анонимными, хотя программа 11.1 и рис. 11.1 и иллюстрируют ситуацию, в которой анонимные каналы оказываются исключительно полезными. Во всех случаях, когда требуется, чтобы канал связи был двунаправленным, ориентированным на обмен сообщениями или доступным для нескольких клиентских процессов, следует применять именованные каналы. Попытки реализации последующих примеров с использованием анонимных каналов натолкнулись бы на значительные трудности.

Использование именованных каналов

Функция CreateNamedPipe создает первый экземпляр именованного канала и возвращает дескриптор. При вызове этой функции указывается также максимально допустимое количество экземпляров каналов, а следовательно, и количество клиентов, одновременная поддержка которых может быть обеспечена.

Как правило, создающий процесс рассматривается в качестве *сервера. Клиентские процессы,* которые могут выполняться и на других системах, открывают канал с помощью функции CreateFile.

На рис. 11.2 в иллюстративной форме представлены отношения "клиент/сервер", а также псевдокод, отражающий одну из возможных схем применения именованных каналов. Обратите внимание, что сервер создает множество экземпляров одного и того же канала, каждый из которых обеспечивает поддержку одного клиента. Кроме того, для каждого экземпляра именованного канала сервер создает поток, так что для каждого клиента существует

выделенный поток и экземпляр именованного канала. Следовательно, рис. 11.2 показывает, как реализовать модель многопоточного сервера, впервые представленную на рис. 7.1.

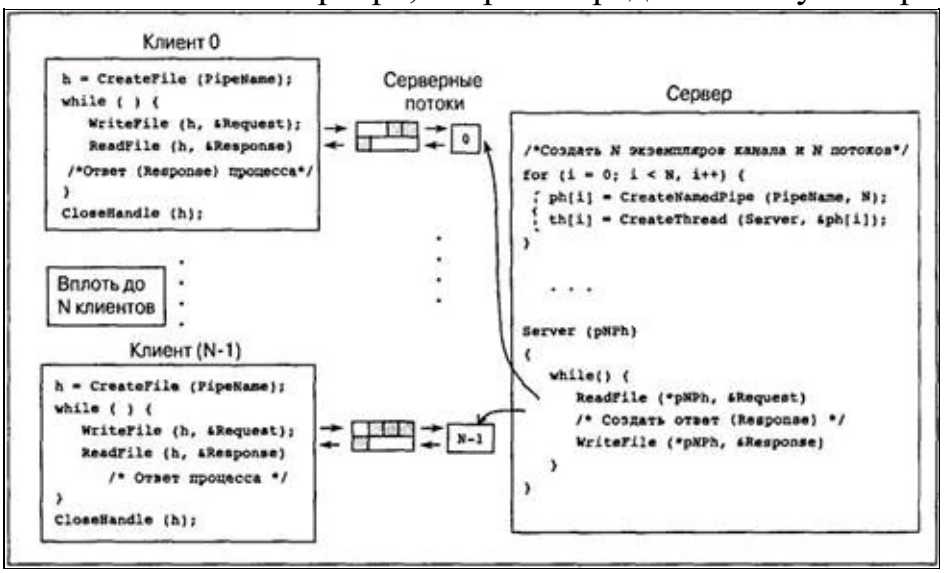


Рис. 11.2. Взаимодействие клиентов с сервером через именованные каналы

Создание именованных каналов

Серверами именованных каналов могут быть только системы на основе Windows NT (как обычно, здесь имеются в виду версия 4.0 и последующие); системы на базе Windows 9x могут выступать только в роли клиентов.

Прототип функции CreateNamedPipe представлен ниже.

```
HANDLE CreateNamedPipe(LPCTSTR lpName, DWORD dwOpenMode, DWORD
dwPipeMode, DWORD nMaxInstances, DWORD nOutBufferSize, DWORD nInBufferSize,
DWORD nDefaultTimeOut, LPSECURITY ATTRIBUTES lpSecurityAttributes)
```

Параметры

lpName — указатель на имя канала, который должен иметь следующую форму:
`\\.\pipe\[path]pipename`

Точка (.) обозначает локальный компьютер; таким образом, создать канал на удаленном компьютере невозможно.

dwOpenMode — указывает один из следующих флагов:

• PIPE_ACCESS_DUPLEX — этот флаг эквивалентен комбинации значений GENERIC_READ и GENERIC_WRITE.

• PIPE_ACCESS_INBOUND — данные могут передаваться только в направлении от клиента к серверу; эквивалентно GENERIC_READ.

• PIPE_ACCESS_OUTBOUND — этот флаг эквивалентен GENERIC_WRITE.

При задании режима могут также указываться значения FILE_FLAG_WRITE_THROUGH (не используется с каналами сообщений) и FILE_FLAG_OVERLAPPED (перекрывающиеся операции рассматриваются в главе 14).

dwPipeMode — имеются три пары взаимоисключающих значений этого параметра. Эти значения указывают, ориентирована ли запись на работу с сообщениями или байтами, ориентировано ли чтение на работу с сообщениями или блоками, и блокируются ли операции чтения.

• PIPE_TYPE_BYTE и PIPE_TYPE_MESSAGE — указывают, соответственно, должны ли данные записываться в канал как поток байтов или как сообщения. Для всех экземпляров каналов с одинаковыми именами следует использовать одно и то же значение.

• PIPE_READMODE_BYTE и PIPE_READMODE_MESSAGE — указывают, соответственно, должны ли данные считываться как поток байтов или как сообщения. Значение PIPE_READMODE_MESSAGE требует использования значения PIPE_TYPE_MESSAGE.

• PIPE_WAIT и PIPE_NOWAIT — определяют, соответственно, будет или не будет блокироваться операция ReadFile. Следует использовать значение PIPE_WAIT, поскольку для обеспечения асинхронного ввода/вывода существуют лучшие способы.

nMaxInstances — определяет количество экземпляров каналов, а следовательно, и количество одновременно поддерживаемых клиентов. Как показано на рис. 11.2, при каждом вызове функции CreateNamedPipe для данного канала должно использоваться одно и то же значение. Чтобы предоставить ОС возможность самостоятельно определить значение этого параметра на основании доступных системных ресурсов, следует указать значение PIPE_UNLIMITED_INSTANCES.

nOutBufferSize и nInBufferSize — позволяют указать размеры (в байтах) выходного и входного буферов именованных каналов. Чтобы использовать размеры буферов по умолчанию, укажите значение 0.

nDefaultTimeOut — длительность интервала ожидания по умолчанию (в миллисекундах) для функции WaitNamedPipe, которая обсуждается в следующем разделе. Эта ситуация, в которой функция, создающая объект, устанавливает интервал ожидания для родственной функции, является уникальной.

В случае ошибки возвращается значение INVALID_HANDLE_VALUE, поскольку дескрипторы каналов аналогичны дескрипторам файлов. При попытке создания именованного канала под управлением Windows 9x, которая не может выступать в качестве сервера именованных каналов, возвращаемым значением будет NULL, что может стать причиной недоразумений.

lpSecurityAttributes — имеет тот же смысл, что и в случае любой функции, создающей объект.

При первом вызове функции CreateNamedPipe происходит создание самого именованного канала, а не просто его экземпляра. Закрытие последнего открытого дескриптора экземпляра именованного канала приводит к уничтожению этого экземпляра (обычно существует по одному дескриптору на каждый экземпляр). Уничтожение последнего экземпляра именованного канала приводит к уничтожению самого канала, в результате чего имя канала становится вновь доступным для повторного использования.

Подключение клиентов именованных каналов

Как показано на рис. 11.2, для подключения клиента к именованному каналу применяется функция CreateFile, при вызове которой указывается имя именованного канала. Часто клиент и сервер выполняются на одном компьютере, и в этом случае для указания имени канала используется следующая форма:

```
\\.\pipe\[path]pipename
```

Если сервер находится на другом компьютере, для указания имени канала используется следующая форма:

```
\\servername\pipe\[path]pipename
```

Использование точки (.) вместо имени локального компьютера в случае, когда сервер

является локальным, позволяет значительно сократить время подключения.

Функции состояния именованных каналов

Предусмотрены две функции, позволяющие получать информацию о состоянии каналов, и еще одна функция, позволяющая устанавливать данные состояния канала. Краткая характеристика этих функций приводится ниже, а одна из этих функций используется в программе 11.2.

• GetNamedPipeHandleState — возвращает для заданного открытого дескриптора информацию относительно того, работает ли канал в блокируемом или неблокируемом режиме, ориентирован ли он на работу с сообщениями или байтами, каково количество экземпляров канала и тому подобное.

• SetNamedPipeHandleState — позволяет программе устанавливать атрибуты состояния. Параметр режима (NpMode) передается не по значению, а по адресу, что может стать причиной недоразумений. Применение этой функции демонстрируется в программе 11.2.

• GetNamedPipeInfo — определяет, принадлежит ли дескриптор экземпляру клиента или сервера, размеры буферов и тому подобное.

Функции подключения именованных каналов

После создания именованного канала сервер может ожидать подключения клиента (осуществляемого с помощью функции CreateFile или функции CallNamedFile, описанной далее в этой главе), используя для этого функцию ConnectNamedPipe, которая является серверной функцией лишь в случае Windows NT:

```
Bool ConnectNamedPipe(HANDLE hNamedPipe, LPOVERLAPPED lpOverlapped)
```

Если параметр lpOverlapped установлен равным NULL, то функция ConnectNamedPipe осуществляет возврат сразу же после установления соединения с клиентом. В случае успешного выполнения функции возвращаемым значением является TRUE. Если же подключение клиента происходит в течение промежутка времени между вызовами сервером функций CreateNamedPipe и ConnectNamed-Pipe, то возвращаемым значением будет FALSE. В этом случае функция GetLastError вернет значение ERROR_PIPE_CONNECTED.

После возврата из функции ConnectNamedPipe сервер может выполнять чтение запросов с помощью функции ReadFile и запись ответов посредством функции WriteFile. Наконец, сервер должен вызвать функцию DisconnectNamedPipe, чтобы освободить дескриптор (экземпляра канала) для соединения с другим возможным клиентом.

Последняя функция, WaitNamedPipe, используется клиентами для синхронизации соединений с сервером. Функция осуществляет успешный возврат, когда на сервере имеется незавершенный вызов функции ConnectNamedPipe, указывающий на наличие доступного экземпляра именованного канала. Используя WaitNamedPipe, клиент имеет возможность убедиться в том, что сервер готов к образованию соединения, после чего может вызвать функцию CreateFile. Вместе с тем, вызов клиентом функции CreateFile может завершиться ошибкой, если в это же время другой клиент открывает экземпляр именованного канала или дескриптор экземпляра закрывается сервером. При этом неудачного завершения вызванной сервером функции ConnectNamedPipe не произойдет. Заметьте, что для функции WaitNamedPipe предусмотрен интервал ожидания, который, если он указан, отменяет значение интервала

ожидания, заданного при вызове серверной функции CreateNamedPipe.

Подключение клиентов и серверов именованных каналов

Операции по подключению клиентов и серверов к именованным каналам выполняются в описанном ниже порядке. Сначала мы рассмотрим последовательность операций, выполняемых сервером, при помощи которых сервер создает соединение с клиентом, взаимодействует с клиентом до тех пор, пока тот не разорвет соединение (вынуждая функцию ReadFile вернуть значение FALSE), разрывает соединение на стороне сервера, а затем образует соединение с другим клиентом:

```
    /* Последовательность операций при создании соединения с использованием именованного канала для сервера. */
    hNp = CreateNamedPipe("\\\\.\\pipe\\my_pipe", …);
    while (… /* Цикл продолжается вплоть до завершения работы сервера.*/) {
     ConnectNamedPipe(hNp, NULL);
     while (ReadFile(hNp, Request, …) {
      …
      WriteFile(hNp, Response, …);
     }
     DisconnectNamedPipe(hNp);
    }
    CloseHandle(hNp);
```

Перейдем к рассмотрению последовательности операций, выполняемых клиентом, в которой клиент прекращает выполнение после завершения работы, давая возможность подключиться к тому же экземпляру именованного канала другому клиенту. Как показано ниже, клиент может соединиться с сервером в сети, если ему известно сетевое имя сервера (ServerName):

```
    /* Последовательность операций при создании соединения с использованием именованного канала для клиента. */
    WaitNamedPipe("\\\\ServerName\\pipe\\my_pipe", NMPWAIT_WAIT_FOREVER);
    hNp = CreateFile("\\\\ServerName\\pipe\\my_pipe", …);
    while (…/*Цикл выполняется до тех пор, пока не прекратятся запросы.*/ {
     WriteFile(hNp, Request, …);
     …
     ReadFile(hNp, Response);
    }
    CloseHandle (hNp); /* Разорвать соединение с сервером. */
```

Обратите внимание, что клиент и сервер состязаются за ресурсы. Прежде всего, клиентский вызов функции WaitNamedPipe завершится ошибкой, если именованный канал к этому моменту еще не был создан сервером; для краткости тестирование успешности выполнения в нашем примере опущено, однако оно включено в примеры программ, доступные на Web-сайте. Далее, в редких случаях вызов CreateFile может быть выполнен еще до того, как сервер вызовет функцию ConnectNamedPipe. В этом случае функция ConnectNamedPipe вернет серверу значение FALSE, однако взаимодействия посредством именованного канала по-прежнему будет функционировать надлежащим образом.

Экземпляр именованного канала является глобальным ресурсом, поэтому, когда клиент разрывает соединение с сервером, к нему может подключиться другой клиент.

Функции транзакций именованных каналов

На рис. 11.2 показана типичная конфигурация клиента, в которой клиент выполняет следующие операции:

• Открывает экземпляр канала, создавая долговременное соединение с сервером и занимая экземпляр канала.

• Периодически посылает запросы и ожидает получения ответов.

• Закрывает соединение.

Встречающуюся здесь последовательность вызовов функций WriteFile и ReadFile можно рассматривать как единую клиентскую транзакцию, и Windows предоставляет соответствующую функцию для каналов сообщений:

```
BOOL TransactNamedPipe(HANDLE hNamedPipe, LPVOID lpWriteBuf, DWORD cbWriteBuf, LPVOID lpReadBuf, DWORD cbReadBuf, LPDWORD lpcbRead, LPOVERLAPPED lpOverlapped)
```

Смысл всех параметров здесь должен быть ясен, поскольку данная функция сочетает в себе функции WriteFile и ReadFile, применяемые к дескриптору именованного канала. Указываются как выходной, так и входной буфер, а разыменованный указатель lpcbRead предоставляет размер сообщения. Перекрывающиеся операции (глава 14) возможны, однако в более типичных случаях функция ожидает ответа.

Функция TransactNamedPipe удобна в использовании, однако, как показывает рис. 11.2, она требует создания постоянного соединения, что ограничивает число возможных клиентов[32].

Ниже приводится прототип второй клиентской вспомогательной функции.

```
BOOL CallNamedPipe(LPCTSTR lpPipeName, LPVOID lpWriteBuf, DWORD cbWriteBuf, LPVOID lpReadBuf, DWORD cbReadBuf, LPDWORD lpcbRead, DWORD dwTimeOut)
```

Функция CallNamedPipe не требует образования постоянного соединения; вместо этого она создает временное соединение, объединяя в себе выполнение следующей последовательности операций:

```
CreateFile
WriteFile
ReadFile
CloseHandle
```

Преимуществом такого способа является лучшее использование канала за счет снижения накладных расходов системных ресурсов на один запрос.

Смысл параметров этой функции тот же, что и в случае функции TransactNamedPipe, если не считать того, что вместо дескриптора для указания канала используется его имя. Функция CallNamedPipe выполняется синхронном режиме (отсутствует структура OVERLAPPED). Указываемая при ее вызове длительность периода ожидания (dwTimeOut) (в миллисекундах) относится к соединению, а не транзакции. Параметр dwTimeOut имеет три специальных значения:

• NMPWAIT_NOWAIT

• NMPWAIT_WAIT_FOREVER

• NMPWAIT_USE_DEFAULT_WAIT, которое приводит к использованию интервала ожидания по умолчанию, заданного в вызове функции CreateNamedPipe.

Определение наличия сообщений в именованных каналах

В дополнение к возможности чтения данных из именованного канала с помощью функции ReadFile можно также определить, имеются ли в канале фактические сообщения, используя для этого функцию PeekNamedPipe. Это средство может быть использовано для опроса именованного канала (неэффективная операция), определения размера сообщения, чтобы распределить память для буфера перед выполнением чтения, или просмотра поступающих сообщений с целью назначения им приоритетов для последующей обработки.

```
BOOL PeekNamedPipe(HANDLE hPipe, LPVOID lpBuffer, DWORD cbBuffer,
LPDWORD lpcbRead, LPDWORD lpcbAvail, LPDWORD lpcbMessage)
```

Функция PeekNamedPipe обеспечивает считывание любого байта или сообщения из канала без их удаления, но ее невозможно блокировать, и она осуществляет возврат сразу же по завершении выполнения.

Чтобы определить, имеются ли в канале данные, необходимо проверить значение *lpcbAvail; если данные в канале присутствуют, оно должно быть больше 0. В этом случае параметры lpBuffer и lpcbRead могут иметь значения NULL. Если же буфер определен параметрами lpBuffer и cbBuffer, то значение *lpcbMessage укажет вам, остается ли еще некоторое количество байтов сообщений, которые не умещаются в буфере, что позволяет распределять буфер большего размера, прежде чем осуществлять чтение из именованного канала. Для канала, работающего в режиме считывания байтов, это значение равно 0.

Следует помнить, что функция PeekNamedPipe осуществляет чтение, не уничтожая данные, и поэтому для удаления сообщений или байтовых данных из канала требуется последующее применение функции ReadFile.

Каналы UNIX FIFO аналогичны именованным каналам и, таким образом, обеспечивают взаимодействие не связанных между собой процессов. Однако по сравнению с именованными каналами Windows их возможности являются несколько ограниченными:

- Каналы FIFO являются полудуплексными.
- Каналы FIFO действуют только в пределах одного компьютера.
- Каналы FIFO ориентированы на работу с байтами, поэтому в клиент-серверных приложениях проще всего использовать записи фиксированной длины. Тем не менее, отдельные операции чтения и записи являются атомарными.

Сервер, на котором применяется это средство, должен использовать для каждого ответа клиентам отдельный канал FIFO, хотя все клиенты могут посылать запросы по одному и тому же известному каналу. В соответствии с общепринятой практикой клиенты включают имя канала FIFO в запрос соединения.

Функция UNIX mkfifo является ограниченной версией функции CreateNamedFile.

Если клиенты и сервер должны находиться в сети, используйте сокеты или аналогичный механизм транспортировки сетевых сообщений. Сокеты работают в дуплексном режиме, однако требуют использования отдельного соединения для каждого клиента.

Пример: клиент-серверный процессор командной строки

Теперь мы располагаем всем необходимым для построения клиент-серверной системы, работающей с запросами и ответами. В данном примере будет представлен сервер командной строки, выполняющий команду по требованию клиента. Система характеризуется следующими особенностями:

• С сервером могут взаимодействовать несколько клиентов.

• Клиенты могут находиться на различных системах в сети, хотя допускается и их расположение на компьютере сервера.

• Сервер является многопоточным, причем каждому именованному каналу назначается отдельный поток. Это означает, что существует пул потоков (thread pool), в который входят рабочие потоки, готовые к использованию подключающимися клиентами. Рабочие потоки предоставляются клиентам посредством экземпляра именованного канала, который система выделяет клиенту.

• Отдельные потоки сервера в каждый момент времени обрабатывают один запрос, что упрощает управление параллелизмом их выполнения. Каждый из потоков самостоятельно обрабатывает свои запросы. Тем не менее, требуется предпринимать обычные меры предосторожности на тот случай, если несколько различных потоков сервера пытаются получить доступ к одному и тому же файлу или иному ресурсу.

В программе 11.2 представлен однопоточной клиент, а в программе 11.3 — его сервер. Сервер соответствует модели, представленной на рисунках 7.1 и 11.2. Запросом клиента является обычная командная строка. Ответом сервера является результирующий вывод, который посылается в виде нескольких сообщений. Кроме того, в программе используется находящийся на Web-сайте заголовочный файл ClntSrvr.h, в котором определены структуры данных запроса и ответа, а также имена каналов клиента и сервера.

В программе 11.2 клиент вызывает функцию LocateServer, которая находит имя канала сервера. Функция LocateServer использует почтовый ящик (mailslot), описанный в одном из последующих разделов и представленный в программе 11.5.

В объявлениях записей имеются поля длины, тип которых определен как DWORD32; это сделано для того, чтобы программы, получая возможность их последующего перенесения на платформу Win64, могли взаимодействовать с серверами и клиентами, выполняющимися под управлением любой системы Windows.

Программа 11.2. clientNP: клиент, ориентированный на соединение посредством именованного канала

```
/* Глава 11. Клиент-серверная система. ВЕРСИЯ КЛИЕНТА.
   clientNP — клиент, ориентированный на установку соединения. */
/* Выполнить командную строку (на сервере); отобразить ответ. */
/* Клиент создает долговременное соединение с сервером (захватывая */
/* экземпляр канала) и выводит приглашение пользователю для ввода команд.*/

#include "EvryThng.h"
#include "ClntSrvr.h" /* Определяет структуры записей запроса и ответа. */

int _tmain(int argc, LPTSTR argv[]) {
 HANDLE hNamedPipe = INVALID_HANDLE_VALUE;
```

```
     TCHAR PromptMsg[] = _T("\nВведите команду: ");
     TCHAR QuitMsg[] = _T("$Quit");
     TCHAR ServerPipeName[MAX_PATH];
     REQUEST Request; /* См. файл ClntSrvr.h. */
     RESPONSE Response; /* См. файл ClntSrvr.h. */
     DWORD nRead, nWrite, NpMode = PIPE_READMODE_MESSAGE | PIPE_WAIT;
     LocateServer(ServerPipeName);
     /* Ожидать появления экземпляра именованного канала и "вступить в борьбу" за
право его открытия. */
     while (INVALID_HANDLE_VALUE == hNamedPipe) {
      WaitNamedPipe(ServerPipeName, NMPWAIT_WAIT_FOREVER);
        hNamedPipe = CreateFile(ServerPipeName, GENERIC_READ | GENERIC_WRITE, 0,
NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
     }
     /* Задать блокирование дескриптора именованного канала; режим сообщений.*/
     SetNamedPipeHandleState(hNamedPipe, &NpMode, NULL, NULL);
     /* Вывести приглашение пользователю для ввода команд. Завершить выполнение по
получении команды "$quit." */
      while (ConsolePrompt(PromptMsg, Request.Record, MAX_RQRS_LEN, TRUE) &&
(_tcscmp(Request.Record, QuitMsg) != 0)) {
      WriteFile(hNamedPipe, &Request, RQ_SIZE, &nWrite, NULL);
      /* Считать каждый ответ и направить его на стандартный вывод.
          Response.Status = 0 означает "конец ответного сообщения." */
         while (ReadFile(hNamedPipe, &Response, RS_SIZE, &nRead, NULL) &&
(Response.Status == 0)) _tprintf(_T("%s"), Response.Record);
     }
     _tprintf(_T("Получена команда завершения работы. Соединение разрывается."));
     CloseHandle(hNamedPipe);
     return 0;
    }
```

Программа 11.3 — это серверная программа, включающая функцию потока сервера, которая обрабатывает запросы, генерируемые с помощью программы 11.2. Кроме того, сервер создает "широковещательный серверный поток" ("server broadcast thread") (см. программу 11.4), который используется для широковещательной рассылки имени своего канала всем клиентам, желающим подключиться, посредством почтового ящика. В программе 11.2 вызывается функция LocateServer, представленная в программе 11.5, которая считывает информацию, отправленную данным процессом. Почтовые ящики описываются далее в настоящей главе.

Хотя соответствующий код и не включен в программу 11.4, в ней предусмотрена возможность защиты сервером (представлен на Web-сайте) своего именованного канала с целью предотвращения доступа к нему клиентов, не имеющих должных полномочий. Вопросы безопасности объектов рассматриваются в главе 15, где будет также показано, как использовать указанную возможность.

Программа 11.3. serverNP: многопоточный сервер именованного канала

```
    /* Глава 11. ServerNP. */
    /* Многопоточный сервер командной строки. Версия на основе именованных каналов.
*/

    #include "EvryThng.h"
    #include "ClntSrvr.h" /* Определения сообщений запроса и ответа. */

    typedef struct { /* Аргумент серверного потока. */
     HANDLE hNamedPipe; /* Экземпляр именованного канала. */
```

```
    DWORD ThreadNo;
    TCHAR TmpFileName[MAX_PATH]; /* Имя временного файла. */
   } THREAD_ARG;

   typedef THREAD_ARG *LPTHREAD_ARG;
   volatile static BOOL ShutDown = FALSE;
   static DWORD WINAPI Server(LPTHREAD_ARG);
   static DWORD WINAPI Connect(LPTHREAD_ARG);
   static DWORD WINAPI ServerBroadcast(LPLONG);
   static BOOL WINAPI Handler(DWORD);
   static TCHAR ShutRqst[] = _T("$ShutDownServer");

   _tmain(int argc, LPTSTR argv[]) {
    /* Определение MAX_CLIENTS содержится в файле ClntSrvr.h. */
    HANDLE hNp, hMonitor, hSrvrThread[MAXCLIENTS];
    DWORD iNp, MonitorId, ThreadId;
    LPSECURITY_ATTRIBUTES pNPSA = NULL;
    THREAD_ARG ThArgs[MAXCLIENTS];
     /* Обработчик управляющих сигналов консоли, используемый для остановки
сервера. */
    SetConsoleCtrlHandler(Handler, TRUE);
    /* Периодически создавать имя широковещательного канала потока. */
     hMonitor = (HANDLE)_beginthreadex(NULL, 0, ServerBroadcast, NULL, 0,
&MonitorId);
    /* Создать экземпляр канала и временный файл для каждого серверного потока. */
    for (iNp = 0; iNp < MAX_CLIENTS; iNp++) {
     hNp = CreateNamedPipe(SERVER_PIPE, PIPE_ACCESS_DUPLEX, PIPE_READMODE_MESSAGE
| PIPE_TYPE_MESSAGE | PIPE_WAIT, MAXCLIENTS, 0, 0, INFINITE, pNPSA);
     ThArgs[iNp].hNamedPipe = hNp;
     ThArgs[iNp].ThreadNo = iNp;
     GetTempFileName(_T("."), _T("CLP"), 0, ThArgs[iNp].TmpFileName);
      hSrvrThread[iNp] = (HANDLE)_beginthreadex(NULL, 0, Server, &ThArgs[iNp], 0,
&ThreadId);
    }
    /* Ждать завершения выполнения всех потоков. */
    WaitForMultipleObjects(MAXCLIENTS, hSrvrThread, TRUE, INFINITE);
    WaitForSingleObject(hMonitor, INFINITE);
    CloseHandle(hMonitor);
    for (iNp = 0; iNp < MAXCLIENTS; iNp++) {
     /* Закрыть дескрипторы канала и удалить временные файлы. */
     CloseHandle(hSrvrThread[iNp]);
     DeleteFile(ThArgs[iNp].TmpFileName);
    }
    _tprintf(_T("Серверный процесс завершил выполнение.\n"));
    return 0;
   }

   static DWORD WINAPI Server(LPTHREAD_ARG pThArg)
   /* Функция потока сервера; по одной для каждого потенциального клиента. */
   {
    HANDLE hNamedPipe, hTmpFile = INVALID_HANDLE_VALUE, hConTh, hClient;
    DWORD nXfer, ConThId, ConThStatus;
    STARTUPINFO StartInfoCh;
    SECURITY_ATTRIBUTES TempSA = {sizeof(SECURITY_ATTRIBUTES), NULL, TRUE};
    PROCESS_INFORMATION ProcInfo;
    FILE *fp;
    REQUEST Request;
    RESPONSE Response;
    GetStartupInfo(&StartInfoCh);
```

```
    hNamedPipe = pThArg->hNamedPipe;
     hTmpFile  =  CreateFile(pThArg->TmpFileName,  GENERIC_READ  |  GENERIC_WRITE,
FILE_SHARE_READ       |       FILE_SHARE_WRITE,       &TempSA,       CREATE_ALWAYS,
FILE_ATTRIBUTE_TEMPORARY, NULL);
    while (!ShutDown) { /* Цикл соединений. */
     /* Создать поток соединения; ждать его завершения. */
     hConTh = (HANDLE)_beginthreadex(NULL, 0, Connect, pThArg, 0, &ConThId);
     /* Ожидание соединения с клиентом и проверка флага завершения работы.*/
      while (!ShutDown && WaitForSingleObject(hConTh, CS_TIMEOUT) == WAIT_TIMEOUT)
{ /* Пустое тело цикла. */ };
     CloseHandle(hConTh);
     if (ShutDown) continue; /*Флаг может быть установлен любым потоком.*/
     /* Соединение существует. */
     while (!ShutDown && ReadFile(hNamedPipe, &Request, RQ_SIZE, &nXfer, NULL)) {
      /* Получать новые команды до отсоединения клиента. */
      ShutDown = ShutDown || (_tcscmp(Request.Record, ShutRqst) == 0);
      if (ShutDown) continue; /* Проверяется на каждой итерации. */
      /* Создать процесс для выполнения команды. */
      StartInfoCh.hStdOutput = hTmpFile;
      StartInfoCh.hStdError = hTmpFile;
      StartInfoCh.hStdInput = GetStdHandle(STD_INPUT_HANDLE);
      StartInfoCh.dwFlags = STARTF_USESTDHANDLES;
        CreateProcess(NULL,  Request.Record,  NULL,  NULL,  TRUE,  /*  Унаследовать
дескрипторы. */
       0, NULL, NULL, &StartInfoCh, &ProcInfo);
      /* Выполняется процесс сервера. */
      CloseHandle(ProcInfo.hThread);
      WaitForSingleObject(ProcInfo.hProcess, INFINITE);
      CloseHandle(ProcInfo.hProcess);
       /* Отвечать по одной строке за один раз. Здесь удобно использовать функции
библиотеки C для работы со строками. */
      fp = _tfopen(pThArg->TmpFileName, _T("r"));
      Response.Status = 0;
              while(_fgetts(Response.Record,    MAX_RQRS_LEN,    fp)    !=    NULL)
WriteFile(hNamedPipe, &Response, RS_SIZE, &nXfer, NULL);
      FlushFileBuffers(hNamedPipe);
      fclose(fp);
      /* Уничтожить содержимое временного файла. */
      SetFilePointer(hTmpFile, 0, NULL, FILE_BEGIN);
      SetEndOfFile(hTmpFile);
      /* Отправить признак конца ответа. */
      Response.Status = 1;
      strcpy(Response.Record, "");
      WriteFile(hNamedPipe, &Response, RS_SIZE, &nXfer, NULL);
     }
     /* Конец основного командного цикла. Получить следующую команду. */
     /* Принудительно завершить выполнение потока, если он все еще активен.*/
     GetExitCodeThread(hConTh, &ConThStatus);
     if (ConThStatus == STILL_ACTIVE) {
       hClient  =  CreateFile(SERVER_PIPE,  GENERIC_READ  |  GENERIC_WRITE,  0,  NULL,
OPEN EXISTING, FILE ATTRIBUTE NORMAL, NULL);
      if (hClient != INVALID_HANDLE_VALUE) CloseHandle (hClient);
      WaitForSingleObject (hConTh, INFINITE);
     }
     /* Клиент отсоединился или имеется запрос останова. */
     FlushFileBuffers(hNamedPipe);
     DisconnectNamedPipe(hNamedPipe);
    }
    /* Конец командного цикла. Освободить ресурсы; выйти из потока. */
    if (hTmpFile != INVALID_HANDLE_VALUE) CloseHandle(hTmpFile);
```

```
 DeleteFile(pThArg->TmpFileName);
 _tprintf(_T("Выход из потока номер %d\n"), pThArg->ThreadNo);
 _endthreadex(0);
}

static DWORD WINAPI Connect(LPTHREAD_ARG pThArg) {
 /* Поток соединения разрешает серверу опрос флага ShutDown. */
 ConnectNamedPipe(pThArg->hNamedPipe, NULL);
 _endthreadex(0);
 return 0;
}

BOOL WINAPI Handler(DWORD CtrlEvent) {
 /* Завершить работу системы. */
 ShutDown = TRUE;
 return TRUE;
}
```

Комментарии по поводу клиент-серверного процессора командной строки

Данное решение характеризуется рядом особенностей и ограничений, которые будут обсуждаться в последующих главах.

• Соединяться с сервером и выполнять параллельные запросы могут сразу несколько серверов; каждому клиенту назначается серверный (или рабочий) поток, выделяемый из пула потоков.

• Сервер и клиенты могут выполняться либо в ответ на отдельные подсказки командной строки, либо под управлением программы JobShell (программа 6.3).

• Если во время попыток клиента соединиться с сервером все экземпляры именованного канала оказываются задействованными, то новый клиент будет находиться в состоянии ожидания до тех пор, пока другой клиент не разорвет соединение в ответ на получение команды $Quit, тем самым делая его доступным для ожидающего клиента. Возможны ситуации, когда сразу несколько новых клиентов будут одновременно пытаться создать соединение с сервером, соревнуясь между собой за право открытия доступного экземпляра; потоки, проигравшие в этой конкурентной борьбе, будут вынуждены вновь перейти в состояние ожидания.

• Каждый серверный поток выполняет синхронные операции ввода/вывода, но одни из этих потоков могут обрабатывать запросы, в то время как другие — ожидать соединения или поступления клиентских запросов.

• С учетом ограничений, свойственных именованным каналам, о чем говорилось ранее в этой главе, расширение программы на случай сетевых клиентов не составляет труда. Для этого достаточно заменить имена каналов в заголовочном файле или добавить параметр, указывающий имя сервера в командной строке клиента.

• Каждый рабочий поток сервера создает простой поток, осуществляющий соединение, который вызывает функцию ConnectNamedPipe и завершает выполнение сразу же после подключения клиента. Это позволяет организовать ожидание дескриптора потока соединения рабочим потоком с использованием конечного интервала ожидания и периодическое тестирование глобального флага завершения работы (ShutDown). Если бы рабочие потоки блокировались при выполнении функции ConnectNamedPipe, они не могли бы тестировать этот флаг, и сервер не мог бы завершить работу. По этой причине поток сервера осуществляет вызов CreateFile, используя дескриптор именованного канала, чтобы заставить поток соединения возобновиться и завершить выполнение. Альтернативным вариантом было бы использование асинхронного ввода/вывода (глава 14), что дало бы возможность связать событие с вызовом функции ConnectNamedPipe. Другие возможные варианты реализации и дополнительная информация предоставляются в комментариях к исходному тексту программы, размещенному на Web-сайте книги. Без этого решения потоки соединения могли бы никогда не завершить работу самостоятельно, что привело бы к утечке ресурсов в DLL. Этот вопрос обсуждается в главе 12.

• Существует ряд благоприятных предпосылок для усовершенствования данной системы. Например, можно предусмотреть опцию выполнения внутрипроцессного сервера (in-process server), используя библиотеку DLL, которая реализует некоторые из команд. Это усовершенствование вводится в программу в главе 12.

• Количество серверных потоков ограничивается при вызове функции WaitForMultipleObjects в основном потоке. Хотя это ограничение легко преодолимо, в данном

случае система не обладает истинной масштабируемостью; как было показано в главе 10, чрезмерное увеличение количества потоков может оказать отрицательное влияние на производительность. В главе 14 для решения этой проблемы используются порты асинхронного ввода/вывода.

Почтовые ящики

Как и именованные каналы, почтовые ящики (mailslots) Windows снабжаются именами, которые могут быть использованы для обеспечения взаимодействия между независимыми каналами. Почтовые ящики представляют собой широковещательный механизм, основанный на дейтаграммах (описаны в главе 12), и ведут себя иначе по сравнению с именованными каналами, что делает их весьма полезными в ряде ограниченных ситуаций, которые, тем не менее, представляют большой интерес. Из наиболее важных свойств почтовых ящиков можно отметить следующие:

• Почтовые ящики являются однонаправленными.

• С одним почтовым ящиком могут быть связаны несколько записывающих программ (writers) и несколько считывающих программ (readers), но они часто связаны между собой отношениями "один ко многим" в той или иной форме.

• Записывающей программе (клиенту) не известно достоверно, все ли, только некоторые или какая-то одна из программ считывания (сервер) получили сообщение.

• Почтовые ящики могут находиться в любом месте сети.

• Размер сообщений ограничен.

Использование почтовых ящиков требует выполнения следующих операций:

• Каждый сервер создает дескриптор почтового ящика с помощью функции CreateMailSlot.

• После этого сервер ожидает получения почтового сообщения, используя функцию ReadFile.

• Клиент, обладающий только правами записи, должен открыть почтовый ящик, вызвав функцию CreateFile, и записать сообщения, используя функцию WriteFile. В случае отсутствия ожидающих программ считывания попытка открытия почтового ящика завершится ошибкой (наподобие "имя не найдено").

Сообщение клиента может быть прочитано *всеми* серверами; все серверы получают одно и то же сообщение.

Существует еще одна возможность. В вызове функции CreateFile клиент может указать имя почтового ящика в следующем виде:

```
\\*\mailslot\mailslotname
```

При этом символ звездочки (*) действует в качестве группового символа (wildcard), и клиент может обнаружить любой сервер в пределах *имени домена* — группы систем, объединенных общим именем, которое назначается администратором сети.

Использование почтовых ящиков

Рассмотренный перед этим клиент-серверный процессор командной строки предполагает несколько возможных способов его использования. Рассмотрим один из сценариев, в котором решается задача обнаружения сервера в только что упомянутой клиент-серверной системе (программы 11.2 и 11.3).

Сервер приложения (application server), действуя в качестве *почтового клиента* (mailslot client), периодически осуществляет широковещательную рассылку своего имени и имени именованного канала. Любой *клиент приложения* (application client), которому требуется найти сервер, может получить это имя, действуя в качестве *сервера почтовых ящиков* (mailslot server). Аналогичным образом сервер командной строки может периодически осуществлять широковещательную рассылку своего состояния, включая информацию о коэффициенте

использования, клиентам. Это соответствует ситуации, в которой имеется одна записывающая программа (почтовый клиент) и несколько считывающих программ (почтовых серверов). Если бы почтовых клиентов (то есть серверов приложения) было несколько, то ситуация описывалась бы отношением типа "многие ко многим".

Возможен и другой вариант, когда одна считывающая программа получает сообщения от многочисленных записывающих программ, которые, например, предоставляют информацию о своем состоянии. Этот вариант, соответствующий, например, электронной доске объявлений, оправдывает использование термина *почтовый ящик.* Оба описанных варианта использования — широковещательная рассылка имени и информации о состоянии — могут быть объединены, чтобы клиент мог выбирать наиболее подходящий сервер.

Обмен ролями терминов *клиент* и *сервер* в данном контексте может несколько сбивать с толку, однако заметьте, что сервер именованного канала и почтовый сервер выполняют вызовы функций CreateNamedPipe (или CreateMailSlot), тогда как клиент (именованного канала или почтового ящика) создает соединение, используя функцию CreateFile. Кроме того, в обоих случаях первый вызов функции WriteFile выполняется клиентом, а первый вызов функции ReadFile выполняется сервером.

Использование почтовых ящиков в соответствии с первым из описанных возможных вариантов иллюстрируется на рис. 11.3.

Создание и открытие почтового ящика

Для создания почтового ящика и получения дескриптора, который можно будет использовать в операциях ReadFile, почтовые серверы (программы считывания) вызывают функцию CreateMailslot. На одном компьютере может находиться только один почтовый ящик с данным именем, но один и тот же почтовый ящик может использоваться несколькими системами в сети, что обеспечивает возможность работы с ним нескольких программ считывания.

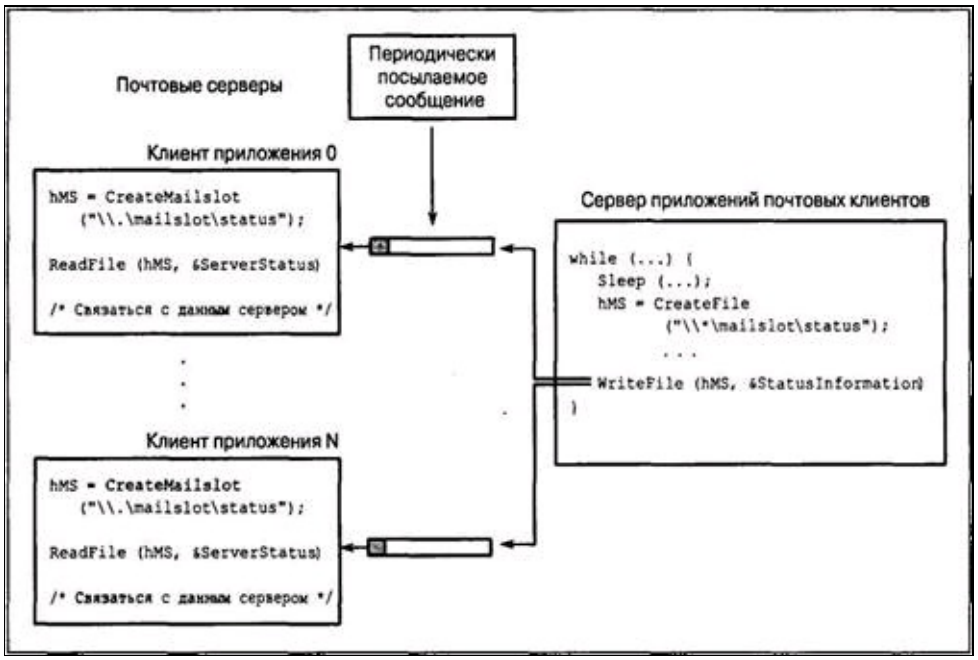


Рис. 11.3. Использование клиентами почтового ящика для обнаружения сервера

```
HANDLE CreateMailslot(LPCTSTR lpName, DWORD cbMaxMsg, DWORD
dwReadTimeout, LPSECURITY_ATTRIBUTES lpsa)
```

Параметры

lpName — указатель на строку с именем почтового ящика, которое должно иметь следующий вид:

`\\.\mailslot\[путь]`*имя*

Имя должно быть уникальным. Точка (.) указывает на то, что почтовый ящик создается на локальном компьютере.

cbMaxMsg — максимальный размер сообщения (в байтах), которые может записывать клиент. Значению 0 соответствует отсутствие ограничений.

dwReadTimeOut — длительность интервала ожидания (в миллисекундах) для операции чтения. Значению 0 соответствует немедленный возврат, а значению MAILSLOT_WAIT_FOREVER — неопределенный период ожидания (который может длиться сколь угодно долго).

Во время открытия почтового ящика с помощью функции CreateFile клиент (записывающая программа) может указывать его имя в следующем виде:

• \\ .\mailslot\ [путь]имя — определяет локальный почтовый ящик. *Примечание.* В Windows 95 длина имени ограничена 11 символами.

• \\имя_компьютера\mailslot\[путь]имя — определяет почтовый ящик, расположенный на компьютере с заданным именем.

• \\имя_домена\mailslot\[путь]имя — определяет все почтовые ящики с данным именем, расположенные на компьютерах, принадлежащих данному домену. В этом случае максимальный размер сообщения составляет 424 байта.

• *\mailslot\[путь]имя — определяет все почтовые ящики с данным именем, расположенные на компьютерах, принадлежащих главному домену системы. В этом случае максимальный размер сообщения составляет 424 байта.

Наконец, клиент должен указывать флаг FILE_SHARE_READ. Функции GetMailslotInfo и SetMailslotInfo похожи на свои аналоги, работающие с именованными каналами.

> Средства, сопоставимые с почтовыми ящиками, в UNIX отсутствуют. Однако для этой цели могут быть использованы широковещательные (broadcast) или групповые (multicast) дейтаграммы протокола TCP/IP.

Создание, подключение и именование каналов и почтовых ящиков

В табл. 11.1 сведены все допустимые формы имен каналов, которые могут использоваться клиентами и серверами приложения. Здесь же перечислены все функции, которые следует использовать для создания именованных каналов и соединения с ними.

Аналогичная информация для почтовых ящиков приведена в табл. 11.2. Вспомните, что почтовый клиент (или сервер) не обязательно должен выполняться тем же процессом или даже на той же системе, что и клиент (или сервер) приложения.

Таблица 11.1. Именованные каналы: создание, подключение и именование

	Клиент приложения	**Сервер приложения**
Дескриптор именованного канала или соединение	CreateFile CallNamedPipe TransactNamedPipe	CreateNamedPipe
Имя канала	\\.\имя канала (канал является локальным) \\имя системы\имя канала (канал является локальным или удаленным)	\\.\имя канала (канал создается локальным)

Таблица 11.2. Почтовые ящики: создание, подключение и именование

	Почтовый клиент	**Почтовый сервер**
Дескриптор почтового ящика	CreateFile	CreateMailslot
Имя почтового ящика	\\.\имя почтового ящика (почтовый ящик является локальным) \\имя системы\имя почтового ящика (почтовый ящик располагается на указанной удаленной системе) *\имя почтового ящика (все почтовые ящики, имеющие одно и то же указанное имя)	\\.\имя почтового ящика (почтовый ящик создается локальным)

Пример: сервер, обнаруживаемый клиентами

Программа 11.4 представляет функцию потока, которую сервер командной строки (программа 11.3), *выступающий в роли почтового клиента,* использует для широковещательной рассылки имени своего канала ожидающим клиентам. Возможно существование нескольких серверов с различными характеристиками и именами каналов, и клиенты получают их имена, используя почтовый ящик с известным именем. Эта функция запускается как поток программой 11.3.

Примечание

На практике многие клиент-серверные системы инвертируют используемую здесь логику поиска. Суть альтернативного варианта заключается в том, что клиент приложения действует и как почтовый клиент, осуществляя широковещательную рассылку сообщений, требующих, чтобы сервер ответил с использованием указанного именованного канала (имя канала определяется клиентом и включается в сообщение). Затем сервер приложения, действующий в качестве почтового сервера, считывает запрос и создает соединение с использованием указанного именованного канала.

Программа 11.4. SrvrBcst: функция потока почтового клиента

```
    static DWORD WINAPI ServerBroadcast(LPLONG pNull) {
     MS_MESSAGE MsNotify;
     DWORD nXfer;
     HANDLE hMsFile;
     /*Открыть почтовый ящик для записывающей программы почтового "клиента"*/
      while (!ShutDown) { /* Цикл выполняется до тех пор, пока имеются серверные
потоки. */
      /* Ждать, пока другой клиент не откроет почтовый ящик. */
      Sleep(CS_TIMEOUT);
       hMsFile = CreateFile(MS_CLTNAME, GENERIC_WRITE, FILE_SHARE_READ, NULL, OPEN
EXISTING, FILE ATTRIBUTE NORMAL, NULL);
      if (hMsFile == INVALID_HANDLE_VALUE) continue;
      /* Отправить сообщение в почтовый ящик. */
      MsNotify.msStatus = 0;
      MsNotify.msUtilization = 0;
      _tcscpy(MsNotify.msName, SERVER_PIPE);
            if    (WriteFile(hMsFile,    &MsNotify,    MSM_SIZE,    &nXfer,    NULL))
ReportError(_T("Ошибка записи почтового сервера."), 13, TRUE);
      CloseHandle(hMsFile);
     }
     _tprintf(_T("Закрытие контролирующего потока.\n"));
     _endthreadex(0);
     return 0;
    }
```

В программе 11.5 представлена функция, которая вызывается клиентом (см. программу 11.2) для обнаружения сервера.

Программа 11.5. LocSrvr: почтовый сервер

```
    /* Глава 11. LocSrver.c */
```

```
    /* Найти сервер путем считывания информации из почтового ящика, используемого
для широковещательной рассылки имен серверов. */

    #include "EvryThng.h"
    #include "ClntSrvr.h" /* Определяет имя почтового ящика. */

    BOOL LocateServer(LPTSTR pPipeName) {
     HANDLE MsFile;
     MS_MESSAGE ServerMsg;
     BOOL Found = FALSE;
     DWORD cbRead;
     MsFile = CreateMailslot(MS_SRVNAME, 0, CS_TIMEOUT, NULL);
     while (!Found) {
      _tprintf(_T("Поиск сервера.\n"));
      Found = ReadFile(MsFile, &ServerMsg, MSM_SIZE, &cbRead, NULL);
     }
     _tprintf(_T("Сервер найден.\n"));
     CloseHandle(MsFile);
     /* Имя канала сервера. */
     _tcscpy(pPipeName, ServerMsg.msName);
     return TRUE;
    }
```

Комментарии по поводу многопоточных моделей

Для описания методов проектирования многопоточных программ используются такие термины, как *пул потоков* (thread pool), *симметричные потоки* (symmetric threads) и *асимметричная потоковая организация программ* (asymmetric threading), а мы при создании примеров использовали модель "хозяин/рабочий", именованные каналы и другие классические модели организации многопоточного выполнения программ.

В этом разделе дано краткое объяснение некоторых полезных описательных терминов, которые являются неотъемлемой частью объектно-ориентированной технологии, основанной на разработанной компанией Microsoft модели компонентных объектов (Component Object Model, COM; см. [3]): однопоточная модель (single threading), модель апартаментных потоков (apartment model) и модель свободных потоков (free threading). В COM эти модели реализуются за счет использования функций Windows, предназначенных для управления потоками и синхронизации их выполнения. Каждая из перечисленных моделей обладает отличными от других характеристиками производительности и предъявляет свои требования к синхронизации.

• Пул потоков — это совокупность потоков, доступных для использования по мере необходимости. С помощью рис. 7.1 и программы 11.3 иллюстрируется пул потоков, которые могут назначаться новым клиентам, подключающимся к соответствующему именованному каналу. При отсоединении клиента поток возвращается в пул.

• Потоковая модель является симметричной, если группа потоков выполняют одну и ту же задачу с использованием одной и той же функции потока. Симметричные потоки используются в программе grepMT (программа 7.1): все потоки выполняют один и тот же код поиска шаблона. Обратите внимание, что эти потоки не образуют пула; каждый из них создается для выполнения определенной задачи и завершается сразу же после того, как задача выполнена. Пул симметричных потоков создается в программе 11.3.

• Потоковая модель является асимметричной, если различные потоки выполняют различные задачи с использованием различных функций потока. Так, функция потока широковещательной рассылки сообщений, представленная на рис. 7.1 и реализованная в программе 11.4, и функция сервера соответствуют модели асимметричных потоков.

• В соответствии с терминологией COM объект является однопоточным, если доступ к нему может получать только один поток. Это означает, что доступ к такому объекту сериализуется. В случае сервера базы данных таким объектом будет сама база данных. В примерах, приведенных в настоящей главе, многопоточная модель используется для организации доступа к объекту, в качестве которого могут рассматриваться программы и файлы, расположенные на компьютере сервера.

• В соответствии с терминологией COM об апартаментной модели следует говорить тогда, когда для каждого экземпляра объекта назначается отдельный поток. Так, отдельные потоки могут назначаться для осуществления доступа к разным базам данных или частям базы данных. Доступ к объекту сериализуется с помощью единственного потока.

• Объект, соответствующий модели свободных потоков, имеет поток, который обычно назначается ему из пула потоков по запросу. Обсуждавшийся в настоящей главе сервер можно считать сервером со свободными потоками, если соединение рассматривать как запрос. Аналогично, если сервер базы данных поддерживается потоками, то можно говорить о том, что база данных соответствует модели свободных потоков.

Некоторые программы, например sortMT (программа 7.2), в рамки ни одной из перечисленных моделей точно не укладываются. Вспомните также, что нами уже

использовались и другие модели, а именно, модель "хозяин/рабочий", именованные каналы и клиент-серверные модели, применение которых является общепринятым, однако не находит отражения в моделях Microsoft.

Применение указанных моделей многопоточного программирования оказывается уместным и в главе 12, в которой вводятся внутрипроцессные серверы, и компания Microsoft использует соответствующие термины в некоторой части своей документации. Не забывайте о том, что эти термины определены применительно к COM-объектам; предыдущее обсуждение показало, как использовать их в более широком контексте. COM — это слишком большой и сложный предмет, чтобы мы могли полностью описать его в данной книге. В списке литературы, приведены некоторые ссылки, которыми вы можете воспользоваться для получения более подробных сведений по этому вопросу.

Резюме

Каналы и почтовые ящики Windows, доступ к которым осуществляется с помощью операций файлового ввода/вывода, обеспечивают поточное межпроцессное и сетевое взаимодействие. В примерах продемонстрировано, как организовать передачу данных из одного процесса в другой при помощи каналов и как построить простую многопоточную клиент-серверную систему. Кроме того, каналы обеспечивают дополнительную возможность синхронизации потоков, поскольку считывающий поток блокируется до тех пор, пока другой поток не выполнит запись в канал.

В следующих главах

В главе 12 для осуществления межпроцессного и сетевого взаимодействия вместо оригинальных механизмов Windows привлечены стандартные механизмы. Прежний вариант клиент-серверной системы переработан с использованием стандартных методов и дополнительно усовершенствован за счет некоторого улучшения серверной части.

Упражнения

11.1. Проведите эксперименты с целью проверки справедливости приведенных ранее утверждений относительно повышения производительности, обеспечиваемого использованием функции TransactNamedPipe. Для этого вам придется внести в существующий код сервера некоторые изменения. Кроме того, сопоставьте полученные результаты с теми, к которым приводит текущая реализация.

11.2. Воспользуйтесь программой JobShell из главы 6 для запуска сервера и нескольких клиентов, причем каждый из клиентов должен создаваться в режиме "отсоединения". Для завершения работы остановите сервер путем посылки управляющего сигнала консоли с помощью команды kill. Имеются ли у вас какие-либо соображения по поводу улучшения логики закрытия программы serverNP, чтобы подключенный серверный поток мог проверять флаг завершения работы, будучи блокированным в ожидании запроса клиента? *Подсказка.* Создайте считывающий поток, аналогичный потоку соединения.

11.3. Усовершенствуйте сервер таким образом, чтобы имя его канала задавалось в виде аргумента в командной строке. Организуйте несколько серверных процессов с различными именами каналов, используя программу управления заданиями из главы 6. Убедитесь в том, что одновременно несколько клиентов получают доступ к этой многопроцессной серверной системе.

11.4. Запустите клиент и сервер на различных системах, чтобы проверить их работу в условиях сети. Измените программу SrvrBcst (программа 11.4) таким образом, чтобы она включала имя компьютера сервера в имя канала. Кроме того, видоизмените имя почтового ящика, используемого в программе 11.4.

11.5. Модифицируйте сервер таким образом, чтобы можно было измерить степень его загрузки (иными словами, чтобы можно было определить, какая доля использованного времени приходится на сервер). Организуйте накопление данных, касающихся производительности, и передачу этой информации клиенту в ответ на его запрос. Для этой цели можно использовать поле Request.Command.

11.6. Усовершенствуйте программы, предназначенные для обнаружения сервера, таким образом, чтобы поиск сервера клиентом осуществлялся с наименьшей степенью загрузки.

