Библиотека системного программиста

двадцать шестой том

Программирование для Windows NT

(часть первая)

АННОТАЦИЯ

Новая книга в серии “Библиотека системного программиста” посвящена программированию для одной из наиболее перспективных операционных систем - Microsoft Windows NT. В ней мы рассмотрим наиболее интересные архитектурные особенности этой операционной системы, расскажем о системе управления памятью, об использовании мультизадачности, о файловой системе и затронем другие важнейшие вопросы. Книга будет полезна также тем, кто создает приложения для операционной системы Windows 95.

Материал иллюстрируется большим количеством исходных текстов приложений, которые вы можете приобрести отдельно на дискете.

ВВЕДЕНИЕ

Если вы думаете, что такая серьезная операционная система, как Microsoft Windows NT, не для вас, то не исключено, что находитесь в заблуждении. Благодаря значительному прогрессу в технологии изготовления модулей оперативной памяти, дисков и других периферийных устройств, стала возможной установка этой операционной системы в компьютер стоимостью не более 1300 - 1500 долларов. За последний год цена основных компонент компьютера снизилась почти в два раза. Поэтому теперь вряд ли у программиста возникнет зависть при взгляде на компьютер с 16 Мбайтами оперативной памяти и диском с объемом 540 Мбайт - такая конфигурация стала доступна не только разработчикам программного обеспечения, но и для многим пользователям. А ведь этих ресурсов вполне достаточно для нормальной работы операционной системы Microsoft Windows NT Workstation версии 3.51 или 4.0.

Обладатель компьютера, оснащенного 16 Мбайт оперативной памяти, может выбирать между операционными системами Microsoft Windows NT, Microsoft Windows 95 или IBM OS/2 Warp. Что касается операционной системы IBM OS/2 Warp, то ей мы посвятили 20 и 25 тома “Библиотеки системного программиста”, рассмотрев вопросы установки, использования и программирования для системы Presentation Manager. Однако многие пользователи, работавшие ранее с операционной системой Microsoft Windows версии 3.1, выберут Microsoft Windows 95 или Microsoft Windows NT.

Чем же отличаются друг от друга эти две операционные системы?

Операционная система Microsoft Windows 95 создавалась таким образом, чтобы максимально облегчить переход от 16-разрядной Microsoft Windows версии 3.1 к 32-разрядной операционной среде. При разработке Microsoft Windows 95 очень много внимания уделялось вопросам совместимости с программами MS-DOS и старыми 16-разрядными приложениями Windows, в результате чего пострадала устойчивость работы этой операционной системы. Несмотря на то, что в целом Microsoft Windows 95 ведет себя значительно устойчивее по сравнению с Microsoft Windows версии 3.1, плохо спроектированные приложения или приложения, содержащие ошибки, могут полностью вывести ее из строя. А что может быть хуже для пользователя, чем зависание системы или ее аварийное завершение, в результате которого оказываются потеряны важные данные!

Что же касается того, как Microsoft Windows 95 работает с файлами, то в этом она недалеко ушла от операционной системы MS-DOS. Фактически кроме разрешения использовать для имен каталогов и файлов строки увеличенного размера, новая файловая система VFAT не дает пользователям ничего нового. В частности, по-прежнему никак не решен вопрос разграничения доступа к данным и защиты данных от несанкционированного доступа.

Есть и другие, менее заметные для пользователей недостатки, источником которых является совместимость со старыми программами. Например, использование дисковых драйверов реального режима, загружаемых через файл config.sys (например, драйвера магнитооптического устройства), может привести к снижению производительности системы.

Теперь сделаем замечание относительно требований Microsoft Windows 95 к объему оперативной памяти.

Хотя нам удавалось запустить эту операционную систему в 4 Мбайтах памяти, работала она при этом исключительно медленно, так как выполнялся непрерывный свопинг. Если мы устанавливали 8 Мбайт оперативной памяти, ситуация сильно улучшалась, однако при использовании “тяжеловесных” офисных приложений, таких как Microsoft Word или Microsoft Excel, быстродействие системы оставляло желать лучшего. Особенно, если было запущенно одновременно несколько таких приложений или выполнялось редактирование объекта OLE. Лишь только после установки 16 Мбайт памяти свопинг сократился настолько, что он перестал существенным образом ухудшать производительность системы. Кстати, операционная система Microsoft Windows Workgroups версии 3.11 ведет себя аналогичным образом, особенно если используются ее сетевые возможности.

Таким образом, для нормальной работы офисных приложений в среде Microsoft Windows 95 вам придется установить не менее 16 Мбайт оперативной памяти. Но именно столько нужно и для работы Microsoft Windows NT Workstation. При этом последняя будет обладать такой же (если не большей) производительнстью, что и Microsoft Windows 95. У вас нет возможности работать с Microsoft Windows NT только в том случае, если в вашем компьютере установлена оперативная память объемом 8 Мбайт и вы не в состоянии расширить ее до 16 Мбайт, заплатив 150 - 160 долларов.

Что же касается операционной системы Microsoft Windows NT, то при ее создании разработчики из Microsoft руководствовались в первую очередь вопросами обеспечения надежности, а не совместимости со старыми программами. Сказанное не означает, что вы не сможете запустить в среде Microsoft Windows NT программу MS-DOS или 16-разрядное приложение Windows. Однако в отличие от Microsoft Windows 95, операционная система Microsoft Windows NT контролирует выполняемые такими программами действия намного строже. В результате у вас, например, могут возникнуть трудности при запуске некторых игровых программ, рассчитанных на MS-DOS, или других программ, обращающихся непосредственно к аппаратуре компьютера.

Раньше, когда 32-разрядных программ, предназначенных специально для Microsoft Windows NT, было очень мало, ограниченная совместимость со старыми программами MS-DOS и повышенные требования к системным ресурсам сдерживали распространение этой операционной системы. На

сегодняшний день 32-разрядных программ появилось очень много. Более того, практически прекратился выпуск новых 16-разрядных приложений Windows и программ MS-DOS. Вы можете приобрести 32-разрядную версию большинства известных 16-разрядных приложений, а также игровые программы, специально предназначенные для работы в среде Microsoft Windows NT или Microsoft Windows 95. Например, потребности большинства фирм удовлетворяет пакет Microsoft Office for Windows 95, который прекрасно работает в среде Microsoft Windows NT. В результате развития рынка 32-разрядных приложений актуальность совместимости со старыми программами значительно снизилась.

Что же дает пользователю переход от операционных систем Microsoft Windows версии 3.1 или Microsoft Windows 95 к Microsoft Windows NT?

Прежде всего, увеличится надежность и производительность. К другим преимуществам можно отнести возможность использования мощных средств разграничения доступа.

Файловая система NTFS, использованная в Microsoft Windows NT, является на сегодняшний день одной из лучших и надежных. У нее отсутствуют все недостатки файловой системы FAT, такие как сильные ограничения на длину имени файлов и каталогов, быстрая фрагментация дискового пространства при интенсивной работе. Система разграничения доступа, встроенная в Microsoft Windows NT, очень удобна, если компьютером пользуются несколько человек. Она позволяет организовать работу таким образом, чтобы пользователи имели доступ только к своим каталогам, а также каталогам, выделенным в совместное пользование. При необходимости администратор компьютера (да, теперь вы можете завести в вашем домашнем компьютере отдельного пользователя с правами администратора) может защитить некоторые каталоги, предоставив к ним доступ только на чтение. Это значительно сократит ущерб, например, в результате вирусной атаки или неосторожных действий начинающего пользователя.

К другим преимуществам файловой системы NTFS можно отнести использование системы транзакций, а также B-деревьев для поиска файлов в каталогах. Первое из этих преимуществ увеличивает надежность файловой системы, второе - быстродействие при поиске файлов в каталогах.

Конечно, при установке Microsoft Windows NT могут возникнуть и трудности. Так как эта операционная система, в отличие от Microsoft Windows 95 и Microsoft Windows версии 3.1, не может работать с драйверами для MS-DOS, вы должны иметь полный комплект драйверов от ваших периферийных устройств для Microsoft Windows NT. Если вы устанавливаете Microsoft Windows NT на новый компьютер, проблемы с драйверами у вас скорее всего не возникнут. Однако в любом случае перед установкой вам необходимо убедиться, что контроллер диска, видеоадаптер, устройство чтения CD-ROM и звуковой адаптер имеют драйверы для работы в Microsoft Windows NT. Если нужного драйвера нет, вы можете попытаться отыскать его в сети Internet. О работе в этой сети мы рассказали в 23 томе “Библиотеки системного программиста”, который называется “Глобальные сети компьютеров”.

Пользовательский интерфейс операционной системы Microsoft Windows NT версии 3.51 напоминает интерфейс Microsoft Windows версии 3.1, что в значительной мере упрощает переучивание пользователей. Тем не менее, будущее за объектно-ориентированным интерфейсом, использованном в Microsoft Windows 95. Уже сейчас вы можете бесплатно получить бета-версию объектно-ориентированной оболочки с сервера WWW корпорации Microsoft. А в самое ближайшее время выйдет Microsoft Windows NT версии 4.0, в которую такая оболочка будет встроена.

Подводя итог сказанному выше, мы делаем заключение, что операционная система Microsoft Windows NT готовится корпорацией Microsoft “на первые роли”. При общей тенденции снижения цен на компьютерное оборудование и лавинообразное появление новых 32-разрядных приложений пользователи смогут, наконец, выполнять всю свою работу в среде высоконадежной и удобной операционной системы Microsoft Windows NT.

Теперь о том, что вам потребуется для работы с нашей книгой.

Прежде всего, мы предполагаем, что вы уже умеете программировать для операционной системы Microsoft Windows версии 3.1. Несмотря на то что в Microsoft Windows NT появилось очень много нового, базовые понятия, такие как окна или обработка сообщений, не изменились. Создавая новую операционную систему, в Microsoft позаботились о том, чтобы программисты смогли легче перейти к ней от старой, 16-разрядной версии. Во многом это получилось. Поэтому мы решили рассказывать о программировании для Microsoft Windows NT не с самого начала, а в предположении о наличии у программиста определенных знаний о Microsoft Windows версии 3.1.

Если же вы ранее создавали программы только для MS-DOS или IBM OS/2, имеет смысл вначале прочитать 11 - 17 тома “Библиотеки системного программиста”, где мы рассказываем о программировании для Microsoft Windows версии 3.1.

Мы также очень рекомендуем вам ознакомиться с 22 томом “Библиотеки системного программиста”, который называется “Операционная система Windows 95 для программиста”. Практически все, изложенное в этой книге, применимо и для операционной системы Microsoft Windows NT.

Что же касается сетевых возможностей этой операционной системы, то в 23 томе “Библиотеки системного программиста”, который называется “Глобальные сети персональных компьютеров”, мы рассказали об использовании программного интерфейса Windows Sockets. С помощью этого интферфеса вы сможете создавать приложения для Microsoft Windows NT и Microsoft Windows 95, способные передавать данные по локальным или глобальным сетям с использованием протокола TCP/IP.

Примеры приложений, приведенные в книге, транслировались в системе разработки Microsoft Visual C++ версии 4.0. Вы таже можете воспользоваться

версией 2.0 или 4.1 этой системы. Для того чтобы не набирать исходные тексты вручную и избежать ошибок, мы рекомендуем приобрести дискету с исходными текстами приложений, которая продается вместе с книгой.

Для проверки приложений мы использовали компьютер на базе процессора Pentium-90 с объемом оперативной памяти 16 Мбайт. Хотя такой объем памяти вполне достаточен для трансляции наших примеров, при возможности имеет смысл увеличить его до 32 Мбайт. В этом случае работа над большими проектами пойдет заметно быстрее. Объем дисковой памяти в нашем компьютере составляет 2,5 Гбайт, однако для установки Microsoft Windows NT и Microsoft Visual C++ вполне хватит 540 Мбайт.

Для установки Microsoft Windows NT и системы разработки Microsoft Visual C++ удобно иметь устройство чтения компакт-дисков. Выбирая это устройство, поинтересуйтесь, есть ли к нему драйвер для Microsoft Windows NT. Мы использовали 4-скоростное устройство Mitsumi FX-400.

Что касается видеомонитора, то мы рекомендуем использовать такой, который может работать с разрешением 800x600, а лучше 1024x768 пикселов или даже с еще более высоким разрешением. В этом случае использовать систему Microsoft Visual C++ будет намного удобнее. Мы приобрели 15-дюймовый монитор Sony Multiscan 15sf, отличающийся достаточно малым размером зерна - 0,25 мм, что позволило нам работать при разрешении 1024x768 пикселов. В качестве видеоадаптера был применен Diamond Stealth 64 DRAM PCI, для которого в составе Microsoft Windows NT имеется подходящий драйвер. Для работы с мультимедиа необходим звуковой адаптер. Мы использовали адаптер Creative Sound Blaster 16.

Отладку мультизадачных приложений мы выполняли на четырехпроцессорном компьютере Compaq Proliant-2000.

Если вы собираетесь заниматься разработкой приложений для Microsoft Windows NT профессионально, вам следует приобрести набор компакт-дисков “Microsoft Developer Network. Developer Platform”. В состав этого набора входят многочисленные версии Microsoft Windows, MS-DOS, огромный запас различной документации и средств разработки. Стоимость комплекта невысока (около 250 долларов США), особенно если учесть все, что в него входит.

БЛАГОДАРНОСТИ

Авторы выражают благодарность сотруднику фирмы Interactive Products Inc. Максиму Синеву за многочисленные консультации.

Мы также благодарим корректора Кустова В. С. и сотрудников издательского отдела АО “Диалог-МИФИ” Голубева О. А., Голубева А. О., Дмитриеву Н. В., Виноградову Е. К., Кузьминову О. А.

КАК СВЯЗАТЬСЯ С АВТОРАМИ

Вы можете передать нам свои замечания и предложения по содержанию этой и других наших книг через электронную почту:

Сеть	*Наш адрес*	*Сеть*	*Наш адрес*
Relcom	frolov@glas.apc.org	CompuServe	>internet: frolov@glas.apc.org
GlasNet	frolov@glas.apc.org	UUCP	cdp!glas!frolov
Internet	frolov@glas.apc.org		

Если электронная почта вам недоступна, присылайте ваши отзывы в АО “Диалог-МИФИ” по адресу:

115409, Москва, ул. Москворечье, 31, корп. 2,

тел. 324-43-77

Приносим свои извинения за то что не можем ответить на каждое письмо. Мы также не занимаемся рассылкой дискет и исходных текстов к нашим книгам. По этому вопросу обращайтесь непосредственно в издательство “Диалог-МИФИ”.

1 УПРАВЛЕНИЕ ПАМЯТЬЮ

Система управления памятью, встроенная в ядро Microsoft Windows NT, является самой сложной и самой совершенной из тех, с которыми нам приводилось встречаться до сих пор. Она наилучшим образом отвечает потребностям современных приложений, “пожирающих” оперативную память мегабайтами и десятками мегабайт.

От того, насколько хорошо вы будете понимать принципы, положенные в основу системы управления памятью, и от того, насколько хорошо вы будете владеть программным интерфейсом этой системы, во многом зависит эффективность (а то и работоспособность) создаваемых вами приложений.

Сложность системы управления памятью обусловлена тем, что она использует все аппаратные возможности современных процессоров, в том числе механизм страничной адресации памяти. Так как операционная система Microsoft Windows NT является мультизадачной и может работать в мультипроцессорных системах, программисту приходится учитывать это при организации работы приложения с памятью.

В распоряжении программиста имеются функции, предназначенные для работы с памятью на разных уровнях. Наиболее низкоуровневые средства обеспечивают работу с виртуальной памятью на уровне отдельных страниц. Более высокоуровневые средства позволяют получать блоки памяти практически любого размера из отдельных пулов, принадлежащих процессу. Можно также использовать классические функции стандартной библиотеки транслятора языка C, такие как malloc и free. Кроме того, вам доступны функции, которые пришли из программного интерфейса 16-разрядной операционной системы Microsoft Windows весии 3.1. Особое место занимают функции, предназначенные для отображения файлов в виртуальную оперативную память, позволяющие работать с файлами как с обычными массивами.

Словом, возможностей много. Мы постараемся все их описать, а выбор, как всегда, за вами.

Немного истории

Для того чтобы вам было легче разобраться с системой управления памятью Microsoft Windows NT и ощутить, насколько она совершенна, сделаем краткий обзор принципов управления памятью, положенных в основу операционных систем MS-DOS и Microsoft Windows версии 3.1.

Управление памятью в MS-DOS

Принципы управления памятью в операционной системе MS-DOS и соответствующий программный интерфейс были рассмотрены нами в 19 томе “Библиотеки системного программиста”, который называется “MS-DOS для программиста”.

Напомним, что MS-DOS использует так называемый реальный режим работы процессора. Такие операционные системы, как Microsoft Windows версии 3.1, Microsoft Windows 95, Microsoft Windows NT и IBM OS/2 на этапе своей загрузки переводят процессор в защищенный режим работы. Подробно эти режимы мы описали в 6 томе “Библиотеки системного программиста”, который называется “Защищенный режим работы процессоров 80286/80386/80486”.

В реальном режиме работы процессора *физический адрес*, попадающий на шину адреса системной платы компьютера составляется из двух 16-разрядных компонент, которые называются сегментом и смещением (рис. 1.1).

Рис.1.1. Получение физического адреса в реальном режиме работы процессора

Процедура получения физического адреса из сегмента и смещения очень проста и выполняется аппаратурой процессора. Значение сегментной компоненты сдвигается влево на 4 бита и дополняется справа четырьмя нулевыми битами. Компонента смещения расширяется до 20 разрядов записью нулей в четыре старших разряда. Затем полученные числа складываются, в результате чего образуется 20-разрядный физический адрес.

Задавая произвольные значения для сегмента и смещения, программа может сконструировать физический адрес для обращения к памяти размером чуть больше одного мегабайта. Точное значение с учетом наличия области старшей памяти High Memory Area равно 1 Мбайт + 64 Кбайт - 16 байт.

Хорошо знакомая вам из программирования для MS-DOS комбинация компонет [сегмент : смещение] называется *логическим адресом* реального режима. В общем виде процедура преобразования логического адреса в физический схематически изображена на рис. 1.2.

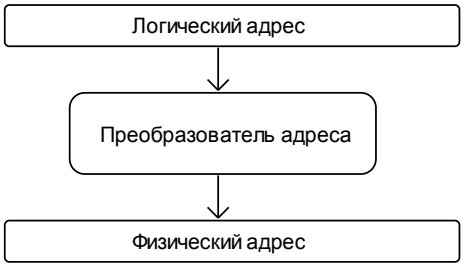

Рис. 1.2. Преобразование логического адреса в физический

Программы никогда не указывают физические адреса памяти, а всегда работают только с логическими адресами. Это верно как в реальном режиме работы процессора, так и в защищенном. Так как перобразователь адреса реализован аппаратно в процессоре, процесс преобразования не замедляет работу программы.

Логические адреса реального режима находятся в диапазоне от [0000h : 0000h] до [FFFFh : 000Fh]. Это соответствует диапазону физических адресов от 00000h до FFFFFh, лежащих в пределах первого мегабайта оперативной памяти. Задавая логические адреса в пределах от [FFFFh : 0010h] до [FFFFh : FFFFh], можно адресовать область старшей памяти High Memory Area, имеющей размер 64 Кбайт - 16 байт.

Таким образом, схема преобразования адреса реального режима не позволяет адресовать больше одного мегабайта памяти, что явно недостаточно для работы современных приложений. Даже если в вашем компьютере установлено 16 Мбайт памяти, операционная система MS-DOS не сможет адресовать непосредственно старшие 15 Мбайт (рис. 1.3).

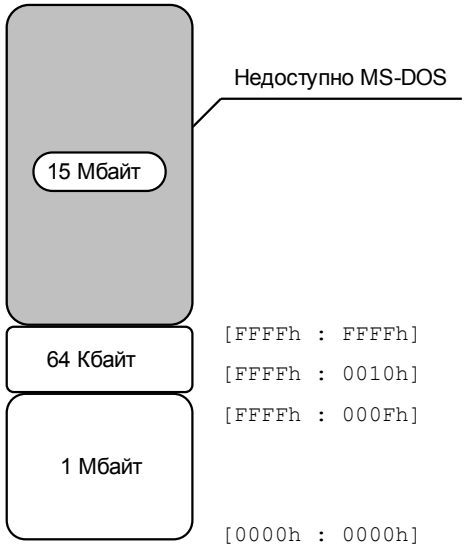

Рис. 1.3. Адресация памяти в MS-DOS

Несмотря на то что, как мы только что сказали, программы используют не физические, а логические адреса, преобразователь адреса реального режима позволяет программам легко сконструировать логический адрес для любого нужного ей физического адреса. В этом смысле можно говорить о возможности физической адресации памяти в реальном режиме.

Такая возможность сильно снижает надежность операционной системы MS-DOS, так как любая программа может записать данные в любую область памяти. В том числе, например, в область памяти, принадлежащей операционной системе или в векторную таблицу прерываний. Неудивительно, что компьютер, работающий под управлением MS-DOS, часто зависает, особенно при использовании плохо отлаженных программ.

Что же касается программного интерфейса для управления памятью в MS-DOS, то такой существует в рамках прерывания INT 21h. Он позволяет программам заказывать и освобождать области памяти, лежащие в границах первого мегабайта. Однако ничто не мешает программам выйти за пределы полученной области памяти, выполнив при этом самоликвидацию или уничтожение операционной системы.

Управление памятью в Microsoft Windows версии 3.1

Подробно схема управления памятью в Microsoft Windows версии 3.1 и соответствующие функции программного интерфейса мы описали в 13 томе

“Библиотеки системного программиста”, который называется “Операционная система Microsoft Windows 3.1 для программиста. Часть третья”. Здесь же мы остановимся только на самых важных моментах, необходимых для понимания отличий этой схемы от схемы адресации памяти в Microsoft Windows NT.

Адресация памяти

Приложения, запущенные в среде Microsoft Windows версии 3.1, работают с логическими адресами защищенного режима. Логический адрес защищенного режима, так же как и реального режима, состоит из двух компонент. Однако для защищенного режима это не сегмент и смещение, а селектор и смещение.

В стандартном режиме Microsoft Windows селектор и смещение всегда являются 16-разрядными. Когда Microsoft Windows работает в расширенном режиме, одна из компонент этой операционной системы (виртуальные драйверы VxD) работает в режиме 32-разрядной адресации. При этом селектор является 16-разрядным, а смещение - 32-разрядным. Что же касается обычных приложений, то и в расширенном режиме они не используют все возможности процессора 80386. В частности, размеры сегментов обычного приложения Windows не превышают 64 Кбайт.

Тем не менее расширенный режим работы обладает большим преимуществом: в нем используется страничная адресация и виртуальная память. Схема адресации для стандартного режима была описана в 13 томе “Библиотеки системного программиста”. Виртуальные драйверы описаны в 17 томе этой же серии книг.

В 32-разрядном режиме адресации (который в Microsoft Windows версии 3.1 используется только виртуальными драйверами) выполняется двухступенчатое преобразование логического адреса в физический, показанное на рис. 1.4.

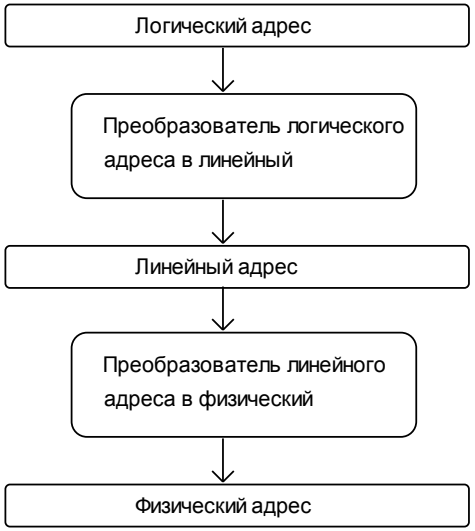

Рис. 1.4. Преобразование логического адреса в физический для 32-разрядного режима

На первом этапе логический адрес, состоящий из 16-разрядного селектора и 32-разрядного смещения, преобразуется в 32-разрядный *линейный адрес*.

Если бы линейный адрес отображался один к одному на физический (что возможно), то с его помощью можно было бы адресовать $2^{32} = 4294967296$ байт памяти, то есть 4 Гбайт. Дальнейшие преобразования линейного адреса в 32-разрядный физический адрес с использованием механизма страничной адресации позволяют еще больше расширить размер адресуемой памяти. И хотя в Microsoft Windows версии 3.1 приложения не могут использовать больше 256 Мбайт виртуальной памяти, операционная система Microsoft Windows NT успешно преодалевает этой барьер. Забегая вперед, скажем, что каждому приложению Microsoft Windows NT доступно ни много ни мало как… 2 Гбайта виртуальной памяти! Лишь бы в компьютере были установлены диски подходящей емкости.

Как же выполняется преобразование логического адреса в линейный в защищенном режиме работы процессора?

Как мы говорили в 6 и 13 томах “Библиотеки системного программиста”, для данного преобразования используется одна глобальная таблица дескрипторов GDT (Global Descriptor Table) и несколько локальных таблиц дескрипторов LDT (Local Descroptor Table). Это справедливо как для Microsoft Windows версии 3.1, так и для Microsoft Windows NT.

В таблицах дескрипторов хранятся элементы, описывающие отдельные сегменты памяти. В этих элементах среди всего прочего хранится одна из компонент линейного адреса - так называемый *базовый адрес*, имеющий 32 разряда. Преобразователь логического адреса в линейный складывает базовый адрес и 32-разрядное смещение, в результате чего получается 32-разрядный линейный адрес.

Селектор предназначен для индексации внутри одной из перечисленных выше таблиц дескрипторов. Селектор состоит из трех полей - индекса, индикатора TI и поля уровня привилегий RPL (рис. 1.5).

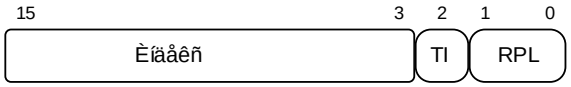

Рис. 1.5. Формат селектора

Поле TI (Table Indicator) используется для выбора глобальной или локальной таблицы дескрипторов. В любой момент времени может использоваться одна глобальная таблица дескрипторов и одна локальная таблица дескрипторов. Если бит TI равен 0, для выборки базового адреса используется глобальная таблица дескрипторов GDT, если 1 - локальная LDT.

Поле RPL (Requested Privilege Level) селектора содержит уровень привилегий, запрошенный приложением при обращении к сегменту памяти, который описывается дескриптором. Программа может обращаться только к таким сегментам, которые имеют соответствующий уровень привилегий. Поэтому программа не может, например, воспользоваться глобальной таблицей дескрипторов для получения доступа к описанным в ней системным сегментам, если она не обладает достаточным уровнем привилегий. На этом основана защита системных данных от разрушения (преднамеренного или в результате программной ошибки) со стороны прикладных программ.

Упрощенная схема преобразования логического адреса в линейный показана на рис. 1.6.

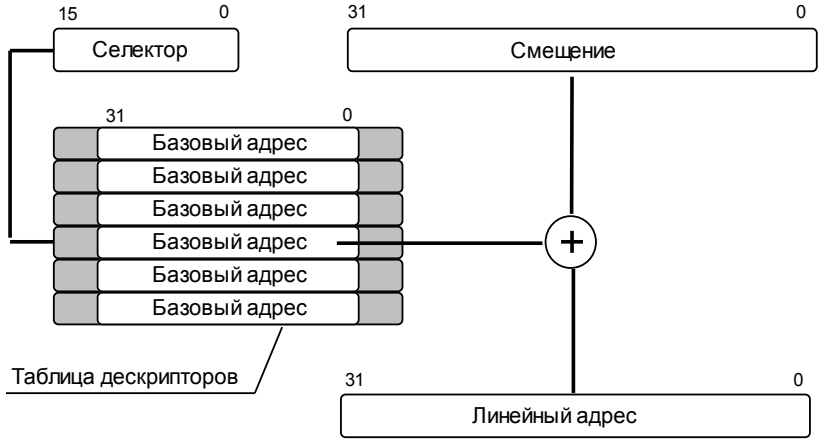

Рис. 1.6. Преобразование логического адреса в линейный

Таблицы дескрипторов создаются операционной системой на этапе ее инициализации перед переключением процессора в защищенный режим и впоследствии могут изменяться как самой операционной системой, так и приложениями (явно или неявно).

Помимо базового адреса, дескрипторы содержат и другую информацию, описывающую сегменты памяти. В частности, в них находится размер сегмента, а также уровень доступа, которым должно обладать приложение для доступа к сегменту.

Если приложение попытается адресовать память, лежащую за пределами области, описанной дескриптором, произойдет аппаратное прерывание. Аналогичная реакция будет и на попытки приложения выполнить неразрешенный доступ, например, сделать попытку записи в сегмент, для которого разрешено только чтение, или в программный сегмент, содержащий исполняемый код. Приложение также не может использовать селекторы, для которых нет заполненных дескрипторов.

Таким образом, в защищенном режиме приложение не может делать с адресами все, что ему вздумается, как это было в реальном режиме. Более подробную информацию об этом вы можете получить из 6 тома “Библиотеки системного программиста”.

Зачем нужно использовать дескрипторные таблицы двух типов?

В мультизадачной операционной системе можно использовать одну глобальную таблицу дескрипторов для описания областей памяти, принадлежащей операционной системе и несколько локальных таблиц дескрипторов для каждого процесса. В этом случае при соответствующей настройке базовых адресов можно изолировать адресные пространства

операционной системы и отдельных процессов. Если сделать так, что каждый процесс будет пользоваться только своей таблицей дескрипторов, любой процесс сможет адресоваться только к своим сегментам памяти, описанным в соответствующей таблице, и к сегментам памяти, описанным в глобальной таблице дескрипторов. В системе может существовать только одна глобальная таблица дескрипторов.

Заметим, что операционная система Microsoft Windows версии 3.1 создает одну общую глобальную таблицу дескрипторов, одну таблицу локальных дескрипторов для системной виртуальной машины, в рамках которой работают все приложения Windows, и по одной локальной таблице дескрипторов для каждой виртуальной машины DOS. Подробнее об этом вы можете узнать из главы “Драйверы для Windows” 17 тома “Библиотеки системного программиста”, который называется “Операционная система Microsoft Windows 3.1 для программиста. Дополнительные главы”.

Примечательно, что все 16-разрядные приложения Windows версии 3.1 работают в одном адресном пространстве, которое представлено одной локальной таблицей дескрипторов. Это служит одной из причин нестабильности Microsoft Windows версии 3.1, так как плохо отлаженное приложение может разрушить области памяти, принадлежащие другим приложениям, или даже самой операционной системе.

Займемся теперь преобразователем линейного адреса в физический. Процесс такого преобразования имеет самое непосредственное отношение к страничной адресации памяти.

Линейный адрес разделяется на три поля:

- номер таблицы в каталоге таблиц страниц (10 бит);
- номер страницы в таблице страниц (10 бит);
- смещение внутри страницы (12 бит)

Вся память при этом делится на страницы размером 4096 байт, адресуемые с помощью специальных таблиц страниц. В системе существует один каталог таблиц страниц и много таблиц страниц. Таблица страниц содержит 32-разрядные физические адреса страниц памяти.

В процессе преобразования линейного адреса в физический происходит выбор нужного элемента каталога таблиц и таблицы страниц. Далее к базовому физическому адресу страницы прибавляется смещение, взятое из третьего поля линейного адреса и таким образом получается физический адрес (рис. 1.7).

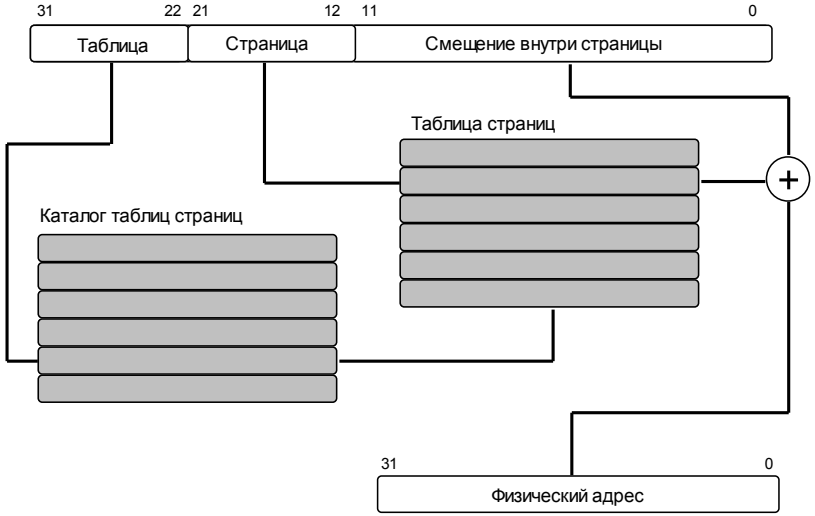

Рис. 1.7. Преобразование линейного адреса в физический

Отметим, что преобразование линейного адреса в физический выполняется аппаратурой процессора с помощью каталога таблиц страниц и таблиц страниц, подготовленных операционной системой. Приложение Windows никогда не работает с таблицами страниц или каталогом таблиц страниц. Оно пользуется логическими адресами в формате [селектор : смещение].

Основное преимущество системы управления памятью расширенного режима Windows заключается в возможности использования виртуальной памяти.

Виртуальная память работает на уровне страниц (описанных в каталогах страниц) и совершенно прозрачна для программиста. Операционная система Windows полностью управляет виртуальной памятью. Если приложение пытается обратиться к странице, отсутствующей в памяти и выгруженной на диск, происходит прерывание, после чего страница подгружается с диска. Вслед за этим работа приложения продолжается.

Пользуясь виртуальной памятью, приложение может заказывать для себя блоки памяти огромного размера. При этом возникает иллюзия работы с очень большой оперативной памятью.

Пулы памяти в Microsoft Windows версии 3.1

Напомним, что в Microsoft Windows версии 3.1 существует глобальная область памяти (глобальный пул Global Heap), доступная для всех

приложений. Кроме того, для каждого приложения выделяется “в личное пользование” локальный пул Local Heap размером 64 Кбайт.

Если одно приложение получает область памяти из глобального пула, оно может передать адрес этой области другому приложению и то сможет им воспользоваться (так как все приложения работают в одном адресном пространстве). В операционной системе Microsoft Windows NT этот прием работать не будет, так как каждое приложение работает в отдельном адресном пространстве с использованием отдельной локальной таблицы дескрипторов.

Для получения памяти из глобального и локального пула в программном интерфейсе Microsoft Windows версии 3.1 были предусмотрены отдельные функции с именами GlobalAlloc и LocalAlloc. Выделенную этими функциями область памяти перед использованием было необходимо зафиксировать, вызвав, соответственно, функции GlobalLock и LocalLock. Для выделяемых областей памяти автоматически создавались дескрипторы в локальной таблице дескрипторов.

При необходимости приложение могло изменять содержимое глобальной или локальной таблицы дескрипторов, для чего в программном интерфейсе Microsoft Windows версии 3.1 были предусмотрены соответствующие функции. Эти функции, описанные нами в 13 томе “Библиотеки системного программиста”, давали обычным приложениям Windows почти ничем не ограниченные права доступа к памяти. Поэтому можно утверждать, что несмотря на использование защищенного режима работы процессора, по надежности Microsoft Windows версии 3.1 недалеко ушла от операционной системы MS-DOS.

Виртуальная память в Microsoft Windows NT

В основе всей системы управления памятью Microsoft Windows NT лежит система виртуальной памяти, встроенная в ядро операционной системы. Эта система позволяет приложениям использовать области памяти, размер которых значительно превышает объем установленной в компьютере физической оперативной памяти.

Насколько значительно?

Каждое приложение, запущенное в среде Microsoft Windows NT, может адресовать до 2 Гбайт виртуальной памяти. Причем для каждого приложения выделяется отдельное пространство виртуальной памяти, так что если, например, вы запустили 10 приложений, то в сумме они могут использовать до 20 Гбайт виртуальной памяти. При всем при этом в компьютере может быть установлено всего лишь 16 Мбайт физической оперативной памяти.

Казалось бы, зачем приложениям столько памяти?

Заметим, однако, что требования программ к объему оперативной памяти растут очень быстро. Еще совсем недавно были в ходу компьютеры с объемом оперативной памяти всего 1 - 4 Мбайт. Однако современные приложения, особенно связанные с обработкой графической или мультимедийной информации предъявляют повышенные требования к этому ресурсу. Например, размер файла с графическим изображением True Color может достигать десятков Мбайт, а размер файла с видеоизображением - сотен Мбайт. Без использования виртуальной памяти возможность работы с такими файлами была бы проблематичной.

Так же как и в операционной системе Microsoft Windows версии 3.1, в Microsoft Windows NT для создания виртуальной памяти используются дисковые устройства (рис. 1.8). Система виртуальной памяти Microsoft Windows NT позволяет создать до 16 отдельных файлов страниц, расположенных на разных дисковых устройствах, установленных в компьютере. Так как максимальный объем доступной виртуальной памяти определяется объемом использованных для нее дисковых устройств, то при необходимости вы можете, например, подключить к компьютеру несколько дисков большой емкости и разместить на них файлы страниц. Сегодня стали уже вполне доступными дисковые устройства с объемом 4 - 10 Гбайт, что позволяет в Microsoft Windows NT создавать виртуальную память действительно большого размера.

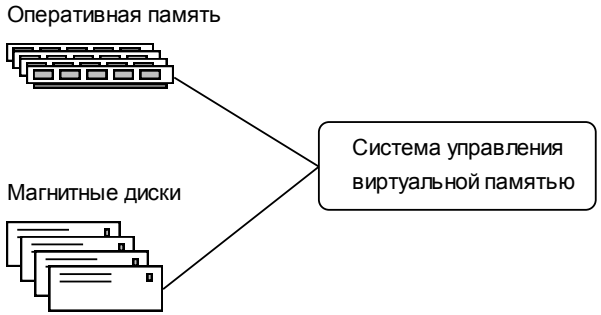

Рис. 1.8. Виртуальная память в Microsoft Windows NT создается с использованием оперативной памяти и дисковых устройств

Еще один случай, когда вашему приложению может пригодится виртуальная память объемом 2 Гбайта, это использование файлов, отображаемых на память. Выполнив такое отображение в свое адресное пространство, приложение может обращаться к файлу, как к обычному массиву, расположенному в оперативной памяти. При этом все необходимые операции ввода/вывода выполняются автоматически системой управления виртуальной памятью, так что приложение не должно об этом заботиться. Отобразив на память файл реляционной базы данных, вы можете обращаться к записям этого файла по индексу как к элементам массива, что значительно упрощает программирование.

Несегментированная модель памяти FLAT

Операционная система Microsoft Windows NT использует все возможности защищенного режима процессора, в частности, переключение задач и страничную адресацию. Схема преобразования адреса похожа на ту, что была применена в Microsoft Windows версии 3.1, однако есть много отличий.

Наиболее значительное - полный отказ от использования сегментированной модели памяти в 32-разрядных приложениях, к которой вы привыкли, создавая программы для MS-DOS и Microsoft Windows версии 3.1. В самом деле, используя 32-разрядное смещение, приложение может работать с памятью объемом 4 Гбайт без использования сегментных регистров процессора.

Поэтому, хотя логический адрес по-прежнему состоит из компонент селектора и смещения, приложения Microsoft Windows NT при обращении к памяти указывают только компоненту смещения. Сегментные регистры процессора, хранящие селекторы, заполняются операционной системой и приложение не должно их изменять.

Несегментированная модель памяти называется сплошной моделью памяти FLAT. При программировании в этой модели памяти вам не потребуются ключевые слова near и far, так как все объекты, расположенные в памяти, адресуются с использованием только одного смещения. В этом модель памяти FLAT напоминает модель памяти TINY, с тем исключением, однако, что в последней размер адресуемой памяти не может превышать 64 Кбайт (из-за того что в модели TINY используется один сегмент и 16-разрядное смещение).

Таким образом, в распоряжении каждого приложения (или точнее говоря, каждого процесса) Microsoft Windows NT предоставляется линейное адресное пространство размером 4 Гбайта. Область размером 2 Гбайта с диапазоном адресов от 00000000h до 7FFFFFFFh предоставлена в распоряжение приложению, другие же 2 Гбайта адресуемого пространства зарезервированы для использования операционной системой (рис. 1.9).