11.7. Усовершенствуйте сервер таким образом, чтобы запрос включал в себя рабочий каталог. Сервер должен устанавливать свой рабочий каталог, выполнять команду, а затем восстанавливать рабочий каталог до прежнего значения. *Предостережение.* Серверный поток не должен устанавливать рабочий каталог процесса; вместо этого каждый поток должен поддерживать строку, представляющую его рабочий каталог, и присоединять эту строку в начало соответствующих путей доступа.

11.8. Программа serverNP спроектирована таким образом, что она выполняется как сервер в течение неопределенного времени, давая клиентам возможность подключаться, получать услуги и разрывать соединение. Очень важно, чтобы при отключении клиента сервер освобождал все соответствующие ресурсы, например, память, дескрипторы файлов или потоков. Если это не делается, то в результате утечки ресурсов системные ресурсы, в конце концов, исчерпаются, что приведет аварийному завершению работы сервера с возможным предшествующим ухудшением показателей производительности. Тщательно проверьте код программы serverNP, чтобы убедиться в отсутствии утечки ресурсов, и в случае необходимости внесите необходимые исправления. (Просьба информировать автора обо всех обнаруженных ошибках, используя указанный в предисловии адрес электронной почты.) *Примечание.* Утечка ресурсов является

весьма распространенным серьезным дефектом многих промышленных систем. Никакие попытки вывода изделия на уровень "промышленных стандартов" нельзя считать успешными, если указанной проблеме не было уделено должного внимания.

11.9. *Расширенное упражнение.* Объекты синхронизации могут использоваться для синхронизации потоков, выполняющихся в различных процессах на одной и той же машине, но с их помощью нельзя синхронизировать потоки процессов, которые выполняются на разных машинах. Используя именованные каналы и почтовые ящики, создайте эмулированные мьютексы, события и семафоры, чтобы преодолеть это ограничение.

ГЛАВА 12
Сетевое программирование с помощью сокетов Windows

Именованные каналы пригодны для организации межпроцессного взаимодействия как в случае процессов, выполняющихся на одной и той же системе, так и в случае процессов, выполняющихся на компьютерах, связанных друг с другом локальной или глобальной сетью. Эти возможности были продемонстрированы на примере клиент-серверной системы, разработанной в главе 11, начиная с программы 11.2.

Однако как именованные каналы, так и почтовые ящики (в отношении которых для простоты мы будем использовать далее общий термин — "именованные каналы", если различия между ними не будут играть существенной роли) обладают тем недостатком, что они не являются промышленным стандартом. Это обстоятельство усложняет перенос программ наподобие тех, которые рассматривались в главе 11, в системы, не принадлежащие семейству Windows, хотя именованные каналы не зависят от протоколов и могут выполняться поверх многих стандартных промышленных протоколов, например TCP/IP.

Возможность взаимодействия с другими системами обеспечивается в Windows поддержкой сокетов (sockets) Windows Sockets — совместимого и почти точного аналога сокетов Berkeley Sockets, де-факто играющих роль промышленного стандарта. В этой главе использование API Windows Sockets (или "Winsock") показано на примере модифицированной клиент-серверной системы из главы 11. Результирующая система способна функционировать в глобальных сетях, использующих протокол TCP/IP, что, например, позволяет серверу принимать запросы от клиентов UNIX или каких-либо других, отличных от Windows систем.

Читатели, знакомые с интерфейсом Berkeley Sockets, при желании могут сразу же перейти непосредственно к рассмотрению примеров, в которых не только используются сокеты, но также вводятся новые возможности сервера и демонстрируются дополнительные методы работы с библиотеками, обеспечивающими безопасную многопоточную поддержку.

Привлекая средства обеспечения взаимодействия между разнородными системами, ориентированные на стандарты, интерфейс Winsock открывает перед программистами возможность доступа к высокоуровневым протоколам и приложениям, таким как ftp, http, RPC и COM, которые в совокупности предоставляют богатый набор высокоуровневых моделей, обеспечивающих поддержку межпроцессного сетевого взаимодействия для систем с различной архитектурой.

В данной главе указанная клиент-серверная система используется в качестве механизма демонстрации интерфейса Winsock, и в процессе того, как сервер будет модифицироваться, в него будут добавляться новые интересные возможности. В частности, нами будут впервые использованы *точки входа DLL* (глава 5) и *внутрипроцессные серверы DLL*. (Эти новые средства можно было включить уже в первоначальную версию программы в главе 11, однако это отвлекло бы ваше внимание от разработки основной архитектуры системы.) Наконец, дополнительные примеры покажут вам, как создаются безопасные реентерабельные многопоточные библиотеки.

Поскольку интерфейс Winsock должен соответствовать промышленным стандартам, принятые в нем соглашения о правилах присвоения имен и стилях программирования несколько отличаются от тех, с которыми мы сталкивались в процессе работы с описанными ранее функциями Windows. Строго говоря, Winsock API не является частью Win32/64. Кроме того, Winsock предоставляет дополнительные функции, не подчиняющиеся стандартам; эти функции

используются лишь в случае крайней необходимости. Среди других преимуществ, обеспечиваемых Winsock, следует отметить улучшенную переносимость результирующих программ на другие системы.

Сокеты Windows

Winsock API разрабатывался как расширение Berkley Sockets API для среды Windows и поэтому поддерживается всеми системами Windows. К преимуществам Winsock можно отнести следующее:

• Перенос уже имеющегося кода, написанного для Berkeley Sockets API, осуществляется непосредственно.

• Системы Windows легко встраиваются в сети, использующие как версию IPv4 протокола TCP/IP, так и постепенно распространяющуюся версию IPv6. Помимо всего остального, версия IPv6 допускает использование более длинных IP-адресов, преодолевая существующий 4-байтовый адресный барьер версии IPv4.

• Сокеты могут использоваться совместно с перекрывающимся вводом/выводом Windows (глава 14), что, помимо всего прочего, обеспечивает возможность масштабирования серверов при увеличении количества активных клиентов.

• Сокеты можно рассматривать как дескрипторы (типа HANDLE) файлов при использовании функций ReadFile и WriteFile и, с некоторыми ограничениями, при использовании других функций, точно так же, как в качестве дескрипторов файлов сокеты применяются в UNIX. Эта возможность оказывается удобной в тех случаях, когда требуется использование асинхронного ввода/вывода и портов завершения ввода/вывода.

• Существуют также дополнительные, непереносимые расширения.

Инициализация Winsock

Winsock API поддерживается библиотекой DLL (WS2_32.DLL), для получения доступа к которой следует подключить к программе библиотеку WS_232.LIB. Эту DLL следует инициализировать с помощью нестандартной, специфической для Winsock функции WSAStartup, которая должна быть первой из функций Winsock, вызываемых программой. Когда необходимость в использовании функциональных возможностей Winsock отпадает, следует вызывать функцию WSACleanup. *Примечание.* Префикс WSA означает "Windows Sockets asynchronous …" ("Асинхронный Windows Sockets …"). Средства асинхронного режима Winsock нами здесь не используются, поскольку при возникновении необходимости в выполнении асинхронных операций мы можем и будем использовать потоки.

Хотя функции WSAStartup и WSACleanup необходимо вызывать в обязательном порядке, вполне возможно, что они будут единственными нестандартными функциями, с которыми вам придется иметь дело. Распространенной практикой является применение директив препроцессора #ifdef для проверки значения символической константы _WIN32 (обычно определяется Visual C++ на стадии компиляции), в результате чего функции WSA будут вызываться только тогда, когда вы работаете в Windows). Разумеется, такой подход предполагает, что остальная часть кода не зависит от платформы.

```
int WSAStartup(WORD wVersionRequired, LPWSADATA ipWSAData);
```

Параметры

wVersionRequired — указывает старший номер версии библиотеки DLL, который вам

требуется и который вы можете использовать. Как правило, версии 1.1 вполне достаточно для того, чтобы обеспечить любое взаимодействие с другими системами, в котором у вас может возникнуть необходимость. Тем не менее, во всех системах Windows, включая Windows 9x, доступна версия Winsock 2.0, которая и используется в приведенных ниже примерах. Версия 1.1 считается устаревшей и постепенно выходит из употребления.

Функция возвращает ненулевое значение, если запрошенная вами версия данной DLL не поддерживается.

Младший байт параметра wVersionRequired указывает основной номер версии, а старший байт — дополнительный. Обычно используют макрос MAKEWORD; таким образом, выражение MAKEWORD (2,0) представляет версию 2.0.

ipWSAData — указатель на структуру WSADATA, которая возвращает информацию о конфигурации DLL, включая старший доступный номер версии. О том, как интерпретировать ее содержимое, вы можете прочитать в материалах оперативной справки Visual Studio.

Чтобы получить более подробную информацию об ошибках, можно воспользоваться функцией WSAGetLastError, но для этой цели подходит также функция GetLastError, а также функция ReportError, разработанная в главе 2.

По окончании работы программы, а также в тех случаях, когда необходимости в использовании сокетов больше нет, следует вызывать функцию WSACleanup, чтобы библиотека WS_32.DLL, обслуживающая сокеты, могла освободить ресурсы, распределенные для этого процесса.

Создание сокета

Инициализировав Winsock DLL, вы можете использовать стандартные (Berkeley Sockets) функции для создания сокетов и соединений, обеспечивающих взаимодействие серверов с клиентами или взаимодействие равноправных узлов сети между собой.

Используемый в Winsock тип данных SOCKET аналогичен типу данных HANDLE в Windows, и его даже можно применять совместно с функцией ReadFile и другими функциями Windows, требующими использования дескрипторов типа HANDLE. Для создания (или открытия) сокета служит функция socket.

```
SOCKET socket(int af, int type, int protocol);
```

Параметры

Тип данных SOCKET фактически определяется как тип данных int, потому код UNIX остается переносимым, не требуя привлечения типов данных Windows.

af — обозначает семейство адресов, или протокол; для указания протокола IP (компонент протокола TCP/IP, отвечающий за протокол Internet) следует использовать значение PF_INET (или AF_INET, которое имеет то же самое числовое значение, но обычно используется при вызове функции bind).

type — указывает тип взаимодействия: ориентированное на установку соединения (connection-oriented communication), или потоковое (SOCK_STREAM), и дейтаграммное (datagram communication) (SOCK_DGRAM), что в определенной степени сопоставимо соответственно с именованными каналами и почтовыми ящиками.

protocol — является излишним, если параметр af установлен равным AF_INET; используйте

значение 0.

В случае неудачного завершения функция socket возвращает значение INVALID_SOCKET.

Winsock можно использовать совместно с протоколами, отличными от TCP/IP, указывая различные значения параметра protocol; мы же будем использовать только протокол TCP/IP.

Как и в случае всех остальных стандартных функций, имя функции socket не должно содержать прописных букв. Это является отходом от соглашений, принятых в Windows, и продиктовано необходимостью соблюдения промышленных стандартов.

Серверные функции сокета

В нижеследующем обсуждении под *сервером* будет пониматься процесс, который принимает запросы на образование соединения через заданный порт. Несмотря на то что сокеты, подобно именованным каналам, могут использоваться для создания соединений между равноправными узлами сети, введение указанного различия между узлами является весьма удобным и отражает различия в способах, используемых обеими системами для соединения друг с другом.

Если не оговорено иное, типом сокетов в наших примерах всегда будет SOCK_STREAM. Сокеты типа SOCK_DGRAM рассматривается далее в этой главе.

Связывание сокета

Следующий шаг заключается в привязке сокета к его адресу и *конечной точке* (endpoint) (направление канала связи от приложения к службе). Вызов socket, за которым следует вызов bind, аналогичен созданию именованного канала. Однако не существует имен, используя которые можно было бы различать сокеты данного компьютера. Вместо этого в качестве конечной точки службы используется *номер порта* (port number). Любой заданный сервер может иметь несколько конечных точек. Прототип функции bind приводится ниже.

```
int bind(SOCKET s, const struct sockaddr *saddr, int namelen);
```

Параметры

s — несвязанный сокет, возвращенный функцией socket.

saddr — заполняется перед вызовом и задает протокол и специфическую для протокола информацию, как описано ниже. Кроме всего прочего, в этой структуре содержится номер порта.

namelen — присвойте значение sizeof (sockaddr).

В случае успешного выполнения функция возвращает значение 0, иначе SOCKET_ERROR. Структура sockaddr определяется следующим образом:

```
struct sockaddr {
 u_short sa_family;
 char sa_data[14] ;
};
typedef struct sockaddr SOCKADDR, *PSOCKADDR;
```

Первый член этой структуры, sa_family, обозначает протокол. Второй член, sa_data, зависит от протокола. Internet-версией структуры sa_data является структура sockaddr_in:

```
struct sockaddr_in {
 short sin_family; /* AF_INET */
 u_short sin_port;
 struct in_addr sin_addr; /* 4-байтовый IP-адрес */
 char sin_zero[8];
};
typedef struct sockaddr_in SOCKADDR_IN, *PSOCKADDR IN;
```

Обратите внимание на использование типа данных short integer для номера порта. Кроме того, номер порта и иная информация должны храниться с соблюдением подходящего порядка следования байтов, при котором старший байт помещается в крайней позиции справа (big-endian), чтобы обеспечивалась двоичная совместимость с другими системами. В структуре sin_addr содержится подструктура s_addr, заполняемая уже знакомым нам 4-байтовым IP-адресом, например 127.0.0.1, указывающим систему, чей запрос на образование соединения должен быть принят. Обычно удовлетворяются запросы любых систем, в связи с чем следует использовать значение INADDR_ANY, хотя этот символический параметр должен быть преобразован к корректному формату, как показано в приведенном ниже фрагменте кода.

Для преобразования текстовой строки с IP-адресом к требуемому формату можно использовать функцию inet_addr, поэтому член sin_addr.s_addr переменной sockaddr_in инициализируется следующим образом:

```
sa.sin_addr.s_addr = inet_addr("192 .13.12.1");
```

О связанном сокете, для которого определены протокол, номер порта и IP-адрес, иногда говорят как об *именованном сокете* (named socket).

Перевод связанного сокета в состояние прослушивания

Функция listen делает сервер доступным для образования соединения с клиентом. Аналогичной функции для именованных каналов не существует.

```
int listen(SOCKET s, int nQueueSize);
```

Параметр nQueueSize указывает число запросов на соединение, которые вы намерены помещать в очередь сокета. В версии Winsock 2.0 значение этого параметра не имеет ограничения сверху, но в версии 1.1 оно ограничено предельным значением SOMAXCON (равным 5).

Прием клиентских запросов соединения

Наконец, сервер может ожидать соединения с клиентом, используя функцию accept, возвращающую новый подключенный сокет, который будет использоваться в операциях ввода/вывода. Заметьте, что исходный сокет, который теперь находится в состоянии прослушивания (listening state), используется исключительно в качестве параметра функции accept, а не для непосредственного участия в операциях ввода/вывода.

Функция accept блокируется до тех пор, пока от клиента не поступит запрос соединения, после чего она возвращает новый сокет ввода/вывода. Хотя рассмотрение этого и выходит за рамки данной книги, возможно создание неблокирующихся сокетов, а в сервере (программа 12.2) для приема запроса используется отдельный поток, что позволяет создавать также неблокирующиеся серверы.

```
SOCKET accept(SOCKET s, LPSOCKADDR lpAddr, LPINT lpAddrLen);
```

Параметры

s — прослушивающий сокет. Чтобы перевести сокет в состояние прослушивания,

необходимо предварительно вызвать функции socket, bind и listen.

lpAddr — указатель на структуру sockaddr_in, предоставляющую адрес клиентской системы.

lpAddrLen — указатель на переменную, которая будет содержать размер возвращенной структуры sockaddr_in. Перед вызовом функции accept эта переменная должна быть инициализирована значением sizeof(struct sockaddr_in).

Отключение и закрытие сокетов

Для отключения сокетов применяется функция shutdown(s, how). Аргумент how может принимать одно из двух значений: 1, указывающее на то, что соединение может быть разорвано только для посылки сообщений, и 2, соответствующее разрыву соединения как для посылки, так и для приема сообщений. Функция shutdown не освобождает ресурсы, связанные с сокетом, но гарантирует завершение посылки и приема всех данных до закрытия сокета. Тем не менее, после вызова функции shutdown приложение уже не должно использовать этот сокет.

Когда работа с сокетом закончена, его следует закрыть, вызвав функцию closesocket(SOCKET s). Сначала сервер закрывает сокет, созданный функцией accept, а не прослушивающий сокет, созданный с помощью функции socket. Сервер должен закрывать прослушивающий сокет только тогда, когда завершает работу или прекращает принимать клиентские запросы соединения. Даже если вы работаете с сокетом как с дескриптором типа HANDLE и используете функции ReadFile и WriteFile, уничтожить сокет одним только вызовом функции CloseHandle вам не удастся; для этого следует использовать функцию closesocket.

Пример: подготовка и получение клиентских запросов соединения

Ниже приводится фрагмент кода, показывающий, как создать сокет и организовать прием клиентских запросов соединения.

В этом примере используются две стандартные функции: htons ("host to network short" — "ближняя связь") и htonl ("host to network long" — "дальняя связь"), которые преобразуют целые числа к форме с обратным порядком байтов, требуемой протоколом IP.

Номером порта сервера может быть любое число из диапазона, допустимого для целых чисел типа short integer, но для определенных пользователем служб обычно используются числа в диапазоне 1025—5000. Порты с меньшими номерами зарезервированы для таких известных служб, как telnet или ftp, в то время как порты с большими номерами предполагаются для использования других стандартных служб.

```
struct sockaddr_in SrvSAddr; /* Адресная структура сервера. */
struct sockaddr_in ConnectAddr;
SOCKET SrvSock, sockio;
…
SrvSock = socket(AF_INET, SOCK_STREAM, 0);
SrvSAddr.sin_family = AF_INET;
SrvSAddr.sin_addr.s_addr = htonl(INADDR_ANY);
SrvSAddr.sin_port = htons(SERVER_PORT);
bind(SrvSock, (struct sockaddr *)&SrvSAddr, sizeof SrvSAddr);
listen(SrvSock, 5);
AddrLen = sizeof(ConnectAddr);
sockio = accept(SrvSock, (struct sockaddr *) &ConnectAddr, &AddrLen);
… Получение запросов и отправка ответов …
shutdown(sockio);
closesocket(sockio);
```

Клиентские функции сокета

Клиентская станция, которая желает установить соединение с сервером, также должна создать сокет, вызвав функцию socket. Следующий шаг заключается в установке соединения сервером, а, кроме того, необходимо указать номер порта, адрес хоста и другую информацию. Имеется только одна дополнительная функция – connect.

Установление клиентского соединения с сервером

Если имеется сервер с сокетом в режиме прослушивания, клиент может соединиться с ним при помощи функции connect.

```
int connect(SOCKET s, LPSOCKADDR lpName, int nNameLen);
```

Параметры

s — сокет, созданный с использованием функции socket.

lpName — указатель на структуру sockaddr_in, инициализированную значениями номера порта и IP-адреса системы с сокетом, связанным с указанным портом, который находится в состоянии прослушивания.

Инициализируйте nNameLen значением sizeof (struct sockaddr_in).

Возвращаемое значение 0 указывает на успешное завершение функции, тогда как значение SOCKET_ERROR указывает на ошибку, которая, в частности, может быть обусловлена отсутствием прослушивающего сокета по указанному адресу.

Сокет s не обязательно должен быть связанным с портом до вызова функции connect, хотя это и может иметь место. При необходимости система распределяет порт и определяет протокол.

Пример: подключение клиента к серверу

Показанный ниже фрагмент кода обеспечивает соединение клиента с сервером. Для этого нужны только два вызова функций, но адресная структура должна быть инициализирована до вызова функции connect. Проверка возможных ошибок здесь отсутствует, но в реальные программы она должна включаться. В примере предполагается, что IP-адрес (текстовая строка наподобие "192.76.33.4") задается в аргументе argv[1] командной строки.

```
    SOCKET ClientSock;
    …
    ClientSock = socket(AF_INET, SOCK_STREAM, 0);
    memset(&ClientSAddr, 0, sizeof(ClientSAddr));
    ClientSAddr.sin_family = AF_INET;
    ClientSAddr.sin_addr.s_addr = inet_addr(argv[1]);
    ClientSAddr.sin_port = htons(SERVER_PORT);
    ConVal     =     connect(ClientSock,     (struct     sockaddr     *)&ClientSAddr,
sizeof(ClientSAddr));
```

Отправка и получение данных

Программы, использующие сокеты, обмениваются данными с помощью функций send и recv, прототипы которых почти совпадают (перед указателем буфера функции send помещается модификатор const). Ниже представлен только прототип функции send.

```
int send(SOCKET s, const char * lpBuffer, int nBufferLen, int nFlags);
```

Возвращаемым значением является число фактически переданных байтов. Значение SOCKET_ERROR указывает на ошибку.

nFlags — может использоваться для обозначения степени срочности сообщений (например, экстренных сообщений), а значение MSG_PEEK позволяет просматривать получаемые данные без их считывания.

Самое главное, что вы должны запомнить — это то, что функции send и recv *не являются атомарными* (atomic), и поэтому нет никакой гарантии, что затребованные данные будут действительно отправлены или получены. Передача "коротких" сообщений ("short sends") встречается крайне редко, хотя и возможна, что справедливо и по отношению к приему "коротких" сообщений ("short receives"). Понятие сообщения в том смысле, который оно имело в случае именованных каналов, здесь отсутствует, и поэтому вы должны проверять возвращаемое значение и повторно отправлять или принимать данные до тех пор, пока все они не будут переданы.

С сокетами могут использоваться также функции ReadFile и WriteFile, только в этом случае при вызове функции необходимо привести сокет к типу HANDLE.

Сравнение именованных каналов и сокетов

Именованные каналы, описанные в главе 11, очень похожи на сокеты, но в способах их использования имеются значительные различия.

• Именованные каналы могут быть ориентированными на работу с сообщениями, что значительно упрощает программы.

• Именованные каналы требуют использования функций ReadFile и WriteFile, в то время как сокеты могут обращаться также к функциям send и recv.

• В отличие от именованных каналов сокеты настолько гибки, что предоставляют пользователям возможность выбрать протокол для использования с сокетом, например, TCP или UDP. Кроме того, пользователь имеет возможность выбирать протокол на основании характера предоставляемой услуги или иных факторов.

• Сокеты основаны на промышленном стандарте, что обеспечивает их совместимость с системами, отличными от Windows.

Имеются также различия в моделях программирования сервера и клиента.

Сравнение серверов именованных каналов и сокетов

Установка соединения с несколькими клиентами при использовании сокетов требует выполнения повторных вызовов функции accept. Каждый из вызовов возвращает очередной подключенный сокет. По сравнению с именованными каналами имеются следующие отличия:

• В случае именованных каналов требуется, чтобы каждый экземпляр именованного канала и дескриптор типа HANDLE создавались с помощью функции CreateNamedPipe, тогда как для создания экземпляров сокетов применяется функция accept.

• Допустимое количество клиентских сокетов ничем не ограничено (функция listen ограничивает лишь количество клиентов, помещаемых в очередь), в то время как количество экземпляров именованных каналов, в зависимости от того, что было указано при первом вызове функции CreateNamedPipe, может быть ограниченным.

• Не существует вспомогательных функций для работы с сокетами, аналогичных функции TransactNamedPipe.

• Именованные каналы не имеют портов с явно заданными номерами и различаются по именам.

В случае сервера именованных каналов получение пригодного для работы дескриптора типа HANDLE требует вызова двух функций (CreateNamedPipe и ConnectNamedPipe), тогда как сервер сокета требует вызова четырех функций (socket, bind, listen и accept).

Сравнение клиентов именованных каналов и сокетов

В случае именованных каналов необходимо последовательно вызывать функции WaitNamedPipe и CreateFile. Если же используются сокеты, этот порядок вызовов обращается, поскольку можно считать, что функция socket создает сокет, а функция connect — блокирует.

Дополнительное отличие состоит в том, что функция connect является функцией клиента сокета, в то время как функция ConnectNamedPipe используется сервером именованного канала.

Пример: функция приема сообщений в случае сокета

Часто оказывается удобным отправлять и получать сообщения в виде единых блоков. Как было показано в главе 11, каналы позволяют это сделать. Однако в случае сокетов требуется создание заголовка, содержащего размер сообщения, за которым следует само сообщение. Для приема таких сообщений предназначена функция ReceiveMessage, которая будет использоваться в примерах. То же самое можно сказать и о функции SendMessage, предназначенной для передачи сообщений.

Обратите внимание, что сообщение принимается в виде двух частей: заголовка и содержимого. Ниже мы предполагаем, что пользовательскому типу MESSAGE соответствует 4-байтовый заголовок. Но даже для 4-байтового заголовка требуются повторные вызовы функции recv, чтобы гарантировать его полное считывание, поскольку функция recv не является атомарной.

Примечание, относящееся к Win64

В качестве типа переменных, используемых для хранения размера сообщения, выбран тип данных фиксированной точности LONG32, которого будет вполне достаточно для размещения значений параметра размера, включаемого в сообщения при взаимодействии с системами, отличными от Windows, и который годится для возможной последующей перекомпиляции программы для ее использования на платформе Win64 (см. главу 16).

```
    DWORD ReceiveMessage (MESSAGE *pMsg, SOCKET sd) {
     /* Сообщение состоит из 4-байтового поля размера сообщения, за которым следует
собственно содержимое. */
     DWORD Disconnect = 0;
     LONG32 nRemainRecv, nXfer;
     LPBYTE pBuffer;
     /* Считать сообщение. */
     /* Сначала считывается заголовок, а затем содержимое. */
     nRemainRecv = 4; /* Размер поля заголовка. */
     pBuffer = (LPBYTE)pMsg; /* recv может не передать все запрошенные байты. */
     while (nRemainRecv > 0 && !Disconnect) {
      nXfer = recv(sd, pBuffer, nRemainRecv, 0);
      Disconnect = (nXfer == 0);
      nRemainRecv -=nXfer;
      pBuffer += nXfer;
     }
     /* Считать содержимое сообщения. */
     nRemainRecv = pMsg->RqLen;
     while (nRemainRecv > 0 && !Disconnect) {
      nXfer = recv(sd, pBuffer, nRemainRecv, 0);
      Disconnect = (nXfer == 0);
      nRemainRecv -=nXfer;
      pBuffer += nXfer;
     }
     return Disconnect;
    }
```

Пример: клиент на основе сокета

Программа 12.1 представляет собой переработанный вариант клиентской программы clientNP (программа 11.2), которая использовалась в случае именованных каналов. Преобразование программы осуществляется самым непосредственным образом и требует лишь некоторых пояснений.

• Вместо обнаружения сервера с помощью почтовых ящиков пользователь вводит IP-адрес сервера в командной строке. Если IP-адрес не указан, используется заданный по умолчанию адрес 127.0.0.1, соответствующий локальной системе.

• Для отправки и приема сообщений применяются функции, например, ReceiveMessage, которые здесь не представлены.

• Номер порта, SERVER_PORT, определен в заголовочном файле ClntSrvr.h.

Хотя код написан для выполнения под управлением Windows, единственная зависимость от Windows связана с использованием вызовов функций, имеющих префикс WSA.

Программа 12.1. clientSK: клиент на основе сокетов

```
/* Глава 12. clientSK.c */
/* Однопоточный клиент командной строки. */
/* ВЕРСИЯ НА ОСНОВЕ WINDOWS SOCKETS. */
/* Считывает последовательность команд для пересылки серверному процессу*/
/* через соединение с сокетом. Дожидается ответа и отображает его. */

#define _NOEXCLUSIONS /* Требуется для включения определений сокета. */
#include "EvryThng.h"
#include "ClntSrvr.h" /* Определяет структуры записей запроса и ответа. */

/* Функции сообщения для обслуживания запросов и ответов. */
/* Кроме того, ReceiveResponseMessage отображает полученные сообщения. */
static DWORD SendRequestMessage(REQUEST *, SOCKET);
static DWORD ReceiveResponseMessage(RESPONSE *, SOCKET);
struct sockaddr_in ClientSAddr; /* Адрес сокета клиента. */
int _tmain(DWORD argc, LPTSTR argv[]) {
 SOCKET ClientSock = INVALID_SOCKET;
 REQUEST Request; /* См. ClntSrvr.h. */
 RESPONSE Response; /* См. ClntSrvr.h. */
 WSADATA WSStartData; /* Структура данных библиотеки сокета. */
 BOOL Quit = FALSE;
 DWORD ConVal, j;
 TCHAR PromptMsg[] = _T("\nВведите команду> ");
 TCHAR Req[MAX_RQRS_LEN];
 TCHAR QuitMsg[] = _T("$Quit");
 /* Запрос: завершить работу клиента. */
 TCHAR ShutMsg[] = _T("$ShutDownServer"); /* Остановить все потоки. */
 CHAR DefaultIPAddr[] = "127.0.0.1"; /* Локальная система. */
  /* Инициализировать библиотеку WSA; задана версия 2.0, но будет работать и
версия 1.1. */
 WSAStartup(MAKEWORD(2, 0), &WSStartData);
 /* Подключиться к серверу. */
   /*  Следовать  стандартной  процедуре  вызова  последовательности  функций
socket/connect клиентом. */
 ClientSock = socket(AF_INET, SOCK_STREAM, 0);
 memset(&ClientSAddr, 0, sizeof(ClientSAddr));
```

```
    ClientSAddr.sin_family = AF_INET;
    if (argc >= 2) ClientSAddr.sin_addr.s_addr = inet_addr(argv [1]);
    else ClientSAddr.sin_addr.s_addr = inet_addr(DefaultIPAddr);
    ClientSAddr.sin_port = htons(SERVER_PORT);
    /* Номер порта определен равным 1070. */
    connect(ClientSock, (struct sockaddr *)&ClientSAddr, sizeof(ClientSAddr));
     /* Основной цикл для вывода приглашения на ввод команд, посылки запроса и
получения ответа. */
    while (!Quit) {
     _tprintf(_T("%s"), PromptMsg);
      /* Ввод в формате обобщенных строк, но команда серверу должна указываться в
формате ASCII. */
     _fgetts(Req, MAX_RQRS_LEN-1, stdin);
     for (j = 0; j <= _tcslen(Req) Request.Record[j] = Req[j];
     /* Избавиться от символа новой строки в конце строки. */
     Request.Record[strlen(Request.Record) - 1] = '\0';
      if (strcmp(Request.Record, QuitMsg) == 0 || strcmp(Request.Record, ShutMsg)
== 0) Quit = TRUE;
     SendRequestMessage(&Request, ClientSock);
     ReceiveResponseMessage(&Response, ClientSock);
    }
    shutdown(ClientSock, 2); /* Запретить посылку и прием сообщений. */
    closesocket(ClientSock);
    WSACleanup();
    _tprintf(_T("\n****Выход из клиентской программы\n"));
    return 0;
   }
```

Пример: усовершенствованный сервер на основе сокетов

Программа serverSK (программа 12.2) аналогична программе serverNP (программа 11.3), являясь ее видоизмененным и усовершенствованным вариантом.

• В усовершенствованном варианте программы серверные потоки создаются *по требованию* (on demand), а не в виде пула потоков фиксированного размера. Каждый раз, когда сервер принимает запрос клиента на соединение, создается серверный рабочий поток, и когда клиент прекращает работу, выполнение потока завершается.

• Сервер создает отдельный *поток приема* (accept thread), что позволяет основному потоку опрашивать глобальный флаг завершения работы, пока вызов accept остается блокированным. Хотя сокеты и могут определяться как неблокирующиеся, потоки обеспечивают удобное универсальное решение. Следует отметить, что значительная часть расширенных функциональных возможностей Winsock призвана поддерживать асинхронные операции, тогда как потоки Windows дают возможность воспользоваться более простой и близкой к стандартам функциональностью синхронного режима работы сокетов.

• За счет некоторого усложнения программы усовершенствовано управление потоками, что позволило обеспечить поддержку состояний каждого потока.

• Данный сервер поддерживает также *внутрипроцессные серверы* (in-process servers), что достигается путем загрузки библиотеки DLL во время инициализации. Имя библиотеки DLL задается в командной строке, и серверный поток сначала пытается определить точку входа этой DLL. В случае успеха серверный поток вызывает точку входа DLL; в противном случае сервер создает процесс аналогично тому, как это делалось в программе serverNP. Пример DLL приведен в программе 12.3. Поскольку генерация исключений библиотекой DLL будет приводить к уничтожению всего серверного процесса, вызов функции DLL защищен простым обработчиком исключений.

При желании можно включить внутрипроцессные серверы и в программу serverNP. Самым большим преимуществом внутрипроцессных серверов является то, что они не требуют никакого контекстного переключения на другие процессы, в результате чего производительность может заметно улучшиться.

Поскольку в коде сервера использованы специфические для Windows возможности, в частности, возможности управления потоками и некоторые другие, он, в отличие от кода клиента, оказывается привязанным к Windows.

Программа 12.2. serverSK: сервер на основе сокета с внутрипроцессными серверами

```
    /* Глава 12. Клиент-серверная система. ПРОГРАММА СЕРВЕРА. ВЕРСИЯ НА ОСНОВЕ
СОКЕТА. */
    /* Выполняет указанную в запросе команду и возвращает ответ. */
    /* Если удается обнаружить точку входа разделяемой библиотеки, команды */
    /* выполняются внутри процесса, в противном случае - вне процесса. */
    /* ДОПОЛНИТЕЛЬНАЯ ВОЗМОЖНОСТЬ: argv [1] может содержать имя библиотеки */
    /* DLL, поддерживающей внутрипроцессные серверы. */

    #define _NOEXCLUSIONS
    #include "EvryThng.h"
    #include "ClntSrvr.h" /* Определяет структуру записей запроса и ответа. */

    struct sockaddr_in SrvSAddr;
```

```
    /* Адресная структура сокета сервера. */
    struct sockaddr_in ConnectSAddr; /* Подключенный сокет. */
    WSADATA WSStartData; /* Структура данных библиотеки сокета. */

    typedef struct SERVER_ARG_TAG { /* Аргументы серверного потока. */
     volatile DWORD number;
     volatile SOCKET sock;
     volatile DWORD status;
     /* Пояснения содержатся в комментариях к основному потоку. */
     volatile HANDLE srv_thd;
     HINSTANCE dlhandle; /* Дескриптор разделяемой библиотеки. */
    } SERVER_ARG;

    volatile static ShutFlag = FALSE;
    static SOCKET SrvSock, ConnectSock;
    int _tmain(DWORD argc, LPCTSTR argv[]) {
     /* Прослушивающий и подключенный сокеты сервера. */
     BOOL Done = FALSE;
     DWORD ith, tstatus, ThId;
     SERVER_ARG srv_arg[MAX_CLIENTS];
     HANDLE hAcceptTh = NULL;
     HINSTANCE hDll = NULL;
      /* Инициализировать библиотеку WSA; задана версия 2.0, но будет работать и
версия 1.1. */
     WSAStartup(MAKEWORD(2, 0), &WSStartData);
      /* Открыть динамическую библиотеку команд, если ее имя указано в командной
строке. */
     if (argc > 1) hDll = LoadLibrary(argv[1]);
     /* Инициализировать массив arg потока. */
     for (ith = 0; ith < MAXCLIENTS; ith++) {
      srv_arg[ith].number = ith;
      srv_arg[ith].status = 0;
      srv_arg[ith].sock = 0;
      srv_arg[ith].dlhandle = hDll;
      srv_arg[ith].srv_thd = NULL;
     }
       /*  Следовать  стандартной  процедуре  вызова  последовательности  функций
socket/bind/listen/accept клиентом. */
     SrvSock = socket(AF_INET, SOCK_STREAM, 0);
     SrvSAddr.sin_family = AF_INET;
     SrvSAddr.sin_addr.s_addr = htonl(INADDR_ANY);
     SrvSAddr.sin_port = htons(SERVER_PORT);
     bind(SrvSock, (struct sockaddr *)&SrvSAddr, sizeof SrvSAddr);
     listen(SrvSock, MAX_CLIENTS);

     /* Основной поток становится потоком прослушивания/соединения/контроля.*/
     /* Найти пустую ячейку в массиве arg потока сервера. */
      /* параметр состояния: 0 - ячейка свободна; 1 - поток остановлен; 2 — поток
выполняется; 3 - остановлена вся система. */
     while (!ShutFlag) {
      for (ith = 0; ith < MAX_CLIENTS && !ShutFlag; ) {
       if (srv_arg[ith].status==1 || srv_arg[ith].status==3) { /* Выполнение потока
завершено либо обычным способом, либо по запросу останова. */
        WaitForSingleObject(srv_arg[ith].srv_thd INFINITE);
        CloseHandle(srv_arg[ith].srv_tnd);
        if (srv_arg[ith].status == 3) ShutFlag = TRUE;
        else srv_arg[ith].status = 0;
        /* Освободить ячейку данного потока. */
       }
```

```
      if (srv_arg[ith].status == 0 || ShutFlag) break;
      ith = (ith + 1) % MAXCLIENTS;
      if (ith == 0) Sleep(1000);
      /* Прервать цикл опроса. */
        /* Альтернативный вариант: использовать событие для генерации сигнала,
указывающего на освобождение ячейки. */
     }
     /* Ожидать попытки соединения через данный сокет. */
     /* Отдельный поток для опроса флага завершения ShutFlag. */
      hAcceptTh = (HANDLE)_beginthreadex(NULL, 0, AcceptTh, &srv_arg[ith], 0,
&ThId);
     while (!ShutFlag) {
      tstatus = WaitForSingleObject(hAcceptTh, CS_TIMEOUT);
      if (tstatus == WAIT_OBJECT_0) break; /* Соединение установлено. */
     }
     CloseHandle(hAcceptTh);
     hAcceptTh = NULL; /* Подготовиться к следующему соединению. */
    }
    _tprintf(_T("Остановка сервера. Ожидание завершения всех потоков сервера\n"));
    /* Завершить принимающий поток, если он все еще выполняется. */
    /* Более подробная информация об используемой логике завершения */
    /* работы приведена на Web-сайте книги. */
    if (hDll != NULL) FreeLibrary(hDll);
    if (hAcceptTh != NULL) TerminateThread(hAcceptTh, 0);
    /* Ожидать завершения всех активных потоков сервера. */
    for (ith = 0; ith < MAXCLIENTS; ith++) if (srv_arg [ith].status != 0) {
     WaitForSingleObject(srv_arg[ith].srv_thd, INFINITE);
     CloseHandle(srv_arg[ith].srv_thd);
    }
    shutdown(SrvSock, 2);
    closesocket(SrvSock);
    WSACleanup();
    return 0;
   }

   static DWORD WINAPI AcceptTh(SERVER_ARG * pThArg) {
     /* Принимающий поток, который предоставляет основному потоку возможность
опроса флага завершения. Кроме того, этот поток создает серверный поток. */
    LONG AddrLen, ThId;
    AddrLen = sizeof(ConnectSAddr);
    pThArg->sock = accept(SrvSock, /* Это блокирующий вызов. */
     (struct sockaddr *)&ConnectSAddr, &AddrLen);
    /* Новое соединение. Создать серверный поток. */
    pThArg->status = 2;
    pThArg->srv_thd = (HANDLE)_beginthreadex (NULL, 0, Server, pThArg, 0, &ThId);
    return 0; /* Серверный поток продолжает выполняться. */
   }

   static DWORD WINAPI Server(SERVER_ARG * pThArg)
   /* Функция серверного потока. Поток создается по требованию. */
   {
    /* Каждый поток поддерживает в стеке собственные структуры данных запроса,
ответа и регистрационных записей. */
    /* … Стандартные объявления из serverNP опущены … */
    SOCKET ConnectSock;
    int Disconnect = 0, i;
    int (*dl_addr)(char *, char *);
    char *ws = " \0\t\n"; /* Пробелы. */
    GetStartupInfo(&StartInfoCh);
```

```
    ConnectSock = pThArg->sock;
    /* Создать имя временного файла. */
    sprintf(TempFile, "%s%d%s", "ServerTemp", pThArg->number, ".tmp");
    while (!Done && !ShutFlag) { /* Основной командный цикл. */
     Disconnect = ReceiveRequestMessage(&Request, ConnectSock);
        Done = Disconnect || (strcmp(Request.Record, "$Quit") == 0) ||
(strcmp(Request.Record, "$ShutDownServer") == 0);
     if (Done) continue;
      /* Остановить этот поток по получении команды "$Quit" или "$ShutDownServer".
*/
         hTrapFile = CreateFile(TempFile, GENERIC_READ | GENERIC_WRITE,
FILE_SHARE_READ | FILE_SHARE_WRITE, &TempSA, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL,
NULL);
     /* Проверка наличия этой команды в DLL. Для упрощения команды */
     /* разделяемой библиотеки имеют более высокий приоритет по сравнению */
     /* с командами процесса. Прежде всего, необходимо извлечь имя команды.*/
     i = strcspn(Request.Record, ws); /* Размер лексемы. */
     memcpy(sys_command, Request.Record, i) ;
     sys_command[i] = '\0';
      dl_addr = NULL; /* Будет установлен в случае успешного выполнения функции
GetProcAddress. */
      if (pThArg->dlhandle != NULL) {/* Проверка поддержки "внутрипроцессного"
сервера. */
         dl_addr = (int (*)(char *, char *))GetProcAddress(pThArg->dlhandle,
sys_command);
      if (dl_addr != NULL) __try {
       /* Защитить серверный процесс от исключений, возникающих в DLL*/
       (*dl_addr)(Request.Record, TempFile);
      } __except (EXCEPTION_EXECUTE_HANDLER) {
       ReportError(_T("Исключение в DLL"), 0, FALSE);
      }
     }
     if (dl_addr == NULL) { /* Поддержка внутрипроцессного сервера отсутствует. */
      /* Создать процесс для выполнения команды. */
      /* … То же, что в serverNP … */
     }
     /* … То же, что в serverNP … */
    } /* Конец основного командного цикла. Получить следующую команду. */
    /* Конец командного цикла. Освободить ресурсы; выйти из потока. */
    _tprintf(_T("Завершение работы сервера# %d\n"), pThArg->number);
    shutdown(ConnectSock, 2);
    closesocket(ConnectSock);
    pThArg->status = 1;
    if (strcmp(Request.Record, "$ShutDownServer") == 0) {
     pThArg->status = 3;
     ShutFlag = TRUE;
    }
    return pThArg->status;
   }
```

Замечания по поводу безопасности

В том виде, как она здесь представлена, данная клиент-серверная система *не* является безопасной. Если на вашей системе выполняется сервер и кому-то известен номер порта, через который вы работаете, и имя компьютера, то он может атаковать вашу систему. Другой пользователь, запустив клиентскую программу на своем компьютере, сможет выполнить на вашей системе команды, позволяющие, например, удалить или изменить файлы.

Полное обсуждение методов построения безопасных систем выходит за рамки данной книги. Тем не менее, в главе 15 показано, как обезопасить объекты Windows, а в упражнении 12.14 предлагается воспользоваться протоколом SSL.

Внутрипроцессные серверы

Как ранее уже отмечалось, основное усовершенствование программы serverSK связано с включением в нее внутрипроцессных серверов. В программе 12.3 показано, как написать библиотеку DLL, обеспечивающую услуги подобного рода. В программе представлены две уже известные вам функции — функция, осуществляющая подсчет слов, и функция toupper.

В соответствии с принятым соглашением первым параметром является командная строка, а вторым — имя выходного файла. Кроме того, следует всегда помнить о том, что функция будет выполняться в том же потоке, что и сервер, и это диктует необходимость соблюдения жестких требований относительно безопасности потоков, включая, но не ограничиваясь только этим, следующее:

• Функции никоим образом не должны изменять окружение процесса. Например, если одна из функций изменит рабочий каталог, то это окажет воздействие на весь процесс.

• Аналогично, функции не должны перенаправлять стандартный ввод и вывод.

• Такие ошибки программирования, как выход индекса или указателя за пределы отведенного диапазона или переполнение стека, могут приводить к порче памяти, относящейся к другому потоку или самому процессу.

• Утечка ресурсов, возникшая, например, в результате того, что системе не была своевременно возвращена освободившаяся память или не были закрыты дескрипторы, в конечном счете, окажет отрицательное воздействие на работу всей серверной системы.

Столь жесткие требования не предъявляются к процессам по той причине, что один процесс, как правило, не может нанести ущерб другим процессу, а после того, как процесс завершает свое выполнение, занимаемые им ресурсы автоматически освобождаются. В связи с этим служба, как правило, разрабатывается и отлаживается как поток, и лишь после того, как появится уверенность в надежности ее работы, она преобразуется в DLL.

В программе 12.3 представлена небольшая библиотека DLL, включающая две функции.

Программа 12.3. command: пример внутри процессных серверов

```
    /* Глава 12. commands.c. */
    /* Команды внутрипроцессного сервера для использования в serverSK и так далее.
*/
    /* Имеется несколько команд, реализованных в виде библиотек DLLs. */
    /* Функция каждой команды принимает два параметра и обеспечивает */
    /* безопасное выполнение в многопоточном режиме. Первым параметром */
    /* является строка: команда arg1 arg2 … argn */
    /* (то есть обычная командная строка), а вторым – имя выходного файла. … */

    static void extract_token(int, char *, char *);

    _declspec(dllexport)
    int wcip(char * command, char * output_file)
    /* Счетчик слов; внутрипроцессный. */
    /* ПРИМЕЧАНИЕ: упрощенная версия; результаты могут отличаться от тех, которые
обеспечивает утилита wc. */
    {
     extract_token(1, command, input_file);
     fin = fopen(input_file, "r");
     /* … */
```

```
 ch = nw = nc = nl = 0;
 while ((c = fgetc(fin)) != EOF) {
  /* … Стандартный код — для данного примера не является существенным … */
 }
 fclose(fin);
 /* Записать результаты. */
 fout = fopen(output_file, "w");
 if (fout == NULL) return 2;
 fprintf(fout, " %9d %9d %9d %s\n", nl, nw, nc, input_file);
 fclose(fout);
 return 0;
}

_declspec(dllexport)
int toupperip(char * command, char * output_file)
/* Преобразует входные данные к верхнему регистру; выполняется внутри процесса.
*/
/* Вторая лексема задает входной файл (первая лексема - "toupperip"). */
{
 /* … */
 extract_token(1, command, input_file);
 fin = fopen(input_file, "r");
 if (fin == NULL) return 1;
 fout = fopen(output_file, "w");
 if (fout == NULL) return 2;
 while ((c = fgetc (fin)) != EOF) {
  if (c == '\0') break;
  if (isalpha(c)) c = toupper(c);
  fputc(c, fout);
 }
 fclose(fin);
 fclose(fout);
 return 0;
}

static void extract_token(int it, char * command, char * token) {
 /* Извлекает из "команды" лексему номер "it" (номером первой лексемы */
 /* является "0"). Результат переходит в "лексему" (token) */
 /* В качестве разделителей лексем используются пробелы. … */
 return;
}
```

Ориентированные на строки сообщения, точкив хода DLL и TLS

Программы serverSK и clientSK взаимодействуют между собой, обмениваясь сообщениями, каждое из которых состоит из 4-байтового заголовка, содержащего размер сообщения, и собственно содержимого. Обычной альтернативой такому подходу служат сообщения, отделяемые друг от друга символами конца строки (или перевода строки).

Трудность работы с такими сообщениями заключается в том, что длина сообщения заранее не известна, в связи с чем приходится проверять каждый поступающий символ. Однако получение по одному символу за один раз крайне неэффективно, и поэтому символы сохраняются в буфере, содержимое которого может включать один или несколько символов конца строки и составные части одного или нескольких сообщений. *При этом в промежутках между вызовами функции получения сообщений необходимо поддерживать неизменным содержимое и состояние буфера.* В однопоточной среде для этой цели могут быть использованы ячейки статической памяти, но совместное использование несколькими потоками одной и той же статической переменной невозможно.

В более общей формулировке, мы сталкиваемся здесь с *проблемой сохранения долговременных состояний в многопоточной среде* (multithreaded persistent state problem). Эта проблема возникает всякий раз, когда безопасная в отношении многопоточного выполнения функция должна поддерживать сохранение некоторой информации от одного вызова функции к другому. Такая же проблема возникает при работе с функцией strtook, входящей в стандартную библиотеку C, которая предназначена для просмотра строки для последовательного нахождения экземпляров определенной лексемы.

Решение проблемы долговременных состояний в многопоточной среде

В искомом решении сочетаются несколько компонентов:

• Библиотека DLL, в которой содержатся функции, обеспечивающие отправку и прием сообщений.

• Функция, представляющая точку входа в DLL.

• Локальная область хранения потока (TLS, глава 7). Подключение процесса к библиотеке сопровождается созданием индекса DLL, а отключение — уничтожением. Значение индекса хранится в статическом хранилище, доступ к которому имеют все потоки.

• Структура, в которой хранится буфер и его текущее состояние. Структура распределяется всякий раз, когда к библиотеке подключается новый поток, и его адрес сохраняется в записи TLS для данного потока. При отсоединении потока от библиотеки память, занимаемая его структурой, освобождается.

Таким образом, TLS играет роль статического хранилища, и у каждого потока имеется собственная уникальная копия этого хранилища.

Пример: безопасная многопоточная DLL для обмена сообщениями через сокет

Программа 12.4 представляет собой DLL, содержащую две функции для обработки символьных строк (в именах которых в данном случае присутствует "CS", от *character string* — строка символов), или потоковые функции сокета (socket streaming functions): SendCSMessage и ReceiveCSMessage, а также точку входа DllMain (см. главу 5). Указанные две функции играют ту же роль, что и функция ReceiveMessage, а также функции, использованные в программах 12.1 и 12.2, и фактически заменяют их.

Функция DllMain служит характерным примером решения проблемы долговременных состояний в многопоточной среде и объединяет TLS и библиотеки DLL.

Освобождать ресурсы при отсоединении потоков (случай DLL_THREAD_DETACH) особенно важно в случае серверной среды; если этого не делать, то ресурсы сервера, в конечном счете, исчерпаются, что может привести к сбоям в его работе или снижению производительности или к тому и другому одновременно.

Примечание

Некоторые из иллюстрируемых ниже концепций прямого отношения к сокетам не имеют, но, тем не менее, рассматриваются именно здесь, а не в предыдущих главах, поскольку данный пример предоставляет удобную возможность для иллюстрации методов создания безопасных многопоточных DLL в реалистических условиях.

Использующие эту DLL коды клиента и сервера, незначительно измененные по сравнению с программами 12.1 и 12.2, доступны на Web-сайте книги.

Программа 12.4. SendReceiveSKST: безопасная многопоточная DLL

```
/* SendReceiveSKST.c — DLL многопоточного потокового сокета. */
/* В качестве разделителей сообщений используются символы конца */
/* строки ('\0'), так что размер сообщения заранее не известен. */
/* Поступающие данные буферизуются и сохраняются в промежутках между */
/* вызовами функций. */
/* Для этой цели используются локальные области хранения потоков */
/* (Thread Local Storage, TLS), обеспечивающие каждый из потоков */
/* собственным закрытым "статическим хранилищем". */

#define _NOEXCLUSIONS
#include "EvryThng.h"
#include "ClntSrvr.h" /* Определяет записи запроса и ответа. */

typedef struct STATIC_BUF_T {
 /* "static_buf" содержит "static_buf_len" байтов остаточных данных. */
 /* Символы конца строки (нулевые символы) могут присутствовать, а могут */
 /* и не присутствовать. */
 char static_buf[MAX_RQRS_LEN] ;
 LONG32 static_buf_len;
} STATIC_BUF;

static DWORD TlsIx = 0; /* Индекс TLS - ДЛЯ КАЖДОГО ПРОЦЕССА СВОЙ ИНДЕКС.*/
 /* Для однопоточной библиотеки использовались бы следующие определения:
```

```
     static char static_buf [MAX_RQRS_LEN];
     static LONG32 static_buf_len; */
    /* Основная функция DLL. */

   BOOL WINAPI DllMain(HINSTANCE hinstDLL, DWORD fdwReason, LPVOID lpvReserved) {
    STATIC_BUF * pBuf;
    switch (fdwReason) {
    case DLL_PROCESS_ATTACH:
     TlsIx = TlsAlloc();
     /* Для основного потока подключение отсутствует, поэтому во время подключения
процесса необходимо выполнить также операции по подключению потока. */
    case DLL_THREAD_ATTACH:
     /* Указать, что память не была распределена. */
     TlsSetValue(TlsIx, NULL);
     return TRUE; /* В действительности это значение игнорируется. */
    case DLL_PROCESS_DETACH:
     /* Отсоединить также основной поток. */
     pBuf = TlsGetValue(TlsIx);
     if (pBuf != NULL) {
      free(pBuf);
      pBuf = NULL;
     }
     return TRUE;
    case DLL_THREAD_DETACH:
     pBuf = TlsGetValue(TlsIx);
     if (pBuf != NULL) {
      free(pBuf);
      pBuf = NULL;
     }
     return TRUE;
    }
   }


   _declspec(dllexport)
   BOOL ReceiveCSMessage(REQUEST *pRequest, SOCKET sd) {
    /* Возвращаемое значение TRUE указывает на ошибку или отсоединение. */
    BOOL Disconnect = FALSE;
    LONG32 nRemainRecv = 0, nXfer, k; /* Должны быть целыми со знаком. */
    LPSTR pBuffer, message;
    CHAR TempBuf[MAX_RQRS_LEN + 1];
    STATIC_BUF *p;
    p = (STATIC_BUF *)TlsGetValue(TlsIx);
    if (p == NULL) { /* Инициализация при первом вызове. */
     /* Распределять это хранилище будут только те потоки, которым оно */
     /* необходимо. Другие типы потоков могут использовать TLS для иных целей. */
     p = malloc(sizeof(STATIC_BUF));
     TlsSetValue(TlsIx, p);
     if (p == NULL) return TRUE; /* Ошибка. */
     p->static_buf_len = 0; /* Инициализировать состояние. */
    }
    message = pRequest->Record;
    /* Считать до символа новой строки, оставляя остаточные данные в статическом
буфере. */
    for (k = 0; k < p->static_buf_len && p->static_buf[k] != '\0'; k++) {
     message[k] = p->static_buf[k];
    } /* k - количество переданных символов. */
    if (k < p->static_buf_len) { /* В статическом буфере обнаружен нулевой символ.
*/
     message[k] = '\0';
```

```
      p->static_buf_len -= (k + 1); /* Скорректировать состояние статического
буфера. */
      memcpy(p->static_buf, &(p->static_buf[k + 1]), p->static_buf_len);
      return FALSE; /* Входные данные сокета не требуются. */
     }

     /* Передан весь статический буфер. Признак конца строки не обнаружен.*/
     nRemainRecv = sizeof(TempBuf) - 1 - p->static_buf_len;
     pBuffer = message + p->static_buf_len;
     p->static_buf_len = 0;
     while (nRemainRecv > 0 && !Disconnect) {
      nXfer = recv(sd, TempBuf, nRemainRecv, 0);
      if (nXfer <= 0) {
       Disconnect = TRUE;
       continue;
      }
      nRemainRecv -= nXfer;
      /* Передать в целевое сообщение все символы вплоть до нулевого, если таковой
имеется. */
      for (k =0; k < nXfer && TempBuf[k] != '\0'; k++) {
       *pBuffer = TempBuf[k];
       pBuffer++;
      }
      if (k >= nXfer) { /*Признак конца строки не обнаружен, читать дальше*/
       nRemainRecv -= nXfer;
      } else { /* Обнаружен признак конца строки. */
       *pBuffer = '\0';
       nRemainRecv = 0;
       memcpy(p->static_buf, &TempBuf[k + 1], nXfer - k - 1);
       p->static_buf_len = nXfer - k - 1;
      }
     }
     return Disconnect;
    }

    _declspec(dllexport)
    BOOL SendCSMessage(RESPONSE *pResponse, SOCKET sd) {
     /* Послать запрос серверу в сокет sd. */
     BOOL Disconnect = FALSE;
     LONG32 nRemainSend, nXfer;
     LPSTR pBuffer;
     pBuffer = pResponse->Record;
     nRemainSend = strlen(pBuffer) + 1;
     while (nRemainSend > 0 && !Disconnect) {
      /* Отправка еще не гарантирует, что будет отослано все сообщение. */
      nXfer = send(sd, pBuffer, nRemainSend, 0);
      if (nXfer <= 0) {
       fprintf(stderr, "\nОтключение сервера до посылки запроса завершения");
       Disconnect = TRUE;
      }
      nRemainSend -=nXfer;
      pBuffer += nXfer;
     }
     return Disconnect;
    }
```

Комментарии по поводу DLL и безопасной многопоточной среды

• Всякий раз, когда создается новый поток, вызывается функция DllMain с опцией DLL_THREAD_ATTACH, но для основного потока отдельного вызова с опцией DLL_THREAD_ATTACH не существует. В случае основного потока должна использоваться опция DLL_PROCESS_ATTACH.

• Вообще говоря, в том числе и в данном случае (возьмите, например, поток, принимающий сообщения (accept thread)), некоторым потокам распределение памяти может и не требоваться, но DllMain не в состоянии различать отдельные типы потоков. Поэтому на участке кода, соответствующем варианту выбора DLL_THREAD_ATTACH, фактического распределения памяти не происходит; здесь только инициализируется параметр TLS. Распределение памяти осуществляется точкой входа ReceiveCSMessage при первом ее вызове. Благодаря этому собственная память выделяется только тем потокам, которые в этом действительно нуждаются, и различные типы потоков получают ровно столько ресурсов, сколько им требуется.

• Хотя рассматриваемая библиотека DLL и обеспечивает безопасную многопоточную поддержку, любой поток в каждый момент времени может работать только с одним сокетом, поскольку долговременные состояния ассоциируются не с сокетами, а с потоками. Этот момент учитывается в следующем примере.

• Исходным кодом DLL, размещенным на Web-сайте, предусмотрен вывод общего количества вызовов DllMain в соответствии с их типами.

• Даже при таком решении существует риск утечки ресурсов. Некоторые потоки, например поток приема сообщений, могут вообще не завершаться, и поэтому не будут отсоединены от библиотеки DLL. Для остающихся активных потоков функция ExitProcess вызовет DllMain с опцией DLL_PROCESS_DETACH, а не DLL_THREAD_DETACH. В данном случае никаких проблем не возникает, поскольку поток приема сообщений никаких ресурсов не распределяет, а освобождение памяти происходит по завершении процесса. Однако, проблемы возможны в тех случаях, когда потоки распределяют такие ресурсы, как временные файлы. Поэтому окончательное решение должно предусматривать создание глобально доступного списка ресурсов. Тогда участок кода, соответствующий опции DLL_PROCESS_DETACH, мог бы взять на себя просмотр этого списка и освобождение ненужных ресурсов.

Пример: альтернативная стратегия создания безопасных библиотек DLL с много поточной поддержкой

Хотя программа 12.4 и демонстрирует пример типичного объединения TLS и DllMain для создания библиотек, обеспечивающих безопасное многопоточное выполнение, в ней имеется одно слабое место, о котором говорится в комментариях к предыдущему разделу. В частности, "состояние" ассоциируется не с сокетом, а с потоком, поэтому в каждый момент времени любой поток может работать только с одним сокетом.

Эффективной альтернативой безопасной библиотеке функций является создание структуры, выступающей в качестве своего рода дескриптора, передаваемого при каждом вызове функции. Тогда состояние можно было бы хранить в этой структуре. Во многих системах на основе UNIX эта методика используется для создания безопасных библиотек C, обеспечивающих многопоточную поддержку. Основной недостаток такого подхода заключается в том, что для указания структуры состояния требуется вводить дополнительный параметр при вызове функции.

Программа 12.5 является видоизмененным вариантом программы 12.4. Заметьте, что DllMain теперь не требуется, но появились две новые функции, предназначенные для инициализации и освобождения ресурсов структуры состояния. Для функций send и receive потребовались лишь самые минимальные изменения. Соответствующая программа сервера, serverSKHA, доступна на Web-сайте книги и содержит лишь незначительные изменения, обеспечивающие создание и закрытие дескриптора сокета (HA означает "handle" — дескриптор).

Программа 12.5. SendReceiveSKHA: безопасная многопоточная DLL со структурой состояния

```
/* SendReceiveSKHA.c - многопоточный потоковый сокет. */
/* Данная программа представляет собой модифицированную версию программы*/
/* SendReceiveSKST.c, которая иллюстрирует другую методику, основанную */
/* на безопасной библиотеке с многопоточной поддержкой. */
/* Состояние сохраняется не в TLS, а в структуре состояния, напоминающей*/
/* дескриптор HANDLE. Благодаря этому поток может использовать сразу */
/* несколько сокетов. Сообщения разделяются символами конца строки ('\0')*/
#define _NOEXCLUSIONS
#include "EvryThng.h"
#include "ClntSrvr.h " /* Определяет записи запроса и ответа. */

typedef struct SOCKET_HANDLE_T {
 /* Текущее состояние сокета в структуре "handle". */
 /* Структура содержит "static_buf_len" символов остаточных данных. */
 /* Символы конца строки (нулевые символы) могут присутствовать, */
 /* а могут и не присутствовать. */
 SOCKET sk; /* Сокет, связанный с указанной структурой "handle". */
 char static_buf[MAX_RQRS_LEN];
 LONG32 static_buf_len;
} SOCKET_HANDLE, * PSOCKET_HANDLE;

/* Функции для создания и закрытия "дескрипторов потоковых сокетов". */
_declspec(dllexport)
PVOID CreateCSSocketHandle(SOCKET s) {
 PVOID p;
```

```
 PSOCKET_HANDLE ps;
 p = malloc(sizeof(SOCKET_HANDLE));
 if (p == NULL) return NULL;
 ps = (PSOCKET_HANDLE)p;
 ps->sk = s;
 ps->static_buf_len = 0; /* Инициализировать состояние буфера. */
 return p;
}

_declspec(dllexport)
BOOL CloseCSSocketHandle(PVOID p) {
 if (p == NULL) return FALSE;
 free(p);
 return TRUE;
}

_declspec(dllexport)
BOOL ReceiveCSMessage(REQUEST *pRequest, PVOID sh)
/* Тип PVOID используется для того, чтобы избежать включения */
/* в вызывающую программу определения структуры SOCKET_HANDLE. */
{
 /* Возвращаемое значение TRUE указывает на ошибку или отсоединение. … */
 PSOCKET_HANDLE p;
 SOCKET sd;
 p = (PSOCKET_HANDLE)sh;
 if (p == NULL) return FALSE;
 sd = p->sk;
 /* Этим исчерпываются все отличия от SendReceiveSKST! … */
}

_declspec(dllexport)
BOOL SendCSMessage(RESPONSE *pResponse, PVOID sh) {
 /* Послать запрос серверу в сокет sd. … */
 SOCKET sd;
 PSOCKET_HANDLE p;
 p = (PSOCKET_HANDLE)sh;
 if (p == NULL) return FALSE;
 sd = p->sk;
 /* Этим исчерпываются все отличия от SendReceiveSKST! … */
}
```

Дейтаграммы

Дейтаграммы аналогичны почтовым ящикам и используются при сходных обстоятельствах. Соединение между отправителем и получателем отсутствует, а получателей может быть несколько. Ни почтовые ящики, ни дейтаграммы не гарантируют доставку данных получателю, а последовательные сообщения не обязательно будут получены в той же очередности, в которой они были отправлены.

Первым шагом при использовании дейтаграмм является создание сокета посредством вызова функции socket с указанием значения SOCK_DGRAM в поле type.

Далее необходимо использовать функции sendto и recvfrom, которые принимают те же аргументы, что и функции send и recv, но имеют по два дополнительных аргумента, относящихся к станции-партнеру. Так, функция sendto имеет следующий прототип:

```
int sendto(SOCKET s, LPSTR lpBuffer, int nBufferLen, int nFlags,
LPSOCKADDR lpAddr, int nAddrLen);
```

lpAddr — указывает на адресную структуру, в которой вы можете задать имя конкретной системы и номер порта или же указать на необходимость рассылки дейтаграммы заданной совокупности систем.

Используя функцию recvfrom, вы указываете систему или системы (возможно, все), от которых вы хотите принимать дейтаграммы.

Использование дейтаграмм для удаленного вызова процедур

Обычно дейтаграммы применяются для реализации RPC. По сути дела, в самых распространенных ситуациях клиент посылает запрос серверу, используя дейтаграммы. Поскольку доставка запроса не гарантируется, клиент должен повторно передать запрос, если по истечении заданного периода ожидания ответ от сервера (для посылки которого также используются дейтаграммы) не получен. Сервер должен быть готов к тому, что один и тот же запрос может направляться ему несколько раз.

Важно отметить, что ни клиенту, ни серверу RPC служебные сигналы, которые, например, необходимы при образовании соединения через потоковый сокет, не требуются; вместо этого они связываются друг с другом посредством запросов и ответов. В качестве дополнительной возможности RPC может гарантировать надежность взаимодействия путем повторной передачи запросов по истечении периода ожидания, что упрощает разработку приложений. Выражаясь иначе, часто говорят о том, что клиент и сервер RPC *не имеют состояния* (они не хранят никакой информации относительно состояния текущего запроса или запросов, на которые еще не получен ответ). Отсюда следует, что результат обработки на сервере множества идентичных клиентских запросов будет тем же, что и результат обработки одиночного запроса. Это также значительно упрощает проектирование приложений и реализацию их логики.

Сравнение Berkeley Sockets и Windows Sockets

Программы, использующие стандартные вызовы Berkeley Sockets, будут работать и с Windows Sockets, если вы учтете следующие важные моменты:

• Для инициализации Winsock DLL вы должны вызвать функцию WSAStartup.

• Для закрытия сокета вы должны использовать не функцию close (которая является переносимой), а функцию closesocket (которая таковой не является).

• Для закрытия библиотеки DLL вы должны вызвать функцию WSACleanup.

При желании вы можете использовать типы данных Windows, например, SOCKET и LONG вместо int, как было сделано в этой главе. Программы 12.1 и 12.2 были перенесены из UNIX, и для этого потребовались самые минимальные усилия. Вместе с тем, потребовалось модифицировать DLL и разделы, осуществляющие управление процессами. В упражнении 12.13 вам предлагается перенести эти две программы обратно в UNIX.

Перекрывающийся ввод/вывод с использованием Windows Sockets

В главе 14 описывается асинхронный ввод/вывод, позволяющий потоку продолжать свое выполнение в процессе выполнения операции ввода/вывода. В той же главе обсуждается и совместное использование сокетов с асинхронным вводом/выводом Windows.

Большинство задач, связанных с программированием асинхронных операций, можно легко решить, применяя однотипный подход с использованием потоков. Так, в программе serverSK вместо неблокирующегося сокета используется принимающий поток (accept thread). Тем не менее, порты завершения ввода/вывода, связанные с асинхронным вводом/выводом, играют важную роль в обеспечении масштабируемости в случае большого количества клиентов. Эта тема также рассматривается в главе 14.

Windows Sockets 2

Версия Windows Sockets 2 вводит новые сферы функциональности и доступна на всех системах Windows, хотя системы Windows 9x требуют установки пакета обновления. В примерах использована версия 2.0, но можно применять и версию 1.1, если требуется совместимость с необновленными системами Windows 9x. Кроме того, возможностей версии 1.1 в большинстве случаев вам будет вполне достаточно. Версия Windows Sockets 2.0 обеспечивает, в частности, следующие возможности:

• Стандартизованная поддержка перекрывающегося ввода/вывода (см. главу 14). Эту возможность можно считать самым важным усовершенствованием.

• Фрагментированный ввод/вывод (scatter/gather I/O) (при посылке и получении данных используются буферы, расположенные в памяти вразброс).

• Возможность запрашивать качество обслуживания (скорость и надежность передачи информации).

• Возможность групповой организации сокетов. Допускается конфигурирование качества обслуживания группы сокетов, поэтому можно не делать этого для каждого сокета по отдельности. Кроме того, входящим в группу сокетам можно назначать приоритеты.

• Имеется возможность совмещения передачи прямых и обратных пакетов с запросами соединения (piggybacking).

• Создание многоточечных соединений (multipoint connections) (сопоставимо с подключениями по типу конференц-связи).

Резюме

Интерфейс Windows Sockets предоставляет возможность использования API, отвечающего требованиям промышленного стандарта, что гарантирует работу ваших программ на различном оборудовании и почти полную переносимость на уровне исходного кода. Winsock способен поддержать практически любой сетевой протокол, однако в большинстве случаев применяется протокол TCP/IP.

Winsock сопоставим с именованными каналами (и почтовыми ящиками) как в отношении функциональных возможностей, так и в отношении производительности, в наибольшей степени проявляя свои преимущества в тех случаях, когда на первый план выступают вопросы совместимости и переносимости программного обеспечения. Имейте в виду, что сокеты ввода/вывода не являются атомарными, поэтому необходимо специально заботиться о том, чтобы сообщения передавались полностью.

В этой главе были изложены наиболее существенные сведения о Winsock, которых достаточно для построения работоспособной системы. Вместе с тем, за рамками нашего рассмотрения осталось очень многое, в том числе и применение Winsock в асинхронных операциях; для получения более подробной информации по этому вопросу обратитесь к источникам, указанным в разделе "Дополнительная литература".

Кроме того, в этой главе были приведены примеры использования библиотек DLL для реализации внутрипроцессных серверов и создания безопасных в отношении многопоточного выполнения библиотек.

В следующих главах

В главах 11 и 12 было показано, как разрабатывать серверы, отвечающие на запросы клиентов. Серверы, в их различных воплощениях, являются распространенным типом приложений Windows. В главе 13 описываются службы Windows (Windows Services), которые обеспечивают стандартный способ создания серверов и управления ими в виде служб, что дает возможность организовать их запуск, остановку и мониторинг в автоматическом режиме. В главе 13 показано, как превратить сервер в управляемую службу.

Дополнительная литература

Windows Sockets

Сокетам Windows посвящена книга [28], а также сайт поддержки http://www.sockets.com. Однако указанная книга во многих аспектах устарела, и в ней не используются потоки. Более полезными для многих читателей будут книги, которые упоминаются ниже.

Berkeley Sockets и TCP/IP

В книге [41] рассмотрены не только сокеты, но и многое другое, тогда как в первых двух томах этой серии описаны протоколы и их реализация. Исчерпывающее рассмотрение интересующего нас вопроса содержится в книге [42], которая представляет ценность даже для

тех, кто имеет дело с другими семействами ОС. Среди источников, заслуживающих внимания, можно назвать [8] и [12].

Упражнения

12.1. Используя функцию WSAStartup, определите старший и младший номера версий Winsock, поддерживаемые на доступных вам системах.

12.2. Используйте программу JobShell из главы 6 для запуска сервера и нескольких клиентов, причем каждый клиент должен создаваться с опцией "отсоединения" (-d). Для окончания работы остановите сервер, послав сигнал управляющего события консоли посредством команды kill. Можете ли вы предложить какие-либо улучшения в организации остановки сервера в программе serverSK.

12.3. Модифицируйте программы клиента и сервера (программы 12.1 и 12.2) таким образом, чтобы для обнаружения сервера использовались дейтаграммы. В качестве отправной точки может быть использовано решение на основе почтового ящика из главы 11.

12.4. Модифицируйте сервер именованного канала из главы 11 (программа 11.3) таким образом, чтобы в нем использовались не потоки из пула потоков сервера, а потоки, создаваемые по требованию. Вместо предварительного указания максимально допустимого количества экземпляров именованного канала предоставьте системе возможность самостоятельно определять максимальное значение этого параметра.

12.5. Проведите эксперименты, чтобы определить, действительно ли внутрипроцессные серверы работают быстрее внепроцессных. Для этого, например, может быть использована программа подсчета слов (программа 12.3); имеется исполняемый файл этой программы (wc), а также функция библиотеки DLL, представленная в программе 12.3.

12.6. Количество клиентов, поддержку которых может обеспечить программа serverSK, ограничивается размером массива аргументов потоков сервера. Модифицируйте программу, сняв это ограничение. Для этого вам потребуется создать структуру данных, позволяющую добавлять и удалять аргументы потоков, а также обеспечить возможность просмотра структуры с целью отслеживания потоков сервера, завершивших выполнение.

12.7. Разработайте внутрипроцессные серверы другого рода. Например, с этой целью преобразуйте соответствующим образом программу grep (см. главу 6).

12.8. Усовершенствуйте сервер (программа 12.2) таким образом, чтобы можно было указывать несколько библиотек DLL в командной строке. Если разместить все DLL в памяти невозможно, разработайте стратегию их загрузки и выгрузки.

12.9. Исследуйте функцию setsockopt и использование опции SO_LINGER. Примените указанную опцию в одном из примеров сервера.

12.10. Используйте возможности фрагментированного ввода/вывода Windows Sockets 2.0 для упрощения функций отправки и приема сообщений в программах 12.1 и 12.2.

12.11. Обеспечьте невозможность утечки ресурсов в программе serverSK (за дополнительными разъяснениями обратитесь к упражнению 11.8). Проделайте то же самое с программой serverSKST, которая была модифицирована для использования DLL в программе 12.4.

12.12. Расширьте возможности обработчика исключений в программе 12.3 таким образом, чтобы он заносил информацию об исключении и типе исключения в конец временного файла, используемого для сохранения результатов работы сервера.

12.13. *Расширенное упражнение (требуется дополнительное оборудование).* Если у вас имеется доступ к UNIX-системе, связанной через сеть с Windows-системой, перенесите на UNIX-систему программу clientSK и попытайтесь, получив с ее помощью доступ к программе serverSK, запускать различные Windows-программы. Разумеется, при этом вам придется

преобразовать такие типы данных, как DWORD или SOCKET, в другие типы (в данном случае, соответственно, в unsigned int и int). Кроме того, вы должны убедиться в том, что данные, образующие сообщения, передаются в формате с обратным порядком байтов. Для выполнения соответствующих преобразований данных используйте такие функции, как htonl. Наконец, перенесите в UNIX программу serverSK, чтобы Windows-системы могли выполнять команды в UNIX. Преобразуйте вызов DLL в вызовы функций разделяемой библиотеки.

12.14. Ознакомьтесь с протоколом защищенных сокетов (Secure Sockets Layer, SSL), обратившись к материалам MSDN или источникам, указанным в разделе "Дополнительная литература". Усовершенствуйте программы, применив SSL для обеспечения безопасности связи клиента с сервером.s

ГЛАВА 13
Windows Services

Серверные программы, рассмотренные в главах 11 и 12, являются консольными приложениями, выполняющимися как фоновые задачи. Вообще говоря, эти серверы могут выполняться в течение неопределенно длительного времени, обслуживая многочисленных клиентов по мере того, как те будут подключаться к серверу, посылать запросы, принимать ответы и разрывать соединения. Таким образом, указанные серверы могут работать как службы непрерывного действия, однако, чтобы быть в полной мере эффективными, эти службы должны быть управляемыми.

Службы Windows Services,[33] известные ранее под названием NT Services, предоставляют все средства управления, необходимые для превращения наших серверов в службы, которые могут активизироваться по команде или во время запуска системы еще до входа в нее пользователей, приостанавливаться, а также возобновлять или прекращать свое выполнение. Службы могут даже осуществлять мониторинг работоспособности самих служб. Информация о службах хранится в системном реестре.

В конечном счете, любая серверная система наподобие тех, которые были разработаны в главах 11 и 12, должна быть преобразована в службу, особенно в тех случаях, когда она предназначена для использования широким кругом клиентов или внутри организации.

Windows предоставляет целый ряд служб; в качестве примера можно привести службы telnet, отправки и приема факсимильных сообщений, а также службы управления безопасностью учетных записей и драйверы устройств. Доступ ко всем службам можно получить через пиктограмму Administrative Tools (Администрирование), который находится в окне панели управления.

Примитивную форму управления сервером можно было наблюдать в приведенной в главе 6 программе JobShell (программа 6.3), которая обеспечивает возможность перевода сервера под управление задачи и его остановку путем посылки сигнала завершения работы. В то же время, службы Windows Services предоставляют гораздо более широкие возможности и отличаются высокой надежностью, как это будет продемонстрировано в данной главе на примере преобразования программы к форме, обеспечивающей управление службами Windows Services.

В данной главе также показано, как преобразовать существующее консольное приложение в службу Windows, осуществить ее установку, а также организовать мониторинг и управление этой службой. Кроме того, здесь рассматривается ведение журнала учета событий, что обеспечивает регистрацию действий службы.

Написание программ, реализующихслужбы Windows Services: обзор

Службы Windows выполняются под управлением диспетчера управления службами (Service Control Manager, SCM). Преобразование консольного приложения, такого как serverNP или serverSK, в службу Windows осуществляется в три этапа, после выполнения которых программа переходит под управление SCM.

1. Создание новой точки входа main(), которая регистрирует службу в SCM, предоставляя точки входа и имена логических служб.

2. Преобразование прежней функции точки входа main() в функцию *ServiceMain*(), которая регистрирует обработчик управляющих команд службы и информирует SCM о своем состоянии. Остальная часть кода, по существу, сохраняет прежний вид, хотя и может быть дополнена командами регистрации событий. Имя *ServiceMain*() является заменителем имени логической службы, причем логических служб может быть несколько.

3. Написание функции обработчика управляющих команд службы, которая должна предпринимать определенные действия в ответ на команды, поступающие от SCM.

По мере описания каждого из этих трех этапов будут даваться отдельные разъяснения, касающиеся создания служб, их запуска и управления ими. Более подробные сведения приводятся в последующих разделах, а взаимодействие между отдельными компонентами службы иллюстрируется на рис. 13.1 далее в этой главе.

Функция main()

Задачей новой функции main(), которая вызывается SCM, является регистрация службы в SCM и запуск диспетчера службы (service control dispatcher). Для этого необходимо вызвать функцию StartServiceControlDispatcher, передав ей имя (имена) и точку (точки) входа одной или нескольких логических служб.

```
BOOL StartServiceCtrlDispatcher(LPSERVICE_TABLE_ENTRY lpServiceStartTable)
```

Эта функция принимает единственный аргумент *lpServiceStartTable,* являющийся адресом массива элементов SERVICE_TABLE_ENTRY, каждый из которых представляет имя и точку входа логической службы. Конец массива обозначается двумя последовательными значениями NULL.

Функция возвращает значение TRUE, если регистрация службы прошла успешно. Если служба уже выполняется или возникают проблемы с обновлением записей реестра (HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services), функция завершается с ошибками, обработка которых может осуществляться обычным путем.

Основной поток процесса службы, которая вызывает функцию StartService-ControlDispatcher, связывает поток с SCM. SCM регистрирует службу с вызывающим потоком в качестве потока диспетчера службы. SCM не осуществляет возврата в вызывающий поток до тех пор, пока не завершат выполнение все службы. Заметьте, однако, что фактического запуска логических служб в этот момент не происходит; запуск службы требует вызова функции StartService, которая описывается далее в этой главе.

Типичная основная программа службы, соответствующая случаю единственной логической службы, представлена в программе 13.1.

Программа 13.1. main: точка входа main службы

```
#include "EvryThng.h"

void WINAPI ServiceMain(DWORD argc, LPTSTR argv[]);

static LPTSTR ServiceName = _T("SocketCommandLineService");

/* Главная программа запуска диспетчера службы. */
VOID _tmain(int argc, LPTSTR argv[]) {
 SERVICE_TABLE_ENTRY DispatchTable[] = {
  { ServiceName, ServiceMain },
  { NULL, NULL }
 };

  if (!StartServiceCtrlDispatcher(DispatchTable)) ReportError(_T("Ошибка при запуске диспетчера службы."), 1, TRUE);
 /* ServiceMain() начнет выполняться только после того, как ее */
 /* запустит SCM. Возврат сюда осуществляется только после того, */
 /* как завершится выполнение всех служб. */
 return;
}
```

Функции ServiceMain()

Эти функции, которые указываются в таблице диспетчеризации, фигурирующей в программе 13.1, представляют логические службы. По сути, эти функции являются усовершенствованными версиями основной программы, преобразуемой в службу, и каждая логическая служба будет активизироваться в ее собственном потоке SCM. В свою очередь, логическая служба может запускать дополнительные потоки, например, рабочие потоки сервера, которые использовались в программах serverSK и serverNP. Часто внутри службы существует только одна логическая служба. Логическая служба в программе 13.2 получена путем соответствующей адаптации основного сервера из программы 12.2. В то же время, логические службы на основе сокетов и именованных каналов могут выполняться в рамках одной и той же службы Windows, что потребует предоставления основных функций обеих служб.

Несмотря на то что функция *ServiceMain*() является адаптированным вариантом функции main() с ее параметрами, представляющими количество аргументов и содержащую их строку, между ними имеется одно незначительное отличие: функция службы должна быть объявлена с типом void, а не иметь возвращаемое значение типа int, как в случае обычной функции main().

Для регистрации обработчика управляющих команд службы, который представляет собой функцию, вызываемую SCM для осуществления управления службой, требуется дополнительный код.

Регистрация управляющей программы службы

Обработчик управляющих команд службы, вызываемый SCM, должен обеспечивать управление соответствующей логической службой. Возможности обработчиков такого рода в ограниченном виде иллюстрирует обработчик управляющих сигналов консоли в сервере serverSK, устанавливающий глобальный флаг завершения выполнения. Однако, прежде всего, каждая логическая служба должна немедленно зарегистрировать свой обработчик с помощью функции RegisterServiceCtrlHandlerEx. Вызов этой функции должен помещаться в начало функции *ServiceMain*() и впоследствии нигде не повторяться. Обработчик вызывается SCM после получения запроса службы.

```
    RegisterServiceCtrlHandlerEx(LPCTSTR                    lpServiceName,
LPHANDLER_FUNCTION_EX lpHandlerProc, LPVOID lpContext)
```

Параметры

lpServiceName — определяемое пользователем имя службы, которое предоставляется в соответствующем поле таблицы диспетчеризации, отведенном для данной логической функции.

lpHandlerProc — адрес функции расширенного обработчика, которая описывается в следующем разделе. Расширенный обработчик был добавлен в NT5, причем функция RegisterServiceCtrlHandlerEx заменяет функцию Register-ServiceCtrlHandler. Следующий параметр также был введен в NT5.

lpContext — определяемые пользователем данные, передаваемые обработчику. Благодаря этому обработчик может различать ассоциированные с ним службы, которых может быть несколько.

В случае ошибки возвращаемое функцией значение, которым является объект SEPARARE_STATUS_HANDLE, равно 0, а для анализа ошибок могут быть использованы обычные методы.

Настройка состояния службы

Теперь, когда управляющая программа зарегистрирована, необходимо сразу же перевести службу в состояние SERVICE_START_PENDING, воспользовавшись для этого функцией SetServiceStatus. Функция SetServiceStatus будет применяться еще в других местах для установки различных значений параметра состояния, информируя SCM о текущем состоянии службы. Описания других возможных состояний службы, характеризуемых значениями параметра состояния, отличными от SERVICE_STATUS_PENDING, приведены в табл. 13.3.

Обработчик службы должен устанавливать состояние службы при каждом вызове, даже если ее состояние не менялось.

Далее, любой из потоков службы может в любой момент вызвать функцию SetServiceStatus, чтобы сообщить данные, характеризующие степень выполнения задачи, а также предоставить информацию об ошибках или иную информацию, причем для периодического обновления состояния многие службы часто выделяют отдельный поток. Длительность временного промежутка между вызовами обновления состояния указывается в одном из полей структуры данных, выступающей в качестве параметра. Если в пределах указанного промежутка времени состояние не обновлялось, то SCM может предположить, что произошла ошибка.

```
BOOL SetServiceStatus(SERVICE_STATUS_HANDLE hServiceStatus, LPSERVICE
STATUS lpServiceStatus)
```

Параметры

hServiceStatus — дескриптор типа SERVICE_STATUS_HANDLE, возвращенный функцией RegisterCtrlHandlerEx. Поэтому вызову функции SetServiceStatus должен предшествовать вызов функции RegisterCtrlHandlerEx.

lpServiceStatus — указатель на структуру SERVICE_STATUS, содержащую описание свойств, состояния и возможностей службы.

Структура SERVICE_STATUS

Ниже приведено определение структуры SERVICE_STATUS.

```
typedef struct _SERVICE_STATUS {
 DWORD dwServiceType;
 DWORD dwCurrentState;
 DWORD dwControlsAccepted;
 DWORD dwWin32ExitCode;
 DWORD dwServiceSpecificExitCode;
 DWORD dwCheckPoint;
 DWORD dwWaitHint;
} SERVICE_STATUS, *LPSERVICE_STATUS;
```

Параметры

dwWin32ExitCode — обычный код завершения потока, используемый логической службой. Служба должна установить этот код равным NO_ERROR в процессе выполнения и при нормальном завершении.

dwServiceSpecificExitCode — может использоваться в качестве кода завершения, когда ошибка возникает при запуске или остановке службы, но это значение игнорируется, если значение параметра dwWin32ExitCode не было установлено равным ERROR_SERVICE_SPECIFIC_ERROR.

dwCheckPoint — служба должна периодически увеличивать значение этого параметра для индикации выполнения на всех стадиях, включая стадии инициализации и остановки. Этот параметр не действует и должен устанавливаться равным 0, если служба не находится в состоянии запуска, остановки, паузы и не выполняет никаких длительных операций.

dwWaitHint — ожидаемая длительность интервалов времени (в миллисекундах) между последовательными вызовами функции SetServiceStatus, осуществляемыми с увеличенным значением параметра dwCheckPoint или измененным значением параметра dwCurrentState. Как уже отмечалось ранее, если на протяжении этого промежутка времени вызова функции SetServiceStatus не происходит, то SCM предполагает, что это вызвано возникновением ошибки.

Остальные элементы структуры SERVICE_STATUS обсуждаются ниже по отдельности.

Тип службы

Параметр dwServiceType должен иметь одно из значений, описанных в таблице 13.1.

В наших примерах в качестве типа службы почти всегда будет предполагаться тип SERVICE_WIN32_OWN_PROCESS, но из приведенных в таблице значений видно, что службы могут играть множество различных ролей.

Таблица 13.1. Типы служб

Значение	**Описание**
SERVICE WIN32 OWN PROCESS	Указывает на службу Windows, выполняющуюся в собственном процессе с собственными ресурсами. *Используется в программе 13.2.*
SERVICE_WIN32_SHARE_PROCESS	Указывает на службу Windows, разделяющую процесс с другими службами, в результате чего несколько служб могут совместно использовать одни и те же ресурсы, переменные окружения и так далее.
SERVICE_KERNEL_DRIVER	Указывает на драйвер устройства Windows.
SERVICE_FILE_SYSTEM_DRIVER	Определяет драйвер файловой системы Windows.
SERVICE_INTERACTIVE_PROCESS	Указывает на процесс службы Windows, который может взаимодействовать с пользователем через рабочий стол.

Состояние службы

Значение параметра dwCurrentState указывает на текущее состояние службы. Возможные значения этого параметра перечислены в табл. 13.2.

Таблица 13.2. Значения параметра состояния службы

Значение	**Описание**
SERVICE_STOPPED	Служба не выполняется.
SERVICE_START_PENDING	Служба находится на стадии запуска, но пока не готова отвечать на запросы. Например, могут еще не быть запущены рабочие потоки.
SERVICE_STOP_PENDING	Служба находится на стадии остановки, но еще не завершила своего выполнения. Например, мог быть установлен глобальный флаг завершения, но рабочие потоки еще не успели на это отреагировать.
SERVICE_RUNNING	Служба выполняется.
SERVICE CONTINUE_PENDING	Служба переходит в состояние выполнения после нахождения в состоянии паузы.
SERVICE_PAUSE_PENDING	Служба переходит в состояние паузы, но ее безопасное нахождение в этом состоянии пока не обеспечено.
SERVICE PAUSED	Служба находится в состоянии паузы.

Воспринимаемые управляющие коды

Параметр dwControlsAccepted определяет управляющие коды, которые служба будет воспринимать и обрабатывать с помощью своего обработчика (см. следующий раздел). В табл. 13.3 указаны четыре возможных значения, которые могут объединяться посредством операции поразрядного "или" (|). Версия программы serverSK, которую мы впоследствии разработаем, будет воспринимать лишь три первых значения. Дополнительные значения приведены в разделе MSDN, содержащем описание структуры SERVICE_STATUS.

Таблица 13.3. Коды, воспринимаемые службой (неполный перечень)

Значение	**Описание**
SERVICE_ACCEPT_STOP	Разрешает посылку команды SERVICE_CONTROL_STOP.
SERVICE_ACCEPT_PAUSE_CONTINUE	Разрешает посылку команд SERVICE_CONTROL_PAUSE и SERVICE_CONTROL_CONTINUE.
SERVICE_ACCEPT_SHUTDOWN	Уведомляет службу о прекращении работы системы. Это дает системе возможность послать службе команду SERVICE_CONTROL_SHUTDOWN.
SERVICE_ACCEPT_PARAMCHANGE	Требуется NT5. Обеспечивает изменение параметров запуска без выполнения самого перезапуска. Соответствующей командой является SERVICE_CONTROL_PARAMCHANGE

Специфический для службы код

После того как обработчик зарегистрирован и для службы установлено состояние SERVICE_START_PENDING, служба может инициализировать себя и вновь установить свое состояние. Если говорить о преобразованной версии serverSK, то сразу же после того, как сокеты будут инициализированы, а сервер готов к работе с клиентами, должно быть установлено состояние SERVICE_RUNNING.

Обработчик управляющих команд службы

Обработчик управляющих команд службы, то есть функция косвенного вызова, определяемая с помощью функции RegisterServiceCtrlHandlerEx, имеет следующий прототип:

```
DWORD WINAPI HandlerEx(DWORD dwControl, DWORD dwEventType, LPVOID lpEventData, LPVOID lpContext)
```

dwControl — обозначает фактическую управляющую команду, поступившую в обработчик от SCM. До появления NT5 и введения функции RegisterServiceCtrlHandlerEx этот параметр был единственным параметром обработчика.

Всего существует 14 возможных значений параметра dwControl, включая те, которые перечислены в табл. 13.3, хотя некоторые команды поддерживаются только в NT5 или XP. Нас будут интересовать следующие значения, которые используются в примере:

```
SERVICE_CONTROL_STOP
SERVICE_CONTROL_PAUSE
SERVICE_CONTROL_CONTINUE
SERVICE_CONTROL_INTERROGATE
SERVICE_CONTROL_SHUTDOWN
```

Разрешены также пользовательские значения, определяемые в интервале 128-255, однако нам они не понадобятся.

dwEventType — обычно принимает значение 0, в то время как ненулевые значения используются для управления устройствами, но рассмотрение этого вопроса выходит за рамки данной книги. Параметр dwEventType определяет дополнительную информацию, которая требуется соответствующим событиям.

Наконец, lpContext — пользовательские данные, передаваемые в функцию RegisterServiceCtrlHandlerEx во время регистрации обработчика.

Обработчик активизируется SCM в том же потоке, что и основная программа, и обычно содержит ряд операторов switch, как будет показано в приведенных ниже примерах.

Пример: "интерфейсная оболочка" службы

Программа 13.2 реализует преобразованный вариант программы serverSK, который мы перед этим обсуждали. Преобразование сервера в службу сопряжено с решением всех ранее описанных задач. После внесения незначительных изменений существующий код сервера помещается в функцию ServiceSpecific. Поэтому представленный ниже код, по сути, является оболочкой (wrapper) существующей программы сервера, точка входа которой main заменена на ServiceSpecific.

Другим дополнением, которое здесь не представлено, но включено в вариант программы, находящийся на Web-сайте книги, является использование журнала службы, поскольку службы часто запускаются без интерактивной консоли, никак себя видимо не проявляя.

Программа 13.2. SimpleService: оболочка службы

```
    /* Глава 13. serviceSK.c
       Преобразование сервера serverSK в службу Windows.
         Несмотря на рассмотрение частного случая, оболочка имеет универсальный
характер. */

    #include "EvryThng.h"
    #include "ClntSrvr.h"
    #define UPDATE_TIME 1000 /* Интервал обновления - 1 секунда. */

    VOID LogEvent(LPCTSTR, DWORD, BOOL);
    void WINAPI ServiceMain(DWORD argc, LPTSTR argv[]);
    VOID WINAPI ServerCtrlHandlerEx(DWORD; DWORD, LPVOID, LPVOID);
    void UpdateStatus (int, int); /* Вызывает, функцию SetServiceStatus. */
    int ServiceSpecific (int, LPTSTR *); /* Ранее программа main. */
    volatile static BOOL ShutDown = FALSE, PauseFlag = FALSE;
    static SERVICE_STATUS hServStatus;
    static SERVICE_STATUS_HANDLE hSStat; /* Дескриптор, используемый при установке
состояния. */

    static LPTSTR ServiceName = _T("SocketCommandLineService");
    static LPTSTR LogFileName = _T("CommandLineServiceLog.txt");

    /* Основная процедура, запускающая диспетчер управления службой. */
    VOID _tmain(int argc, LPTSTR argv[]) {
     SERVICE_TABLE_ENTRY DispatchTable[] = {
      { ServiceName, ServiceMain }, { NULL, NULL }
     };
     StartServiceCtrlDispatcher(DispatchTable);
     return 0;
    }

    /* Точка входа ServiceMain, вызываемая при создании службы. */
    void WINAPI ServiceMain(DWORD argc, LPTSTR argv[]) {
     DWORD i, Context = 1;
      /* Установить текущий каталог и открыть файл журнала, присоединяемый к
существующему файлу. */
     /* Определить все элементы структуры состояния сервера. */
     hServStatus.dwServiceType = SERVICE_WIN32_OWN_PROCESS;
     hServStatus.dwCurrentState = SERVICE_START_PENDING;
```

```
    hServStatus.dwControlsAccepted = SERVICE_ACCEPT_STOP | SERVICE_ACCEPT_SHUTDOWN
| SERVICE_ACCEPT_PAUSE_CONTINUE;
    hServStatus.dwWin32ExitCode = ERROR_SERVICE_SPECIF0C_ERROR;
    hServStatus.dwServiceSpecificExitCode = 0;
    hServStatus.dwCheckPoint = 0;
    hServStatus.dwWaitHint = 2 * CS_TIMEOUT;
      hSStat   =   RegisterServiceCtrlHandlerEx(ServiceName,   ServerCtrlHandler,
&Context);
    SetServiceStatus(hSStat, &hServStatus);
    /* Запустить специфическую для службы обработку; выполнение типового участка
кода завершено. */
    if (ServiceSpecific(argc, argv) != 0) {
     hServStatus.dwCurrentState = SERVICE_STOPPED;
     hServStatus.dwServiceSpecificExitCode = 1;
     /* Ошибка при инициализации сервера. */
     SetServiceStatus(hSStat, &hServStatus);
     return;
    }
     /*  Возврат  сюда  будет  осуществлен  лишь  после  завершения  функции
ServiceSpecific, указывающего на прекращение работы системы. */
    UpdateStatus(SERVICE_STOPPED, 0);
    return;
   }


   void UpdateStatus(int NewStatus, int Check)
   /* Определить новое состояние и контрольную точку — задается либо истинное
значение, либо приращение. */
   {
    if (Check < 0) hServStatus.dwCheckPoint++;
    else hServStatus.dwCheckPoint = Check;
    if (NewStatus >= 0) hServStatus.dwCurrentState = NewStatus;
    SetServiceStatus(hSStat, &hServStatus);
    return;
   }


   /* Функция обработчика, активизируемая SCM для выполнения в том же */
   /* потоке, что и основная программа. */
   /* Последние три параметра не используются, так что обработчики, написанные*/
   /* для версий Windows младше NT5, в этом примере также будут работать. */
   VOID  WINAPI  ServerCtrlHandlerEx(DWORD  Control,  DWORD  EventType,  LPVOID
lpEventData, LPVOID lpContext) {
    switch (Control) {
    case SERVICE_CONTROL_SHUTDOWN:
    case SERVICE_CONTROL_STOP:
     ShutDown = TRUE; /* Установить глобальный флаг завершения. */
     UpdateStatus(SERVICE_STOP_PENDING, -1);
     break;
    case SERVICE_CONTROL_PAUSE:
     PauseFlag = TRUE; /* Периодический опрос. */
     break;
    case SERVICE_CONTROL_CONTINUE:
     PauseFlag = FALSE;
     break;
    case SERVICE_CONTROL_INTERROGATE:
     break;
    default:
     if (Control > 127 && Control < 256) /*Пользовательские сигналы.*/
      break;
    }
```

```
     UpdateStatus(-1, -1); /* Инкрементировать контрольную точку. */
     return;
    }

    /* Эта специфическая для службы функция играет роль функции "main" и вызывается
из более общей функции ServiceMain. Вообще говоря, вы можете взять любой сервер,
например ServerNP.c, и поместить его код прямо сюда, переименовав функцию "main" в
"ServiceSpecific". Однако для кода обновления состояния потребуются некоторые
изменения. */
    int ServiceSpecific(int argc, LPTSTR argv[]) {
     UpdateStatus(-1, -1); /* Инкрементировать контрольную точку. */
     /* … Инициализация системы … */
     /* Обеспечьте периодическое обновление контрольной точки. */
     return 0;
    }
```

Управление службами Windows

Следующее, что потребуется сделать после написания кода службы — поместить ее под управление SCM, что позволит запускать и останавливать службу, а также осуществлять любые иные формы управления, какие только могут понадобиться.

Для этого необходимо выполнить несколько шагов, включая открытие SCM, создание службы под управлением SCM и последующий ее запуск. При этом вы воздействуете не непосредственно на службу, а на SCM, который, в свою очередь, и осуществляет управление заданной службой.

Открытие SCM

Для создания службы требуется отдельный процесс, выступающий в качестве "администратора" и играющий во многом ту же роль, что и программа JobShell, которая использовалась в главе 6 для запуска задач. Первый шаг состоит в открытии SCM и получении дескриптора, который впоследствии будет использован для создания службы.

```
SC_HANDLE OpenSCManager(LPCTSTR lpMachineName, LPCTSTR lpDatabaseName,
DWORD dwDesiredAccess)
```

Параметры

lpMachineName — указатель на строку с именем сетевого компьютера, на котором установлен SCM, или NULL, если SCM установлен на локальном компьютере.

lpDatabaseName — обычно принимает значение NULL.

dwDesiredAccess — обычно указывается значение SC_MANAGER_ALL_ACCESS, соответствующее полным правам доступа, но можно задать и ограничить эти права, о чем более подробно говорится в оперативной справочной системе.

Создание и удаление службы

Для регистрации службы следует вызвать функцию CreateService:

```
SC_HANDLE CreateService(SC_HANDLE hSCManager, LPCTSTR lpServiceName,
LPCTSTR lpDisplayName, DWORD dwDesiredAccess, DWORD dwServiceType, DWORD
dwStartType, DWORD dwErrorControl, LPCTSTR lpBinaryPathName, LPCTSTR
lpLoadOrderGroup, LPDWORD lpdwTagId, LPCTSTR lpDependencies, LPCTSTR
lpServiceStartName, LPCTSTR lpPassword);
```

Информация о новых службах записывается в следующий раздел реестра:
`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services`

Параметры

hSCManager — дескриптор типа SC_HANDLE, полученный через функцию OpenSCManager.

lpServiceName — имя, используемое при последующих ссылках на службу и являющееся

одним из имен логических служб, определенных в диспетчерской таблице при вызове функции StartServiceCtrlDispatcher. Заметьте, что для каждой логической службы используется отдельный вызов CreateService.

lpDisplayName — имя, которое будет отображаться в реестре в качестве его раздела, а также в административной утилите Services (доступ к которой открывается через пиктограмму Administrative Tools в панели управления). Это имя появится в указанных местах сразу же после успешного завершения функции CreateService.

dwDesiredAccess — может принимать значение SERVICE_ALL_ACCESS или комбинацию значений GENERIC_READ, GENERIC_WRITE и GENERIC_EXECUTE. Дополнительную информацию вы можете получить, ознакомившись с оперативной справочной документацией.

dwServiceType — возможные значения перечислены в табл. 13.1.

dwStartType — указывает способ запуска службы. В наших примерах используется значение SERVICE_DEMAND_START, соответствующее запуску по требованию, тогда как другие значения (SERVICE_BOOT_START и SERVICE_SYSTEM_START) обеспечивают запуск служб драйверов устройств на стадии начальной загрузки или во время загрузки системы, а значение SERVICE_AUTO_START указывает на то, что служба должна быть запущена во время запуска системы.

lpBinaryPathName — имя исполняемого файла службы; указывать расширение .exe не требуется.

Другие параметры используются для указания имени учетной записи и пароля, групп, объединяющих несколько служб, а также зависимостей, если существует несколько отдельных служб.

Конфигурационные параметры существующей службы можно изменить с помощью функции ChangeServiceConfig или, в случае NT5, ChangeService-Config2. Служба идентифицируется по своему дескриптору, и для большинства параметров вы можете указать новые значения. Например, можно предоставить новые значения параметров dwServiceType или dwStartType, но в случае параметра dwAccess это сделать невозможно.

Доступна также функция OpenService, которая позволяет получить дескриптор именованной службы. Для удаления службы из реестра используется функция DeleteService, а для закрытия дескрипторов SC_HANDLE — функция CloseServiceHandle.

Запуск службы

Созданная служба сразу не выполняется. Для этого необходимо вызвать функцию ServiceMain(), указав дескриптор, полученный при помощи функции CreateService, а также параметры командной строки argc и argv, ожидаемые основной функцией службы (то есть функцией, указанной в таблице диспетчеризации).

```
BOOL StartService(SC_HANDLE hService, DWORD argc, LPTSTR argv[])
```

Управление службой

Чтобы начать управление службой, вы должны сообщить SCM о необходимости активизации обработчика управляющих команд службы с указанным управляющим кодом.

```
BOOL ControlService(SC_HANDLE hService, DWORD dwControlCode,
LPSERVICE_STATUS lpServStat)
```

Параметр dwControlCode, если доступ разрешен, может принимать одно из следующих значений:

```
SERVICE_CONTROL_STOP
SERVICE_CONTROL_PAUSE
SERVICE_CONTROL_CONTINUE
SERVICE_CONTROL_INTERROGATE
SERVICE_CONTROL_SHUTDOWN
```

или значение, определенное пользователем, лежащее в пределах диапазона 128–255. Эти значения совпадают с теми, которые использовались вместе с флагом dwControl в функции ServerCtrlHandler.

lpServStat — указатель на структуру SERVICE_STATUS, которая получает текущее состояние. Это та же структура, которая использовалась функцией SetServiceStatus.

Опрос состояния службы

Для получения структурой SERVICE_STATUS текущего состояния службы используется следующая функция:

```
BOOL QueryServiceStatus(SC_HANDLE hService, LPSERVICE_STATUS lpServiceStatus)
```

Резюме: функционирование и управление службой

На рис. 13.1 показано, каким образом SCM связан со службами и программой управления службами, подобной программе 13.3, которая рассматривается в следующем разделе. В частности, SCM должен зарегистрировать службу, и все команды, предназначенные для службы, должны пропускаться через SCM.

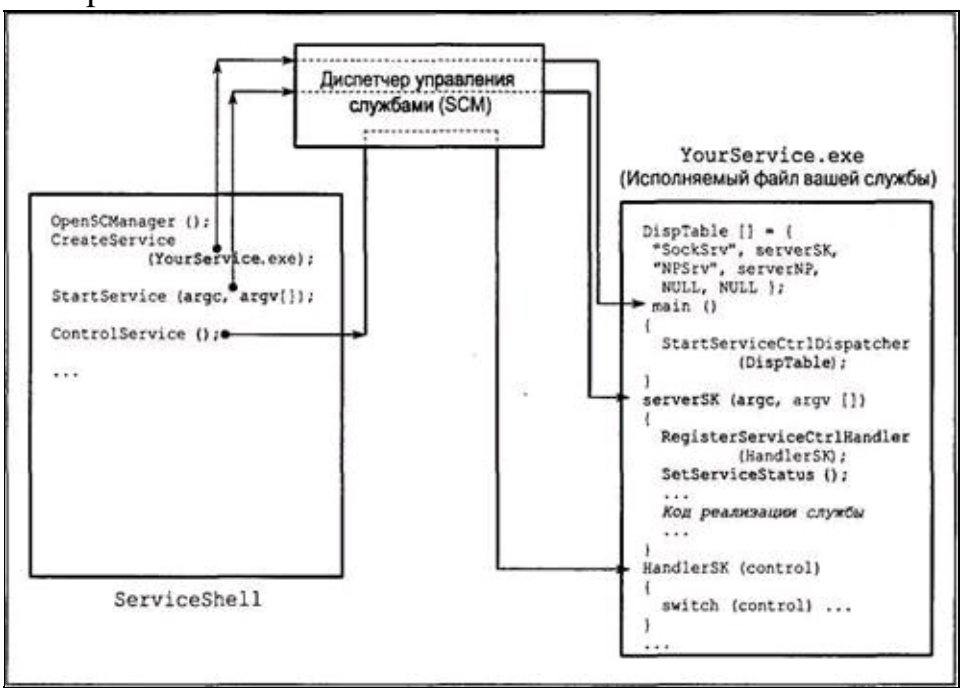


Рис. 13.1. Управление службами Windows через SCM

Пример:командная оболочка управления службами

Управление службами часто осуществляется посредством утилит, входящих в группу Administrative Tools, доступ к которым открывается через пиктограмму Services (Службы). Для управления пользовательскими службами можно также использовать оболочку ServiceShell (программа 13.3), представляющую собой видоизмененный вариант программы JobShell из главы 6 (программа 6.3).

Программа 13.3. ServiceShell: программа управления службами

```
    /* Глава 13. */
    /* ServiceShell.c. Программа командной оболочки управления службами Windows.
          Эта программа является видоизмененным вариантом программы управления
задачами из главы 6, но только управляет службами, а не задачами. */
    /* Поддерживаемые команды:
       create — создание службы
       delete - удаление службы
       start - запуск службы
       control - управление службой */
    #include "EvryThng.h"

    static SC_HANDLE hScm;
    static BOOL Debug;

    int _tmain(int argc, LPTSTR argv[]) {
     BOOL Exit = FALSE;
     TCHAR Command[MAX_COMMAND_LINE + 10], *pc;
     DWORD i, LocArgc; /* Локальный параметр argc. */
     TCHAR argstr[MAX_ARG][MAX_COMMAND_LINE];
     LPTSTR pArgs[MAX_ARG];
     /* Подготовить локальный массив "argv" в виде указателей на строки. */
     for (i = 0; i < MAX_ARG; i++) pArgs[i] = argstr[i];
     /* Открыть диспетчер управления службами на локальной машине. */
     hScm = OpenSCManager(NULL, NULL, SC_MANAGER_ALL_ACCESS);
     /* Главный цикл обработки команд. */
     _tprintf(_T("\nУправление службами Windows Services"));
     while (!Exit) {
      _tprintf(_T ("\nSM$"));
      _fgetts(Command, MAX_COMMAND_LINE, stdin);
      … Как для JobShell …
      if (_tcscmp(argstr [0], _T("create")) == 0) {
       Create(LocArgc, pArgs, Command);
      }
      … Аналогичным образом для всех команд …
     }
     CloseServiceHandle(hScm);
     return 0;
    }

    int Create(int argc, LPTSTR argv[], LPTSTR Command) {
     /* Создание новой службы в виде службы, запускаемой "по требованию":
        argv[1]: имя службы
        argv[2]: отображаемое имя службы
        argv[3]: название исполняемого файла */
```

```
     SC_HANDLE hSc;
     TCHAR CurrentDir[MAX_PATH +1], Executable[MAX_PATH + 1];
       hSc    =    CreateService(hScm,    argv[1],    argv[2],    SERVICE_ALL_ACCESS,
SERVICE_WIN32_OWN_PROCESS, SERVICE_DEMAND_START, SERVICE_ERROR_NORMAL, Executable,
NULL, NULL, NULL, NULL, NULL);
     return 0;
    }

    /* Удаление службы - argv [1]: имя удаляемой службы. */
    int Delete(int argc, LPTSTR argv[], LPTSTR Command) {
     SC_HANDLE hSc;
     hSc = OpenService(hScm, argv[1], DELETE);
     DeleteService(hSc);
     CloseServiceHandle(hSc);
     return 0;
    }

    /* Запуск именованной службы - argv [1] : имя запускаемой службы. */
    int Start(int argc, LPTSTR argv[], LPTSTR Command) {
     SC_HANDLE hSc;
     TCHAR WorkingDir[MAX_PATH + 1];
     LPTSTR pWorkingDir = WorkingDir;
     LPTSTR argvStart[] = {argv[1], WorkingDir};
     GetCurrentDirectory(MAX_PATH + 1, WorkingDir);
     hSc = OpenService(hScm, argv[1], SERVICE_ALL_ACCESS);
     /* Запустить службу с одним аргументом — именем рабочего каталога. */
     /* Примечание: по умолчанию имя службы совпадает с именем, */
     /* связанным с дескриптором hSc посредством функции OpenService. */
     /* Вместе с тем, функция ServiceMain это не проверяет. */
     StartService(hSc, 2, argvStart);
     CloseServiceHandle(hSc);
     return 0;
    }

    /* Управление именованной службой.
       argv[1]: имя управляемой службы.
         argv[2]: управляющая команда: stop (остановка), pause (пауза), resume
(возобновление), interrogate (опрос). */
    static LPCTSTR Commands[] = {"stop," "pause," "resume," "interrogate," "user"};
    static DWORD Controls[] = {
     SERVICE_CONTROL_STOP, SERVICE_CONTROL_PAUSE,
     SERVICE_CONTROL_CONTINUE, SERVICE_CONTROL_INTERROGATE, 128
    };

    int Control(int argc, LPTSTR argv[], LPTSTR Command) {
     SC_HANDLE hSc;
     SERVICE_STATUS ServiceStatus;
     DWORD dwControl, i;
     BOOL Found = FALSE;
      for  (i=  0;  i  <  sizeof(Controls)/sizeof(DWORD)  &&  !Found;  i++)  Found  =
(_tcscmp(Commands [i], argv[2]) == 0);
     if (!Found) {
      _tprintf(_T("\nНесуществующая команда управления %s"), argv[1]);
      return 1;
     }
     dwControl = Controls[i - 1];
     hSc = OpenService(hScm, argv[1], SERVICE_INTERROGATE | SERVICE_PAUSE_CONTINUE
| SERVICE_STOP | SERVICE_USER_DEFINED_CONTROL | SERVICE_QUERY_STATUS);
     ControlService(hSc, dwControl, &ServiceStatus);
```

```
 if (dwControl == SERVICE_CONTROL_INTERROGATE) {
  QueryServiceStatus (hSc, &ServiceStatus);
  printf(_T("Состояние, полученное при помощи QueryServiceStatus\n"));
  printf(_T("Состояние службы\n"));
  … Вывести всю остальную информацию о состоянии …
 }
 if (hSc != NULL) CloseServiceHandle(hSc);
 return 0;
}
```

Совместное использование объектов ядра приложениями и службами

Возможны ситуации, в которых служба и приложения разделяют объект ядра. Например, служба может использовать именованный мьютекс для защиты разделяемой области памяти, используемой для обмена данными с приложениями. Более того, в нашем примере также будет применяться разделяемый объект ядра, которым в данном случае является отображение файла.