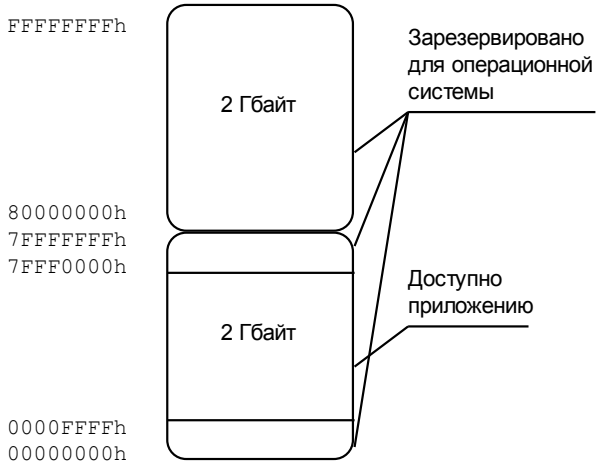

Рис. 1.9. Адресное пространство приложения Microsoft Windows NT

Как это показано на рис. 1.9, небольшая часть адресного пространства в пределах первых 2 Гбайт также зарезервирована для операционной системы. Это области размером 64 Кбайта, расположенные в самом начале адресного пространства, а также в конце первых 2 Гбайт, и предназначенные для обнаружения попыток использования неправильных указателей. Таким образом, приложения не могут адресовать память в диапазонах адресов 00000000h - 0000FFFFh и 7FFF0000h - 7FFFFFFFh.

Изолированные адресные пространства

Для обеспечения необходимой надежности работы в Microsoft Windows NT адресные пространства всех запущенных приложений разделены. Такое разделение выполняется с помощью назначения приложениям индивидуальных наборов таблиц страниц виртуальной памяти. В результате для каждого приложения выполняется отображение линейных адресов в собственный набор страниц виртуальной памяти, не пересекающийся с набором страниц других приложений.

Заметим, что приложение не имеет физической возможности выполнить адресацию памяти в пространстве, принадлежащем другому приложению. Какой бы линейный адрес ни задавало приложение, этот адрес всегда будет соответствовать одной из страниц, принадлежащих самому приложению. Такое положение дел обеспечивается системой управления виртуальной памятью Microsoft Windows NT и значительно повышает устойчивость операционной системы.

Однако полное изолирование адресных пространств создает трудности при необходимости организации обмена данными между различными приложениями. Вы не можете просто так передать указатель на область памяти из одного приложения в другое, так как в контексте другого приложения содержимое этого указателя не будет иметь смысла. Так как адресные пространства приложений изолированы, одному и тому же значению линейного адреса будут соотвтетсвовать ячейки памяти, расположенные в разных страницах.

В операционной системе Microsoft Windows версии 3.1 все приложения работали в одном адресном пространстве. Поэтому для передачи данных между приложениями можно было заказать область памяти в глобальном пуле с помощью функции GlobalAlloc и затем передать адрес этой области другому приложению. В Microsoft Windows NT этот метод работать не будет.

Выход, тем не менее, есть. Ничто не мешает операционной системе создать в каталоге страниц нескольких приложений специальный дескриптор, указывающий на страницы виртуальной памяти общего пользования. Такие дескрипторы называются дескрипторами прототипа PTE (Prototype Page Table Entry) и используются для совместного использования страниц памяти в операционной системе Microsoft Windows NT.

Дескрипторы PTE создаются для совместного использования страниц, содержащих исполнимый программный код, а также для работы с файлами, отображаемыми на память. Есть также способ организации общей памяти при помощи библиотек динамической компоновки DLL.

Поэтому если вам потребуется организовать передачу данных между приложениями, вы сможете всегда это сделать, например, через файл, отображаемый на память.

Дескрипторы страниц памяти

Как мы уже говорили, таблица страниц содержит дескрипторы, описывающие отдельные страницы памяти. Эти дескрипторы содержат физические адреса страниц, а так же другую информацию.

На рис. 1.10 показан формат дескриптора страницы.

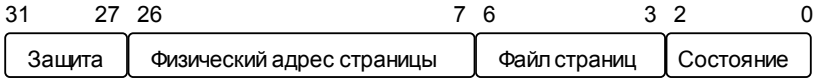

Рис. 1.10. Формат дескриптора страницы

Физический адрес страницы имеет 20 разрядов. Для получения 32-разрядного физического адреса байта внутри страницы к нему добавляются 12 байт смещения, взятые из линейного адреса, как это было показано на рис. 1.7.

Устанавливая соответствующим образом биты защиты, операционная система может отметить страницу как доступную для чтения и записи, только для чтения, или как недоступную. При попытке выполнить обращение для выполнения неразрешенной операции возникает аппаратное прерывание.

Биты с 3 по 6 содержат номер файла страниц, в котором находится страница, соответствующая данному дескриптору. Напомним, что в отличие от Microsoft Windows версии 3.1, операционная система Microsoft Windows NT позволяет создать до 16 файлов страниц.

Биты с 0 по 2 описывают состояние страницы памяти. Станица может быть отмечена флагами T (находится в переходном состоянии), D (обновленная, но не сохраненная в файле страниц), и P (присутствующая в памяти). Если приложение выполняет попытку обращения к странице памяти, которой нет в памяти, возникает аппаратное прерывание и нужная страница автоматически читается из соответствующего файла страниц в физическую оперативную память. После этого работа приложения продолжается.

Состояние страниц памяти

В дополнение к трем битам состояния страниц, хранящихся в дескрипторах страниц, система управления виртуальной памятью хранит состояние страниц в специальной базе данных страниц. В этой базе данных страница может быть отмечена как имеющая одно из следующих состояний:

Состояние	*Описание*
Свободная	Страница доступна для использования после ее заполнения нулями
Заполненная нулями	Свободная страница, заполненная нулями и доступная для использования приложениями
Правильная	Страница используется активным процессом
Измененная	Содержимое страницы было изменено, однако она не были еще сохранена на диске в файле страниц
Запасная	Страница удалена из рабочего набора страниц процесса
Плохая	При обращении к этой странице возникла аппаратная ошибка

Обратите внимание, что если часть оперативной памяти неисправна, есть вероятность, что операционная система Microsoft Windows NT сможет продолжить работу. Неисправные страницы будут отмечены в базе данных страниц как плохие и к ним не будет выполняться обращение.

Функции для работы с виртуальной памятью

В программном интерфейсе Microsoft Windows NT имеются средства, предназначенные для работы с виртуальной памятью. Они работают на уровне страниц памяти, имеющих размер 4096 байт, поэтому обычно нет смысла использовать эти функции только для того чтобы заказать в программе буфер размером в несколько десятков байт.

Получение виртуальной памяти

У приложения есть две возможности заказать для себя страницы виртуальной памяти. Первая возможность заключается в резервировании заданного диапазона адресов в адресном пространстве приложения, вторая - в фактическом получении в пользование страниц виртуальной памяти, к которым можно выполнять обращение.

Процесс резервирования не отнимает много времени и не приводит к изменениям в файлах страниц. При резервировании приложение только отмечает область памяти, лежащую в заданном диапазоне адресов как зарезервированную.

Для чего может потребоваться резервирование диапазона адресов?

Например, при отображении файла, имеющего большие размеры, в память, приложение может зарезервировать область виртуальной памяти, имеющую размер, равный размеру файла (даже если файл занимает несколько сотен Мбайт). При этом гарантируется, что для адресации этой области памяти можно будет использовать сплошной диапазон адресов. Это удобно, если вы собираетесь работать с файлом, как с массивом, расположенным в оперативной памяти.

Если же приложение не резервирует, а получает страницы памяти для непосредственного использования, эти страницы физически создаются в виртуальной памяти и заполняются нулями. При этом может происходить запись в файлы страниц. Разумеется, такой процесс отнимает гораздо больше времени, чем резервирование.

Для того чтобы зарезервировать или получить в свое распоряжение некоторое количество страниц виртуальной памяти, приложение должно воспользоваться функцией VirtualAlloc, прототип которой представлен ниже:

```
LPVOID VirtualAlloc(
  LPVOID lpvAddress,       // адрес области
  DWORD  cbSize,           // размер области
  DWORD  fdwAllocationType, // способ получения памяти
  DWORD  fdwProtect);      // тип доступа
```

Параметры lpvAddress и cbSize задают, соответственно, начальный адрес и размер резервируемой либо получаемой в пользование области памяти. При резервировании адрес округляется до ближайшей границы блока размером 64 Кбайт. В остальных случаях адрес округляется до границы ближайшей страницы памяти.

Заметим, что параметр lpvAddress можно указать как NULL. При этом операционная система выберет начальный адрес самостоятельно.

Что же касается параметра cbSize, то он округляется до целого числа страниц. Поэтому если вы пытаетесь с помощью функции VirtualAlloc получить область памяти размером в один байт, вам будет выделена страница размером 4096 байт. Аналогично, при попытке получить блок памяти размером 4097 байт вы получите две страницы памяти общим размером 8192 байта. Как мы уже говорили, программный интерфейс системы управления виртуальной памятью не предназначен для работы с областями малого размера.

Для параметра fdwAllocationType вы можете использовать одно из следующих значений:

Значение	*Описание*
MEM_RESERVE	Функция VirtualAlloc выполняет резервирование диапазона адресов в адресном пространстве приложения
MEM_COMMIT	Выполняется выделение страниц памяти для непосредственной работы с ними. Выделенные страницы заполняются нулями
MEM_TOP_DOWN	Память выделяется в области верхних адресов адресного пространства приложения

С помощью параметра fdwProtect приложение может установить желаемй тип доступа для заказанных страниц. Можно использвать одно из следующих значений:

Значение	*Разрешенный доступ*
PAGE_READWRITE	Чтение и запись
PAGE_READONLY	Только чтение
PAGE_EXECUTE	Только исполнение программного кода
PAGE_EXECUTE_READ	Исполнение и чтение
PAGE_EXECUTE_READWRITE	Исполнение, чтение и запись
PAGE_NOACCESS	Запрещен любой вид доступа
PAGE_GUARD	Сигнализация доступа к старнице. Это значение можно использовать вместе с любыми другими, кроме PAGE_NOACCESS

PAGE_NOCACHE	Отмена кэширования для страницы памяти. Используется драйверами устройств. Это значение можно использовать вместе с любыми другими, кроме PAGE_NOACCESS

Если страница отмечена как PAGE_READONLY, при попытке записи в нее возникает аппаратное прерывание защиты доступа (access violation). Эта страница также не может содержать исполнимый код. Попытка выполнения такого кода приведет к возникновению прерывания.

С другой стороны, у вас есть возможность получения страниц, предназначенных только для хранения исполнимого кода. Если такие страницы отмечены как PAGE_EXECUTE, для них не разрешаются операции чтения и записи.

При необходимости зафиксировать обращение к той или иной странице приложение может отметить ее как PAGE_GUARD. Если произойдет попытка обращения к такой странице, возникнет исключение с кодом STATUS_GUARD_PAGE, после чего признак PAGE_GUARD будет автоматически сброшен.

В случае успешного завершения функция VirtualAlloc возвратит адрес зарезервированной или полученной области страниц. При ошибке будет возвращено значение NULL.

Приложение может вначале зарезервировать страницы, вызвав функцию VirtualAlloc с параметром MEM_RESERVE, а затем получить их в пользование, вызвав эту же функцию еще раз для полученной области памяти, но уже с параметром MEM_COMMIT.

Освобождение виртуальной памяти

После использования вы должны освободить полученную ранее виртуальную память, вызвав функцию VirtualFree:

```
BOOL VirtualFree(
  LPVOID lpvAddress,   // адрес области
  DWORD  cbSize,       // размер области
  DWORD  fdwFreeType); // выполняемая операция
```

Через параметры lpvAddress и cbSize передаются, соответственно, адрес и размер освобождаемой области.

Если вы зарезервировали область виртуальной памяти функцией VirtualAlloc с параметром MEM_RESERVE для последующего получения страниц в пользование и затем вызвали эту функцию еще раз с параметром MEM_COMMIT, вы можете либо совсем освободить область памяти, обозначив соответствующие страницы как свободные, либо оставить их зарезервированными, но не используемыми.

В первом случае вы должны вызвать функцию VirtualFree с параметром fdwFreeType, равным MEM_RELEASE, во втором - с параметром MEM_DECOMMIT.

Три состояния страниц виртуальной памяти

Страницы виртуальной памяти, принадлежащие адресному пространству процесса в Microsoft Windows NT, могут находиться в одном из трех состояний. Они могут быть свободными (free), зарезервированными (reserved) или выделенными для использования (committed). В адресном пространстве приложения есть также относительно небольшое количество страниц, зарезервированных для себя операционной системой. Эти страницы недоступны приложению.

Функция VirtualAlloc может либо зарезервировать свободные страницы памяти (для чего ее нужно вызвать с параметром MEM_RESERVE), либо выделить свободные или зарезервированные страницы для непосредственного использования (для этого функция вызывается с параметром MEM_COMMIT). Приложение может либо сразу получить страницы памяти в использование, либо предварительно зарезервировать их, обеспечив доступное сплошное адресное пространство достаточного размера.

Для того чтобы зарезервированная или используемая область памяти стала свободной, вы должны вызвать для нее функцию VirtualFree с параметром MEM_RELEASE.

Вы можете перевести страницы используемой области памяти в зарезервированное состояние, не освобождая соответствующего адресного пространства. Это можно сделать при помощи функции VirtualFree с параметром MEM_DECOMMIT.

На рис. 1.11 мы показали три состояния страниц виртуальной памяти и способы перевода страниц из одного состояния в другое.

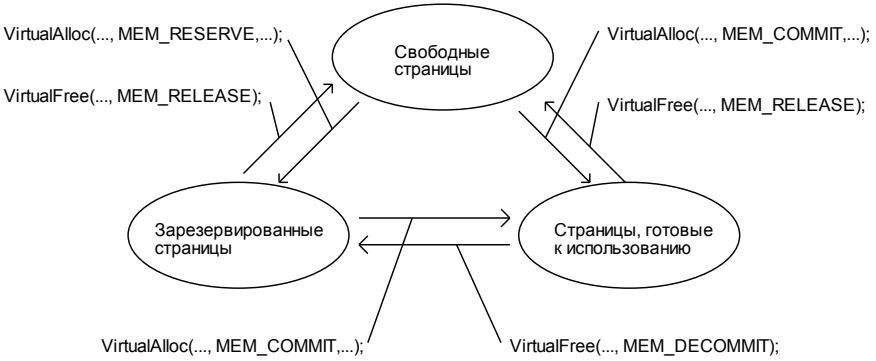

Рис. 1.11. Три состояния страниц виртуальной памяти

Фиксирование страниц виртуальной памяти

Вирутальная память по сравнению с физической оперативной памятью обладает одним существенным недостатком - невысоким быстродействием. Это из-за того, что файл страниц расположен на диске. В том случае, когда вы, например, создаете драйвер периферийного устройства, обрабатывающий аппаратные прерывания, вам могут потребоваться области памяти, гарантированно размещенные в физической оперативной памяти.

В программном интерфейсе Microsoft Windows NT есть функция VirtualLock, с помощью которой нетрудно зафиксировать нужное вам количество страниц в физической памяти.

Прототип функции VirtualLock представлен ниже:

```
BOOL VirtualLock(
  LPVOID lpvAddress, // адрес начала фиксируемой
                     // области памяти
  DWORD  cbSize);    // размер области в байтах
```

Через параметр lpvAddress вы должны передать адрес фиксируемой области памяти, расположенной в страницах, готовых к использованию.

Параметр cbSize, задающий размер фиксируемой области памяти, может иметь значение, которое не кратно размеру страницы. В результате вызова функции будет зафиксировано столько страниц, сколько нужно для размещения указанной области.

Для расфиксирования страниц памяти следует вызвать функцию VirtualUnlock, имеющую аналогичное назначение параметров:

```
BOOL VirtualUnlock(
  LPVOID lpvAddress, // адрес начала расфиксируемой
                     // области памяти
  DWORD  cbSize);    // размер области в байтах
```

Сколько страниц памяти можно зафиксировать функцией VirtualLock?

Не очень много. По умолчанию приложение может зафиксировать не более 30 страниц виртуальной памяти. И это сделано не зря - фиксирование большого количества страниц одним приложением уменьшает объем физической памяти, доступной для других приложений и может отрицательно сказаться на производительности всей системы в целом. Однако при необходимости вы можете увеличить это значение при помощи функции SetProcessWorkingSetSize, описанной в SDK.

Отметим, что обычным приложениям, не занимающимся обработкой аппаратных прерываний или решением других задач реального времени, не следует фиксировать страницы памяти, чтобы не мешать работе других приложений и операционной системе.

Изменение типа разрешенного доступа для страниц памяти

При получении страниц памяти в пользование функцией VirtualAlloc вы можете в последнем параметре указать тип доступа, разрешенного для этих страниц. В процессе работы приложение может изменять тип доступа для полученных им страниц при помощи функции VirtualProtect, прототип которой представлен ниже:

```
BOOL VirtualProtect(
  LPVOID lpvAddress,      // адрес области памяти
  DWORD  cbSize,          // размер области памяти в байтах
  DWORD  fdwNewProtect,   // новый тип разрешенного доступа
  PDWORD pfdwOldProtect); // указатель на переменную,
                          // в которую будет записан прежний код доступа
```

Через параметр lpvAddress вы должны передать адрес области памяти, расположенной в готовых для использования страницах (а не в зарезервированных страницах).

Новый тип доступа передается через параметр fdwNewProtect. Здесь вы можете использовать все константы, что и для последнего параметра функции VirtualAlloc, например, PAGE_READWRITE или PAGE_READONLY.

Зачем вам может пригодиться изменение кода доступа?

Например, вы можете получить страницы страницы памяти, доступные на запись, а затем, записав туда данные, разрешить доступ только на чтение или на исполнение. Устанавливая тип доступа PAGE_NOACCESS для страниц, которые в данный момент не используются приложением, вы можете, например, обнаружить во время исполнения кода ошибки, связанные

с использованием неправильных указателей. Для решения аналогичных задач можно также устанавливать тип доступа PAGE_GUARD.

Заметим, что функция VirtualProtect позволяет изменить код доступа только для тех страниц, которые созданы вызывающим ее процессом. При необходимости приложение может изменить код доступа страниц другого процесса, имеющего код доступа PROCESS_VM_OPERATION (например, процесса, созданного приложением). Это можно сделать при помощи функции VirtualProtectEx, прототип которой представлен ниже:

```
BOOL VirtualProtectEx(
  HANDLE hProcess,       // идентификатор процесса
  LPVOID lpvAddress,     // адрес области памяти
  DWORD  cbSize,         // размер области памяти в байтах
  DWORD  fdwNewProtect,  // новый тип разрешенного доступа
  PDWORD pfdwOldProtect); // указатель на переменную,
          // в которую будет записан прежний код доступа
```

Через параметр hProcess функции VirtualProtectEx следует передать идентификатор процесса. Подробнее об этом идентификаторе вы узнаете из главы, посвященной мультизадачности в операционной системе Microsoft Windows NT.

Получение информации об использовании виртуальной памяти

В программном интерфейсе Microsoft Windows NT есть средства для получения справочной информации об использовании процессами виртуальной памяти. Это функции VirtualQuery и VirtualQueryEx. С помощью них процесс может исследовать, соответственно, собственное адресное пространство и адресное пространство других процессов, определяя такие характеристики областей виртуальной памяти, как размеры, тип доступа, состояние и так далее. Из-за ограниченного объема книги мы не будем рассматривать эти функции. При необходимости вы сможете найти их подробное описание в справочной системе, поставляющейся вместе с SDK или системой разработки Microsoft Visual C++.

С помощью приложения Process Walker, которое поставляется в составе SDK, вы сможете визуально исследовать распределение виртуальной памяти для процессов. На рис. 1.12 показан фрагмент такого распределения для приложения CALC.EXE. Приложение вызывает функции VirtualQuery и VirtualQueryEx (исходные тексты приложения поставляется в составе SDK; вы найдете их в каталоге win32sdk\mstools\samples\sdktools\winnt\pviewer).

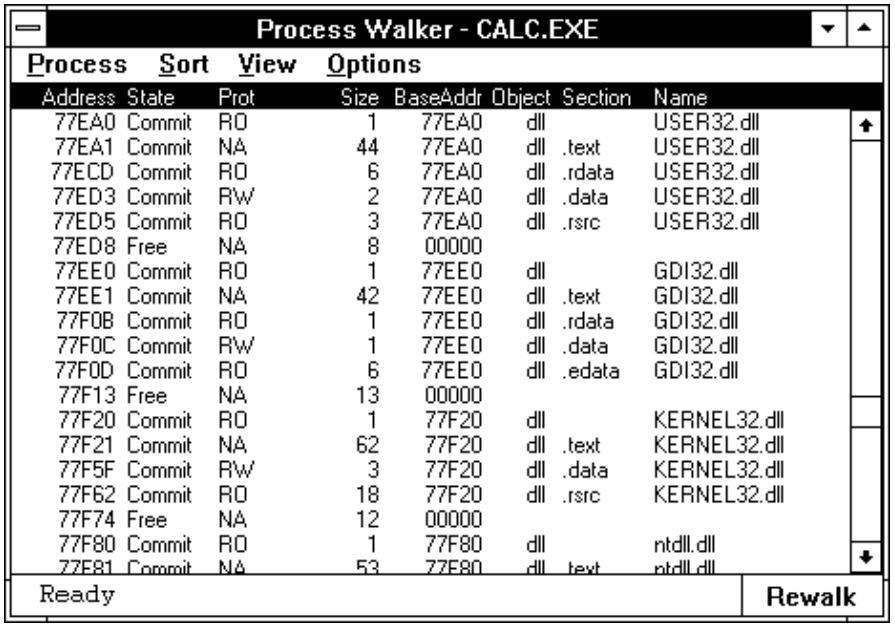

Рис. 1.12. Исследование распределения виртуальной памяти при помощи приложения Process Walker

Для загрузки и исследования процесса вы должны выбрать из меню Process строку Load Process. Затем при помощи диалоговой панели Open executable image следует выбрать загрузочный файл нужного вам приложения. В окне появится распределение памяти в виде таблицы.

В столбце Address отображается логический адрес областей памяти. Обратите внимание, что в модели памяти FLAT он состоит только из смещения. Базовый адрес отображается в столбце BaseAddr и имеет смысл только для зарезервированных (Reserve) или готовых к использованию (Commit) страниц. Для свободных страниц (Free) он равен нулю.

В столбце Prot в виде двухбуквенного сокращения имен сответствующих констант отображается тип доступа, разрешенный для страниц. Например, страницы, доступные только на чтение, отмечены в этом столбце как RO (PAGE_READONLY).

В линейном адресном пространстве процесса находятся не только код и данные самого процесса. В него также отображаются страницы системных библиотек динамической загрузки DLL (как это видно из рис. 1.12). При этом в столбце Object может отображаться тип объекта (библиотека DLL или

исполняемый EXE-модуль), а в столбцах Section и Name, соответственно, имя секции и имя библиотеки DLL.

Если сделать двойной щелчок левой клавишей мыши по строке, соответствующей страницам памяти, отмеченным как Commit, на экране появится окно с дампом содержимого этих страниц (рис. 1.13).

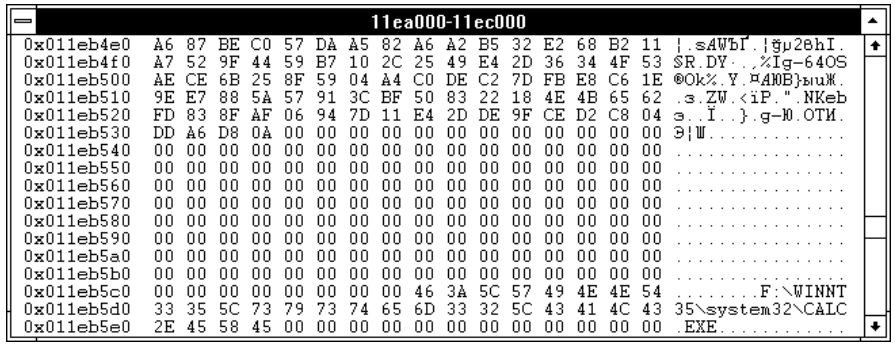

Рис. 1.13. Просмотр содержимого страниц, готовых для использования

Вы можете использовать приложение Process Walker для отладки создаваемых вами приложений, например, контролируя использование ими виртуальной памяти.

Приложение VIRTUAL

Для демонстрации описанных выше функций, работающих со страницами виртуальной памяти, мы подготовили исходные тексты приложения VIRTUAL. Это приложение получает из адресного пространства приложения одну страницу виртуальной памяти и позволяет вам вручную устанавливать тип доступа для нее, проверяя результат на операциях чтения, записи и фиксирования страницы.

При помощи меню Set protection (рис. 1.14) вы можете установить для полученной страницы памяти тип доступа PAGE_NOACCES, PAGE_READONLY, PAGE_READWRITE, а также комбинацию типов доступа PAGE_READWRITE и PAGE_GUARD.

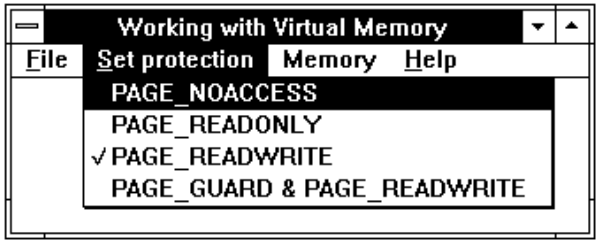

Рис. 1.14. Меню Set protection, предназначенное для установки типа доступа

При помощи строк меню Memory (рис. 1.15) вы можете выполнять над полученной страницей памяти операции чтения, записи, фиксирования и расфиксирования (соответственно, строки Read, Write, Lock и Unlock).

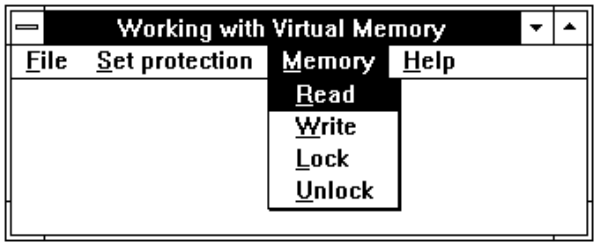

Рис. 1.15. Меню Memory, предназначенное для выполнения различных операций над страницей памяти

Первоначально для страницы устанавливается код доступа PAGE_NOACCES, запрещающий любой доступ, поэтому при выборе из меню Memory любой строки вы получите сообщение об ошибке.

Если установить код доступа PAGE_READONLY, то, как и следовало ожидать, вы сможете выполнить только операции чтения, фиксирования и расфиксирования страницы. Тип доступа PAGE_READWRITE дополнительно разешает выполнение операции записи.

В том случае, когда для страницы устанволена комбинация типов доступа PAGE_READWRITE и PAGE_GUARD, при первой попытке выполнения над этой страницей любой операции появляется сообщение об ошибке, так как возникает исключение. Во второй раз эта же операция выполнится без ошибки, так как после возникновения исключения тип доступа PAGE_GUARD сбрасывается автоматически.

Когда любое исключение возникает в 16-разрядном приложении Microsoft Windows версии 3.1, оно завершает свою работу с сообщением о фатальной ошибке. С этим не может ничего сделать ни пользователь, ни программист, создающий такие приложения. Что же касается операционной системы

Microsoft Windows NT, то приложение может самостоятельно выполнять обработку исключений.

На рис. 1.16 показано сообщение, которое возникло бы на экране при попытке приложения, не обрабатывающего исключения, обратиться к странице с типом доступа PAGE_NOACCES для чтения. Прочитав его, пользователь может нажимать кнопку OK, что приведет к аварийному завершению работы приложения.

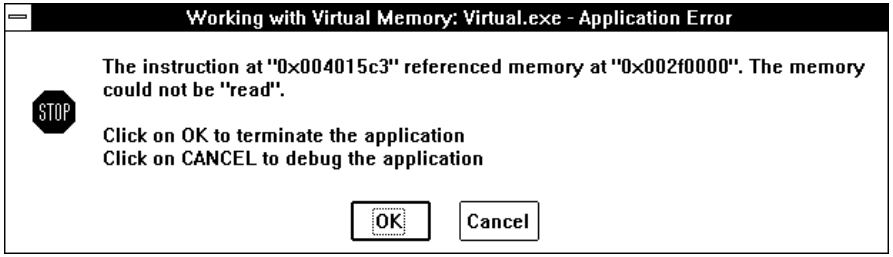

Рис. 1.16. Сообщение, возникающее на экране при попытке обращения к странице с типом доступа PAGE_NOACCES

Информация, представленная в этом сообщении, ничего не дает пользователю. Не сможет он воспользоваться и предложением запустить отладчик, нажав кнопку Cancel, так как едва ли у него есть исходные тексты приложения и, что самое главное, желание разбираться с ними.

Аналогично, при попытке приложения обратиться к странице с типом доступа PAGE_GUARD, на экране появится сообщение, показанное на рис. 1.17.

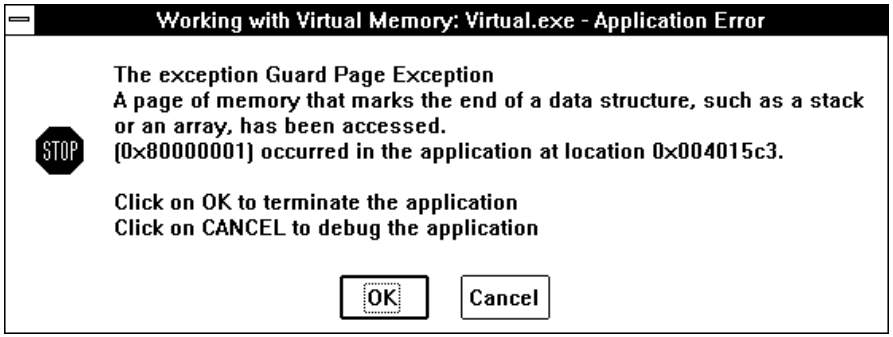

Рис. 1.17. Сообщение, возникающее при попытке обращения к странице с типом доступа PAGE_GUARD

Наше приложение VIRTUAL способно обрабатывать исключения, поэтому даже при попытках выполнения неразрешенного вида доступа все ограничивается выводом соответствующего сообщения, после чего работа приложения может быть продолжена. На рис. 1.18 показано такое сообщение, возникающее при обращении к памяти с типом доступа PAGE_GUARD.

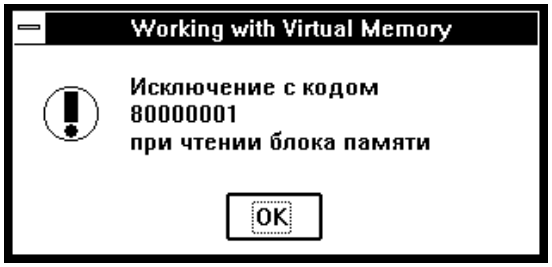

Рис. 1.18. Сообщение об исключении, отображаемое приложением VIRTUAL

Обработку исключений мы рассмотрим в отдельной главе одной из следующих наших книг, посвященных операционной системе Microsoft Windows NT, а сейчас только скажем, что при ее использовании вы можете значительно повысить устойчивость работы приложения, встроив в него мощную систему проверки (и, в некоторых случаях, даже исправления) различных ошибок.

Исходные тексты приложения

Основной файл исходных текстов приложения VIRTUAL представлен в листинге 1.1. Заметим, что вы можете приобрести дискету, которая продается вместе с нашей книгой, избавив себя и от необходимости набирать исходные тексты приложений вручную, и от ошибок, неизбежно возникающих при этом.

Листинг 1.1. Файл virtual/virtual.c

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <stdio.h>
#include "resource.h"
#include "afxres.h"
#include "virtual.h"
```

```
// Размер блока виртуальной памяти, над которым будут
// выполняться операции
#define MEMBLOCK_SIZE 4096

HINSTANCE hInst;
char szAppName[]  = "VirtualApp";
char szAppTitle[] = "Working with Virtual Memory";

// Указатель на блок памяти
LPVOID lpMemoryBuffer;

// Идентификатор меню Set protection
HMENU hSetMenu;

// -------------------------------------------------------
// Функция WinMain
// -------------------------------------------------------
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
  WNDCLASSEX wc;
  HWND hWnd;
  MSG msg;

  // Сохраняем идентификатор приложения
  hInst = hInstance;

  // Преверяем, не было ли это приложение запущено ранее
  hWnd = FindWindow(szAppName, NULL);
  if(hWnd)
  {
    // Если было, выдвигаем окно приложения на
    // передний план
    if(IsIconic(hWnd))
      ShowWindow(hWnd, SW_RESTORE);
    SetForegroundWindow(hWnd);
    return FALSE;
  }

  // Регистрируем класс окна
  memset(&wc, 0, sizeof(wc));
  wc.cbSize = sizeof(WNDCLASSEX);
  wc.hIconSm = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICONSM), IMAGE_ICON, 16, 16, 0);
  wc.style = 0;
  wc.lpfnWndProc = (WNDPROC)WndProc;
  wc.cbClsExtra  = 0;
  wc.cbWndExtra  = 0;
  wc.hInstance = hInst;
  wc.hIcon = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICON), IMAGE_ICON, 32, 32, 0);
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
  wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
  wc.lpszClassName = szAppName;
  if(!RegisterClassEx(&wc))
    if(!RegisterClass((LPWNDCLASS)&wc.style))
      return FALSE;

  // Создаем главное окно приложения
  hWnd = CreateWindow(szAppName, szAppTitle,
    WS_OVERLAPPEDWINDOW, CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInst, NULL);
  if(!hWnd) return(FALSE);

  // Отображаем окно и запускаем цикл
  // обработки сообщений
  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  while(GetMessage(&msg, NULL, 0, 0))
  {
```

```
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return msg.wParam;
}

// -------------------------------------------------------
// Функция WndProc
// -------------------------------------------------------
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
  switch(msg)
  {
    HANDLE_MSG(hWnd, WM_CREATE,  WndProc_OnCreate);
    HANDLE_MSG(hWnd, WM_DESTROY, WndProc_OnDestroy);
    HANDLE_MSG(hWnd, WM_COMMAND, WndProc_OnCommand);

    default:
      return(DefWindowProc(hWnd, msg, wParam, lParam));
  }
}

// -------------------------------------------------------
// Функция WndProc_OnCreate
// -------------------------------------------------------
BOOL WndProc_OnCreate(HWND hWnd,
                      LPCREATESTRUCT lpCreateStruct)
{
  // Временный указатель
  LPVOID lpReserved;

  // Резервируем страницы виртуальной памяти
  lpReserved = VirtualAlloc(NULL, MEMBLOCK_SIZE,
    MEM_RESERVE, PAGE_NOACCESS);

  // При ошибке завершаем работу приложения
  if(lpReserved == NULL)
  {
    MessageBox(hWnd,
      "Ошибка при резервировании памяти",
       szAppTitle, MB_OK | MB_ICONEXCLAMATION);

    return FALSE;
  }

  // Получаем память в пользование
  lpMemoryBuffer = VirtualAlloc(lpReserved,
    MEMBLOCK_SIZE, MEM_COMMIT, PAGE_NOACCESS);

  // При ошибке освобождаем зарезервированную ранее
  // память и завершаем работу приложения
  if(lpMemoryBuffer == NULL)
  {
    MessageBox(hWnd,
      "Ошибка при получении памяти для использования",
       szAppTitle, MB_OK | MB_ICONEXCLAMATION);

    VirtualFree(lpReserved, 0, MEM_RELEASE);
    return FALSE;
  }

  // Получаем идентификатор меню Set protection
  hSetMenu = GetSubMenu(GetMenu(hWnd), 1);

  // Отмечаем строку PAGE_NOACCESS этого меню
  CheckMenuItem(hSetMenu,
    ID_SETPROTECTION_PAGENOACCESS, MF_CHECKED);

  return TRUE;
}
```

```
// ------------------------------------------------------
// Функция WndProc_OnDestroy
// ------------------------------------------------------
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{

  // Перед завершением работы приложения освобождаем
  // полученную ранее память
  if(lpMemoryBuffer != NULL)
    VirtualFree(lpMemoryBuffer, 0, MEM_RELEASE);

  PostQuitMessage(0);
  return 0L;
}

// ------------------------------------------------------
// Функция WndProc_OnCommand
// ------------------------------------------------------
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id,
  HWND hwndCtl, UINT codeNotify)
{
  int   test;
  DWORD dwOldProtect;
  char  chBuff[256];

  switch (id)
  {
    // Устанавливаем тип доступа PAGE_NOACCESS
    case ID_SETPROTECTION_PAGENOACCESS:
    {
      VirtualProtect(lpMemoryBuffer, MEMBLOCK_SIZE,
        PAGE_NOACCESS, &dwOldProtect);

      // Отмечаем строку PAGE_NOACCESS меню Set protection
      CheckMenuItem(hSetMenu,
        ID_SETPROTECTION_PAGENOACCESS, MF_CHECKED);

      // Убираем отметку с других строк меню Set protection
      CheckMenuItem(hSetMenu,
        ID_SETPROTECTION_PAGEREADONLY, MF_UNCHECKED);
      CheckMenuItem(hSetMenu,
        ID_SETPROTECTION_PAGEREADWRITE, MF_UNCHECKED);
      CheckMenuItem(hSetMenu,
        ID_SETPROTECTION_PAGEGUARD, MF_UNCHECKED);

      break;
    }

    // Устанавливаем тип доступа PAGE_READONLY
    case ID_SETPROTECTION_PAGEREADONLY:
    {
      VirtualProtect(lpMemoryBuffer, MEMBLOCK_SIZE,
        PAGE_READONLY, &dwOldProtect);

      // Отмечаем строку PAGE_READONLY меню Set protection
      CheckMenuItem(hSetMenu,
        ID_SETPROTECTION_PAGEREADONLY, MF_CHECKED);

      // Убираем отметку с других строк меню Set protection
      CheckMenuItem(hSetMenu,
        ID_SETPROTECTION_PAGENOACCESS, MF_UNCHECKED);
      CheckMenuItem(hSetMenu,
        ID_SETPROTECTION_PAGEREADWRITE, MF_UNCHECKED);
      CheckMenuItem(hSetMenu,
        ID_SETPROTECTION_PAGEGUARD, MF_UNCHECKED);
      break;
    }

    // Устанавливаем тип доступа PAGE_READWRITE
```

```
case ID_SETPROTECTION_PAGEREADWRITE:
{
 VirtualProtect(lpMemoryBuffer, MEMBLOCK_SIZE,
  PAGE_READWRITE, &dwOldProtect);

 // Отмечаем строку PAGE_READWRITE меню Set protection
 CheckMenuItem(hSetMenu,
  ID_SETPROTECTION_PAGEREADWRITE, MF_CHECKED);

 // Убираем отметку с других строк меню Set protection
 CheckMenuItem(hSetMenu,
  ID_SETPROTECTION_PAGENOACCESS, MF_UNCHECKED);
 CheckMenuItem(hSetMenu,
  ID_SETPROTECTION_PAGEREADONLY, MF_UNCHECKED);
 CheckMenuItem(hSetMenu,
  ID_SETPROTECTION_PAGEGUARD, MF_UNCHECKED);
 break;
}

// Устанавливаем тип доступа
//   PAGE_READWRITE | PAGE_GUARD
case ID_SETPROTECTION_PAGEGUARD:
{
 VirtualProtect(lpMemoryBuffer, MEMBLOCK_SIZE,
  PAGE_READWRITE | PAGE_GUARD, &dwOldProtect);

 // Отмечаем строку PAGE_READWRITE & PAGE_GUARD
 // меню Set protection
 CheckMenuItem(hSetMenu,
  ID_SETPROTECTION_PAGEGUARD, MF_CHECKED);

 // Убираем отметку с других строк меню Set protection
 CheckMenuItem(hSetMenu,
  ID_SETPROTECTION_PAGENOACCESS, MF_UNCHECKED);
 CheckMenuItem(hSetMenu,
  ID_SETPROTECTION_PAGEREADONLY, MF_UNCHECKED);
 CheckMenuItem(hSetMenu,
  ID_SETPROTECTION_PAGEREADWRITE, MF_UNCHECKED);
 break;
}