Существует одна трудность, связанная с тем, что контекст безопасности приложений отличается от контекста безопасности служб, выполняющихся от имени системной учетной записи. Даже если защита не требуется, было бы нелогично создавать и (или) открывать разделяемые объекты ядра с указателем атрибутов безопасности, установленным в NULL (см. глава 15). Вместо этого необходим, по крайней мере, нулевой список разграничительного контроля доступа (см. главу 15), то есть приложения и служба должны использовать ненулевую структуру атрибутов защиты. В общем случае вы захотите защитить объекты, и этот вопрос также будет рассмотрен в главе 15.

Следует обратить ваше внимание также на то, что если служба выполняется от имени системной учетной записи, то могут возникать трудности с доступом службы к таким ресурсам, как разделяемые файлы, находящиеся на других машинах.

Регистрация событий

Службы часто выполняются, внешне ничем себя не проявляя, без диалогового взаимодействия с пользователем. Некоторые службы создают консоль, окно сообщений [34] или окно для взаимодействия с пользователем, но лучше всего записывать информацию о событиях в файл регистрации событий или использовать соответствующие функциональные возможности, предоставляемые Windows. Такая информация сохраняется в реестре, и ее можно просматривать с помощью специальной программы просмотра событий, предоставляемой группой инструментов Administrative Tools, пиктограмма которой находится в панели управления.

В доступных на Web-сайте книги программах ServiceSK.c и SimpleService.c показано, как организовать регистрацию заслуживающих внимания событий и ошибок в регистрационном файле, а в комментариях к текстам упомянутых программ рассказано о том, как используется регистрация событий. Для этого имеются три функции, описанные в оперативной справочной документации.

1. RegisterEventSource — позволяет получить дескриптор регистрационного файла.
2. ReportEvent — используется для внесения записи в регистрационный файл.
3. DeregisterEventSource — закрывает дескриптор регистрационного файла.

Замечания по отладке службы

Предполагается, что служба будет выполняться непрерывно, поэтому она должна быть надежной и по возможности лишенной каких бы то ни было дефектов. Несмотря на возможность подключения службы к отладчику и использования журнала регистрации событий для отслеживания операций, выполняемых службой, эти методы являются наиболее подходящими в условиях, когда служба уже развернута.

Однако на стадии первоначальной разработки и отладки службы часто гораздо легче воспользоваться преимуществами оболочки службы, представленной в программе 13.2.

• Разработайте сначала "предварительную" версию службы в виде отдельной программы. В таком ключе, например, была разработана программа serverSK.

• Используйте в программе средства регистрации событий или предусмотрите регистрационный файл.

• Когда вы придете к заключению, что программа готова к развертыванию в виде службы, переименуйте основную точку входа и свяжите ее с кодом оболочки службы, представленным программой 13.2 (он находится на Web-сайте книги вместе с двумя программами: SimpleService.c и serviceSK.c).

• Весьма важную роль играет дальнейшее тестирование службы для обнаружения дополнительных логических ошибок и проблем с обеспечением безопасности. Службы могут выполняться от имени системной учетной записи, но не иметь доступа к пользовательским объектам, и обнаружения проблем подобного рода "предварительная" версия службы не гарантирует.

• Если служба нуждается в интенсивной поддержке, извлеките ее код из оболочки и превратите его вновь в отдельную программу или консольное приложение, используя для этого GUI или текстовый интерфейс. Можно поступить и по-другому, предусмотрев для функции *ServiceMain* дополнительный аргумент командной строки, используемый в качестве флага отладки или трассировки.

Резюме

Службы Windows предоставляют стандартные возможности подключения пользовательских служб к системе Windows. Используя методы, обсуждавшиеся в этой главе, любую независимую программу можно превратить в службу.

Для создания служб, а также управления ими и контроля их функционирования можно воспользоваться средствами Administrative Tools (Администрирование) или представленной в этой главе программой ServiceShell. Управление развернутыми службами и их мониторинг осуществляются через SCM, и информация обо всех служб заносится в реестр.

В следующих главах

В главе 14 описывается асинхронный ввод/вывод, который предоставляет два метода, позволяющих выполнять множественные операции ввода/вывода параллельно с другими видами обработки. При этом использование нескольких потоков не является обязательным и требуется всего лишь один пользовательский поток.

В большинстве случаев программировать многопоточное выполнение проще, чем асинхронный ввод/вывод, и производительность в первом случае, как правило, выше. В то же время, без асинхронного ввода/вывода не обойтись, работая с портами завершения ввода/вывода, которые оказываются чрезвычайно полезными при построении масштабируемых серверов, способных взаимодействовать с большим количеством клиентов.

В главе 14 также рассматриваются таймеры ожидания.

Дополнительная литература

Эта тема подробно обсуждается в [21]. Драйверы устройств и их взаимодействие со службами в настоящей главе не рассматривались; соответствующая информация содержится, например, в [24].

Упражнения

13.1. Расширьте возможности службы serviceSK таким образом, чтобы она могла воспринимать команды приостановки.

13.2. При опросе состояния службы оболочка ServiceShell просто выводит соответствующие числовые данные. Расширьте возможности службы, обеспечив вывод информации о состоянии службы в текстовой форме.

13.3. Преобразуйте сервер serverNP (программа 11.3) в службу.

13.4. Видоизмените службу serviceSK, введя в нее средства регистрации событий.

ГЛАВА 14
Асинхронный ввод/вывод и порты завершения

Операциям ввода и вывода присуща более медленная скорость выполнения по сравнению с другими видами обработки. Причиной такого замедления являются следующие факторы:

• Задержки, обусловленные затратами времени на поиск нужных дорожек и секторов на устройствах произвольного доступа (диски, компакт-диски).

• Задержки, обусловленные сравнительно низкой скоростью обмена данными между физическими устройствами и системной памятью.

• Задержки при передаче данных по сети с использованием файловых, серверов, хранилищ данных и так далее.

Во всех предыдущих примерах операции ввода/вывода выполняются *синхронно с потоком,* поэтому весь поток вынужден простаивать, пока они не завершатся.

В этой главе показано, каким образом можно организовать продолжение выполнения потока, не дожидаясь завершения операций ввода/вывода, что будет соответствовать выполнению потоками *асинхронного* ввода/вывода. Различные методики, доступные в Windows, иллюстрируются примерами.

Некоторые из этих методик используются в таймерах ожидания, которые также описываются в настоящей главе.

Наконец, что особенно важно, изучив стандартные асинхронные операции ввода/вывода, мы сможем использовать *порты завершения ввода/вывода,* которые оказываются чрезвычайно полезными при построении масштабируемых серверов, способных обеспечивать поддержку большого количества клиентов без создания для каждого из них отдельного потока. Программа 14.4 представляет собой модифицированный вариант разработанного ранее сервера, позволяющий использовать порты завершения ввода/вывода.

Обзор методов асинхронного ввода/вывода Windows

В Windows выполнение асинхронного ввода/вывода обеспечивается в соответствии с тремя методиками.

• **Многопоточный ввод/вывод (Multihreaded I/O).** Каждый из потоков внутри процесса или набора процессов выполняет обычный синхронный ввод/вывод, но при этом другие потоки могут продолжать свое выполнение.

• **Перекрывающийся ввод/вывод (Overlapped I/O).** Запустив операцию чтения, записи или иную операцию ввода/вывода, поток продолжает свое выполнение. Если потоку для продолжения выполнения требуются результаты ввода/вывода, он ожидает, пока не станет доступным соответствующий дескриптор или не наступит заданное событие. В Windows 9x перекрывающийся ввод/вывод поддерживается только для последовательных устройств, например именованных каналов.

• **Процедуры завершения (или расширенный ввод/вывод) (Completion routines (extended I/O)).** Когда наступает завершение операций ввода/вывода, система вызывает специальную *процедуру завершения,* выполняющуюся внутри потока. Расширенный ввод/вывод для дисковых файлов в Windows 9x не поддерживается.

Многопоточный ввод/вывод с использованием именованных каналов применен в сервере с многопоточной поддержкой, который рассматривался в главе 11. Программа grepMT (программа 7.1) управляет параллельным выполнением операций ввода/вывода с участием нескольких файлов. Таким образом, мы уже располагаем рядом программ, которые выполняют многопоточный ввод/вывод и тем самым обеспечивают одну из форм асинхронного ввода/вывода.

Перекрывающийся ввод/вывод является предметом рассмотрения следующего раздела, а в приведенных в нем примерах, реализующих преобразование файлов (из ASCII в UNICODE), эта методика применена для иллюстрации возможностей последовательной обработки файлов. С этой целью используется видоизмененный вариант программы 2.4. Вслед за перекрывающимся вводом/выводом рассматривается расширенный ввод/вывод, использующий процедуры завершения.

Примечание

Методы перекрывающегося и расширенного ввода/вывода часто оказываются сложными в реализации, редко обеспечивают какие-либо преимущества в отношении производительности, иногда даже становясь причиной ее ухудшения, а в случае файлового ввода/вывода способны работать лишь под управлением Windows NT. Эти проблемы преодолеваются с помощью потоков, *поэтому, вероятно, многие читатели захотят сразу же перейти к разделам, посвященным таймерам ожидания и портам завершения ввода/вывода,* возвращаясь к этому разделу по мере необходимости. С другой стороны, элементы асинхронного ввода/вывода присутствуют как в устаревших, так и в новых технологиях, в связи с чем эти методы все-таки стоит изучить.

Так, технология COM на платформе NT5 поддерживает асинхронный вызов методов, поэтому указанная методика может пригодиться многим читателям, которые используют или собираются использовать технологию COM. Кроме того, много общего с расширенным вводом/выводом имеют операции асинхронного вызова процедур (глава 10), и хотя лично я предпочитаю использовать потоки, другие могут отдать предпочтение именно этому механизму.

Перекрывающийся ввод/вывод

Первое, что необходимо сделать для организации асинхронного ввода/вывода, будь то перекрывающегося или расширенного, — это установить атрибут перекрывания (overlapped attribute) для файлового или иного дескриптора. Для этого при вызове CreateFile или иной функции, в результате которого создается файл, именованный канал или иной дескриптор, следует указать флаг FILE_FLAG_OVERLAPPED.

В случае сокетов (глава 12), независимо от того, были они созданы с использованием функции socket или accept, атрибут перекрывания устанавливается по умолчанию в Winsock 1.1, но должен устанавливаться явным образом в Winsock 2.0. Перекрывающиеся сокеты могут использоваться в асинхронном режиме во всех версиях Windows.

До этого момента структуры OVERLAPPED использовались нами совместно с функцией LockFileEx, а также в качестве альтернативы использованию функции SetFilePointer (глава 3), но они также являются существенным элементом перекрывающегося ввода/вывода. Эти структуры выступают в качестве необязательных параметров при вызове четырех приведенных ниже функций, которые могут блокироваться при завершении операций.

```
ReadFile
WriteFile
TransactNamedPipe
ConnectNamedPipe
```

Вспомните, что при указании флага FILE_FLAG_OVERLAPPED в составе параметра dwAttrsAndFlags (в случае функции CreateFile) или параметра dwOpen-Mode (в случае функции CreateNamedPipe) соответствующие файл или канал могут использоваться только в режиме перекрывания. С анонимными каналами перекрывающийся ввод/вывод не работает.

> **Примечание**
>
> В документации по функции CreateFile есть упоминание о том, что использование флага FILE_FLAG_NO_BUFFERING улучшает характеристики быстродействия перекрывающегося ввода/вывода. Эксперименты показывают лишь незначительное повышение производительности (примерно на 15%, что может быть проверено путем экспериментирования с программой 14.1), но вы должны убедиться в том, что суммарный размер считываемых данных при выполнении операций ReadFile или WriteFile, кратен размеру сектора диска.

Перекрывающиеся сокеты

Одним из наиболее важных нововведений в Windows Sockets 2.0 (глава 12) является стандартизация перекрывающегося ввода/вывода. В частности, сокеты уже не создаются автоматически как дескрипторы файлов с перекрытием. Функция socket создает неперекрывающийся дескриптор. Чтобы создать перекрывающийся сокет, следует вызвать функцию WSASocket, явно запросив создание перекрывающегося совета путем указания значения WSA_FLAG_OVERLAPPED для параметра dwFlags функции WSASocket.

```
SOCKET WSAAPI WSASocket(int iAddressFamily, int iSocketType, int
iProtocol, LPWSAPROTOCOL_INFO lpProtocolInfo, GROUP g, DWORD dwFlags);
```

Для создания сокета используйте вместо функции socket функцию WSASocket. Любой сокет, возвращенный функцией accept, будет иметь те же свойства, что и аргумент.

Следствия применения перекрывающегося ввода/вывода

Перекрывающийся ввод/вывод выполняется в асинхронном режиме. Это имеет несколько следствий.

• Операции перекрывающегося ввода/вывода не блокируются. Функции ReadFile, WriteFile, TransactNamedPipe и ConnectNamedPipe осуществляют возврат, не дожидаясь завершения операции ввода/вывода.

• Возвращаемое функцией значение не может быть использовано в качестве критерия успешности или неудачи ее выполнения, поскольку операция ввода/вывода к этому моменту еще не успевает завершиться. Для индикации состояния выполнения ввода/вывода требуется привлечение другого механизма.

• Возвращенное значение количества переданных байтов также приносит мало пользы, поскольку передача данных могла не завершиться до конца. Для получения такого рода информации Windows должна предоставить другой механизм.

• Программа может многократно предпринимать попытки чтения или записи с использованием одного и того же перекрывающегося дескриптора файла. Поэтому незначащим оказывается и указатель файла, соответствующий такому дескриптору. Следовательно, должен быть предусмотрен дополнительный метод, обеспечивающий указание позиции в файле для каждой операции чтения или записи. В случае именованных каналов, в силу присущего им последовательного характера обработки данных, это не является проблемой.

• Для программы должна быть обеспечена возможность ожидания (синхронизации) завершения ввода/вывода. При наличии нескольких незавершенных операций ввода/вывода, связанных с одним и тем же дескриптором, программа должна быть в состоянии определить, какие из операций уже завершились. Операции ввода/вывода вовсе не обязательно завершаются в том же порядке, в каком они начинали выполняться.

Для преодоления двух последних из перечисленных выше трудностей используются структуры OVERLAPPED.

Структуры OVERLAPPED

С помощью структуры OVERLAPPED (указываемой, например, параметром lpOverlapped функции ReadFile) можно указывать следующую информацию:

• Позицию в файле (64 бита), с которой должно начинаться выполнение операции чтения или записи в соответствии с обсуждением, которое содержится в главе 3.

• Событие (сбрасываемое вручную), которое будет переходить в сигнальное состояние по завершении соответствующей операции.

Ниже приводится определение структуры OVERLAPPED.

```
typedef struct_OVERLAPPED {
 DWORD Internal;
 DWORD InternalHigh;
 DWORD Offset;
 DWORD OffsetHigh;
 HANDLE hEvent;
} OVERLAPPED
```

Для задания позиции в файле (указателя) должны использоваться оба поля Offset и

OffsetHigh, хотя старшая часть указателя (OffsetHigh) во многих случаях равна 0. Не следует использовать поля Internal и InternalHigh, зарезервированные для системных нужд.

Параметр hEvent — дескриптор события (созданного посредством функции CreateEvent). Это событие может быть как именованным, так и неименованным, но оно *должно* быть обязательно сбрасываемым вручную (см. главу 8), если используется для перекрывающегося ввода/вывода; причины этого будут вскоре объяснены. По завершении операции ввода/вывода событие переходит в сигнальное состояние.

В другом возможном варианте его использования дескриптор hEvent имеет значение NULL; в этом случае программа может ожидать перехода в сигнальное состояние дескриптора файла, который также может выступать в роли объекта синхронизации (см. приведенные далее предостережения). Система использует для отслеживания завершения операций сигнальные состояния дескриптора файла, если дескриптор hEvent равен NULL, то есть объектом синхронизации в этом случае является дескриптор файла.

> **Примечание**
>
> В целях удобства термин "дескриптор файла" ("file handle"), используемый по отношению к дескрипторам, указываемым при вызове функций ReadFile, WriteFile и так далее, будет применяться нами даже в тех случаях, когда речь идет о дескрипторах именованного канала или устройства, а не файла.

При выполнении вызова функций ввода/вывода это событие сразу же сбрасывается системой (устанавливается в несигнальное состояние). Когда операция ввода/вывода завершается, событие устанавливается в сигнальное состояние и остается в нем до тех пор, пока не будет использовано другой операцией ввода/вывода. Событие должно быть сбрасываемым вручную, если его перехода в сигнальное состояние могут ожидать несколько потоков (хотя в наших примерах используется всего один поток), и на момент завершения операции они могут не находиться в состоянии ожидания.

Даже если дескриптор файла является синхронным (то есть созданным без флага FILE_FLAG_OVERLAPPED), структура OVERLAPPED может послужить в качестве альтернативы функции SetFilePointer для указания позиции в файле. В этом случае возврат после вызова функции ReadFile или иного вызова не происходит до тех пор, операция ввода/вывода пока не завершится. Этой возможностью мы уже воспользовались в главе 3. Также обратите внимание на то, что незавершенные операции ввода/вывода однозначно идентифицируются комбинацией дескриптора файла и соответствующей структуры OVERLAPPED.

Ниже перечислены некоторые предостережения, которые следует принимать во внимание.

• Не допускайте повторного использования структуры OVERLAPPED в то время, когда связанная с ней операция ввода/вывода, если таковая имеется, еще не успела завершиться.

• Аналогичным образом, избегайте повторного использования события, указанного в структуре OVERLAPPED.

• Если существует несколько незакрытых запросов, относящихся к одному и тому же перекрывающемуся дескриптору, используйте для синхронизации не дескрипторы файлов, а дескрипторы событий.

• Если структура OVERLAPPED или событие выступают в качестве автоматических переменных внутри блока, обеспечьте невозможность выхода из блока до синхронизации с операцией ввода/вывода. Кроме того, во избежание утечки ресурсов следует позаботиться о закрытии дескриптора до выхода из блока.

Состояния перекрывающегося ввода/вывода

Возврат из функций ReadFile и WriteFile, а также двух указанных выше функций, относящихся к именованным каналам, в случаях, когда они используются для выполнения перекрывающихся операций ввода вывода, осуществляется немедленно. В большинстве случаев операция ввода/вывода к этому моменту завершена не будет, и возвращаемым значением при чтении и записи будет FALSE. Функция GetLastError возвратит в этой ситуации значение ERROR_IO_PENDING.

По окончании ожидания перехода объекта синхронизации (события или, возможно, дескриптора файла) в сигнальное состояние, свидетельствующее о завершении операции, вы должны выяснить, сколько байтов было передано. В этом и состоит основное назначение функции GetOverlappedResult.

```
BOOL GetOverlappedResult(HANDLE hFile, LPOVERLAPPED lpOverlapped,
LPWORD lpcbTransfer, BOOL bWait)
```

Указание конкретной операции ввода/вывода обеспечивается сочетанием дескриптора и структуры OVERLAPPED. Значение TRUE параметра bWait указывает на то, что до завершения операции функция GetOverlappedResult должна находиться в состоянии ожидания; в противном случае возврат из функции должен быть немедленным. В любом случае эта функция будет возвращать значение TRUE только после успешного завершения операции. Если возвращаемым значением функции GetOverlappedResult является FALSE, то функция GetLastError возвратит значение ERROR_IO_INCOMPLETE, что позволяет вызывать эту функцию для опроса завершения ввода/вывода.

Количество переданных байтов хранится в переменной *lpcbTransfer. Всегда убеждайтесь в том, что с момента ее использования в операции перекрывающегося ввода/вывода структура OVERLAPPED остается неизменной.

Отмена выполнения операций перекрывающегося ввода/вывода

Булевская функция CancelIO позволяет отменить выполнение незавершенных операций перекрывающегося ввода/вывода, связанных с указанным дескриптором (у этой функции имеется всего лишь один параметр). Отменяется выполнение всех инициированных вызывающим потоком операций, использующих данный дескриптор. На операции, инициированные другими потоками, вызов этой функции никакого влияния не оказывает. Отмененные операции завершаются С ошибкой ERROR OPERATION ABORTED.

Пример: использование дескриптора файла в качестве объекта синхронизации

Перекрывающийся ввод/вывод очень удобно и просто реализуется в тех случаях, когда может существовать только одна незавершенная операция. Тогда для целей синхронизации программа может использовать не событие, а дескриптор файла.

Приведенный ниже фрагмент кода показывает, каким образом программа может инициировать операцию чтения для считывания части файла, продолжить свое выполнение для осуществления других видов обработки, а затем перейти в состояние ожидания перехода дескриптора файла в сигнальное состояние.

```
OVERLAPPED ov = { 0, 0, 0, 0, NULL /* События не используются. */ };
HANDLE hF;
DWORD nRead;
BYTE Buffer[BUF_SIZE];
…
hF = CreateFile( …, FILE_FLAG_OVERLAPPED, … );
ReadFile(hF, Buffer, sizeof(Buffer), &nRead, &ov);
/* Выполнение других видов обработки. nRead не обязательно достоверно.*/
/* Ожидать завершения операции чтения. */
WaitForSingleObject(hF, INFINITE);
GetOverlappedResult(hF, &ov, &nRead, FALSE);
```

Пример: преобразование файлов с использованием перекрывающегося ввода/вывода и множественной буферизации

Программа 2.4 (atou) осуществляла преобразование ASCII-файла к кодировке UNICODE путем последовательной обработки файла, а в главе 5 было показано, как выполнить такую же последовательную обработку с помощью отображения файлов. В программе 14.1 (atouOV) та же самая задача решается с использованием перекрывающегося ввода/вывода и множественных буферов, в которых хранятся записи фиксированного размера.

Рисунок 14.1 иллюстрирует организацию программы с четырьмя буферами фиксированного размера. Программа реализована таким образом, чтобы количество буферов можно было определять при помощи символической константы препроцессора, но в нижеследующем обсуждении мы будем предполагать, что существуют четыре буфера.

Сначала в программе выполняется инициализация всех элементов структур OVERLAPPED, определяющих события и позиции в файлах. Для каждого входного и выходного буферов предусмотрена отдельная структура OVERLAPPED. После этого для каждого из входных буферов инициируется операция перекрывающегося чтения. Далее с помощью функции WaitForMultipleObjects в программе организуется ожидание одиночного события, указывающего на завершение чтения или записи. При завершении операции чтения входной буфер копируется и преобразуется в соответствующий выходной буфер, после чего инициируется операция записи. При завершении записи инициируется следующая операция чтения. Заметьте, что события, связанные с входными и выходными буферами размещаются в единственном массиве, который используется в качестве аргумента при вызове функции WaitForMultipleObjects.

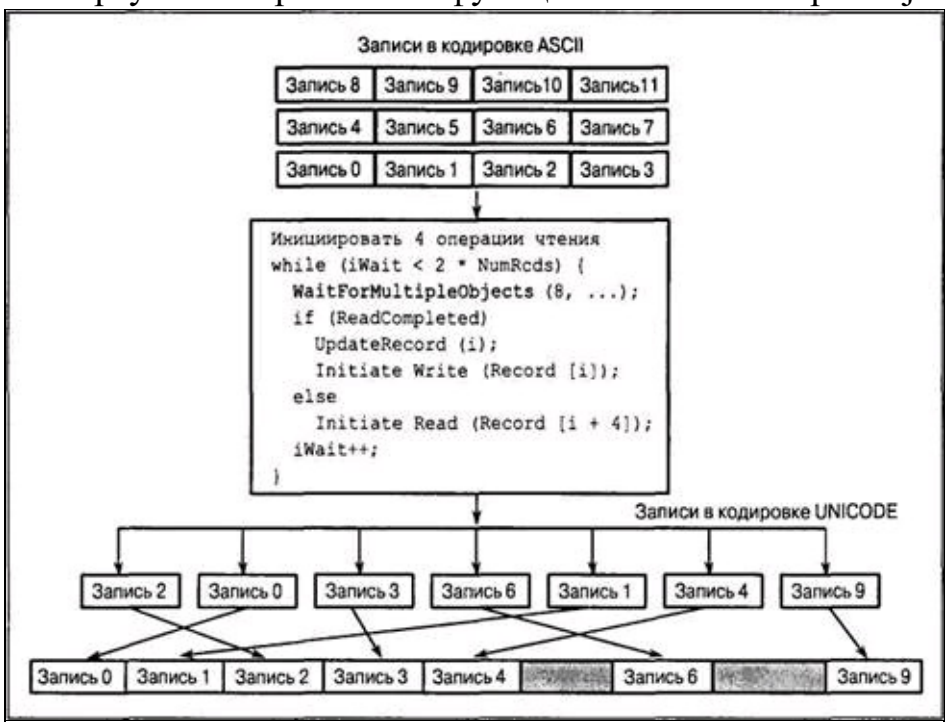


Рис. 14.1. Модель асинхронного обновления файла

Программа 14.1. atouOV: преобразование файла с использованием перекрывающегося ввода/вывода

```
/* Глава 14. atouOV
```

```
    Преобразование файла из кодировки ASCII в кодировку Unicode с использованием
перекрывающегося ввода/вывода. Программа работает только в Windows NT. */

    #include "EvryThng.h"

    #define MAX_OVRLP 4 /* Количество перекрывающихся операций ввода/вывода.*/
    #define REC_SIZE 0x8000 /* 32 Кбайт: Минимальный размер записи, обеспечивающий
приемлемую производительность. */
    #define UREC_SIZE 2 * REC_SIZE

    int _tmain(int argc, LPTSTR argv[]) {
     HANDLE hInputFile, hOutputFile;
     /* Каждый из элементов определенных ниже массивов переменных */
     /* и структур соответствует отдельной незавершенной операции */
     /* перекрывающегося ввода/вывода. */
     DWORD nin[MAX_OVRLP], nout[MAX_OVRLP], ic, i;
     OVERLAPPED OverLapIn[MAX_OVRLP], OverLapOut[MAX_OVRLP];
     /* Необходимость использования сплошного, двумерного массива */
     /* диктуется Функцией WaitForMultipleObjects. */
     /* Значение 0 первого индекса соответствует чтению, значение 1 - записи.*/
     HANDLE hEvents[2][MAX_OVRLP];
     /* В каждом из определенных ниже двух буферных массивов первый индекс */
     /* нумерует операции ввода/вывода. */
     CHAR AsRec[MAX_OVRLP][REC_SIZE];
     WCHAR UnRec[MAX_OVRLP][REC_SIZE];
     LARGE_INTEGER CurPosIn, CurPosOut, FileSize;
     LONGLONG nRecord, iWaits;
      hInputFile  =  CreateFile(argv[1],  GENERIC_READ,  0,  NULL,  OPEN_EXISTING,
FILE_FLAG_OVERLAPPED, NULL);
      hOutputFile  =  CreateFile(argv[2],  GENERIC_WRITE,  0,  NULL,  CREATE_ALWAYS,
FILE_FLAG_OVERLAPPED, NULL);
     /* Общее количество записей, подлежащих обработке, вычисляемое */
     /* на основе размера входного файла. Запись, находящаяся в конце, */
     /* может быть неполной. */
     FileSize.LowPart = GetFileSize(hInputFile, &FileSize.HighPart);
     nRecord = FileSize.QuadPart / REC_SIZE;
     if ((FileSize.QuadPart % REC_SIZE) != 0) nRecord++;
     CurPosIn.QuadPart = 0;
     for (ic = 0; ic < MAX_OVRLP; ic++) {
      /* Создать события чтения и записи для каждой структуры OVERLAPPED.*/
      hEvents[0][ic] = OverLapIn[ic].hEvent /* Событие чтения.*/
       = CreateEvent(NULL, TRUE, FALSE, NULL);
      hEvents[1][ic] = OverLapOut[ic].hEvent /* Событие записи. */
       = CreateEvent(NULL, TRUE, FALSE, NULL);
      /* Начальные позиции в файле для каждой структуры OVERLAPPED. */
      OverLapIn[ic].Offset = CurPosIn.LowPart;
      OverLapIn[ic].OffsetHigh = CurPosIn.HighPart;
        /*  Инициировать  перекрывающуюся  операцию  чтения  для  данной  структуры
OVERLAPPED. */
       if (CurPosIn.QuadPart < FileSize.QuadPart) ReadFile(hInputFile, AsRec[ic],
REC_SIZE, &nin[ic], &OverLapIn[ic]);
      CurPosIn.QuadPart += (LONGLONG)REC_SIZE;
     }
     /* Выполняются все операции чтения. Ожидать завершения события и сразу же
сбросить его. События чтения и записи хранятся в массиве событий рядом друг с
другом. */
     iWaits =0; /* Количество выполненных к данному моменту операций ввода/вывода.
*/
     while (iWaits < 2 * nRecord) {
```

```
      ic = WaitForMultipleObjects(2 * MAX_OVRLP, hEvents[0], FALSE, INFINITE) -
WAIT_OBJECT_0;
     iWaits++; /* Инкрементировать счетчик выполненных операций ввода вывода.*/
     ResetEvent(hEvents[ic / MAX_OVRLP][ic % MAX_OVRLP]);
     if (ic < MAX_OVRLP) {
      /* Чтение завершено. */
      GetOverlappedResult(hInputFile, &OverLapIn[ic], &nin[ic], FALSE);
      /* Обработать запись и инициировать операцию записи. */
      CurPosIn.LowPart = OverLapIn[ic].Offset;
      CurPosIn.HighPart = OverLapIn[ic].OffsetHigh;
      CurPosOut.QuadPart = (CurPosIn.QuadPart / REC_SIZE) * UREC_SIZE;
      OverLapOut[ic].Offset = CurPosOut.LowPart;
      OverLapOut[ic].OffsetHigh = CurPosOut.HighPart;
      /* Преобразовать запись из ASCII в Unicode. */
      for (i =0; i < REC_SIZE; i++) UnRec[ic][i] = AsRec[ic][i];
      WriteFile(hOutputFile, UnRec[ic], nin[ic] * 2, &nout[ic], &OverLapOut[ic]);
      /* Подготовиться к очередному чтению, которое будет инициировано после того,
как завершится начатая выше операция записи. */
      CurPosIn.QuadPart += REC_SIZE * (LONGLONG)(MAX_OVRLP);
      OverLapIn[ic].Offset = CurPosIn.LowPart;
      OverLapIn[ic].OffsetHigh = CurPosIn.HighPart;
     } else if (ic < 2 * MAX_OVRLP) { /* Операция записи завершилась. */
      /* Начать чтение. */
      ic -= MAX_OVRLP; /* Установить индекс выходного буфера. */
       if (!GetOverlappedResult (hOutputFile, &OverLapOut[ic], &nout[ic], FALSE))
ReportError(_T("Ошибка чтения."), 0, TRUE);
      CurPosIn.LowPart = OverLapIn[ic].Offset;
      CurPosIn.HighPart = OverLapIn[ic].OffsetHigh;
      if (CurPosIn.QuadPart < FileSize.QuadPart) {
       /* Начать новую операцию чтения. */
       ReadFile(hInputFile, AsRec[ic], REC_SIZE, &nin[ic], &OverLapIn[ic]);
      }
     }
    }
    /* Закрыть все события. */
    for (ic = 0; ic < MAX_OVRLP; ic++) {
     CloseHandle(hEvents[0][ic]);
     CloseHandle(hEvents[1][ic]);
    }
    CloseHandle(hInputFile);
    CloseHandle(hOutputFile);
    return 0;
   }
```

Программа 14.1 способна работать только под управлением Windows NT. Средства асинхронного ввода/вывода Windows 9x не позволяют использовать дисковые файлы. В приложении В приведены результаты и комментарии, свидетельствующие о сравнительно низкой производительности программы atouOV. Как показали эксперименты, для достижения приемлемой производительности размер буфера должен составлять, по крайней мере, 32 Кбайт, но даже и в этом случае обычный синхронный ввод/вывод работает быстрее. К тому же, производительность этой программы не повышается и в условиях SMP, поскольку в данном примере, в котором обрабатываются всего лишь два файла, ЦП не является критическим ресурсом.

Расширенный ввод/вывод с использованием процедуры завершения

Существует также другой возможный подход к использованию объектов синхронизации. Вместо того чтобы заставлять поток ожидать поступления сигнала завершения от события или дескриптора, система может инициировать вызов определенной пользователем процедуры завершения сразу же по окончании выполнения операции ввода/вывода. Далее процедура завершения может запустить очередную операцию ввода/вывода и выполнить любые необходимые действия по учету использования системных ресурсов. Эта косвенно вызываемая (callback) процедура завершения аналогична асинхронному вызову процедуры, который применялся в главе 10, и требует использования состояний дежурного ожидания (alertable wait states).

Каким образом процедура завершения может быть указана в программе? Среди параметров или структур данных функций ReadFile и WriteFile не остается таких, которые можно было бы использовать для хранения адреса процедуры завершения. Однако существует семейство расширенных функций ввода/вывода, которые обозначаются суффиксом "Ex" и содержат дополнительный параметр, предназначенный для передачи адреса процедуры завершения. Функциями чтения и записи являются, соответственно, ReadFileEx и WriteFileEx. Кроме того, требуется использование одной из указанных ниже функций дежурного ожидания.

- `WaitForSingleObjectEx`
- `WaitForMultipleObjectsEx`
- `SleepEx`
- `SignalObjectAndWait`
- `MsgWaitForMultipleObjectsEx`

Расширенный ввод/вывод иногда называют *дежурным вводом/выводом* (alertable I/O). О том, как использовать расширенные функции, рассказывается в последующих разделах.

> **Примечание**
>
> Под управлением Windows 9x расширенный ввод/вывод не может работать с дисковыми файлами и коммуникационными портами. В то же время, средства расширенного ввода/вывода Windows 9x способны работать с именованными каналами, почтовыми ящиками, сокетами и последовательными устройствами.

Функции ReadFileEx, WriteFileEx и процедурызавершения

Расширенные функции чтения и записи могут использоваться совместно с дескрипторами открытых файлов, именованных каналов и почтовых ящиков, если соответствующий объект открывался (создавался) с установленным флагом FILE_FLAG_OVERLAPPED. Заметьте, что этот флаг устанавливает атрибут дескриптора, и хотя перекрывающийся и расширенный ввод/вывод отличаются друг от друга, к дескрипторам обоих типов асинхронного ввода/вывода применяется один и тот же флаг.

Перекрывающиеся сокеты (глава 12) могут использоваться совместно с функциями ReadFileEx и WriteFileEx во всех версиях Windows.

```
    BOOL    ReadFileEx(HANDLE    hFile,    LPVOID    lpBuffer,    DWORD
nNumberOfBytesToRead,              LPOVERLAPPED              lpOverlapped,
LPOVERLAPPED_COMPLETION_ROUTINE lpcr)
    BOOL    WriteFileEx(HANDLE    hFile,    LPVOID    lpBuffer,    DWORD
```

```
nNumberOfBytesToWrite, LPOVERLAPPED lpOverlapped,
LPOVERLAPPED_COMPLETION_ROUTINE lpcr)
```

С обеими функциями вы уже знакомы, если не считать того, что каждая из них имеет дополнительный параметр, позволяющий указать адрес процедуры завершения.

Каждой из функций необходимо предоставлять структуру OVERLAPPED, но надобность в указании элемента hEvent этой структуры отсутствует; система игнорирует его. Вместе с тем, этот элемент оказывается очень полезным для передачи такой, например, информации, как порядковый номер, используемый для различения отдельных операций ввода/вывода, что демонстрируется в программе 14.2.

Сравнивая с функциями ReadFile и WriteFile, можно заметить, что расширенные функции не требуют параметров для хранения количества переданных байтов. Эта информация передается функции завершения, которая должна включаться в программу.

В функции завершения предусмотрены параметры для счетчика байтов, кода ошибки и адреса структуры OVERLAPPED. Последний из названных параметров требуется для того, чтобы процедура завершения могла определить, какая именно из невыполненных операций завершилась. Заметьте, что ранее высказанные предостережения относительно повторного использования или уничтожения структур OVERLAPPED справедливы здесь в той же мере, что и в случае перекрывающегося ввода/вывода.

```
VOID WINAPI FileIOCompletionRoutine(DWORD dwError, DWORD cbTransferred,
LPOVERLAPPED lpo)
```

Как и в случае функции CreateThread, при вызове которой также указывается имя некоторой функции, имя *FileIOCompletionRoutine* является заменителем, а не фактическим именем процедуры завершения.

Значения параметра dwError ограничены 0 (успешное завершение) и ERROR_HANDLE_EOF (при попытке выполнить чтение с выходом за пределы файла). Структура OVERLAPPED — это та структура, которая использовалась завершившимся вызовом ReadFileEx или WriteFileEx.

Прежде чем процедура завершения будет вызвана системой, должны произойти две вещи:

1. Должна завершиться операция ввода/вывода.

2. Вызывающий поток должен находиться в состоянии дежурного ожидания, извещая систему о том, что требуется выполнить процедуру завершения, находящуюся в очереди.

Каким образом поток переходит в состояние дежурного ожидания? Он должен выполнить явный вызов одной из функций дежурного ожидания, описанных в следующем разделе. Тем самым поток создает условия, делающие преждевременное выполнение процедуры завершения невозможным. В состоянии дежурного ожидания поток может находиться только на протяжении того времени, пока длится вызов функции дежурного ожидания; после возврата из этой функции поток выходит из указанного состояния.

Если оба эти условия удовлетворены, выполняются процедуры завершения, помещенные в очередь в результате завершения операций ввода/вывода. *Процедуры завершения выполняются в том же потоке, который выполнил первоначальный вызов функции ввода/вывода и находится в состоянии дежурного ожидания.* Поэтому поток должен переходить в состояние дежурного ожидания только тогда, когда для выполнения процедур завершения существуют безопасные условия.

Функции дежурного ожидания

Всего предусмотрено пять функций дежурного ожидания, но ниже приводятся прототипы только трех из них, которые представляют для нас непосредственный интерес:

```
    DWORD WaitForSingleObjectEx(HANDLE hObject, DWORD dwMilliseconds, BOOL
bAlertable)
    DWORD WaitForMultipleObjectsEx(DWORD cObjects, LPHANDLE lphObjects,
BOOL fWaitAll, DWORD dwMilliseconds, BOOL bAlertable)
    DWORD SleepEx(DWORD dwMilliseconds, BOOL bAlertable)
```

В каждой из функций дежурного ожидания имеется флаг bAlertable, который в случае асинхронного ввода/вывода должен устанавливаться в TRUE. Приведенные выше функции являются расширением знакомых вам функций Wait и Sleep.

Длительность интервалов ожидания указывается, как обычно, в миллисекундах. Каждая из этих трех функций осуществляет возврат, как только наступает *любая* из перечисленных ниже ситуаций:

• Дескриптор (дескрипторы) переходит (переходят) в сигнальное состояние, чем удовлетворяются стандартные требования двух из функций ожидания.

• Истекает интервал ожидания.

• Все процедуры завершения, находящиеся в очереди потока, прекращают свое выполнение, а значение параметра bAlertable равно TRUE. Процедура завершения помещается в очередь тогда, когда завершается соответствующая ей операция ввода/вывода (рис. 14.2).

Заметьте, что со структурами OVERLAPPED в функциях ReadFileEx и WriteFileEx не связаны никакие события, поэтому ни один из дескрипторов, указываемых при вызове функции ожидания, не связывается непосредственно с какой-либо определенной операцией ввода/вывода. В то же время, функция SleepEx не связана с объектами синхронизации, и поэтому ее проще всего использовать. В случае функции SleepEx в качестве длительности интервала ожидания обычно указывают значение INFINITE, поэтому возврат из этой функции произойдет только после того, как закончится выполнение одной или нескольких процедур завершения, которые в настоящий момент находятся в очереди.

Выполнение процедуры завершения и возврат из функции дежурного ожидания

По окончании выполнения операции расширенного ввода/вывода связанная с ней процедура завершения со своими аргументами, определяющими структуру OVERLAPPED, счетчик байтов и код ошибки, помещается в очередь для выполнения.

Все процедуры завершения, находящиеся в очереди потока, начинают выполняться тогда, когда поток переходит в состояние дежурного ожидания. Они выполняются поочередно, но не обязательно в той же последовательности, в которой завершились операции ввода/вывода. Возврат из функции дежурного ожидания происходит только после того, как осуществят возврат процедуры завершения. Эту особенность важно учитывать для обеспечения правильного функционирования большинства программ, поскольку при этом предполагается, что процедуры завершения получают возможность подготовиться к очередному использованию структуры OVERLAPPED и выполнить другие необходимые действия для перевода программы в известное состояние, прежде чем будет осуществлен возврат из состояния дежурного ожидания.

Если возврат из функции SleepEx обусловлен выполнением одной или нескольких процедур завершения, находящихся в очереди, то возвращаемым значением функции будет WAIT_TO_COMPLETION, и это же значение будет возвращено функцией GetLastError,

вызванной после выполнения возврата одной из функций ожидания.

В заключение отметим два момента:

1. При вызове любой из функций дежурного ожидания в качестве значения параметра интервала ожидания используйте INFINITE. В отсутствие возможности истечения интервала ожидания возврат из функций будет осуществляться лишь после того, как закончится выполнение всех процедур завершения или дескрипторы перейдут в сигнальное состояние.

2. Для передачи информации процедуре завершения общепринято использовать элемент данных hEvent структуры OVERLAPPED, поскольку это поле игнорируется ОС.

Взаимодействие между основным потоком, процедурами завершения и функциями дежурного ожидания иллюстрирует рис. 14.2. В этом примере запускаются три параллельные операции чтения, две из которых завершаются к тому моменту, когда начинается выполнение дежурного ожидания.

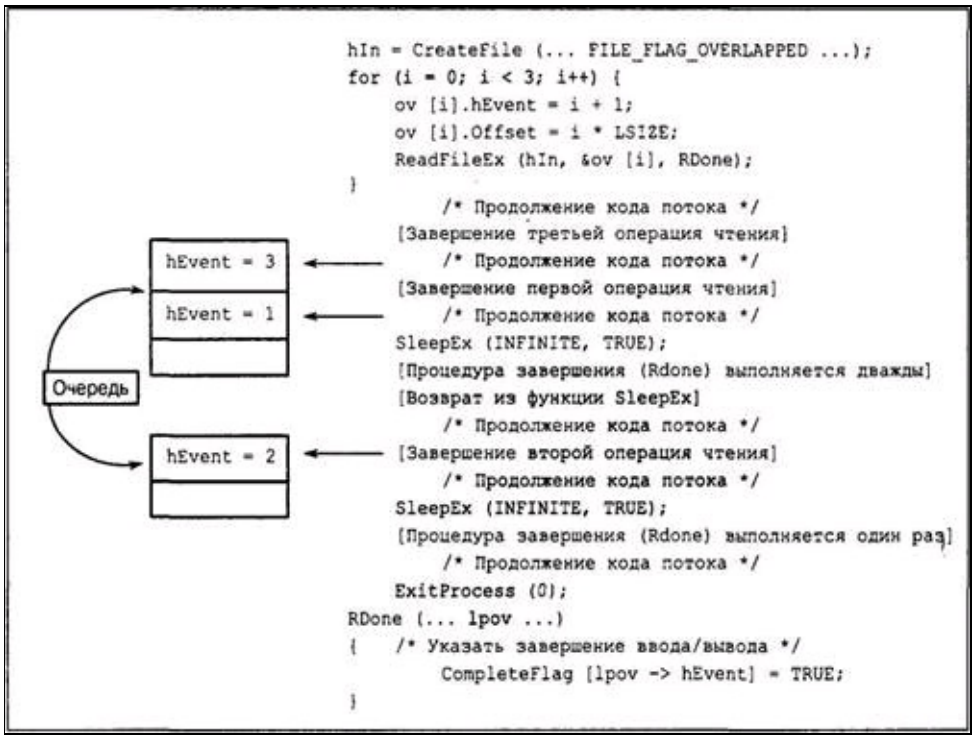


Рис. 14.2. Асинхронный ввод/вывод с использованием процедур завершения

Пример: преобразование файла с использованием расширенного ввода/вывода

Программа 14.3 (atouEX) представляет собой переработанную версию программы 14.1. Эти программы иллюстрируют различие между двумя методами асинхронного ввода/вывода. Программа atouEx аналогична программе 14.1, но большая часть кода, предназначенного для упорядочения ресурсов, перемещена в ней в процедуру завершения, а многие переменные сделаны глобальными, чтобы процедура завершения могла иметь к ним доступ. Вместе с тем, в приложении В показано, что в отношении быстродействия программа atouEx вполне может конкурировать с другими методами, в которых не используется отображение файлов, тогда как программа atouOV работает медленнее.

Программа 14.2. atouEx: преобразование файла с использованием расширенного ввода/вывода

```
    /* Глава 14. atouEX
           Преобразование файла из ASCII в Unicode средствами РАСШИРЕННОГО
ВВОДА/ВЫВОДА. */
    /* atouEX файл1 файл2 */

    #include "EvryThng.h"
    #define MAX_OVRLP 4
    #define REC_SIZE 8096 /* Размер блока не имеет столь важного значения в
отношении производительности, как в случае atouOV. */
    #define UREC_SIZE 2 * REC_SIZE

    static VOID WINAPI ReadDone(DWORD, DWORD, LPOVERLAPPED);
    static VOID WINAPI WriteDone(DWORD, DWORD, LPOVERLAPPED);

    /* Первая структура OVERLAPPED предназначена для чтения, а вторая — для записи.
Структуры и буферы распределяются для каждой предстоящей операции. */
    OVERLAPPED OverLapIn[MAX_OVRLP], OverLapOut [MAX_OVRLP];
    CHAR AsRec[MAX_OVRLP][REC_SIZE];
    WCHAR UnRec[MAX_OVRLP][REC_SIZE];
    HANDLE hInputFile, hOutputFile;
    LONGLONG nRecord, nDone;
    LARGE_INTEGER FileSize;

    int _tmain(int argc, LPTSTR argv[]) {
     DWORD ic;
     LARGE_INTEGER CurPosIn;
      hInputFile = CreateFile(argv[1], GENERIC_READ, 0, NULL, OPEN_EXISTING,
FILE_FLAG_OVERLAPPED, NULL);
      hOutputFile = CreateFile(argv[2], GENERIC_WRITE, 0, NULL, CREATE_ALWAYS,
FILE_FLAG_OVERLAPPED, NULL);
     FileSize.LowPart = GetFileSize(hInputFile, &FileSize.HighPart);
     nRecord = FileSize.QuadPart / REC_SIZE;
     if ((FileSize.QuadPart % REC_SIZE) != 0) nRecord++;
     CurPosIn.QuadPart = 0;
     for (ic = 0; ic < MAX_OVRLP; ic++) {
      OverLapIn[ic].hEvent = (HANDLE)ic; /* Перегрузить событие. */
      OverLapOut[ic].hEvent = (HANDLE)ic; /* Поля. */
      OverLapIn[ic].Offset = CurPosIn.LowPart;
```

```
      OverLapIn[ic].OffsetHigh = CurPosIn.HighPart;
       if (CurPosIn.QuadPart < FileSize.QuadPart) ReadFileEx(hInputFile, AsRec[ic],
REC_SIZE, &OverLapIn [ic], ReadDone);
      CurPosIn.QuadPart += (LONGLONG)REC_SIZE;
     }
     /* Выполняются все операции чтения. Войти в состояние дежурного ожидания и
оставаться в нем до тех пор, пока не будут обработаны все записи.*/
     nDone = 0;
     while (nDone < 2 * nRecord) SleepEx(INFINITE, TRUE);
     CloseHandle(hInputFile);
     CloseHandle(hOutputFile);
     _tprintf(_T("Преобразование из ASCII в Unicode завершено.\n"));
     return 0;
    }

    static VOID WINAPI ReadDone(DWORD Code, DWORD nBytes, LPOVERLAPPED pOv) {
     /* Чтение завершено. Преобразовать данные и инициировать запись. */
     LARGE_INTEGER CurPosIn, CurPosOut;
     DWORD ic, i;
     nDone++;
     /* Обработать запись и инициировать операцию записи. */
     ic = (DWORD)(pOv->hEvent);
     CurPosIn.LowPart = OverLapIn[ic].Offset;
     CurPosIn.HighPart = OverLapIn[ic].OffsetHigh;
     CurPosOut.QuadPart = (CurPosIn.QuadPart / REC_SIZE) * UREC_SIZE;
     OverLapOut[ic].Offset = CurPosOut.LowPart;
     OverLapOut[ic].OffsetHigh = CurPosOut.HighPart;
     /* Преобразовать запись из ASCII в Unicode. */
     for (i = 0; i < nBytes; i++) UnRec[ic][i] = AsRec[ic][i];
     WriteFileEx(hOutputFile, UnRec[ic], nBytes*2, &OverLapOut[ic], WriteDone);
     /* Подготовить структуру OVERLAPPED для следующего чтения. */
     CurPosIn.QuadPart += REC_SIZE * (LONGLONG)(MAX_OVRLP);
     OverLapIn[ic].Offset = CurPosIn.LowPart;
     OverLapIn[ic].OffsetHigh = CurPosIn.HighPart;
     return;
    }

    static VOID WINAPI WriteDone(DWORD Code, DWORD nBytes, LPOVERLAPPED pOv) {
     /* Запись завершена. Инициировать следующую операцию чтения. */
     LARGE_INTECER CurPosIn;
     DWORD ic;
     nDone++;
     ic = (DWORD)(pOv->hEvent);
     CurPosIn.LowPart = OverLapIn[ic].Offset;
     CurPosIn.HighPart = OverLapIn[ic].OffsetHigh;
     if (CurPosIn.QuadPart < FileSize.QuadPart) {
      ReadFileEx(hInputFile, AsRec[ic], REC_SIZE, &OverLapIn[ic], ReadDone);
     }
     return;
    }
```

Асинхронный ввод/вывод сиспользованием нескольких потоков

Перекрывающийся и расширенный ввод/вывод позволяют добиться асинхронного выполнения операций ввода/вывода в пределах единственного потока, хотя для поддержки этой функциональности ОС создает собственные потоки. В том или ином виде методы этого типа часто используются во многих ранних ОС для поддержки ограниченных форм выполнения асинхронных операций в однопоточных системах.

Однако Windows обеспечивает многопоточную поддержку, поэтому становится возможным достижение того же эффекта за счет выполнения синхронных операций ввода/вывода в нескольких, выполняемых независимо потоках. Ранее эти возможности уже были продемонстрированы на примере многопоточных серверов и программы grepMT (глава 7). Кроме того, потоки обеспечивают концептуально последовательный и, предположительно, гораздо более простой способ выполнения асинхронных операций ввода/вывода. В качестве альтернативы методам, используемым в программах 14.1 и 14.2, можно было бы предоставить каждому потоку собственный дескриптор файла, и тогда каждый из потоков мог бы обрабатывать в синхронном режиме каждую четвертую запись.

Такой способ использования потоков продемонстрирован в программе atouMT, которая в книге не приводится, но включена в материал, размещенный на Web-сайте. Программа atouMT не только способна выполняться под управлением любой версии Windows, но и более проста по сравнению с любым из двух вариантов программ асинхронного ввода/вывода, поскольку учет использования ресурсов в этом случае менее сложен. Каждый поток просто поддерживает собственные буферы в собственном стеке и выполняет в цикле последовательность синхронных операций чтения, преобразования и записи. При этом производительность программы остается на достаточно высоком уровне.

> **Примечание**
>
> В программе atouMT.c, которая находится на Web-сайте, содержатся комментарии по поводу нескольких возможных "ловушек", которые могут поджидать вас при организации доступа одновременно нескольких потоков к одному и тому же файлу. В частности, все отдельные дескрипторы файлов должны создаваться с помощью функции CreateHandle, а не функции DuplicateHandle.

Лично я предпочитаю использовать многопоточную обработку файлов, а не асинхронные операции ввода/вывода. Потоки легче программировать, и в большинстве случаев они обеспечивают более высокую производительность.

Существуют два исключения из этого общего правила. Первое из них, как было показано ранее в этой главе, касается ситуаций, в которых может быть только одна невыполненная операция, и в целях синхронизации можно использовать дескриптор файла. Второе, более важное исключение встречается в случае портов завершения асинхронного ввода/вывода, о чем будет говориться в конце настоящей главы.

Таймеры ожидания

Windows NT поддерживает таймеры ожидания (waitable timers), являющихся одним из типов объектов ядра, осуществляющих ожидание.

Вы всегда можете создать собственный сигнал синхронизации, создав синхронизирующий поток, который устанавливает событие в результате пробуждения после вызова функции Sleep. В программе serverNP (программа 11.3) сервер также использует синхронизирующий поток для периодической широковещательной рассылки имени своего канала. Поэтому таймеры ожидания обеспечивают хотя и несколько избыточный, но удобный способ организации выполнения задач на периодической основе или в соответствии с определенным расписанием. В частности, таймер ожидания можно настроить таким образом, чтобы сигнал был сгенерирован в строго определенное время.

Таймер ожидания может быть либо синхронизирующим (synchronization timer), либо сбрасываемым вручную уведомляющим (manual-reset notification timer) таймером. Синхронизирующий таймер связывается с функцией косвенного вызова, аналогичной процедуре завершения расширенного ввода/вывода, тогда как для синхронизации по сбрасываемому вручную уведомляющему таймеру используется функция ожидания.

Для начала потребуется создать дескриптор таймера, используя для этого функцию CreateWaitableTimer.

```
    HANDLE   CreateWaitableTimer(LPSECURITY_ATTRIBUTES   lpTimerAttributes,
BOOL bManualReset, LPCTSTR lpTimerName);
```

Второй параметр, bManualReset, определяет, таймер какого типа должен быть создан — синхронизирующий или уведомляющий. В программе 14.3 используется синхронизирующий таймер, но, изменив комментарии и настройку параметра, вы легко превратите его в уведомляющий таймер. Заметьте, что существует также функция OpenWaitableTimer, которая может использовать необязательное имя, предоставляемое третьим аргументом.