// Выполняем попытку чтения
case ID_MEMORY_READ:
{
 __try
 {
  test = *((int *)lpMemoryBuffer);
 }

 // Если возникло исключение, получаем и
 // отображаем его код
 __except (EXCEPTION_EXECUTE_HANDLER)
 {
  sprintf(chBuff, "Исключение с кодом\n"
   "%lX\nпри чтении блока памяти",
   GetExceptionCode());

  MessageBox(hWnd, chBuff,
   szAppTitle, MB_OK | MB_ICONEXCLAMATION);
  break;
 }

 // Если операция завершилась успешно,
 // сообщаем об этом пользователю
 MessageBox(hWnd, "Чтение выполнено",
  szAppTitle, MB_OK | MB_ICONEXCLAMATION);
 break;
}

// Выполняем попытку записи
case ID_MEMORY_WRITE:
{
```

```
  __try
  {
   *((int *)lpMemoryBuffer) = 1;
  }

  // Если возникло исключение, получаем и
  // отображаем его код
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
   sprintf(chBuff, "Исключение с кодом\n"
     "%lX\nпри записи в блок памяти",
     GetExceptionCode());

   MessageBox(hWnd, chBuff,
     szAppTitle, MB_OK | MB_ICONEXCLAMATION);
   break;
  }

  // Если операция завершилась успешно,
  // сообщаем об этом пользователю
  MessageBox(hWnd, "Запись выполнена",
    szAppTitle, MB_OK | MB_ICONEXCLAMATION);
  break;
}

// Выполняем попытку фиксирования блока памяти
case ID_MEMORY_LOCK:
{
 if(VirtualLock(lpMemoryBuffer,
   MEMBLOCK_SIZE) == FALSE)
 {
   MessageBox(hWnd,
     "Ошибка при фиксировании страниц в памяти",
     szAppTitle, MB_OK | MB_ICONEXCLAMATION);
   break;
 }
  MessageBox(hWnd, "Фиксирование выполнено",
    szAppTitle, MB_OK | MB_ICONEXCLAMATION);
  break;
}

// Выполняем попытку расфиксирования блока памяти
case ID_MEMORY_UNLOCK:
{
 if(VirtualUnlock(lpMemoryBuffer,
   MEMBLOCK_SIZE) == FALSE)
 {
   MessageBox(hWnd,
     "Ошибка при расфиксировании страниц в памяти",
     szAppTitle, MB_OK | MB_ICONEXCLAMATION);
   break;
 }

 MessageBox(hWnd, "Расфиксирование выполнено",
   szAppTitle, MB_OK | MB_ICONEXCLAMATION);
 break;
}

case ID_FILE_EXIT:
{
 // Завершаем работу приложения
 PostQuitMessage(0);
 return 0L;
 break;
}

case ID_HELP_ABOUT:
{
 MessageBox(hWnd,
   "Working with Virtual Memory\n"
   "(C) Alexandr Frolov, 1996\n"
```

```
      "Email: frolov@glas.apc.org",
      szAppTitle, MB_OK | MB_ICONINFORMATION);
    return 0L;
    break;
   }
  default:
   break;
 }
 return FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify,
  DefWindowProc);
}
```

В файле virtual.h (листинг 1.2) представлены прототипы функций, определенных в нашем приложении.

Листинг 1.2. Файл virtual/virtual.h

```
LRESULT WINAPI
 WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
void WndProc_OnCommand(HWND hWnd, int id,
 HWND hwndCtl, UINT codeNotify);
```

Файл resource.h (листинг 1.3) создается автоматически системой разработки Microsoft Visual C++ и содержит определения констант, использованных в файле описания ресурсов приложения.

Листинг 1.3. Файл virtual/resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Virtual.RC
//
#define IDR_APPMENU                 102
#define IDI_APPICON                 103
#define IDI_APPICONSM               104
#define ID_FILE_EXIT                40001
#define ID_HELP_ABOUT               40003
#define ID_SETPROTECTION_PAGENOACCESS   40035
#define ID_SETPROTECTION_PAGEREADONLY   40036
#define ID_SETPROTECTION_PAGEREADWRITE  40037
#define ID_SETPROTECTION_PAGEGUARD      40038
#define ID_MEMORY_READ              40039
#define ID_MEMORY_WRITE             40040
#define ID_MEMORY_LOCK              40041
#define ID_MEMORY_UNLOCK            40042

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        121
#define _APS_NEXT_COMMAND_VALUE         40043
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

Файл описания ресурсов приложения virtual.rc (листинг 1.4) также автоматически создается системой разработки Microsoft Visual C++. В нем определены пиктограммы приложения, меню и текстовые строки (которые мы в данном случае не используем).

Листинг 1.4. Файл virtual/virtual.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

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

```
#undef APSTUDIO_READONLY_SYMBOLS

/////////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

/////////////////////////////////////////////////////////////////
// Menu
//
IDR_APPMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit", ID_FILE_EXIT
    END
    POPUP "&Set protection"
    BEGIN
      MENUITEM "PAGE_NOACCESS", ID_SETPROTECTION_PAGENOACCESS
      MENUITEM "PAGE_READONLY", ID_SETPROTECTION_PAGEREADONLY
      MENUITEM "PAGE_READWRITE",ID_SETPROTECTION_PAGEREADWRITE
      MENUITEM "PAGE_GUARD && PAGE_READWRITE",
          ID_SETPROTECTION_PAGEGUARD
    END
    POPUP "&Memory"
    BEGIN
        MENUITEM "&Read",           ID_MEMORY_READ
        MENUITEM "&Write",          ID_MEMORY_WRITE
        MENUITEM "&Lock",           ID_MEMORY_LOCK
        MENUITEM "&Unlock",             ID_MEMORY_UNLOCK
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",       ID_HELP_ABOUT
    END
END

#ifdef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////
// TEXTINCLUDE
//
1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

/////////////////////////////////////////////////////////////////
// Icon

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_APPICON         ICON    DISCARDABLE     "virtual.ico"
IDI_APPICONSM       ICON    DISCARDABLE     "virtsm.ico"
```

```
//////////////////////////////////////////////////////////////
// String Table
//
STRINGTABLE DISCARDABLE
BEGIN
    ID_FILE_EXIT            "Quits the application"
END

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 3 resource.
//
//////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Описание исходных текстов приложения

Если вы читали нашу книгу “Операционная система Microsoft Windows 95 для программиста” (22 том “Библиотеки системного программиста”), то вы уже сталкивались с 32-разрядными приложениями, работающими в сплошной модели памяти. При создании приложения VIRTUAL мы использовали те же инструментальные средства, что и в упомянутой книге, а именно Microsoft Visual C++ версии 4.0.

Определения и глобальные переменные

Константа MEMBLOCK_SIZE определяет размер блока памяти в байтах, над которым выполняются операции. Мы работаем с блоком памяти размером 4096 байт, что соответствует одной странице, однако вы можете попробовать и другие значения. Например, вы можете попытаться задать очень большие значения и убедиться в том, что при этом хотя память и будет предоставлена в ваше распоряжение, вы не сможете выполнить фиксирование соответствующих страниц.

В глобальных переменных hInst, szAppName и szAppTitle хранится, соответственно, идентификатор приложения, полученный при запуске через параметр hInstance функции WinMain, имя и заголовок главного окна приложения.

Глобальная переменная lpMemoryBuffer содержит указатель на блок виртуальной памяти размером MEMBLOCK_SIZE байт. Это именно тот блок памяти, с которым в нашем приложении выполняются все операции.

И, наконец, в глобальной переменной hSetMenu хранится идентификатор меню Set protection, который необходим для выполнения операции отметки строк этого меню.

Функция WinMain

Изучая исходный текст функции WinMain, вы по большому счету не найдете никаких отличий от исходных текстов этой функции, приведенных в приложениях Microsoft Windows 95 из 22 тома “Библиотеки системного программиста”. В этом нет ничего удивительного, так как и в Microsoft Windows 95, и в Microsoft Windows NT реализован 32-разрядный интерфейс WIN32, которым мы должны пользоваться. Более того, разрабатывая приложения для Microsoft Windows 95, вы должны проверять их работоспособность в среде Microsoft Windows NT, так как в противном случае вы не сможете стать соискателем логотипа “Designed for Microsoft Windows 95”.

Итак, вернемся к нашей функции WinMain.

Прежде всего эта функция сохраняет идентификатор приложения hInstance в глобальной переменной hInst для дальнейшего использования.

Затем выполняется поиск нашего приложения среди уже запущенных. При этом используется функция FindWindow. Если эта функция среди активных приложений найдет приложение с именем szAppName, она возвратит идентификатор его окна, если нет - значение NULL.

Если приложение найдено и его окно не свернуто в пиктограмму (что проверяется при помощи функции IsIconic), оно выдвигается на передний план при помощи функции SetForegroundWindow. После этого функция WinMain завершает свою работу.

В среде Microsoft Windows NT, так же как и в среде Microsoft Windows 95, мы не можем проверить, было ли наше приложение запущенно ранее, анализируя значение параметра hPrevInstance функции WinMain, так как в отличие от Microsoft Windows версии 3.1, в указанных операционных системах через этот параметр всегда передается значение NULL. Об этом мы подробно говорили в 22 томе “Библиотеки системного программиста”.

Если приложение не найдено, выполняется регистрация класса окна. В отличие от операционной системы Microsoft Windows версии 3.1, в Microsoft Windows 95 и Microsoft Windows NT следует использовать регистрацию при помощи функции RegisterClassEx, подготовив для нее структуру WNDCLASSEX.

По сравнению со знакомой вам структурой WNDCLASS операционной системы Microsoft Windows версии 3.1 в структуре WNDCLASSEX имеются два дополнительных поля с именами cbSize и hIconSm. В первое из них необходимо записать размер структуры WNDCLASSEX, а во второе -

идентификатор пиктограммы малого размера. Эта пиктограмма в Microsoft Windows NT версии 4.0 будет отображаться в левом верхнем углу главного окна приложения, играя роль пиктограммы системного меню.

Для загрузки пиктограмм из ресурсов приложения мы воспользовались функцией LoadImage, описанной нами в 22 томе “Библиотеки системного программиста”.

Заметим, что если вызов функции регистрации класса окна RegisterClassEx закончился неудачно (это может произойти, например, если вы запустите приложение в среде Microsoft Windows NT версии 3.5 или более ранней версии), мы выполняем регистрацию старым способом, который тоже работает, - при помощи функции RegisterClass.

После регистрации функция WinMain создает главное окно приложения, отображает и обновляет его, а затем запускает цикл обработки сообщений. Все эти процедуры мы описывали в 11 томе “Библиотеки системного программиста”, который называется “Операционная система Microsoft Windows 3.1 для программиста. Часть первая”.

Функция WndProc

Функция WndProc обрабатывает сообщения WM_CREATE, WM_DESTROY и WM_COMMAND. Соответствующие функции обработки WndProc_OnCreate, WndProc_OnDestroy и WndProc_OnCommand назначаются для этих сообщений при помощи макрокоманды HANDLE_MSG. Этот метод обработки сообщений был нами подробно рассмотрен в 22 томе “Библиотеки системного программиста”, поэтому сейчас мы не будем его описывать.

Функция WndProc_OnCreate

Напомним, что при создании окна его функции окна передается сообщение WM_CREATE. Функция WndProc_OnCreate, определенная в нашем приложении, выполняет обработку этого сообщения.

Прежде всего, функция резервирует область виртуальной памяти размером MEMBLOCK_SIZE байт, вызывая функцию VirtualAlloc с параметром MEM_RESERVE:

```
lpReserved = VirtualAlloc(NULL, MEMBLOCK_SIZE,
 MEM_RESERVE, PAGE_NOACCESS);
```

Через первый параметр мы передаем функции VirtualAlloc значение NULL, поэтому операционная система сама определит для нас начальный адрес резервируемой области. Этот адрес мы сохраняем во временной локальной переменной lpReserved.

В случае ошибки выводится соответствующее сообщение. Если же резервирование адресного пространства выполнено успешно, функция получает память в использование, вызывая для этого функцию VirtualAlloc еще раз, но уже с параметром MEM_COMMIT:

```
lpMemoryBuffer = VirtualAlloc(lpReserved,
 MEMBLOCK_SIZE, MEM_COMMIT, PAGE_NOACCESS);
```

Так как в качестве первого параметра функции VirtualAlloc передается значение lpReserved, выделение страниц памяти выполняется в зарезервированной ранее области адресов.

При невозможности получения памяти в диапазоне зарезервированных адресов мы отдаем зарезервированные адреса системе и завершаем работу приложения, запрещая создание его главного окна:

```
VirtualFree(lpReserved, 0, MEM_RELEASE);
return FALSE;
```

Заметим, что мы могли бы и не вызывать функцию VirtualFree, так как после завершения процесса операционная система Microsoft Windows NT автоматически освобождает все распределенные для него ранее страницы виртуальной памяти.

Последнее, что делает обработчик сообщения WM_CREATE, это получение идентификатора меню Set protection и отметку в этом меню строки PAGE_NOACCESS:

```
hSetMenu = GetSubMenu(GetMenu(hWnd), 1);
CheckMenuItem(hSetMenu,
 ID_SETPROTECTION_PAGENOACCESS, MF_CHECKED);
```

Использованные при этом функции были описаны в главе “Меню” 13 тома “Библиотеки системного программиста”, который называется “Операционная система Microsoft Windows 3.1 для программиста. Часть третья”.

Функция WndProc_OnDestroy

Обработчик сообщения WM_DESTROY проверяет содержимое указателя lpMemoryBuffer и, если оно не равно NULL, освобождает память при помощи функции VirtualFree:

```
if(lpMemoryBuffer != NULL)
 VirtualFree(lpMemoryBuffer, 0, MEM_RELEASE);
```

Как мы уже говорили, при завершении работы приложения полученная память будет освобождена операционной системой. Однако хороший стиль программирования предполагает освобождение ресурсов, которые больше не нужны приложению, поэтому мы вызываем эту функцию сами.

После освобождения памяти приложение вызывает функцию PostQuitMessage, что приводит к завершению цикла обработки сообщений и, следовательно, к завершению работы нашего приложения.

Функция WndProc_OnCommand

Функция WndProc_OnCommand обрабатывает сообщение WM_COMMAND, поступающее от главного меню приложения. Выбирая строки меню Set protection, пользователь может изменять тип доступа, разрешенного для блока памяти, заказанного приложением при обработке

сообщения WM_CREATE. Меню Memory позволяет пользователю выполнять над этим блоком операции чтения, записи, фиксирования и расфиксирования.

Изменение типа доступа выполняется при помощи функции VirtualProtect. Например, установка типа доступа PAGE_NOACCESS выполняется следующим образом:

```
VirtualProtect(lpMemoryBuffer, MEMBLOCK_SIZE,
  PAGE_NOACCESS, &dwOldProtect);
```

При этом старый тип доступа записывается в переменную dwOldProtect, но никак не используется нашим приложением.

После изменения типа доступа обработчик сообщения WM_COMMAND изменяет соответствующим образом отметку строк меню Set protection, для чего используется макрокоманда CheckMenuItem.

Теперь рассмотрим обработку сообщения WM_COMMAND в том случае, когда оно приходит от меню Memory.

Если пользователь выполняет попытку чтения блока памяти, выбирая из меню Memory строку Read, выполняется следующий фрагмент кода:

```
case ID_MEMORY_READ:
{
  __try
  {
    test = *((int *)lpMemoryBuffer);
  }
  __except (EXCEPTION_EXECUTE_HANDLER)
  {
    sprintf(chBuff, "Исключение с кодом\n"
      "%lX\nпри чтении блока памяти", GetExceptionCode());
    MessageBox(hWnd, chBuff,
      szAppTitle, MB_OK | MB_ICONEXCLAMATION);
    break;
  }
  MessageBox(hWnd, "Чтение выполнено",
    szAppTitle, MB_OK | MB_ICONEXCLAMATION);
  break;
}
```

Здесь в области действия оператора __try, ограниченной фигурными скобками, содержимое первого слова буфера lpMemoryBuffer читается во временную переменную test. Эта, безопасная на первый взгляд операция может привести в приложении Microsoft Windows NT к возникновению исключения, так как соответствующая страница памяти может оказаться недоступной для чтения. Если не предусмотреть обработку исключения, при его возникновении работа приложения завершится аварийно.

Тело обработчика исключения находится в области действия оператора __except и выполняется только при возникновении исключения. В нашем случае обработка исключения очень проста и заключается в отображении сообщения с кодом исключения, полученного при помощи функции GetExceptionCode.

Таким образом, если исключение возникнет, пользователь увидит сообщение с кодом исключения, а если нет - на экране появится сообщение о том, что чтение выполнено.

Попытка записи выполняется при выборе из меню Memory строки Write. Вот фрагмент кода, выполняющий запись:

```
__try
{
  *((int *)lpMemoryBuffer) = 1;
}
```

Так как при записи могут возникать исключения, мы предусмотрели обработчик, аналогичный только что рассмотренному.

Когда пользователь выбирает из меню Memory строку Lock, выполняется попытка зафиксировать страницы блока памяти при помощи функции VirtualLock, как это показано ниже:

```
case ID_MEMORY_LOCK:
{
  if(VirtualLock(lpMemoryBuffer, MEMBLOCK_SIZE) == FALSE)
  {
    MessageBox(hWnd,
      "Ошибка при фиксировании страниц в памяти",
      szAppTitle, MB_OK | MB_ICONEXCLAMATION);
    break;
  }
  MessageBox(hWnd, "Фиксирование выполнено",
    szAppTitle, MB_OK | MB_ICONEXCLAMATION);
  break;
}
```

Обратите внимание, что в данном случае мы не выполняем обработку исключений, но проверяем код возврата функции VirtualLock. При попытке зафиксировать страницу с кодом доступа PAGE_NOACCESS не произойдет аварийного завершения работы приложения. В этом случае функция VirtualLock просто вернет значение FALSE, означающее ошибку.

Расфиксирование страниц блока памяти выполняется аналогичным образом, когда пользователь выбирает из меню Memory строку Unlock.

Работа с пулами памяти

Функции, предназначенные для работы с виртуальной памятью, которые мы рассмотрели выше, обычно используют для получения в пользование блоков памяти относительно большого размера (больше одной страницы). Однако наиболее часто приложению требуется всего несколько десятков байт, например, для создания динамических структур или для загрузки ресурсов приложения в оперативную память. Очевидно, оперируя с отдельными страницами, вы едва ли сможете легко и эффективно работать с блоками памяти небольшого объема.

Поэтому в программном интерфейсе Microsoft Windows NT были предусмотрены другие функции, к изучению которых мы и переходим. Все эти функции вызывают только что рассмотренные нами функции, работающие с виртуальной памятью.

Пулы памяти в Microsoft Windows NT

Как мы уже говорили, приложения Microsoft Windows версии 3.1 могли заказывать память из двух областей или двух пулов - из глобального пула, доступного всем приложениям, и локального, создаваемого для каждого приложения.

Адресные пространства приложений Microsoft Windows NT разделены, поэтому в этой операционной системе нет глобальных пулов памяти. Вместо этого каждому приложению по умолчанию выделяется один стандартный пул памяти в его адресном пространстве. При необходимости приложение может создавать (опять же в своем адресном пространстве) произвольное количество так называемых динамических пулов памяти.

По умолчанию для стандартного пула резервируется 1 Мбайт сплошного адресного пространства, причем 4 Кбайта памяти выделяются приложению для непосредственного использования. Если приложению требуется больше памяти, в адресном пространстве резервируется еще один или несколько Мбайт памяти.

Если ваше приложение работает с большими объемами данных, для загрузки этих данных в непрерывное адресное пространство можно увеличить размер стандартного пула двумя способами.

Во-первых, параметры стандартного пула можно задать в параметре /HEAP редактора связи:

```
/HEAP: 0x2000000, 0x10000
```

В данном случае для стандартного пула будет зарезервировано 2 Мбайта памяти, причем сразу после загрузки приложения 10 Кбайт памяти будет получено в пользование.

Во-вторых, параметры стандартного пула можно указать в файле определения модуля (который является необязательным). Например, так:

```
HEAPSIZE 0x2000000 0x10000
```

Однако для резервирования очень больших адресных пространств памяти лучше создавать динамические пулы. Создание динамического пула не ведет к излишней загрузке физической оперативной памяти, так как пока вы не получаете из этого пула память, соответствующее адресное пространство является зарезервированным. Как мы уже говорили, резервирование адресного пространства не вызывает выделения памяти и изменения файлов страниц. Так что резервируйте сколько угодно, но в только в пределах 2 Гбайт.

Функции для работы с пулами памяти

Итак, в распоряжении приложения Microsoft Windows NT имеется один стандартный пул и произвольное количество динамических пулов памяти.

В качестве первого парметра всем функциям, предназначенным для получения памяти из стандартного или динамического пула, необходимо передать идентификатор пула.

Получение идентификатора стандартного пула

Идентификатор стандартного пула получить очень просто. Этот идентификатор возвращает функция GetProcessHeap, не имеющая параметров:

```
HANDLE GetProcessHeap(VOID);
```

Создание динамического пула

Если вам нужен динамический пул, вы можете его создать при помощи функции HeapCreate:

```
HANDLE HeapCreate(
  DWORD  flOptions,     // флаг создания пула
  DWORD  dwInitialSize, // первоначальный размер пула в байтах
  DWORD  dwMaximumSize);// максимальный размер пула в байтах
```

Параметры dwMaximumSize и dwInitialSize определяют, соответственно, размер зарезервированной для пула памяти и размер памяти, полученной для использования.

Через параметр flOptions вы можете передать нулевое значение, а также значения HEAP_NO_SERIALIZE и HEAP_GENERATE_EXCEPTIONS.

Параметр HEAP_NO_SERIALIZE имеет отношение к мультизадачности, которая будет рассмотрена в отдельной главе нашей книги. Если этот параметр не указан, работающие параллельно задачи одного процесса не могут одновременно получать доступ к такому пулу. Вы можете использовать

флаг HEAP_NO_SERIALIZE для повышения производительности, если создаваемым вами пулом будет пользоваться только одна задача процесса.

При выделении памяти из пула могут возникать ошибочные ситуации. Если не указан флаг HEAP_GENERATE_EXCEPTIONS, при ошибках соответствующий функции будут возвращать значение NULL. В противном случае в приложении будут генерироваться исключения. Флаг HEAP_GENERATE_EXCEPTIONS удобен в тех случаях, когда в вашем приложении предусмотрена обработка исключений, позволяющая исправлять возникающие ошибки.

В случае удачи функция HeapCreate возвращает идентификатор созданного динамического пула памяти. При ошибке возвращается значение NULL (либо возникает исключение, если указан флаг HEAP_GENERATE_EXCEPTIONS).

Удаление динамического пула

Для удаления динамического пула памяти, созданного функцией HeapCreate, вы должны использовать функцию HeapDestroy:

```
BOOL HeapDestroy(HANDLE hHeap);
```

Через единственный параметр этой функции передается идентификатор удаляемого динамического пула. Заметим, что вам не следует удалять стандартный пул, передавая этой функции значение, полученное от функции GetProcessHeap.

Функция HeapDestroy выполняет безусловное удаление пула памяти, даже если из него были получены блоки памяти и на момент удаления пула они не были возвращены системе.

Получение блока памяти из пула

Для получения памяти из стандартного или динамического пула приложение должно воспользоваться функцией HeapAlloc, прототип которой мы привели ниже:

```
LPVOID HeapAlloc(
  HANDLE hHeap,    // идентификатор пула
  DWORD  dwFlags,  // управляющие флаги
  DWORD  dwBytes); // объем получаемой памяти в байтах
```

Что касается параметра hHeap, то для него вы можете использовать либо идентификатор страндартного пула памяти, полученного от функции GetProcessHeap, либо идентификатор динамического пула, созданного приложением при помощи функции HeapCreate.

Параметр dwBytes определяет нужный приложению объем памяти в байтах.

Параметр dwFlags может быть комбинацией следующих значений:

Значение	*Описание*
HEAP_GENERATE_EXCEPTIONS	Если при выполнении функции произойдет ошибка, возникнет исключение
HEAP_NO_SERIALIZE	Если указан этот флаг, не выполняется блокировака одновременного обращения к блоку памяти нескольких задач одного процесса
HEAP_ZERO_MEMORY	Выделенная память заполняется нулями

Изменение размера блока памяти

С помощью функции HeapReAlloc приложение может изменить размер блока памяти, выделенного ранее функцией HeapAlloc, уменьшив или увеличив его. Прототип функции HeapReAlloc приведен ниже:

```
LPVOID HeapReAlloc(
  HANDLE hHeap,    // идентификатор пула
  DWORD  dwFlags,  // флаг изменения размера блока памяти
  LPVOID lpMem,    // адрес блока памяти
  DWORD  dwBytes); // новый размер блока памяти в байтах
```

Для пула hHeap эта функция изменяет размер блока памяти, расположенного по адресу lpMem. Новый размер составит dwBytes байт.

В случае удачи функция HeapReAlloc возвратит адрес нового блока памяти, который не обязательно будет совпадать с адресом, полученным этой функцией через параметр lpMem.

Через параметр dwFlags вы можете передавать те же параметры, что и через аналогичный параметр для функции HeapAlloc. Дополнительно можно указать параметр HEAP_REALLOC_IN_PLACE_ONLY, определяющий, что при изменении размера блока памяти его нужно оставить на прежнем месте адресного пространства. Очевидно, что если указан этот параметр, в случае успешного завершения функция HeapReAlloc вернет то же значение, что было передано ей через параметр lpMem.

Определение размера блока памяти

Зная адрес блока памяти, полученного из пула, вы можете определить его размер при помощи функции HeapSize:

```
DWORD HeapSize(
  HANDLE  hHeap,   // идентификатор пула
  DWORD   dwFlags, // управляющие флаги
  LPCVOID lpMem);  // адрес проверяемого блока памяти
```

В случае ошибки эта функция возвращает значение 0xFFFFFFFF.

Если блоком памяти пользуется только одна задача процесса, вы можете передать через параметр dwFlags значение HEAP_NO_SERIALIZE.

Освобождение памяти

Память, выделенную с помощью функции HeapAlloc, следует освободить, как только в ней отпадет надобность. Это нужно сделать при помощи функции HeapFree:

```
BOOL HeapFree(
  HANDLE hHeap,   // идентификатор пула
  DWORD  dwFlags, // флаги освобождения памяти
  LPVOID lpMem);  // адрес освобождаемого блока памяти
```

Если блоком памяти пользуется только одна задача процесса, вы можете передать через параметр dwFlags значение HEAP_NO_SERIALIZE.

Если размер блока памяти, выделенного функцией HeapAlloc, был изменен функцией HeapReAlloc, для освобождения такого блока памяти вы все равно должны использовать функцию HeapFree.

Использование функций malloc и free

В библиотеке Microsoft Visual C++ имеются стандартные функции, предназначенные для динамического получения и освобождения памяти, такие как malloc и free.

У нас есть хорошая новость для вас - в среде Microsoft Windows NT вы можете использовать эти функции с той же эффективностью, что и функции, предназначенные для работы с пулами - HeapAlloc, HeapFree и так далее. Правда, эти функции получают память только из стандартного пула.

Одно из преимуществ функций malloc и free заключается в возможности их использования на других платформах, отличных от Microsoft Windows NT.

Старые функции управления памятью

В 32-разрядных приложениях Microsoft Windows NT вы можете пользоваться многими функциями управления памятью операционной системы Microsoft Windows версии 3.1, которые оставлены в новой операционной системе для совместимости. Мы подробно рассмотрели эти функции в главе “Управление памятью” 13 тома “Библиотеки системного программиста”.

Напомним, что в 16-разрядном программном интерфейсе Microsoft Windows версии 3.1 существует два набора функций (глобальные и локальные), предназначенных для работы с глобальным и локальным пулом памяти. Это такие функции, как GlobalAlloc, LocalAlloc, GlobalFree, LocalFree и так далее. В 32-разрядных приложениях Microsoft Windows NT вы можете пользоваться как глобальными, так и локальными функциями, причем результат будет совершенно одинаковый. Причина этого заключается в том, что все эти функции пользуются функциями программного интерфейса Microsoft Windows NT, предназначенными для работы со стандартным пулом памяти: HeapAlloc, HeapReAlloc, HeapFree и так далее.

Вот список функций старого программного интерфейса, доступных приложениям Microsoft Windows NT:

Имя функции	*Описание*
GlobalAlloc, LocalAlloc	Получение глобального (локального для функции LocalAlloc) блока памяти
GlobalReAlloc, LocalReAlloc	Изменение размера глобального (локального) блока памяти
GlobalFree, LocalFree	Освобождение глобального (локального) блока памяти
GlobalLock, LocalLock	Фиксирование глобального (локального) блока памяти
GlobalUnlock, LocalUnlock	Расфиксирование глобального (локального) блока памяти
GlobalSize, LocalSize	Определение размера глобального (локального) блока памяти
GlobalDiscard, LocalDiscard	Принудительное удаление глобального (локального) блока памяти
GlobalFlags, LocalFlags	Определение состояния глобального (локального) блока памяти
GlobalHandle, LocalHandle	Определение идентификатора глобального (локального) блока памяти

Заметим, что хотя при получении памяти с помощью функции GlobalAlloc вы по-прежнему можете указывать флаг GMEM_DDESHARE, другие приложения, запущенные в среде Microsoft Windows NT, не будут иметь к этой памяти доступ. Причина очевидна - адресные пространства приложений изолированы. Однако в документации SDK сказано, что этот флаг можно использовать для увеличения производительности приложений, использующих механизм динамической передачи сообщений DDE. Этот механизм мы подробно описали в главе “Обмен данными через DDE” в 17 томе “Библиотеки системного программиста”, который называется “Операционная система Microsoft Windows 3.1. Дополнительные главы”.

Обратим ваше внимание также на то, что в среде Microsoft Windows версии 3.1 вы могли получать фиксированную (fixed), перемещаемую (moveable) и удаляемую (discardable) память.

В среде Microsoft Windows NT вы по-прежнему можете пользоваться различными типами памяти, если для получения блоков памяти используете функции GlobalAlloc или LocalAlloc. Однако теперь вам едва ли потребуется перемещаемая память, так как новая система управления памятью выполняет операцию перемещения с помощью механизма страничной адресации, не изменяя значение логического адреса.

В том случае, если вы все же решили получить блок перемещаемой памяти, перед использованием его необходимо зафиксировать функцией GlobalLock или LocalLock (соответственно, для блоков памяти, полученных функциями GlobalAlloc и LocalAlloc). Это нужно сделать потому что если вы заказываете перемещаемый блок памяти, функции GlobalAlloc и LocalAlloc возвращают не адрес блока памяти, а его идентификатор.

Если же вы получаете фиксированный блок памяти, то функции GlobalAlloc и LocalAlloc вернут вам его адрес, который можно немедленно использовать. При этом надо иметь в виду, что операционная система сможет перемещать этот блок памяти без изменения его логического адреса.

Что же касается удаляемой памяти, то ее можно использовать для хранения таких данных, которые можно легко восстановить, например, прочитав их из ресурсов приложений.

Приложение HEAPMEM

На примере приложения HEAPMEM мы покажем вам, как можно использовать функции, предназначенные для работы с пулами памяти.

В отличие от предыдущего приложения, приложение HEAPMEM работает в так называемом консольном (или текстовом) режиме. Такое приложение может пользоваться стандартными функциями консольного ввода/вывода из библиотеки C++. Для него система создает отдельное окно (рис. 1.19).

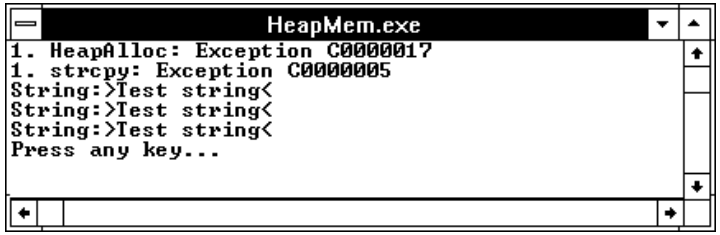

Рис. 1.19. Окно консольного приложения HEAPMEM

Что делает наше приложение?

Приложение HEAPMEM тремя различными способами решает одну и ту же задачу: получение небольшого блока памяти, запись в нее текстовой строки и отображение этой строки в консольном окне, показанном на рис. 1.19.

Первый способ предполагает использование динамического пула памяти и обработку исключений. В приложении намеренно создаются две ситуации, в которых происходят исключения с кодами C0000017 и C0000005. Во второй раз приложение работает со стандартным пулом памяти и не обрабатыает исключения, проверяя код завершения функций. И, наконец, третий способ связан с использованием функций malloc и free.

Исходный текст приложения

Исходный текст приложения HEAPMEM представлен в листинге 1.5. Файлы описания ресурсов и определения модуля не используются.

Листинг 1.5. Файл heapmem/heapmem.c

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>

int main()
{
  // Идентификатор динамического пула
  HANDLE hHeap;

  // Указатель, в который будет записан адрес
  // полученного блока памяти
  char *lpszBuff;

  // ================================================
  // Работа с динамическим пулом
  // Используем структурную обработку исключений
  // ================================================

  // Создаем динамический пул
  hHeap = HeapCreate(0, 0x1000, 0x2000);

  // Если произошла ошибка, выводим ее код и
  // завершаем работу приложения
  if(hHeap == NULL)
```

```
{
  fprintf(stdout,"HeapCreate: Error %ld\n",
    GetLastError());
  getch();
  return 0;
}

// Пытаемся получить из пула блок памяти
__try
{
  lpszBuff = (char*)HeapAlloc(hHeap,
    HEAP_GENERATE_EXCEPTIONS, 0x1500);
}

// Если память недоступна, происходит исключение,
// которое мы обрабатываем
__except (EXCEPTION_EXECUTE_HANDLER)
{
  fprintf(stdout,"1. HeapAlloc: Exception %lX\n",
    GetExceptionCode());
}

// Пытаемся записать в буфер текстовую строку
__try
{
  strcpy(lpszBuff, "Строка для проверки");
}

// Если содержимое указателя lpszBuff равно NULL,
// произойдет исключение
__except (EXCEPTION_EXECUTE_HANDLER)
{
  fprintf(stdout,"1. strcpy: Exception %lX \n",
    GetExceptionCode());
}

// Выполняем повторную попытку, указывая меньший
// размер блока памяти
__try
{
  lpszBuff = (char*)HeapAlloc(hHeap,
    HEAP_GENERATE_EXCEPTIONS, 0x100);
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
  fprintf(stdout,"2. HeapAlloc: Exception %lX\n",
    GetExceptionCode());
}

__try
{
  strcpy(lpszBuff, "Test string");
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
  fprintf(stdout,"2. strcpy: Exception %lX \n",
    GetExceptionCode());
}

// Отображаем записанную строку
if(lpszBuff != NULL)
  printf("String:>%s<\n", lpszBuff);

// Изменяем размер блока памяти
__try
{
  HeapReAlloc(hHeap, HEAP_GENERATE_EXCEPTIONS |
    HEAP_REALLOC_IN_PLACE_ONLY, lpszBuff, 150);
}
__except (EXCEPTION_EXECUTE_HANDLER)
{
  fprintf(stdout,"HeapReAlloc: Exception %lX \n",
```

```
    GetExceptionCode());
}

// Освобождаем блок памяти
if(lpszBuff != NULL)
  HeapFree(hHeap, HEAP_NO_SERIALIZE, lpszBuff);

// Удаляем пул памяти
if(!HeapDestroy(hHeap))
  fprintf(stdout,"Ошибка %ld при удалении пула\n",
    GetLastError());

// ================================================
// Работа со стандартным пулом
// Исключения не обрабатываем
// ================================================

// Получаем блок памяти из стандартного пула
lpszBuff = (char*)HeapAlloc(GetProcessHeap(),
  HEAP_ZERO_MEMORY, 0x1000);

// Если памяти нет, выводим сообщение об ошибке
// и завершаем работу программы
if(lpszBuff == NULL)
{
  fprintf(stdout,"3. HeapAlloc: Error %ld\n",
    GetLastError());

  getch();
  return 0;
}

// Выполняем копирование строки
strcpy(lpszBuff, "Test string");

// Отображаем скопированную строку
  printf("String:>%s<\n", lpszBuff);

  // Освобождаем блок памяти
  if(lpszBuff != NULL)
    HeapFree(GetProcessHeap(), HEAP_NO_SERIALIZE, lpszBuff);

  // ================================================
  // Работа со стандартными функциями
  // Исключения не обрабатываем
  // ================================================

  lpszBuff = malloc(1000);

  if(lpszBuff != NULL)
  {
    strcpy(lpszBuff, "Test string");
    printf("String:>%s<\n", lpszBuff);
    free(lpszBuff);
  }

  printf("Press any key...");
  getch();
  return 0;
}
```

В отличие от обычного приложения Microsoft Windows NT, исходный текст консольного приложения должен содержать функцию main (аналогично программе MS-DOS). В теле этой функции мы определили переменные hHeap и lpszBuff. Первая из них используется для хранения идентификатора динамического пула памяти, вторая - для хранения указателя на полученный блок памяти.

Работа с динамическим пулом памяти

Вначале наше приложение создает динамический пул памяти, вызывая для этого функцию HeapCreate. Для пула резервируется 2 Кбайта памяти, причем для непосредственного использования выделяется только 1 Кбайт.

При возникновении ошибки ее код определяется с помощью функции GetLastError и отображается в консольном окне хорошо знакомой вам из MS-DOS функцией fprintf. Затем работа приложения завершается.

Заметим, что многие (но не все) функции программного интерфейса Microsoft Windows NT в случае возникновения ошибки перед возвращением управления устанавливают код ошибки, вызывая для этого функцию SetLastError. При необходимости приложение может извлечь этот код сразу после вызова функции, как это показано в нашем приложении.

Далее приложение пытается получить блок памяти размером 0x1500 байт, вызывая функцию HeapAlloc:

```
__try
{
  lpszBuff = (char*)HeapAlloc(hHeap,
    HEAP_GENERATE_EXCEPTIONS, 0x1500);
}
```

Так как во втором параметре мы передали этой функции значение HEAP_GENERATE_EXCEPTIONS, в случае ошибки возникнет исключение. Поэтому вызов функции HeapAlloc выполняется с обработкой исключений. Соответствующий обработчик получает код исключения при помощи функции GetExceptionCode и отображает его в консольном окне.

В нашем приложении мы пытаемся получить больше памяти, чем доступно, поэтому исключение действительно произойдет.

На следующем шаге, невзирая на исключение, наше приложение пытается записать в блок памяти, указатель на который находится в переменной lpszBuff, текстовую строку:

```
__try
{
  strcpy(lpszBuff, "Строка для проверки");
}
```

Так как при получении блока памяти произошло исключение, в указателе lpszBuff находится неправильный адрес. Это, в свою очередь, приведет к возникновению исключения при попытке записи строки. Поэтому на рис. 1.19 в верхней части консольного окна находятся два сообщения об исключениях.

После обработки второго исключения наше приложение выполняет вторую попытку получения блока памяти, но уже меньшего размера. Эта попытка закончится удачно, после чего приложение выведет строку в консольное окно, пользуясь другой хорошо знакомой вам функцией printf.

Затем приложение пытается изменить размер полученного блока памяти, вызывая функцию HeapReAlloc:

```
__try
{
  HeapReAlloc(hHeap, HEAP_GENERATE_EXCEPTIONS |
    HEAP_REALLOC_IN_PLACE_ONLY, lpszBuff, 150);
}
```

Так как указан флаг HEAP_REALLOC_IN_PLACE_ONLY, при изменении размера блок не будет перемещен, поэтому мы игнорируем значение, возвращаемое функцией HeapReAlloc.

А что произойдет, если размер блока увеличится настолько, что он не поместится в адресном пространстве, отведенном для него ранее?

Мы указали флаг HEAP_GENERATE_EXCEPTIONS, поэтому в этом случае произойдет исключение, которое наше приложение обработает.

После изменения размера блока памяти приложение освобождает его функцией HeapFree, а затем удаляет динамический пул памяти, так как мы больше не будем с ним работать.

Работа со стандартным пулом памяти

Второй способ выделения блока памяти основан на использовании стандартного пула. Для получения памяти из стандартного пула мы пользуемся функцией HeapAlloc, передавая ей в качестве первого параметра значение идентификатора стандартного пула памяти, полученное от функции GetProcessHeap:

```
lpszBuff = (char*)HeapAlloc(GetProcessHeap(),
  HEAP_ZERO_MEMORY, 0x1000);
```

Так как мы указали флаг HEAP_ZERO_MEMORY, полученный блок памяти будет расписан нулями. Флаг HEAP_GENERATE_EXCEPTIONS не указан, поэтому после вызова функции мы должны проверить значение, полученное от нее.

На следующем этапе приложение выполняет копирование строки в блок памяти и отображение ее в консольном окне:

```
strcpy(lpszBuff, "Test string");
printf("String:>%s<\n", lpszBuff);
```

Так как исключения не обрабатываются, при их возникновении работа приложения завершится аварийно.

После использования приложение освобождает блок памяти, полученный из стандартного пула, для чего вызывается функция HeapFree:

```
HeapFree(GetProcessHeap(), HEAP_NO_SERIALIZE, lpszBuff);
```

Последний фрагмент приложения демонстрирует использование функций malloc и free для работы со стандартным пулом памяти и в комментариях не нуждается.

2 МУЛЬТИЗАДАЧНОСТЬ

Способность человека выполнять несколько задач сразу ни у кого сомнений не вызывает. Несмотря на то что врачи считают это вредным, многие любят читать во время еды или смотреть телевизор, а то и делать все это одновременно, ухитряясь при этом еще и реагировать на реплики окружающих. Как это ни странно, в мир персональных компьютеров мультизадачность вторглась относительно недавно, и далеко не каждый владелец компьютера умеет использовать все ее преимущества.

В те времена, когда повсеместно наибольшей популярностью пользовалась однозадачная операционная система MS-DOS, пользователю была доступна так называемая *переключательная мультизадачность*, основанная главном образом на резидентных программах. Резидентные программы калькуляторов позволяли, например, не прерывая работу программы редактора текста или другой программы, выполнить арифметические вычисления. Для переключения от выполнения основной задачи к работе с резидентной программой было нужно нажать ту или иную комбинацию клавиш.

К моменту появления мультизадачных операционных систем OS/2 и Windows было создано великое множество самых разнообразных и часто несовместимых между собой резидентных программ для MS-DOS. Среди них были достаточно мощные системы, такие, например, как Borland SideKick.

Появление операционной системы Microsoft Windows версии 3.0, работавшей как оболочка для MS-DOS, стимулировало появление приложений для Microsoft Windows, работавших в режиме *невытесняющей мультизадачности*. При этом приложения, составленные определенным образом, время от времени передавали друг другу управление, в результате чего создавалась иллюзия одновременной работы нескольких приложений. Аналогичный принцип использовалася в сетевой операционной системе Novell NetWare и в компьютерах фирмы Apple.

Невытесняющая мультизадачность решила проблемы совместимости, которые были слабым местом резидентных программ. Теперь пользователь мог запустить сразу несколько приложений и переключаться между ними при необходимости. Многие пользователи так и делали, однако возможности мультизадачности при этом были фактически не задействованы, так как пользователи работали с приложениями по очереди в режиме переключательной мультизадачности. Несмотря на то что формально операционная система Microsoft Windows версии 3.1 позволяет запустить, например, форматирование дискеты и на этом фоне работать с другими приложениями, едва ли найдется много желающих поступать таким образом. Дело, очевидно, в том, что пока дискета не будет отформатирована, все остальные запущенные приложения будут работать очень медленно.

Еще один существенный недостаток невытесняющей мультизадачности проявляется при запуске недостаточно хорошо отлаженных приложений. Если по какой либо причине приложение не сможет периодически передавать управление другим запущенным приложениям, работа всей системы будет заблокирована и пользователю останется только нажать комбинацию из трех известных клавиш либо кнопку аппаратного сброса, расположенную на корпусе компьютера.

Однако не следует думать, что у Microsoft не хватило ума организовать *вытесняющую мультизадачность*, когда всем запущенным приложениям выделяются кванты времени с использованием системного таймера. Вытесняющая мультизадачность была использована в операционной системе OS/2 версий 1.0 - 1.3, которая в те времена разрабатывалась совместно Microsoft и IBM. Однако слабая архитектура процессора Intel 80286, недостаточная производительность выпускавшихся тогда компьютеров и малый объем оперативной памяти, установленной в компьютерах подавляющего числа пользователей (1 - 2 Мбайта) помешали широкому распространению OS/2. Эта операционная система с истинной вытесняющей мультизадачностью работала очень медленно и была вытеснена более легковесной оболочкой Microsoft Windows версии 3.1.

Сегодня ситуация изменилась. Современные операционные системы для персональных компьютеров, такие как Microsoft Windows 95, Microsoft Windows NT, IBM OS/2 Warp работают в режиме вытесняющей мультизадачности, когда все приложения гарантированно получают для себя кванты времени по прерыванию от таймера. При этом накладные расходы на мультизадачность компенсируются высокой производительностью компьютеров, поэтому пользователь не будет их чуствовать (конечно, если для уменьшения свопинга в компьютере установлено не менее 16 Мбайт оперативной памяти, что уже не редкость).

В операционной системе Microsoft Windows NT реализовано практически все, что было создано за время развития компьютеров в области мультизадачности. Как результат, у пользователя появилась возможность не просто переключаться с одной задачи на другую, а реально работать одновременно с несколькими активными приложениями. Программисты же получили в свои руки новый инструмент, с помощью которого они могут реализовать многозадачную обработку данных, даже не заостряя на этом внимание пользователя. Например, в процессе редкатирования документа текстовый процессор может заниматься нумерацией листов или подготовкой документа для печати на принтере.

Процессы и задачи в Microsoft Windows NT

В операционной системе Microsoft Windows NT существуют два понятия, имеющие отношение к мультизадачности. Это процессы и задачи.

Процесс (process) создается, когда программа загружается в память для выполнения. Вы создаете процессы запуская, например, консольные прогрммы или графические приложения при помощи Program Manager. Как мы уже говорили, процессу выделяется в монопольное владение 2 Гбайта изолированного адресного пространства, в которое другие процессы не имеют никакого доступа.

Сразу после запуска процесса создается одна *задача* (thread), или, как ее еще называют в отечественной литературе, поток. Задача - это просто фрагмент кода приложения, который может выполняться автономно и независимо от других задач в рамках одного процесса. Например, функция WinMain в приведенных нами ранее примерах приложений может рассматриваться как задача, которая запускается сразу после запуска процесса. При необходимости эта задача может запускать другие задачи, реализуя таким образом мультизадачность в рамках процесса. Все задачи имеют доступ к памяти, выделенной запустившему их процессу.

Из сказанного выше следует, что с одной стороны, в операционной системе Microsoft Windows NT могут работать одновременно несколько процессов, с другой - в рамках каждого процесса могут параллельно работать несколько задач. Пользователь может запустить процесс, загрузив ту или иную программу в память для выполнения, но он не может запустить задачу, так как эта операция выполняется только процессами.

Распределение времени между задачами

Важно заметить, что кванты времени для работы выделяются не процессам, а запущенным ими задачам (рис. 2.1). При этом если в системе установлен только один процессор, то задачи выполняются по очереди, создавая иллюзию параллельного выполнения.

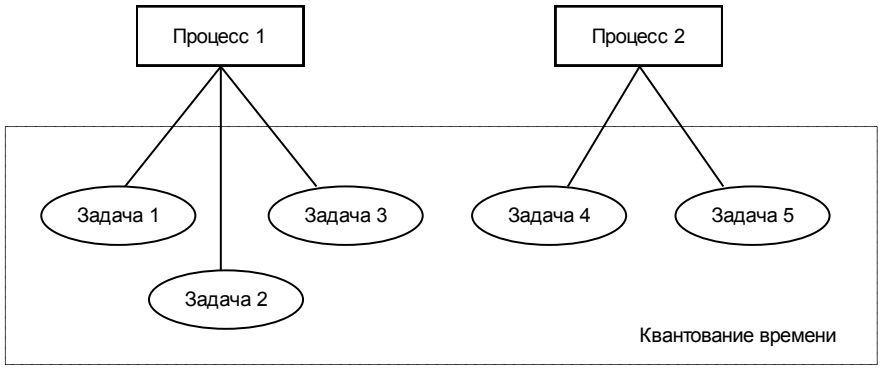

Рис. 2.1. Квантование времени выполняется для задач

Если же в компьютере установлено несколько процессоров, то операционная система выделяет процессоры для выполнения задач и тогда в действительности несколько задач могут работать параллельно. Обычно для совместимости приложения составляются таким образом, чтобы они “не знали” о количестве процессоров в системе.

Для оптимальной работы системы необходимо правильно установить закон, по которому кванты времени выделяются задачам. В операционной системе Microsoft Windows NT используется приоритетное планирование задач, когда и процессы, и задачи имеют свои уровни приоритета. При необходимости операционная система может автоматически в небольших пределах изменять приоритеты, повышая, например, приоритет задач, активно работающих с периферийными устройствами компьютера.

Операционная система устанавливает уровень приоритета задач в диапазоне от 1 до 31, причем значение 31 соответствует максимальному приоритету.

В процессе планирования кванты времени выделяются задачам с максимальным приоритетом. Менее приоритетные задачи получают управление только в том случае, если более приоритетные задачи переходят в состояние ожидания. Так как рано или поздно это обязательно происходит, то даже задачи с приоритетом, равным 1, имеют большие шансы получить кванты времени.

С другой стороны, если во время работы менее приоритеных задач запускается задача с более высоким приоритетом, все низкоприоритетные задачи приостанавливаются, а кванты времени выделяются более приоритетной задаче.

На первый взгляд может показаться странным, что приложения не могут устанавливать конкретное значение приоритета задач из указанного

интервала значений. Вместо этого используется двухступенчатая система установки приоритетов для процессов и задач.

Классы приоритета процессов

При запуске процесса с помощью функции CreateProcess (которая будет рассмотрена позже), ему можно назначить один из четырех классов приоритета:

Класс приоритета	*Уровень приоритета*
REALTIME_PRIORITY_CLASS	24 - процессы реального времени
HIGH_PRIORITY_CLASS	13 - высокоприоритетные процессы
NORMAL_PRIORITY_CLASS	9 или 7 - обычные процессы
IDLE_PRIORITY_CLASS	4 - низкоприоритетные процессы

Когда приоритет процесса не указывается, то по умолчанию он получает приоритет класса NORMAL_PRIORITY_CLASS. Если это приложение работает в фоновом режиме, операционная система снижает его уровень приоритета до 7, если же окно приложения выдвигается на передний план - увеличивает до 9. Таким образом уровень приоритета приложения, с которым в данный момент работает пользователь, автоматически увеличивается.

Класс IDLE_PRIORITY_CLASS используется для приложений, которые не должны тормозить работу других приложений. Это могут быть приложения, предназначенные для выполнения какой-либо фоновой работы, отображения постоянно меняющейся информации или приложения, выполняющий большой объем вычислений, сильно загружающих процессор. Например, если ваше приложение выполняет многочасовой расчет, имеет смысл назначить ему класс приоритета IDLE_PRIORITY_CLASS. При этом во время расчета пользователь сможет выполнять и другую работу, например, редактирование текста.

В тех случаях, когда приложение должно немедленно отзываться на действия пользователя, ему, возможно, следует назначить класс приоритета HIGH_PRIORITY_CLASS. Этот класс приоритета имеет, например, приложение Task Manager, с помощью которого пользователь может переключаться между запущенными приложениями а также завершать приложения. Не следует увлекаться созданием высокоприоритетных приложений, так как если в системе их будет запущенно много, то работа схемы, обеспечивающей оптимальный баланс производительности процессов, будет нарушена. В результате вы не получите желаемого эффекта.

Что же касается класса приоритета REALTIME_PRIORITY_CLASS, то он должен использоваться только системными процессами или драйверами. В противном случае будет блокирована работа клавиатуры и мыши, так как они обслуживаются с меньшим уровнем приоритета, чем приоритет классса REALTIME_PRIORITY_CLASS.

Относительный приоритет задач

Как мы уже говорили, в рамках одного процесса может быть запущено несколько задач. Точно также как невозможно задать явным образом уровень приоритета процессов (лежащий в диапазоне значений от 1 до 31), невозможно задать и уровень приоритета задач, запущенных процессом. Вместо этого процесс при необходимости устанавливает функцией SetThreadPriority относительный приоритет задач, который может быть несколько ниже или выше приоритета процесса.

Указанной выше функции можно передать одно из следующих значений, определяющих новый приоритет задачи относительно приоритета процесса:

Значение	*Относительное изменение уровня приоритета*
THREAD_PRIORITY_TIME_CRITICAL	Устанавливается абсолютный уровень приоритета 15 или 31
THREAD_PRIORITY_HIGHEST	+2
THREAD_PRIORITY_ABOVE_NORMAL	+1
THREAD_PRIORITY_NORMAL	0
THREAD_PRIORITY_BELOW_NORMAL	-1
THREAD_PRIORITY_LOWEST	-2
THREAD_PRIORITY_IDLE	Устанавливается абсолютный уровень приоритета 1 или 16

Если процесс имеет класс приоритета, равный значению REALTIME_PRIORITY_CLASS, использование относительного приоритета THREAD_PRIORITY_TIME_CRITICAL приведет к тому, что уровень приоритета задачи будет равен 31. Если же это значение относительного приоритета укажет процесс более низкого класса приоритета, уровень приоритета задачи установится равным 15.

Для процесса с классом приоритета REALTIME_PRIORITY_CLASS использование относительного приоритета THREAD_PRIORITY_IDLE приведет к тому, что будет установлен уровень приоритета задачи, равный 16. Если же значение THREAD_PRIORITY_IDLE будет использовано менее приоритетным процессом, уровень приоритета задачи будет равен 1.

Операционная система может автоматически изменять приоритет задач, повышая его, когда задача начинает взаимодействовать с пользователем, а затем постепенно уменьшая. Приоритет задач, находящихся в состоянии ожидания, также уменьшается.

Процесс может запустить задачу, а потом увеличить ее приоритет. В этом случае главная задача процесса, запустившая более приоритетную задачу, будет временно приостановлена. Если же процесс запустит задачу и уменьшит ее приоритет таким образом, что он станет меньше приоритета главной задачи процесса, будет приостановлена запущенная задача.

В составе Resource Kit for Windows NT и в составе SDK поставляется приложение Process Viewer (рис. 2.2), пользуясь которым можно просмотреть и в некоторых случаях изменить приоритеты процессов и задач, а также получить другую информацию о запущенных задачах (использованное процессорное время с момента запуска, процент работы системного и пользовательского кода, использование виртуальной памяти и так далее).

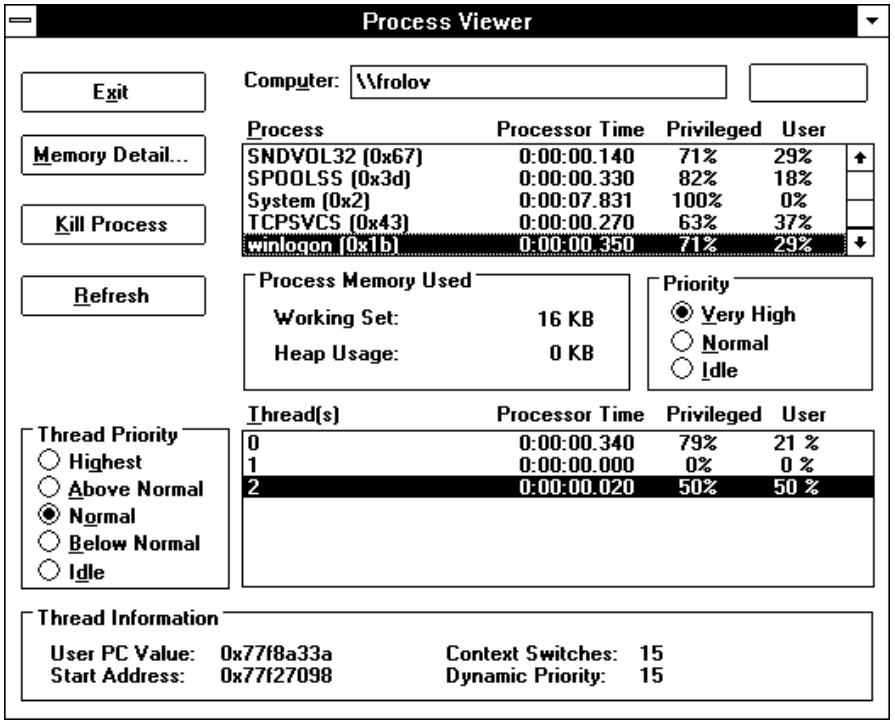

Рис. 2.2. Приложение Process Viewer

С помощью этого приложения вы также можете завершить работу процесса, если возникнет такая необходимость.

Проблемы синхронизации задач и процессов

Создавая мультизадачное приложение, необходимо тщательно планировать взаимодействие задач, избегая конфликтных ситуаций, когда различные задачи пытаются одновременно использовать один и тот же ресурс. Например, одна задача может выполнять запись в ячейку памяти нового значения, а вторая - чтение из нее. Результат чтения при этом будет зависеть от момента, когда произойдет чтение - до записи нового значения или после. Возможно возникновение взаимных блокировок задач, когда задачи будут бесконечно ждать друг друга.

Операционная система организует для процессов и задач последовательный доступ к таким ресурсам, как принтер или последовательный асинхронный порт, однако если несколько задач вашего приложения рисуют что-либо в одном окне, вы должны сами позаботиться об организации последовательного доступа к таким ресурсам, как контекст отображения, кисти, шрифты и так далее.

Специально для организации взаимодействия задач в операционной системе Microsoft Windows NT предусмотрены так называемые *объекты синхронизации*. Это средства организации последовательного использования ресуросов (mutex), семафоры (semaphore) и события (event). Мы рассмотрим перечисленные средства позже в этой главе.

Поиск ошибок, возникающих в результате неправильно организованной синхронизации задач, может отнять много времени, поэтому прежде чем принять решение об использовании мультизадачности в вашем приложении, следует хорошо подумать, нужно ли это и оценить полученные в результате преимущества (если они вообще будут).

Передача данных между процессами и задачами

Так как процессы выполняются в изолированных адресных пространствах, возникают некоторые трудности при необходимости организовать обмен данными между процессами. Вы не можете просто предать из одного процесса другому указатели на глобальные области памяти или идентфикаторы каких-либо ресурсов (например, идентификатор кисти, растрового изображения и так далее), так как в контексте другого процесса эти указатели и идентификаторы не имеют смысла.

Что же делать?

Для передачи данных между процессами вы должны использовать такие средства, как файлы, отображаемые в память, средства динамической передачи данных DDE, трубы и другие специальные средства, которые мы рассмотрим позже. При этом дополнительно необходимо использовать средства синхронизации процессов.

Что же касается задач, запущенных в рамках одного процесса, то здесь ситуация несколько легче. Так как все задачи работают в едином адресном

пространстве, то они могут обмениваться данными, например, через глобальные переменные. Разумеется, и в этом случае без средств синхронизации не обойтись.

Запуск задач

Существует три способа запустить задачу в приложениях, составленных на языке программирования C.

Во-первых, можно использовать фукнцию CreateThread, которая входит в программный интерфейс операционной системы Microsoft Windows NT. Этот способ предоставляет наибольшие возможности по управлению запущенными задачами, позволяя, в частности, присваивать запущенным задачам атрибуты защиты и создавать задачи в приостановленном состоянии.

Во-вторых, в вашем распоряжении имеется функция из библиотеки системы разработки Microsoft Visual C++ с названием _beginthread. Задачи, созданные с использованием этой функции, могут обращаться ко всем стандартным функциям библиотеки и к переменной errno.

И, наконец, в-третьих, можно запустить задачу при помощи функции _beginthreadex, которая определена в библиотеке Microsoft Visual C++, но имеет возможности, аналогичные функции CreateThread.

Мы рассмотрим все эти способы.

Функция CreateThread

Прототип функции CreateThread, с помощью которой процессы могут создавать задачи, представлен ниже:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes,// атрибуты защиты
  DWORD dwStackSize,      // начальный размер стека в байтах
  LPTHREAD_START_ROUTINE lpStartAddress,// адрес функции
                          // задачи
  LPVOID  lpParameter,        // параметры для задачи
  DWORD   dwCreationFlags,    // параметры создания задачи
  LPDWORD lpThreadId);        // адрес переменной для
                          // идентификатора задачи
```

Через параметр lpThreadAttributes передается адрес структуры SECURITY_ATTRIBUTES, определяющей атрибуты защиты для создаваемой задачи, или значение NULL. В последнем случае для задачи будут использованы атрибуты защиты, принятые по умолчанию. Это означает, что идентификатор созданной задачи можно использовать в любых функциях, выполняющих любые операции над задачами. Указывая атрибуты защиты, вы можете запретить использование тех или иных функций.

Приведем структуру SECURITY_ATTRIBUTES:

```
typedef struct _SECURITY_ATTRIBUTES
{
  DWORD  nLength;             // размер структуры в байтах
  LPVOID lpSecurityDescriptor; // указатель на дескриптор
                          //   защиты
  BOOL   bInheritHandle;      // флаг наследования
                          //   идентификатора
} SECURITY_ATTRIBUTES;
```

При подготовке структуры в поле nLength следует записать размер структуры SECURITY_ATTRIBUTES.

Поле указателя на дескриптор защиты lpSecurityDescriptor не заполняется приложением непосредственно. Вместо этого для установки дескритора защиты используется набор функций, которым в качестве одного из параметров передается указатель на структуру SECURITY_ATTRIBUTES. Эти функции подробно описаны в SDK. В нашей книге для экономии места мы не будем на них останавливаться. Система защиты Microsoft Windows NT достаточно мощная и потому заслуживает отдельного рассмотрения.

Параметр dwStackSize функции CreateThread позволяет указать начальный размер стека для запускаемой задачи. Если указать для этого параметра нулевое значение, размер стека запущенной задачи будет равен размеру стека главной задачи процесса. При необходимости размер стека автоматически увеличивается.

Таким образом, первые два параметра функции CreateThread не вызывают затруднений. Они могут быть в большинстве случаев указаны как NULL и 0, соответственно.

Параметр lpStartAddress задает адрес функции, которая будет выполняться как отдельная задача. Здесь вы можете просто указать имя этой функции. Функция задачи имеет один 32-разрядный параметр и возвращает 32-разрядное значение. Указанный параметр передается функции CreateThread через параметр lpParameter.

Если значение параметра dwCreationFlags равно нулю, после вызова функции CreateThread задача немедленно начнет свое выполнение. Если же в этом параметре указать значение CREATE_SUSPENDED, задача будет загружена, но приостановлена. Возобновить выполнение приостановленной задачи можно будет позже с помощью функции ResumeThread.

И, наконец, через параметр lpThreadId вы должны передать адрес переменной типа DWORD, в которую бедет записан системный номер созданной задачи (thread identifier).

В случае успеха функция CreateThread возвращает идентификатор задачи (thread handle), пользуясь которым можно выполнять над задачей те или иные операции. Не путайте этот идентификатор с системным номером задачи. При ошибке функция CreateThread возвращает значение NULL.

Ниже мы приведи пример использования функции CreateThread:

```
hThread = CreateThread(NULL, 0,
 (LPTHREAD_START_ROUTINE)ThreadRoutine,
 (LPVOID)hwndChild, 0,(LPDWORD)&dwIDThread);
```

Здесь мы не используем атрибуты защиты и устанавливаем размер стека запускаемой задачи, равным размеру стека главной задачи процесса.

В качестве функции задачи мы использовали функцию с именем ThreadRoutine, имеющую следующий вид:

```
DWORD ThreadRoutine(HWND hwnd)
{
 . . .
 // Оператор return завершает выполнение задачи
 return 0;
}
```

Эта функция имеет один параметр, который в нашем случае будет принимать значение hwndChild.

Наша функция задачи завершает свое выполнение оператором return, хотя, как вы увидите дальше, есть и другие способы. Для проверки значения, возвращенного функцией задачи, процесс может воспользоваться функцией GetExitCodeThread, которая будет описана позже.

Так как при запуске задачи мы указали значение параметра dwCreationFlags, равное нулю, сразу после запуска задача начнет свою работу. Системный номер созданной задачи будет записан в переменную dwIDThread.

Функция _beginthread

Если вы создаете мультизадачное приложение и собираетесь при этом использовать функции стандартной библиотеки С, перед вами могут встать неожиданные проблемы. Дело в том, что первоначальные версии библиотек времени выполнения систем разработки не были рассчитаны на то, что в рамках одной программы могут существовать параллельно работающие задачи. В результате, например, могут возникнуть проблемы при использовании стандартных глобальных переменных, таких как errno, и функций, работающих с глобальными переменными.

Напомним, что после выполнения стандратных функции билиотеки С в переменную errno записывается код ошибки. Однако в мультизадачных приложениях содержимое этой переменной может быть установлено из любой задачи, поэтому для каждой задачи необходимо предусмотреть свою собственную глобальную переменную errno.

Для решения этой проблемы в системе разработки Microsoft Visual C++ предусмотрены отдельные библиотеки для создания однозадачных и мультизадачных приложений.

Для выбора правильной библиотеки для проекта Microsoft Visual C++ версии 4.0 или 4.1 выберите из меню Build строку Settings. На экране появится блокнот настройки параметров проекта Project Settings. Открыв страницу C/C++ этого блокнота, выберите в списке Category строку Code Generation. После этого на странице появится список Use run-time library, в котором имеются следующие строки:

- Single-Treaded;
- Multithreaded;
- Multithreaded DLL;
- Debug Single-Treaded;
- Debug Multithreaded;
- Debug Multithreaded DLL

Если ваше приложение является однозадачным, имеет смысл выбрать библиотеку Single-Treaded или Debug Single-Treaded (при отладке). Функции из этой библиотеки будут работать быстрее, чем из библиотеки Multithreaded (которую тоже можно использовать в однозадачных приложениях), так как не будет накладных расходов на мультизадачность.

В том случае, если вы создаете мультизадачное приложение, необходимо использовать библиотеки Multithreaded и Multithreaded DLL (для создания мультизадачных библиотек DLL) либо отладочные версии этих библиотек.

Заметим, что для однозадачных приложений используется библиотека с именем libc.lib, а для мультизадачных - с именем libcmt.lib.

После того как вы указали мультизадачную библиотеку, вы можете использовать функции _beginthread и _beginthreadex для запуска задач. Приведем прототип функции _beginthread, описанный в файле process.h:

```
unsigned long _beginthread(
  void(*StartAddress)(void*), // адрес функции задачи
  unsigned uStackSize,    // начальный размер стека в байтах
  void     *ArgList);     // параметры для задачи
```

Заметим, что функция задачи, запускаемой при помощи функции _beginthread, не возвращает никакого значения. Ее адрес передается функции _beginthread через параметр StartAddress. Через параметр ArgList вы можете передать функции задачи один параметр.

Начальный размер стека, выделяемого задаче, указывается через параметр uStackSize. Так же как и в случае с функцией CreateThread, для размера стека можно указать нулевое значение. При этом для задачи создается стек такого же размера, что и для главной задачи процесса.

В случае успеха функция _beginthread возвращает идентификатор запущенной задачи. Если же произошла ошибка, возвращается значение -1.

Приведем пример использования функции _beginthread:

```
_beginthread(ThreadRoutine, 0, (void*)(Param));
```

Здесь запускается задача, функция которой имеет имя ThreadRoutine. Ей передается в качестве параметра значение Param.

Функция ThreadRoutine должна выглядеть следующим образом:

```
void ThreadRoutine(void *Param)
{
 . . .
 _endthread();
}
```

Заметим, что для завершения задачи здесь используется функция _endthread, не имеющая параметров. С помощью этой функции вы можете завершить задачу в любом месте функции задачи. Однако в приведенном выше фрагменте функцию _endthread можно было бы и не использовать, так как операция возврата из функции задачи также приведет к неявному вызову функции _endthread и, как следствие, к завершению задачи.

Функция _beginthreadex

В том случае, если вам нужны возможности функции CreateThread (например, необходимо создать задачу в приостановленном состоянии) и вместе с тем необходимо использовать функции библиотеки транслятора, имеет смысл обратить внимание на функцию_beginthreadex. Прототип этой функции мы привели ниже:

```
unsigned long _beginthreadex(
  void    *Security,      // указатель на дескриптор защиты
  unsigned StackSize,             // начальный размер стека
  unsigned (*StartAddress)(void*), // адрес функции задачи
  void    *ArgList,           // параметры для задачи
  unsigned Initflag,           // параметры создания задачи
  unsigned *ThrdAddr);         // адрес созданной задачи
```

Для запуска задачи в приостановленном состоянии через параметр Initflag необходимо передать значение CREATE_SUSPENDED.

Функция задачи, которая запускается с помощью функции _beginthreadex, имеет один параметр и возвращает 32-разрядное значение, аналогично функции задачи, запускаемой функцией CreateThread. Для завершения своего выполнения функция задачи должна использовать либо оператор возврата, либо функцию _endthreadex, не имеющую параметров.

В случае успеха функция _beginthreadex возвращает идентификатор запущенной задачи. Если же произошла ошибка, возвращается значение 0 (а не -1, как это было для функции _beginthread).

Управление запущенными задачами

После того как задача запущена, запустившая задача знает идентификатор дочерней задачи. Этот идентификатор возвращается функциями CreateThread, _beginthread и _beginthreadex. Пользуясь этим идентификатором, запустившая задача может управлять состоянием дочерней задачи, изменяя ее приоритет, приостанавливая, возобновляя или завершая ее работу.

Изменение приоритета задачи

В разделе “Относительный приоритет задач” нашей книги мы рассказали вам о том, как в операционной системе Microsoft Windows NT устанавливаются приоритеты задач. Родительская задача может изменить относительный приоритет запущенной ей дочерней задачи с помощью функции SetThreadPriority:

```
BOOL SetThreadPriority(
  HANDLE hThread,    // идентификатор задачи
  int    nPriority); // новый уровень приоритета задачи
```

Через параметр hThread этой функции передается идентификатор задачи, для которой необходимо изменить относительный приоритет.

Новое значение относительного приоритета передается через параметр nPriority и может принимать одно из следующих значений:

- THREAD_PRIORITY_TIME_CRITICAL;
- THREAD_PRIORITY_HIGHEST;
- THREAD_PRIORITY_ABOVE_NORMAL;
- THREAD_PRIORITY_NORMAL;
- THREAD_PRIORITY_BELOW_NORMAL;
- THREAD_PRIORITY_LOWEST;
- THREAD_PRIORITY_IDLE

Абсолютный уровень приоритета, который получит задача, зависит от класса приоритета процесса. Забегая вперед, скажем, что класс приоритета процесса можно изменить при помощи функции SetPriorityClass.

Определение приоритета задачи

Зная идентификатор задачи, нетрудно определить ее относительный приоритет. Для этого следует воспользоваться функцией GetThreadPriority:

```
int GetThreadPriority(HANDLE hThread);
```

Эта функция возвращает одно из значений, перечисленных выше в разделе “Изменение приоритета задачи” или значение THREAD_PRIORITY_ERROR_RETURN при возникновении ошибки.

Приостановка и возобновление выполнения задачи

В некоторых случаях имеет смысл приостановить выполнение задачи. Например, если пользователь работает с многооконным приложением, и для каждого окна запускается отдельная задача, для повышения производительности можно приостановить выполнение задач в неактивных окнах.

Приостановка выполнения задачи выполняется с помощью функции SuspendThread:

```
DWORD SuspendThread(HANDLE hThread);
```

Через единственный параметр этой функции нужно передать идентификатор приостанавливаемой задачи.

Для каждой задачи операционная система хранит счетчик приостановок, который увеличивается при каждом вызове фукнции SuspendThread. Если значение этого счетчика больше нуля, задача приостанавливается.

Для уменьшения значения счетчика приостановок и, соответственно, для возобновления выполнения задачи вы должны использовать фукнцию ResumeThread:

```
DWORD ResumeThread(HANDLE hThread);
```

В случае ошибки функции SuspendThread и ResumeThread возвращают значение 0xFFFFFFFF.

Временная приостановка работы задачи

С помощью функции Sleep задача может приостановить свою работу на заданный период времени:

```
VOID Sleep(DWORD cMilliseconds); // время в миллисекундах
```

Задача, выполняющая ожидание с помощью этой функции, не снижает производительность системы, так как ей не распределяются кванты времени. Через единственный параметр вы можете задать функции время ожидания в миллисекундах.

Если для времени ожидания указать нулевой значение, то при наличии в системе запущенных задач с таким же приоритетом, как и у задачи, вызвавшей эту функцию, ожидающая задача отдает оставшееся время от своего кванта другой задаче. В том случае, когда в системе нет других задач с таким же значением приоритета, функция Sleep немедленно возвращает управление вызвавшей ее задаче, которая сама будет использовать остаток своего кванта времени.

Есть еще одна возможность: можно организовать бесконечную задержку, передав функции Sleep значение INFINITE.

Завершение задачи

Задача может завершиться как по собственной инициативе, так и по инициативе другой задачи. В первом случае задача либо выполняет оператор возврата из функции задачи, либо пользуется специальными функциями.

Для того чтобы завершить свое выполнение, задача, запущенная с помощью фукнции CreateThread, может вызвать функцию ExitThread, передав ей код завершения:

```
VOID ExitThread(DWORD dwExitCode);
```

Для принудительного завершения дочерней задачи родительская задача может использовать функцию TerminateThread, передав ей идентификатор завершаемой задачи и код завершения:

```
BOOL TerminateThread(
  HANDLE hThread,     // идентификатор завершаемой задачи
  DWORD  dwExitCode); // код завершения
```

Как родительская задача может получить код завершения дочерней задачи?

Для этого она должна вызвать фукнцию GetExitCodeThread:

```
BOOL GetExitCodeThread(
  HANDLE  hThread,     // идентификатор завершаемой задачи
  LPDWORD lpExitCode); // адрес для приема кода завершения
```

Если задача, для которой вызвана фукнция GetExitCodeThread, все еще работает, вместо кода завершения возвращается значение STILL_ACTIVE.

Как мы уже говорили, если задача была запущена с помощью функций _beginthread или _beginthreadex, она может завершать свое выполнение только с помощью функций, соответственно, _endthread и _endthreadex. Функцию ExitThread в этом случае использовать нельзя, так как при этом не будут освобождены ресурсы, заказанные для работы с мультизадачным вариантом библиотеки времени выполнения.

Освобождение идентификатора задачи

После завершения процесса идентификаторы всех созданных им задач освобождаются. Однако лучше, если приложение будет самостоятельно

освобождать ненужные ей идентификаторы задач, так как они являются ограниченным системным ресурсом.

Для освобождения идентификатора задачи вы должны передать его функции CloseHandle, имеющей единственный параметр.

Еще одно замечание относительно освобождения идентификаторов задач и процессов приведено в следующей главе, посвященной процессам.

Приложение MultiSDI

Наше первое мультизадачное приложение называется MultiSDI. В его главном окне (рис. 2.3) рисуют четыре задачи, одна из которых является главной и соответствует функции WinMain.

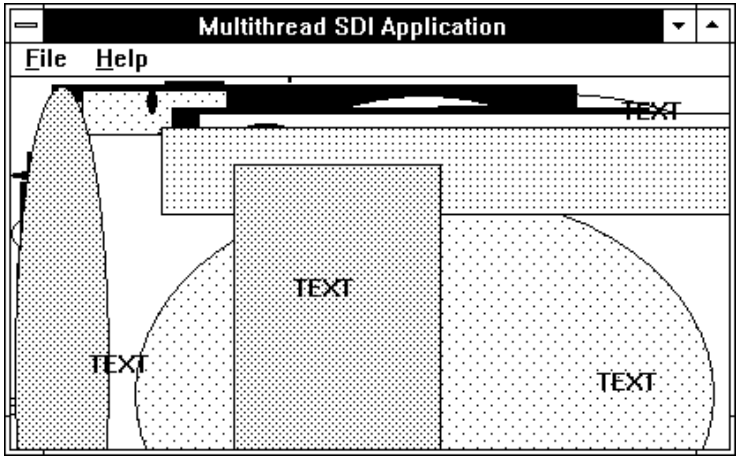

Рис. 2.3. Главное окно приложения MultiSDI

Немного позже мы приведем исходные тексты MDI-приложения MultiMDI, в котором для каждого дочернего MDI-окна создается отдельная задача.

Главная задача приложения MultiSDI рисует в главном окне приложения во время обработки сообщения WM_PAINT. Кроме того, она создает еще три задачи, которые рисуют в окне приложения, соответственно, эллипсы, прямоугольники и текстовую строку TEXT. Цвет и размеры фигур, а также цвет текстовой строки и цвет фона, на котором отображается эта строка, выбираются случайным образом, поэтому содержимое главного окна изменяется хаотически. Это выглядит достоточно забавно.

Так как все задачи выполняют рисование в одном окне, нам пришлось использовать простейшее средство синхронизации задач - критическую секцию. Проблемы синхронизации задач, работающих параллельно, мы рассмотрим в отдельной главе нашей книги.

Исходные тексты приложения

Главный файл исходных текстов приложения MultiSDI представлен в листинге 2.1. Заметим, что для сборки проекта мультизадачного приложения необходимо использовать мультизадачный вариант библиотеки времени выполнения. Об этом мы уже говорили в разделе “Функция _beginthread”.

Листинг 2.1. Файл multisdi/multisdi.c

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <process.h>
#include <stdio.h>
#include "resource.h"
#include "afxres.h"
#include "multisdi.h"

HINSTANCE hInst;
char szAppName[]  = "MultiSDI";
char szAppTitle[] = "Multithread SDI Application";

// Критическая секция для рисования в окне
CRITICAL_SECTION csWindowPaint;

// Признак завершения всех задач
BOOL fTerminate = FALSE;

// Массив идентификаторов запущенных задач
HANDLE hThreads[3];

// ----------------------------------------------------
// Функция WinMain
// ----------------------------------------------------
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
```

```
      LPSTR lpCmdLine, int nCmdShow)
{
 WNDCLASSEX wc;
 HWND hWnd;
 MSG msg;

 // Сохраняем идентификатор приложения
 hInst = hInstance;

 // Преверяем, не было ли это приложение запущено ранее
 hWnd = FindWindow(szAppName, NULL);
 if(hWnd)
 {
  // Если было, выдвигаем окно приложения на
  // передний план
  if(IsIconic(hWnd))
    ShowWindow(hWnd, SW_RESTORE);
  SetForegroundWindow(hWnd);
  return FALSE;
 }

 // Регистрируем класс окна
 memset(&wc, 0, sizeof(wc));
 wc.cbSize = sizeof(WNDCLASSEX);
 wc.hIconSm = LoadImage(hInst,
  MAKEINTRESOURCE(IDI_APPICONSM),
  IMAGE_ICON, 16, 16, 0);
 wc.style = 0;
 wc.lpfnWndProc = (WNDPROC)WndProc;
 wc.cbClsExtra  = 0;
 wc.cbWndExtra  = 0;
 wc.hInstance = hInst;
 wc.hIcon = LoadImage(hInst,
  MAKEINTRESOURCE(IDI_APPICON),
  IMAGE_ICON, 32, 32, 0);
 wc.hCursor = LoadCursor(NULL, IDC_ARROW);
 wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
 wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
 wc.lpszClassName = szAppName;
 if(!RegisterClassEx(&wc))
  if(!RegisterClass((LPWNDCLASS)&wc.style))
          return FALSE;

 // Создаем главное окно приложения
 hWnd = CreateWindow(szAppName, szAppTitle,
   WS_OVERLAPPEDWINDOW,
   CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
   NULL, NULL, hInst, NULL);
 if(!hWnd) return(FALSE);

 // Отображаем окно и запускаем цикл
 // обработки сообщений
 ShowWindow(hWnd, nCmdShow);
 UpdateWindow(hWnd);
 while(GetMessage(&msg, NULL, 0, 0))
 {
  TranslateMessage(&msg);
  DispatchMessage(&msg);
 }
 return msg.wParam;
}

// ----------------------------------------------------
// Функция WndProc
// ----------------------------------------------------
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
      LPARAM lParam)
{
 switch(msg)
 {
  HANDLE_MSG(hWnd, WM_CREATE,    WndProc_OnCreate);
```

```
    HANDLE_MSG(hWnd, WM_DESTROY,    WndProc_OnDestroy);
    HANDLE_MSG(hWnd, WM_PAINT,      WndProc_OnPaint);
    HANDLE_MSG(hWnd, WM_COMMAND,    WndProc_OnCommand);

    default:
      return(DefWindowProc(hWnd, msg, wParam, lParam));
  }
}

// ------------------------------------------------------
// Функция WndProc_OnCreate
// ------------------------------------------------------
BOOL WndProc_OnCreate(HWND hWnd,
                      LPCREATESTRUCT lpCreateStruct)
{
  // Инициализируем критическую секцию
  InitializeCriticalSection(&csWindowPaint);

  // Сбрасываем флаг завершения задач
  fTerminate = FALSE;

  // Запускаем три задачи, сохраняя их идентификаторы
  // в массиве
  hThreads[0] = (HANDLE)_beginthread(PaintEllipse,
    0, (void*)hWnd);
  hThreads[1] = (HANDLE)_beginthread(PaintRect,
    0, (void*)hWnd);
  hThreads[2] = (HANDLE)_beginthread(PaintText,
    0, (void*)hWnd);

  return TRUE;
}

// ------------------------------------------------------
// Функция WndProc_OnDestroy
// ------------------------------------------------------
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{
  // Устанавливаем флаг завершения задач
  fTerminate = TRUE;

  // Дожидаемся завершения всех трех задач
  WaitForMultipleObjects(3, hThreads, TRUE, INFINITE);

  // Перед удаляем критическую секцию
  DeleteCriticalSection(&csWindowPaint);

  // Останавливаем цикл обработки сообщений, расположенный
  // в главной задаче
  PostQuitMessage(0);
  return 0L;
}

// ------------------------------------------------------
// Функция WndProc_OnPaint
// ------------------------------------------------------
#pragma warning(disable: 4098)
void WndProc_OnPaint(HWND hWnd)
{
  HDC hdc;
  PAINTSTRUCT ps;
  RECT rc;

  // Входим в критическую секцию
  EnterCriticalSection(&csWindowPaint);

  // Перерисовываем внутреннюю область окна
  hdc = BeginPaint(hWnd, &ps);

  GetClientRect(hWnd, &rc);
  DrawText(hdc, "SDI Window", -1, &rc,
```

```
    DT_SINGLELINE | DT_CENTER | DT_VCENTER);

  EndPaint(hWnd, &ps);

  // Выходим из критической секции
  LeaveCriticalSection(&csWindowPaint);
  return 0;
}
// ----------------------------------------------------
// Функция WndProc_OnCommand
// ----------------------------------------------------
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id,
  HWND hwndCtl, UINT codeNotify)
{
  switch (id)
  {
    case ID_FILE_EXIT:
    {
      // Завершаем работу приложения
      PostQuitMessage(0);
      return 0L;
      break;
    }

    case ID_HELP_ABOUT:
    {
      MessageBox(hWnd,
        "Multithread SDI Application\n"
        "(C) Alexandr Frolov, 1996\n"
        "Email: frolov@glas.apc.org",
        szAppTitle, MB_OK | MB_ICONINFORMATION);
      return 0L;
      break;
    }

    default:
      break;
  }
  return FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify,
    DefWindowProc);
}

// ----------------------------------------------------
// Функция задачи PaintEllipse
// ----------------------------------------------------
void PaintEllipse(void *hwnd)
{
  HDC hDC;
  RECT rect;
  LONG xLeft, xRight, yTop, yBottom;
  short nRed, nGreen, nBlue;
  HBRUSH hBrush, hOldBrush;

  // Инициализация генератора случайных чисел
  srand((unsigned int)hwnd);

  // Задача работает в бесконечном цикле,
  // который будет прерван при завершении приложения
  while(!fTerminate)
  {
    // Входим в критическую секцию
    EnterCriticalSection(&csWindowPaint);

    // Отображаем эллипс, имеющий случайный цвет,
    // форму и расположение

    // Получаем контекст отображения
    hDC = GetDC(hwnd);

    // Получаем случайные цветовые компоненты
    nRed   = rand() % 255;
```

```
  nGreen = rand() % 255;
  nBlue  = rand() % 255;

  // Получаем случйные размеры эллипса
  GetWindowRect(hwnd, &rect);

  xLeft   = rand() % (rect.left   + 1);
  xRight  = rand() % (rect.right  + 1);
  yTop    = rand() % (rect.top    + 1);
  yBottom = rand() % (rect.bottom + 1);

  // Создаем кисть на основе случайных цветов
  hBrush = CreateSolidBrush(RGB(nRed, nGreen, nBlue));

  // Выбираем кисть в контекст отображения
  hOldBrush = SelectObject(hDC, hBrush);

  // Рисуем эллипс
  Ellipse(hDC, min(xLeft, xRight), min(yTop, yBottom),
          max(xLeft, xRight), max(yTop, yBottom));

  // Выбираем старую кисть в контекст отображения
  SelectObject(hDC, hOldBrush);

  // Удаляем созданную кисть
  DeleteObject(hBrush);

  // Освобождаем контекст отображения
  ReleaseDC(hwnd, hDC);

  // Выходим из критической секции
  LeaveCriticalSection(&csWindowPaint);

  // Выполняем задержку на 500 мс
  Sleep(500);
}
```

```
}

// -------------------------------------------------------
// Функция задачи PaintRect
// -------------------------------------------------------
void PaintRect(void *hwnd)
{
  HDC hDC;
  RECT rect;
  LONG xLeft, xRight, yTop, yBottom;
  short nRed, nGreen, nBlue;
  HBRUSH hBrush, hOldBrush;

  srand((unsigned int)hwnd + 1);

  while(!fTerminate)
  {
    EnterCriticalSection(&csWindowPaint);

    hDC = GetDC(hwnd);

    nRed   = rand() % 255;
    nGreen = rand() % 255;
    nBlue  = rand() % 255;

    GetWindowRect(hwnd, &rect);

    xLeft   = rand() % (rect.left   + 1);
    xRight  = rand() % (rect.right  + 1);
    yTop    = rand() % (rect.top    + 1);
    yBottom = rand() % (rect.bottom + 1);

    hBrush = CreateSolidBrush(RGB(nRed, nGreen, nBlue));
    hOldBrush = SelectObject(hDC, hBrush);

    Rectangle(hDC, min(xLeft, xRight), min(yTop, yBottom),
```

```
            max(xLeft, xRight), max(yTop, yBottom));

    SelectObject(hDC, hOldBrush);
    DeleteObject(hBrush);
    ReleaseDC(hwnd, hDC);

    LeaveCriticalSection(&csWindowPaint);
    Sleep(500);
  }
}

// -------------------------------------------------------
// Функция задачи PaintText
// -------------------------------------------------------
void PaintText(void *hwnd)
{
  HDC hDC;
  RECT rect;
  LONG xLeft, xRight, yTop, yBottom;
  short nRed, nGreen, nBlue;

  srand((unsigned int)hwnd + 2);

  while(!fTerminate)
  {
    EnterCriticalSection(&csWindowPaint);

    hDC = GetDC(hwnd);

    GetWindowRect(hwnd, &rect);

    xLeft   = rand() % (rect.left   + 1);
    xRight  = rand() % (rect.right  + 1);
    yTop    = rand() % (rect.top    + 1);
    yBottom = rand() % (rect.bottom + 1);

    // Устанавливаем случайный цвет текста
    nRed   = rand() % 255;
    nGreen = rand() % 255;
    nBlue  = rand() % 255;
    SetTextColor(hDC, RGB(nRed, nGreen, nBlue));

    // Устанавливаем случайный цвет фона
    nRed   = rand() % 255;
    nGreen = rand() % 255;
    nBlue  = rand() % 255;
    SetBkColor(hDC, RGB(nRed, nGreen, nBlue));

    TextOut(hDC, xRight - xLeft,
      yBottom - yTop,"TEXT", 4);

    ReleaseDC(hwnd, hDC);

    LeaveCriticalSection(&csWindowPaint);
    Sleep(500);
  }
}
```

Файл multisdi.h (листинг 2.2) содержит прототипы функций, определенных в приложении MultiSDI. Это функция главного окна приложения WndProc, функции обработки сообщений WM_CREATE, WM_DESTROY, WM_PAINT, WM_COMMAND (с именами, соответственно, WndProc_OnCreate, WndProc_OnDestroy, WndProc_OnPaint и WndProc_OnCommand), а также функции задач PaintEllipse, PaintRect и PaintText.

Листинг 2.2. Файл multisdi/multisdi.h

```
// -------------------------------------------------------
// Описание функций
// -------------------------------------------------------
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL WndProc_OnCreate(HWND hWnd, LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
```

```
void WndProc_OnPaint(HWND hWnd);
void WndProc_OnCommand(HWND hWnd, int id,
  HWND hwndCtl, UINT codeNotify);

void PaintEllipse(void *hwnd);
void PaintRect(void *hwnd);
void PaintText(void *hwnd);
```

Файл resource.h (листинг 2.3) создается автоматически и содержит определения констант для файла описания ресурсов приложения.

Листинг 2.3. Файл multisdi/resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by MultiSDI.RC
//
#define IDR_APPMENU                 102
#define IDI_APPICON                 103
#define IDI_APPICONSM               104
#define ID_FILE_EXIT                40001
#define ID_HELP_ABOUT               40003

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        121
#define _APS_NEXT_COMMAND_VALUE         40004
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

В файле описания ресурсов приложения MultiSDI (листинг 2.4), который тоже создается автоматически, определено главное меню приложения IDR_APPMENU и две пиктограммы (стандартного и уменьшенного размера), а также таблица строк, которая не используется.

Листинг 2.4. Файл multisdi/multisdi.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////
// Menu
//

IDR_APPMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",                       ID_FILE_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",                   ID_HELP_ABOUT
```

```
    END
END

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED

//////////////////////////////////////////////////////////////
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_APPICON             ICON    DISCARDABLE     "multisdi.ico"
IDI_APPICONSM           ICON    DISCARDABLE     "multissm.ico"

//////////////////////////////////////////////////////////////
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    ID_FILE_EXIT            "Quits the application"
END

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 3 resource.
//
//////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Определения и глобальные переменные

Помимо обычных include-файлов, характерных для наших приложений, в приложении MultiSDI используется файл process.h. Этот файл содержит прототип функции _beginthread, с помощью которой наше приложение создает задачи.

Для синхронизации задач, выполняющих рисование в окне приложения, мы определили глобальную структуру csWindowPaint типа CRITICAL_SECTION, которая будет использоваться в качестве критической секции:

```
CRITICAL_SECTION csWindowPaint;
```

В глобальной переменной fTerminate хранится флаг, установка которого в состояние TRUE вызывает завершение всех трех задач рисования.

И, наконец, в массиве hThreads хранятся идентификаторы запущенных задач. Этот массив будет использован, в частности, для того, чтобы перед удалением критической секции убедиться, что все задачи рисования завершили свою работу.

Описание функций

Приведем описание функций, определенных в нашем приложении.

Функция WinMain

Функция WinMain не имеет никаких особенностей. Сразу после того как она получит управление, функция проверяет, не было ли данное приложение уже запущено. Если было, окно запущенного приложения выдвигается на передний план.

Далее функция WinMain выполняет регистрацию класса окна приложения, создает главное окно, отображает его и запускает цикл обработки сообщений. Словом, все как всегда.

Функция WndProc

Функция главного окна приложения WndProc обрабатывает сообщения WM_CREATE, WM_DESTROY, WM_PAINT и WM_COMMNAD. Для этого с помощью макрокоманды HANDLE_MSG она вызывает, соответственно, функции WndProc_OnCreate, WndProc_OnDestroy, WndProc_OnPaint и WndProc_OnCommand.

Необработанные сообщения передаются фукнции DefWindowProc.

Функция WndProc_OnCreate

При создании главного окна приложения функция WndProc_OnCreate инициализирует структуру критической секции csWindowPaint, что необходимо для использования этого средства синхронизации задач:

```
InitializeCriticalSection(&csWindowPaint);
```

После выполнения такой инициализации в глобальную переменную fTerminate записывается значение FALSE, разрешающее работу задач. Содержимое этой глобальной переменной проверяется функциями задач, рисующих в главном окне приложения.

Последнее, что делает функция WndProc_OnCreate перед возвращением управления, это запуск задач PaintEllipse, PaintRect и PaintText. Для запуска в приложении MultiSDI мы использовали функцию _beginthread:

```
hThreads[0] = (HANDLE)_beginthread(PaintEllipse, 0,
  (void*)hWnd);
hThreads[1] = (HANDLE)_beginthread(PaintRect,    0,
  (void*)hWnd);
hThreads[2] = (HANDLE)_beginthread(PaintText,    0,
  (void*)hWnd);
```

Через первый параметр мы передаем фукнции _beginthread имя функции задачи.

Второй параметр, определяющий начальный размер стека задачи, равен нулю, поэтому этот размер равен начальному размеру стека главной задачи приложения.

Через третий парметр передается идентификатор главного окна приложения, в котором функции задач будут выполнять рисование.

В приложении с многооконным интерфейсом MultiMDI мы воспользуемся другой функцией предназначенной для запуска задач - функцией CreateThread.

Функция WndProc_OnDestroy

Когда наше приложение завершает свою работу, функция WndProc_OnDestroy, обрабатывающая сообщение WM_DESTROY, устанавливает содержимое глобальной переменной fTerminate в состояние TRUE. Функции задач периодически проверяют эту переменную, и как только ее содержимое станет равным TRUE, они завершают свою работу.

Так как в нашем приложении была создана критическая секция, ее следует удалить перед завершением приложения. Однако прежде чем это сделать, необходимо убедиться, что все фукнции задач завершили свою работу. Это можно сделать с помощью функции WaitForMultipleObjects, которая будет описана позже в разделе, посвященном синхронизации задач:

```
WaitForMultipleObjects(3, hThreads, TRUE, INFINITE);
```

В качестве первого параметра этой функции передается количество задач. Через второй параметр передается адрес массива с идентификаторами этих задач. Третий параметр, имеющий значение TRUE, указывает, что нужно дождаться завершения именно всех задач, а не одной из них. И, наконец, последний параметр определяет, что ожидание может длиться неограниченно долго.

Как только все три задачи завершат свою работу, функция WndProc_OnDestroy удалит критическую секцию с помощью функции DeleteCriticalSection:

```
DeleteCriticalSection(&csWindowPaint);
```

После этого вызывается функция PostQuitMessage, в результате чего завершается цикл обработки сообщений.

Функция WndProc_OnPaint

Функция WndProc_OnPaint выполняет очень простую вещь - она рисует в центре главного окна приложения текстовую строку SDI Window, используя для этого известную вам из программирования для Microsoft Windows версиии 3.1 функцию DrawText.

Обратите внимание, что перед получением контекста отображения мы входим в критическую секцию, вызывая функцию EnterCriticalSection. После завершения рисования выполняется выход из критической секции с помощью фукнции LeaveCriticalSection:

```
EnterCriticalSection(&csWindowPaint);
hdc = BeginPaint(hWnd, &ps);
```

```
GetClientRect(hWnd, &rc);
DrawText(hdc, "SDI Window", -1, &rc,
  DT_SINGLELINE | DT_CENTER | DT_VCENTER);
EndPaint(hWnd, &ps);
LeaveCriticalSection(&csWindowPaint);
```

Функции задач, исходный текст которых мы скоро рассмотрим, выполняют рисование в окне аналогичным способом, используя для синхронизации критическцю секцию csWindowPaint. Не вдаваясь в подробности (которыми мы займемся позже), скажем, что в результате в любой момент времени в главном окне приложения будет рисовать только одна задача.

Почему это важно?

Дело в том, что для повышения производительности графическая система Microsoft Windows NT не содержит средств синхронизации задач. Поэтому для того чтобы функции графического интерфейса работали правильно, такую синхронизацию должно выполнять само приложение.

Функция WndProc_OnCommand

Эта функция обрабатывает сообщение WM_COMMAND, поступающее от главного меню приложения. Она не имеет никаких особенностей.

Функция задачи PaintEllipse

Функция задачи PaintEllipse (наряду с другими двумя функциями задач PaintRect и PaintText) запускается при помощи функции _createthread во время инициализации главного окна приложения.

Прежде всего функция PaintEllipse выполняет инициализацию генератора случайных чисел, передавая функции инициализации srand в качестве начального значения идентификатор окна приложения. Вы можете, разумеется, использовать здесь любое другое 32-разрядное значение.

Затем функция PaintEllipse входит в цикл, который будет завершен при установке глобальной переменной fTerminate в состояние TRUE. Это произойдет при уничтожении главного окна приложения.

Перед рисованием эллипса функция PaintEllipse входит в критическую секцию csWindowPaint, что необходимо для синхронизации с другими задачами, выполняющими рисование в окне приложения.

Способ рисования эллипса не имеет никаких особенностей. Соответствующая функция Ellipse была нами описана в 14 томе “Библиотеки системного программиста”, который называется “Графичский интерфейс GDI в MS Windows”.

После того как эллипс будет нарисован, мы покидаем критическую секцию, вызывая функцию LeaveCriticalSection, и с помощью функции Sleep выполняем небольшую задержку. Функция Sleep приостанавливает выполнение задачи на количество миллисекунд, указанное в единственном параметере. Во время задержки задача не получает квантов времени, поэтому ожидание, выполняемое с помощью фукнции Sleep, не снижает производительности системы.

Функция задачи PaintRect

Функция задачи PaintRect аналогична только что рассмотренной функции PaintEllipse, за исключением того что она рисует прямоугольники. Для инициализации генератора случайных чисел используется другое значение, а рисование прямоугольника выполняется функцией Rectangle. Эта функция имеет такие же параметры, что и функция Ellipse.

Для синхронизации задача PaintRect использует все ту же критическую секцию csWindowPaint.

Функция задачи PaintText

Задача PaintText рисует текстовую строку TEXT, используя для этого функцию TextOut, описанную в 11 томе “Библиотеки системного программсита”. Синхронизация задачи выполняется с помощью критической секции csWindowPaint.

Приложение MultiMDI

В предыдущем разделе мы привели исходные тексты однооконного мультизадачного приложения. Больший интерес, на наш взгляд, имеет создание многооконного MDI-приложения, в котором для каждого окна создается своя задача. В качестве шаблона для создания такого приложения вы можете взять исходные тексты приложения MultiMDI, которые мы приведем в этом разделе.

В главном окне приложения MultiMDI вы можете создать дочерние MDI-окна, в которых выполняется циклическое отображение эллипсов случайной формы и цвета (рис. 2.4).

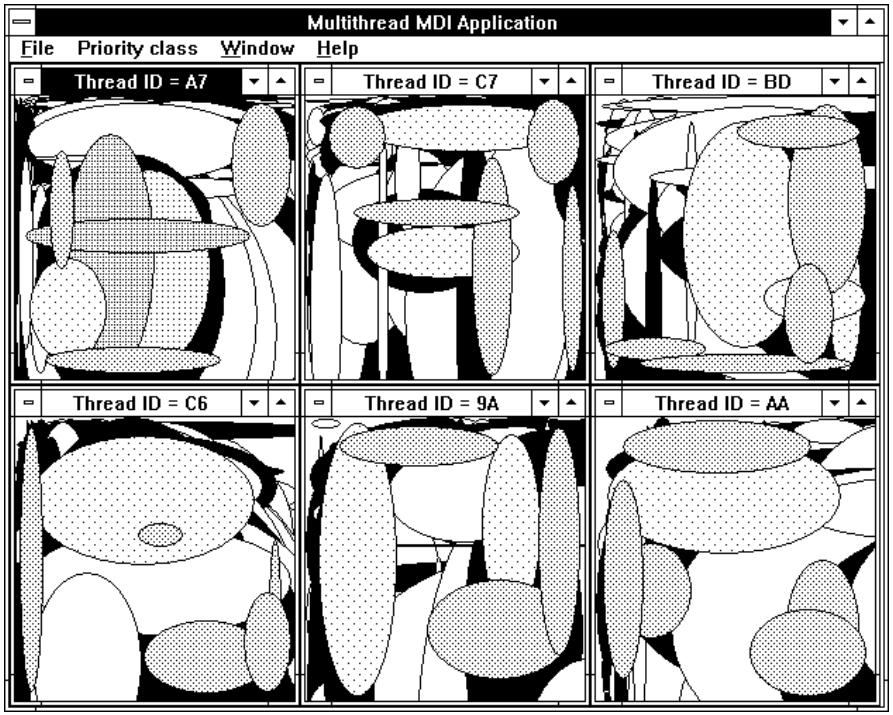

Рис. 2.4. Мультизадачное MDI-приложение MultiMDI

Кроме того что вы можете создавать дочерние окна и с помощью стандартного для MDI-приложений меню Window изменять расположение дочерних окон и представляющих их в минимизированном виде пиктограмм, у вас есть возможность управлять классом приоритета процесса, в рамках которого выполняется приложение, а также устанавливать относительный приоритет отдельных задач, приостанавливать их, возобновлять выполнение приостановленных задач, удалять задачи и закрывать дочерние окна.

С помощью меню Priotitry class, показанном на рис. 2.5, вы можете устанавливать один из четырех классов приоритета текущего процесса.

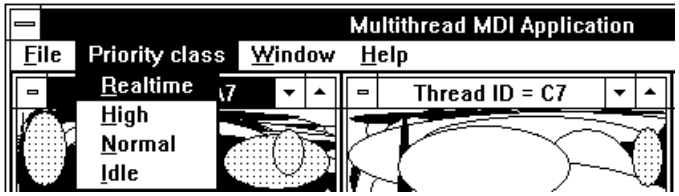

Рис. 2.5. Меню Priotitry class для установки приоритета процесса

Если сделать щелчок правой клавишей мыши во внутренней области дочернего MDI-окна, на экране появится плавающее меню, с помощью которого можно управлять задачей, запущенной для данного окна. Это меню показано на рис. 2.6.

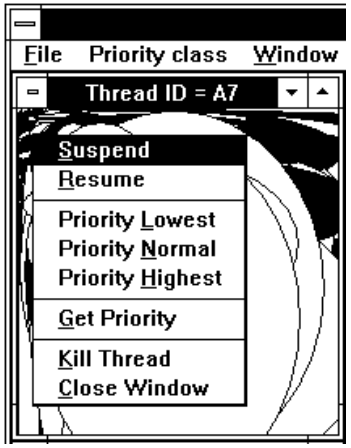

Рис. 2.6. Плавающее меню, предназначенное для управления задачей, запущенной для MDI-окна

С помощью строки Suspend можно приостановить работу задачи, а с помощью строки Resume - восстановить. Так как для каждой задачи система создает счетчик приостановок, то если вы два раза подряд приостановили задачу, выбрав строку Suspend, то для возобновления ее работы вам придется два раза выбрать строку Resume.

Изменяя относительный приоритет отдельных окон, вы можете заметить, что скорость перерисовки эллипсов также изменяется. Если вы работаете на быстром компьютере, эффект будет заметнее, если создать не менее 20 - 30 дочерних MDI-окон.

Выбрав строку Get Priority, вы можете узнать текущий относительный приоритет для любого дочернего MDI-окна. Значение относительного приоритета будет показано в отдельной диалоговой панели (рис. 2.7).

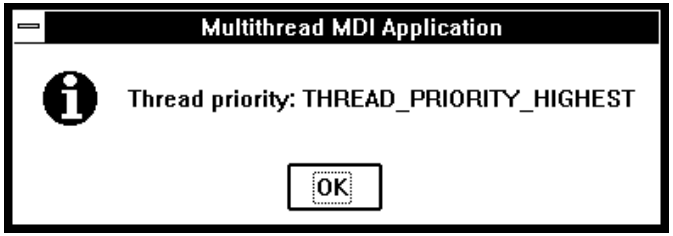

Рис. 2.7. Просмотр относительного приоритета задачи, запущенной для MDI-окна

Выбрав строку Kill Thread, вы принудительно завершите работу соответствующей задачи, после чего она не будет отзываться на любые команды.

С помощью строки Close Window вы можете закрыть любой дочернее MDI-окно, в том числе то, для которого было выполнено принудительное завершение работы задачи. Дочернее окно можно также закрыть, сделав двойной щелчок левой клавишей мыши по системному меню дочернего окна. В меню Window есть строка Close all, позволяющая закрыть сразу все дочерние MDI-окна.

Исходные тексты приложения

В качестве прототипа для создания исходных текстов приложения MultiMDI мы взяли исходные тексты приложения MDIAPP, описанного в 17 томе “Библиотеки системного программиста”, который называется “Microsoft Windows 3.1 для программиста. Дополнительные главы”. В этом томе описаны принципы работы MDI-приложений, которые в Microsoft Windows NT остались такими же, что и в Microsoft Windows версии 3.1.

В те фрагменты кода, которые выполняют создание дочерних MDI-окон, мы внесли небольшие изменения, связанные с использованием мультизадачного режима работы. В частности, для создания MDI-окна мы использовали функцию CreateMDIWindow, которая, как сказано в документации SDK, позволяет создавать для дочерних окон отдельные задачи.

Кроме того, для обеспечения необходимой синхронизации главной задачи и задач, запущенных для дочерних MDI-окон, мы создаем критические секции (по одной для каждого дочернего MDI-окна).

Главный файл исходных текстов приложения MultiMDI приведен в листинге 2.5.

Листинг 2.5. Файл multimdi/multimdi.c

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <stdio.h>
#include "resource.h"
#include "afxres.h"
#include "multimdi.h"

// Имена классов окна
char const szFrameClassName[] = "MDIAppClass";
char const szChildClassName[] = "MDIChildAppClass";

// Заголовок окна
char const szWindowTitle[] = "Multithread MDI Application";

HINSTANCE hInst;

HWND hwndFrame;  // окно Frame Window
HWND hwndClient; // окно Client Window
HWND hwndChild;  // окно Child Window

// Структура, которая создается для каждого дочернего окна
typedef struct _CHILD_WINDOW_TAG
{
  // Признак активности задачи
  BOOL fActive;

  // Критическая секция для рисования в окне
  CRITICAL_SECTION csChildWindowPaint;

  // Идентификатор задачи
  HANDLE hThread;
} CHILD_WINDOW_TAG;

typedef CHILD_WINDOW_TAG *LPCHILD_WINDOW_TAG;
```

```
// ====================================
// Функция WinMain
// ====================================
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
  MSG  msg;         // структура для работы с сообщениями
  hInst = hInstance; // сохраняем идентификатор приложения

  // Инициализируем приложение
  if(!InitApp(hInstance))
    return FALSE;

  // Создаем главное окно приложения - Frame Window
  hwndFrame = CreateWindow(
    szFrameClassName,    // имя класса окна
    szWindowTitle,       // заголовок окна
    WS_OVERLAPPEDWINDOW, // стиль окна
    CW_USEDEFAULT, 0,    // задаем размеры и расположение
    CW_USEDEFAULT, 0,    // окна, принятые по умолчанию
    0,                   // идентификатор родительского окна
    0,                   // идентификатор меню
    hInstance,           // идентификатор приложения
    NULL);   // указатель на дополнительные параметры

  // Если создать окно не удалось, завершаем приложение
  if(!hwndFrame)
    return FALSE;

  // Рисуем главное окно
  ShowWindow(hwndFrame, nCmdShow);
  UpdateWindow(hwndFrame);

  // Запускаем цикл обработки сообщений
  while(GetMessage(&msg, NULL, 0, 0))
  {
    // Трансляция для MDI-приложения
    if(!TranslateMDISysAccel(hwndClient, &msg))
    {
      TranslateMessage(&msg);
      DispatchMessage(&msg);
    }
  }
  return msg.wParam;
}

// ====================================
// Функция InitApp
// Выполняет регистрацию класса окна
// ====================================

BOOL
InitApp(HINSTANCE hInstance)
{
  ATOM aWndClass; // атом для кода возврата
  WNDCLASS wc;    // структура для регистрации

  // Регистрируем класс для главного окна приложения
  // (для окна Frame Window)
  memset(&wc, 0, sizeof(wc));
  wc.lpszMenuName  = "APP_MENU";
  wc.style         = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc   = (WNDPROC)FrameWndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon(hInstance, "APP_ICON");
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_APPWORKSPACE + 1);
  wc.lpszClassName = (LPSTR)szFrameClassName;
```

```
  aWndClass = RegisterClass(&wc);

  if(!aWndClass)
    return FALSE;

  // Регистрируем класс окна для
  //дочернего окна Document Window
  memset(&wc, 0, sizeof(wc));
  wc.lpszMenuName  = 0;
  wc.style         = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc   = (WNDPROC)ChildWndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon(hInstance, "APPCLIENT_ICON");
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
  wc.lpszClassName = (LPSTR)szChildClassName;
  aWndClass = RegisterClass(&wc);

  if(!aWndClass)
    return FALSE;

  return TRUE;
}

// ====================================
// Функция FrameWndProc
// ====================================

LRESULT CALLBACK
FrameWndProc(HWND hwnd, UINT msg,
             WPARAM wParam, LPARAM lParam)
{
  HWND   hwndChild;
  HANDLE hThread;
  DWORD  dwIDThread;
  CHAR   szBuf[255];

  // Структура для создания окна Client Window
  CLIENTCREATESTRUCT clcs;

  // Указатель на структуру для хранения
  // состояния дочернего окна
  LPCHILD_WINDOW_TAG lpTag;

  switch (msg)
  {
    // При создании окна Frame Window создаем
    // окно Client Window, внутри которого будут создаваться
    // дочерние окна Document Window
    case WM_CREATE:
    {
      // Получаем и сохраняем в структуре clcs идентификатор
      // временного меню Window. Так как это третье слева
      // меню, его позиция равна 2 (меню File имеет позицию 0)
      clcs.hWindowMenu = GetSubMenu(GetMenu(hwnd), 2);

      // Идентификатор первого дочернего окна Document Window
      clcs.idFirstChild = 500;

      // Создаем окно Client Window
      hwndClient = CreateWindow(
        "MDICLIENT",    // имя класса окна
        NULL,           // заголовок окна
        WS_CHILD | WS_CLIPCHILDREN | WS_VISIBLE |
          WS_HSCROLL | WS_VSCROLL,
        0, 0, 0, 0,
        hwnd,           // идентификатор родительского окна
        (HMENU)1,       // идентификатор меню
        hInst,          // идентификатор приложения
        (LPSTR)&clcs);// указатель на дополнительные параметры
```

```
  break;
}

// Обработка сообщений от главного меню приложения
case WM_COMMAND:
{
  switch (wParam)
  {
    // Создание нового окна Document Window
    case CM_FILENEW:
    {
      // Используем функцию CreateMDIWindow, которая
      // специально предназначена для работы с
      // многозадачными приложениями MDI
      hwndChild = CreateMDIWindow(
        (LPSTR)szChildClassName,      // класс окна
        "MDI Child Window",           // заголовок окна
        0,                            // дополнительные стили
        CW_USEDEFAULT, CW_USEDEFAULT, // размеры окна
        CW_USEDEFAULT, CW_USEDEFAULT, // Document Window
        hwndClient,  // идентификатор окна Client Window
        hInst,       // идентификатор приложения
        0);          // произвольное значение

      // Получаем память для структуры, в которой будет
      // хранится состояние окна
      lpTag = malloc(sizeof(CHILD_WINDOW_TAG));

      // Устанавливаем признак активности
      lpTag->fActive = 1;

      // Инициализируем критическую секцию
      InitializeCriticalSection(
        &(lpTag->csChildWindowPaint));

      // Устанавливаем адрес структуры состояния в
      // памяти окна
      SetWindowLong(hwndChild, GWL_USERDATA, (LONG)lpTag);

      // Создаем задачу для дочернего окна
      hThread = CreateThread(NULL, 0,
        (LPTHREAD_START_ROUTINE)ThreadRoutine,
        (LPVOID)hwndChild, 0,(LPDWORD)&dwIDThread);

      if(hThread == NULL)
      {
        MessageBox(hwnd,"Ошибка при создании задачи",
                   szWindowTitle, MB_OK | MB_ICONEXCLAMATION);
      }

      // Сохраняем идентификатор созданной задачи
      lpTag->hThread = hThread;

      // Отображаем идентификатор задачи в заголовке
      // дочернего окна
      sprintf(szBuf, "Thread ID = %lX", dwIDThread);
      SetWindowText(hwndChild, szBuf);

      break;
    }

    // Размещение окон Document Window рядом друг с другом
    case CM_WINDOWTILE:
    {
      SendMessage(hwndClient, WM_MDITILE, 0, 0);
      break;
    }

    // Размещение окон Document Window с перекрытием
    case CM_WINDOWCASCADE:
    {
```

```
  SendMessage(hwndClient, WM_MDICASCADE, 0, 0);
  break;
}

// Размещение пиктограм минимизированых окон
// Document Window в нижней части окна Client Window
case CM_WINDOWICONS:
{
  SendMessage(hwndClient, WM_MDIICONARRANGE, 0, 0);
  break;
}

// Уничтожение всех окон Document Window
case CM_WINDOWCLOSEALL:
{
  HWND hwndTemp;

  // Скрываем окно Client Window для того чтобы
  // избежать многократной перерисовки окон
  // Document Window во время их уничтожения
  ShowWindow(hwndClient, SW_HIDE);

  while(TRUE)
  {
    // Получаем идентификатор дочернего окна
    // для окна Client Window
    hwndTemp = GetWindow(hwndClient, GW_CHILD);
    // Если дочерних окон больше нет, выходим из цикла
    if(!hwndTemp)
      break;

    // Пропускаем окна-заголовки
    while(hwndTemp && GetWindow(hwndTemp, GW_OWNER))
      hwndTemp = GetWindow(hwndTemp, GW_HWNDNEXT);

    // Удаляем дочернее окно Document Window
    if(hwndTemp)
    {
      // Завершаем задачу, запущенную для окна
      lpTag = (LPCHILD_WINDOW_TAG)GetWindowLong(
        hwndTemp, GWL_USERDATA);
      lpTag->fActive = 0;

      SendMessage(hwndClient, WM_MDIDESTROY,
        (WPARAM)hwndTemp, 0);
    }
    else
      break;
  }

  // Отображаем окно Client Window
  ShowWindow(hwndClient, SW_SHOW);
  break;
}

// Устанавливаем классы приоритета процесса
case ID_PRIORITYCLASS_REALTIME:
{
  SetPriorityClass(GetCurrentProcess(),
    REALTIME_PRIORITY_CLASS);
  break;
}
case ID_PRIORITYCLASS_HIGH:
{
  SetPriorityClass(GetCurrentProcess(),
    HIGH_PRIORITY_CLASS);
  break;
}
case ID_PRIORITYCLASS_NORMAL:
{
  SetPriorityClass(GetCurrentProcess(),
    NORMAL_PRIORITY_CLASS);
```

```
    break;
   }
   case ID_PRIORITYCLASS_IDLE:
   {
    SetPriorityClass(GetCurrentProcess(),
      IDLE_PRIORITY_CLASS);
    break;
   }

   case CM_HELPABOUT:
   {
    MessageBox(hwnd,
            "Демонстрация использования мультизадачности\n"
      "в MDI-приложениях\n"
      "(C) Alexandr Frolov, 1996\n"
      "Email: frolov@glas.apc.org",
      szWindowTitle, MB_OK | MB_ICONINFORMATION);
    break;
   }

   // Завершаем работу приложения
   case CM_FILEEXIT:
   {
    DestroyWindow(hwnd);
    break;
   }

   default:
    break;
  }

  // Определяем идентификатор активного окна
  // Document Window
  hwndChild =
   (HWND)LOWORD(SendMessage(hwndClient,
   WM_MDIGETACTIVE, 0, 0l));

    // Если это окно, посылаем ему сообщение WM_COMMAND
    if(IsWindow(hwndChild))
     SendMessage(hwndChild, WM_COMMAND, wParam, lParam);

    return DefFrameProc(
      hwnd, hwndClient, msg, wParam, lParam);
   }

   case WM_DESTROY:
   {
    PostQuitMessage(0);
    break;
   }

   default:
    break;
  }
  return DefFrameProc(hwnd, hwndClient, msg, wParam, lParam);
}

// ====================================
// Функция ChildWndProc
// ====================================

LRESULT CALLBACK
ChildWndProc(HWND hwnd, UINT msg,
        WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT ps;
  RECT rc;
  LPCHILD_WINDOW_TAG lpMyWndTag;
  HMENU hmenuPopup;
  POINT pt;
  CHAR szBuf[256];
```

```
switch (msg)
{
 case WM_PAINT:
 {
  // Получаем адрес структуры состояния окна
  lpMyWndTag =
   (LPCHILD_WINDOW_TAG)GetWindowLong(hwnd, GWL_USERDATA);

  // Входим в критическую секцию
  EnterCriticalSection(&(lpMyWndTag->csChildWindowPaint));

  // Перерисовываем внутреннюю область дочернего окна
  hdc = BeginPaint(hwnd, &ps);
  GetClientRect(hwnd, &rc);

  DrawText(hdc, "Child Window", -1, &rc,
   DT_SINGLELINE | DT_CENTER | DT_VCENTER);

  EndPaint(hwnd, &ps);

  // Выходим из критической секции
  LeaveCriticalSection(&(lpMyWndTag->csChildWindowPaint));
  break;
 }

 case WM_CLOSE:
 {
  // Сбрасываем признак активности задачи
  lpMyWndTag =
   (LPCHILD_WINDOW_TAG)GetWindowLong(hwnd, GWL_USERDATA);

  lpMyWndTag->fActive = 0;
  break;
 }

// Когда пользователь нажимает правую кнопку мыши
// в дочернем окне, отображаем плавающее меню
case WM_RBUTTONDOWN:
{
 pt.x = LOWORD(lParam);
 pt.y = HIWORD(lParam);
 ClientToScreen(hwnd, &pt);

 hmenuPopup = GetSubMenu(
  LoadMenu(hInst, "IDR_POPUPMENU"), 0);
 TrackPopupMenu(hmenuPopup,
  TPM_CENTERALIGN | TPM_LEFTBUTTON,
  pt.x, pt.y, 0, hwnd, NULL);
 DestroyMenu(hmenuPopup);
 break;
}

// Обрабатываем команды, поступающие от плавающего меню
case WM_COMMAND:
{
 switch (wParam)
 {
  // Приостановка выполнения задачи
  case ID_THREADCONTROL_SUSPEND:
  {
   lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
     hwnd, GWL_USERDATA);

   // Входим в критическую секцию
   EnterCriticalSection(
    &(lpMyWndTag->csChildWindowPaint));

   SuspendThread(lpMyWndTag->hThread);

   // Выходим из критической секции
```

```
  LeaveCriticalSection(
    &(lpMyWndTag->csChildWindowPaint));
  break;
}

// Возобновление выполнения задачи
case ID_THREADCONTROL_RESUME:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);
  ResumeThread(lpMyWndTag->hThread);
  break;
}

// Изменение относительного приоритета
case ID_THREADCONTROL_PRIORITYLOWEST:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);
  SetThreadPriority(lpMyWndTag->hThread,
    THREAD_PRIORITY_LOWEST);
  break;
}
case ID_THREADCONTROL_PRIORITYNORMAL:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);
  SetThreadPriority(lpMyWndTag->hThread,
    THREAD_PRIORITY_NORMAL);
  break;
}
case ID_THREADCONTROL_PRIORITYHIGHEST:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);
  SetThreadPriority(lpMyWndTag->hThread,
    THREAD_PRIORITY_HIGHEST);
  break;
}

// Определение и отображение относительного приоритета
case ID_THREADCONTROL_GETPRIORITY:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);

  strcpy(szBuf, "Thread priority: ");

  switch (GetThreadPriority(lpMyWndTag->hThread))
  {
    case THREAD_PRIORITY_LOWEST:
    {
      strcat(szBuf, "THREAD_PRIORITY_LOWEST");
      break;
    }
    case THREAD_PRIORITY_BELOW_NORMAL:
    {
      strcat(szBuf, "THREAD_PRIORITY_BELOW_NORMAL");
      break;
    }
    case THREAD_PRIORITY_NORMAL:
    {
      strcat(szBuf, "THREAD_PRIORITY_NORMAL");
      break;
    }
    case THREAD_PRIORITY_ABOVE_NORMAL:
    {
```

```
      strcat(szBuf, "THREAD_PRIORITY_ABOVE_NORMAL");
      break;
     }
     case THREAD_PRIORITY_HIGHEST:
     {
      strcat(szBuf, "THREAD_PRIORITY_HIGHEST");
      break;
     }
    }

    MessageBox(hwnd, szBuf,
     szWindowTitle, MB_OK | MB_ICONINFORMATION);

    break;
   }

   // Удаление задачи
   case ID_THREADCONTROL_KILLTHREAD:
   {
    lpMyWndTag =
     (LPCHILD_WINDOW_TAG)GetWindowLong(
     hwnd, GWL_USERDATA);
    TerminateThread(lpMyWndTag->hThread, 5);
    break;
   }

   // Уничтожение дочернего окна
   case ID_THREADCONTROL_CLOSEWINDOW:
   {
    SendMessage(hwndClient, WM_MDIDESTROY,
     (WPARAM)hwnd, 0);
    break;
   }
   default:
    break;
  }
```

```
    break;
   }
   default:
    break;
  }
  return DefMDIChildProc(hwnd, msg, wParam, lParam);
}

// ====================================
// Функция ThreadRoutine
// Задача, которая выполняется для каждого
// дочернего окна
// ====================================

DWORD ThreadRoutine(HWND hwnd)
{
  LONG lThreadWorking = 1L;
  HDC hDC;
  RECT rect;
  LONG xLeft, xRight, yTop, yBottom;
  short nRed, nGreen, nBlue;
  HBRUSH hBrush, hOldBrush;
  LPCHILD_WINDOW_TAG lpMyWndTag;

  // Определение адреса структуры, содержащей состояние
  // дочернего окна
  lpMyWndTag =
   (LPCHILD_WINDOW_TAG)GetWindowLong(hwnd, GWL_USERDATA);

  // Инициализация генератора случайных чисел
  srand((unsigned int)hwnd);

  // Задача работает в бесконечном цикле,
  // который будет прерван при удалении дочернего окна
  while(TRUE)
  {
```

```
// Получаем признак активности задачи
// Если он не равен 1, завершаем задачу
if(!lpMyWndTag->fActive)
  break;

// Входим в критическую секцию
EnterCriticalSection(&(lpMyWndTag->csChildWindowPaint));

// Отображаем эллипс, имеющий случайный цвет,
// форму и расположение

// Получаем контекст отображения
hDC = GetDC(hwnd);

// Получаем случайные цветовые компоненты
nRed   = rand() % 255;
nGreen = rand() % 255;
nBlue  = rand() % 255;

// Получаем случйные размеры эллипса
GetWindowRect(hwnd, &rect);

xLeft   = rand() % (rect.left   + 1);
xRight  = rand() % (rect.right  + 1);
yTop    = rand() % (rect.top    + 1);
yBottom = rand() % (rect.bottom + 1);

// Создаем кисть на основе случайных цветов
hBrush = CreateSolidBrush(RGB(nRed, nGreen, nBlue));

// Выбираем кисть в контекст отображения
hOldBrush = SelectObject(hDC, hBrush);

// Рисуем эллипс
Ellipse(hDC, min(xLeft, xRight), min(yTop, yBottom),
        max(xLeft, xRight), max(yTop, yBottom));

  // Выбираем старую кисть в контекст отображения
  SelectObject(hDC, hOldBrush);

  // Удаляем созданную кисть
  DeleteObject(hBrush);

  // Освобождаем контекст отображения
  ReleaseDC(hwnd, hDC);

  // Выходим из критической секции
  LeaveCriticalSection(&(lpMyWndTag->csChildWindowPaint));

  // Выполняем задержку на 1 мс
  Sleep(1);
 }

 // Перед обычным завершением задачи удаляем критическую
 // секцию и освобождаем память, полученную для
 // хранения состояния дочернего окна
 DeleteCriticalSection(&(lpMyWndTag->csChildWindowPaint));
 free(lpMyWndTag);

 // Оператор return завершает выполнение задачи
 return 0;
}
```

Прототипы функций, определенных в приложении MultiMDI, определены в файле multimdi.h (листинг 2.6).