Первоначально таймер создается в неактивном состоянии, но с помощью функции SetWaitableTimer его можно активизировать и указать начальную временную задержку, а также длительность промежутка времени между периодически вырабатываемыми сигналами.

```
    BOOL  SetWaitableTimer(HANDLE  hTimer,  const  LARGE_INTEGER  *pDueTime,
LONG     lPeriod,     PTIMERAPCROUTINE     pfnCompletionRoutine,     LPVOID
lpArgToCompletionRoutine, BOOL fResume);
```

hTimer — действительный дескриптор таймера, созданного с использованием функции CreateWaitableTimer.

Второй параметр, на который указывает указатель pDueTime, может принимать либо положительные значения, соответствующие абсолютному времени, либо отрицательные, соответствующие относительному времени, причем фактические значения выражаются в единицах времени длительностью 100 наносекунд, а их формат описывается структурой FILETIME. Переменные типа FILETIME были введены в главе 3 и уже использовались нами в главе 6 в программе timer (программа 6.2).

Величина интервала между сигналами, указываемая в третьем параметре, выражается в миллисекундах. Если это значение установлено равным 0, то таймер переводится в сигнальное состояние только один раз. При положительных значениях этого параметра таймер является периодическим и срабатывает периодически до тех пор, пока его действие не будет прекращено

вызовом функции CancelWaitableTimer. Отрицательные значения указанного интервала не допускаются.

Четвертый параметр, pfnCompletionRoutine, применяется в случае синхронизирующего таймера и указывает адрес процедуры завершения, которая вызывается при переходе таймера в сигнальное состояние *и при условии,* что поток переходит в состояние дежурного ожидания. При вызове этой процедуры в качестве одного из аргументов используется указатель, определяемый пятым параметром, plArgToComplretionRoutine.

Установив синхронизирующий таймер, вы можете перевести поток в состояние дежурного ожидания путем вызова функции SleepEx, чтобы обеспечить возможность вызова процедуры завершения. В случае сбрасываемого вручную уведомляющего таймера следует организовать ожидание перехода дескриптора таймера в сигнальное состояние. Дескриптор будет оставаться в сигнальном состоянии до следующего вызова функции SetWaitableTimer. Полная версия программы 14.3, находящаяся на Web-сайте, предоставляет вам возможность проводить собственные эксперименты, используя таймер выбранного типа в сочетании с процедурой завершения или ожиданием перехода дескриптора таймера в сигнальное состояние, что в итоге дает четыре различные комбинации.

Последний параметр, fResume, связан с режимами энергосбережения. Для получения более подробной информации по этому вопросу обратитесь к справочной документации.

Функция CancelWaitableTimer используется для отмены действия вызванной перед этим функции SetWaitableTimer, но при этом не изменяет сигнальное состояние таймера. Чтобы это сделать, необходимо в очередной раз вызвать функцию SetWaitableTimer.

Пример: использование таймера ожидания

В программе 14.3 демонстрируется применение таймера ожидания для генерации периодических сигналов.

Программа 14.3. TimeBeep: генерация периодических сигналов

```
    /* Глава 14. TimeBeep.c. Периодическое звуковое оповещение.  */
    /* Использование: TimeBeep период (в миллисекундах). */

    #include "EvryThng.h"
    static BOOL WINAPI Handler(DWORD CntrlEvent);
    static VOID APIENTRY Beeper(LPVOID, DWORD, DWORD);
    volatile static BOOL Exit = FALSE;
    HANDLE hTimer;

    int _tmain(int argc, LPTSTR argv[]) {
     DWORD Count = 0, Period;
     LARGE_INTEGER DueTime;
      /* Перехват нажатия комбинации клавиш <Ctrl-c> для прекращения операции. См. главу 4. */
     SetConsoleCtrlHandler(Handler, TRUE);
     Period = _ttoi(argv[1]) * 1000;
     DueTime.QuadPart = -(LONGLONG)Period * 10000;
      /* Параметр DueTime отрицателен для первого периода ожидания и задается относительно текущего времени. Период ожидания измеряется в мс (10^-3 с), а DueTime — в единицах по 100 нс (10^-7 с) для согласования с типом FILETIME. */
     hTimer = CreateWaitableTimer(NULL, FALSE /* "Таймер синхронизации" */, NULL);
     SetWaitableTimer(hTimer, &DueTime, Period, Beeper, &Count, TRUE);
     while (!Exit) {
      _tprintf(_T("Count = %d\n"), Count);
      /* Значение счетчика увеличивается в процедуре таймера. */
      /* Войти в состояние дежурного ожидания. */
      SleepEx(INFINITE, TRUE);
     }
     _tprintf(_T("Завершение. Счетчик = %d"), Count);
     CancelWaitableTimer(hTimer);
     CloseHandle(hTimer);
     return 0;
    }

    static VOID APIENTRY Beeper(LPVOID lpCount, DWORD dwTimerLowValue, DWORD dwTimerHighValue) {
     *(LPDWORD)lpCount = *(LPDWORD)lpCount + 1;
     _tprintf(_T("Генерация сигнала номер: %d\n"), *(LPDWORD) lpCount);
     Beep(1000 /* Частота. */, 250 /* Длительность (мс). */);
     return;
    }

    BOOL WINAPI Handler(DWORD CntrlEvent) {
     Exit = TRUE;
     _tprintf(_T("Завершение работы\n"));
     return TRUE;
    }
```

Комментарии к примеру с таймером ожидания

Исходя из типа таймера и используя либо процедуру завершения, либо ожидание перехода дескриптора в сигнальное состояние, можно образовать четыре различных комбинации. Программа 14.3 иллюстрирует использование процедуры завершения и синхронизирующего таймера. Вы сможете тестировать каждую из четырех возможных комбинаций, изменяя комментарии в версии программы TimeBeep.c, доступной на Web-сайте.

Порты завершения ввода/вывода

Порты завершения ввода/вывода, поддерживаемые лишь на NT-платформах, объединяют в себе возможности перекрывающегося ввода/вывода и независимых потоков и используются чаще всего в серверных программах. Чтобы выяснить, какими требованиями это может диктоваться, обратимся к серверам, построенным в главах 11 и 12, где каждый клиент поддерживался отдельным рабочим потоком, связанным с сокетом или экземпляром именованного канала. Это решение хорошо работает лишь в тех случаях, когда число клиентов невелико.

Посмотрим, однако, что произойдет, если число клиентов достигнет 1000. В имеющейся модели для этого потребуется 1000 потоков, для каждого из которых необходимо выделить значительный объем виртуальной памяти. Так, по умолчанию каждому потоку выделяется 1 Мбайт стекового пространства, так что для 1000 потоков потребуется 1 Гбайт, и переключение контекстов потоков может увеличить задержки, обусловленные ошибками из-за отсутствия страниц.[35] Кроме того, потоки будут состязаться между собой за право владения общими ресурсами как на уровне планировщика, так и внутри процесса, и это, как было показано в главе 9, может приводить к снижению производительности. В связи с этим требуется механизм, позволяющий небольшому пулу рабочих потоков обслуживать большое количество клиентов.

Искомое решение обеспечивается портами завершения ввода/вывода, которые предоставляют возможность создавать ограниченное количество серверных потоков в пуле потоков, имея очень большое количество дескрипторов именованных каналов (или сокетов). При этом дескрипторы не соединяются попарно с отдельными рабочими серверными потоками; серверный поток может обслуживать любой дескриптор, данные которого нуждаются в обработке.

Итак, порт завершения ввода/вывода — это набор перекрывающихся дескрипторов, и потоки ожидают перехода порта в сигнальное состояние. Когда завершается операция чтения или записи с участием какого-либо дескриптора, один из потоков пробуждается и принимает данные и результаты выполнения операции ввода/вывода. Далее поток может обработать данные и вновь перейти в состояние ожидания перехода порта в сигнальное состояние.

Прежде всего необходимо создать порт завершения ввода/вывода и присоединить к нему перекрывающиеся дескрипторы.

Управление портами завершения ввода/вывода

Для создания порта и присоединения к нему дескрипторов используется одна и та же функция — CreateCompletionPort. Необходимость выполнения этой функцией двух разных задач соответственно усложняет использование ее параметров.

```
HANDLE CreateIoCompletionPort(HANDLE FileHandle, HANDLE
ExistingCompletionPort, DWORD CompletionKey, DWORD
NumberOfConcurrentThreads);
```

Порт завершения ввода/вывода представляет собой совокупность дескрипторов файлов, открытых в режиме OVERLAPPED. Параметр FileHandle — это перекрывающийся дескриптор, присоединяемый к порту. Если задать его значение равным INVALID_DESCRIPTOR_HANDLE, то функция создаст новый порт завершения ввода/вывода и возвратит его дескриптор. В этом случае следующий параметр, ExistingCompletionPort, должен быть установлен в NULL.

ExistingCompletionPort — порт, созданный при первом вызове функции, к которому должен быть присоединен дескриптор, указанный в первом параметре. В случае успешного выполнения функция возвращает дескриптор порта, иначе — NULL.

CompletionKey — указывает ключ, который будет включен в пакет завершения для дескриптора FileHandle. Обычно в качестве ключа используется значение индекса массива структур данных, содержащих тип операции, дескриптор и указатель на буфер данных.

NumberOfConcurrentThreads — предельно допустимое количество потоков, которым разрешено параллельное выполнение. При наличии других потоков сверх этого количества, ожидающих перехода порта в сигнальное состояние, они будут оставаться блокированными, даже если существует дескриптор с доступными данными. Если этот параметр установлен равным 0, то в качестве предела используется количество процессоров, установленных в системе.

Количество перекрывающихся дескрипторов, которые могут быть связаны с одним портом завершения ввода/вывода, ничем не ограничивается. Первоначальный вызов функции CreateCompletionPort используется для создания порта и указания максимального количества потоков. Для каждого очередного перекрывающегося дескриптора, подлежащего связыванию с данным портом, следует повторно вызывать эту же функцию. К сожалению, способов, позволяющих удалить дескриптор из порта завершения, не существует, и это упущение значительно ограничивает гибкость программ.

Дескрипторы, связанные с портом не должны использоваться совместно с функциями ReadFileEx и WriteFileEx. В документации Microsoft не рекомендуется разделять файлы и объекты иного типа, используя другие открытые дескрипторы.

Ожидание порта завершения ввода/вывода

Для выполнения ввода/вывода с участием дескрипторов, связанных с портом, используются функции ReadFile и WriteFile со структурами OVERLAPPED (дескрипторы событий не требуются). Далее операция ввода/вывода помещается в очередь порта завершения.

Поток ожидает завершения перекрывающейся операции ввода/вывода, находящейся в очереди, не путем ожидания события, а путем вызова функции GetQueueCompletionStatus с указанием порта завершения (completion port). Когда вызывающий поток пробуждается, функция возвращает ключ, который был указан при первоначальном присоединении к порту дескриптора, чья операция завершилась, и этот ключ может указывать количество переданных байтов и идентификационные данные фактического дескриптора, связанного с завершившейся операцией.

Следует отметить, что уведомление о завершении операции получит не обязательно тот же поток, который инициировал чтение или запись. Уведомление о завершении операции может быть получено любым ожидающим потоком. Поэтому необходимо, чтобы ключ мог идентифицировать дескриптор, связанный с завершившейся операцией.

Имеется также возможность использовать конечный интервал ожидания (time-out).

```
BOOL GetQueuedCompletionStatus(HANDLE CompletionPort, LPDWORD
lpNumberOfBytesTransferred, LPDWORD lpCompletionKey, LPOVERLAPPED
*lpOverlapped, DWORD dwMilliseconds);
```

Иногда может оказаться удобным, чтобы операция не помещалась в очередь порта завершения ввода/вывода. В этом случае поток может ожидать наступления перекрывающегося

события, как показано в программе 14.4 и дополнительном примере, atouMTCP, который находится на Web-сайте. Для указания того, что перекрывающаяся операция *н е* должна помещаться в очередь порта завершения, вы должны установить младший бит дескриптора события (hEvent) в структуре OVERLAPPED; тогда вы получите возможность ожидать наступления события для данной конкретной операции. Такое решение является довольно странным, однако оно документировано, хотя особо и не подчеркивается.

Отправка уведомления порту завершения ввода/вывода

Поток может отправить в порт событие завершения вместе с ключом, чтобы завершить остающийся невыполненным вызов функции GetQueueCompletionStatus. Вся необходимая для этого информация предоставляется функцией PostQueueCompletionStatus.

```
BOOL PostQueuedCompletionStatus(HANDLE CompletionPort, DWORD
dwNumberOfBytesTransferred, DWORD dwCompletionKey, LPOVERLAPPED
lpOverlapped);
```

Для пробуждения ожидающих потоков даже в условиях отсутствия завершившихся операций иногда используют метод, суть которого заключается в предоставлении фиктивного значения ключа, например, –1. Ожидающие потоки должны проверять значения ключей, и эта методика может использоваться, например, для того, чтобы сигнализировать потоку о необходимости завершить работу.

Альтернативы портам завершенияввода/вывода

В главе 9 было показано, как использовать семафор для ограничения количества готовых к выполнению потоков, и этот же метод можно эффективно применять для регулирования пропускной способности в условиях, когда множество потоков соревнуются между собой за право владения ограниченными ресурсами.

Эту же методику мы могли бы применить и в серверах serverSK (программа 12.2) и serverNP (программа 11.3). Все, что для этого требуется — это организовать ожидание перехода семафора в сигнальное состояние после завершения запроса на чтение, выполнение этого запроса, создание ответа и освобождение семафора перед тем, как записать ответ. Такое решение гораздо проще того, которое реализовано в примере с портом завершения ввода/вывода, приведенном в следующем разделе. Единственная проблема состоит в том, что потоков может оказаться очень много, и для каждой из них требуется собственное стековое пространство, что приведет к большому расходу виртуальной памяти. Остроту этой проблемы можно несколько ослабить, тщательно распределяя необходимые объемы стекового пространства. Упражнение 14.6 включает в себя выполнение экспериментов с альтернативным решением подобного рода, а реализация соответствующего примера находится на Web-сайте.

Существует еще одна возможность, которую можно использовать при создании масштабируемых серверов. Выборка пакетов рабочих заготовок (work items) из очереди (см. главу 10) может осуществляться с использованием ограниченного количества потоков. Поступающие рабочие заготовки могут помещаться в очередь одной или несколькими главными потоками, как показано в программе 10.5.

Пример: сервер, использующий порты завершения ввода/вывода

Программа 14.4 представляет видоизмененный вариант программы serverNP (программа 11.3), в котором используются порты завершения ввода/вывода. Этот сервер создает небольшой пул серверных потоков и больший пул дескрипторов перекрывающихся каналов, а также ключей завершения, по одному для каждого дескриптора. Перекрывающиеся дескрипторы присоединяются к порту завершения, а затем вызывается функция ConnectNamedPipe. Серверные потоки ожидают сигналов завершения, связанных как с подключениями клиентов, так и с операциями чтения. Когда регистрируется операция чтения, обрабатывается соответствующий клиентский запрос, и результаты возвращаются без использования порта завершения. Вместо этого серверный поток ожидает наступления события после выполнения операции записи, причем младший бит дескриптора события в структуре OVERLAPPED устанавливается в 1.

В другом возможном варианте решения, отличающемся большей гибкостью, можно было бы закрывать дескриптор при каждом отсоединении клиента и создавать новый дескриптор для каждого нового подключения. Этот способ аналогичен тому, который использовался в случае сокетов в главе 12. Вместе с тем, имеется одна трудность, обусловленная невозможностью удаления дескрипторов из порта завершения, в результате чего использование короткоживущих дескрипторов подобного рода будет приводить к утечке ресурсов.

Поскольку с большей частью кода вы уже знакомы по предыдущим примерам, она здесь не приводится.

Программа 14.4. serverCP: сервер, использующий порт завершения

```
/* Глава 14. ServerCP. Многопоточный сервер.
   Версия на основе именованного канала, пример ПОРТА ЗАВЕРШЕНИЯ.
   Использование: Server [ИмяПользователя ИмяГруппы]. */

#include "EvryThng.h"
#include "ClntSrvr.h"

/* Здесь определяются сообщения запроса и ответа. */
typedef struct { /*Структуры, на которые указывают ключи портов завершения*/
 HANDLE hNp; /* и которые представляют еще не выполненные операции */
 REQUEST Req; /* ReadFile и ConnectNamedPipe. */
 DWORD Type; /* 0 - ConnectNamedPipe; 1 - ReadFile. */
 OVERLAPPED Ov;
} CP_KEY;

static CP_KEY Key[MAX_CLIENTS_CP]; /* Доступно всем потокам. */
/* … */

_tmain(int argc, LPTSTR argv[]) {
 HANDLE hCp, hMonitor, hSrvrThread[MAXCLIENTS];
 DWORD iNp, iTh, MonitorId, ThreadId;
 THREAD_ARG ThArgs[MAX_SERVER_TH];
 /*…*/
 hCp = CreateIoCompletionPort(INVALID_HANDLE_VALUE, NULL, 0, MAX_SERVER_TH);
 /* Создать перекрывающийся именованный канал для каждого потенциального */
 /* клиента, добавить порт завершения и ожидать соединения. */
 /* Предполагается, что максимальное количество клиентов намного */
```

```
    /* превышает количество серверных потоков. */
    for (iNp = 0; iNp < MAX_CLIENTS_CP; iNp++) {
     memset(&Key[iNp], 0, sizeof(CP_KEY));
          Key[iNp].hNp   =   CreateNamedPipe(SERVER_PIPE,   PIPE_ACCESS_DUPLEX   |
FILE_FLAG_OVERLAPPED,   PIPE_READMODE_MESSAGE   |   PIPE_TYPE_MESSAGE   |   PIPE_WAIT,
MAX_CLIENTS_CP, 0, 0, INFINITE, pNPSA);
     CreateIoCompletionPort(Key[iNp].hNp, hCp, iNp, MAX_SERVER_TH + 2);
     Key[iNp].Ov.hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
     ConnectNamedPipe(Key[iNp].hNp, &Key[iNp].Ov);
    }
     /* Создать рабочие серверные потоки и имя временного файла для каждой из
них.*/
    for (iTh = 0; iTh < MAX_SERVER_TH; iTh++) {
     ThArgs[iTh].hCompPort = hCp;
     ThArgs[iTh].ThreadNo = iTh;
     GetTempFileName(_T("."), _T("CLP"), 0, ThArgs[iTh].TmpFileName);
      hSrvrThread[iTh] = (HANDLE)_beginthreadex (NULL, 0, Server, &ThArgs[iTh], 0,
&ThreadId);
    }
    /* Дождаться завершения всех потоков и "убрать мусор". */
    /* … */
    return 0;
   }


   static DWORD WINAPI Server(LPTHREAD_ARG pThArg)
   /* Функция потока сервера.
      Имеется по одному потоку для каждого потенциального клиента. */
   {
    HANDLE hCp, hTmpFile = INVALID_HANDLE_VALUE;
    HANDLE hWrEvent = CreateEvent(NULL, TRUE, FALSE, NULL);
    DWORD nXfer, KeyIndex, ServerNumber;
    /* … */
    BOOL Success, Disconnect, Exit = FALSE;
    LPOVERLAPPED pOv;
    OVERLAPPED ovResp = {0, 0, 0, 0, hWrEvent}; /*Для ответных сообщений.*/
     /*  Чтобы  избежать  помещения  перекрывающейся  операции  в  очередь  порта
завершения, должен быть установлен младший бит события. Несмотря на всю странность
этого способа, он документирован. */
    ovResp.hEvent = (HANDLE)((DWORD)hWrEvent | 0x1);
    GetStartupInfo(&StartInfoCh);
    hCp = pThArg->hCompPort;
    ServerNumber = pThArg->ThreadNo;
    while(!ShutDown && !Exit) __try {
     Success = FALSE; /* Устанавливается только в случае успешного завершения всех
операций. */
     Disconnect = FALSE;
     GetQueuedCompletionStatus(hCp, &nXfer, &KeyIndex, &pOv, INFINITE);
     if (Key [KeyIndex].Type == 0) { /* Соединение установлено. */
      /* Открыть временный файл с результатами для этого соединения. */
      hTmpFile = CreateFile(pThArg->TmpFileName, /* … */);
      Key[KeyIndex].Type = 1;
         Disconnect  =  !ReadFile(Key[KeyIndex].hNp,  &Key[KeyIndex].Req,  RQ_SIZE,
&nXfer,  &Key[KeyIndex].Ov)  &&  GetLastError  ()  ==  ERROR_HANDLE_EOF;  /*  Первая
операция чтения. */
      if (Disconnect) continue;
      Success = TRUE;
     } else {
      /* Чтение завершилось. Обработать запрос. */
      ShutDown = ShutDown || (_tcscmp (Key[KeyIndex].Req.Record, ShutRqst) == 0);
      if (ShutDown) continue;
```

```
       /* Создать процесс для выполнения команды. */
       /* … */
       /* Отвечать по одной строке за один раз. На данном этапе удобно использовать
функции библиотеки C для работы со строками. */
       fp = _tfopen(pThArg->TmpFileName, _T("r"));
       Response.Status = 0;
        /* Поскольку младший бит события установлен, ответные сообщения в очередь
порта завершения не помещаются. */
       while(_fgetts(Response.Record, MAX_RQRS_LEN, fp) != NULL) {
        WriteFile(Key [KeyIndex].hNp, &Response, RS_SIZE, &nXfer, &ovResp);
        WaitForSingleObject(hWrEvent, INFINITE);
       }
       fclose(fp);
       /* Уничтожить содержимое временного файла. */
       SetFilePointer(hTmpFile, 0, NULL, FILE_BEGIN);
       SetEndOfFile(hTmpFile);
       /* Отправить признак конца ответа. */
       Response.Status = 1;
       strcpy(Response.Record, "");
       WriteFile(Key[KeyIndex].hNp, &Response, RS_SIZE, &nXfer, &ovResp);
       WaitForSingleObject(hWrEvent, INFINITE);
       /* Конец основного командного цикла. Получить следующую команду.*/
          Disconnect = !ReadFile(Key[KeyIndex].hNp, &Key[KeyIndex].Req, RQ_SIZE,
&nXfer, &Key[KeyIndex].Ov) && GetLastError() == ERROR_HANDLE_EOF; /* Следующее
чтение */
       if (Disconnect) continue;
       Success = TRUE;
      }
     } __finally {
      if (Disconnect) {
       /* Создать еще одно соединение по этому каналу. */
       Key[KeyIndex].Type = 0;
       DisconnectNamedPipe(Key[KeyIndex].hNp);
       ConnectNamedPipe(Key[KeyIndex].hNp, &Key[KeyIndex].Ov);
      }
      if (!Success) {
       ReportError(_T("Ошибка сервера"), 0, TRUE);
       Exit = TRUE;
      }
     }
     FlushFileBuffers(Key[KeyIndex].hNp);
     DisconnectNamedPipe(Key[KeyIndex].hNp);
     CloseHandle(hTmpFile);
     /* … */
     _endthreadex(0);
     return 0;
     /* Подавление предупреждающих сообщений компилятора. */
    }
```

Резюме

Для выполнения асинхронных операций ввода/вывода в Windows предусмотрены три метода. Самой распространенной и наиболее простой является методика, основанная на использовании потоков, которая, в отличие от двух остальных, способна работать даже под управлением Windows 9x. Каждый из потоков отвечает за выполнение определенной последовательности действий, состоящей из одной или нескольких последовательно выполняющихся, блокирующихся операций ввода/вывода. Кроме того, каждый поток должен располагать собственным дескриптором файла или канала.

Перекрывающийся ввод/вывод обеспечивает возможность выполнения асинхронных операций одним потоком с использованием одного дескриптора файла, но каждой отдельной операции вместо пары "поток—дескриптор файла" должен предоставляться дескриптор события. При этом требуется организовать ожидание завершения выполнения каждой конкретной операции ввода/вывода по отдельности, а затем очищать системные ресурсы или выполнять любые другие действия, необходимые для управления последовательностью выполнения операций.

С другой стороны, расширенный ввод/вывод автоматически вызывает код завершения и не требует использования дополнительных событий.

Одним неоспоримым преимуществом перекрывающегося ввода/вывода является то, что он предоставляет возможность создания портов завершения ввода/вывода, однако, о чем ранее уже говорилось и что иллюстрируется программой atouMTCP, которая находится на Web-сайте, но ценность и этого преимущества несколько снижается из-за того, что для ограничения количества активных потоков в пуле рабочих потоков могут быть использованы семафоры. Дополнительным недостатком портов завершения является то, что они не допускают удаления присоединенных к ним дескрипторов.

> UNIX обеспечивает поддержку потоков средствами Pthreads, что ранее уже обсуждалось.
>
> В System V UNIX асинхронный ввод/вывод ограничивается потоками и не может использоваться для выполнения операций с файлами и каналами.
>
> В версии BSD 4.3 для указания события, связанного с дескриптором файла, и выбора функции с целью определения состояния готовности дескрипторов файлов используется комбинация сигналов (SIGIO). Для дескрипторов файлов должен устанавливаться режим O_ASYNC. Такой подход может использоваться только с терминалами и в сетевых коммуникациях.

В следующих главах

Глава 15 завершает наше обсуждение Windows API демонстрацией методов обеспечения безопасности объектов Windows. Основное внимание уделяется защите файлов, но те же самые методы можно применять и к другим объектам, например, именованным каналам или процессам.

Упражнения

14.1. Воспользуйтесь асинхронным вводом/выводом для слияния нескольких отсортированных файлов в один отсортированный файл большего размера.

14.2. Приводит ли использование флага FILE_FLAG_MO_BUFFERING к повышению производительности программ atouOV и atouEX, как того можно было бы ожидать в соответствии с утверждениями, содержащимися в описании функции CreateFile? Существуют ли какие-либо ограничения, касающиеся размера файлов?

14.3. Модифицируйте программу timebeep (программа 14.3), введя в нее сбрасываемый вручную уведомляющий таймер.

14.4. Модифицируйте клиент именованного канала в программе clientNP (программа 11.2), введя в него перекрывающийся ввод/вывод, чтобы клиент мог продолжать работу после отправки запроса. В результате этого один клиент сможет иметь нескольких невыполненных запросов.

14.5. Перепишите программу serversk (программа 12.2), представляющую сервер на базе сокетов, введя в нее порты завершения ввода/вывода.

14.6. Перепишите одну из программ serverSK или serverNP таким образом, чтобы количество готовых к выполнению рабочих потоков ограничивалось семафором. Выполните эксперименты с большим пулом потоков, чтобы выяснить, насколько эффективен такой альтернативный вариант. Находящаяся на Web-сайте программа serverSM является модифицированным вариантом программы serverNP. С увеличением объемов доступной физической памяти и распространением платформы Win64 относительная ценность этого подхода и портов завершения может варьироваться.

14.7. Используйте программу управления заданиями JobShell (программа 6.3) для работы с большим количеством клиентов и исследуйте сравнительную способность к реагированию серверов serverNP и serverCP. Дополнительную нагрузку могут составить сетевые клиенты. Определите оптимальный интервал значений для количества активных потоков.

ГЛАВА 15
Безопасность объектов Windows

Windows поддерживает тщательно продуманную модель безопасности, которая исключает возможность несанкционированного доступа к таким объектам, как файлы, процессы или отображения файлов. Защитить можно почти любой из совместно используемых (разделяемых) объектов, и программист располагает возможностями управления правами доступа с высокой степенью их детализации.

Windows как единая система зарегистрирована в Оранжевой книге Управления национальной безопасности США (National Security Agency Orange Book) как система с сертифицированным уровнем безопасности C2, который требует обеспечения разграничительного контроля доступа с возможностью разрешения или запрещения тех или иных прав доступа к объекту на основании идентификационных данных пользователя, пытающегося получить доступ к объекту. Кроме того, система безопасности Windows распространяется на сетевую среду.

Тема безопасности слишком обширна, чтобы ее можно было полностью рассмотреть в рамках одной главы. Поэтому внимание в данной главе сосредоточено непосредственно на демонстрации того, каким образом API безопасности Windows используется для защиты объектов от несанкционированного доступа. Хотя средства контроля доступа образуют лишь подмножество функциональных средств безопасности Windows, они представляют самый непосредственный интерес для тех, кто хочет ввести элементы защиты в программы, приведенные в данной книге. Самый первый пример, программа 15.1, демонстрирует эмуляцию системы *полномочий* (permissions) на доступ к файлам, принятой в UNIX, в случае файлов NTFS, тогда как во втором примере в роли защищаемых объектов выступают именованные каналы. Те же принципы далее могут быть использованы для организации защиты других объектов. В списке литературы указаны источники, обратившись к которым вы сможете получить дополнительную информацию по обеспечению безопасности объектов.

Описанные средства защиты будут работать только под управлением Windows NT, и их нельзя использовать в системах семейства Windows 9x.

Атрибуты безопасности

В этой главе мы исследуем средства контроля доступа Windows сверху вниз, чтобы увидеть, как строится система безопасности объектов. Вслед за общим обзором, но перед тем, как мы приступим к примерам, будут подробно описаны соответствующие функции Windows. В случае файлов для проверки и изменения некоторых атрибутов безопасности объектов NTFS можно воспользоваться проводником (Windows Explorer).

Почти для всех объектов, создаваемых при помощи системного вызова Create, предусмотрен параметр атрибутов безопасности (security attributes). Следовательно, программы могут защищать файлы, процессы, потоки, события, семафоры, именованные каналы и так далее. Первым шагом является включение указателя на структуру SECURITY_ATTRIBUTES в вызов Create. До сих пор мы всегда указывали в своих программах значение NULL для этого указателя или же использовали структуру SECURITY_ATTRIBUTES просто для создания наследуемых дескрипторов (глава 6). В реализации защиты объекта важную роль играет элемент lpSecurityDescriptor структуры SECURITY_ATTRIBUTES, являющийся указателем на *дескриптор безопасности* (security descriptor), который содержит описание владельца объекта и определяет, каким пользователям предоставлены те или иные права доступа или в каких правах им отказано.

Структура SECURITY_ATTRIBUTES была введена в главе 6, но для удобства мы еще раз приведем ее полное определение.

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

Значение параметра nLength следует устанавливать равным:

```
sizeof(SECURITY_ATTRIBUTES)
```

Параметр bInheritHandle управляет свойствами наследования дескриптора объекта другими процессами.

Отдельные компоненты дескриптора безопасности описываются в следующем разделе.

Общий обзор средств безопасности: дескриптор безопасности

Анализ дескриптора безопасности предоставляет хорошую возможность для общего ознакомления с наиболее важными элементами системы безопасности Windows. В этом разделе речь будет идти о самых различных элементах этой системы и функциях, которые ими управляют, и мы приступим к этому, рассмотрев структуру дескриптора безопасности.

Дескриптор безопасности инициализируется функцией InitializeSecurityDescriptor и состоит из следующих элементов:

• Идентификационный номер владельца (Security Identifier, SID) (описывается в следующем разделе, в котором рассматривается все, что связано с владельцами объектов).

• SID группы.

• Список разграничительного контроля доступа (Discretionary Access Control List, DACL) — список элементов, в явной форме регламентирующих права доступа к объекту для определенных пользователей или групп. В нашем обсуждении термин "ACL", употребляемый без префикса "D", будет относиться к DACL.

• Системный ACL (System ACL, SACL), иногда называемый *ACL аудиторского доступа* (audit access ACL).

Функции SetSecurityDescriptorOwner и SetSecurityDescriptorGroup связывают идентификаторы SID с дескрипторами безопасности, о чем говорится далее в разделе "Идентификаторы безопасности".

ACL инициализируются функцией Initialize ACL, а затем связываются с дескриптором безопасности с помощью функций SetSecurityDescriptorDacl и SetSecurityDescriptorSacl.

Атрибуты безопасности подразделяются на *абсолютные* (absolute) и *самоопределяющиеся относительные* (self-relative). На данном этапе мы не будем делать различия между ними, но вернемся к этому вопросу далее в настоящей главе. Дескриптор безопасности и его компоненты представлены на рис. 15.1.

Списки контроля доступа

Каждый ACL состоит из совокупности элементов контроля доступа (Access Control Entry, ACE). Существует два типа ACE: для разрешения данного вида доступа (allowed) и его запрета (denied).

Сначала список ACL инициализируют посредством функции InitializeAcl, а затем добавляют в него элементы ACE. Каждый ACE содержит SID и *маску доступа* (access mask), определяющую, какие именно права доступа предоставляются пользователю или группе, идентифицируемым по их SID, а в каких им отказано. В качестве типичного примера прав доступа к файлам можно привести права доступа FILE_GENERIC_READ и DELETE.

Добавление элементов ACE в разграничительные списки ACL осуществляется при помощи двух функций — AddAccessAllowedAce и AddAccessDenieddAce. Функция AddAuditAccessAce служит для добавления элементов в SACL, что позволяет отслеживать попытки доступа, осуществляемые с использованием указанного SID.

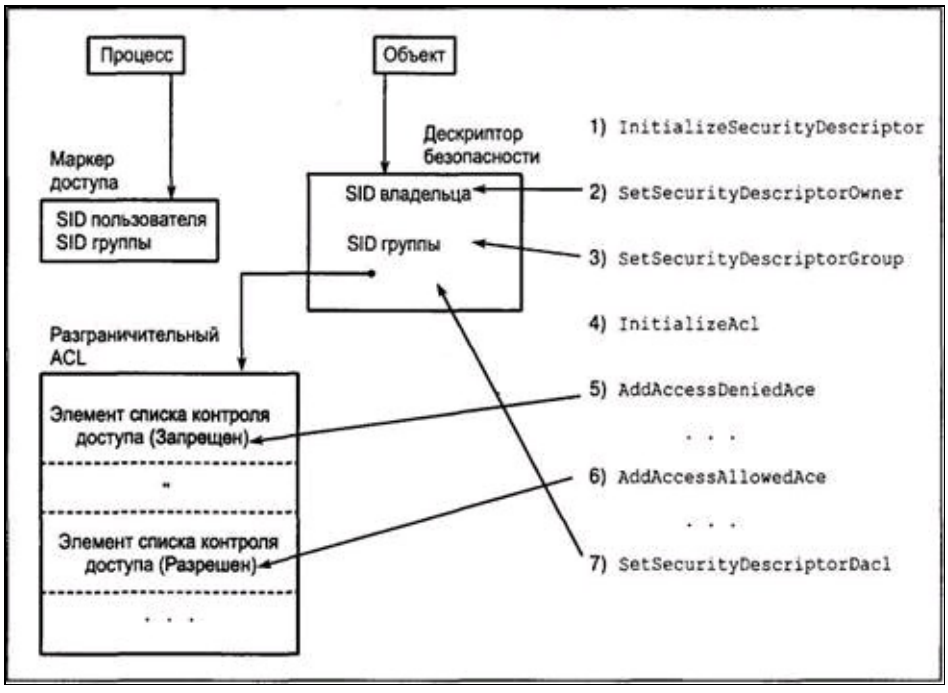


Рис. 15.1. Строение дескриптора безопасности

Наконец, для удаления ACE из списка используется функция DeleteAce, а для извлечения — функция GetAce.

Использование объектов безопасности Windows

В дескриптор безопасности вносятся многочисленные подробные данные, и на рис. 15.1 отражены лишь основные элементы его структуры. Заметьте, что у каждого процесса также имеется свой SID (содержащийся в маркере доступа), который используется ядром для того, чтобы определить, какие виды доступа разрешены или какие виды доступа подлежат аудиту. Кроме того, маркер доступа (access token) может предоставлять владельцу определенные *привилегии* (privileges) (свойственная данному владельцу способность выполнять операции, перекрывающая *права* (rights), указанные в списке ACL). Так, администратор может иметь привилегии на выполнение операций чтения и записи ко всем файлам, не имея на это прав, явно заданных в списке ACL данного файла.

Если пользовательские или групповые идентификаторы доступа не обеспечивают, ядро просматривает права доступа, указанные в ACL. Определяющую роль играет первый встреченный элемент, дающий возможность воспользоваться данной запрошенной услугой или отказывающий в этом. Поэтому очередность, в которой в список вносятся элементы ACE, имеет большое значение. Во многих случаях ACE, запрещающие доступ, располагаются первыми, чтобы конкретный пользователь, которому необходимо запретить данный вид доступа, не мог получить его, воспользовавшись членством в группе, которой этот вид доступа предоставлен. В то же время, для получения желаемой семантики в программе 15.1 существенно, чтобы элементы ACE, предоставляющие и запрещающие доступ, могли располагаться в произвольном порядке. ACE, отказывающий во всех видах доступа, может располагаться последним для гарантии того, что доступ не будет разрешен никому, если только он конкретно не указан в ACE.

Права объектов и доступ к объектам

Любой объект, например файл, получает свои права доступа при создании, однако

впоследствии эти права могут быть изменены. Процессу требуется доступ к объекту, когда он запрашивает дескриптор, используя для этого, например, вызов функции CreateFile. В одном из параметров запроса дескриптора содержится указание на желаемый вид доступа, например FILE_GENERIC_READ. Если у процесса имеются необходимые права на получение требуемого доступа, запрос завершается успешно. Для различных дескрипторов одного и того же объекта может быть определен различный доступ. Для указания флагов доступа используются те же значения, которые использовались для предоставления прав или отказа в них при создании ACL.

Стандартом UNIX (без C2 или других расширений) поддерживается более простая модель безопасности. Эта модель ограничивается файлами и основана на предоставлении полномочий доступа к файлам. В рассматриваемых в настоящей главе примерах программ эмулируется система полномочий доступа UNIX.

Инициализация дескриптора безопасности

Сначала необходимо инициализировать дескриптор безопасности с помощью функции InitializeSecurityDescriptor. Параметр pSecurityDescriptor должен указывать адрес действительной структуры SECURITY_DESCRIPTOR. Эти структуры являются непрозрачными для пользователя и управляются специальными функциями.

Для параметра dwRevision следует устанавливать значение:

```
SECURITY_DESCRIPTOR_REVISION
```

```
BOOL InitializeSecurityDescriptor(PSECURITY_DESCRIPTOR pSecurityDescriptor, DWORD dwRevision)
```

Управляющие флаги дескриптора безопасности

Флаги, входящие в структуру Control дескриптора безопасности, а именно, флаги SECURITY_DESCRIPTOR_CONTROL, определяют, какой смысл приписывается дескриптору безопасности. Некоторые из них устанавливаются и сбрасываются при помощи функций, которые будут рассмотрены далее. Доступ к управляющим флагам обеспечивают функции GetSecurityDescriptorControl и GetSecurityDescriptorControl (доступны в версии NT5), однако эти флаги не будут непосредственно использоваться в наших примерах.

Идентификаторы безопасности

Для идентификации пользователей и групп Windows использует идентификаторы SID. Программа может отыскивать SID по учетному имени (account name), которое может относиться к пользователю, группе, домену и так далее. Учетное имя может относиться и к удаленной системе. Сначала мы рассмотрим определение SID по учетному имени.

```
BOOL LookupAccountName(LPCTSTR lpSystemName, LPCTSTR lpAccountName, PSID Sid, LPDWORD cbSid, LPTSTR ReferencedDomainName, LPDWORD cbReferencedDomainName, PSID_NAME_USE peUse)
```

Параметры

lpSystemName и lpAccountName — указывают на системное и учетное имена. Для параметра lpSystemName часто используется значение NULL, обозначающее локальную систему.

Sid — возвращаемая информация, хранящаяся в структуре размером *cbSid. Если размер буфера недостаточно велик, функция выполняется с ошибкой, возвращая размер, который требуется.

ReferencedDomainName — строка, состоящая из *cbReferencedDomainName символов. Параметр длины строки должен инициализироваться размером буфера (для обработки ошибок используются обычные методы). Возвращаемое значение указывает домен, в котором обнаружено данное имя. В случае учетного имени Administrators возвращается значение BUILTIN, тогда как в случае пользовательского учетного имени возвращается имя этого пользователя.

peUse — указывает на переменную SID_NAME_USE (перечислительный тип данных), проверяемыми значениями которой могут быть SidTypeWellKnownGroup, SidTypeUser, SidTypeGroup и так далее.

Получение имени учетной записи и имени пользователя

При известном SID можно обратить процесс и получить имя учетной записи, используя функцию LookupAccountSid. Эта функция требует указания SID и возвращает соответствующее имя. Возвращаемым именем может быть любое имя, доступное процессу. Некоторые из имен, например Everyone, известны системе.

```
BOOL LookupAccountSid(LPCTSTR lpSystemName, PSID Sid, LPTSTR lpAccountName, LPDWORD cbName, LPTSTR ReferencedDomainName, LPDWORD cbReferencedDomainName, PSID NAME USE peUse)
```

Для получения учетного имени пользователя процесса (пользователя, вошедшего в систему) служит функция GetUserName.

```
BOOL GetUserName(LPTSTR lpBuffer, LPDWORD nSize)
```

Указатель на строку с именем пользователя и длина этой строки возвращаются обычным образом.

Для создания SID и управления ими могут использоваться такие функции, как InitializeSid и AllocateAndInitializeSid. Однако в примерах мы ограничимся использованием только SID, получаемых по учетному имени.

Полученные SID можно вносить в инициализированные дескрипторы безопасности.

```
    BOOL                    SetSecurityDescriptorOwner(PSECURITY_DESCRIPTOR
pSecurityDescriptor, PSID pOwner, BOOL bOwnerDefaulted)
    BOOL                    SetSecurityDescriptorGroup(PSECURITY_DESCRIPTOR
pSecurityDescriptor, PSID pGroup, BOOL bGroupDefaulted)
```

pSecurityDescriptor — указатель на соответствующий дескриптор безопасности, а oOwner (или pGroup) — адрес SID владельца (группы). Если для параметра bOwnerDefaulted (или bGroupDefaulted) установлено значение TRUE, то для извлечения информации о владельце (или первичной группе) используется механизм, заданный по умолчанию. В соответствии с этими двумя параметрами устанавливаются флаги SE_OWNER_DEFAULTED и SE_GROUP_DEFAULTED в структуре SECURITY_DESCRIPTOR_CONTROL.

Аналогичные функции GetSecurityDescriptorOwner и GetSecurityDescriptorGroup возвращают SID (пользователя или группы), извлекая соответствующую информацию из дескриптора безопасности.

Работа с ACL

В этом разделе показано, как работать со списками ACL, связывать ACL с дескриптором безопасности и добавлять ACL. Взаимосвязь между этими объектами и соответствующими функциями представлена на рис. 15.1.

Сначала необходимо инициализировать структуру ACL. Поскольку непосредственный доступ к ACL осуществляться не должен, его внутреннее строение для нас безразлично. Однако программа должна предоставить буфер, выступающий в роли ACL; обработка содержимого выполняется соответствующими функциями.

```
BOOL InitializeAcl(PACL pAcl, DWORD cbAcl, DWORD dwAclRevision)
```

pAcl — адрес предоставляемого программистом буфера размером cbAcl байт. В ходе последующего обсуждения и в программе 15.4 будет показано, как определить размер ACL, но для большинства целей размера 1 Кбайт будет вполне достаточно. Значение параметра dwAclRevision следует устанавливать равным ACL_REVISION.

Далее мы должны добавить ACE в желаемом порядке, используя функции AddAccessAllowedAce и AddAccessDeniedAce.

```
    BOOL AddAccessAllowedAce(PACL pAcl, DWORD dwAclRevision DWORD
dwAccessMask, PSID pSid)
    BOOL AddAccessDeniedAce(PACL pAcl, DWORD dwAclRevision, DWORD
dwAccessMask, PSID pSid)
```

Параметр pAcl указывает на ту же структуру ACL, которая была инициализирована функцией InitializeACL, а параметр dwAclRevision также следует устанавливать равным ACL_REVISION. Параметр pSid указывает на SID, например на тот, который был получен с помощью функции LookupAccountName.

Права, которые предоставляются или в которых отказывается пользователю или группе, идентифицируемым данным SID, определяются маской доступа (dwAccessMask).

Последнее, что потребуется сделать — это связать ACL с дескриптором безопасности. В случае разграничительного ACL для этого используется функция SetSecurityDescriptorDacl.

```
    BOOL SetSecurityDescriptorDacl(PSECURITY_DESCRIPTOR
pSecurityDescriptor, BOOL bDaclPresent, PACL pAcl, BOOL fDaclDefaulted)
```

Значение параметра bDaclPresent, равное TRUE, указывает на то, что в структуре pAcl имеется ACL. Если этот параметр равен FALSE, то следующие два параметра, pAcl и fDaclDefaulted, игнорируются. Флаг SE_DACL_PRESENT структуры SECURITY_DESCRIPTOR_CONTROL также устанавливается равным значению этого параметра.

Значение FALSE параметра fDaclDefaulted указывает на то, что ACL был сгенерирован программистом. В противном случае ACL был получен с использованием механизма, принятого по умолчанию, например, путем наследования; вместе с тем, для указания того, что имеется ACL, значение параметра должно быть равным TRUE. Флаг SE_DACL_PRESENT структуры SECURITY_DESCRIPTOR_CONTROL также устанавливается равным значению этого параметра.

Доступны и другие функции, предназначенные для удаления и считывания ACE из ACL; они обсуждаются далее в этой главе. А теперь настало время обратиться к примеру.

Пример: использование разрешений на доступ в стиле UNIX к файлам NTFS

Система разрешений на доступ к файлам, принятая в UNIX, предоставляет удобную возможность проиллюстрировать работу системы безопасности Windows, хотя последняя по своему характеру является гораздо более общей, чем стандартные средства защиты UNIX. В приведенной ниже реализации создается девять ACE, предоставляющих или запрещающих доступ по чтению, записи или запуску файлов на выполнение владельцу (owner), группе (group) и прочим пользователям (everyone). Предусмотрены две команды.

1. chmodW — имитирует UNIX-команду chmod. В данной реализации возможности команды расширены за счет того, что в случае отсутствия указанного файла он будет создан, а также за счет того, что пользователю предоставляется возможность указывать имя группы.

2. lsFP — расширенный вариант команды lsW (программа 3.2). Если запрошен вывод подробной информации, то отображается имя пользователя-владельца файла, а также результат интерпретации существующих ACL, которые могли быть установлены командой chmodW.

Указанные две команды представлены программами 15.1 и 15.2. В программах 15.3, 15.4 и 15.5 реализованы три вспомогательные функции.

1. InitializeUnixSA, которая создает действительную структуру атрибутов безопасности, соответствующих набору разрешений доступа UNIX. Эта функция обладает достаточной общностью, чтобы ее можно было применять по отношению к таким объектам, отличным от файлов, как процессы (глава 6), именованные каналы (глава 11) и объекты синхронизации (глава 8).

2. ReadFilePermissions.

3. ChangeFilePermissions.

> **Примечание**
>
> Приведенные ниже программы являются упрощенными вариантами программ, представленных на Web-сайте книги. В полных вариантах программ используются отдельные массивы AllowedAceMasks и DeniedAceMasks, в то время как в листингах ниже задействован только один массив.
>
> Использование отдельного массива DeniedAceMasks обеспечивает невозможность запрета прав доступа SYNCHRONIZE, поскольку флаг SYNCHRONIZE устанавливается во всех трех макросах FILE_GENERIC_READ, FILE_GENERIC_WRITE и FILE_GENERIC_EXECUTE, которые являются комбинациями нескольких флагов (см. заголовочный файл WINNT.H). Дополнительные разъяснения предоставляются в полном варианте программы, доступном на Web-сайте. Кроме того, в полном варианте программы проверяется, не указано ли в командной строке групповое имя; ниже мы будем везде предполагать, что указывается имя пользователя.

Программа 15.1. chmodW: изменение разрешений на доступ к файлу

```
/* Глава 15. Команда chmodW. */
/* chmodW [опции] режим файл [ИмяГруппы].
   Изменение режима доступа к именованному файлу.
   Опции:
     -f Принудительный режим — не выводить предупреждающие сообщения в случае
```

```
невозможности изменения режима.
          -c Создать файл, если он не существует. Необязательное имя группы
указывается после имени файла. */
    /* Требуются NTFS и Windows NT (под управлением Windows 9x программа работать
не будет). */

    #include "EvryThng.h"

    int _tmain(int argc, LPTSTR argv[]) {
     HANDLE hFile, hSecHeap;
     BOOL Force, CreateNew, Change, Exists;
     DWORD Mode, DecMode, UsrCnt = ACCT_NAME_SIZE;
     TCHAR UsrNam[ACCT_NAME_SIZE];
     int FileIndex, GrpIndex, ModeIndex;
      /* Массив прав доступа к файлу, следующих в том порядке, который принят в
UNIX. */
     /* Эти права будут различными для объектов различного типа. */
     /*ПРИМЕЧАНИЕ: в полном варианте программы, находящемся на Web-сайте, */
     /*используются отдельные массивы масок разрешения и запрещения доступа.*/
     DWORD AceMasks[] = {
      FILE_GENERIC_READ, FILE_GENERIC_WRITE, FILE_GENERIC_EXECUTE
     };
     LPSECURITY_ATTRIBUTES pSa = NULL;
     ModeIndex = Options(argc, argv, _T("fc"), &Force, &CreateNew, NULL);
     GrpIndex = ModeIndex + 2;
     FileIndex = ModeIndex + 1;
     DecMode = _ttoi(argv[ModeIndex]);
     /* Режим защиты представляет собой восьмеричное число. */
     Mode = ((DecMode / 100) % 10) * 64 /*Преобразовать в десятичное число.*/
          + ((DecMode / 10) % 10) * 8 + (DecMode % 10);
     Exists = (_taccess(argv[FileIndex], 0) == 0);
     if (!Exists && CreateNew) {
      /* Файл не существует; создать новый файл. */
      GetUserName(UsrNam, &UsrCnt);
      pSa = InitializeUnixSA(Mode, UsrNam, argv[GrpIndex], AceMasks, &hSecHeap);
          hFile   =   CreateFile(argv[FileIndex],   0,   0,   pSa,   CREATE_ALWAYS,
FILE_ATTRIBUTE_NORMAL, NULL);
      CloseHandle(hFile);
         HeapDestroy(hSecHeap);  /*  Освободить  память,  занимаемую  структурами
безопасности. */
     }
     else if (Exists) { /* Файл существует; изменить разрешения доступа. */
      Change = ChangeFilePermissions(Mode, argv[FileIndex], AceMasks);
     }
     return 0;
    }
```

В программе 15.2 представлена соответствующая часть команды lsFP, а именно, функция Process Item.

Программа 15.2. lsFP: перечисление разрешений на доступ к файлу

```
    static  BOOL  ProcessItem(LPWIN32_FIND_DATA  pFileData,  DWORD  NumFlags,  LPBOOL
Flags)
    /* Вывести список атрибутов с указанием разрешений доступа и владельца. */
    /* Требуются NTFS и Windows NT (под управлением Windows 9x программа работать
не будет). */
```

```
    {
     DWORD FType = FileType(pFileData), Mode, i;
     BOOL Long = Flags[1];
     TCHAR GrpNam[ACCT_NAME_SIZE], UsrNam[ACCT_NAME_SIZE];
     SYSTEMTIME LastWrite;
     TCHAR PermString[] = _T("---------");
     const TCHAR RWX[] = {'r','w','x'}, FileTypeChar[] = {' ', 'd'};
     if (FType != TYPE_FILE && FType != TYPE_DIR) return FALSE;
     _tprintf(_T("\n"));
     if (Long) {
      Mode = ReadFilePermissions(pFileData->cFileName, UsrNam, GrpNam);
      if (Mode == 0xFFFFFFFF) Mode = 0;
      for (i = 0; i < 9; i++) {
       if (Mode >> (8 - i) & 0x1) PermString[i] = RWX[i % 3];
      }
       _tprintf(_T("%c%s 18.7s %8.7s%10d"), FileTypeChar[FType - 1], PermString,
UsrNam, GrpNam, pFileData->nFileSizeLow);
      FileTimeToSystemTime(&(pFileData->ftLastWriteTime), &LastWrite);
           _tprintf(_T("    %02d/%02d/%04d    %02d:%02d:%02d"),    LastWrite.wMonth,
LastWrite.wDay,      LastWrite.wYear,      LastWrite.wHour,      LastWrite.wMinute,
LastWrite.wSecond);
     }
     _tprintf(_T(" %s"), pFileData->cFileName);
     return TRUE;
    }
```

Далее мы рассмотрим реализацию вспомогательных функций.

Пример: инициализация атрибутов защиты

Программа 15.3 представляет вспомогательную функцию InitializeUnixSA. Эта функция создает структуру атрибутов безопасности, которая содержит ACL с элементами ACE, эмулирующими разрешения на доступ к файлам в UNIX. Существует девять ACE, предоставляющих или запрещающих доступ по чтению, записи или запуску файлов на выполнение владельцу (owner), группе (group) и прочим пользователям (everyone). Эта структура не является локальной переменной функции и должна распределяться и инициализироваться, а затем возвращаться вызывающей программе; обратите внимание на массив AceMasks в программе 15.1.