Листинг 2.6. Файл multimdi/multimdi.h

```
// Прототипы функций
BOOL InitApp(HINSTANCE);
LRESULT CALLBACK FrameWndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK ChildWndProc(HWND, UINT, WPARAM, LPARAM);
DWORD ThreadRoutine(HWND hwnd);
VOID AddThreadToList(HANDLE hThread);
```

Файл resource.h (листинг 2.7), создаваемый автоматически, содержит определение констант для ресурсов приложения MultiMDI.

Листинг 2.7. Файл multimdi/resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by MDIAPP.RC
//
#define CM_HELPABOUT                100
#define CM_FILEEXIT                 101
#define CM_FILENEW                  102
#define CM_WINDOWTILE               103
#define CM_WINDOWCASCADE            104
#define CM_WINDOWICONS              105
#define CM_WINDOWCLOSEALL           106
#define ID_THREADCONTROL_SUSPEND       40001
#define ID_THREADCONTROL_RESUME        40002
#define ID_THREADCONTROL_PRIORITYLOWEST 40003
#define ID_THREADCONTROL_PRIORITYNORMAL 40004
#define ID_THREADCONTROL_PRIORITYHIGHEST 40005
#define ID_THREADCONTROL_GETPRIORITY    40006
#define ID_THREADCONTROL_KILLTHREAD     40007
#define ID_THREADCONTROL_CLOSEWINDOW    40008
#define ID_PRIORITYCLASS_REALTIME       40009
#define ID_PRIORITYCLASS_NORMAL         40010
#define ID_PRIORITYCLASS_HIGH           40011
#define ID_PRIORITYCLASS_IDLE           40012

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NO_MFC                 1
#define _APS_NEXT_RESOURCE_VALUE        102
#define _APS_NEXT_COMMAND_VALUE         40013
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

Ресурсы приложения MultiMDI определены в файле mdiapp.rc, который приведен в листинге 2.8.

Листинг 2.8. Файл multimdi/mdiapp.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////
// Menu
//

APP_MENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
```

```
    BEGIN
        MENUITEM "&New",          CM_FILENEW
        MENUITEM SEPARATOR
        MENUITEM "E&xit",         CM_FILEEXIT
    END
    POPUP "Priority class"
    BEGIN
        MENUITEM "&Realtime",     ID_PRIORITYCLASS_REALTIME
        MENUITEM "&High",         ID_PRIORITYCLASS_HIGH
        MENUITEM "&Normal",       ID_PRIORITYCLASS_NORMAL
        MENUITEM "&Idle",         ID_PRIORITYCLASS_IDLE
    END
    POPUP "&Window"
    BEGIN
        MENUITEM "&Tile",         CM_WINDOWTILE
        MENUITEM "&Cascade",      CM_WINDOWCASCADE
        MENUITEM "Arrange &Icons", CM_WINDOWICONS
        MENUITEM "Close &All",    CM_WINDOWCLOSEALL
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",     CM_HELPABOUT
    END
END

IDR_POPUPMENU MENU DISCARDABLE
BEGIN
    POPUP "&Thread Control"
    BEGIN
        MENUITEM "&Suspend",        ID_THREADCONTROL_SUSPEND
        MENUITEM "&Resume",         ID_THREADCONTROL_RESUME
        MENUITEM SEPARATOR
        MENUITEM "Priority &Lowest", ID_THREADCONTROL_PRIORITYLOWEST
        MENUITEM "Priority &Normal", ID_THREADCONTROL_PRIORITYNORMAL
        MENUITEM "Priority &Highest",ID_THREADCONTROL_PRIORITYHIGHEST
        MENUITEM SEPARATOR
        MENUITEM "&Get Priority",   ID_THREADCONTROL_GETPRIORITY
        MENUITEM SEPARATOR
        MENUITEM "&Kill Thread",    ID_THREADCONTROL_KILLTHREAD
        MENUITEM "&Close Window",   ID_THREADCONTROL_CLOSEWINDOW
    END
END

//////////////////////////////////////////////////////////////
// Icon
//

// Icon with lowest ID value placed first to ensure
// application icon
// remains consistent on all systems.
APP_ICON            ICON    DISCARDABLE    "multimdi.ico"
APPCLIENT_ICON      ICON    DISCARDABLE    "mdicl.ico"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
```

```
    "\0"
END


#endif    // APSTUDIO_INVOKED


#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 3 resource.
//
/////////////////////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Определения и глобальные переменные

Так как для запуска задач в приложении MultiMDI мы пользуемся функцией CreateThread, нам не нужно включать файл process.h.

В глобальных переменных szFrameClassName и szChildClassName хранятся указатели, соответственно, на имена классов для окна Frame Window и окна Child Window.

Адрес заголовка приложения хранится в глобальной переменной szWindowTitle.

Переменная hInst используется для хранения идентификатора приложения.

Переменные hwndFrame, hwndClient и hwndChild используются, соответственно, для хранения идентификаторов окон Frame Window, Client Window и дочернего окна (в момент его создания).

Для каждого дочернего MDI-окна мы создаем структуру типа CHILD_WINDOW_TAG, в которой сохраняем такую информацию, как признак активности задачи, запущенной для этого окна, критическую секцию для рисования в окне, а также идентификатор задачи, запущенной для окна:

```
typedef struct _CHILD_WINDOW_TAG
{
  BOOL fActive;
  CRITICAL_SECTION csChildWindowPaint;
  HANDLE hThread;
} CHILD_WINDOW_TAG;
```

Кроме того, мы определили указатель на эту структуру:

```
typedef CHILD_WINDOW_TAG *LPCHILD_WINDOW_TAG;
```

Описание функций приложения

Приведем описание функций, определенных в нашем приложении. Для экономии места мы не будем подробно описывать фрагменты кода, связанные с созданием дочерних MDI-окон. При необходимости обращайтесь к 17 тому “Библиотеки системного программиста”, в котором приведена подробная информация о создании MDI-приложений.

Функция WinMain

Функция WinMain характерна для MDI-приложений. В ней вызывается функция инициализации приложения InitApp, которая регистрирует классы главного окна Frame Window и дочернего MDI-окна Document Window. Затем создается и отображается главное окно, а затем запускается цикл обработки сообщений. В этом цикле вызывается функция трансляции TranslateMDISysAccel, котурую используют MDI-приложения.

Функция FrameWndProc

Эта функция обрабатывает сообщения для главного окна приложения Frame Window, в частности, сообщения, которые поступают от главного меню приложения.

При создании окна функция FrameWndProc получает сообщение WM_CREATE. В ответ на это сообщение она сохраняет в структуре clcs типа CLIENTCREATESTRUCT идентификатор временного меню Window. При создании дочерних MDI-окон это меню будет расширяться.

Затем обработчик сообщения WM_CREATE сохраняет в структуре clcs идентификатор первого дочернего MDI-окна, для которого выбрано произвольное число 500. После этого выполняется создание дочернего MDI-окна, для чего вызывается функция CreateWindow. Описание этой функции вы найдете в 11 томе “Библиотеки системного программиста”.

Обработчик сообщения WM_COMMAND получает управление, когда пользователь выбирает строки главного меню приложения.

Создание нового дочернего окна

При выборе из меню File строки New создается новое дочернее MDI-окно и задача для него. Первое действие выполняется с помощью функции CreateMDIWindow, специально созданной для мультизадачных MDI-приложений. Задача создается при помощи функции CreateThread. Рассмотрим процесс создания дочернего окна подробнее.

Параметры фукнции CreateMDIWindow являются комбинацией значений, сохраняемых в структуре MDICREATESTRUCT полюс идентификатор окна Client Window. Напомним, что адрес этой структуры передается через второй параметр сообщения WM_MDICREATE, предназначенного для создания дочерних MDI-окон в однозадачных приложениях. Вот как создается такое окно в нашем приложении:

```
hwndChild = CreateMDIWindow(
  LPSTR)szChildClassName,       // класс окна
  "MDI Child Window",           // заголовок окна
  0,                            // дополнительные стили
  CW_USEDEFAULT, CW_USEDEFAULT,  // размеры окна
  CW_USEDEFAULT, CW_USEDEFAULT,  //    Document Window
  hwndClient,           // идентификатор окна Client Window
  hInst,                  // идентификатор приложения
  0);                     // произвольное значение
```

Вы можете сравнить это со способом, описанным нами в 17 томе “Библиотеки системного программиста” в разделе “Создание и уничтожение окна Document Window”.

Для каждого дочернего окна мы создаем и инициализируем структуру типа CHILD_WINDOW_TAG, содержащую такие значения, как идентификатор задачи, запущенной для окна, признак активности задачи и критическую секцию. Память для структуры мы получаем простейшим способом - с помощью функции malloc:

```
lpTag = malloc(sizeof(CHILD_WINDOW_TAG));
```

Далее в процессе инициализации этой структуры мы устанавливаем начальное знзачение для признака активности и выполняем инициализацию критической секции:

```
lpTag->fActive = 1;
InitializeCriticalSection(&(lpTag->csChildWindowPaint));
```

Это нужно сделать обязательно до запуска задачи, которая будет пользоваться критической секцией и проверять флаг активности.

Адрес структуры мы сохраняем в области данных окна, используя для этого функцию SetWindowLong:

```
SetWindowLong(hwndChild, GWL_USERDATA, (LONG)lpTag);
```

Впоследствии этот адрес будет извлечен функцией задачи.

На следующем шаге мы выполняем создание задачи для дочернего MDI-окна, которая будет заниматься рисованием произвольных эллипсов. Для запуска используется функция CreateThread:

```
hThread = CreateThread(NULL, 0,
  (LPTHREAD_START_ROUTINE)ThreadRoutine,
  (LPVOID)hwndChild, 0,(LPDWORD)&dwIDThread);
```

В качестве третьего параметра мы передаем этой функции адрес функции задачи ThreadRoutine, а в качестве четвертого - идентификатор дочернего MDI-окна, который необходим для выполенния рисования.

Заметим, что мы могли бы передать через четвертый параметр функции CreateThread адрес структуры CHILD_WINDOW_TAG, дополнив эту структуру полем для хранения идентификатора дочернего MDI-окна. При этом функция задачи получила бы доступ ко всем необходимым ей параметрам. Однако в нашем приложении мы продемонстрировали оба способа, связанных с использованием параметра функции задачи и памяти окна.

После того как задача создана, ее идентификатор сохраняется в структуре CHILD_WINDOW_TAG:

```
lpTag->hThread = hThread;
```

В дальнейшем это значение будет использовано для управления задачей - для ее приостановки, возобновления выполнения, изменения относительного приоритета и так далее.

На заключительном шаге в заголовке дочернего MDI-окна отображается системный номер задачи, который записывается функцией CreateThread в переменную dwIDThread:

```
sprintf(szBuf, "Thread ID = %lX", dwIDThread);
SetWindowText(hwndChild, szBuf);
```

Напомним, что системный номер задачи и ее идентификатор - разные по смыслу и значению величины.

Обработка сообщений от меню Window

Меню Window предназначено для управления дочерними MDI-окнами. Выбирая строки этого меню, пользователь может упорядочить расположение окон одним из двух способов, упорядочить расположение пиктограмм минимизированных окон, а также закрыть все дочерние MDI-окна. Все эти операции выполняются посылкой соответствующих сообщений окну Client Window. Для экономии места мы не будем останавливаться на подробном описании соответствующих фрагментов кода. При необходимости обратитесь к 17 тому “Библиотеки системного программиста”.

Обработка сообщений от меню Priority

Меню Priority позволяет изменить класс приоритета процесса, в рамках которого работает приложение.

Процессам в нашей книге посвящена отдельная глава, однако изменение класса приоритета - достаточно простая операция, которая выполняется с помощью функции SetPriorityClass:

```
SetPriorityClass(GetCurrentProcess(),
  REALTIME_PRIORITY_CLASS);
```

В качестве первого параметра этой функции необходимо передать идентификатор процесса, класс приоритета которого будет изменяться. Мы

получаем идентификатор текущего процесса с помощью функции GetCurrentProcess.

Через второй параметр функции SetPriorityClass передается новое значение класса приоритета.

Функция ChildWndProc

Функция ChildWndProc обрабатывает сообщения, поступающие в дочерние MDI-окна.

Обработка сообщения WM_PAINT

Обработчик сообщения WM_PAINT получает управление, когда возникает необходимость перерисовать внутреннюю область дочернего MDI-окна. Однако заметим, что в то же самое время во внутренней области дочернего окна может рисовать функция задачи, запущенная для этого окна. В результате возникает необходимость выполнить синхронизацию главной задачи приложения и задачи дочернего окна.

Так как задачи, запущенные для дочерних окон, работают независимо друг от друга и от главной задачи приложения, мы будем выполнять синхронизацию каждой задачи при помощи отдельной критической секции. Эта критическая секция располагается в структуре типа CHILD_WINDOW_TAG и ее инициализация была выполнена перед созданием соответствующей задачи.

Для получения адреса структуры, который был записан в память дочернего окна, мы вызываем функцию GetWindowLong:

```
lpMyWndTag = (LPCHILD_WINDOW_TAG)GetWindowLong(hwnd, GWL_USERDATA);
```

После этого выполняется вход в критическую секцию, получение контекста отображения, рисование во внутренней области дочернего MDI-окна строки Child Window, освобождение контекста отображения и выход из критической секции:

```
EnterCriticalSection(&(lpMyWndTag->csChildWindowPaint));
hdc = BeginPaint(hwnd, &ps);
GetClientRect(hwnd, &rc);
DrawText(hdc, "Child Window", -1, &rc,
  DT_SINGLELINE | DT_CENTER | DT_VCENTER);
EndPaint(hwnd, &ps);
LeaveCriticalSection(&(lpMyWndTag->csChildWindowPaint));
```

Обработка сообщения WM_CLOSE

Это сообщение поступает в функцию дочернего окна при уничтожении последнего (например, если пользователь сделал двойной щелчок левой клавишей мыши по пиктограмме системного меню дочернего MDI-окна).

Задача обработчика сообщения WM_CLOSE заключается в сбросе признака активности задачи, в результате чего задача завершает свою работу:

```
lpMyWndTag = (LPCHILD_WINDOW_TAG)GetWindowLong(hwnd,
  GWL_USERDATA);
lpMyWndTag->fActive = 0;
```

Обработка сообщения WM_RBUTTONDOWN

Для управления задачами, запущенными в дочерних MDI-окнах, в нашем приложении используется плавающее меню. Это меню создается когда пользователь делает щелчок правой клавишей мыши во внутренней области дочернего MDI-окна.

Плавающее меню определено в ресурсах приложения с идентификатором IDR_POPUPMENU. Оно отображается на экране с помощью функции TrackPopupMenu. Соответствующая техника была нами описана в главе “Меню” 13 тома “Библиотеки системного программиста”.

Обработка сообщения WM_COMMAND

Сообщение WM_COMMAND поступает в функцию дочернего MDI-окна, когда пользователь выбирает строки плавающего меню.

Если пользователь выбирает из этого меню, например, строку Suspend, выполняется приостановка работы задачи, запущенной для данного дочернего MDI-окна. Приостановка выполняется при помощи функции SuspendThread. Идентификатор задачи, необходимый для нее, извлекается из поля hThread структуры типа CHILD_WINDOW_TAG:

```
lpMyWndTag = LPCHILD_WINDOW_TAG)GetWindowLong(hwnd,
  GWL_USERDATA);
EnterCriticalSection(&(lpMyWndTag->csChildWindowPaint));
SuspendThread(lpMyWndTag->hThread);
LeaveCriticalSection(&(lpMyWndTag->csChildWindowPaint));
```

Для возобновления выполнения приостановленной задачи мы использовали функцию ResumeThread:

```
ResumeThread(lpMyWndTag->hThread);
```

Заметим, что перед выполнением приостановки задачи мы входим в критическую секцию. Это необходимо для того, чтобы избежать полной блокировки главной задачи процесса в момент, когда блокировка рисующей задачи выполняется после входа одной из задач в критическую секцию. В самом деле, если этой произойдет, главная задача не сможет войти в критическую секцию, так как она уже занята другой задачей. Если при этом задача, вошедшая в критическую секцию, оказалась заблокирована до своего

выхода из критической секции, главная задача так и останется в состоянии ожидания. При этом пользователь не сможет, например, работать с меню приложения.

Относительный приоритет задачи изменяется фукнцией SetThreadPriority, как это показано ниже:

```
SetThreadPriority(lpMyWndTag->hThread,
  THREAD_PRIORITY_LOWEST);
```

Если выбрать из плавающего меню строку Get priority, с помощью функции GetThreadPriority определяется текущий относительный приоритет задачи, запущенной для данного дочернего MDI-окна. Значение этого приоритета отображается затем на экране при помощи простейшей диалоговой панели, создаваемой функцией MessageBox.

При выборе из плавающего меню строки Kill Thread задача будет принудительно уничтожена функцией TerminateThread:

```
TerminateThread(lpMyWndTag->hThread, 5);
```

В качестве кода завершения здесь передается произвольно выбранное нами значение 5.

С помощью плавающего меню вы можете удалить дочернее MDI-окно, завершив работу соответствующей задачи. Для этого окну Client Window посылается сообщение WM_MDIDESTROY:

```
SendMessage(hwndClient, WM_MDIDESTROY, (WPARAM)hwnd, 0);
```

Все остальное делает обработчик сообщения WM_CLOSE, который получит управление при удалении дочернего MDI-окна. А именно, этот обработчик сбрасывает признак активности задачи, в результате чего задача завершает свою работу.

Функция задачи ThreadRoutine

Задача ThreadRoutine запускается для каждого вновь создаваемого дочернего MDI-окна. Ее функция получает один параметр - идентификатор этого дочернего окна, который необходим для выполнения рисования. Другие параметры, нужные для работы функции задачи ThreadRoutine, извлекаются из структуры типа CHILD_WINDOW_TAG. В свою очередь, адрес этой структуры извлекается из памяти окна перед началом цикла рисования:

```
lpMyWndTag = (LPCHILD_WINDOW_TAG)GetWindowLong(hwnd,
  GWL_USERDATA);
```

Бесконечный цикл, в котором работает задача, прерывается в том случае, если признак активности задачи не равен единице:

```
if(!lpMyWndTag->fActive)
  break;
```

В цикле задача выполняет рисование эллипса случайной формы и расположения. Перед получением контекста отображения задача входит в критическую секцию, которая находится в структуре CHILD_WINDOW_TAG. После выполенния рисования задача освобождает контекст отображения, выходит из критической секции и выполняет небольшую задержку.

После прерывания цикла функция задачи удаляет ненужную более критическую секцию и освобождает память, заказанную для структуры CHILD_WINDOW_TAG функцией malloc:

```
DeleteCriticalSection(&(lpMyWndTag->csChildWindowPaint));
free(lpMyWndTag);
```

Затем задача завершает свое выполнение с помощью оператора return.

3 ПРОЦЕССЫ

Использование мультизадачности в рамках одного процесса позволяет относительно просто организовать параллельную работу. Однако необходимо помнить, что при этом все задачи работают в одном адресном пространстве, а значит, они не защищены друг от друга.

При необходимости организации параллельной обработки данных в отдельном адресном пространстве приложение может запустить отдельный процесс. Разумеется, процесс требует намного больше ресурсов, чем задача, и, кроме того, возникает проблема организации обмена данными. Так как дочерний процесс работает в своем адресном пространстве, родительский процесс не может использовать для передачи данных, например, глобальные переменные, - в адресном пространстве дочернего процесса они будут недоступны.

Тем не менее, передача данных между различными процессами возможна, например, с использованием динамической передачи данных DDE, файлов, отображаемых в память и так далее. В дальнейшем в одной из наших следующих книг мы подробно рассмотрим средства взаимодействия процессов, встроенные в операционную систему Microsoft Windows NT, в том числе процессов, запущенных на различных компьютерах в сети.

Предметом же этой главы будет изучение способов запуска процессов и управления процессами.

Запуск процесса

Пользователь запускает процесс при помощи приложений Program Manager и File Manager. Он может также воспользоваться командной строкой в системном приглашении консоли Microsoft Windows NT.

Что же касается приложения, то оно может выполнить эту задачу как при помощи уже известных вам из программирования для Microsoft Windows версии 3.1 функций WinExec и LoadModule, так и при помощи функции CreateProcess, специально предназначенной для запуска процессов. Заметим, что с помощью этой функции в середе Microsoft Windows NT версии 3.51 для платформы Intel вы можете запустить как 32-разрядные, так и 16-разрядные приложения Windows, а также программы MS-DOS и 16-разрядные консольные приложения OS/2 (напомним, что первые версии операционной системы OS/2 разрабатывались совместно фирмами Microsoft и IBM).

Параметры функции CreateProcess

Функция CreateProcess имеет много параметров, однако ей не так сложно пользоваться, как это может показаться на первый взгляд:

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,  // указатель на имя исполняемого
                              // модуля
  LPTSTR  lpCommandLine,      // указатель на командную строку
  LPSECURITY_ATTRIBUTES lpProcessAttributes, // указатель на
                //          атрибуты защиты процесса
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  // указатель на
                //          атрибуты защиты задачи
  BOOL bInheritHandles, // флаг наследования идентификатора
  DWORD dwCreationFlags,// флаги создания процесса
  LPVOID lpEnvironment, // указатель на блок среды выполнения
  LPCTSTR lpCurrentDirectory,  // указатель на имя текущего
                               // каталога
  LPSTARTUPINFO lpStartupInfo, // указатель на структуру
                               //  STARTUPINFO
  LPPROCESS_INFORMATION lpProcessInformation); // указатель на
                            // структуру PROCESS_INFORMATION
```

Если функция CreateProcess завершается успешно, она возвращает значение TRUE. В противном случае возвращается значение FALSE. Код ошибки вы можете получить, вызвав функцию GetLastError.

lpApplicationName

lpCommandLine

Параметры lpApplicationName и lpCommandLine используются для указания имя программного файла и параметров запуска процесса. Вы можете использовать один или оба этих параметра.

Через параметр lpApplicationName вы можете передать указатель на строку, закрытую двоичным нулем и содержащую полный либо частичный путь к программному файлу. При этом параметр lpCommandLine может иметь значение NULL либо указывать на строку параметров запуска приложения. Если это потребуется, приложение может извлечь адрес строки параметров через параметы функции WinMain, либо при помощи функции GetCommandLine, не имеющей параметров. Последняя возвращает адрес искомой строки.

Вы можете также передать через параметр lpApplicationName значение NULL, указав адрес строки пути к программному файлу через параметр lpCommandLine. В этом случае можно передать параметры процессу, добавив их в этой строке к пути файла через пробел. Таким образом, в параметре lpCommandLine можно указать адрес командной строки для запуска процесса.

lpProcessAttributes

lpThreadAttributes

Параметры lpProcessAttributes и lpThreadAttributes указывают атрибуты защиты, соответвтвенно, процесса и его главной задачи, которая получает управление при запуске процесса. В наших примерах мы будем указывать оба этих параметра как NULL, используя значения атрибутов защиты, принятые по умолчанию.

bInheritHandles

Флаг наследования идентификатора процесса и задачи bInheritHandles определяет, как это видно из названия, возможность использования данных идентификаторов в порожденных процессах и задачах. Если такая возможность вам необходима, укажите для этого параметра значение TRUE. В наших приложениях мы будем указывать значение FALSE, отменяющее наследование.

dwCreationFlags

Рассмотрим теперь параметр dwCreationFlags.

Этот параметр определяет способ, которым будет запущен процесс, а также класс приоритета процесса. Соответственно, для параметра dwCreationFlags можно использовать флаги создания процесса, объединяя их между собой оператором логического ИЛИ, а также с одним из значений, определяющих класс приоритета процесса.

Флаги создания процесса перечислены ниже:

- CREATE_SUSPENDED

Сразу после создания процесса его главная задача будет находиться в приостановленном состоянии. Работу этой задачи можно возобновить при помощи функции ResumeThread. Этот флаг может быть использован при отладке процесса.

- DEBUG_PROCESS

Флаг DEBUG_PROCESS используется отладчиками, которые создают процесс для отладки. Если указан этот флаг, родительский процесс (то есть отладчик) информируется о различных событиях, возникающих в отлаживаемом процессе.

- DEBUG_ONLY_THIS_PROCESS

Аналогично предыдущему, однако отладчик извещается о тех событиях, которые происходят только в отлаживаемом процессе, но не в процессах, запущенных отлаживаемым процессом.

- CREATE_UNICODE_ENVIRONMENT

Этот флаг используется в том случае, если для блока среды процесса, адрес которого передается через параметр lpEnvironment, используется кодировка Unicode. В противном случае предполагается, что для блока среды используются символы в коде ANSI.

Рассмотрение кодировки Unicode выходит за рамки этой книги, однако мы, возможно, расскажем вам о ней в одной из следующих наших книг, посвященных операционной системе Microsoft Windows NT.

- CREATE_NEW_CONSOLE

Используется для консольных процессов. Если указан флаг CREATE_NEW_CONSOLE, для нового процесса создается новая консоль. Консольные процессы мы не будем пока рассматривать для экономии места в книге. Этот флаг несовместим с флагом DETACHED_PROCESS.

- DETACHED_PROCESS

Используется для консольных процессов. Если указан этот флаг, новый консольный процесс не имеет доступа к родительской консоли. При необходимости он может создать новую консоль. Этот флаг несовместим с флагом CREATE_NEW_CONSOLE.

- CREATE_NEW_PROCESS_GROUP

Используется для консольных процессов. Новый процесс будет корневым для группы процессов.

- CREATE_SEPARATE_WOW_VDM

Используется для запуска 16-разрядных приложоений Microsoft Windows. Если установлен флаг CREATE_SEPARATE_WOW_VDM, для работы приложения создается отдельная виртуальная машина DOS. Если произойдет ошибка в этом приложении, то она не скажется на работе остальных 16-разрядных приложений Microsoft Windows, работающих на других виртуальных машинах (так как последние находятся в другом адресном пространстве).

- CREATE_DEFAULT_ERROR_MODE

Новый процесс не наследует режим обработки ошибок, установленный родительским процессом при помощи функции SetErrorMode, и должен устанавливать этот режим самостоятельно.

- REALTIME_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS

- IDLE_PRIORITY_CLASS

Приведенные выше четыре флага указывают класс приоритета нового процесса. Обычно вы должны использовать значение NORMAL_PRIORITY_CLASS.

lpEnvironment

Наряду с регистрационной базой данных, в которой приложения могут хранить необходимые им для выполнения программы, операционная система Microsoft Windows NT сохранила такой атавизм MS-DOS, как среда выполнения программ (очевидно, сохранила для обратной совместимости с этой операционной системой). В MS-DOS переменные среды устанавливались при помощи команд в файле autoexec.bat. Операционная система Microsoft Windows NT версии 3.51 позволяет это сделать через приложение Control Panel.

Если дочерний процесс должен использовать родительский блок среды выполнения, через параметр lpEnvironment необходимо передать значение NULL.

Родительский процесс может также подготовить новый блок среды, состоящий из расположенных друг за другом символьных строк, закрытых двоичным нулем. Последняя из этих строк должна быть закрыта двумя двоичными нулями.

При необходимости процесс может получить адрес блока среды при помощи функции GetEnvironmentStrings, не имеющей параметров. Функция GetEnvironmentVariable позовляет узнать значение отдельных переменных блока среды. Описание этой функции вы найдете в SDK.

lpCurrentDirectory

Параметр lpCurrentDirectory задает путь к символьной строке, закрытой двоичным нулем, содержащей путь к каталогу, который будет использован как рабочий при запуске процесса. Если указать этот параметр как NULL, в качестве рабочего для нового процесса будет использован рабочий каталог родительского процесса.

lpStartupInfo

Через параметр lpStartupInfo вы должны передать функции CreateProcess указатель на структуру типа STARTUPINFO, определяющую внешний вид окна, создаваемого для процесса:

```
typedef struct _STARTUPINFO
{
 DWORD  cb;            // размер структуры в байтах
 LPTSTR lpReserved;    // зарезервировано
 LPTSTR lpDesktop;     // рабочий стол и станция для процесса
 LPTSTR lpTitle;       // заголовок окна консольного процесса
 DWORD  dwX;           // координата угла окна в пикселах
 DWORD  dwY;           // координата угла окна в пикселах
 DWORD  dwXSize;       // ширина окна в пикселах
 DWORD  dwYSize;       // высота окна в пикселах
 DWORD  dwXCountChars; // ширина консольного окна
 DWORD  dwYCountChars; // высота консольного окна
 DWORD  dwFillAttribute; // атрибуты текста консольного окна
 DWORD  dwFlags;       // заполненные поля структуры
 WORD   wShowWindow;   // размеры окна по умолчанию
 WORD   cbReserved2;   // зарезервировано
 LPBYTE lpReserved2;   // зарезервировано
 HANDLE hStdInput;     // консольный буфер ввода
 HANDLE hStdOutput;    // консольный буфер вывода
 HANDLE hStdError; // консольный буфер вывода сообщений
                   // об ошибках
} STARTUPINFO, *LPSTARTUPINFO;
```

Несмотря на внушительный размер этой структуры, ее заполнение не вызовет у вас особых трудностей, так как большинство полей можно не использовать.

cb

Поле cb должно содержать размер структуры STARTUPINFO в байтах.

dwFlags

Поле dwFlags содержит флаги, определяющие, содержимое каких полей структуры STARTUPINFO необходимо учитывать при запуске нового процесса, а также три дополнительных флага:

Значение	*Используемые поля*
STARTF_USESHOWWINDOW	wShowWindow
STARTF_USEPOSITION	dwX, dwY
STARTF_USESIZE	dwXSize, dwYSize
STARTF_USECOUNTCHARS	dwXCountChars, dwYCountChars
STARTF_USEFILLATTRIBUTE	dwFillAttribute
STARTF_USESTDHANDLES	hStdInput, hStdOutput и hStdError

Дополнительными являются флаги STARTF_FORCEONFEEDBACK, STARTF_FORCEOFFFEEDBACK и STARTF_SCREENSAVER.

Использование первого из этих флагов приводит к тому что при запуске приложения курсор мыши принимает форму песочных часов со стрелкой. При этом пользователь может работать с другими приложениями, зная, что процесс запуска данного приложения еще не завершен. Как только приложение будет запущено и станет активным, курсор примет свою обычную форму.

Если же указан флаг STARTF_FORCEOFFFEEDBACK, в процессе запуска приложения курсор имеет обычную форму.

Флаг STARTF_SCREENSAVER используется для создания приложений, предохраняющих экран монитора от преждеверменного выгорания. Этот флаг вызывает снижение класса приоритета после запуска до IDLE_PRIORITY_CLASS, несмотря на то что при инициализации класс приоритета был NORMAL_PRIORITY_CLASS. Если пользователь активизирует окно такого приложения, его класс приоритета автоматически повышается.

lpDesktop

Поле lpDesktop используется для выбора рабочего стола и рабочей станции, где будет запущен процесс. Если вы собираетесь использовать текущий рабочий стол и текущую рабочую станцию, укажите в этом поле значение NULL.

lpTitle

Для консольного процесса (и только для него) вы можете указать заголовок окна в поле lpTitle. Если же в этом поле задать значение NULL, для заголовка будет использовано имя файла.

dwX

dwY

Поля dwX и dwY определяют, соответственно, координаты X и Y левого верхнего угла окна графического приложения в пикселах. Эти значения используются для позиционирования главного окна приложения только в том случае, если при создании окна функцией CreateWindow расположение окна было указано как CW_USEDEFAULT.

dwXSize

dwYSize

Аналогично, поля dwXSize и dwYSize определяют, соответственно, ширину и высоту окна графического приложения в пикселах. Эти значения используются в том случае, если при создании окна функцией CreateWindow размыры окна были указаны как CW_USEDEFAULT.

dwXCountChars

dwYCountChars

Поля dwXCountChars и dwYCountChars задают, соответственно, ширину и высоту окна консольного приложения в символах.

dwFillAttribute

Содержимое поля dwFillAttribute задает цвет текста и фона для окна консольного приложнения.

wShowWindow

Поле wShowWindow определяет значение по умолчанию, которое будет использовано при первом вызове функции ShowWindow для главного окна приложения.

Остановимся на этом подробнее.

Как вы, наверное, помните, в приложениях Microsoft Windows версии 3.1 функция WinMain получала параметр nCmdShow, определяющий, в каком виде должно отображаться окно - в нормальном, минимизированном, максимизированном и так далее.

В среде Microsoft Windows NT параметр nCmdShow функции WinMain всегда имеет значение SW_SHOWDEFAULT. В этом случае для определения внешнего вида главного окна прилоджения используется содержимое поля wShowWindow структуры STARTUPINFO. Здесь, а также в качестве параметров функции ShowWindow, вы можете использовать следующие значения:

Значение	*Внешний вид окна приложения*
SW_MINIMIZE	Минимизировано
SW_MAXIMIZE	Максимизировано
SW_RESTORE	Восстановлено в исходное состояние (это значение используется при восстановлении размеров минимизированного ранее окна)
SW_HIDE	Скрыто
SW_SHOW	Отображается с использованием текущих размеров и расположения
SW_SHOWDEFAULT	Отображается с использованием размеров и

	расположения, заданных в структуре STARTUPINFO при создании процесса функцией CreateProcess
SW_SHOWMAXIMIZED	Окно активизируется и отображается в максимизированном виде
SW_SHOWMINIMIZED	Окно активизируется и отображается в минимизированном виде
SW_SHOWMINNOACTIVE	Минимизируется, но не становится активным
SW_SHOWNA	Окно отображается в текущем виде, но не активизируется
SW_SHOWNOACTIVATE	Устанавливаются размеры и расположение окна, которые оно только что имело. Активизация окна не выполняется
SW_SHOWNORMAL	Окно активизируется и отображается. Минимизированное окно восстанавливается

Вернемся к полям структуры STARTUPINFO.

hStdInput

hStdOutput

hStdError

Поля hStdInput, hStdOutput и hStdError определяют идентификаторы буферов, которые используются консольными приложениями, соответственно, для ввода, вывода обычной информации и вывода сообщений об ошибках.

lpProcessInformation

Перед вызовом функции CreateProcess вы должны передать ей через параметр lpProcessInformation адрес структуры типа PROCESS_INFORMATION, в которую будут записаны идентификаторы и системные номера созданного процесса и его главной задачи:

```
typedef struct _PROCESS_INFORMATION
{
  HANDLE hProcess;    // идентификатор процесса
  HANDLE hThread;     // идентификатор главной задачи процесса
  DWORD  dwProcessId; // системный номер процесса
  DWORD  dwThreadId;  // системный номер главной задачи
                      // процесса
} PROCESS_INFORMATION;
```

Пользуясь идентификаторами, полученными из полей hProcess и hThread, родительский процесс может управлять порожденным процессом и его главной задачей, например, изменяя класс приоритета процесса или относительный приоритет его главной задачи.

Важно отметить, что после использования родительский процесс обязан закрыть полученные идентификаторы порожденного процесса и задачи при помощи функции CloseHandle.

Завершение процесса

Для завершения процесса используются функции ExitProcess и TerminateProcess. Первая из этих функций нужна в том случае, если процесс сам желает завершить свою работу. Именно эта функция вызывается библиотекой времени выполнения при возвращении управления функцией WinMain. Функция TerminateProcess используется родительским процессом для завершения своего дочернего процесса или любого другого процесса, идентификатор которого ей известен (и к которому имеется соотвтетствующий доступ).

Функция ExitProcess имеет один параметр - код завершения процесса:

```
VOID ExitProcess(UINT uExitCode);
```

Этот код после завершения работы процесса родительский процесс может определить при помощи функции GetExitCodeProcess.

Функции TerminateProcess необходимо передать два параметра - идентификатор завершаемого процесса и код завершения процесса:

```
BOOL TerminateProcess(
  HANDLE hProcess,   // идентификатор завершаемого процесса
  UINT   uExitCode); // код завершения процесса
```

Независимо от способа, при завершении процесса закрываются идентификаторы всех объектов, созданных задачами процесса, а все задачи процесса завершают свое выполнение.

Приложение PSTART

С помощью приложения PSTART, исходные тексты которого мы привели в этом разделе, вы сможете запускать процессы, определяя для них класс приоритета.

Главное окно приложения PSTART показано на рис. 3.1.

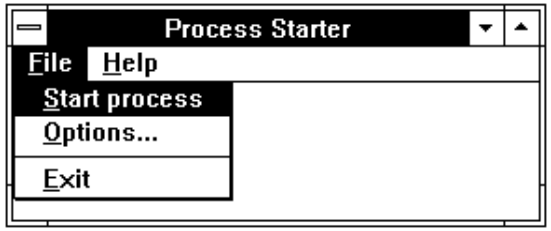

Рис. 3.1. Главное окно приложения PSTART

Выбрав из меню File строку Start process, вы можете выбрать в диалоговой панели Start Process программный файл запускаемого приложения (рис. 3.2).

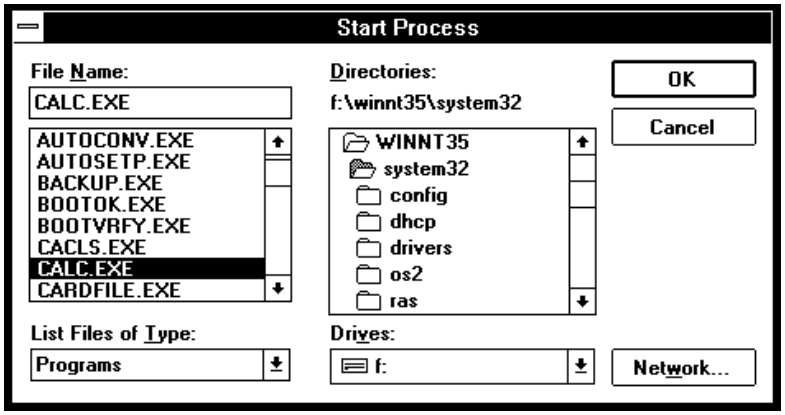

Рис. 3.2. Диалоговая панель Start Process

Для выбора параметров запуска воспользуйтесь диалоговой панелью Start Options (рис. 3.3), которая появится на экране при выборе из меню File строки Options.

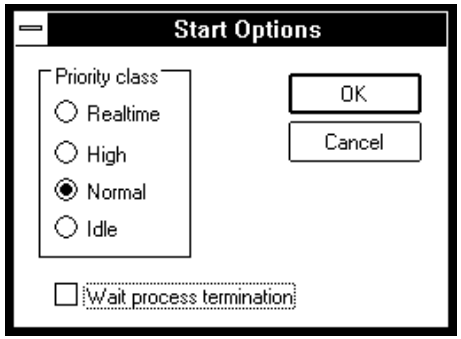

Рис. 3.3. Диалоговая панель Start Options

В этой диалоговой панели вы можете выбрать класс приоритета запускаемого процесса, а также, включив переключатель Wait process termination, использовать режим запуска, при котором запускающий процесс дожидается завершения выполнения дочернего процесса.

Исходные тексты приложения

Главный файл исходных текстов приложения PSTART представлены в листинге 3.1.

Листинг 3.1. Файл pstart/pstart.c

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <commctrl.h>
#include <stdio.h>
#include <memory.h>
#include "resource.h"
#include "afxres.h"
#include "pstart.h"

HINSTANCE hInst;
char szAppName[]  = "PStartApp";
char szAppTitle[] = "Process Starter";

// Параметры запуска процессов
```

```
DWORD dwCreationFlags;

// Флаг ожидания завершения процессов
BOOL  fWaitTermination;

// -------------------------------------------------------
// Функция WinMain
// -------------------------------------------------------
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
  WNDCLASSEX wc;
  HWND hWnd;
  MSG msg;

  // Сохраняем идентификатор приложения
  hInst = hInstance;

  // Преверяем, не было ли это приложение запущено ранее
  hWnd = FindWindow(szAppName, NULL);
  if(hWnd)
  {
    // Если было, выдвигаем окно приложения на
    // передний план
    if(IsIconic(hWnd))
      ShowWindow(hWnd, SW_RESTORE);
    SetForegroundWindow(hWnd);
    return FALSE;
  }

  // Регистрируем класс окна
  memset(&wc, 0, sizeof(wc));
  wc.cbSize = sizeof(WNDCLASSEX);
  wc.hIconSm = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICONSM),
    IMAGE_ICON, 16, 16, 0);
  wc.style = 0;
  wc.lpfnWndProc = (WNDPROC)WndProc;
  wc.cbClsExtra  = 0;
  wc.cbWndExtra  = 0;
  wc.hInstance = hInst;
  wc.hIcon = LoadImage(hInst,
    MAKEINTRESOURCE(IDI_APPICON),
    IMAGE_ICON, 32, 32, 0);
  wc.hCursor = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
  wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
  wc.lpszClassName = szAppName;
  if(!RegisterClassEx(&wc))
    if(!RegisterClass((LPWNDCLASS)&wc.style))
      return FALSE;

  // Создаем главное окно приложения
  hWnd = CreateWindow(szAppName, szAppTitle,
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
    NULL, NULL, hInst, NULL);
  if(!hWnd) return(FALSE);

  // Отображаем окно и запускаем цикл
  // обработки сообщений
  ShowWindow(hWnd, nCmdShow);
  UpdateWindow(hWnd);
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return msg.wParam;
}
```

```
// -------------------------------------------------------
// Функция WndProc
// -------------------------------------------------------
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
  switch(msg)
  {
    HANDLE_MSG(hWnd, WM_COMMAND, WndProc_OnCommand);
    default:
      return(DefWindowProc(hWnd, msg, wParam, lParam));
  }
}

// -------------------------------------------------------
// Функция WndProc_OnCommand
// -------------------------------------------------------
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id,
  HWND hwndCtl, UINT codeNotify)
{
  switch (id)
  {
    // Отображаем диалоговую панель для выбора
    // программного файла и запускаем процесс
    case ID_FILE_STARTPROCESS:
    {
      StartProcess(hWnd);
      break;
    }

    case ID_FILE_OPTIONS:
    {
      // Отображаем диалоговую панель, предназначенную
      // для настройки параметров запуска процессов
      DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1),
        hWnd, DlgProc);
      break;
    }

    case ID_FILE_EXIT:
    {
      // Завершаем работу приложения
      PostQuitMessage(0);
      return 0L;
      break;
    }

    case ID_HELP_ABOUT:
    {
      MessageBox(hWnd,
        "Process Starter\n"
        "(C) Alexandr Frolov, 1996\n"
        "Email: frolov@glas.apc.org",
        szAppTitle, MB_OK | MB_ICONINFORMATION);
      return 0L;
      break;
    }

    default:
      break;
  }
  return FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify,
    DefWindowProc);
}

// -------------------------------------------------------
// Функция StartProcess
// -------------------------------------------------------
void StartProcess(HWND hwnd)
{
```

```
OPENFILENAME ofn;
STARTUPINFO si;
PROCESS_INFORMATION pi;
char szBuf[256];
DWORD dwExitCode;

char szFile[256];
char szDirName[256];
char szFileTitle[256];
char szFilter[256] = "Programs\0*.exe\0Any Files\0*.*\0";
char szDlgTitle[] = "Start Process";


memset(&si, 0, sizeof(STARTUPINFO));
si.cb = sizeof(si);

memset(&ofn, 0, sizeof(OPENFILENAME));
GetCurrentDirectory(sizeof(szDirName), szDirName);
szFile[0] = '\0';

// Подготавливаем структуру для выбора файла
ofn.lStructSize     = sizeof(OPENFILENAME);
ofn.hwndOwner       = hwnd;
ofn.lpstrFilter     = szFilter;
ofn.lpstrInitialDir = szDirName;
ofn.nFilterIndex    = 1;
ofn.lpstrFile       = szFile;
ofn.nMaxFile        = sizeof(szFile);
ofn.lpstrFileTitle  = szFileTitle;
ofn.nMaxFileTitle   = sizeof(szFileTitle);
ofn.lpstrTitle      = szDlgTitle;
ofn.Flags = OFN_PATHMUSTEXIST | OFN_FILEMUSTEXIST
  | OFN_HIDEREADONLY;

// Выводим на экран диалоговую панель, предназначенную
// для выбора файла
if(GetOpenFileName(&ofn))
{
  // Если файл выбран, запускаем его на выполнение
  if (*ofn.lpstrFile)
  {
    // Создаем процесс
    if(CreateProcess(NULL, ofn.lpstrFile, NULL, NULL,
      FALSE, dwCreationFlags, NULL, NULL, &si, &pi))
    {
      // Если было указано, что нужно ждать завершение
      // процесса, выполняем такое ожидание
      if(fWaitTermination)
      {
        // Освобождаем идентификатор задачи запущенного
        // процесса, так как он нам не нужен
        CloseHandle(pi.hThread);

        // Выполняем ожидание завершения процесса
        if(WaitForSingleObject(pi.hProcess,
          INFINITE) != WAIT_FAILED)
        {
          // Получаем и отображаем код завершения процесса
          GetExitCodeProcess(pi.hProcess, &dwExitCode);

          sprintf(szBuf, "Process terminated\n"
            "Exit Code = %lX", dwExitCode);

          MessageBox(hwnd, szBuf,
            szAppTitle, MB_OK | MB_ICONINFORMATION);
        }

        // Освобождаем идентификатор процесса
        CloseHandle(pi.hProcess);
      }

      // Если процесс был запущен без ожидания,
```

```
      // освобождаем идентификаторы задачи и процесса
      else
      {
        CloseHandle(pi.hProcess);
        CloseHandle(pi.hThread);
      }
    }

    // Если про запуске процесса произошла ошибка,
    // получаем и отображаем ее код
    else
    {
      sprintf(szBuf, "CreateProcess: error %ld",
        GetLastError());

      MessageBox(hwnd, szBuf,
        szAppTitle, MB_OK | MB_ICONEXCLAMATION);
    }
   }
  }
}

// -------------------------------------------------------
// Функция DlgProc
// -------------------------------------------------------
LRESULT WINAPI
DlgProc(HWND hdlg, UINT msg, WPARAM wParam,
      LPARAM lParam)
{
 switch(msg)
 {
   HANDLE_MSG(hdlg, WM_INITDIALOG, DlgProc_OnInitDialog);
   HANDLE_MSG(hdlg, WM_COMMAND,    DlgProc_OnCommand);

   default:
    return FALSE;
  }
}

// -------------------------------------------------------
// Функция DlgProc_OnInitDialog
// -------------------------------------------------------

BOOL DlgProc_OnInitDialog(HWND hdlg, HWND hwndFocus,
                  LPARAM lParam)
{
 // По умолчанию мы будем запускать процессы
 // с нормальным приоритетом без ожидания
 // их завершения
 CheckDlgButton(hdlg, IDC_NORMAL, 1);
 return TRUE;
}

// -------------------------------------------------------
// Функция DlgProc_OnCommand
// -------------------------------------------------------
#pragma warning(disable: 4098)
void DlgProc_OnCommand(HWND hdlg, int id,
 HWND hwndCtl, UINT codeNotify)
{
 switch (id)
 {
  case IDOK:
  {
   dwCreationFlags = 0;
   fWaitTermination = FALSE;

   // Устанавливаем класс приоритета
   if(IsDlgButtonChecked(hdlg, IDC_Realtime))
   {
    dwCreationFlags = REALTIME_PRIORITY_CLASS;
   }
```

```
    else if(IsDlgButtonChecked(hdlg, IDC_HIGH))
    {
      dwCreationFlags = HIGH_PRIORITY_CLASS;
    }
    else if(IsDlgButtonChecked(hdlg, IDC_NORMAL))
    {
      dwCreationFlags = NORMAL_PRIORITY_CLASS;
    }
    else if(IsDlgButtonChecked(hdlg, IDC_IDLE))
    {
      dwCreationFlags = IDLE_PRIORITY_CLASS;
    }

    // Проверяем и устанавливаем режим ожидания
    if(IsDlgButtonChecked(hdlg, IDC_TERMINATION))
    {
      fWaitTermination = TRUE;
    }

    EndDialog(hdlg, 0);
    return TRUE;
   }

   case IDCANCEL:
   {
    EndDialog(hdlg, 0);
    return TRUE;
   }

   default:
     break;
  }
  return FALSE;
}
```

Файл pstart.h (листинг 3.2) содержит прототипы функций, определенных в приложении PSTART.

Листинг 3.2. Файл pstart/pstart.h

```
// ------------------------------------------------------
// Описание функций
// ------------------------------------------------------
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
void WndProc_OnCommand(HWND hWnd, int id,
  HWND hwndCtl, UINT codeNotify);
void StartProcess(HWND hwnd);

LRESULT WINAPI
DlgProc(HWND hdlg, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL DlgProc_OnInitDialog(HWND hwnd, HWND hwndFocus,
                LPARAM lParam);
void DlgProc_OnCommand(HWND hdlg, int id,
  HWND hwndCtl, UINT codeNotify);
```

Файл resource.h (листинг 3.3) создается автоматически и содержит определения констант для файла описания ресурсов приложения PSTART.

Листинг 3.3. Файл pstart/resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by PStart.rc
//
#define IDR_MENU1                   102
#define IDR_APPMENU                 102
#define IDI_APPICON                 103
#define IDI_APPICONSM               104
#define IDD_DIALOG1                 105
#define IDC_Realtime                1004
#define IDC_HIGH                    1005
#define IDC_NORMAL                  1006
#define IDC_IDLE                    1007
```

```
#define IDC_TERMINATION             1008
#define IDC_DETACHED                1009
#define ID_FILE_EXIT                40001
#define ID_HELP_ABOUT               40002
#define ID_FILE_STARTPROCESS        40003
#define ID_FILE_OPTIONS             40004

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE    106
#define _APS_NEXT_COMMAND_VALUE     40005
#define _APS_NEXT_CONTROL_VALUE     1010
#define _APS_NEXT_SYMED_VALUE       101
#endif
#endif
```

В файле определения ресурсов приложения PSTART (листинг 3.4), который создается автоматически системой разработки Microsoft Visual C++, определены главное меню приложения, диалоговая панель для изменения параметров запуска процессов, пиктограммы и текстовые строки.

Листинг 3.4. Файл pstart/pstart.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"

////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
   "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
   "#include ""afxres.h""\r\n"
   "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
   "\r\n"
   "\0"
END
#endif    // APSTUDIO_INVOKED

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

```
// Menu
//

IDR_APPMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "&Start process",          ID_FILE_STARTPROCESS
        MENUITEM "&Options...",             ID_FILE_OPTIONS
        MENUITEM SEPARATOR
        MENUITEM "&Exit",                   ID_FILE_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About",                  ID_HELP_ABOUT
    END
END

/////////////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure application icon
// remains consistent on all systems.
IDI_APPICON             ICON    DISCARDABLE     "PSTART.ICO"
IDI_APPICONSM           ICON    DISCARDABLE     "PSTARTSM.ICO"

/////////////////////////////////////////////////////////////////////
//
// Dialog
//

IDD_DIALOG1 DIALOG DISCARDABLE  0, 0, 163, 97
STYLE DS_MODALFRAME | WS_POPUP | WS_VISIBLE | WS_CAPTION | WS_SYSMENU
CAPTION "Start Options"
FONT 8, "MS Sans Serif"
BEGIN
    DEFPUSHBUTTON   "OK",IDOK,103,9,50,14
    PUSHBUTTON      "Cancel",IDCANCEL,103,26,50,14
    GROUPBOX        "Priority class",IDC_STATIC,9,4,57,69
    CONTROL         "Realtime",IDC_Realtime,"Button",
                    BS_AUTORADIOBUTTON | WS_GROUP,15,18,43,10
    CONTROL         "High",IDC_HIGH,"Button",
                    BS_AUTORADIOBUTTON,15,32,31,10
    CONTROL         "Normal",IDC_NORMAL,"Button",
                    BS_AUTORADIOBUTTON,15,45,38,10
    CONTROL         "Idle",IDC_IDLE,"Button",
                    BS_AUTORADIOBUTTON,15,58,27,10
    CONTROL         "Wait process termination",
                    IDC_TERMINATION,"Button",
                    BS_AUTOCHECKBOX | WS_TABSTOP,15,81,93,10
END

/////////////////////////////////////////////////////////////////////
//
// DESIGNINFO
//

#ifdef APSTUDIO_INVOKED
GUIDELINES DESIGNINFO DISCARDABLE
BEGIN
    IDD_DIALOG1, DIALOG
    BEGIN
        LEFTMARGIN, 7
        RIGHTMARGIN, 156
        TOPMARGIN, 7
        BOTTOMMARGIN, 90
    END
END
#endif    // APSTUDIO_INVOKED
```

```
#endif    // English (U.S.) resources
/////////////////////////////////////////////////////////////


#ifndef APSTUDIO_INVOKED
/////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
/////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Определения и глобальные переменные

Для работы со стандартной диалоговой панелью, с помощью которой выбирается запускаемый программный файл, в исходный текст приложения включается файл commctrl.h.

В области глобальных переменных нашего приложения определены переменные dwCreationFlags и fWaitTermination. Первая из них содержит флаги создания процессов, которые настраиваются при помощи диалоговой панели Start Options и используются при запуске процессов функцией CreateProcess. Во второй переменной хранится признак необходимости ожидания завершения процессов. Содержимое этой переменной изменяется также при помощи диалоговой панели Start Options.

Описание функций

Приведем краткое описание наиболее важных функций, определенных в нашем приложении.

Функция WinMain

Функция WinMain выполняет обычные действия, создавая главное окно приложения и запуская цикл обработки сообщений.

Функция WndProc

В задачу функции WndProc входит обработка сообщения WM_COMMAND, которая выполняется с помощью функции WndProc_OnCommand. Все остальные сообщения передаются функции DefWindowProc.

Функция WndProc_OnCommand

Эта функция обрабатывает сообщение WM_COMMAND, поступающее в главное окно приложения от меню. Когда пользователь выбирает из меню File строку Start process, функция WndProc_OnCommand вызывает функцию StartProcess, определенную в нашем приложении. Последняя отображает на экране стандартную панель выбора программного файла и в случае успешного выбора запускает этот файл на выполнение как отдельный процесс. При этом используются параметры запуска, установленные при помощи диалоговой панели Start Options.

В том случае когда пользователь выбирает из меню File строку Options, на экране отображается модальная диалоговая панель Start Options, имеющая идентификатор MAKEINTRESOURCE(IDD_DIALOG1).

Отображение диалоговой панели выполняется с помощью функции DialogBox:

```
DialogBox(hInst, MAKEINTRESOURCE(IDD_DIALOG1), hWnd, DlgProc);
```

Эта функция, а также все, что относится к диалоговым панелям, мы описали в главе “Диалоговые панели” 12 тома “Библиотеки системного программиста”.

Заметим, что в среде 32-разрядных операционных систем Microsoft Windows 95 и Microsoft Windows NT в качестве последнего параметра функции DialogBox можно указывать имя функции диалога. При этом вам не требуется создавать переходник диалоговой функции, вызывая функцию MakeProcInstance. Функция MakeProcInstance не используется 32-разрядными приложениями.

Функция StartProcess

Функция StartProcess выполняет запуск программного файла, выбранного пользователем.

Для выбора программного файла используется функция GetOpenFileName, которую мы подробно описали в разделе “Стандартные диалоговые панели для открытия файлов” 13 тома “Библиотеки системного программиста”. Путь к выбранному файлу записывается в поле lpstrFile структуры ofn.

Запуск процесса выполняется функцией CreateProcess, которая вызывается следующим образом:

```
if(CreateProcess(NULL, ofn.lpstrFile, NULL, NULL,
     FALSE, dwCreationFlags, NULL, NULL, &si, &pi))
{
  . . .
}
```

Первый параметр функции CreateProcess указан как NULL, поэтому адрес символьной строки, содержащей путь к программному файлу, передается через второй параметр. Напомним, что с помощью этого параметра можно передать функции CreateProcess командную строку (с параметрами) для запуска приложения.

Третий и четвертый параметры функции CreateProcess, задающие атрибуты защиты процесса и его главной задачи, указаны как NULL. В

результате используются значения атрибутов защиты, принятые по умолчанию.

Пятый параметр, определяющий возможность наследования идентификаторов процесса и главной задачи указан как FALSE, поэтому наследование не допускается.

Через шестой параметр функции CreateProcess передаются флаги создания процесса. Здесь мы используем глобальную переменную dwCreationFlags, содержимое которой устанавливается функцией диалога диалоговой панели Create Options. Эта функция будет описана ниже.

Седьмой и восьмой параметры функции CreateProcess, задают, соответственно, адрес блока среды и путь к рабочему каталогу. Так как оба этих параметра указаны как NULL, создаваемый дочерний процесс пользуется родительской средой и родительским рабочим каталогом.

Через девятый параметр функции CreateProcess передается адрес структуры si типа STARTUPINFO. Ни одно из полей этой структуры, кроме поля cb, не используется, поэтому инициализация структуры выполняется очень просто:

```
memset(&si, 0, sizeof(STARTUPINFO));
si.cb = sizeof(si);
```

И, наконец, через последний, десятый параметр функции CreateProcess передается адрес структуры pi типа PROCESS_INFORMATION. Напомним, что после удачного запуска процесса в эту структуру записываются идентификаторы, а также системные номера процесса и его главной задачи.

В случае успешного запуска процесса функция StartProcess проверяет необходимость ожидания его завершения. По умолчанию ожидание не выполняется, однако если пользователь включил в диалоговой панели Start Options переключатель Wait process termination, функция этой диалоговой панели записывает в глобальную переменную fWaitTermination значение TRUE. При этом функция StartProcess будет ожидать завершение запущенного процесса.

В режиме ожидания функция StartProcess вначале закрывает ненужный ей больше идентификатор главной задачи процесса pi.hThread, а затем вызывает функцию WaitForSingleObject, передавая ей в качестве первого параметра идентификатор процесса pi.hProcess, завершение которого необходимо дождаться. Второй параметр задает ожилание в течении неограниченного времени:

```
if(WaitForSingleObject(pi.hProcess, INFINITE) != WAIT_FAILED)
{
 . . .
}
```

Функция WaitForSingleObject будет описана подробно в главе, посвященной синхронизации задач и процессов. Сейчас отметим только, что эта функция переводит главную задачу приложения PSTART в состояние ожидания, когда ей не выделяются кванты процессорного времени. Такое ожидание не снижает производительность работы системы.

После того как запущенный процесс завершит свою работу, функция WaitForSingleObject вернет управление вызвавшей ее функции StartProcess. Вслед за этим приложение получит и отобразит код завершения процесса.

Для получения кода завершения процесса мы воспользовались функцией GetExitCodeProcess:

```
GetExitCodeProcess(pi.hProcess, &dwExitCode);
```

Через первый параметр этой функции передается идентификатор завершившегося процесса, а через второй - адрес переменной типа DWORD, в которую будет записан код завершения.

После отображения кода завершения идентификатор процесса освобождается функцией CloseHandle.

В том случае, когда процесс был запущен без ожидания его завершения, сразу после запуска функция StartProcess закрывает идентификаторы процесса и его главной задачи:

```
CloseHandle(pi.hProcess);
CloseHandle(pi.hThread);
```

При выполнении функции CreateProcess возможно возникновение ошибок. Например, вы можете выбрать не программный, а текстовый файл и попытаться запустить его на выполнение. Возможно, для запуска процесса не хватит виртуальной памяти или других ресурсов. В этом случае приложение PSTART с помощью функции GetLastError получает код ошибки и отображает его на экране.

Функция DlgProc

Функция диалога DlgProc обарбатывает сообщения WM_INITDIALOG и WM_COMMAND, поступающие от диалоговой панели Start Options. Для обработки этих сообщений она вызывает, соответственно, функции DlgProc_OnInitDialog и DlgProc_OnCommand.

Функция DlgProc_OnInitDialog

Эта функция обрфабатывает сообщение WM_INITDIALOG, поступающее в функцию диалога при инициализации последнего. Задачей обработчика сообщения WM_INITDIALOG является установка органов управления, расположенных в диалоговой панели, в начальное состояние. Мы устанавливаем во включенное состояние переключатель Normal, который определяет нормальный класс приоритета для запускаемых процессов.

Установка переключателей выполняется макрокомандой CheckDlgButton, подробно описанной в 12 томе “Библиотеки системного программиста”:

```
CheckDlgButton(hdlg, IDC_NORMAL, 1);
```

Функция DlgProc_OnCommand

Задачей функции DlgProc_OnCommand является обработка сообщения WM_COMMAND, поступающего в функцию диалогоа от органов управления, расположенных в диалоговой панели Start Options.

Когда пользователь нажимает кнопку OK, функция DlgProc_OnCommand определяет текущее состояние органов управления и устанавливает соответствующим образом содержимое двух глобальных переменных с именами dwCreationFlags и fWaitTermination.

В переменную dwCreationFlags записывается выбранный класс приоритета запускаемого процесса. Состояние переключателя с зависимой фиксацией, отвечающего за выбор класса приоритета, определяется с помощью макрокоманды IsDlgButtonChecked.

Что же касается глобальной переменной fWaitTermination, то ее значение устанавливается в соответствии с сотоянием переключателя с независимой фиксацией Wait process termination.

После того как содержимое глобальных переменных будет установлено, диалоговая панель удаляется функцией EndDialog.

Если пользователь отменяет работу с диалоговой панели, то происходит удаление последней без изменения содержимого глобальных переменных dwCreationFlags и fWaitTermination.

4 СИНХРОНИЗАЦИЯ ЗАДАЧ И ПРОЦЕССОВ

Если вы создаете мультизадачное приложение или приложение, запускающее процессы, вам необходимо уделить много внимания обеспечению синхронизации создаваемых задач и процессов. В противном случае могут появиться ошибки, которые трудно поддаются локализации. Плохо спроектированное и плохо отлаженное мультизадачное приложение может по-разному вести себя в однопроцессорных и мультипроцессорных конфигурациях компьютера, поэтому крайне желательно выполнять отладку мультизадачных приложений на мультипроцессорных конфигурациях.

В чем заключается синхронизация задач и приложений?

Если сказать кратко, то синхронизация задач (в том числе принадлежащих разным приложениям) заключается в том, что некоторые задачи должны приостанавливать свое выполнение до тех пор, пока не произойдут те или иные события, связанные с другими задачами.

Здесь возможно очень много вариантов. Например, главная задача создает задачу, которая выполняет какую-либо длительную работу, например, подсчет страниц в текстовом документе. Если теперь потребуется отобразить количество страниц на экране, главная задача должна дождаться завершения работы, выполняемой созданной задачей.

Другая проблема возникает, если, например, вы собираетесь организовать параллельную работу различных задач с одними и теми же данными. Особенно сложно выполнить необходимую синхронизацию в том случае, когда некоторые из этих задач выполняют чтение общих данных, а некоторые - изменение. Типичный пример - текстовый процессор, в котором одна задача может заниматься редактированием документа, другая - подсчетом страниц, третья - печатью документа на принтере и так далее.

Синхронизация необходима и в том случае, когда вы, например, создаете ядро системы управления базой данных, допускающее одновременную работу нескольких пользователей. При этом вам, вероятно, потребуется выполнить синхронизацию задач, принимающих запросы от пользователей, и выполняющих эти запросы.

Как мы уже говорили, если несколько задач пользуются одними и теми же графическими объектами (контекстом отображения, кистями, палитрами и так далее), необходимо обеспечить последовательное обращение этих задач к графическим ресурсам.

В программном интерфейсе операционной системы Microsoft Windows NT предусмотрены различные средства синхронизации задач, как выполняющихся в рамках одного процесса, так и принадлежащих разным процессам. Это уже упоминавшиеся ранее криические секции, события, семафоры и так далее. Такие средства синхронизации будут предметом изучения в данной главе.

Легко ли ждать

Начнем с простого: расскажем о том, как задача может дождаться завершения другой задачи либо завершения процесса.

Если вам нужно сделать так, чтобы одна задача дожидалась завершения другой задачи, работающей в рамках того же процесса, вы можете впасть в соблазн выполнить такое ожидание с помощью глобальной переменной, содержимое которой изменяется при завершении работы второй задачи. Первая задача могла бы при этом опрашивать содержимое этой глобальной переменной в цикле, дожидаясь момента, когда оно изменится.

Очевидный недостаток такого подхода заключается в том, что ожидающая задача получает кванты процессорного времени. При этом снижается общая производительность системы, поэтому необходимо найти другой способ выполнения ожидания. Было бы хорошо организовать ожидание таким образом, чтобы планировщик задач не выделял процессорное время тем задачам, которые чего-то ждут.

Ожидание завершения задачи или процесса

Если нужно дождаться завершения одной задачи или одного процесса, лучше всего воспользоваться для этого функцией WaitForSingleObject, с которой вы уже знакомы из предыдущих приложений.

Прототип функции WaitForSingleObject представлен ниже:

```
DWORD WaitForSingleObject(
  HANDLE hObject,    // идентификатор объекта
  DWORD  dwTimeout); // время ожидания в миллисекундах
```

В качестве параметра hObject этой функции нужно передать идентификатор объекта, для которого выполняется ожидание, а в качестве параметра dwTimeout - время ожидания в миллисекундах (ожидание может быть и бесконечным, если для времени указать значение INFINITE).

Многие объекты операционной системы Microsoft Windows NT, такие, например, как идентификаторы задач, процессов, файлов, могут находиться в двух состояниях - отмеченном (signaled) и неотмеченном (nonsignaled). В частности, если задача или процесс находятся в состоянии выполнения (то есть работают), соответствующие идентификаторы находятся в неотмеченном состоянии. Когда же задача или процесс завершают свою работу, их идентификаторы отмечаются (то есть переходят в отмеченное состояние).

Если задача создает другую задачу или процесс, и затем вызывает функцию WaitForSingleObject, указав ей в качестве первого параметра идентификатор созданной задачи, а в качестве второго - значение INFINITE, родительская задача переходит в состояние ожидания. Она будет находиться в состоянии ожидания до тех пор, пока дочерняя задача или процесс не завершит свою работу.

Заметим, что функция WaitForSingleObject не проверяет состояние идентификатора дочерней задачи или процесса в цикле, дожидаясь ее (или его) завершения. Такое действие привело бы к тому, что родительской задаче выделялись бы кванты процессорного времени, а это как раз то, чего мы хотели бы избежать. Вместо этого функция WaitForSingleObject сообщает планировщику задач, что выполнение родительской задачи, вызвавшей эту функцию, необходимо приостановить до тех пор, пока дочерняя задача или процесс не завершит свою работу.

Принимая решение о выделении родительской задаче кванта процессорного времени, планировщик проверяет состояние дочерней задачи. Квант времени выделяется планировщиком родительской задаче только в том случае, если дочерняя задача завершила свою работу и ее идентификатор находится в отмеченном состоянии. Такая проверка не отнимает много времени и, следовательно, не приводит к заметному снижению производительности системы.

На рис. 4.1 показано, как задача с номером 1 ожидает завершение задачи с номером 2, имеющей идентификатор hThread2.

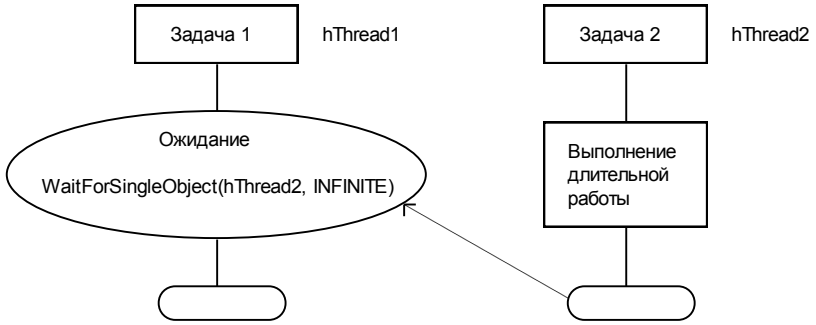

Рис. 4.1. Ожидание завершения задачи

Пунктирной стрелкой здесь показано событие, которое ведет к завершению ожидания. Этим событием, очевидно, является завершение работы задачи hThread2.

В качестве примера использования функции WaitForSingleObject для ожидания завершения дочернего процесса, рассмотрим немного измененный фрагмент исходного текста приложения PSTART, описанного в предыдущей главе:

```
if(CreateProcess(NULL, ofn.lpstrFile, NULL, NULL,
      FALSE, dwCreationFlags, NULL, NULL, &si, &pi))
{
 . . .
 if(WaitForSingleObject(pi.hProcess, INFINITE) !=
   WAIT_FAILED)
 {
  GetExitCodeProcess(pi.hProcess, &dwExitCode);
  . . .
 }
 CloseHandle(pi.hProcess);
 CloseHandle(pi.hThread);
}
```

Здесь главная задача с помощью функции CreateProcess запускает процесс. Идентификатор этого процесса сохраняется в поле hProcess структуры pi. Если запуск процесса произошел успешно, главная задача приложения приостанавливает свою работу до тех пор, пока запущенный процесс не завершит свою работу. Для этого она вызывает функцию WaitForSingleObject, передавая ей идентификатор запущенного процесса.

Если родительский процесс не интересуется судьбой своего дочернего процесса, функция WaitForSingleObject не нужна. В этом случае главная задача может сразу закрыть идентификаторы дочернего процесса и главной задачи дочернего процесса, оборвав “родительские узы”:

```
if(CreateProcess(NULL, ofn.lpstrFile, NULL, NULL,
      FALSE, dwCreationFlags, NULL, NULL, &si, &pi))
{
 CloseHandle(pi.hProcess);
 CloseHandle(pi.hThread);
}
```

Запущенный таким образом процесс называется отсоединенным (detached). Он будет жить своей жизнью независимо от состояния запустившего его процесса.

Теперь поговорим о коде завершения функции WaitForSingleObject.

В случае ошибки функция возвращает значение WAIT_FAILED. При этом код ошибки можно получить при помощи функции GetLastError.

Если же функция завершилась успешно, она может вернуть одно из следующих трех значений: WAIT_OBJECT_0, WAIT_TIMEOUT или WAIT_ABANDONED.

Если состояние идентификатора объекта, для которого выполнялось ожидание, стало отмеченным, функция фозвращает значение WAIT_OBJECT_0. Таким образом, когда мы ожидаем завершение задачи и задача завершилась “естественным образом”, функция WaitForSingleObject вернет именно это значение.

Если время ожидания, заданное во втором параметре функции WaitForSingleObject истекло, но объект так и не перешел в отмеченное состояние, возвращается значение WAIT_TIMEOUT. Очевидно, при бесконечном ожидании вы никогда не получите этот код завершения.

Код завершения WAIT_ABANDONED возвращается для объекта синхронизации типа Mutex (его мы рассмотрим позже), который не был освобожден задачей, завершившей свою работу. Таким образом, в этом случае ожидание было отменено и соответствующий объект (Mutex) не перешел в отмеченное состояние.

Ожидание завершения нескольких задач или процессов

Часто одна задача должна дожидаться завершения сразу нескольких задач или процессов, либо одной из нескольких задач или процессов. Такое ожидание нетрудно выполнить с помощью функции WaitForMultipleObjects, прототип которой приведен ниже:

```
DWORD WaitForMultipleObjects(
  DWORD cObjects,     // количество идентификаторов в массиве
  CONST HANDLE *lphObjects, // адрес массива идентификаторов
  BOOL  fWaitAll,          // тип ожидания
  DWORD dwTimeout);        // время ожидания в миллисекундах
```

Через параметр lphObjects функции WaitForMultipleObjects нужно передать адрес массива идентификаторов. Размер этого массива передается через параметр cObjects.

Если содержимое параметра fWaitAll равно TRUE, задача переводится в состояние ожидания до тех пор, пока все задачи или процессы, идентификаторы которых хранятся в массиве lphObjects, не завершат свою работу. В том случае, когда значение параметра fWaitAll равно FALSE, ожидание прекращается, когда одна из указанных задач или процессов завершит свою работу. Для выполнения бесконечного ожидания, как и в случае функции WaitForSingleObject, через параметр dwTimeout следует передать значение INFINITE.

Как пользоваться этой функцией?

Пример вы можете найти в исходных текстах приложеения MultiSDI, описанного ранее.

Прежде всего вам необходимо подготовить массив для хранения идентификаторов задач или процессов, завершения которых нужно дождаться:

```
HANDLE hThreads[3];
```

После этого в массив следует записать идентификаторы запущенных задач или процессов:

```
hThreads[0] = (HANDLE)_beginthread(PaintEllipse, 0,
  (void*)hWnd);
hThreads[1] = (HANDLE)_beginthread(PaintRect,    0,
  (void*)hWnd);
hThreads[2] = (HANDLE)_beginthread(PaintText,    0,
  (void*)hWnd);
```

Для выполнения ожидания следует вызвать функцию WaitForMultipleObjects, передав ей количество элементов в массиве идентификаторов, адрес этого массива, тип и время ожидания:

```
WaitForMultipleObjects(3, hThreads, TRUE, INFINITE);
```

В данном случае задача, вызвавшая функцию WaitForMultipleObjects, перейдет в состояние ожидания до тех пор, пока все три задачи не завершат свою работу.

Функция WaitForMultipleObjects может вернуть одно из следующих значений:

- WAIT_FAILED (при ошибке);
- WAIT_TIMEOUT (если время ожидания истекло);
- значение в диапазоне от WAIT_OBJECT_0 до (WAIT_OBJECT_0 + cObjects - 1), которое, в зависимости от содержимого параметра fWaitAll, либо означает что все ожидаемые идентификаторы перешли в отмеченное состояние (если fWaitAll был равен TRUE), либо это значение, если из него вычесть константу WAIT_OBJECT_0, равно индексу идентификатора отмеченного объекта в массиве идентфикаторов lphObjects.
- значение в диапазоне от WAIT_ABANDONED_0 до (WAIT_ABANDONED_0 + cObjects - 1), если ожидание для объектов Mutex было отменено. Индекс соответствующего объекта в массиве lphObjects можно определить, если вычесть из кода возврата значение WAIT_ABANDONED_0.

Синхронизация задач с помощью событий

Завершение работы задачи или процесса можно считать событием, которое произошло в системе, и в этом смысле только что расмотренные приемы синхронизации являются ничем иным, как ожиданием события.

Однако синхронизация по завершению работы задачи далеко не всегда удобна, так как две активные задачи могут выполнять некоторые действия, требующие синхронизации во время их работы.

Операционная система Microsoft Windows NT позволяет создавать объекты синхронизации, которые называются событиями (event object). Эти объекты могут находиться в отмеченном или неотмеченном состоянии, причем установка состояния выполняется вызовом соответствующей функции.

Схема использования событий достаточно проста.

Одна из задач создает объект-событие, вызывая для этого функцию CreateEvent. При этом событие имеет имя, которое доступно всем задачам активных процессов. В процессе создания или позже эта задача устанавливает событие в исходное состояние (отмеченное или неотмеченное).

Вызывая функции WaitForSingleObject или WaitForMultipleObjects, задача может выполнять ожидание момента, когда событие перейдет в отмеченное состояние.

Другая задача, принадлежащая тому же самому или другому процессу, может получить идентификатор события по его имени, например, с помощью функции OpenEvent. Далее, пользуясь функциями SetEvent, ResetEvent или PulseEvent, эта задача может изменить состояние события.

На рис. 4.2 приведен пример использования события для синхронизации двух задач, работающих одновременно.

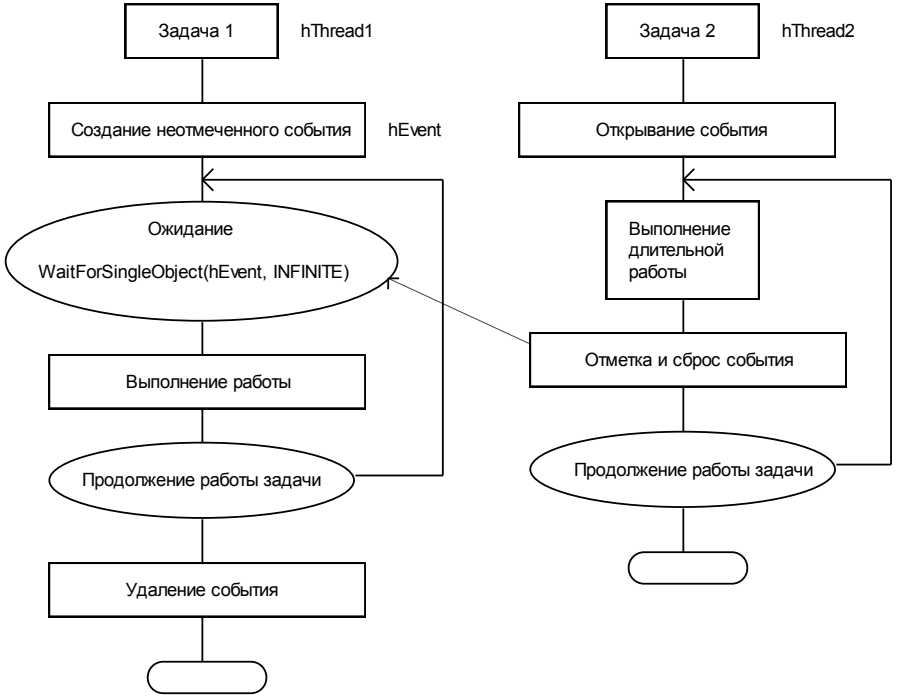

Рис. 4.2. Пример использования события для синхронизации двух задач

Представим себе, что первая задача занимается отображением данных, которые готовятся второй задачей для отображения небольшими порциями (например, читаются с диска).