Программа 15.3. InitUnFp: инициализация атрибутов защиты

```
    /* Задание режима доступа в стиле UNIX посредством элементов ACE, хранящихся в
структуре SECURITY_ATTRIBUTES. */

    #include "EvryThng.h"
    #define ACL_SIZE 1024
    #define INIT_EXCEPTION 0x3
    #define CHANGE_EXCEPTION 0x4
    #define SID_SIZE LUSIZE
    #define DOM_SIZE LUSIZE

    LPSECURITY_ATTRIBUTES InitializeUnixSA(DWORD UnixPerms, LPCTSTR UsrNam, LPCTSTR
GrpNam, LPDWORD AceMasks, LPHANDLE pHeap) {
     HANDLE SAHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0, 0);
     LPSECURITY_ATTRIBUTES pSA = NULL;
     PSECURITY_DESCRIPTOR pSD = NULL;
     PACL pAcl = NULL;
     BOOL Success;
     DWORD iBit, iSid, UsrCnt = ACCT_NAME_SIZE;
      /* Таблицы имен пользователя (User), группы (Group) и прочих пользователей
(Everyone), идентификаторов SID и так далее для LookupAccountName и создания SID.
*/
     LPCTSTR pGrpNms[3] = {EMPTY, EMPTY, _T("Everyone")};
     PSID pSidTable[3] = {NULL, NULL, NULL};
     SID_NAME_USE sNamUse[3] = {SidTypeUser, SidTypeGroup, SidTypeWellKnownGroup};
     TCHAR RefDomain[3][DOM_SIZE];
     DWORD RefDomCnt[3] = {DOM_SIZE, DOM_SIZE, DOM_SIZE};
     DWORD SidCnt[3] = {SID_SIZE, SID_SIZE, SID_SIZE};
     __try { /* Блок try-except для исключений при распределении памяти. */
      *pHeap = SAHeap;
      pSA = HeapAlloc(SAHeap, 0, sizeof (SECURITY_ATTRIBUTES));
      pSA->nLength = sizeof(SECURITY_ATTRIBUTES);
      pSA->bInheritHandle = FALSE;
      /* Программист может выполнить эти установки позже. */
      pSD = HeapAlloc(SAHeap, 0, sizeof(SECURITY_DESCRIPTOR));
      pSA->lpSecurityDescriptor = pSD;
      InitializeSecurityDescriptor(pSD, SECURITY DESCRIPTOR REVISION);
      /* Получить SID пользователя, группы и прочих пользователей.
       * Другие важные подробности можно найти на Web-сайте. */
      pGrpNms[0] = UsrNam;
      pGrpNms[1] = GrpNam;
      for (iSid = 0; iSid < 3; iSid++) {
```

```
       pSidTable[iSid] = HeapAlloc(SAHeap, 0, SID_SIZE);
          LookupAccountName(NULL, pGrpNms[iSid], pSidTable[iSid], &SidCnt[iSid],
RefDomain[iSid], &RefDomCnt[iSid], &sNamUse[iSid]);
       }
       SetSecurityDescriptorOwner(pSD, pSidTable[0], FALSE);
       SetSecurityDescriptorGroup(pSD, pSidTable[1], FALSE);
       pAcl = HeapAlloc(ProcHeap, HEAP_GENERATE_EXCEPTIONS, ACL_SIZE);
       InitializeAcl(pAcl, ACL_SIZE, ACL_REVISION);
       /* Добавить все элементы ACE, разрешающие и запрещающие доступ. */
       for (iBit = 0; iBit < 9; iBit++) {
          if ((UnixPerms >> (8 - iBit) & 0x1) != 0 && AceMasks[iBit%3] != 0)
AddAccessAllowedAce(pAcl, ACL_REVISION, AceMasks [iBit%3], pSidTable [iBit/3]);
          else if (AceMasks[iBit%3] != 0) AddAccessDeniedAce(pAcl, ACL_REVISION,
AceMasks [iBit%3], pSidTable [iBit/3]);
       }
       /* Добавить запрет доступа для всех ACE категории "Прочие". */
            Success = Success && AddAccessDeniedAce(pAcl, ACL_REVISION,
STANDARD_RIGHTS_ALL | SPECIFIC_RIGHTS_ALL, pSidTable[2]);
       /* Связать ACL с атрибутом защиты. */
       SetSecurityDescriptorDacl(pSD, TRUE, pAcl, FALSE);
       return pSA;
      } /* Конец блока try-except. */
      __except(EXCEPTION_EXECUTE_HANDLER) { /* Освободить все ресурсы. */
       if (SAHeap != NULL) HeapDestroy(SAHeap);
       pSA = NULL;
      }
      return pSA;
     }
```

Комментарии к программе 15.3

Хотя структура программы 15.3 и может показаться несложной, выполняемую ею операцию вряд ли можно назвать простой. Кроме того, программа иллюстрирует целый ряд моментов, заслуживающих внимания, которые касаются использования средств безопасности Windows.

• Необходимо распределить в памяти несколько областей, предназначенных для хранения нужной информации, например, идентификаторов SID. Эти области создаются в специально выделенной для этих целей куче, которая по завершении работы должна быть уничтожена вызывающей программой.

• В данном примере структура атрибутов безопасности относится к файлам, но она также может использоваться с другими объектами, например именованными каналами (глава 11). В программе 15.4 показано, как встроить такую структуру при работе с файлами.

• Для эмуляции поведения UNIX существенное значение имеет порядок следования элементов ACE. Обратите внимание на то, что ACE, разрешающие и запрещающие доступ, добавляются в ACL по мере обработки битов, кодирующих полномочия, в направлении слева (Owner/Read) направо (Everyone/Execute). Благодаря этому биты полномочий, заданные, например, кодом защиты 460 (в восьмеричном представлении), будут запрещать пользователю доступ по записи даже в том случае, если он входит в состав группы.

• Права доступа описываются в ACE такими значениями, как FILE_GENERIC_READ или FILE_GENERIC_WRITE, которые аналогичны флагам, используемым в функции CreateFile, хотя добавляются и другие флаги доступа, например SYNCHRONIZE. Эти права указываются в вызывающей программе (в данном случае в программе 15.1), чтобы обеспечить их соответствие объекту.

• Значение, определенное для константы ACL_SIZE, выбрано достаточно большим, чтобы выделенных для него разрядов хватило для хранения девяти элементов ACE. После того как мы рассмотрим программу 15.5, способ определения требуемого размера элемента данных станет для вас очевидным.

• В функции используются три SID, по одному для каждой из следующих категорий пользователей: User (Пользователь), Group (Группа) и Everyone (Прочие). Для получения имени, используемого в качестве аргумента при вызове функции LookupAccountName, используются три различные методики. Имя обычного пользователя поступает из функции GetUserName. Именем пользователя, относящегося к категории прочих пользователей, является Everyone в SidTypeWellknownGroup. Групповое имя должно предоставляться в виде аргумента командной строки и отыскиваться как SidTypeGroup. Для нахождения групп, которым принадлежит пользователь, требуются определенные сведения о дескрипторах процесса, и решить эту задачу вам предлагается в упражнении 15.12.

• В версии программы, находящейся на Web-сайте книги, в отличие от той, которая представлена здесь, большое внимание уделено проверке ошибок. В ней даже предусмотрена проверка действительности сгенерированных структур с помощью функций IsValidSecurityDescriptor, IsValidSid и IsValidAcl, названия которых говорят сами за себя. Указанное тестирование ошибок оказалось чрезвычайно полезным на стадии отладки.

Чтение и изменение дескрипторов безопасности

После того как дескриптор безопасности связан с файлом, следующим шагом является определение кода защиты существующего файла и его возможное изменение. Для получения и установления кода защиты файла в терминах дескрипторов безопасности используются следующие функции:

```
    BOOL GetFileSecurity(LPCTSTR lpFileName, SECURITY_INFORMATION secInfo,
PSECURITY_DESCRIPTOR    pSecurityDescriptor,    DWORD    cbSd,    LPDWORD
lpcbLengthNeeded)
    BOOL SetFileSecurity(LPCTSTR lpFileName, SECURITY_INFORMATION secInfo,
PSECURITY_DESCRIPTOR pSecurityDescriptor)
```

Переменная secInfo имеет тип перечисления и принимает значения:

```
OWNER_SECURITY_INFORMATION
GROUP_SECURITY_INFORMATION
DACL_SECURITY_INFORMATION
SACL_SECURITY_INFORMATION
```

которые позволяют указать, какую часть дескриптора безопасности необходимо получить или установить. Эти значения могут объединяться при помощи поразрядной операции "или".

Наилучшим способом определения необходимого размера возвращаемого буфера для функции GetFileSecurity является двукратный вызов этой функции. Во время первого вызова значение параметра cbSd может быть задано равным 0. После того как буфер выделен, вызовите эту функцию второй раз. Этот принцип применяется в программе 15.4.

Вряд ли следует подчеркивать тот факт, что для выполнения этих операций требуются соответствующие полномочия. Так, для успешного выполнения функции SetFileSecurity необходимо либо иметь полномочия на уровне WRITE_DAC, либо быть владельцем объекта.

Функции GetSecurityDescriptorOwner и GetSecurityDescriptorGroup позволяют извлекать идентификаторы SID из дескриптора безопасности, полученного при помощи функции GetFileSecurity. Для получения ACL следует воспользоваться функцией GetSecurityDescriptorDacl.

```
    BOOL                    GetSecurityDescriptorDacl(PSECURITY_DESCRIPTOR
pSecurityDescriptor,    LPBOOL    lpbDaclPresent,    PACL    *pAcl,    LPBOOL
lpbDaclDefaulted)
```

Параметры этой функции почти полностью совпадают с параметрами функции GetSecurityDescriptorDacl за исключением того, что возвращаются флаги,указывающие на то, действительно ли представлен разграничительный ACL и был ли он установлен по умолчанию или пользователем.

Чтобы иметь возможность интерпретировать список ACL, необходимо выяснить, сколько элементов ACE в нем содержится.

```
    BOOL   GetAclInformation(PACL   pAcl,   LPVOID   pAclInformation,   DWORD
cbAclInfo, ACL INFORMATION CLASS dwAclInfoClass)
```

В большинстве случае параметр информационного класса ACL, dwAclInfoClass, равен AclSizeInformation, а параметр pAclInformation представляет собой структуру типа ACL_SIZE_INFORMATION. Другим возможным значением параметра класса является AclRevisionInformation.

В структуру ACL_SIZE_INFORMATION входят три элемента, наиболее важным из которых является AceCount, который указывает, сколько элементов содержится в списке. Чтобы выяснить, достаточно ли велик размер ACL, проверьте значения элементов AclBytesInUse и AclBytesFree структуры ACL_SIZE_INFORMATION.

Функция GetAce извлекает извлекает ACE по заданному индексу.

```
BOOL GetAce(PACL pAcl, DWORD dwAceIndex, LPVOID *pAce)
```

Для получения определенного элемента ACE (их общее количество теперь известно) следует указать его индекс. pAce указывает на структуру ACE, в которой имеется элемент под названием Header, содержащий, в свою очередь, элемент AceType. Для проверки типа можно использовать значения ACCESS_ALLOWED_ACE и ACCESS DENIED ACE.

Пример: чтение разрешений на доступ к файлу

Программа 15.4 представляет собой функцию ReadFilePermissions, которая используется программами 15.1 и 15.2. Эта программа методично использует описанные выше функции для извлечения нужной информации. Правильная работа этой программы зависит от того факта, что ACL были созданы с помощью программы 15.3. Функция включена в тот же исходный модуль, что и программа 15.3, поэтому соответствующие объявления не повторяются.

Программа 15.4. ReadFilePermissions: чтение атрибутов безопасности

```
    DWORD ReadFilePermissions(LPCTSTR lpFileName, LPTSTR UsrNm, LPTSTR GrpNm)
    /* Возвращает разрешения на доступ к файлу в стиле UNIX. */
    {
     PSECURITY_DESCRIPTOR pSD = NULL;
     DWORD LenNeeded, PBits, iAce;
     BOOL DaclF, AclDefF, OwnerDefF, GroupDefF;
     BYTE DAcl[ACL_SIZE];
     PACL pAcl = (PACL)&DAcl;
     ACL_SIZE_INFORMATION ASizeInfo;
     PACCESS_ALLOWED_ACE pAce;
     BYTE AType;
     HANDLE ProcHeap = GetProcessHeap();
     PSID pOwnerSid, pGroupSid;
     TCHAR RefDomain[2][DOM_SIZE];
     DWORD RefDomCnt[] = {DOM_SIZE, DOM_SIZE);
     DWORD AcctSize[] = {ACCT_NAME_SIZE, ACCT_NAME_SIZE};
     SID_NAME_USE sNamUse[] = {SidTypeUser, SidTypeGroup};
     /* Получить требуемый размер дескриптора безопасности. */
            GetFileSecurity(lpFileName,         OWNER_SECURITY_INFORMATION         |
GROUP_SECURITY_INFORMATION | DACL_SECURITY_INFORMATION, pSD, 0, &LenNeeded);
     pSD = HeapAlloc(ProcHeap, HEAP_GENERATE_EXCEPTIONS, LenNeeded);
            GetFileSecurity(lpFileName,         OWNER_SECURITY_INFORMATION         |
GROUP_SECURITY_INFORMATION    |    DACL_SECURITY_INFORMATION,    pSD,    LenNeeded,
&LenNeeded);
     GetSecurityDescriptorDacl(pSD, &DaclF, &pAcl, &AclDefF);
         GetAclInformation(pAcl,     &ASizeInfo,     sizeof(ACL_SIZE_INFORMATION),
AclSizeInformation);
     PBits = 0; /* Вычислить разрешения на доступ на основе ACL. */
     for (iAce = 0; iAce < ASizeInfo.AceCount; iAce++) {
      GetAce(pAcl, iAce, &pAce);
      AType = pAce->Header.AceType;
      if (AType == ACCESS_ALLOWED_ACE_TYPE) PBits |= (0x1 << (8-iAce));
     }
     /* Определить имя владельца и владеющей группы. */
     GetSecurityDescriptorOwner(pSD, &pOwnerSid, &OwnerDefF);
     GetSecurityDescriptorGroup(pSD, &pGroupSid, &GroupDefF);
       LookupAccountSid(NULL,   pOwnerSid,   UsrNm,   &AcctSize[0],   RefDomain[0],
&RefDomCnt[0], &sNamUse[0]);
       LookupAccountSid(NULL,   pGroupSid,   GrpNm,   &AcctSize[1],   RefDomain[1],
&RefDomCnt[1], &sNamUse[1]);
     return PBits;
    }
```

Пример: изменение разрешений на доступ к файлу

Программа 15.5 является последней в нашем собрании функций, предназначенных для работы со средствами защиты файлов. Эта функция, ChangeFilePermissions, заменяет существующий дескриптор безопасности новым, сохраняя идентификаторы SID пользователя и группы, но создавая новый разграничительный список ACL.

Программа 15.5. ChangeFilePermissions: изменение атрибутов безопасности

```
    BOOL ChangeFilePermissions(DWORD fPm, LPCTSTR FNm, LPDWORD AceMsk)
    /* Изменить разрешения на доступ к существующему файлу. Разрешения на доступ
для группы остаются неизменными. */
    {
     TCHAR UsrNm[ACCT_NAME_SIZE], GrpNm[ACCT_NAME_SIZE];
     LPSECURITY_ATTRIBUTES pSA;
     PSECURITY_DESCRIPTOR pSD = NULL;
     HANDLE hSecHeap;
     if (_taccess(FNm, 0) != 0) return FALSE;
     ReadFilePermissions(FNm, UsrNm, GrpNm);
     pSA = InitializeUnixSA(fPm, UsrNm, GrpNm, AceMsk, &hSecHeap);
     pSD = pSA->lpSecurityDescriptor;
     SetFileSecurity(FileName, DACL_SECURITY_INFORMATION, pSD);
     HeapDestroy(hSecHeap);
     return TRUE;
    }
```

Комментарии по поводу разрешений на доступ к файлам

В процессе выполнения этих программ весьма интересно контролировать файловую систему через проводник Windows. Эта служебная программа не в состоянии интерпретировать ACE, разрешающие и запрещающие доступ, и не может отображать соответствующие разрешения. В Windows 4.0 проводник, сталкиваясь с такими ACE, будет генерировать исключения.

Вместе с тем, использование ACL, разрешающих и запрещающих доступ, необходимо для эмуляции семантики UNIX. Если этим пренебречь, то проводник Windows сможет обеспечить просмотр разрешений. Тогда, например, при коде защиты 0446 пользователь и члены группы смогут осуществлять запись в файл, поскольку это разрешено всем пользователям категории Everyone. В то же время, UNIX действует иначе; пользователю и членам группы эта операция будет запрещена.

Понаблюдайте также за тем, что происходит, когда вы пытаетесь создать защищенный файл на дискете или в другой FAT-системе, а также когда программа выполняется под управлением Windows 9x.

Защита объектов ядра и коммуникаций

В предыдущих разделах мы говорили главным образом о защите файлов, хотя те же методы можно применять и к другим объектам, построенным по типу файлов, например, именованным каналам (глава 11) или объектам ядра. Наш следующий пример, программа 15.6, предназначен для работы с именованными каналами, с которыми можно обращаться во многом точно так же, как с файлами.

Защита именованных каналов

Хотя соответствующая часть кода в листинге программы 11.3 опущена, сервер, полный программный код которого находится на Web-сайте книги, предоставляет возможность защиты его именованных каналов для предотвращения доступа к ним пользователей, не обладающих необходимыми полномочиями. Необязательные параметры командной строки позволяют указать имя пользователя и групповое имя:

```
Server [ИмяПользователя ИмяГруппы]
```

Если имена пользователя и группы опущены, используются коды защиты, заданные по умолчанию. Заметьте, что для создания необязательных атрибутов защиты в полной версии программы 11.3 (которая доступна на Web-сайте) и в программе 15.6 используются методы из программы 15.3. В то же время, вместо вызова функции InitUnixSA мы теперь вызываем более простую функцию InitializeAccessOnlySA, которая обеспечивает предоставление только доступа, разрешенного элементами ACE, и помещает последний ACE, запрещающий доступ, в конец списка ACL. В программе 15.6 представлены соответствующие участки кода, которые не были отражены в листинге программы 11.3. В случае именованных каналов важное значение имеют следующие права доступа:

- FILE_GENERIC_READ
- FILE_GENERIC_WRITE
- SYNCHRONIZE (разрешает потоку ожидать освобождения канала)

Если при подключении клиента требуется предоставить все права доступа, можно просто указать уровень доступа STANDARD_RIGHTS_REQUIRED. Для получения полного доступа (дуплексного, входящего, исходящего и так далее) вам также придется воспользоваться маской 0x1FF. В сервере, представленном в программе 15.6, предусмотрена защита экземпляров его именованных каналов с использованием этих прав доступа. Доступ к каналу имеют только клиенты, запущенные на выполнение владельцем канала, хотя предоставление доступа к каналу также членам группы не вызывает никаких сложностей.

Программа 15.6. ServerNP: защита именованного канала

```
/* Глава 15. ServerNP. Предусмотрена защита именованного канала.
 * Многопоточный сервер командной строки. Версия на основе
 * именованного канала.
 * Использование: Server [ИмяПользователя ИмяГруппы]. */
…
_tmain(int argc, LPTSTR argv[]) {
 …
 HANDLE hNp, hMonitor, hSrvrThread[MAXCLIENTS];
 DWORD iNp, MonitorId, ThreadId;
```

```
    DWORD AceMasks[] = /* Права доступа к именованному каналу. */
     {STANDARD_RIGHTS_REQUIRED | SYNCHRONIZE | 0X1FF, 0, 0 };
    LPSECURITY_ATTRIBUTES pNPSA = NULL;
    …
    if (argc == 4) /* Необязательный параметр защиты канала. */
     pNPSA = InitializeAccessOnlySA(0440, argv[1], argv[2], AceMasks, &hSecHeap);
    …
    /* Создать экземпляр канала для каждого серверного потока. */
    …
    for (iNp = 0; iNp < MAXCLIENTS; iNp++) {
      hNp = CreateNamedPipe(SERVER_PIPE, PIPE_ACCESS_DUPLEX, PIPE_READMODE_MESSAGE
| PIPE_TYPE_MESSAGE | PIPE_WAIT, MAXCLIENTS, 0, 0, INFINITE, pNPSA);
        if (hNp == INVALID_HANDLE_VALUE) ReportError(_T("Невозможно открыть
именованный канал."), 1, TRUE);
    }
    …
   }
```

Защита объектов ядра и приватных объектов

Многие объекты, такие как процессы, потоки или мьютексы, являются *объектами ядра* (kernel objects). Для получения и установки дескрипторов безопасности ядра используются функции GetKernelObjectsSecurity и SetKernelObjectsSecurity, аналогичные функциям защиты файлов, описанным в настоящей главе. Однако при этом вы должны знать, какие права доступа соответствуют данному объекту; в следующем разделе показано, как определить эти права.

Существует также возможность связывания дескрипторов безопасности с приватными, сгенерированными программой объектами, такими как объекты Windows Sockets или патентованные базы данных. Соответствующими функциями являются GetPrivateObjectSecurity и SetPrivateObjectSecurity. Ответственность за принудительное введение определенных прав доступа к таким объектам несет программист, который для изменения дескрипторов безопасности должен использовать функции CreatePrivateObjectSecurity и DestroyPrivateObjectSecurity.

Значения маски ACE

Модели "пользователь, группа, прочие", которую реализует функция InitUnixSA в большинстве случаев будет вполне достаточно, хотя с использованием тех же базовых методов могут реализовываться и другие модели.

Вместе с тем, для этого необходимо знать фактические значения маски ACE, которые соответствуют тому или иному объекту ядра. Эти значения не всегда достаточно хорошо документированы, но для их нахождения можно воспользоваться несколькими способами.

• Прочитайте документацию с описанием функции открытия интересующего вас объекта. Флаги доступа имеют те же значения, что и флаги, используемые в маске ACE. Так, функция OpenMutex использует флаги MUTEX_ALL_ACCESS и SYNCHRONIZE (второй из указанных флагов требуется для любого объекта, который может использоваться с функциями WaitForSingleObject или WaitForMultipleObjects). Другие объекты, например процессы, имеют множество других дополнительных флагов доступа.

• Полезная в этом отношении информация может содержаться также в документации по функциям "создания" объектов.

• Проверьте, не содержатся ли флаги, применимые к интересующему вас объекту, в

заголовочных файлах WINNT.H и WINBASE.H.

Пример: защита процесса и его потоков

В документации по функции OpenProcess представлена подробная градация прав доступа, соответствующих самым разнообразным функциям, выполнение которых требует применения дескриптора процесса.

Так, значение PROCESS_TERMINATE параметра доступа разрешает процессу (а фактически — потока внутри процесса) использовать дескриптор процесса в функции TerminateProcess для завершения процесса.

Доступ на уровне PROCESS_QUERY_INFORMATION разрешает использование дескриптора процесса при вызове функций GetExitCodeProcess или GetPriorityClass, тогда как уровень доступа PROCESS_ALL_ACCESS разрешает любой доступ, а доступ SYNCHRONIZE требуется для выполнения функций ожидания завершения процесса.

Чтобы проиллюстрировать эти идеи, на основе программы JobShell, рассмотренной в главе 6, была разработана программа JobShellSecure.c, которая разрешает доступ к управляемому процессу только его владельцу (или администратору). Эта программа находится на Web-сайте книги.

Обзор дополнительных возможностей защиты объектов

О средствах безопасности Windows можно было сказать намного больше, но настоящая глава является лишь введением в эту тему, показывая, как организовать защиту объектов Windows, используя API системы безопасности. В последующих разделах кратко рассмотрены дополнительные вопросы, относящиеся к этой тематике.

Удаление элементов ACE

Функция DeleteAce удаляет ACE, определяемый с помощью индекса аналогично тому, как это делается в случае функции GetAce.

Абсолютные и самоопределяющиеся относительные дескрипторы безопасности

Программа 15.5, позволяющая изменять ACL, удобна тем, что просто заменяет один дескриптор безопасности (SD) другим. В то же время, при замене существующих SD следует проявлять осторожность, поскольку они бывают двух типов: абсолютные (absolute) и самоопределяющиеся относительные (self-relative). Внутреннее устройство этих структур данных для наших целей не имеет значения, однако вы должны понимать, в чем состоит различие между ними, и как переходить от одного из них к другому.

• В процессе создания SD они являются абсолютными, и входящие в них указатели указывают на различные структуры, находящиеся в памяти. По сути, функция InitializeSecurityDescriptor создает абсолютный SD.

• При связывании SD с постоянно существующим объектом, например файлом, ОС объединяет все данные, относящиеся к SD, в одну компактную самоопределяющуюся структуру. В то же время, изменение SD (например, изменение ACL) порождает трудности при управлении пространством в пределах структуры абсолютного SD.

• Имеется возможность преобразовывать SD из одной формы в другую при помощи соответствующих функций Windows. Чтобы преобразовать самоопределяющийся относительный SD, например, возвращенный функцией GetFileSecurity, в абсолютный, используйте функцию MakeAbsoluteSD. Для обратного преобразования SD после внесения необходимых изменений служит функция MakeSelfRelativeSD. Функция MakeAbsoluteSD относится к числу тех функций Windows, которым огромное количество параметров придает устрашающий вид: из одиннадцати ее параметров по два приходится на каждый из четырех компонентов SD, по одному — на входной и выходной SD, а последний параметр предназначен для хранения размера результирующего абсолютного SD.

Системные списки ACL

Для управления системными списками ACL предусмотрен полный набор функций, однако использовать их может только системный администратор. Системные ACL определяют, какие разрешения на доступ к объекту должны быть зарегистрированы. Основной является функция AddAuditAccessAce, аналогичная функции AddAccessAllowed. В случае системных списков ACL понятие запрещенного доступа отсутствует.

Двумя другими функциями, предназначенными для работы с системными ACL, являются

функции GetSecurityDescrtiptorSacl и SetSecurityDescrtiptorSacl. Эти функции сопоставимы с их аналогами, предназначенными для работы с разграничительными ACL, — GetSecurityDescrtiptorDacl и SetSecurityDescrtiptorDacl.

Информация, хранящаяся в маркерах доступа

Программа 15.1 не решает задачи получения имен групп, связанных с процессом в его маркере доступа (access token). В ней просто требуется, чтобы имя группы указывал пользователь. Для получения соответствующей информации предназначена функция GetTokenInformation, требующая использования дескриптора процесса (глава 6). Эта задача решается в упражнении 15.12, в котором содержится подсказка к правильному решению. Сам код решения можно найти на Web-сайте книги.

Кроме того, в маркере доступа хранится информация о привилегиях доступа, так что процесс получает определенный доступ в соответствии со своими идентификационными данными, а не в соответствии с полномочиями доступа, связанными с объектом. Так, администратору требуется доступ, перекрывающий тот, который предоставляется данным конкретным объектом. Здесь опять необходимо обратить ваше внимание на различие между правами доступа (rights) и привилегиями (privileges).

Управление идентификаторами SID

В наших примерах SID получались по именам пользователя и группы, но вы также можете создавать новые SID с помощью функции AllocateAndlnitializeSid. Дополнительно имеется возможность получать информацию о SID при помощи других функций, а также копировать SID (CopySid) и сравнивать их между собой (CompareSid).

Протокол защищенных сокетов

Интерфейс Windows Sockets (Winsock), описанный в главе 12, обеспечивает связь между системами по сети. Winsock удовлетворяет промышленным стандартам, что делает возможным взаимодействие с системами, не принадлежащими семейству Windows. Протокол защищенных сокетов (Secure Sockets Layer, SSL), являющийся расширением Winsock, располагает уровень протокола безопасной передачи данных поверх базового транспортного протокола, что обеспечивает возможность аутентификации, шифрования и дешифрации сообщений.

Резюме

В Windows реализована тщательно разработанная модель безопасности объектов, возможности которой значительно превышают возможности стандартной системы защиты файлов UNIX. В примерах программ было показано, как эмулировать принятую в UNIX систему разрешений доступа и прав владения, устанавливаемых с помощью функций umask, chmod и chown. Программы также могут устанавливать владельца (группу и пользователя). Описанная эмуляция не является простой, однако результирующие функциональные возможности оказываются гораздо шире стандартных возможностей UNIX. Эта сложность обусловлена требованиями стандарта C2, изложенными в Оранжевой книге (Orange Book C2), в которых для определения списков управления доступом и и владельцами объектов используются маркеры доступа.

В следующей главе

Эта глава завершает наше рассмотрение Windows API. Следующая глава содержит обсуждение Win64, являющегося 64-битовым расширением Win32 API, и демонстрирует, как добиться того, чтобы программы правильно компоновались и выполнялись как в 32-битовом, так и в 64-битовом режимах.

Дополнительная литература

Windows

Администрирование систем безопасности и политики безопасности Windows обсуждаются в [2]. Углубленному рассмотрению проблем безопасности посвящена книга [32].

Строение и архитектура Windows NT

Подробное описание внутренней реализации механизмов безопасности Windows содержится в [38].

Стандарт безопасности Orange Book C2

Протокол безопасной передачи данных по сети C2 и другие уровни безопасности определены в публикации Министерства обороны США *DoD Trusted Computer System Evaluation Criteria.*

Упражнения

15.1. Расширьте возможности программы 15.1 таким образом, чтобы несколько групп могли иметь собственные уникальные разрешения доступа. Пары "имя-разрешение" могут выступать в качестве отдельных аргументов функции.

15.2. Расширьте возможности программы 15.4 таким образом, чтобы она могла выводить список всех групп, в дескрипторах безопасности объектов которых имеются ACE.

15.3. Убедитесь в том, что программа chmodW обеспечивает желаемое ограничение доступа к файлу.

15.4. Исследуйте, какие атрибуты безопасности по умолчанию вы получаете вместе с файлом.

15.5. Назовите другие маски доступа, которые можно использовать вместе с ACE. Дополнительную информацию вы можете найти в документации Microsoft.

15.6. Усовершенствуйте программы chmodW и lsFP таким образом, чтобы при попытке обработки файла, не относящегося к системе NTFS, выводилось сообщение об ошибке. Для этого потребуется использовать функцию GetVolumeInformation.

15.7. Усовершенствуйте программу chmodW, предусмотрев для нее опцию –o, позволяющую указывать, что пользователем программы является пользователь-владелец.

15.8. Определите фактический размер буфера ACL, необходимый программе 15.3 для хранения элементов ACE. В программе 15.3 для этой цели используется 1024 байт. Можете ли вы предложить формулу для расчета необходимого размера буфера ACL?

15.9. На Web-сайте Cygwin (http://www.cygwin.com) предлагается великолепная Linux-среда с открытым исходным кодом для Windows, предоставляющая командный процессор и реализацию таких команд, как chmod и ls. Установите эту среду и сравните варианты команд, реализованные в этой среде, с теми, которые разработаны в данной книге. Например, будет ли программа lsFP правильно отображать разрешения на доступ к файлу, если они были установлены с помощью соответствующей команды Cygwin, и наоборот. Сравните исходный код, представленный на Web-сайте Cygwin, с примерами из данной главы, что позволит вам критически оценить оба подхода к использованию средств безопасности Windows.

15.10. В библиотеку совместимости входят функции _open и _unmask, которые позволяют обрабатывать разрешения на доступ к файлам. Исследуйте, каким образом они эмулируют систему разрешений на доступ к файлам, принятую в UNIX, и сравните их с решениями, приведенными в этой главе.

15.11. Напишите программу для команды whoami, отображающей имя пользователя, который вошел в систему.

15.12. В программе 15.3, в которой создается дескриптор безопасности, требуется, чтобы программист предоставил имя группы. Модифицируйте программу таким образом, чтобы она создавала разрешения для всех пользовательских групп. *Подсказка.* Необходимо воспользоваться функцией OpenTokenProcess, возвращающей массив с именами групп, хотя вам потребуется провести некоторые эксперименты для выяснения способа хранения имен групп в массиве. Частичное решение вы найдете в исходном тексте программы, находящемся на Web-сайте.

15.13. Обратите внимание на то, что в клиент-серверной системе клиенты имеют доступ строго к тем же файлам и другим объектам, которые доступны серверу, установленному на серверной машине с правами доступа сервера. Снимите это ограничение, реализовав так называемое делегирование прав доступа (security delegation), используя функции ImpersonateNamedPipeClient и RevertToSelf. Клиенты, не принадлежащие группе, которая

применялась для защиты канала, подключаться к серверу не смогут.

15.14. Существует ряд других функций Windows, которые вы можете счесть полезными для себя и применить для упрощения или усовершенствования программ, предложенных в качестве примеров. Ознакомьтесь со следующими функциями: AreAllAccessesGranted, AreAnyAccessesGranted, AccessCheck и MapGenericMask. Можете ли вы воспользоваться этими функциями для упрощения или усовершенствования примеров?

ГЛАВА 16
Программирование в среде Win64

Наиболее заметный прогресс в развитии возможностей Windows после появления Windows NT и Windows 95 связан с приходом 64-разрядного программирования и расширением Win32 до Win64. На объединенный API обычно ссылаются просто как на Windows API, и именно такой практики мы придерживались на протяжении всей книги. API Win64 обеспечивает возможность выполнения в Windows наиболее крупных и требовательных в отношении ресурсов приложений уровня предприятий и приложений для научных расчетов. 64-разрядные системы позволяют программам использовать гигантские адресные пространства, которые выходят далеко за предел 4 Гбайт, обусловленный 32-битовой адресацией.

В данной главе описано нынешнее состояние Win64 и преимущества этого интерфейса, а также рассмотрена соответствующая модель программирования и обсуждены вопросы переносимости программ между различными операционными системами и аппаратными платформами. Это рассмотрение проводится безотносительно к фактическому типу 64-разрядного процессора или конкретной версии Windows, обеспечивающих поддержку Win64. Процесс переноса одного из предыдущих примеров иллюстрирует программа 16.1.

Нынешнее состояние Win64

В данном разделе анализируется состояние поддержки компанией Microsoft интерфейса Win64 на различных системах и процессорах, сложившееся к концу первого полугодия 2004 года. Поскольку ситуация постоянно меняется, приведенную ниже информацию следует рассматривать лишь в качестве "моментального снимка" реального положения дел. Тем не менее, на охватываемых здесь аспектах программирования эволюция поддержки Win64 никак не сказывается.

По-видимому, в будущем мы окажемся свидетелями значительного прогресса и изменений в этой области, хотя внедрение Win64 происходит довольно-таки медленно. Приведенная ниже информация почерпнута, как правило, на Web-сайтах соответствующих поставщиков и из отраслевых изданий, так что для получения впоследствии более свежих данных вы можете воспользоваться этими же источниками.

Поддержка процессоров

Win64 поддерживается или, о чем можно говорить почти с полной уверенностью, будет поддерживаться, по крайней мере, на трех различных семействах процессоров:

• Семейство процессоров Itanium (Itanium Processor family, IPF) компании Intel, архитектура которых полностью отличается от известной архитектуры Intel x86. IPF предоставляет большие регистровые файлы (включающие 128 регистров общего назначения), каналы многоадресных команд, встроенные трехуровневые кэши, а также множество других средств, обеспечивающих высокую производительность и 64-битовую адресацию. В настоящее время на рынок поставляются процессоры Itanium 2, и хотя их предшественник — процессор Itanium — является теперь уже устаревшим, нам будет удобно ссылаться на все семейство просто как на "процессоры Itanium".

• Процессоры Opteron и Athlon 64 (AMD64) компании AMD, предназначенные, соответственно, для серверов и рабочих станций. Архитектуру AMD64 можно рассматривать как расширение архитектуры Intel x86, допускающее 64-битовую виртуальную адресацию и параллельное выполнение 32– и 64-битовых операций.

• 32/64-разрядные процессоры компании Intel, сравнимые с процессорами AMD64. Во время написания этой книги ожидалось, что технология 64-разрядного расширения будет применена в первую очередь в процессорах Xeon. Как и прогнозировалось, такие процессоры появились на рынке в конце 2004 года.

Поддержка Windows

API Win64 компании Microsoft предназначен для поддержки 64-разрядных архитектур таким способом, при котором в существующие исходные и двоичные коды требуется вносить лишь минимальные изменения. В настоящее время имеется несколько отдельных версий Win64.

• Windows XP 64-bit Edition доступна в виде, по крайней мере, двух версий. Бета-версия компании Microsoft поддерживает только процессор AMD Opteron. Компания Hewlett Packard выводит на рынок несколько моделей рабочих станций на базе процессоров Itanium с уже установленной системой Windows XP-Itanium2.

• Windows Server 2003 Enterprise Edition for 64-bit Extended Systems в настоящее время также проходит бета-тестирование. Эта версия обеспечивает поддержку процессоров AMD Opteron и

Intel Xeon с использованием технологии 64-разрядного расширения.

• Windows Server 2003 Enterprise Edition for 64-bit Itanium-based Systems поддерживает, как говорит само ее название, серверы и рабочие станции, использующие один или несколько процессоров Itanium. Существует также версия Datacenter Edition. Например, эта версия устанавливается на системах Integrity компании Hewlett Packard, которые в настоящее время также появляются на рынке.

Поддержка сторонних компаний

На платформе Win64 доступны многочисленные базы данных, математические библиотеки, прикладные системы уровня предприятия, системы с открытым исходным кодом, а также целый ряд других систем. Тем не менее, каждый раз, когда планируется перенос программ на эту платформу, доступность необходимых продуктов сторонних компаний должна предварительно проверяться.

Обзор 64-разрядной архитектуры

С точки зрения программиста основная трудность при переходе от 32-разрядной модели к 64-разрядной заключается в том, что размер указателей и таких системных типов данных, как size_t и time_t, теперь может составлять 64 бита. Поэтому виртуальное адресное пространство процесса уже не ограничивается 4 Гбайт (фактически доступны приложениям только 3 Гбайт). Таким образом, перенос программ из Win32 в Win64 по существу требует лишь "удлинения" указателей, с чем связаны лишь самые минимальные последствия для пользовательских данных в модели Windows.

Необходимость в 64-битовой адресации

Возможности доступа к большим адресным пространствам требуются многим приложениям. Можно было бы привести множество примеров, аналогичных тем, которые перечислены ниже.

• **Приложения для обработки изображений.** Системы, использующие адресные пространства размером 4 Гбайт, в состоянии обеспечить лишь 20-секундное воспроизведение телевизионного изображения высокой четкости (High-Definition Television, HDTV) в реалистичных цветах.

• **Автоматизированное проектирование механических (Mechanical Computer-Aided Design, MCAD) и электронных (Electronic Computer-Aided Design, ECAD) устройств.** Для проектирования сложных сборочных узлов требуется наличие более 3 Гбайт памяти, а проектирование микросхем предъявляет к памяти несоизмеримо более высокие требования.

• **Базы данных и хранилища данных.** Использование файлов с размерами в несколько сотен Гбайт не является чем-то необычным, и возможность доступа к виртуальным адресным пространствам сопоставимых размеров значительно упрощает обработку таких файлов.

Теперь поддержка подобных запросов в отношении адресных пространств большого объема стала реальностью. Пройдет совсем немного времени, и 64-разрядные микропроцессоры станут доступными каждому, а большие объемы физической памяти при разумной стоимости будут поддерживаться на многих системах.

Потребность в 64-битовой адресации диктуется теми же факторами, которые делают столь желательными и необходимыми файлы гигантских размеров (свыше 4 Гбайт), и теперь, когда имеются достаточно мощные микропроцессоры Itanium, AMD64, а также процессоры, использующие технологию 64-разрядного расширения, вполне естественно ожидать, что Windows должна будет эволюционировать для удовлетворения этих запросов. Использование 64-разрядных ОС существенно в тех случаях, когда Windows отводится заметная роль в прикладных корпоративных и профессиональных системах.

Тем не менее, многие 32-разрядные приложения смогут работать нормально и на новой платформе, и на первом этапе для их переноса не надо будет ничего предпринимать. Для таких персональных приложений, как Microsoft Office или Adobe PageMaker, в течение некоторого времени переход к 64-битовой адресации, по-видимому, не потребуется. Следовательно, Windows будет поддерживать обратную совместимость.

Как и следовало ожидать, применение существующих 64-разрядных процессоров часто обеспечивает выигрыш в производительности, но этот выигрыш непосредственно никак не сказывается на программировании на уровне исходного кода.

Опыт UNIX

PC-системы всегда отставали от универсальных вычислительных систем (мэйнфреймов) и систем на основе UNIX в том, что касается базовых функциональных возможностей и масштабируемости. То же самое остается справедливым и в случае 64-разрядных архитектур.

• Основные поставщики UNIX-систем предоставляют 48– и 64-разрядные микропроцессоры с начала 90-х годов прошлого столетия.

• Основные поставщики UNIX-систем поддерживают 64-разрядные API на протяжении примерно того же периода времени.

• Сообщество пользователей UNIX остановилось на выборе в качестве стандарта так называемой модели LP64, отличающейся от модели P64, принятой в Win64, о чем далее еще будет говориться.

• Переходы от 32 к 64 битам всегда осуществлялись сравнительно простым, если не сказать — тривиальным образом, и можно ожидать, что то же самое будет наблюдаться и при переходе от Win32 к Win64.

Опыт перехода от 16-разрядных версий Windows к 32-разрядным

Переход от 16-разрядных версий Windows к 32-разрядным начался в начале 90-х годов прошлого столетия с появлением Windows NT, и набрал ускорение после того, как использование Windows 95 стало обычным делом. Каким бы соблазнительным ни казалось предположение о том, что нас ожидает повторение той же истории, рассматриваемые нами ситуации отличаются в нескольких аспектах.

• Windows NT и Windows 95 были первыми из широко используемых "реальных" операционных систем для PC в том смысле, что обе системы поддерживали обмен страницами по запросу, потоки, вытесняющую многозадачность и множество других возможностей, которые были описаны в главе 1.

• Хотя API Win32 значительно расширил полезное адресное пространство, что делает и Win64, усовершенствования этим не ограничивались. Неуклюжие и устаревшие, несмотря на свою популярность, модели расширенной памяти были заменены другими. Аналогичная модель расширенной памяти (не описывается в данной книге) была введена и в Windows 2000, однако общие последствия этого шага в данном случае были не столь существенными.

• В API Win32 было введено множество новых функциональных возможностей, чего нельзя сказать о Win64.

Надолго ли хватит 64 бит?

Что касается мира PC, в котором возникла Windows, то можно утверждать, что первоначальная 16-разрядная модель Intel x86 (фактическое адресное пространство которой является 20-битовым) просуществовала в течение более десяти лет, и столько же времени уже существует и 32-разрядная архитектура. Однако переход к Win64 и 64-разрядному программированию, вообще говоря, происходит медленнее, чем происходил аналогичный переход к 32 битам. Вместе с тем, в обоих случаях переход миникомпьютеров и серверов на следующий уровень осуществлялся, по крайней мере, за 10 лет до того, как это начинало происходить с PC. Тогда вполне естественно задаться вопросом о том, следует ли ожидать перехода серверов или PC к 128 битам в будущем. Берусь утверждать, что любое расширение такого рода произойдет не раньше, чем через 10 лет, исходя из одной лишь величины 64-разрядного адресного пространства.

Предсказания — вещь ненадежная, однако, воспринимая это серьезно лишь наполовину, можно напомнить о часто цитируемом законе Мура, согласно которому отношение "стоимость/производительность" уменьшается вдвое каждые 18 месяцев. В свою очередь, быстродействие и емкость устройств каждые 18 месяцев примерно удваиваются. Применяя эти рассуждения к адресному пространству, можно ожидать, что дополнительный бит адреса нам будет требоваться через каждые 18 месяцев, откуда следует, что 64-разрядная модель будет исправно служить еще целых 48 лет (то есть почти столько же времени, сколько насчитывает вся история современных компьютеров). Оправданы ли такие неформальные выводы, которые встретились мне в одном из официальных источников, покажет время, однако в прошлом запросы к ресурсам PC возрастали гораздо быстрее, чем утверждается в приведенном прогнозе.

Модель программирования Win64

В зависимости от выбора способа представления таких стандартных типов данных C, как указатели и целочисленные типы данных (long, int и short), а также от того, вводятся или не вводятся нестандартные типы данных, возможны несколько вариантов модели 64-разрядного программирования. Напомним, что в стандарте ANSI C размеры типов данных не определяются строго, хотя и требуется, чтобы размер данных типа long int был не меньше размера данных типа int, а размер данных типа int был не меньше размера данных типа short int.

Цели

Цель состоит в том, чтобы ввести единое определение Windows API (то есть, общее для Win32 и Win64), благодаря чему можно будет использовать единый базовый исходный код. Использование этого единого определения может потребовать внесения некоторые изменений в исходный код, но эти изменения должны быть сведены к минимуму.

Microsoft выбрала модель LLP64 (целые числа типа long и 64-битовые указатели), на которую обычно ссылаются просто как на модель P64. В частности, существуют следующие определения типов данных, применимые как к данным со знаком, так и к данным без знака:

- char — 8 бит, и wchar — 16 бит.
- short — 16 бит.
- int — 32 бита.
- long int — также 32 бита.
- Размер указателя любого типа, например PVOID, составляет 64 бита.

Для тех случаев, когда требуются данные строго определенного размера, предусмотрены дополнительные типы данных. Так, компилятор Microsoft различает следующие типы данных: _int16, _int32 и _int64.

Типы данных

Приведенные в этой главе таблицы взяты непосредственно из оперативной справочной системы и представляют единую модель данных Windows (Windows Uniform Data Model). Определения типов можно найти в заголовочном файле BASETSD.H, входящем в состав интегрированной среды разработки приложений Microsoft Visual Studio .NET (версия 7.0) и версию 6.0 этой системы.

Типы данных фиксированной точности

Обозначения типов данных фиксированной точности получаются из обычных обозначений типов данных Win32, таких как DWORD или LONG, добавлением суффикса размера, как показано в табл. 16.1.

Таблица 16.1. Типы данных фиксированной точности

Тип данных	Описание
DWORD32	32-битовое целое без знака
DWORD64	64-битовое целое без знака
INT32	32-битовое целое со знаком
INT64	64-битовое целое со знаком
LONG32	32-битовое целое со знаком
LONG64	64-битовое целое со знаком
UINT32	Целое типа INT32 без знака
UINT64	Целое типа INT64 без знака
ULONG32	Целое типа LONG32 без знака
ULONG64	Целое типа LONG64 без знака

Типы данных, соответствующие точности указателей

Процитируем выдержку из статьи Microsoft под названием "The New Data Types" (доступна на Web-сайте компании Microsoft): "Точность этих типов данных отражает изменение точности указателей (то есть, они становятся 32-битовыми в коде Win32 и 64-битовыми в коде Win64). Поэтому приведение указателей к одному из этих типов при выполнении арифметических операций с указателями является безопасным; при 64-битовой точности указателей размер данных этого типа будет составлять 64 бита. Также и типы данных, соответствующие счетчикам, отражают максимальный размер данных, на которые может ссылаться указатель." Таким образом, эти типы данных обеспечивают автоматическое изменение размеров целочисленных типов данных в зависимости от изменения размеров указателей, в связи с чем их иногда называют полиморфными (polymorphic data types) или *платформо-масштабируемыми* (platform scaled data types) *типами данных*. Типы данных, соответствующие точности указателей, перечислены в табл. 16.2, взятой из той же статьи.

Наиболее важным из них является тип данных SIZE_T, который уже использовался нами при описании размеров блоков памяти в главе 5.

Наконец, заметьте, что в Win64 размер данных типа HANDLE составляет 64 бита.

Таблица 16.2. Типы данных, соответствующие точности указателей

Тип данных	Описание
DWORD_PTR	Длинное целое без знака, соответствующее точности указателей.
HALF_PTR	Половина размера указателя. Используется в структурах, содержащих указатель и два поля небольшого размера.
INT_PTR	Целое со знаком, соответствующее точности указателей.
LONG_PTR	Длинное целое со знаком, соответствующее точности указателей.
SIZE_T	Максимальное количество байтов, на которые может ссылаться указатель. Используется для счетчиков, которые должны охватывать весь диапазон возможных значений указателей.
SSIZE_T	Тип SIZE_T со знаком.
UHALF_PTR	Тип HALF_PTR без знака.
UINT_PTR	Тип INT_PTR без знака.
ULONG_PTR	Тип LONG_PTR без знака.

Пример: использование указательных типов данных

Аргументом потока, передаваемым функции потока при вызове CreateThread и _beginthreadex (см. главу 7), является указатель типа PVOID. Иногда программист может захотеть передать функции потока только целочисленное значение, указывающее, например, номер потока или индекс данных в глобальной таблице. Тогда функцию потока, интерпретирующую параметр как целое без знака, можно было бы написать следующим образом:

```
DWORD WINAPI MyThreadFunc(PVOID Index_PTR) {
 DWORD_PTR Index;
 …
 Index = (DWORD_PTR)Index_PTR;
 …
}
```

Аналогичным образом, зная, что фактический аргумент является целым числом, вы могли бы записать соответствующий участок кода основного потока следующим образом:

```
…
DWORD_PTR Ix;
…
for (Ix = 0; Ix < NumThreads; Ix++) {
 hTh[Ix] = _beginthreadex(NULL, 0, MyThreadFunc, (PVOID)Ix, 0, NULL);
 …
}
```

Заметьте, что в уже существующий код вам придется внести необходимые изменения. Об этом говорится далее в разделе "Перенос существующего кода".

Предостережение

Пока, по крайней мере, в случае первоначальных вариантов реализации, не следует рассчитывать на получение доступа ко всему виртуальному адресному пространству. Размер виртуальных адресных пространств может ограничиваться такими, например, значениями, как 512 Гбайт, что соответствует ограничению данных 39 битами. Можно надеяться, что со временем, по мере эволюции процессоров и систем, указанный верхний предел увеличится.

Различия между Windows и UNIX

В Windows и UNIX выбраны различные стратегии. Большинство поставщиков UNIX-систем реализуют модель LP64, в которой размер как длинного целочисленного, так и указательного типов данных составляет 64 бита. Такую модель иногда называют моделью "I32, LP64", чтобы подчеркнуть тот факт, что размер данных типа int по-прежнему составляет 32 бита. Таким образом, различие между обеими системами в рассматриваемом нами смысле сводится к различию в размерах целых чисел типа long. К тому же, типы данных, перечисленные в таблицах 16.1 и 16.2, приняты только в Windows.

Для каждой из двух моделей имеются разумные обоснования, и в белых страницах "Aspen", фигурирующих в списке дополнительной литературы к этой главе, приводятся аргументы, объясняющие выбор, сделанный в UNIX. И все же, было бы гораздо удобнее, если бы в обеих ОС действовали одни и те же соглашения.

Перенос имеющегося программного кода

Единая модель данных Windows призвана минимизировать объем возможных изменений исходного кода, но полностью избежать необходимости внесения изменений невозможно. Например, такие функции, как HeapCreate и HeapAlloc (глава 5), которые имеют дело непосредственно с распределением памяти и размерами блоков памяти, должны использовать либо 32-битовое, либо 64-битовое поле, в зависимости от модели. Точно так же, следует всегда тщательно проверять код, чтобы выяснить, не используются ли в нем скрытые допущения относительно размеров полей и указателей.

Сначала будут описаны изменения, связанные с использованием API, которые, главным образом, касаются функций управления памятью.

Изменения, связанные с использованием API

Наиболее заметные изменения, связанные с использованием API, затрагивают функции управления памятью, введенные в главе 5. В новых определениях в полях счетчиков используется тип данных SIZE_T (см. табл. 16.2). Например, теперь прототип функции HeapAlloc будет иметь следующий вид:

```
LPVOID HeapAlloc(HANDLE hHeap, DWORD dwFlags, SIZE_T dwBytes);
```

Количество запрошенных байтов, указываемое в третьем поле, выражается данными типа SIZE_T и поэтому является 32– или 64-битовым целым без знака. Ранее данные в этом поле имели тип DWORD (всегда 32 бита).

Данные типа SIZE_T используются в соответствии с необходимостью в главе 5.

Изменения, связанные с устранением неявных допущений относительно предполагаемых размеров элементов данных

Источником многих проблем могут служить различного рода допущения относительно размеров данных. Несколько возможных примеров этого приводятся ниже.

• Тип DWORD больше нельзя использовать при указании размера блоков памяти. Вместо него следует применять типы данных SIZE_T или DWORD64.

• Необходимо тщательно проверять размеры полей, используемых взаимодействующими процессами, независимо от того, выполняются ли они на одной и той же или на разных системах. Так, в главе 12 для того, чтобы перенос программы на системы UNIX или Win64 не приводил к возникновению 64-битовых полей, поля размера в сообщениях сокетов определялись с использованием типа данных LONG32. При организации связи между процессами Windows, использующими разные модели, размеры блоков памяти не должны превышать 2 Гбайт.

• Для вычисления размера структур или типов данных следует использовать функцию sizeof; эти размеры будут разными для Win32 и Win64, если в структуру данных входят указатели или элементы данных SIZE_T. Литеральные константы размеров должны быть исключены (разумеется, этому совету было бы неплохо следовать при любых обстоятельствах).

• Необходимо проверять, не содержаться ли в объединениях, в которых указатели используются совместно с арифметическими типами данными, неявные предположения относительно размеров типов данных.

• Любое приведение типов или иное преобразование, в котором участвуют указатели и данные арифметического типа должно тщательно проверяться. Обратитесь, например, к фрагментам кода, приведенным в разделе "Пример: использование указательных типов данных".

• В частности, остерегайтесь неявного приведения 32-битовых целых к 64-битовым в вызовах функций. Нет никакой гарантии, что старшие 32 бита будут очищены, в результате чего функция может получить в качестве аргумента очень большое 64-битовое целое значение.

• Указатели выравниваются по 8-байтовым границам, в результате чего дополнение структур, обусловленное выравниванием, может увеличить размер структуры данных сверх необходимого и даже отрицательно повлиять на производительность. Перемещение указателей в начало структуры минимизирует последствия ее "разбухания".

• При выводе на печать указателей вместо спецификатора формата %x используйте спецификатор %p, а при выводе платформо-масштабируемых данных, например типа SIZE_T, — спецификатор %ld.

• Функции setjmp и longjmp должны использовать заголовочный файл <setjmp.h>, а не какие-либо допущения относительно возможного размера переменной jmp_buf, в которой должен храниться указатель.

Пример: перенос программы sortMM (программа 5.5)

В программе sortMM (программа 5.5) интенсивно используются указатели, и в частности, арифметика указателей. Подготовка этой программы к переносу, в результате чего ее можно будет компоновать и выполнять под управлением как Win32, так и Win64, иллюстрирует обычно используемые методики, а также демонстрирует, как легко невольно сделать допущения относительно размера указателя.

Использование предупреждающих сообщений компилятора

Какое бы большое значение визуальная проверка кода ни играла для обнаружения и устранении любых проблем, связанных с переходом к Win64, всегда целесообразно использовать компилятор или какое-либо иное средство, обеспечивающее просмотр кода и выдачу соответствующих предупреждающих сообщений.

Входящий в состав Microsoft Visual Studio 7.0 (.NET) компилятор C++ компании Microsoft может конфигурироваться для выдачи таких сообщений. Для этого достаточно задать в командной строке компилятора опции –Wp64 и –W3. В Visual Studio для установки этих опций потребуется выполнить следующие действия:

• Выберите страницу Project Properties (Свойства проекта).
• Откройте папку C++.
• Щелкните на кнопке General (Общие).
• Выберите вкладку Detect 64-bit Portability Issues (Определять элементы переноса в 64 разряда) и выберите вариант Yes (/Wp64) (Да (/Wp64)). Оставьте для уровня диагностики (warning level) значение 3.

После этого, в процессе сборки проекта в окне вывода будут отображаться соответствующие предупреждающие сообщения. При построении в Microsoft Visual Studio 7.0 проектов, которые находятся на Web-сайте книги, вывод предупреждающих сообщений конфигурировался именно так, как описано выше.

Код до подготовки к переносу

Большая часть программного кода sortMM.c не приводит к выдаче предупреждающих сообщений, но один участок кода на шаге 6 (см. программу 5.5) вызывает их генерацию. Соответствующий фрагмент кода вместе с номерами строк представлен в программе 16.1. Имейте в виду, что в последующих версиях этой программы номера строк могут поменяться.

Программа 16.1. sortMM.c: код до подготовки к переносув Win64, часть 1

```
…
54 LPBYTE pXFile = NULL, pX;
55 TCHAR _based (pInFile) *pIn;
…
130
131 if (!NoPrint)
132  for (iKey = 0; iKey < FsX / RSize; iKey++) {
133   WriteFile(hStdOut, &ChNewLine, TSIZE, &nWrite, NULL);
134
```

```
135   /* Приведение типа pX играет весьма важную роль, поскольку это
136      указатель на байт, а нам нужны четыре байта указателя типа _based. */
137   pIn = (TCHAR _based(pInFile)*)*(LPDWORD)pX;
138
139   while ((*pIn != CR || *(pIn + 1) != LF) && (DWORD)pIn < FsIn) {
140    WriteFile(hStdOut, pIn, TSIZE, &nWrite, NULL);
141    pIn++;
142   }
143   pX += RSize;
144  }
```

Сообщения компилятора далее приводятся, но прежде чем ознакомиться с ними, вы, возможно, захотите просмотреть код, чтобы определить возможные причины выдачи будущих предупреждающих сообщений. Не забывайте о том, что нашей целью является придание программе такого вида, который обеспечивает ее сборку и корректное выполнение как в режиме Win32, так и в режиме Win64.

Предупреждающие сообщения компилятора

Предупреждающие сообщения компилятора для этого фрагмента кода отчетливо демонстрируют неявное предположение о том, что размер указателя составляет 4 байта.

```
SORTMM.C(137) : warning C4312: 'type cast' : conversion from 'DWORD' to 'TCHAR
__based(pInFile) *' of greater size
SORTMM.C(139) : warning C4311: 'type cast' : pointer truncation from 'TCHAR
__based(pInFile) *' to 'DWORD'
```

Первое предупреждение (строка 137) является существенным. Разыменование pX после его приведения (type cast) к типу LPDWORD приводит к 32-битовому значению, которое затем назначается указателю pIn. Почти с полной уверенностью можно утверждать, что разыменование pIn вызовет исключение или приведет к возникновению иной серьезной ошибки. Правильным решением для строки 137 будет замена приведения к типу LPDWORD приведением к типу указателя LPTSTR следующим образом:

```
pIn = (TCHAR _based(pInFile)*)*(DWORD_PTR)pX;
```

Сообщение для строки 139 довольно интересно, поскольку мы сравниваем базовый указатель с размером файла. Если предположить, что файл не является гигантским, то на это предупреждение можно не обращать внимания. При этих условиях можно было бы проигнорировать и сообщение для строки 137. Однако мы учтем перспективу и приготовимся к работе с гигантскими файлами, пусть даже типом FsSize пока и является DWORD. Допуская полный диапазон значений указателя, мы должны преобразовать строку 139 следующим образом:

```
while ((*pIn != CR || *(pIn + 1) != LF) && (SIZE_T)pIn < (SIZE_T)FsIn) {
```

Второй сегмент, относящийся к шагу 2b, порождает дополнительные предупреждающие сообщения, связанным с усечением типов (pointer truncation). Соответствующий фрагмент кода представлен в программе 16.2.

Программа 16.2. sortMM: код до подготовки к переносу в Win64, часть 2

```
…
40  DWORD, KStart, KSize;
174 /* Шаг 2b: Получить первый ключ; определить размер и начальный адрес ключа.
*/
175
```

```
176  KStart = (DWORD) pInScan;
177  /* Вычисляем адрес начала поля ключа. */
178  while (*pInScan !=''&& *pInScan != '\t') pInScan++;
179  /* Вычисленный конец поля ключа. */
180
181  KSize = ((DWORD)pInScan - KStart) / TSIZE;
```

Компилятор выводит следующие предупреждающие сообщения:

```
SORTMM.C(176) : warning C4311: 'type cast' : pointer truncation from 'TCHAR
__based(pInFile) *' to 'DWORD'
SORTMM.C(181) : warning C4311: 'type cast' : pointer truncation from 'TCHAR
__based(pInFile) *' to 'DWORD'
```

Исправления сводятся к использованию DWORD_PTR в качестве типа данных в строке 40 и при приведении типов в строках 176 и 181.

Дополнительные сообщения такого же характера появляются на шаге 2c в конце функции CreateIndexFile. На Web-сайте книги находится видоизмененный файл sortMM64.c, который пригоден как для Win32, так и для Win64, и использование которого позволяет избавиться от появления предупреждающих сообщений.

Предупреждающие сообщения и необходимые изменения, касающиеся других программ

Во всех примерах проектов программ, размещенных на Web-сайте книги, опции установлены таким образом, чтобы в необходимых случаях компилятор выводил предупреждающие сообщения, касающиеся 64-битовых типов данных. Большинство программ компилировались без выдачи предупреждающих сообщений, так что никакие изменения для них не потребовались.

В то же время, программа atouEX (программа 14.2) потребовала нескольких изменений, вызванных необходимостью использования типа данных DWORD_PTR для целочисленной переменной, хранящейся в поле hEvent структуры OVERLAPPED. Это обусловлено тем, что в Win64 размер данных типа HANDLE составляет 64 бита. Необходимые изменения отмечены в листинге программы, находящемся на Web-сайте.

Некоторые предупреждения могут быть проигнорированы. Например, такие функции, как strlen(), возвращают значения типа size_t. Длина строки будет часто назначаться переменным типа DWORD, вызывая появление предупреждающих сообщений относительно "потери точности" ("loss of precision"). Во всех практических ситуациях на предупреждения такого рода можно не обращать внимания.

Резюме

64-разрядный Windows API обеспечивает возможность выполнения на Windows-платформах, использующих 64-разрядные процессоры следующего поколения, большинства корпоративных, научных и инженерных приложений с высокими запросами к ресурсам. Предприняв всего лишь нескольких мер предосторожности, можно гарантировать выполнение программ как на платформе Win32, так и на платформе Win64.

Дополнительная литература

Наилучшими информационными источниками являются библиотеки MSDN и информация, размещенные на Web-сайте компании Microsoft. Ниже приведены некоторые рекомендованные ссылки, почерпнутые на Web-сайте компании Microsoft и из других источников.

• Подготовленная специалистами компании Microsoft статья "New Data Types" доступна по адресу http://msdn.microsoft.com/library/default.asp?url=/library/en-us/win64/win64/the_new_data_types.asp. Таблицы 16.1 и 16.2 взяты именно из этой статьи.

• "Introduction to Developing Applications for the 64-bit Version of Windows" — неплохое краткое введение в различные модели программирования. Эта статья доступна по адресу http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnnetserv/html/ws03-64-bitwindevover.asp. Статья содержит также краткий обзор архитектуры процессоров Itanium, хотя Itanium — не единственные процессоры, на которых может выполняться Win64.

• Описание схемы UNIX "Aspen", подводящей прочный фундамент под модель LP64, доступно по адресу http://www.opengroup.org/public/tech/aspen/lp64_wp.htm.

• В статье "Migration Tips", доступной по адресу http://msdn.microsoft.com/library/default.asp?url=/library/en-us/win64/win64/migration_tips.asp, вы найдете хорошие советы по переносу программ с 32-разрядных на 64-разрядные платформы, а также ряд полезных ссылок. Произведя поиск в Web, вы сможете найти дополнительную информацию и рекомендации.

Если вас интересуют общие вопросы архитектуры компьютеров, обратитесь к книге [16], являющейся стандартом в этой области. Информация, касающаяся процессоров Itanium, приводится в [44].

Обширная информация, касающаяся архитектур, основанных на 64-разрядном расширении, представлена на Web-сайтах компаний Intel и AMD:

http://www.intel.com и http://www.amd.com/us-en

Упражнения

16.1. Предположим, что p1 и p2 — указатели, связанные соотношением p1 > p2, и вы хотите получить расстояние между двумя элементами, вычитая один указатель из другого. При каких условиях будет действительным выражение: (DWORD)p1 – (DWORD)p2? Следует ли заменить это выражение на (DWORD)(p1 – p2), если расстояние между элементами невелико? *Подсказка.* Примите во внимание свойства обеих комплементарных арифметик.

16.2. Избавьтесь от выдачи компилятором предупреждающих сообщений относительно 64-битовых переменных, если таковые выводятся, в других программах, например, sortBT (программа 5.1) и ThreeStage (программа 10.5), в которых интенсивно используются указатели.

16.3. Если у вас имеется доступ к системе Win64, протестируйте 64-разрядные программы. Убедитесь также, что компоновка программ в 32-разрядном режиме по-прежнему осуществляется корректно.

ПРИЛОЖЕНИЕ А
Использование примеров программ

На Web-сайте книги (http://www.awprofessional.com/titles/0321256190) находится zip-архив, который содержит исходные тексты всех примеров программ, а также соответствующие заголовочные файлы, служебные функции, файлы проектов и исполняемые файлы. Ряд программ демонстрируют дополнительные возможности и предоставляют решения отдельных упражнений, однако на Web-сайте приведены решения не для всех упражнений и представлены не все из упоминающихся в книге альтернативных вариантов реализации программ.

• Все программы тестировались под управлением Windows 2000, XP и Server 2003 на самых различных системах, от лэптопов до серверов. В необходимых случаях тестирование осуществлялось под управлением Windows 9x, хотя многие программы — особенно те, которые предлагаются на более поздних этапах изложения материала — под управлением Windows 9x и даже NT 4.0 выполняться не будут.

• Сборка и выполнение программ осуществлялись как с включенными определениями UNICODE, так и без таковых. Под управлением Windows 9x будут работать лишь те программы, в которых возможность работы с символами в кодировке UNICODE не предусмотрена.

• В подавляющем большинстве случаев компиляция программ в интегрированной среде разработки Microsoft Visual C++ версий 7.0 и 6.0 не будет сопровождаться выдачей предупреждающих сообщений, если для критерия серьезности ошибок (warning level), которые должны сопровождаться выводом диагностических сообщений компилятора, установлено значение 3. Однако существуют некоторые незначительные исключения, например, вывод сообщения "Отсутствует оператор return в основной программе" ("no return from main program") в случае использования функции ExitProcess.

• Для проектов Microsoft Visual Studio .NET и Microsoft Visual Studio C++ 6.0 предусмотрены разные каталоги, каковыми являются каталоги Projects7 и Projects6. Соответствующие исполняемые файлы программ помещаются в каталоги run7 и run6.

• В программах широко применяются функции обобщенной библиотеки C, а также такие специфические для используемых типов компиляторов ключевые слова, как __try, __except или __leave. Начиная с главы 7, важную роль в программах играют многопоточная библиотека C времени выполнения и функции _beginthreadex и _endthreadex.

• Предоставляются как файлы проектов (в их окончательной (release), а не отладочной (debug) форме), так и make-файлы. Все проекты достаточно просты, характеризуются минимальным количеством зависимостей (dependencies) и их можно быстро создать заново в любой желаемой конфигурации с получением либо отладочной, либо окончательной версии.

• Проекты для построения всех программ, за исключением статических и динамических библиотек, ориентированы на создание *консольных* приложений.

Для сборки программ можно воспользоваться также такими инструментальными средствами, распространяемыми в рамках проекта программного обеспечения с открытым исходным кодом (GNU), как компиляторы gcc и g++, входящие в состав комплекта инструментов Gnu Compiler Collection (http://gcc.gnu.org/). Читатели, заинтересованные в подобных средствах разработки, должны ознакомиться с действующим на условиях GNU проектом MinGW (http://www.mingw.org), который описывается как "совокупность свободно доступных и свободно распространяемых заголовочных файлов и библиотек импорта, специфических для Windows, объединенных с наборами инструментов GNU, что позволяет создавать программы для среды

Windows, не зависящие от динамических библиотек C времени выполнения, выпускаемых третьими сторонами". В то же время, при тестировании большинства примеров программ, приведенных в книге, я эти средства не применял, но весьма успешно использовал возможности MinGW, и мне даже удавалось выполнять межплатформенную сборку для создания исполняемых программ и DLL-библиотек Windows в Linux-системах. Более того, я имел возможность убедиться в чрезвычайно высокой эффективности систем диагностики ошибок и вывода предупреждающих сообщений компиляторов gcc и g++ при разработке 64-разрядных программ.

Структура каталогов

Основной каталог носит название WindowsSmpEd3 (от *Windows Sample Programs, Edition 3*). Для каждой главы предусмотрен отдельный подкаталог. Все заголовочные файлы находятся в каталоге Include, а в каталоге Utilities содержатся такие часто используемые функции, как ReportError или PrintStrings. Готовые проекты помещены в каталоги Projects6 и Projects7 (для Visual C++ 6.0 и 7.0 соответственно). Исполняемые программы и библиотеки DLL для всех проектов хранятся в каталогах run6 и run7. В каталоге TimeTest содержатся файлы, которые требуются для тестирования производительности программ, описанного в приложении B. Прежде чем мы приступим к описанию содержимого отдельных подкаталогов WindowsSmpEd3, необходимо вкратце рассмотреть остальное содержимое Web-сайта.

Учебные пособия (слайды)

В каталоге Overheads содержатся слайды Power Point. Слайды включены для удобства преподавателей колледжей и университетов, которые захотят воспользоваться материалом книги в курсах своих лекций. Слайды не предназначены для коммерческого использования.

Каталог Utility

В каталог Utility включены семь файлов с исходными кодами служебных функций, которые требуются для примеров программ.

1. Файл ReprtErr.c содержит функции ReportError (программа 2.2) и ReportException (программа 4.1). За исключением программ grep и wc, а также программ, приведенных в главе 1, каждая из программ, запускаемых на выполнение как процесс программами из примеров, нуждается в этом файле.

2. Файл PrintMsg.c содержит функции PrintStrings, PrintMsg и ConsolePrint (программа 2.1). Так как эти функции вызываются функцией ReprtErr.c, то этот исходный файл также требуется почти во всех проектах.

3. Файл Options.c содержит функцию, которая обрабатывает параметры командной строки и часто используется, начиная с главы 2. Включайте этот исходный файл в проект любой программы, в которой используются параметры командной строки. Соответствующий листинг приведен в программе A.7.

4. Файл Wstrings.c содержит исходный код функции wmemchr, используемой в файле Options.c. Включайте этот файл в проекты в соответствии с необходимостью. Также не исключено, что вы захотите добавить и другие функции, предназначенные для работы с обобщенными строками.

5. Файл SkipArg.c обеспечивает обработку командной строки путем пропуска одного поля аргумента при каждом вызове. Его листинг приведен в программе A.8.

6. Файл GetArgs.c содержит функцию, которая преобразует строку символов к виду argc, argv[]. Эта функция полезна при разбиении командной строки на отдельные аргументы, как это делается, например, в случае командной строки, получаемой из функции GetCommandLine, введенной в главе 6. Листинг этого файла приведен в программе A.8.

7. Файл Version.c реализует функцию DllGetVersion для библиотеки DLL, построенной из этих модулей.

Перечисленные функции можно компилировать и компоновать вместе с вызывающими

программами. Однако проще скомпоновать их отдельно в виде библиотеки, статической или динамической. В проекте Utility_3_0 эти файлы с исходными кодами используются для создания библиотеки DLL, а в проекте utulityStatic — для создания статической библиотеки.

Каталог Include

В каталоге Include описаны многочисленные файлы. Одни из них используются почти во всех примерах, другие нужны только для одной или двух программ. Перечень наиболее важных файлов приводится ниже.

1. EvryThng.h, как говорит само его название, включает почти все определения, которые требуются для обычных программ, как однопоточных, так и многопоточных. В частности, он включает файлы Envirmnt.h и Support.h. Соответствующий листинг приведен в программе A.1.
2. Exclude.h содержит определения ряда переменных препроцессора, исключающих определения, которые не требуются ни одной из программ, представленных в данной книге. Эта мера позволяет ускорить компиляцию и уменьшить размер предварительно скомпилированных заголовочных файлов.
3. Envirmnt.h содержит согласованные определения переменных препроцессора UNICODE и _UNICODE, а также определения языка и подъязыка, используемые функцией ReportError. Листинг этого файла приведен в программе A.3.
4. Support.h содержит определения многих общих функций, например, ReportError, а также ряд часто используемых символических констант. Соответствующий листинг представлен в программе A.3.
5. ClntSrvr.h используется начиная с главы 11. В нем содержатся определения структур сообщений, используемых для запросов и ответов, а также определения именованных каналов сервера и клиента, почтовых ящиков, длительностей интервалов ожидания и т.п. См. программу A.5.
6. JobMgt.h используется в программах управления задачами в конце главы 6. См. программу A.5.

Распределение программ по главам

В каталоге каждой главы содержатся все программы, относящиеся к данной главе (за исключением тех, которые были помещены в каталог Utility), а также всевозможные дополнительные программы. Соответствующий их перечень, сопровождаемый кратким описанием служебных программ, представлен ниже. В названиях некоторых программ присутствует суффикс "x"; в эти программы намеренно внесены дефекты, чтобы проиллюстрировать распространенные ошибки программирования.

> **Примечание**
>
> Имена многих программ, например, программ tail и touch, которые рассматривались в главе 7, совпадают с названиями утилит UNIX, работу которых они имитируют. Во избежание путаницы вы можете дать этим программам другие имена. Некоторые программы уже переименованы таким образом; в качестве примера можно указать программы lsW и cpW.

• cpC.c — программа 1.1.

• cpW.c — программа 1.2; cpwFA.c — ее модифицированный вариант, обеспечивающий лучшую производительность. См. результаты в приложении B.

• cpCF.c — программа 1.3.

• К числу других программ относятся UNIX-версия этой программы (cpU.c), а также программа (cpUC.c), скомпонованная с использованием библиотеки совместимости UNIX, предоставляемой Visual C++. CpwFA.c — вариант Cpw.c, обеспечивающий повышенное быстродействие за счет использования буферов большого размера, флагов последовательного просмотра и других методик, введенных в главе 2.

Глава 2

• Программы 2.1 и 2.2 находятся в упомянутом ранее каталоге Utility.

• cat.c — программа 2.3.

• atou.c — программа 2.4.

• Asc2Un.c — программа 2.5; Asc2UnFA.c и Asc2UnNB.c — ее версии, обеспечивающие лучшую производительность. Все три файла реализуют функцию Asc2Un, которая вызывается программой 2.5.

• pwd.c — программа 2.6; pwda.c — модифицированный вариант, обеспечивающий выделение необходимого объема памяти для размещения пути доступа.

• cd.c — реализация команды UNIX, осуществляющей переход к другому каталогу; эта программа не совпадает с той, которая используется в главе 2.

Глава 3

• RandomAccess.c — программа 3.1.

• lsW.c — программа 3.2. rmW.c — аналогичная программа, предназначенная для удаления файлов.

• touch.c — программа 3.3.

• getn.c — дополнительная программа для чтения записей фиксированной длины, иллюстрирующая доступ к файлам и вычисление позиции в файле.

• lsReg.c — программа 3.4.

• FileSize.c — приведенное в учебных целях решение, позволяющее определить, является ли выделенное для файла пространство разреженным.

• TestLock.c — осуществляет блокирование файла.

• tail.c — требуется как часть упражнения 3.3.

Глава 4

• Программа 4.1 включена в файл ReprtErr.c, находящийся в каталоге Utulity.

• toupper.c — программа 4.2. toupperX.c содержит преднамеренно внесенные ошибки; их устранение послужит для вас хорошим упражнением.

• Exception.c — программа 4.3, а также функция фильтра — программа 4.4.

• Ctrlc.c — программа 4.5.

Глава 5

• sortBT.c — представляет программы 5.1 и 5.2; sortBTSR.c — вариант, в котором отсутствует опция отказа от сериализации при вызове функций управления памятью, что используется для выяснения влияния этого фактора на производительность в случае простых приложений. Читатель может самостоятельно убедиться в том, что наблюдаемый эффект является весьма незначительным.

• Asc2UnMM.c — функция для программы 5.3.

• sortFL.c — программа 5.4, а sortHP.c — аналогичная программа, за исключением того, что вместо отображения файлов используется их считывание в буфер, выделяемый в памяти.

• sortMM.c — программы 5.5 и 5.6.

• atouEL.c — программа 5.7, а Asc2UnDll.c и Asc2UnmmDLL.c — исходные файлы для требуемых библиотек DLL. Asc2Unmmfl.c — еще один вариант, очищающий память при завершении выполнения, что может приводить к общему замедлению программы, но оставляет систему в безопасном состоянии.

• HeapNoSr.c — тестовая программа для количественной оценки эффекта использования флага HEAP_NO_SERIALIZE при распределении памяти. Эту программу можно использовать при выполнении упражнения 5.1.

• RandFile.c — генерирует текстовые файлы заданного размера со случайными ключами; такие файлы удобны для тестирования функций сортировки и используются для генерирования текстовых файлов большого размера во многих тестах с целью определения временных характеристик выполнения программ.

• clear.c — простая программа, выделяющая и инициализирующая память крупными блоками до наступления сбоя. Эта программа используется в перерывах между тестами синхронизации для гарантии того, что данные не кэшируются в памяти, ибо это могло бы искажать результаты тестов.

Глава 6

• grepMP.c — программа 6.1. grep.c — исходный файл программы поиска заданных символьных шаблонов, которая вызывается как процесс программой grepMP.c.

• timep.c — программа 6.2.

• JobShell.c — программа 6.3, а JobMgt.c предоставляет функции поддержки программ 6.4, 6.5 и 6.6.

• catHA.c и grepMPha.c — модифицированные версии других программ, предназначенных для демонстрации передачи дескриптора в командной строке, что используется при решении упражнения 6.2.

• version.c — получает сведения об операционной системе, включая номер ее версии.

Глава 7

• grepMT.c — программа 7.1. grepMTx.c — ее вариант с преднамеренно введенными дефектами; устранение этих дефектов предлагается в упражнении 7.7.

• sortMT.c — программа 7.2. sortMTx.c — ее вариант с преднамеренно введенными

дефектами.

• wcMT.c — решение упражнения 7.6. Имеются также две версии с преднамеренно введенными дефектами и еще одна версия, сериализующая обработку файла, которая предусмотрена для анализа временных характеристик выполнения программ.

• Во всех соответствующих проектах используется библиотека C с многопоточной поддержкой, о чем говорится в тексте главы.

Глава 8

• simplePC.c — программа 8.1.

• eventPC.c — программа 8.2.

Глава 9

• statsMX.c — программа 9.1. Ее различными вариантами являются statsNS.c, statsCS.cn statsIN.c.

• TimedMutualExclusion.c — используется для исследования временных характеристик, предлагаемого в тексте главы и упражнениях.

Глава 10

• В программе 10.1 содержится часть файла SynchObj.h, находящегося в каталоге Include. Остальная часть указанного файла содержится в программе 10.3.

• ThbObject.c — программа 10.2. testTHB — соответствующая тестовая программа.

• QueueObj.c — программа 10.4, различными вариантами которой являются программы QueueObjCS.c (используется объект CRITICAL_SECTION), QueueObjSOAW.c (используется функция SignalObjectAndWait) и версия, использующая сигнальную модель.

• ThreeStage.c — программа 10.5, для проекта которой требуются файлы Messages.c и QueueObj.c.

• QueueObjCancel.c — программа 10.6, которая работает в сочетании с программой ThreeStageCancel.c.

• MultiSem.c — вместе с тестовой программой TestMultiSem.c образует решение упражнения 10.11.

• MultiPCav.c — использует Pthreads; будет очень неплохо, если в качестве упражнения вы попытаетесь преобразовать эту программу к форме, использующей Windows API или библиотеку Pthreads с открытым исходным кодом.

Глава 11

• pipe.c — программа 11.1. Для демонстрации работы этой программы удобно использовать команду wc.c. pipeNP.c — вариант, использующий именованный канал.

• clientNP.c — программа 11.2.

• serverNP.c — программа 11.3.

• SrvrBcst.c — программа 11.4.
• LocSrver.c — программа 11.5.

Глава 12

• clientSK.c — программа 12.1.
• serverSK.c — программа 12.2.
• command. c — программа 12.3.
• SendReceiveSKST.c — программа 12.4, а serverSKST.c и clientSKST.c — соответствующие варианты программ serverSK.c и clientSK.c, незначительно модифицированных для обеспечения потокового ввода/вывода. Программу SendReceiveSKST.c следует компоновать как библиотеку DLL, которая должна неявно связываться с проектами сервера и клиента.
• SendReceiveSKHA.c — программа 12.5, а serverSKHA.c — соответствующий сервер, который использует DLL. Программа clientSKST.c будет работать с этим сервером.

Глава 13

• SimpleService.c — программа 13.2; дополнительно включает все, что требуется для программы 13.1.
• ServiceShell.c — программа 13.3.
• ServiceSK.c — это программа serverSK (программа 12.2), преобразованная в службу.

Глава 14

• atouOV. c — программа 14.1.
• atouEX.c — программа 14.2, выполняющая ту же задачу с использованием расширенного ввода/вывода.
• atouMT.c — выполняет ту же задачу с использованием многопоточного режима вместо асинхронного ввода/вывода Win32. atouMT_dh.c — неправильная версия, включенная для иллюстрации одной интересной, хотя и сопряженной с определенными рисками возможности дублирования дескрипторов.
• atouMTCP.c — использует порты завершения ввода/вывода.
• TimeBeep.c — программа 14.3.
• serverCP.c — программа 14.4, представляющая собой версию программы serverMT, в которой используются порты завершения ввода/вывода и перекрывающийся ввод/вывод.

Глава 15

• chmodW.c — программа 15.1, в которую добавлены возможности различения элементов ACE, предоставляющих и отменяющих разрешения доступа (как описано в тексте). chmodBSD.c — видоизмененный вариант программы, в котором используется функция BuildSecurityDescriptor.
• lsFP.c — программа 15.2.

• InitUnFp.c — код для программ 15.3, 15.4 и 15.5. Эти функции нужны программам 15.1 и 15.2. Кроме того, в исходном модуле содержится код, показывающий, как получить имя группы-владельца, что вам предлагается самостоятельно сделать в упражнении 15.12.

• TestFp.c — дополнительная тестовая программа, которая оказалась полезной в процессе тестирования.

• serverNP_secure.c — программа 15.6.

• JobShell_secure.c и JobMgt_secure.c — усовершенствованные варианты программ для системы управления заданиями, которая рассматривается в главе 6.

Глава 16

Для этой главы предусмотрен только один файл с исходным кодом, а именно, sortMM64.c, который представляет собой программу sortMM.c из главы 5, усовершенствованную таким образом, чтобы она могла выполняться на обеих платформах Win32 и Win64.

Листинги включаемых файлов

EvryThng.h

Программа A.1. Заголовочный файл EvryThng.h

```
    /* EvryThng.h - Включает все стандартные и пользовательские */
    /* заголовочные файлы. */
    #include "Exclude.h" /* Исключает описания, которые не требуются для примеров
программ.*/
    #include "envirmnt.h"
    #include <windows.h>
    #include <tchar.h>
    #include <stdio.h>
    #include <io.h>
    #include "support.h"
    #ifdef _MT
    #include <process.h>
    /* DWORD_PTR (целое без знака, соответствующее точности указателя)
     * используется для целых чисел, преобразуемых в дескрипторы или указатели.
     * Благодаря этому в Win64 не будут выводиться предупреждающие сообщения,
     * касающиеся взаимных преобразований 32-битовых и 64-битовых данных,
     * поскольку в Win64 дескрипторы HANDLE и указатели являются 64-битовыми
     * (см. главу 16). Этот режим активизируется только в том случае,
     * если определена символическая константа _Wp64.
     */
    #if !defined(_Wp64)
    #define DWORD_PTR DWORD
    #define LONG_PTR LONG
    #define INT_PTR INT
    #endif
```

Envirmnt.h

Программа A.2. Включаемый файл Envirmnt.h

```
    /* Envirmnt.h — Здесь определяются константы UNICODE и _MT. */
    /* Лучше и проще определить константу UNICODE в проекте. */
    /* Используйте команды меню: Project…Settings…C/C++. Затем, перейдя */
    /* в окно Project Options, добавьте в нижней части: /D "UNICODE". */
    /* Проделайте то же самое для констант _MT и _STATIC_LIB. */

    //#define UNICODE
    #undef UNICODE
    #ifdef UNICODE
    #define _UNICODE
    #endif
    #ifndef UNICODE
    #undef _UNICODE
    #endif
    //#define _STATICLIB
    /* Определите _STATICLIB, если создаете */
```

```
    /* или компонуете статическую библиотеку. */
    #define LANG_DFLT LANG_ENGLISH
    #define SUBLANG_DFLT SUBLANG_ENGLISH_US
```

Support.h

Программа А.3. Включаемый файл Support.h

```
    /* Support.h */
    /*  Содержит определения всех символических констант и распространенных служебных функций, используемых в примерах программ. */
    /* НЕСМОТРЯ НА ВКЛЮЧЕНИЕ ОПИСАНИЙ КОНСТАНТ UTILITY_EXPORTS И _STATICLIB, ИХ ЛУЧШЕ ОПРЕДЕЛЯТЬ НЕ ЗДЕСЬ, А В ПРОЕКТЕ. */
    /* Имя "UTILITY_EXPORTS" генерируется средой разработки Visual Studio, если вы создаете проект DLL с именем "Utility", но его также можно определить в командной строке C. */
    // UTILITY_3_0_EXPORTS определяется в проекте UTILITY_3_0.
    #if defined(UTILITY_3_0_EXPORTS)
    #define LIBSPEC _declspec(dllexport)
    #elif defined(__cplusplus)
    #define LIBSPEC extern "C" _declspec(dllimport)
    #else
    #define LIBSPEC _declspec(dllimport)
    #endif

    #define EMPTY _T("")
    #define YES _T("y")
    #define NO _T("n")
    #define CR 0x0D
    #define LF 0x0A
    #define TSIZE sizeof(TCHAR)

    /* Предельные значения и константы. */
    #define TYPE_FILE 1 /* Используется в ls, rm, и lsFP. */
    #define TYPE_DIR 2
    #define TYPE_DOT 3
    #define  MAX_OPTIONS  20  /*  Максимальное  количество  параметров  командной строки.*/
    #define MAX_ARG 1000 /* Максимальное количество аргументов командной строки.*/
    #define MAX_COMMAND_LINE MAX_PATH+50 /*Максимальный размер командной строки*/

    /* Часто используемые функции. */
    LIBSPEC BOOL ConsolePrompt(LPCTSTR, LPTSTR, DWORD, BOOL);
    LIBSPEC BOOL PrintStrings(HANDLE, ...);
    LIBSPEC BOOL PrintMsg(HANDLE, LPCTSTR);
    LIBSPEC VOID ReportError(LPCTSTR, DWORD, BOOL);
    LIBSPEC VOID ReportException(LPCTSTR, DWORD);
    LIBSPEC DWORD Options(int, LPCTSTR *, LPCTSTR, ...);
    LIBSPEC LPTSTR SkipArg(LPCTSTR);
    LIBSPEC VOID GetArgs(LPCTSTR, int *, LPTSTR *);

    /* Набор функций для работы с обобщенными строками в стиле string.h. Создавались по мере необходимости - первоначально была только одна функция! Реализация взята из [27]. */
    LIBSPEC LPCTSTR wmemchr(LPCTSTR, TCHAR, DWORD);
```

```
    #ifdef _UNICODE /* Это объявление уже должно было быть добавлено. */
    #define _tstrrchr wcsrchr
    #else
    #define _tstrrchr strrchr
    #endif

    #ifdef _UNICODE /* Это объявление уже должно было быть добавлено. */
    #define _memtchr wmemchr
    #else
    #define _memtchr memchr
    #endif

    /* Функции безопасности. */
    LPSECURITY_ATTRIBUTES   InitializeUnixSA(DWORD,   LPTSTR,   LPTSTR,   LPDWORD,
LPHANDLE);
    LPSECURITY_ATTRIBUTES  InitializeAccessOnlySA(DWORD,  LPTSTR,  LPTSTR,  LPDWORD,
LPHANDLE);
    DWORD ReadFilePermissions(LPTSTR, LPTSTR, LPTSTR);
    BOOL ChangeFilePermissions(DWORD, LPTSTR, LPDWORD, LPDWORD);
    /* В упрощенной форме доступны в Visual C++ Version 5.0. */
    //PSECURITY_DESCRIPTOR InitializeSD(DWORD, LPTSTR, LPTSTR, LPDWORD);

    /* Константы, которые требуются для функций безопасности. */
    #define LUSIZE 1024
    #define ACCT_NAME_SIZE LUSIZE
```

JobMgt.h

Программа A.4. Включаемый файл JobMgt.h

```
    /* JobMgt.h — Определения, необходимые для управления задачами. Глава 6. */
    /*  Код  выхода  для  программы  управления  задачами  в  случае  прекращения  их
выполнения. */

    #define JM_EXIT_CODE 0x1000

    typedef struct _JM_JOB {
     DWORD ProcessId;
     TCHAR CommandLine[MAX_PATH];
    } JM_JOB;
    #define SJM_JOB sizeof (JM_JOB)

    /* Функции управления задачами. */
    DWORD GetJobNumber(PROCESS_INFORMATION *, LPCTSTR);
    BOOL DisplayJobs(void);
    DWORD FindProcessId(DWORD);
    BOOL GetJobMgtFileName(LPTSTR);
```

ClntSrvr.h

Программа A.5. Включаемый файл ClntSrvr.h

```
    /* Определения для программ, обеспечивающих клиент-серверное взаимодействие*/
    /* Сообщения запроса и ответа. Сообщения имеют кодировку ASCII, поскольку
запрос может поступать от системы Windows 95. */
    #define MAX_RQRS_LEN 0x1000

    typedef struct {
     DWORD32 RqLen; /* Размер структуры запроса, исключая размер этого поля. */
     CHAR Command;
     BYTE Record[MAX_RQRS_LEN];
    } REQUEST;

    typedef struct {
     DWORD32 RsLen; /* Размер структуры ответа, исключая размер этого поля*/
     CHAR Status;
     BYTE Record[MAX_RQRS_LEN];
    } RESPONSE;

    #define RQ_SIZE sizeof(REQUEST)
    #define RQ_HEADER_LEN RQ_SIZE-MAX_RQRS_LEN
    #define RS_SIZE sizeof(RESPONSE)
    #define RS_HEADER_LEN RS_SIZE-MAX_RQRS_LEN

    /* Структура почтового сообщения. */
    typedef struct {
     DWORD msStatus;
     DWORD msUtilization;
     TCHAR msName[MAX_PATH];
    } MS_MESSAGE;

    #define MSM_SIZE sizeof(MS_MESSAGE)
    #define  CS_TIMEOUT  5000  /* Интервал ожидания для взаимодействия через
именованный канал и мониторинга производительности. */
    #define MAXCLIENTS 10
    #define  MAX_SERVER_TH  4  /* Максимальное количество серверных потоков для
программы serverNPCP.*/
    #define  MAX_CLIENTS_CP  16  /* Максимальное количество клиентов для программы
serverNPCP.*/

    /* Имена серверных и клиентских каналов и почтовых ящиков. */
    #define SERVER_PIPE _T("\\\\.\\PIPE\\SERVER")
    #define CLIENT_PIPE _T("\\\\.\\PIPE\\SERVER")
    #define SERVERBROADCAST _T("SrvrBcst.exe")
    #define MS_SRVNAME _T("\\\\.\\MAILSLOT\\CLS_MAILSLOT")
    #define MS_CLTNAME _T("\\\\.\\MAILSLOT\\CLS_MAILSLOT")
    #define MX_NAME _T("ClientServerMutex")
    #define SM_NAME _T("ClientServerSemaphore")

    /* Команды для функции поддержки статистики. */
    #define CS_INIT 1
    #define CS_RQSTART 2
    #define CS_RQCOMPLETE 3
    #define CS_REPORT 4
    #define CS_TERMTHD 5

    /* Функции поддержки клиент-серверной системы. */
    BOOL LocateServer(LPTSTR);
```

Exclude.h

В программе A.6 определяются многочисленные переменные, позволяющие исключить определения, которые не требуются для примеров программ, приведенных в данной книге. Этот вопрос подробно обсуждается в [30].

Программа A.6. Включаемый файл Exclude.h

```
    /* Exclude.h — Определения переменных для исключения ненужных заголовочных
файлов. За дополнительными разъяснениями обратитесь в [30]. */
    #define WIN32_LEAN_AND_MEAN
    /* Весьма эффективная мера, уменьшающая размер предварительно скомпилированного
заголовочного файла (pch) почти в два раза. */
    /* Эти определения также уменьшают размер pch-файла и уменьшают время
компиляции. Все программы в данной книге будут компилироваться с этими
определениями. От использования средств защиты можно отказаться при помощи
оператора #define NOSECURITY. */
    #define NOATOM
    #define NOCLIPBOARD
    #define NOCOMM
    #define NOCTLMGR
    #define NOCOLOR
    #define NODEFERWINDOWPOS
    #define NODESKTOP
    #define NODRAWTEXT
    #define NOEXTAPI
    #define NOGDICAPMASKS
    #define NOHELP
    #define NOICONS
    #define NOTIME
    #define NOIMM
    #define NOKANJI
    #define NOKERNEL
    #define NOKEYSTATES
    #define NOMCX
    #define NOMEMMGR
    #define NOMENUS
    #define NOMETAFILE
    #define NOMSG
    #define NONCMESSAGES
    #define NOPROFILER
    #define NORASTEROPS
    #define NORESOURCE
    #define NOSCROLL
    #define NOSERVICE
    #define NOSHOWWINDOW
    #define NOSOUND
    #define NOSYSCOMMANDS
    #define NOSYSMETRICS
    #define NOSYSPARAMS
    #define NOTEXTMETRIC
    #define NOVIRTUALKEYCODES
    #define NOWH
    #define NOWINDOWSTATION
    #define NOWINMESSAGES
    #define NOWINOFFSETS
```

```
#define NOWIMSTYLES
#define OEMRESOURCE
```

Дополнительные служебные программы

Имеются три дополнительных программы, а именно, Options, SkipArg и GetArgs, которые достаточно полезны, чтобы привести здесь их листинги. В то же время, ни одна из этих программ не привязана жестко к Win32.

Options.c

Эта функция просматривает командную строку в поиске слов, начинающихся с символа "-" (дефис), проверяет отдельные символы и устанавливает булевские параметры. Хотя эта функция и аналогична UNIX-функции getopt, она обладает меньшими возможностями.

Программа A.7. Функция Options

```
    /* Служебная функция для извлечения флагов опций из командной строки. */
    #include "EvryThng.h"
    #include <stdarg.h>

    DWORD Options(int argc, LPCTSTR argv[], LPCTSTR OptStr, ...)
    /* argv — командная строка. Параметры (опции), если они нужны, начинаются с
символа '-' в argv[1], argv[2], ….
    OptStr — текстовая строка, содержащая все возможные параметры, находящиеся во
взаимно-однозначном соответствии с адресами булевских переменных в списке
аргументов (...). Эти флаги устанавливаются тогда и только тогда, когда символ
соответствующей опции встречается в argv[1], argv[2], …Возвращаемым значением
является индекс (в argv) первого аргумента, указанного вслед за опциями. */
    {
     va_list pFlagList;
     LPBOOL pFlag;
     int iFlag = 0, iArg;
     va_start(pFlagList, OptStr);
       while ((pFlag = va_arg(pFlagList, LPBOOL)) != NULL && iFlag <
(int)_tcslen(OptStr)) {
      *pFlag = FALSE;
       for (iArg = 1; !(*pFlag) && iArg < argc && argv[iArg][0] == '-'; iArg++)
*pFlag = _memtchr(argv[iArg], OptStr[iFlag], _tcslen(argv[iArg])) != NULL;
      iFlag++;
     }
     va_end(pFlagList);
     for (iArg = 1; iArg < argc && argv[iArg][0] == '-'; iArg++);
     return iArg;
    }
```

SkipArg.c

Эта функция обрабатывает командную строку, пропуская одно поле, отделенное пробельным символом. Впервые используется в программе timep (программа 6.2).

Программа A.8. Функция SkipArg

```
    /* SkipArg.c
        Пропуск одного аргумента командной строки - символы табуляции и пробела
пропускаются. */
    #include "EvryThng.h"

    LPTSTR SkipArg(LPCTSTR targv) {
     LPTSTR p;
     p = (LPTSTR)targv;
     /* Перейти к следующему символу табуляции или пробела. */
     while (*p != '\0' && *p != TSPACE && *p != TAB) p++;
     /* Пропустить символы табуляции и пробела и перейти к следующему аргументу. */
     while (*p != '\0' && (*p == TSPACE || *p == TAB)) p++;
     return p;
    }
```

GetArgs.c

Эта функция просматривает строку, отыскивая слова, разделенные символами пробелов или табуляции, и помещает результат в массив строк, передаваемый функции. Эта функция может пригодиться для преобразования командной строки в массив argv[] и впервые используется в программе JobShell в главе 6. Функция Win32 CommandLineToArgW решает ту же задачу, но сфера ее применимости ограничивается символами Unicode.

Программа A.9. Функция GetArgs

```
    /* GetArgs. Преобразует командную строку к виду argc/argv. */
    #include "EvryThng.h"

    VOID GetArgs(LPCTSTR Command, int *pArgc, LPTSTR argstr[]) {
     int i, icm = 0;
     DWORD ic = 0;
     for (i = 0; ic < _tcslen(Command); i++) {
      while (ic < _tcslen(Command) && Command[ic] != TSPACE && Command [ic] != TAB)
{
       argstr[i][icm] = Command[ic];
       ic++;
       icm++;
      }
      argstr[i][icm] = '\0';
       while (ic < _tcslen(Command) && (Command[ic] == TSPACE || Command[ic] ==
TAB)) ic++;
      icm = 0;
     }
     if (pArgc != NULL) *pArgc = i;
     return;
    }
```

ПРИЛОЖЕНИЕ Б
Сопоставление функций Windows, UNIX и библиотеки C

В этом приложении приводятся таблицы, в которых представлены функции Windows (Win32 и Win64), описанные в основном тексте, а также сопоставимые с ними функции UNIX/Linux[36] и стандартной библиотеки ANSI C, если таковые имеются.

Таблицы расположены в порядке следования глав (некоторые таблицы объединяют данные, относящиеся к нескольким главам). В пределах каждой главы данные в таблицах отсортированы сначала в соответствии с их функциональным назначением (файловая система, управление каталогами и так далее), а затем по именам функций Windows.

В каждой из строк таблицы представлена следующая информация:

• Функциональная область (категория).

• Имя функции Windows.

• Имя соответствующей функции UNIX. В некоторых случаях существует несколько таких функций.

• Имя соответствующей функции библиотеки C, если таковая имеется.

Используемые в таблицах обозначения нуждаются в некоторых пояснениях.

• В библиотеке функций Microsoft Visual C++ содержатся некоторые функции, совместимые с UNIX. Так, функция _open является функцией библиотеки совместимости, эквивалентной UNIX-функции open. Выделение имени функции UNIX курсивом означает, что эта функция является совместимой. Символ звездочки в конце имени функции указывает на существование версии функции, ориентированной на работу с расширенными символами UNICODE. Так, существует функция _wopen.

• Программа, в которой используются только функции стандартной библиотеки C и отсутствуют вызовы функций Windows или UNIX, должны компилироваться, компоноваться и выполняться в обеих системах. В то же время, возможности такой программы в отношении работы с файлами и выполнения операций ввода/вывода будут ограниченными.

• Функция, следующая за разделительной запятой, является альтернативной версией, часто использующей другие характеристики или эмулирующей какой-то один из аспектов функции Windows.

• Разделение функций символом точки с запятой указывает на то, что эмуляция функции Windows достигается за счет последовательного использования этих функций. Так, функции CreateProcess соответствуют функции fork; exec.

• Подчеркивание имени элемента указывает на глобальную переменную, например errno.

• В некоторых случаях UNIX-эквивалент указывается в обобщенной форме с использованием такой, например, терминологии, как "функции терминального ввода/вывода" в случае Windows-функции AllocConsole. Часто приводится только соответствующий простой комментарий наподобие "Используйте библиотеку C", как это сделано в случае функции GetTempFileName. В других случаях ситуация обращается. Так, для функций управления сигналами в UNIX (функция sigaddset и подобные ей) в столбце "Windows" содержатся записи "Используйте SEH, VEH", означающие, что для обеспечения желаемого поведения программы программист должен установить структурные или векторные обработчики исключений и функции фильтров. В отличии от UNIX, группы процессов в Windows не поддерживаются, и в подобных случаях в столбце "Windows" ставится прочерк, что, впрочем, не помешало нам

эмулировать отношения между процессами при управлении заданиями в главе 6.

• Многочисленные прочерки, особенно, когда они относятся к библиотеке C, встречаются в тех случаях, когда сопоставимые функции или наборы функций отсутствуют. Именно такая ситуация наблюдается, например, для функций управления каталогами.

• В таблицах к главам 7—10 в качестве функций UNIX фигурируют функции потоков POSIX (Pthreads), хотя они и не являются частью UNIX. Кроме того, хотя во многих реализациях UNIX имеются собственные объекты синхронизации, аналогичные событиям, мьютексам и семафорам, мы не пытались отразить их в таблицах.

Как правило, более точная совместимость наблюдается для функций, фигурирующих в начальных главах книги, особенно для функций управления файлами. С переходом к более развитым функциональным возможностям различия между системами становятся все более ощутимыми, и во многих случаях эквивалентные функции библиотеки C отсутствуют. Так, модели безопасности в UNIX и Windows существенно отличаются друг от друга, и поэтому отображенные соотношения между ними являются, в лучшем случае, приближенными.

Указанные функциональные соответствия не являются точными. Между всеми тремя системами имеется множество отличий, как существенных, так и незначительных. Поэтому данные таблицы могут служить лишь ориентиром. Многие из отмеченных отличий отдельно обсуждаются в главах книги.

Главы 2 и 3: управление файлами и каталогами

Область	**Windows**	**UNIX**	**Библиотека C**	**Примечания**
Консольный ввод/вывод	AllocConsole	Терминальный ввод/вывод	-	
Консольный ввод/вывод	FreeConsole	Терминальный ввод/вывод	-	
Консольный ввод/вывод	ReadConsole	*read*	getc, scanf, gets	
Консольный ввод/вывод	SetConsoleMode	ioctl	-	
Консольный ввод/вывод	WriteConsole	*write*	putc, printf, puts	
Управление каталогами	CreateDirectory	*mkdir**	-	Создание нового каталога
Управление каталогами	FindClose	*closedir**	-	Закрытие дескриптора поиска
Управление каталогами	FindFirstFile	*opendir*, readdir**	-	Поиск первого файла, соответствующего шаблону
Управление каталогами	FindNextFile	*readdir**	-	Поиск следующих файлов, соответствующих

				шаблону
Управление каталогами	GetCurrentDirectory	*getcwd**	-	
Управление каталогами	GetFullPathName	-	-	
Управление каталогами	GetSystemDirectory	Известные пути доступа	-	
Управление каталогами	RemoveDirectory	*rmdir, unlink**	remove	
Управление каталогами	SearchPath	Используйте opendir, readdir	-	Поиск указанного файла по указанному пути
Управление каталогами	SetCurrentDirectory	*chdir**, fchdir	-	Смена рабочего каталога
Обработка ошибок	FormatMessage	strerror	perror	
Обработка ошибок	GetLastError	errno	errno	Глобальная переменная
Обработка ошибок	SetLastError	errno	errno	Глобальная переменная
Блокирование файлов	LockFile	fcntl(cmd=F_GETLK, …)	-	
Блокирование файлов	LockFileEx	fcntl(cmd=F_GETLK, …)	-	
Блокирование файлов	UnlockFile	fcntl(cmd=F_GETLK, …)	-	
Блокирование файлов	UnlockFileEx	fcntl(cmd=F_GETLK, …)	-	
Файловая система	CloseHandle (в данном случае закрытие дескриптора файла)	*close**	fclose	CloseHandle не ограничивается файлами
Файловая система	CopyFile	open; read; write; close	fopen; fread; fwrite; fclose	Дублирование файла
Файловая система	CreateFile	*open*, creat**	fopen	Открытие/ создание файла
Файловая система	DeleteFile	*unlink**	remove	Удаление файла
Файловая система	FlushFileBuffers	fsynch	fflush	Запись буферизованных данных в файл
Файловая система	GetFileAttributes	*stat*, fstat**, lstat	-	
				Заполнение

Файловая система	GetFileInformationByHandle	*stat*, fstat*,* lstat	-	структуры информацией о файле
Файловая система	GetFileSize	*stat*, fstat*,* lstat	ftell, fseek	Получение размера файла в байтах
Файловая система	GetFileTime	*stat*, fstat**, lstat	-	
Файловая система	GetFileType	*stat*, fstat*,* lstat	-	Определение типа устройства или файла
Файловая система	GetStdHandle	Используйте файловые дескрипторы 0, 1 или 2	Используйте stdin, stdout, stderr	
Файловая система	GetTempFileName	Используйте библиотеку C	tmpnam	Создание уникального имени файла
Файловая система	GetTempFileName, CreateFile	Используйте библиотеку C	tmpfile	Создание временного файла
Файловая система	GetTempPath	/temp path	-	Получение пути к каталогу для временных файлов
Файловая система	MoveFile, MoveFileEx	Используйте библиотеку C	rename	Переименование файла или каталога
Файловая система	CreateHardLink	link, *unlink**	-	Windows не поддерживает ссылки
Файловая система	-	symlink	-	Создание символической ссылки
Файловая система	-	readlink	-	Чтение имени в символической ссылке
Файловая система	Отсутствует; ReadFile возвращает 0 байт	Отсутствует; read возвращает 0 байт	feof	Количество оставшихся до конца файла байтов
Файловая система	Отсутствует; используйте многократные вызовы ReadFile	readv	Отсутствует; используйте многократные вызовы freads	Фрагментированное чтение
Файловая система	Отсутствует; используйте многократные вызовы WriteFile	writev	Отсутствует; используйте многократные вызовы fwrites	Запись со слиянием
Файловая				Чтение данных из

система	ReadFile	read	fread	файла
Файловая система	SetEndOfFile	*chsize**	-	
Файловая система	SetFileAttributes	fcntl	-	
Файловая система	SetFilePointer	*lseek*	fseek	Установка указателя файла
Файловая система	SetFilePointer (установка в 0)	lseek(0)	rewind	
Файловая система	SetFileTime	*utime**	-	
Файловая система	SetStdHandle	close, *dup*, dup2** или fcntl	freopen	dup2 или fcntl
Файловая система	WriteFile	write	fwrite	Запись данных в файл
Получение сведений о системе	GetDiskFreeSpace	-	-	
Получение сведений о системе	GetSystemInfo	getrusage	-	
Получение сведений о системе	GetVersion	uname	-	
Получение сведений о системе	GetVolumeInformation	-	-	
Получение сведений о системе	GlobalMemoryStatus	getrlimit	-	
Получение сведений о системе	Ряд предопределенных констант	sysconf, pathconf, fpathconf	-	
Дата и время	GetSystemTime	Используйте библиотеку C	time, gmtime	
Дата и время	См. программу ls (Программа 3.2)	Используйте библиотеку C	asctime	
Дата и время	CompareFileTime	Используйте библиотеку C	difftime	Сравнение "календарных" значений даты и времени
Дата и время	FileTimeToLocalFileTime, FileTimeToSystemTime	Используйте библиотеку C	localtime	

Дата и время	FileTimeToSystemTime	Используйте библиотеку C	gmtime
Дата и время	GetLocalTime	Используйте библиотеку C	time, localtime
Дата и время	См. программу touch (программа 3.3)	Используйте библиотеку C	strftime
Дата и время	SetLocalTime	-	-
Дата и время	SetSystemTime	-	-
Дата и время	Вычитание значений отметок времени	Используйте библиотеку C	difftime
Дата и время	SystemTimeToFileTime	Используйте библиотеку C	mktime

Глава 4: обработка исключений

Область	**Windows**	**UNIX**	**Библиотека C**
SEH	__try–__except	Используйте сигналы библиотеки C	Используйте сигналы библиотеки C
SEH	__try–__finally	Используйте сигналы библиотеки C	Используйте сигналы библиотеки C
SEH	AbnormalTermination	Используйте сигналы библиотеки C	Используйте сигналы библиотеки C
SEH	GetExceptionCode	Используйте сигналы библиотеки C	Используйте сигналы библиотеки C
SEH	RaiseException	Используйте сигналы библиотеки C	signal, raise
Сигналы	Используйте блок __finally	Используйте библиотеку C	atexit
Сигналы	Используйте библиотеку C или TerminateProcess	kill	raise
Сигналы	Используйте библиотеку C	Используйте библиотеку C	signal
Сигналы	Используйте SEH, VEH	sigemptyset	-
Сигналы	Используйте SEH, VEH	sigfillset	-
Сигналы	Используйте SEH, VEH	sigaddset	-
Сигналы	Используйте SEH, VEH	sigdelset	-
Сигналы	Используйте SEH, VEH	sigismember	-
Сигналы	Используйте SEH, VEH	sigprocmask	-
Сигналы	Используйте SEH, VEH	sigpending	-
Сигналы	Используйте SEH, VEH	sigaction	-
Сигналы	Используйте SEH, VEH	sigsetjmp	-

Сигналы	Используйте SEH, VEH	siglongjmp	-
Сигналы	Используйте SEH, VEH	sigsuspendf	-
Сигналы	Используйте SEH, VEH	psignal	-
Сигналы	Используйте SEH, VEH или библиотеку C	Используйте библиотеку C	abort

Примечание. Многие поставщики систем UNIX предоставляют собственные средства обработки исключений.

Глава 5: управление памятью, отображение файлов и библиотеки DLL

Область	**Windows**	**UNIX**	**Библиотека C**
Отображение файлов	CreateFileMapping	shmget	**-**
Отображение файлов	MapViewOfFile	mmap, shmat	**-**
Отображение файлов	MapViewOfFileEx	mmap, shmat	**-**
Отображение файлов	OpenFileMapping	shmget	**-**
Отображение файлов	UnmapViewOfFile	munmap, shmdt, shmctl	**-**
Управление памятью	GetProcessHeap	-	**-**
Управление памятью	GetSystemInfo	-	**-**
Управление памятью	HeapAlloc	sbrk, brk или библиотека C	malloc, calloc
Управление памятью	HeapCreate	-	-
Управление памятью	HeapDestroy	-	-
Управление памятью	HeapFree	Используйте библиотеку C	free
Управление памятью	HeapReAlloc	Используйте библиотеку C	realloc
Управление памятью	HeapSize	-	-
Разделяемая память	CloseHandle (в данном случае закрытие дескриптора объекта отображения файла)	shmctl	-
Разделяемая память	CreateFileMapping, OpenFileMapping	shmget	-

Разделяемая память	MapViewOfFile	shmat	-
Разделяемая память	UnmapViewOfFile	shmdt	-
Библиотеки DLL	LoadLibrary	dlopen	-
Библиотеки DLL	FreeLibrary	dlclose	-
Библиотеки DLL	GetProcAddress	dlsyn	-
Библиотеки DLL	DllMain	pthread_once	-

Глава 6: управление процессами

Область	**Windows**	**UNIX**	**Библиоте C**
Управление процессами	CreateProcess	fork(); *execl()*, system()	**-**
Управление процессами	ExitProcess	_exit	exit
Управление процессами	GetCommandLine	argv[]	argv[]
Управление процессами	GetCurrentProcess	*getpid**	-
Управление процессами	GetCurrentProcessId	*getpid**	-
Управление процессами	GetEnvironmentStrings	-	getenv
Управление процессами	GetEnvironmentVariable	-	getenv
Управление процессами	GetExitCodeProcess	wait, waitpid	-
Управление процессами	GetProcessTimes	times, wait3, wait4	-
Управление процессами	GetProcessWorkingSetSize	wait3, wait4	-
Управление процессами	-	*execl*,execv*,execle*,execve*,execlp*,execvp**	-

Управление процессами	-	fork, vfork	-
Управление процессами	-	getppid	-
Управление процессами	-	getgid, getegid	-
Управление процессами	-	getpgrp	-
Управление процессами	-	setpgid	-
Управление процессами	-	setsid	-
Управление процессами	-	tcgetpgrp	-
Управление процессами	-	tcsetpgrp	-
Управление процессами	OpenProcess	-	-
Управление процессами	SetEnvironmentVariable	putenv	-
Управление процессами	TerminateProcess	kill	-
Синхронизация: процесс	WaitForMultipleObjects (в данном случае ожидание дескрипторов процесса)	waitpid	-
Синхронизация: процесс	WaitForSingleObject (в данном случае ожидание дескриптора процесса)	wait, waitpid	-
Таймеры	KillTimer	alarm(0)	-
Таймеры	SetTimer	alarm	-
Таймеры	Sleep	sleep	-
Таймеры	Sleep	poll или select без указания файлового дескриптора	-

Область	Windows	UNIX/Pthreads	Примечания
Управление потоками	CreateRemoteThread	-	
TLS	TlsAlloc	pthread_key_alloc	
TLS	TlsFree	pthread_key_delete	
TLS	TlsGetValue	pthread_getspecific	
TLS	TlsSetValue	pthread_setspecific	
Управление потоками	CreateThread, _beginthreadex	pthread_create	
Управление потоками	ExitThread, _endthreadex	pthread_exit	
Управление потоками	GetCurrentThread	pthread_self	
Управление потоками	GetCurrentThreadId	-	
Управление потоками	GetExitCodeThread	pthread_yield	
Управление потоками	ResumeThread	-	
Управление потоками	SuspendThread	-	
Управление потоками	TerminateThread	pthread_cancel	pthread_cancel является более безопасной
Управление потоками	WaitForSingleObject (в данном случае ожидание дескриптора потока)	pthread_join	
Приоритет потоков	GetPriorityClass	pthread_attr_getschedpolicy, getpriority	
Приоритет потоков	GetThreadPriority	pthread_attr_getschedparam	
Приоритет потоков	SetPriorityClass	pthread_attr_setschedpolicy, setpriority, nice	
Приоритет потоков	SetThreadPriority	pthread_attr_setschedparam	

Примечание. Будучи частью всех современных систем UNIX, потоки Pthreads доступны также в системах, отличных от UNIX.

Главы 8-10: синхронизация потоков

Область	Windows	UNIX/Pthreads	Примечания
Синхронизация: критические	DeleteCriticalSection	Для эмуляции объектов критических разделов	Библиотека C в данном

разделы		используйте мьютексы.	случае не применима
Синхронизация: критические разделы	EnterCriticalSection	Некоторые системы предоставляют собственные эквиваленты.	Библиотека C в данном случае не применима
Синхронизация: критические разделы	InitializeCriticalSection		
Синхронизация: критические разделы	LeaveCriticalSection	↓	
Синхронизация: события	CloseHandle (в данном случае закрытие дескриптора события)	pthread_cond_destroy	
Синхронизация: события	CreateEvent	pthread_cond_init	
Синхронизация: события	PulseEvent	pthread_cond_signal	Вручную сбрасываемое событие
Синхронизация: события	ResetEvent	-	
Синхронизация: события	SetEvent	pthread_cond_broadcast	Автоматически сбрасываемое событие
Синхронизация: события	WaitForSingleObject (в данном случае ожидание дескриптора события)	pthread_cond_wait	
Синхронизация: события	WaitForSingleObject (в данном случае ожидание дескриптора события)	pthread_timed_wait	
Синхронизация: мьютексы	CloseHandle (в данном случае закрытие дескриптора мьютекса)	pthread_mutex_destroy	
Синхронизация: мьютексы	CreateMutex	pthread_mutex_init	
Синхронизация: мьютексы	ReleaseMutex	pthread_mutex_unlock	
Синхронизация: мьютексы	WaitForSingleObject(в данном случае ожидание дескриптора мьютекса)	pthread_mutex_lock	
Синхронизация: семафоры	CreateSemaphore	semget	
Синхронизация: семафоры	-	semctl	Непосредственная поддержка всех опций в Windows отсутствует
Синхронизация: семафоры	OpenSemaphore	semget	

Синхронизация: семафоры	ReleaseSemaphore	semop (+)		
Синхронизация: семафоры	WaitForSingleObject (в данном случае закрытие дескриптора семафора)	semop (-)		Windows может выполнять ожидание только одного счетчика

Глава 11: Взаимодействие между процессами

Область	**Windows**	**UNIX**	**Библиотека C**	**Примечания**
IPC	CallNamedPipe	-	-	CreateFile, WriteFile, ReadFile, CloseHandle
IPC	CloseHandle (pipe handle)	close, msgctl	pclose	
IPC	ConnectNamedPipe	-	-	
IPC	CreateMailslot	-	-	
IPC	CreateNamedPipe	mkfifo, msgget	-	
IPC	CreatePipe	*pipe*	popen	Не является частью стандартной библиотеки C — см. [40]
IPC	DuplicateHandle	*dup, dup2,* or fcntl	-	Или используйте стандартные имена файлов CONIN$, CONOUT$
IPC	GetNamedPipeHandleState	*stat, fstat,* lstat64	-	
IPC	GetNamedPipeInfo	*stat, fstat,* lstat	-	
IPC	ImpersonateNamedPipeClient	-	-	
IPC	PeekNamedPipe	-	-	
IPC	ReadFile (в данном случае используется дескриптор именованного канала)	read (fifo), msgsnd	-	
IPC	RevertToSelf	-	-	
IPC	SetNamedPipeHandleState	-	-	
IPC	TransactNamedPipe	-	-	WriteFile; ReadFile
IPC	WriteFile (в данном случае используется дескриптор именованного канала)	write (fifo), msgrcv	-	
Разное	GetComputerName	uname	-	

Разное	SetComputerName	-	-
Безопасность	SetNamedPipeIdentity	Используйте второй промежуточный бит каталога	-

Глава 14: асинхронный ввод/вывод

Область	Windows	UNIX	Библиотека C	Примечания
Асинхронный ввод/вывод	GetOverlappedResult	-	-	
Асинхронный ввод/вывод	ReadFileEx	-	-	Расширенный ввод/вывод с процедурой завершения
Асинхронный ввод/вывод	SleepEx	-	-	Ожидание в дежурном режиме
Асинхронный ввод/вывод	WaitForMultipleObjects (в данном случае ожидание дескрипторов файлов)	poll, select	-	
Асинхронный ввод/вывод	WaitForMultipleObjectsEx	-	-	Ожидание в дежурном режиме
Асинхронный ввод/вывод	WriteFileEx	-	-	Расширенный ввод/вывод с процедурой завершения
Асинхронный ввод/вывод	WaitForSingleObjectEx	waitpid	-	Ожидание в дежурном режиме

Глава 15: Безопасность объектов Windows

Область	Windows	UNIX	Примечания
Безопасность	AddAccessAllowedAce	chmod, fchmod	
Безопасность	AddAccessDeniedAce	chmod, fchmod	Средства защиты объектов библиотекой C не поддерживаются
Безопасность	AddAuditAce	-	
Безопасность	CreatePrivateObjectSecurity	-	
Безопасность	DeleteAce	chmod, fchmod	
Безопасность	DestroyPrivateObjectSecurity	-	
Безопасность	GetAce	*stat*, fstat*, lstat*	
Безопасность	GetAclInformation	*stat*, fstat*, lstat*	
Безопасность	GetFileSecurity	*stat*, fstat*, lstat*	
Безопасность	GetPrivateObjectSecurity	-	

Безопасность	GetSecurityDescriptorDacl	*stat*, fstat*, lstat*
Безопасность	GetUserName	getlogin
Безопасность	InitializeAcl	-
Безопасность	InitializeSecurityDescriptor	umask
Безопасность	LookupAccountName	getpwnam, getgrnam
Безопасность	LookupAccountSid	getpwuid, getuid, geteuid
Безопасность	-	getpwend, setpwent, endpwent
Безопасность	-	getgrent, setgrent, endgrent
Безопасность	-	setuid, seteuid, setreuid
Безопасность	-	setgid, setegid, setregid
Безопасность	OpenProcessToken	getgroups, setgroups, initgroups
Безопасность	SetFileSecurity	*chmod*, fchmod*
Безопасность	SetPrivateObjectSecurity	-
Безопасность	SetSecurityDescriptorDacl	*umask*
Безопасность	SetSecurityDescriptorGroup	chown, fchown, lchown
Безопасность	SetSecurityDescriptorOwner	chown, fchown, lchown
Безопасность	SetSecurityDescriptorSacl	-

ПРИЛОЖЕНИЕ В
Результаты измерения производительности

В примерах программ представлено широкое разнообразие альтернативных методик решения одних и тех же задач, как это было, например, при рассмотрении копирования файлов или преобразования текстовых файлов из кодировки ASCII в Unicode, и поэтому рассуждения о сравнительных преимуществах этих методик в отношении производительности являются вполне уместными. Однако в процессе создания приложений одних рассуждений подобного рода мало, и требуется точное знание количественных характеристик, позволяющих судить о влиянии того или иного выбора варианта реализации на производительность, а также о том, каковы в этом смысле потенциальные преимущества той или иной версии Windows, конфигурации оборудования или средств Windows, таких как потоки или асинхронный ввод/вывод. Программа timep (программа 6.2) позволяет измерять реальное (истекшее), пользовательское и системное (затраченное функциями ядра) время, необходимое для выполнения программ, и предоставляет удобный способ измерения производительности и определения ее зависимости от выбора методик и принципов программирования.

Тестовые конфигурации

Тестирование производилось путем выполнения на ряде хост-систем репрезентативного набора приложений из числа приведенных в данной книге примеров программ.

Приложения

В приведенных ниже таблицах приведены временные характеристики, полученные с использованием программы timep при выполнении тестовых программ на нескольких различных системах. Для этой цели были выбраны следующие функциональные области:

1. Копирование файлов. Показатели производительности определялись для нескольких различных методик, с помощью которых выполнялась эта операция, включая использование библиотеки C и Windows-функции CopyFile. Копирование файлов позволяет сосредоточить внимание на операциях ввода/вывода, не сопровождаемых обработкой данных.

2. Преобразование символов из кодировки ASCII в кодировку Unicode. В этой серии тестов выяснялась зависимость производительности от использования отображения файлов, буферов большого размера, флагов Windows, задающих последовательный режим обработки, и операций асинхронного ввода/вывода. Преобразование символов позволяет сосредоточить внимание на операциях ввода/вывода, сопровождаемых незначительной обработкой данных в процессе их перемещения из одного буфера в другой.

3. Поиск заданных текстовых шаблонов. Эта серия тестов проводилась с использованием программы grep в ее многопроцессорной и многопоточной формах. Тестировалась также простая последовательная обработка файлов, которая по своей производительности на однопроцессорных системах оказалась вполне конкурентоспособной по отношению к двум другим методикам. При поиске образцов увеличивается доля обработки данных в расходовании процессорного времени и уменьшается доля операций вывода.

4. Сортировка файлов. Эта серия тестов демонстрирует, какое влияние на производительность оказывает использование отображения файлов и обработка файлов в памяти, а также переход к многопоточному режиму выполнения. При сортировке основная доля времени, по крайней мере, в случае крупных файлов, приходится не на файловые операции ввода/вывода, а на обработку данных процессором.

5. Многопоточная система "производитель/потребитель". Эта серия тестов позволила исследовать влияние на производительность различных методов синхронизации, используемых для реализации системы с многопоточной очередизацией, что дало возможность оценить сравнительные достоинства и недостатки обсуждавшихся в главах 8-10 моделей, основанных на применении объектов CRITICAL_SECTION, мьютексов и функции SignalObjectAndWait, а также сигнальной и широковещательной моделей переменных условий.

Программы для всех перечисленных приложений компоновались с использованием Microsoft Visual C++ 7.0 и 6.0 в виде окончательных (release), а не отладочных (debug) версий. Выполнение отладочных версий программ привело бы к заметному искажению картины показателей производительности. Специально проведенные тесты с интенсивной загрузкой процессора показали, что в этом случае доля дополнительных накладных расходов может достигать 80%, не говоря уже о том, что размеры отладочных исполняемых модулей превышают размеры модулей окончательных версий в два-три раза.

Хост-системы

Показатели производительности измерялись на четырех современных (по состоянию на 2004 год) системах, характеризующихся широким разнообразием конфигураций ЦП, памяти и ОС. Во всех случаях использовалась файловая система NTFS. В некоторых случаях приводятся также данные, полученные на устаревших системах.

1. Лэптоп, процессор Pentium с частотой 1 ГГц, ОС Windows 2000 Professional.
2. Лэптоп, процессор Intel Celeron с частотой 2 ГГц, ОС Windows XP.
3. PC с процессором Pentium, ОС Windows 2000.
4. Четырехпроцессорная система Windows 2000 Server с ОС NT 5.0. В качестве процессоров были использованы четыре процессора Intel Xeon с частотой 1,8 ГГц. Эта система позволила исследовать, насколько увеличиваются показатели производительности в результате применения нескольких высокопроизводительных процессоров.

В примерах, связанных с обработкой файлов, представлены также результаты, полученные на устаревшей NT-системе на основе PC, в которой использовался процессор Pentium III с частотой 500 МГц, что позволило сравнить между собой показатели производительности для файловых систем FAT и NTFS, хотя файловая система FAT уже не так распространена, как прежде. Во всех случаях файловые системы были заполнены не более чем на 50% и характеризовались лишь незначительной фрагментацией.

Во время проведения тестов никакие другие задачи, кроме тестовых программ, на системах не выполнялись. Неплохим показателем относительного быстродействия процессоров могут служить результаты, полученные, в частности, при выполнении таких программ, как программы сортировки, интенсивно использующие процессор.

Измерение производительности

Каждое приложение выполнялось на хост-системе по пять раз. Перед каждым запуском приложения физическая память очищалась, чтобы исключить повышение показателей производительности за счет файлов и программ, кэшированных в памяти или файлах подкачки. В представленных в последующих разделах таблицах приведены усредненные данные. Время измеряется в секундах.

Каждая таблица сопровождается комментариями. Вполне очевидно, что делать какие-либо обобщенные выводы в отношении производительности довольно рискованно, поскольку на временные характеристики производительности программ оказывают влияние самые разнообразные факторы, включая свойства самих программ. В то же время, такие тесты позволяют выявлять некоторые закономерности и исследовать изменение производительности в зависимости от использования тех или иных файловых и операционных систем или тех или иных методик программирования. Кроме того, необходимо иметь в виду, что в процессе тестирования измеряется время, прошедшее с момента начала выполнения программы до ее завершения, но не учитывается время, необходимое системе для сброса содержимого буферов на диск. Наконец, при проведении тестов никаких попыток воспользоваться специфическими возможностями или параметрами системы, как то использование разделенных на полосы дисков, варьирование размеров блоков данных при обмене с дисками, разбиение дисков на несколько разделов и тому подобное, не предпринималось.

Монитор производительности Windows, доступный через пиктограмму Administrative Tools (Администрирование), находящуюся в панели управления, отображает данные о работе процессора, ядра, пользовательских и других процессов в графическом виде. Это средство предоставляет отличные дополнительные возможности для изучения свойств программ помимо тех, которые обеспечиваются приведенными здесь результатами измерений.

Полученные результаты указывают на варьирование производительности в широких пределах в зависимости от используемых ЦП, файловой системы, дисковой конфигурации, особенностей программы, а также множества других факторов. Все программы, применяемые для проведения необходимых измерений, находятся на Web-сайте книги, что дает вам возможность самостоятельно выполнить эти тесты на своей системе.

Копирование файлов

Пять различных вариантов реализации программ копирования файлов использовались для копирования файла размером 25,6 Мбайт (400 000 записей размером 64 байта каждая, сгенерированных с помощью программы RandFile из главы 5). В первых двух столбцах табл. В.1 представлены результаты, полученные на лэптопе (LT) устаревшей модели, в котором установлен процессор Pentium с частотой 500 МГц, что позволяет сопоставить между собой показатели производительности в случае файловых систем NTFS и FAT.

1. Программа cpC (программа 1.1) использует библиотеку C. В этом тесте измеряется эффект варианта реализации, выполняющегося поверх Windows, хотя библиотека и предоставляет возможность использования эффективной буферизации или применения других методик.

2. Программа cpW (программа 1.2) является реализацией, в которой используются непосредственно средства Windows с буфером небольшого размера (256 байт).

3. Программа cpwFA — "быстрый" вариант реализации с использованием буфера большого

размера (8192 байта, что кратно размеру сектора диска на всех хост-системах) и флагов Windows, задающих последовательный режим обработки как для входных, так и для выходных файлов.

4. Программа cpCF (программа 1.3) использует функцию Windows CopyFile для выяснения того, является ли реализация, ограничивающаяся единственным системным вызовом, более эффективной по сравнению с другими методиками.

5. Программа cpUC (программа 1.1) — реализация в стиле UNIX, использующая буфер небольшого размера (аналогично программе cpW). Эта программа была незначительно изменена, чтобы обеспечить возможность использования библиотеки совместимости с UNIX, входящей в состав Visual C++.

В то время как приведенные результаты представляют величины, усредненные по пяти тестовым запускам, сами значения истекшего времени могут меняться в широких пределах. Так, для программы cpUC (последний ряд) среднее значение истекшего времени в третьем столбце данных (Pentium LT, W2000) составило 7,77 секунды, тогда как минимальное и максимальное значения составляли соответственно 1,87 и 11,71 секунды. Такая широкая вариация значений была типичной почти во всех случаях и на всех системах.

Комментарии

1. Применение файловой системы NTFS вовсе не гарантирует лучшую по сравнению с системой FAT производительность. Более того, иногда более быстрой оказывается именно FAT, в чем можно убедиться, сравнивая данные, приведенные в столбцах 1 и 2.

2. Библиотеки совместимости C и UNIX обеспечивают сопоставимую производительность, которая во многих случаях превосходит производительность простейших вариантов реализации, использующих средства Windows.

3. Процессорное время, потребляемое как функциями ядра ("Системное время"), так и пользовательскими функциями ("Пользовательское время"), является минимальным. Следовательно, быстродействие процессора оказывает лишь самое незначительное влияние на производительность, оцениваемую по истекшему времени.

4. Как и следовало ожидать, высокопроизводительные серверные SMP-системы действительно обеспечивают гораздо более быструю обработку файлов, чем лэптопы и PC. Дополнительные тесты, выполненные в системе Windows Server 2003, обладающей еще более высоким быстродействием, показали еще лучшие результаты (здесь они не представлены), причем значения истекшего времени оказались примерно в два раза меньшими по сравнению со значениями, приведенными в крайнем справа столбце таблицы.

5. Использование буферов большого размера, флагов последовательной обработки или функции CopyFile не обеспечивают систематического или существенного выигрыша в производительности, оцениваемой по истекшему времени. Вместе с тем, весьма небольшие значения показателей пользовательского времени для программ cpwFA и cpCF представляют интерес, и этим можно воспользоваться в некоторых ситуациях, даже если показатели истекшего времени при этом не улучшатся. Одна из систем, а именно, лэптоп с процессором Pentium, не подходит под данное обобщение. Как ранее уже отмечалось, процессорное время составляет лишь небольшую долю истекшего времени.

6. Показатели истекшего времени являются в высшей степени переменчивыми, причем в некоторых случаях отношение результатов, полученных в идентичных тестах, которые выполнялись в идентичных условиях, достигало значения 10:1.

Таблица B.1. Показатели производительности программ копирования файлов

	ЦП	Pentium III	Pentium III	Pentium LT	Celeron LT	Xeon	4×Xeon
	ОС	**W2000**	**W2000**	**W2000**	**XP**	**W2000**	**W2000**
	Файловая система	**FAT**	**NTFS**	**NTFS**	**NTFS**	**NTFS**	**NTFS**
	Реальное время	8,62	14,69	12,75	7,23	6,03	2,67
cpC	Пользовательское время	0,12	0,12	0,10	0,10	0,09	0,06
	Системное время	0,24	0,52	1,39	0,39	0,25	0,36
	Реальное время	8,49	13,35	25,48	7,10	8,94	2,95
cpW	Пользовательское время	0,13	0,12	0,06	0,04	0,04	0,13
	Системное время	0,88	1,37	4,61	0,62	0,56	0,13
	Реальное время	8,35	12,59	7,35	8,25	9,10	2,36
cpwFA	Пользовательское время	0,01	0,02	0,03	0,01	0,01	0,02
	Системное время	0,40	0,50	0,82	0,29	0,20	0,19
	Реальное время	8,00	11,69	2,57	6,50	7,62	2,97
cpCF	Пользовательское время	0,02	0,01	0,02	0,02	0,01	0,02
	Системное время	0,19	0,25	0,53	0,19	0,12	0,17
	Реальное время	7,84	13,14	21,01	9,98	7,77	3,53
cpUC	Пользовательское время	0,72	0,66	0,47	0,34	0,34	0,42
	Системное время	0,40	0,67	3,12	0,34	0,36	0,41

Преобразование символов из кодировки ASCII в Unicode

Измерения выполнялись для восьми программ, каждая из которых преобразовывала файл размером 12,8 Мбайт в файл размером 25,6 Мбайт. Соответствующие результаты представлены в табл. B.2.

1. Программа atou (программа 2.4) сопоставима с программой cpW, использующей буфер небольшого размера.

2. Программа atouSS — первый из "быстрых" вариантов реализации, основанных на программе atou. В нем применяются флаги последовательного режима обработки и буфер небольшого размера. Эта, а также две следующие программы сгенерированы на основе одного и того же проекта, atouLBSS, но с определением разных комбинаций макросов.

3. Программа atouLB использует буфер большого размера (8192 байта), но не использует флаги последовательного режима обработки.

4. Программа atouLSFP использует буфер большого размера и флаги последовательного режима обработки, но кроме этого предварительно устанавливает требуемый размер выходного файла. Эта мера продемонстрировала свою высокую эффективность.

5. Программа atouMM использует отображение файлов для операций файлового ввода/вывода и вызывает функции, листинг которых приведен в программе 5.3.

6. Программа atouMT представляет собой многопоточную реализацию приведенной в главе 14 программы, основанной на схеме множественной буферизации без применения асинхронного ввода/вывода.

7. Программа atouOV (программа 14.1) использует перекрывающийся ввод/вывод и не

может выполняться на двух системах семейства Windows 9x.

8. Программа atouEX (программа 14.2) использует перекрывающийся ввод/вывод и не будет выполняться на двух системах семейства Windows 9x.

Комментарии

1. Результаты показывают, что применение буферов увеличенного размера и флагов последовательной обработки (а возможно, и сочетания этих факторов) обеспечивает некоторый выигрыш в производительности.

2. Предварительная установка размера выходного файла (atouLSFP) очень эффективна и приводит к резкому повышению производительности на всех однопроцессорных системах. В то же время, преимущества SMP-систем оказались весьма незначительными. Эту же методику можно было применить и в предыдущих примерах копирования файлов.

3. В этих примерах процессорное время составляет лишь незначительную долю общего времени.

4. Помимо того, что использование перекрывающегося ввода/вывода ограничивается системами Windows NT и его трудно программировать, он обеспечивает очень низкую производительность. Заметьте, что основная доля общего времени приходится не на пользовательское или системное время, а на реальное время. Создается впечатление, что в случае NT4 система испытывает трудности с планированием доступа к диску, и это препятствие нельзя было устранить путем изменения размера буфера (как большую, так и в меньшую сторону) до тех пор, пока не были использованы буферы размером 65 Кбайт. В NT5 эта проблема не возникает.

5. Ни расширенный ввод/вывод, ни многопоточный режим не обеспечивают сколько-нибудь заметного повышения производительности.

6. Использование отображения файлов в операциях ввода/вывода способно увеличивать производительность, обеспечивая ее повышение примерно на 30% по сравнению с остальными методами. Результаты для SMP-сервера оказались еще лучшими.

Таблица B.2. Показатели производительности программ преобразования символов из кодировки ASCII в Unicode

	ЦП	**Pentium III**	**Pentium III**	**Pentium LT**	**Celeron LT**	**Xeon**	**4×Xeon**
	ОС	**W2000**	**W2000**	**W2000**	**XP**	**W2000**	**W2000**
	Файловая система	**FAT**	**NTFS**	**NTFS**	**NTFS**	**NTFS**	**NTFS**
atou	Реальное время	3,24	7,16	33,53	6,27	5,77	2,77
	Пользовательское время	0,31	0,33	0,01	0,06	0,06	0,08
	Системное время	0,46	0,72	3,55	0,54	0,63	0,63
atouSS	Реальное время	3,77	6,21	43,53	10,12	5,68	2,48
	Пользовательское время	0,20	0,23	0,11	0,07	0,04	0,14
	Системное время	0,52	0,81	3,17	0,04	0,35	0,81
atouLB	Реальное время	4,38	6,41	28,51	5,95	4,75	2,47
	Пользовательское время	0,10	0,07	0,05	0,03	0,03	0,08
	Системное время	0,26	0,34	0,63	0,19	0,21	0,187

	Реальное время	-	-	5,17	1,38	1,28	2,03
atouLSFP	Пользовательское время	-	-	0,07	0,05	0,09	0,06
	Системное время	-	-	0,61	0,16	0,10	0,11
	Реальное время	4,35	2,75	3,46	3,90	3,74	0,77
atouMM	Пользовательское время	0,27	0,29	0,09	0,07	0,05	0,14
	Системное время	0,19	0,19	0,16	0,14	0,10	0,09
	Реальное время	4,84	6,18	5,83	6,61	5,99	3,55
atouMT	Пользовательское время	0,14	0,15	0,26	0,04	0,06	0,02
	Системное время	0,45	0,46	0,66	0,33	0,15	0,31
	Реальное время	9,54	8,85	32,42	6,84	5,63	3,17
atouOV	Пользовательское время	0,14	0,12	0,21	0,06	0,06	0,06
	Системное время	0,24	0,23	0,42	0,18	0,21	0,17
	Реальное время	5,67	5,92	30,65	6,50	5,19	2,64
atouEX	Пользовательское время	1,10	1,50	0,29	0,35	0,41	0,64
	Системное время	1,19	1,74	0,77	0,69	0,59	1,91

Поиск заданных комбинаций символов

Тестирование производительности путем выполнения поиска определенных текстовых шаблонов в содержимом файлов производилось с использованием трех различных методов, что позволило оценить сравнительную эффективность многопоточного и многопроцессного режимов, а также простой последовательной обработки файлов (см. табл. В.3).

1. Программа grepMP (программа 6.1) использует параллельные процессы, каждый из которых обрабатывает отдельный файл. Результаты измерений системного и пользовательского времени не приводятся, поскольку программа timep позволяет хронометрировать лишь родительские процессы.

2. Программа grepMT (программа 7.1) использует параллельные потоки.

3. Программа grepSQ — это пакетный файл DOS, обеспечивающий выполнение поиска шаблонов по очереди в каждом из файлов. В этом случае также приводятся только результаты, относящиеся к реальному времени.

В этом тесте использовались 20 файлов с размерами в пределах от нескольких Кбайт до 1 Мбайт.

Комментарии

1. В большинстве случаев все три методики приводят к близким результатам на однопроцессорных системах. Исключением является лэптоп с процессором Pentium, для которого версия grepMP систематически оказывалась самой медленной.

2. Многопоточный режим обладает лишь незначительными преимуществами по сравнению с многопроцессным даже на однопроцессорных системах.

3. Показатели пользовательского и системного времени имеют ощутимо заметные значения лишь в случае многопоточных версий

4. SMP-системы демонстрируют выигрыш в производительности, который достигается и при использовании многопоточного режима или нескольких однопоточных процессов.

Заметьте, что общее пользовательское время превышает реальное время, поскольку характеризует одновременно все четыре процесса.

5. Тот факт, что последовательная обработка файлов приводит на однопроцессорных системам к аналогичным результатам, говорит о том, что простейшее решение нередко оказывается и самым лучшим.

Таблица В.3. Показатели производительности программ поисказаданных комбинаций символов

	ЦП	**Pentium LT**	**Celeron LT**	**Xeon**	**4×Xeon**
	ОС	**W2000**	**XP**	**W2000**	**W2000**
	Файловая система	**NTFS**	**NTFS**	**NTFS**	**NTFS**
grepMP	Реальное время	14,72	3,95	10,58	0,63
	Пользовательское время	-	-	-	-
	Системное время	-	-	-	-
grepMT	Реальное время	7,08	3,61	8,09	0,73
	Пользовательское время	0,30	0,41	0,27	2,23
	Системное время	0,09	0,47	0,13	0,28
grepSQ	Реальное время	6,71	3,86	6,71	0,97
	Пользовательское время	-	-	-	-
	Системное время	-	-	-	-

Сортировка файлов

Для тестирования четырех вариантов реализации программ сортировки из главы 5 использовался целевой файл, состоящий из 100 000 записей размером 64 байта каждая (всего 6,4 Мбайт). Вывод отсортированного файла во всех случаях подавлялся, чтобы можно было оценивать только время, необходимое для выполнения собственно сортировки. После этого тестировалась многопоточная сортировка (программа 7.2) файла размером 25 Мбайт, состоящего из 400 000 записей размером 64 байта каждая, с использованием одной, двух и четырех потоков. В каждом отдельном запуске использовался отдельный файл, генерируемый программой RandFile, которая находится в каталоге главы 5. Результаты для разных запусков заметно различались между собой.

1. Программа sortBT (программа 5.1) создает бинарное дерево поиска, требующее выделения минимального объема памяти под каждую запись. Эта программа интенсивно использует процессор.

2. Программа sortFL (программа 5.4) создает отображение файла перед тем, как использовать программу qsort. Тестировалась также программа sortFLSR (доступ к куче подвергался сериализации), однако существенных отличий от предыдущего варианта замечено не было.

3. Текст программы sortHP в книге не приводился. Эта программа предварительно распределяет буфер для файла, а затем сортирует файл, считанный в этот буфер, а не его отображение, как программа sortFL.

4. Программа sortMM (программа 5.5) создает постоянно существующий индексный файл.

5. Программа sortMT (программа 7.2) реализует многопоточную сортировку слиянием.

Результаты представлены в строках sortMT1, sortMT2 и sortMT4 в соответствии с количеством параллельных потоков. Результаты могут значительно меняться в зависимости от характера сортируемых данных, хотя размер и случайный характер распределения значений данных сглаживают эти различия, что, как правило, характерно для базового алгоритма быстрой сортировки, который использован для реализации функции qsort библиотеки C.

Комментарии

1. Реализация, использующая алгоритм бинарного дерева (программа sortBT), интенсивно использует процессор; кроме того, память в ней распределяется отдельно для каждой записи.
2. Применение отображения файлов и чтение файла в предварительно выделенный буфер обеспечивают примерно одинаковую производительность, но в этих тестах отображение файлов ничем особенным себя не проявило, а в некоторых случаях даже значительно ухудшало результаты. Вместе с тем, в ряде случаев как sortFL, так и sortHP обеспечивали превосходные результаты.
3. Суммарное пользовательское и системное время иногда превышает истекшее время, даже если используется только один поток.
4. Программа sortMT демонстрирует возможности SMP-систем. В некоторых случаях использование дополнительных потоков приводило к повышению производительности и на однопроцессорных системах.

Таблица B.4. Показатели производительности программ сортировки файлов

	ЦП	**Pentium LT**	**Celeron LT**	**Xeon**	**4×Xeon**
	ОС	**W2000**	**XP**	**W2000**	**W2000**
	Файловая система	**NTFS**	**NTFS**	**NTFS**	**NTFS**
sortBT	Реальное время	-	9,61	-	-
	Пользовательское время	-	1,84	-	-
	Системное время	-	7,44	-	-
sortFL	Реальное время	11,15	0,78	1,74	5,38
	Пользовательское время	4,81	0,41	0,26	5,19
	Системное время	0,15	0,09	0,09	0,02
sortHP	Реальное время	1,76	0,34	1,52	1,30
	Пользовательское время	1,62	0,22	0,15	1,28
	Системное время	0,11	0,05	0,03	0,04
sortMM	Реальное время	0,99	1,44	1,92	1,39
	Пользовательское время	0,31	0,18	0,15	0,38
	Системное время	0,68	0,47	0,36	1,03
sortMT1	Реальное время	3,18	3,58	6,80	0,14
	Пользовательское время	0,01	0,95	0,01	0,05
	Системное время	0,46	0,16	0,16	0,11
sortMT2	Реальное время	2,10	1,22	6,70	0,13
	Пользовательское время	0,01	1,05	0,01	0,02

	Системное время	0,40	0,16	0,16	0,13
	Реальное время	2,20	1,49	6,22	0,13
sortMT4	Пользовательское время	0,01	1,18	0,01	0,12
	Системное время	0,16	0,15	0,16	0,09

Множество потоков, соревнующихся между собой за обладание единственным ресурсом

Целью этой серии тестов являлось сравнение эффективности различных стратегий реализации функций управления очередями программы 10.4 с использованием программы 10.5 (трехступенчатый конвейер) в качестве тестового приложения. Тесты выполнялись на четырехпроцессорной (Intel Xeon, 1 ГГц) системе Windows 2000 Server с организацией 1, 2, 4, 8, 16, 32 и 64 потоков, но во всех семи случаях каждого потока поручалось выполнение 1000 единиц работы. В идеальном случае можно было бы ожидать линейного увеличения реального времени с увеличением количества потоков, но соревновательность между потоками за право владения единственным мьютексом (или объектом CS) может приводить к нелинейному снижению этого показателя). Обратите внимание, что эти тесты не затрагивают файловую систему.

Использовались шесть различных стратегий реализации, результаты применения которых представлены в отдельных столбцах табл. В.5.

Таблица В.5. Производительность многопоточных реализаций на четырехпроцессорном сервере

К-во потоков		**Широковещат.модель Mtx, Evt T/O 5мс**	**Широковещат.модель CritSec, Evt T/O 25мс**	**Широковещат. модель Mtx, Evt SigObjWait**	**Ci мо M T/ не**
	Реальное время	0,03	0,03	0,05	0,(
1	Пользовательское время	0,03	0,06	0,03	0,(
	Системное время	0,06	0,02	0,09	0,(
	Реальное время	0,14	0,27	0,09	0,(
2	Пользовательское время	0,13	0,05	0,14	0,1
	Системное время	0,11	0,06	0,16	0,(
	Реальное время	0,39	0,59	0,23	0,1
4	Пользовательское время	0,18	0,17	0,22	0,2
	Системное время	0,30	0,22	0,41	0,3
	Реальное время	0,83	0,92	0,73	0,3
8	Пользовательское время	0,34	0,36	0,55	0,5

	Системное время	0,98	1,00	1,00	0,(
	Реальное время	2,42	2,30	2,38	0,1
16	Пользовательское время	1,17	1,31	1,22	0,8
	Системное время	3,69	3,05	3,39	1,4
	Реальное время	7,56	7,50	7,98	1,5
32	Пользовательское время	3,33	3,73	2,56	1,1
	Системное время	12,52	10,72	11,03	3,1
	Реальное время	27,72	26,23	29,31	3,1
64	Пользовательское время	7,89	10,75	7,22	3,1
	Системное время	46,70	40,33	36,67	6,2

В комментариях, помещенных вслед за программой 10.4, обсуждаются результаты и разъясняются преимущества различных реализаций, а здесь мы лишь отметим, что результаты для сигнальной модели изменяются пропорционально изменению количества потоков, тогда как в случае широковещательной модели, особенно для вариантов с 32 и 64 потоками, это не так. Можно также видеть, что в случае широковещательной модели система потребляет значительную долю процессорного времени, ибо выполняются, вычисляют предикат и осуществляют немедленный возврат в состояние ожидания множество потоков.

1. Широковещательная модель, мьютекс (Mtx), событие (Evt), раздельные вызовы функций освобождения и ожидания. Конечный период ожидания (Time-out, TO) устанавливался равным 5 миллисекундам, что являлось оптимальным значением для 16-поточного варианта.

2. Широковещательная модель, объект CRITICAL_SECTION (CritSec), событие, раздельные вызовы функций освобождения и ожидания. Настраиваемый период ожидания устанавливался равным 5 миллисекундам, что являлось оптимальным значением для 16-поточного варианта.

3. Широковещательная модель, мьютекс, событие, атомарный вызов SignalObjectAndWait (SigObjWait).

4. Сигнальная модель, мьютекс, событие, раздельные вызовы функций освобождения и ожидания.

5. Сигнальная модель, объект CRITICAL_SECTION, событие, раздельные вызовы функций освобождения и ожидания.

6. Сигнальная модель, мьютекс, событие, атомарный вызов SignalObjectAndWait.

Выполнение тестов

На Web-сайте книги в каталоге TimeTest находятся пакетные файлы, с помощью которых вы сможете запускать тесты как под управлением Windows 2000/NT, так и под управлением Windows 9x:

- `cpTIME.bat`
- `cpTIME.bat`
- `atouTIME.bat`
- `grepTIME.bat`
- `sortTIME.bat`
- `threeST.bat`

Для всех тестов, кроме тестов последней серии, текстовые ASCII-файлы большого размера создавались с помощью программы RandFile.

Библиография

1. Beveridge, Jim, and Wiener, Robert. *Multithreading Applications in Win32: The Complete Guide to Threads,* Addison-Wesley, Reading, MA, 1997. ISBN: 0-201-44234-5.

2. Bott, Ed, and Siechert, Carl. *Microsoft Windows Security Inside Out for Windows XP and Windows 2000,* Microsoft Press, Redmond, WA, 2002. ISBN: 0-735-61632-9.

3. Box, Don. *Essential COM,* Addison-Wesley, Reading, MA, 1998. ISBN: 0-201-63446-5.

4. Box, Don (editor), et al. *Effective COM: 50 Ways to Improve Your COM and MTS Based Applications,* Addison-Wesley, Reading, MA, 1999. ISBN: 0-20-1-37968-6.

5. Brain, Marshall, and Reeves, Ron. *Win32 System Services: The Heart of Windows 98 and Windows 2000, Third Edition,* Prentice Hall, Englewood Cliffs, NJ, 2000. ISBN: 0-13-022557-6.

6. Butenhof, David. *Programming with POSIX Threads,* Addison-Wesley, Reading, MA, 1997. ISBN: 0-201-63392-2.

7. Cohen, Aaron, Woodring, Mike, and Petrusha, Ronald. *Win32 Multithreaded Programming,* O'Reilley & Associates, Sebastopol, CA, 1998. ISBN: 1-565-92296-4.

8. Comer, Douglas E., and Stevens, David L. *Internetworking with TCP/IP, Volume III: Client-Server Programming and Applications, Windows Sockets Version,* Prentice Hall, Upper Saddle River, NJ, 1997. ISBN: 0-13-848714-6.

9. Custer, Helen. *Inside Windows NT,* Microsoft Press, Redmond, WA, 1993. ISBN: 155615-481-X. *Second edition by David Solomon replaces this book, which in turn is replaced by Solomon and Russinovich (both in this bibliography).*

10. Custer, Helen. *Inside the Windows NT File System,* Microsoft Press, Redmond, WA, 1994. ISBN: 155615-660-X.

11. Department of Defense. *U.S. Department of Defense Trusted Computer System Evaluation Criteria,* formerly known as *DoD Trusted Computer System Evaluation Criteria*, DoD 5200.28-STD, DoD Computer Security Center, 1985. Available at http://www.radium.ncsc.mil/tpep/library/rainbow.

12. Donahoo, Michael, and Calvert, Kenneth. *TCP/IP Sockets in C: Practical Guide for Programmers,* Morgan Kaufmann, San Francisco, CA, 2000. ISBN: 1-55860-826-5.

13. Eddon, G., and Eddon, D. *Inside Distributed COM,* Microsoft Press, Redmond, WA, 1998. ISBN: 1-57231-849-X.

14. Feuer, Alan. *MFC Programming,* Addison-Wesley, Reading, MA, 1997. ISBN: 0-201-63358-2.

15. Gilly, Daniel, and the staff of O'Reilly & Associates, Inc. *UNIX in a Nutshell,* O'Reilly & Associates, Inc., Sebastopol, CA, 1992. ISBN: 1-56592-001-5.

16. Hennessy, John L., and Patterson, David A. *Computer Architecture: A Quantitative Approach, Third Edition,* Morgan Kaufmann, San Francisco, CA, 2003. ISBN: 1-55860-596-7.

17. Hipson, Peter D. *Expert Guide to Windows NT 4 Registry,* Sybex, 1999. ISBN: 0-7821-1983-2.

18. Josutis, Nicolai M. *The C++ Standard Library: A Tutorial and Reference,* Addison-Wesley, Reading, MA, 1999. ISBN: 0-20-137926-0.

19. Kano, Nadine. *Developing International Applications for Windows 95 and Windows NT,* Microsoft Press, Redmond, WA, 1995. ISBN: 1-55615-840-8.

20. Kernighan, Brian W., and Ritchie, Dennis M. *The C Programming Language, Second Edition*, Prentice-Hall, Englewood Cliffs, NJ, 1988. ISBN: 0-13-110370-9.

21. Miller, Kevin. *Professional NT Services,* WROX, Indianapolis, IN, 1998. ISBN: 1-86100-130-4.

22. Naik, Dilip. *Inside Windows Storage — Server Storage Technologies for Windows 2000, Windows Serve r2003, and Beyond,* Addison-Wesley, Boston, MA, 2003. ISBN: 0-321-12698-X.

23. Nottingham, Jason P., Makofsky, Steven, and Tucker, Andrew. *SAMS Teach Yourself Windows CE Programming in 24 Hours,* SAMS, Indianapolis, IN, 1999. ISBN: 0-6723-1658-7.

24. Oney, Walter. *Programming the Microsoft Windows Driver Model, Second Edition*, Microsoft Press, Redmond, WA, 2002. ISBN: 0-735-61803-8.

25. Petzold, Charles. *Programming Windows, Fifth Edition,* Microsoft Press, Redmond, WA, 1998. ISBN: 1-572-31995-X.

26. Pham, Thuan, and Garg, Pankaj. *Multithreaded Programming with Win32,* Prentice-Hall, Englewood Cliffs, NJ, 1998. ISBN: 0-130-10912-6.

27. Plauger, P.J. *The Standard C Library,* Prentice-Hall, Englewood Cliffs, NJ, 1992. ISBN: 0-13-131509-9.

28. Quinn, Bob, and Shute, Dave. *Windows Sockets Network Programming,* Addison-Wesley, Reading, MA, 1996. ISBN: 0-201-63372-8.

29. Raymond, Eric S. *The Art of UNIX Programming,* Addison-Wesley, Boston, MA, 2003. ISBN: 0-131-42901-9.

30. Rector, Brent, and Newcomer, Joseph M. *Win 32 Programming,* Addison-Wesley, Reading, MA, 1997. ISBN: 0-201-63492-9.

31. Richter, Jeffrey. *Programming Applications for Microsoft Windows* (formerly *Advanced Windows NT: The Developer's Guide to the Win32 Application Programming Interface* in previous editions), Microsoft Press, Redmond, WA, 1999. ISBN: 1-57-231996-8.

32. Richter, Jeffrey, and Clark, Jason. *Programming Server-Side Applications for Microsoft Windows 2000,* Microsoft Press, Redmond, WA, 2000. ISBN: 0-73-560753-2.

33. Robbins, Kay A., and Robbins, Steven. *Practical UNIX Programming: A Guide to Concurrency, Communication, and Multithreading,* Prentice-Hall, Englewood Cliffs, NJ, 1995. ISBN: 0-13-443706-3.

34. Sedgewick, Robert. *Algorithms in C,* Addison-Wesley, Reading, MA, 1990. ISBN: 0201-51425-7.

35. Silberschatz, Abraham, Gagne, Greg, and Galvin, Peter B. *Operating System Concepts, Sixth Edition,* Wiley Textbooks, Hoboken, NJ, 2002. ISBN: 0-471-25060-0.

36. Sinha, Alok K. *Network Programming in Windows NT,* Addison-Wesley, Reading, MA, 1996. ISBN: 0-201-59056-5.

37. Solomon, David. *Inside Windows NT, Second Edition,* Microsoft Press, Redmond, WA, 1998. ISBN: 1-57-231677-2.

38. Solomon, David, and Russinovich, Mark. *Inside Windows 2000,* Microsoft Press, Redmond, WA, 2000. ISBN: 1-73-561021-5.

39. Standish, Thomas A. *Data Structures, Algorithms and Software Principles in C,* Addison-Wesley, Reading, MA, 1995. ISBN: 0-201-59118-9.

40. Stevens, W. Richard. *Advanced Programming in the UNIX Environment,* Addison-Wesley, Reading, MA, 1992. ISBN: 0-201-56317-7.

41. Stevens, W. Richard. *TCP/IP Illustrated, Volume3: TCP for Transactions, HTTP, NNTP, and the UNIX Domain Protocols,* Addison-Wesley, Reading, MA, 1996. ISBN:0-201-63495-3.

42. Stevens, W. Richard. *UNIX Network Programming — Networking APIs: Sockets and XTI, Volume I,* Prentice-Hall, Upper Saddle River, NJ, 1998. ISBN: 0-13-490012-X.

43. Sutton, Stephen A. *Windows NT Security Guide,* Addison-Wesley, Reading, MA, 1997. ISBN: 0-201-41969-6.

44. Triebel, Walter A. *Itanium Architecture for Software Developers,* Intel Press, 2000. ISBN: 0-970-28464-0.

45. Unicode Consortium, The. *The Unicode Standard, Version 2.0,* Addison-Wesley, Reading, MA,

1996. ISBN: 0-201-48345-9.

46. Weiss, Mark Allen. *Data Structures and Algorithm Analysis in C,* Addison-Wesley, Reading, MA, 1993. ISBN: 0-8053-5440-9.

47. Williams, Robert, and Walla, Mark. *The Ultimate Windows Server 2003 System Administrator's Guide,* Addison-Wesley, Boston, MA, 2003. ISBN: 0-201-79106-4.

48. Керниган, Брайан, Ритчи, Деннис, *Язык программирования Си,* "Невский Диалект", 2000.

49. Реймонд, Эрик, *Искусство программирования для UNIX,* Издательский дом "Вильямс", 2005.

---

notes

Примечания

1

Тем не менее, в тех местах книги, где речь идет о средствах, неприменимых в Windows 9x, делаются соответствующие оговорки.

Автор вовсе не стремится каким-либо образом дополнить существующие коммерческие продукты, а также продукты с открытым исходным кодом, в которых предоставляются полные наборы утилит UNIX. Хотя приводимые примеры и могут найти практическое применение, они предназначены, главным образом, для того, чтобы продемонстрировать возможности функций Windows. Как бы то ни было, читатели, не знакомые с UNIX, не будут испытывать каких-либо трудностей в понимании программ или их функциональности.

Замечания, сделанные в адрес UNIX, в равной степени относятся также к Linux и некоторым другим системам, поддерживающим POSIX API.

Иногда, имея в виду в основном серверы, но не исключая и персональные приложения, говорят о возможной угрозе преобладанию Windows со стороны Linux. Хотя сама по себе эта тема является чрезвычайно интересной, размышления о путях будущего развития систем, не имеющие непосредственного отношения к рассмотрению сравнительных достоинств и недостатков Windows и Linux, выходят за рамки данной книги.

5

О том, насколько разнообразен круг систем, на которых может быть развернута Windows, говорит хотя бы тот факт, что диапазон компьютеров, использованных для тестирования приведенных в этой книге примеров программ, простирается от давно забытой 486-й модели с 16 Мбайт ОЗУ до четырехпроцессорного (процессоры Xeon с рабочей частотой 2 ГГц) сервера масштаба предприятия, оборудованного ОЗУ емкостью 8 Гбайт.

6

Протоколы Windows Sockets и RPC не являются частью самой Windows, что не воспрепятствовало описанию сокетов в данной книге, поскольку они самым непосредственным образом укладываются в рамки интересующей нас общей темы и используемого подхода.

Несмотря на аналогию между упомянутыми дескрипторами и дескрипторами HWND и HDC, используемыми при написании программ для Windows GUI, между ними существует ряд отличий.

Такие типы, как PVOID, входят в include-файлы без префикса, но в примерах мы будем придерживаться правил их употребления, принятых во многих книгах и документации Microsoft.

9

О том, какими быстрыми темпами улучшаются показатели стоимости и производительности, вы можете судить хотя бы по тому факту, что еще в 1997 году в первом издании этой книги автор, без тени смущения или неловкости, в качестве необходимых требований указывал 16 Мбайт ОЗУ и 256 Мбайт свободного места на жестком диске. Для написания настоящего, третьего издания книги используется лэптоп стоимостью менее $1000, с объемом ОЗУ в более чем 10 раз превышающим прежний (что больше ранее требуемого объема дискового пространства), 100-кратной емкостью жесткого диска и 50-кратным превышением быстродействия процессора по сравнению с аналогичными характеристиками компьютера стоимостью $2500, который использовался при подготовке первого издания.

10

В приложении А показано, как исключить ненужные определения для ускорения компиляции и экономии дискового пространства.

Обратите внимание на то, что логика цикла зависит от принятого в стандарте ANSI C порядка вычисления логических операций "и" (&&) и "или" (||) в направлении слева направо.

Символ подчеркивания (_) указывает на то, что данная функция или ключевое слово предоставляются компилятором Microsoft C, тогда как буквы t и T указывают на то, что данная функция предназначена для работы с обобщенными символами, имеющими расширенную форму. Аналогичные возможности предлагаются и другими средами разработки приложений, хотя используемые в них имена функций и ключевые слова могут отличаться от приведенных выше.

13

В соответствии со сложившейся практикой для обозначения длинных указателей на параметры, представленные строками символов, используется префикс l.

14

Во время написания данной книги даже недорогие системы на базе лэптопов комплектовались жесткими дисками емкостью 40 Гбайт и более, и поэтому даже в случае малых систем средства для работы с файлами, размеры которых превышают 4 Гбайт, является не только желательными, но и необходимыми.

Сравнение функций SetFilePointer и GetCurrentDirectory демонстрирует непоследовательность стиля программирования Windows. В некоторых случаях для передачи входных и выходных значений применяются только параметры.

Вместе с тем, рассчитывать на 100-наносекундную точность не следует; точность измерения времени может быть различной в зависимости от характеристик оборудования.

Гораздо более удобным и последовательным было бы использование для управления реестром дескрипторов типа HANDLE. Существуют также и другие ничем не оправданные отклонения от принятой в Windows практики.

Заметьте, что суффикс "Ex" следует использовать или опускать в точном соответствии с приведенными именами функций. Функция, в названии которой присутствует этот суффикс, является расширением функции, в названии которой этот суффикс отсутствует.

Возможно, это дело вкуса, — то ли индивидуального, то ли корпоративного, — но многие программисты никогда не пользуются оператором goto и избегают употребления оператора break, кроме случаев его совместного использования с операторами switch и иногда — в циклах, а также совместного использования с операторами continue. Те, кто мыслит трезво, не спешат определять свою позицию в этом отношении. Обработчики завершения и исключений могут решать многие из тех задач, для решения которых вам хотелось бы привлечь оператор goto и операторы, снабженные метками.

Этот оператор является специфическим для компилятора Microsoft C и предоставляет эффективный способ выхода из блока try…finally без аварийного завершения выполнения.

21

Цены на модули памяти постоянно снижаются, а "типичный" объем доступной памяти увеличивается, поэтому назвать, какой именно объем памяти является типичным, довольно затруднительно. Во время написания данной книги недорогие системы снабжались памятью объемом 128-256 Мбайт. В большинстве случаев такой памяти будет вполне достаточно, но она не является оптимальной для Windows XP. Для систем Windows Server 2003 требуемый объем памяти обычно гораздо больше указанного.

Понятие потоков вводится в главе 7.

Обычно для создания объектов типа X используются системные вызовы CreateX. Функция HeapCreate является исключением из этого правила.

24

Утверждение относительно согласованности отображенных представлений файлов, видимых разными процессами, неприменимо к сетевым файлам. Файлы должны быть локальными.

В главе 10 рассказывается о косвенном методе, позволяющем одному потоку возбуждать исключения в другом, причем эта же методика применима и к потокам, принадлежащим разным процессам.

26

Рабочий набор — это набор страниц виртуального адресного пространства, которые ОС считает необходимым загрузить в память, прежде чем пытаться запустить любой из потоков процесса. Эта тема освещается в большинстве руководств по ОС.

Как объясняется в следующей главе, мьютексы — это объекты синхронизации, владельцами которых могут быть потоки.

Использование в данном случае контрольной суммы, вычисляемой в результате применения операции исключающего "или" к битам сообщения, носит исключительно иллюстративный характер. Существует множество других, более совершенных методик проверки целостности данных, которые и должны использоваться в промышленных приложениях.

Выбирая необходимый тип объекта, руководствуйтесь следующим правилом: если упоминавшиеся ограничения приемлемы — используйте объекты CRITICAL_SECTION, если же имеется несколько процессов или требуются возможности мьютексов — применяйте мьютексы.

Как показано в главе 10, в упражнении с семафором (упражнение 10.11), системные службы Windows предоставляют возможность организации взаимодействия между процессами также посредством отображаемых файлов. Дополнительные механизмы IPC включают файлы, сокеты, удаленные вызовы процедур, COM и отправку сообщений через почтовые ящики. Сокеты рассматриваются в главе 12.

Это утверждение нуждается в дополнительных разъяснениях. Для большинства сетевых приложений и высокоуровневых протоколов (http, ftp и так далее) более предпочтительным является интерфейс Windows Sockets API, особенно в тех случаях, когда требуется обеспечить межплатформенное взаимодействие с системами, отличными от Windows, на основе протокола TCP/IP. Многие разработчики предпочитают ограничивать использование именованных каналов лишь случаями IPC в пределах обособленной системы или в сетях Windows.

Заметьте, что функция TransactNamedPipe не только предлагает более удобный способ использования пары функций WriteFile и ReadFile, но и обеспечивает определенные преимущества в плане производительности. Один из экспериментов продемонстрировал повышение пропускной способности канала в интервале от 57% (небольшие сообщения) до 24% (крупные сообщения).

Эта терминология может несколько сбивать с толку, поскольку системы Windows предоставляют многочисленные услуги, которые не относятся к услугам, оказываемым службами Windows Services. Однако использование на протяжении всей этой книги термина "Windows" в тех местах, где имеется в виду API, кое-кем также может восприниматься неоднозначно.

Если служба вызывает функцию MessageBox, то в качестве типа окна сообщения следует указать MB_SERVICE_NOTIFICATION. Тогда сообщения будут отображаться на активном рабочем столе, даже если ни один из пользователей еще не успел войти в систему на данном компьютере.

В будущем, благодаря развитию платформы Win64 и предоставлению больших объемов физической памяти, острота этой проблемы, по всей видимости, снизится.

Точнее говоря, "UNIX" означает функции стандарта POSIX, описанные в спецификации *The Single UNIX Specification* (http://www.opengroup.org/onlinepubs/007908799/). Эта спецификация реализуется в UNIX и Linux. В свою очередь, исторически эта спецификация возникла на основе UNIX.