После создания неотмеченного события первая задача переходит в состояние ожидания, пока вторая задача не подготовит для нее данные. Как только это произойдет, вторая задача отмечает и затем сбрасывает событие, что приводит к завершению ожидания первой задачей.

Отобразив подготовленные данные, первая задача опять входит в состояние ожидания, пока вторая задача не подготовит данные и не отметит событие. Таким образом две задачи синхронизуют свою работу с помощью объекта-события.

Создание события

Для создания события задача должна вызвать функцию CreateEvent, прототип которой приведен ниже:

```
HANDLE CreateEvent(
  LPSECURITY_ATTRIBUTES lpEventAttributes, // атрибуты защиты
  BOOL    bManualReset,  // флаг ручного сброса события
  BOOL    bInitialState, // флаг начального состояния события
  LPCTSTR lpName);       // адрес имени объекта-события
```

Параметр lpEventAttributes задает атрибуты защиты и в большинстве случаев может быть указан как NULL.

С помощью параметра bManualReset вы можете выбрать один из двух режимов работы объекта-события: ручной или автоматический. Если значение этого параметра равно TRUE, событие нужно сбрасывать вручную при помощи функции ResetEvent, которую мы рассмотрим немного позже. Если же для параметра bManualReset указать значение FALSE, событие будет сброшено (то есть переведено в неотмеченное состояние) автоматически сразу после того как задача завершит ожидание этого события.

Параметр bInitialState определяет начальное состояние события. Если этот параметр равен TRUE, объект-событие создается в отмеченном состоянии, а если FALSE - в неотмеченном.

Для того чтобы событием могли пользоваться задачи, созданные разными процессами, необходимо с помощью параметра lpName задать имя объекта-события. В качестве имени вы можете выбрать любое имя размером не более MAX_PATH символов, не содержащее символ “\”.

В том случае, когда событие используется задачами, работающими в рамках одного процесса, имя события можно не задавать, указав параметр lpName как NULL. При этом создается безымянное событие.

В случае успешного завершения функция CreateEvent возвращает идентификатор события, которым нужно будет пользоваться при выполнении всех операций над объектом-событием. При ошибке возвращается значение NULL (код ошибки можно получить при помощи функции GetLastError).

Возможна ситуация, когда при создании события вы указали имя уже существующего в системе события, созданного ранее другой задачей. В этом случае функция GetLastError, вызванная сразу после вызова функции CreateEvent, возвращает значение ERROR_ALREADY_EXISTS.

Открытие события

Если событие используется задачами, созданными только в рамках одного процесса, его не нужно открывать. В качестве параметра функциям, изменяющим состояние объекта-события, вы можете передавать идентификатор события, полученный при его создании от функции CreateEvent.

Если же событие используется для синхронизации задач, принадлежащих разным процессам, вы должны при создании события задать его имя. Задача, изменяющая состояние события и принадлежащая другому процессу, должна открыть объект-событие с помощью функции OpenEvent, передав ей имя этого объекта.

Прототип функции OpenEvent представлен ниже:

```
HANDLE OpenEvent(
  DWORD   fdwAccess,     // флаги доступа
  BOOL    fInherit,      // флаг наследования
  LPCTSTR lpszEventName); // адрес имени объекта-события
```

Флаги доступа, передаваемые через параметр fdwAccess, определяют требуемый уровень доступа к объекту-событию. Этот параметр может быть комбинацией следующих значений:

Значение	*Описание*
EVENT_ALL_ACCESS	Указаны все возможные флаги доступа
EVENT_MODIFY_STATE	Полученный идентификатор можно будет использовать для функций SetEvent и ResetEvent
SYNCHRONIZE	Полученный идентификатор можно будет использовать в любых функциях ожидания события

Параметр fInherit определяет возможность наследования полученного идентификатора. Если этот параметр равен TRUE, идентификатор может наследоваться дочерними процессами. Если же он равен FALSE, наследование не допускается.

И, наконец, через параметр lpszEventName вы должны передать функции адрес символьной строки, содержащей имя объекта-события.

Заметим, что с помощью функции OpenEvent несколько задач могут открыть один и тот же объект-событие и затем выполнять одновременное ожидание для этого объекта.

Установка события

Для установки объекта-события в отмеченное состояние используется функция SetEvent:

```
BOOL SetEvent(HANDLE hEvent);
```

В качестве единственного параметра этой функции необходимо передать идентификатор объекта-события, полученного от функции CreateEvent или

OpenEvent. При успешном завершении возвращается значение TRUE, при ошибке - FALSE. В последнем случае можно получить код ошибки при помощи функции GetLastError.

Сброс события

Сброс события (то есть установка его в неотмеченное состояние) выполняется функцией ResetEvent:

```
BOOL ResetEvent(HANDLE hEvent);
```

Если задача создала событие, работающее в автоматическом режиме, оно будет сбрасываться и без помощи этой функции, если только какая-либо задача выполняла ожидание этого события и событие произошло.

Функция PulseEvent

Функция PulseEvent выполняет установку объекта-события в отмеченное состояние с последующим сбросом события в неотмеченное состояние:

```
BOOL PulseEvent(HANDLE hEvent);
```

Если эта функция вызвана для события, работающего в ручном режиме, то все задачи, ожидающие это событие, завершат ожидание и продолжат свою работу. Событие при этом будет установлено в неотмеченное состояние.

Для автоматических объектов-событий выполняются аналогичные действия, однако функция возвращает управление сразу как только одна из ожидающих задач перейдет в активное состояние.

Приложения EVENT и EVENTGEN

В консольных приложениях EVENT и EVENTGEN, исходные тексты которых приведены в этом разделе, мы демонстрируем использование объектов-событий для синхронизации задач, принадлежащих различным процессам.

Первым необходимо запускать приложение EVENT. Его задача заключается в том, чтобы следить за работой второго приложения EVENTGEN. Приложение EVENTGEN позволяет пользователю вводить с помощью клавиатуры и отображать в своем окне произвольные символы. Каждый раз когда пользователь вводит в окне приложения EVENTGEN какой-либо символ, в окне контролирующего приложения EVENT отображается символ ‘*’ (рис. 4.3).

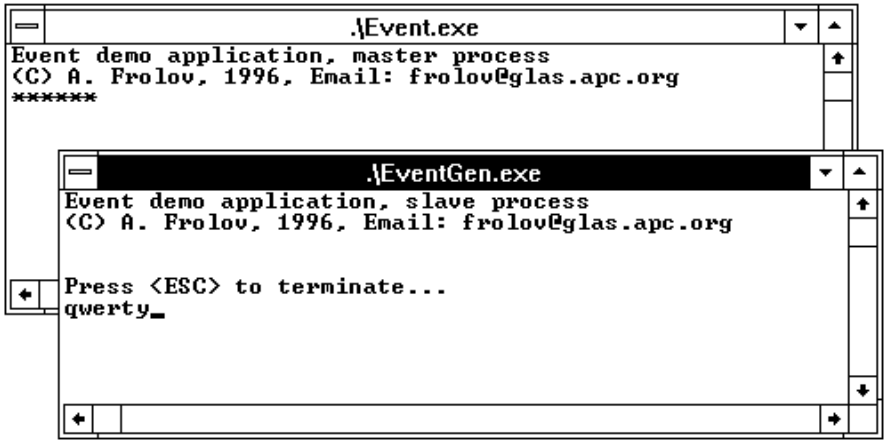

Рис. 4.3. Окна приложений EVENT и EVENTGEN

Так как пока мы не научились еще передавать данные из одного процесса в другой, мы не можем отображать в окне приложения EVENT символы, которые пользователь вводит в окне приложения EVENTGEN. Однако пока нам этого и не надо, так как мы здесь демонстрируем лишь способ синхронизации задач.

Исходные тексты приложения EVENT

Рассмотрим сначала исходные тексты приложения EVENT, представленные в листинге 4.1.

Листинг 4.1. Файл event/event1.c

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>

// Идентификаторы объектов-событий, которые используются
// для синхронизации задач, принадлежащих разным процессам
HANDLE hEvent;
HANDLE hEventTermination;

// Имя объекта-события для синхронизации ввода и отображения
CHAR lpEventName[] =
```

```
  "$MyVerySpecialEventName$";

// Имя объекта-события для завершения процесса
CHAR lpEventTerminationName[] =
  "$MyVerySpecialEventTerminationName$";

int main()
{
  DWORD dwRetCode;

  printf("Event demo application, master process\n"
    "(C) A. Frolov, 1996, Email: frolov@glas.apc.org\n");

  // Создаем объект-событие для синхронизации
  // ввода и отображения, выполняемого в разных процессах
  hEvent = CreateEvent(NULL, FALSE, FALSE, lpEventName);

  // Если произошла ошибка, получаем и отображаем ее код,
  // а затем завершаем работу приложения
  if(hEvent == NULL)
  {
    fprintf(stdout,"CreateEvent: Error %ld\n",
      GetLastError());
    getch();
    return 0;
  }

  // Если объект-событие с указанным именем существует,
  // считаем, что приложение EVENT уже было запущено
  if(GetLastError() == ERROR_ALREADY_EXISTS)
  {
    printf("\nApplication EVENT already started\n"
      "Press any key to exit...");
    getch();
    return 0;
  }

  // Создаем объект-событие для определения момента
  // завершения работы процесса ввода
  hEventTermination = CreateEvent(NULL,
    FALSE, FALSE, lpEventTerminationName);

  if(hEventTermination == NULL)
  {
    fprintf(stdout,"CreateEvent (Termination): Error %ld\n",
      GetLastError());
    getch();
    return 0;
  }

  // Цикл отображения. Этот цикл завершает свою работу
  // при завершении процесса ввода
  while(TRUE)
  {
    // Проверяем состояние объекта-события, отвечающего
    // за контроль завершения процесса ввода. Так как
    // указано нулевое время ожидания, такая проверка
    // не уменьшает заметно скорость работы приложения
    dwRetCode = WaitForSingleObject(hEventTermination, 0);

    // Если объект-событие перешел в отмеченное состояние,
    // если процесс ввода завершил свою работу, или
    // если при ожидании произошла ошибка,
    // останавливаем цикл отображения
    if(dwRetCode == WAIT_OBJECT_0  ||
      dwRetCode == WAIT_ABANDONED ||
      dwRetCode == WAIT_FAILED)
      break;

    // Выполняем ожидание ввода символа в процессе,
    // который работает с клавиатурой
    dwRetCode = WaitForSingleObject(hEvent, INFINITE);
```

```
    // При возникновении ошибки прерываем цикл
    if(dwRetCode == WAIT_FAILED ||
       dwRetCode == WAIT_ABANDONED)
      break;

    // В ответ на каждый символ, введенный процессом, который
    // работает с клавиатурой, отображаем символ '*'
    putch('*');
  }

  // Закрываем идентификаторы объектов-событий
  CloseHandle(hEvent);
  CloseHandle(hEventTermination);

  return 0;
}
```

В глобальных переменных lpEventName и lpEventTerminationName мы храним имена двух объектов-событий, которые будут использоваться для синхронизации. Эти же имена будут указаны функции OpenEvent в приложении EVENTGEN, которое мы рассмотрим чуть позже.

Объекты-события создаются нашим приложением при помощи функции CreateEvent, вызываемой следующим способом:

```
hEvent = CreateEvent(NULL, FALSE, FALSE, lpEventName);
hEventTermination = CreateEvent(NULL, FALSE, FALSE,
  lpEventTerminationName);
```

Здесь в качестве атрибутов защиты мы указываем значение NULL. Через второй и третий параметр функции CreateEvent имеют значение FALSE, поэтому создается автоматический объект-событие, которое изначально находится в неотмеченном состоянии. Имя этого события, доступное всем запущенным приложениям, передается функции CreateEvent через последний параметр.

Обратите внимание на следующий фрагмент кода, который расположен сразу после вызова функции, создающей объект-событие с именем lpEventName:

```
if(GetLastError() == ERROR_ALREADY_EXISTS)
{
  printf("\nApplication EVENT already started\n"
    "Press any key to exit...");
  getch();
  return 0;
}
```

Функция CreateEvent может завершиться с ошибкой и в этом случае она возвращает значение NULL. Однако пользователь может попытаться запустить приложение EVENT два раза. В первый раз при этом будет создан объект-событие с именем lpEventName. Когда же функция CreateEvent будет вызвана для этого имени еще раз при попытке повторного запуска приложения, она не вернет признак ошибки, несмотря на то что объект-событие с таким именем уже существует в системе. Вместо этого функция вернет идентификатор для уже существующего объекта-события.

Как распознать такую ситуацию?

Очень просто - достаточно сразу после вызова функции CreateEvent проверить код завершения при помощи функции GetLastError. Как мы уже говорили, в случае попытки создания объекта-события с уже существующим в системе именем эта функция вернет значение ERROR_ALREADY_EXISTS. Как только это произойдет, наше приложение выдает сообщение о том, что его копия уже запущена, после чего завершает свою работу.

Таким образом мы нашли еще один способ (далеко не последний), с помощью которого в операционной системе Microsoft Windows NT можно предотвратить повторный запуск приложений. До сих пор мы решали эту задачу в графических приложениях с помощью функции FindWindow.

Продолжим изучение исходный текстов приложения EVENT.

После того как приложение создаст два объекта-события, оно входит в цикл отображения символа ‘*’.

В этом цикле прежде всего проверяется состояние объекта-события с идентификатором hEventTermination:

```
dwRetCode = WaitForSingleObject(hEventTermination, 0);
if(dwRetCode == WAIT_OBJECT_0  ||
   dwRetCode == WAIT_ABANDONED ||
   dwRetCode == WAIT_FAILED)
  break;
```

Приложение EVENTGEN переводит этот объект в отмеченное состояние перед своим завершением.

Обращаем ваше внимание на то что функции WaitForSingleObject указано нулевое время ожидания. В этом случае функция проверяет состояние объекта и сразу же возвращает управление.

Далее выполняется ожидание объекта-события с идентификатором hEvent:

```
dwRetCode = WaitForSingleObject(hEvent, INFINITE);
if(dwRetCode == WAIT_FAILED ||
   dwRetCode == WAIT_ABANDONED)
  break;
```

Если это ожидание завершилось с ошибкой, выполняется выход из цикла. Если же оно было выполнено без ошибок, приложение EVENT отображает символ '*', пользуясь для этого функцией putch, известной вам из практики программирования для MS-DOS.

Перед завершением работы приложения мы выполняем освобождение идентификаторов созданных объектов-событий, пользуясь для этого функцией CloseHandle:

```
CloseHandle(hEvent);
CloseHandle(hEventTermination);
```

Исходные тексты приложения EVENTGEN

Исходные тексты приложения EVENTGEN, которое работает в паре с только что рассмотренным приложением EVENT, приведены в листинге 4.2.

Листинг 4.2. Файл event/eventgen/event2.c

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>

// Идентификаторы объектов-событий, которые используются
// для синхронизации задач, принадлежащих разным процессам
HANDLE hEvent;
HANDLE hEventTermination;

// Имя объекта-события для синхронизации ввода и отображения
CHAR lpEventName[] =
  "$MyVerySpecialEventName$";

// Имя объекта-события для завершения процесса
CHAR lpEventTerminationName[] =
  "$MyVerySpecialEventTerminationName$";

int main()
{
CHAR chr;

printf("Event demo application, slave process\n"
  "(C) A. Frolov, 1996, Email: frolov@glas.apc.org\n"
  "\n\nPress <ESC> to terminate...\n");

// Открываем объект-событие для синхронизации
// ввода и отображения
hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, lpEventName);

if(hEvent == NULL)
{
  fprintf(stdout,"OpenEvent: Error %ld\n",
    GetLastError());
  getch();
  return 0;
}

// Открываем объект-событие для сигнализации о
// завершении процесса ввода
hEventTermination = OpenEvent(EVENT_ALL_ACCESS,
  FALSE, lpEventTerminationName);

if(hEventTermination == NULL)
{
  fprintf(stdout,"OpenEvent (Termination): Error %ld\n",
    GetLastError());
  getch();
  return 0;
}

// Цикл ввода. Этот цикл завершает свою работу,
// когда пользователь нажимает клавишу <ESC>,
// имеющую код 27
while(TRUE)
{
```

```
  // Проверяем код введенной клавиши
  chr = getche();

  // Если нажали клавишу <ESC>, прерываем цикл
  if(chr == 27)
    break;

  // Устанавливаем объект-событие в отмеченное
  // состояние. В ответ на это процесс отображения
  // выведет на свою консоль символ '*'
  SetEvent(hEvent);
 }

 // После завершения цикла переключаем оба события
 // в отмеченное состояние для отмены ожидания в
 // процессе отображения и для завершения этого процесса
 SetEvent(hEvent);
 SetEvent(hEventTermination);

 // Закрываем идентификаторы объектов-событий
 CloseHandle(hEvent);
 CloseHandle(hEventTermination);

 return 0;
}
```

В глобальных переменных lpEventName и lpEventTerminationName записаны имена объектов-событий, которые в точности такие же, как и в приложении EVENT.

Функция main приложения EVENTGEN прежде всего открывает объекты-события, созданные приложением EVENT. Для этого она вызывает функцию OpenEvent, как это показано ниже:

```
hEvent = OpenEvent(EVENT_ALL_ACCESS, FALSE, lpEventName);
hEventTermination = OpenEvent(EVENT_ALL_ACCESS,
 FALSE, lpEventTerminationName);
```

Если пользователь случайно запустил первым приложение EVENTGEN, первый из только что приведенных вызовов функции OpenEvent закончится с ошибкой, так как объекты-события с данными именами неизвестны системе. При этом работа приложения EVENTGEN завершается аварийно с выдачей соответствующего сообщения.

Далее функция main приложения EVENTGEN входит в цикл, в котором она выполняет ввод символов с клавиатуры. Этот цикл прерывается, если пользователь нажимает клавишу <ESC>.

После ввода каждого символа приложение EVENTGEN устанавливает объект-событие hEvent в отмеченное состояние, вызывая для этого функцию SetEvent:

```
SetEvent(hEvent);
```

При этом приложение EVENT выходит из состояния ожидания и отображает в своем окне один символ ‘*’. Так как объект-событие работает в автоматическом режиме, после выхода главной задачи приложения EVENT из состояния ожидания объект-событие hEvent автоматически сбрасывается в неотмеченное состояние. В результате после отображения одного символа ‘*’ приложение EVENT вновь переводится в состояние ожидания до тех пор, пока пользователь не нажмет какую-либо клавишу в окне приложения EVENTGEN.

После того как пользователь нажал клавишу <ESC>, оба объекта-события переключаются в отмеченное состояние:

```
SetEvent(hEvent);
SetEvent(hEventTermination);
```

Это приводит к тому что в приложении EVENT завершается цикл отображения.

Последовательный доступ к ресурсам

При создании мультизадачных приложений часто возникает необходимость обеспечить последовательный доступ параллельно работающих задач к какому-либо ресурсу. Типичным примером может послужить приложение MULTISDI, в котором несколько задач одновременно рисуют в одном окне приложения, пользуясь контекстом отображения и графическими функциями. Другой пример - данные, которые изменяются и читаются несколькими задачами сразу. Процедуры изменения и чтения в данном случае необходимо выполнять строго последовательно, иначе данные могут быть искажены.

Приведем еще один типичный пример, когда необходимо обеспечить последовательный доступ к ресурсу.

Пусть на счету в банке у фирмы лежит 2 млн. долларов. Два торговых агента фирмы одновременно пытаются сделать покупки, причем стоимость товара в обоих случаях равна 1,5 млн. долларов. При этом процесс покупки заключается в том, что вначале программа получает запись базы данных, в

которой хранится общая сумма, затем она вычитает из нее стоимость товара и сохраняет новое значение общей суммы.

Допустим, что для выполнения покупки программа создает по одной задаче на каждого торгового агента, и эти задачи будут работать одновременно. Если не предусмотреть средства синхронизации доступа к записи базы данных, возможно возникновение такой ситуации, когда процессы покупки (получение общей суммы - вычитание стоимости товара - сохранение нового значение общей суммы), выполняемые этими двумя задачами, перекроются во времени.

К чему это может привести?

Если перекрытия во времени не произойдет, то вначале задача, запущенная для первого торгового агента, прочитает общую сумму, которая будет равна 2 млн. долларов. Затем она вычтет из нее стоимость покупки и запишет результат (500 тыс. долларов) обратно. Второй торговый агент уже не сможет сделать покупку, так как он обнаружит, что на счету осталось слишком мало денег.

В том случае, если покупки делаются одновременно, события могут развиваться в следующей последовательности:

- первый агент получает значение общей суммы (2 млн. долларов);
- второй агент получает значение общей суммы (2 млн. долларов);
- первый агент вычитает из этой суммы стоимость товара (1,5 млн. долларов) и записывает результат (500 тыс. долларов) в базу данных;
- второй агент вычитает из общей суммы стоимость товара и записывает результат в базу данных

Кто пострадает в данной ситуации?

Вероятно, банк, так как два торговых агента сделают покупки на сумму, превышающую ту, что лежит на счету у фирмы. Возможно также, что и программист, который не предусмотрел возможность возникновения таких накладок.

Ошибку можно было бы предотвратить, если во время выполнения любой задачей операции изменения текущего счета запретить доступ к этому счету других задач. Можно предложить следующий вариант сценария совершения покупок:

- задача изменения счета проверяет, не заблокирован ли доступ к соответствующей записи базы данных. Если заблокирован, задача переводится в состояние ожидания до разблокировки. Если же доступ не заблокирован, задача выполняет блокировку такого доступа для всех других задач;
- задача получает значения общей суммы;
- задача выполняет вычитание стоимости купленного товара из общей суммы и записывает результат в базу данных;
- задача разблокирует доступ к записи базы данных

Таким образом, при использовании приведенного выше сценария во время попытки одновременного изменения значения счета двумя торговыми агентами задача, запущенная чуть позже, перейдет в состояние ожидания. Когда же первый агент сделает покупку, задача второго агента получит правильное значение остатка (500 тыс. долларов), что не позволит ему приобрести товар на 1,5 млн. долларов.

В программном интерфейсе операционной системы Microsoft Windows NT имеются удобные средства организации последовательного доступа к ресурсам. Это критические секции, объекты взаимно исключающего доступа Mutex и блокирующие функции изменения содержимого переменных.

Критические секции

Критические секции являются наиболее простым средством синхронизации задач для обеспечения последовательного доступа к ресурсам. Они работают быстро, не снижая заметно производительность системы, но обладают одним существенным недостатком - их можно использовать только для синхронизации задач в рамках одного процесса. В отличие от критических секций объекты Mutex, которые мы рассмотрим немного позже, допускают синхронизацию задач, созданных разными процессами.

Критическая секция создается как структура типа CRITICAL_SECTION:

```
CRITICAL_SECTION csWindowPaint;
```

Обычно эта структура располагается в области глобальных переменных, доступной всем запущенным задачам процесса. Так как каждый процесс работает в своем собственном адресном пространстве, вы не сможете передать адрес критической секции из одного процесса в другой. Именно поэтому критические секции нельзя использовать для синхронизации задач, созданных разными процессами.

В файле winbase.h (который включается автоматически при включении файла windows.h) структура CRITICAL_SECTION и указатели на нее определены следующим образом:

```
typedef RTL_CRITICAL_SECTION CRITICAL_SECTION;
typedef PRTL_CRITICAL_SECTION PCRITICAL_SECTION;
typedef PRTL_CRITICAL_SECTION LPCRITICAL_SECTION;
```

Определение недокументированной структуры RTL_CRITICAL_SECTION вы можете найти в файле winnt.h:

```
typedef struct _RTL_CRITICAL_SECTION
{
  PRTL_CRITICAL_SECTION_DEBUG DebugInfo;
```

```
  LONG   LockCount;      // счетчик блокировок
  LONG   RecursionCount; // счетчик рекурсий
  HANDLE OwningThread;   // идентификатор задачи, владеющей
                         // секцией
  HANDLE LockSemaphore;  // идентификатор семафора
  DWORD  Reserved;       // зарезервировано
} RTL_CRITICAL_SECTION, *PRTL_CRITICAL_SECTION;
```

Заметим, однако, что нет никакой необходимости изменять поля этой структуры вручную, так как для этого есть специальные функции. Более того, только эти функции и можно использовать для работы с критическими секциями. В документации SDK также отмечено, что структуру типа CRITICAL_SECTION нельзя перемещать или копировать.

Инициализация критической секции

Перед использованием критической секции ее необходимо проинициализировать, вызвав для этого функцию InitializeCriticalSection:

```
CRITICAL_SECTION csWindowPaint;
InitializeCriticalSection(&csWindowPaint);
```

Функция InitializeCriticalSection имеет только один параметр (адрес структуры типа CRITICAL_SECTION) и не возвращает никакого значения.

Удаление критической секции

Если критическая секция больше не нужна, ее нужно удалить при помощи функции DeleteCriticalSection, как это показано ниже:

```
DeleteCriticalSection(&csWindowPaint);
```

При этом освобождаются все ресурсы, созданные операционной системой для критической секции.

Вход в критическую секцию и выход из нее

Две основные операции, выполняемые задачами над критическими секциями, это вход в критическую секцию и выход из критической секции. Первая операция выполняется при помощи функции EnterCriticalSection, вторая - при помощи функции LeaveCriticalSection. Эти функции, не возвращающие никакого значения, всегда используются в паре, как это показано в следующем фрагменте исходного текста рассмотренного нами ранее приложения MultiSDI:

```
EnterCriticalSection(&csWindowPaint);
hdc = BeginPaint(hWnd, &ps);
GetClientRect(hWnd, &rc);
DrawText(hdc, "SDI Window", -1, &rc,
  DT_SINGLELINE | DT_CENTER | DT_VCENTER);
EndPaint(hWnd, &ps);
LeaveCriticalSection(&csWindowPaint);
```

В качестве единственного параметра функциям EnterCriticalSection и LeaveCriticalSection необходимо передать адрес стрктуры типа CRITICAL_SECTION, проинициализированной предварительно функцией InitializeCriticalSection.

Как работают критические секции?

Если одна задача вошла в критическую секцию, но еще не вышла ее, то при попытке других задач войти в ту же самую критическую секцию они будут переведены в состояние ожидания. Задачи пробудут в этом состоянии до тех пор, пока задача, которая вошла в критическую секцию, не выйдет из нее.

Таким образом гарантируется, что фрагмент кода, заключенный между вызовами функций EnterCriticalSection и LeaveCriticalSection, будет выполняться задачами последовательно, если все они работают с одной и той же критической секцией.

Рекурсивный вход в критическую секцию

Операционная система Microsoft Windows NT допускает рекурсивный вход в критическую секцию. Например, приведенный выше фрагмент кода можно было бы составить следующим образом:

```
EnterCriticalSection(&csWindowPaint);
PaintClient(hWnd);
LeaveCriticalSection(&csWindowPaint);
. . .
void PaintClient(HWND hWnd)
{
  . . .
  EnterCriticalSection(&csWindowPaint);
  hdc = BeginPaint(hWnd, &ps);
  GetClientRect(hWnd, &rc);
  DrawText(hdc, "SDI Window", -1, &rc,
    DT_SINGLELINE | DT_CENTER | DT_VCENTER);
  EndPaint(hWnd, &ps);
  LeaveCriticalSection(&csWindowPaint);
}
```

Здесь мы выполняем вызов функции PaintClient, находясь в критической секции csWindowPaint. При этом сама функция PaintClient также пользуется той же критической секцией.

Рекурсивный вход задачи в ту же самую критическую секцию не приводит к тому, что задача переходит в состояние ожидания. Однако для освобождения критической секции необходимо вызывать функцию LeaveCriticalSection столько же раз, сколько раз вызывается функция EnterCriticalSection.

Работа задачи с несколькими критическими секциями

В том случае, когда задача работает с двумя ресурсами, доступ к которым должен выполняться последовательно, она может создать несколько критических секций. Например, приложение MultiMDI, описанное в этой книге, создает критические секции для каждого MDI-окна. Эти секции выполняют синхронизацию главной задачи приложения с задачами, создаваемыми для MDI-окон. Такая синхронизация выполняется с целью предотвращения одновременного рисования в MDI-окне главной задачей и задачей, запущенной для этого MDI-окна.

Заметим, что когда задачи работают с несколькими критическими секциями, все они должны использовать одинаковую последовательность входа в эти критические секции и выхода из них, иначе возможны взаимные блокировки задач.

Пусть, например, в приложении определены две критические секции, синхронизирующие рисование в двух окнах:

```
CRITICAL_SECTION csWindowOnePaint;
CRITICAL_SECTION csWindowTwoPaint;
```

Пусть первая задача выполняет рисование в этих окнах следующим образом:

```
EnterCriticalSection(&csWindowOnePaint);
EnterCriticalSection(&csWindowTwoPaint);
PaintClientWindow(hWndOne);
PaintClientWindow(hWndTwo);
LeaveCriticalSection(&csWindowTwoPaint);
LeaveCriticalSection(&csWindowOnePaint);
```

Пусть вторая задача использует другой порядок входа в критические секции и выхода из них:

```
EnterCriticalSection(&csWindowTwoPaint);
EnterCriticalSection(&csWindowOnePaint);
PaintClientWindow(hWndOne);
PaintClientWindow(hWndTwo);
LeaveCriticalSection(&csWindowOnePaint);
LeaveCriticalSection(&csWindowTwoPaint);
```

При этом есть вероятность того что когда первая задача войдет в критическую секцию csWindowOnePaint, управление будет передано второй задаче, которая войдет в критическую секцию csWindowTwoPaint и перейдет в состояние ожидания. Она будет ждать освобождения критической секции csWindowOnePaint, занятой первой задачей.

Однако первая задача тоже перейдет в состояние ожидания, так как ей нужна критическая секция csWindowTwoPaint, занятая второй задачей. В результате обе задачи навсегда останутся в состоянии ожидания, так как они не смогут освободить критические секции, нужные друг другу.

Если у вас нет необходимости выполнять рисование во втором окне сразу после рисования в первом окне, вы можете избежать взаимной блокировки задач не только используя правильную последовательность входа и выхода в критические секции, но и выполняя последовательное использование этих критических секций:

```
// Рисование в первом окне
EnterCriticalSection(&csWindowOnePaint);
PaintClientWindow(hWndOne);
LeaveCriticalSection(&csWindowOnePaint);

// Рисование во втором окне
EnterCriticalSection(&csWindowTwoPaint);
PaintClientWindow(hWndTwo);
LeaveCriticalSection(&csWindowTwoPaint);
```

Объекты Mutex

Если необходимо обеспечить последовательное использование ресурсов задачами, созданными в рамках разных процессов, вместо критических секций необходимо использовать объекты синхронизации Mutex. Свое название они получили от выражения “mutually exclusive”, что означает “взаимно исключающий”.

Также как и объект-событие, объект Mutex может находится в отмеченном или неотмеченном состоянии. Когда какая-либо задача, принадлежащая любому процессу, становится владельцем объекта Mutex, последний переключается в неотмеченное состояние. Если же задача “отказывается” от владения объектом Mutex, его состояние становится отмеченным.

Организация последовательного доступа к ресурсам с использованием объектов Mutex возможна потому, что в каждый момент только одна задача может владеть этим объектом. Все остальные задачи для того чтобы завладеть объектом, который уже захвачен, должны ждать, например, с помощью уже известных вам функции WaitForSingleObject.

Для того чтобы объект Mutex был доступен задачам, принадлежащим различным процессам, при создании вы должны присвоить ему имя (аналогично тому, как вы это делали для объекта-события).

Создание объекта Mutex

Для создания объекта Mutex вы должны использовать функцию CreateMutex, прототип которой мы привели ниже:

```
HANDLE CreateMutex(
  LPSECURITY_ATTRIBUTES lpMutexAttributes, // атрибуты защиты
  BOOL bInitialOwner, // начальное состояние
  LPCTSTR lpName);    // имя объекта Mutex
```

В качестве первого параметра (атрибуты защиты) вы можете указать значение NULL (как и во всех наших примерах).

Параметр bInitialOwner определяет начальное состояние объекта Mutex. Если он имеет значение TRUE, задача, создающая объект Mutex, будет им владеть сразу после создания. Если же значение этого параметра равно FALSE, после создания объект Mutex не будет принадлежать ни одной задаче, пока не будет захвачен ими явным образом.

Через параметр lpName вы должны передать указатель на имя объекта Mutex, для которого действуют те же правила, что и для имени объекта-события. Это имя не должно содержать символ '\' и его длина не должна превышать значение MAX_PATH.

Если объект Mutex будет использован только задачами одного процесса, вместо адреса имени можно указать значение NULL. В этом случае будет создан "безымянный" объект Mutex.

Функция CreateMutex возвращает идентификатор созданного объекта Mutex или NULL при ошибке.

Возможно возникновение такой ситуации, когда приложение пытается создать объект Mutex с именем, которое уже используется в системе другим объектом Mutex. В этом случае функция CreateMutex вернет идентификатор существующего объекта Mutex, а функция GetLastError, вызванная сразу после вызова функции CreateMutex, вернет значение ERROR_ALREADY_EXISTS. Заметим, что функция создания объектов-событий CreateEvent ведет себя в данной ситуации аналогичным образом.

Освобождение идентификатора объекта Mutex

Если объект Mutex больше не нужен, вы должны освободить его идентификатор при помощи универсальной функции CloseHandle. Заметим, тем не менее, что при завершении процесса освобождаются идентификаторы всех объектов Mutex, созданных для него.

Открытие объекта Mutex

Зная имя объекта Mutex, задача может его открыть с помощью функции OpenMutex, прототип которой приведен ниже:

```
HANDLE OpenMutex(
  DWORD   fdwAccess, // требуемый доступ
  BOOL    fInherit,  // флаг наследования
  LPCTSTR lpszMutexName ); // адрес имени объекта Mutex
```

Флаги доступа, передаваемые через параметр fdwAccess, определяют требуемый уровень доступа к объекту Mutex. Этот параметр может быть комбинацией следующих значений:

Значение	*Описание*
EVENT_ALL_ACCESS	Указаны все возможные флаги доступа
SYNCHRONIZE	Полученный идентификатор можно будет использовать в любых функциях ожидания события

Параметр fInherit определяет возможность наследования полученного идентификатора. Если этот параметр равен TRUE, идентификатор может наследоваться дочерними процессами. Если же он равен FALSE, наследование не допускается.

Через параметр lpszEventName вы должны передать функции адрес символьной строки, содержащей имя объекта Mutex.

С помощью функции OpenMutex несколько задач могут открыть один и тот же объект Mutex и затем выполнять одновременное ожидание для этого объекта.

Как завладеть объектом Mutex

Зная идентификатор объекта Mutex, полученный от функций CreateMutex или OpenMutex, задача может завладеть объектом при помощи функций ожидания событий, например, при помощи функций WaitForSingleObject или WaitForMultipleObjects.

Напомним, что функция WaitForSingleObject возвращает управление, как только идентификатор объекта, передаваемый ей в качестве параметра, перейдет в отмеченное состояние. Если объект Mutex не принадлежит ни одной задаче, его состояние будет отмеченным.

Когда вы вызываете функцию WaitForSingleObject для объекта Mutex, который никому не принадлежит, она сразу возвращает управление. При этом задача, вызвавшая функцию WaitForSingleObject, становится владельцем объекта Mutex. Если теперь другая задача вызовет функцию WaitForSingleObject для этого же объекта Mutex, то она будет переведена в

сотояние ожидания до тех пор, пока первая задача не “откажется от своих прав” на данный объект Mutex. Освобождение объекта Mutex выполняется функцией ReleaseMutex, которую мы сейчас рассмотрим.

Захват объекта Mutex во владение по своему значению аналогичен входу в критическую секцию.

Освобождение объекта Mutex

Для отказа от владения объектом Mutex (то есть для его освобождения) вы должны использовать функцию ReleaseMutex:

```
BOOL ReleaseMutex(HANDLE  hMutex);
```

Через единственный параметр этой функции необходимо передать идентификатор объекта Mutex. Функция возвращает значение TRUE при успешном завершении и FALSE при ошибке.

Проводя аналогию с критическими секциями, заметим, что освобождение объекта Mutex соответствует выходу из критической секции.

Рекурсивное использование объектов Mutex

Так же как и критические секции, объекты Mutex допускают рекурсивное использование. Задача может выполнять рекурсивные попытки завладеть одним и тем же объектом Mutex и при этом она не будет переводиться в состояние ожидания.

В случае рекурсивного использования каждому вызову функции ожидания должен соответствовать вызов функции освобождения объекта Mutex ReleaseMutex.

Блокирующие функции

Если несколько задач выполняют изменение одной и той же глобальной переменной, их необходимо синхронизировать, чтобы одновременно к этой переменной обращалась только одна задача. Для этого вы можете воспользоваться рассмотренными нами ранее критическими секциями или объектами Mutex, однако в программном интерфейсе операционной системы Microsoft Windows NT предусмотрено несколько функций, которые просты в использовании и могут оказаться полезными в данной ситуации.

Функции InterlockedIncrement и InterlockedDecrement выполняют, соответственно, увеличение и уменьшение на единицу значения переменной типа LONG, адрес которой передается им в качестве единственного параметра:

```
LONG InterlockedIncrement(LPLONG lpAddend);
LONG InterlockedDecrement(LPLONG lpAddend);
```

Особенностью этих функций является то, что если одна задача приступила с их помощью к изменению значения переменной, то другая задача не сможет выполнить изменение этой же переменной до тех пор, пока первая задача не завершит такое изменение.

В результате при использовании этих функций можно быть уверенным, что изменение переменной будет выполнено правильно.

А как может произойти ошибка?

Пусть, например, первая задача увеличивает содержимое глобальной переменной lAddend следующим образом:

```
lAddend += 1;
```

Несмотря на то что оператор, выполняющий такое увеличение, разместился на одной строке исходного текста, ему может соответствовать несколько машинных команд. Для того чтобы увеличить значение переменной, программа может вначале прочитать, например, содержимое этой переменной во внутренний регистр процессора, затем увеличить содержимое этого регистра и записать в оперативную память результат. Этот процесс может быть прерван в любое время, так как планировщик задач может выделить квант времени другой задаче.

Если теперь эта другая задача попытается выполнить увеличение той же самой переменной, она будет увеличивать старое значение, которое было до того момента, как первая задача начала его увеличение. В результате значение переменной будет увеличено не два раза, как это должно быть (так как две задачи пытались увеличить значение переменной), а только один раз. Эта ситуация напоминает случай с двумя торговыми агентами, которые перерасходовали деньги своей фирмы. Если же для увеличения переменной использовать функцию InterlockedIncrement, такой ошибки не произойдет.

Очевидно, что операция присваивания глобальной переменной нового значения, если она выполняется несколькими задачами одновременно, таит в себе ту же опасность. Например, следующая строка исходного текста может быть транслирована в несколько машинных команд и, следовательно, может быть прервана другой задачей того же процесса:

```
lTaget = lNewValue;
```

Специально для того чтобы избежать такой опасности, в программном интерфейсе Microsoft Windows NT предусмотрена функция InterlockedExchange:

```
LONG InterlockedExchange(
  LPLONG lpTarget,   // адрес изменяемой переменной
  LONG   lNewValue); // новое значение для переменной
```

Эта функция записывает значение lNewValue по адресу lpTarget. При этом гарантируется, что операция не будет прервана другой задачей, выполняющейся в рамках того же процесса.

Функция InterlockedExchange возвращает старое значение изменяемой переменной.

Что же касается значения, возвращаемого функциями InterlockedIncrement и InterlockedDecrement, то оно равно нулю, если в результате изменений значение переменной стало равно нулю. Если в результате увеличения или уменьшения значение переменной стало больше или меньше нуля, то эти функции возвращают, соответственно, значение, большее или меньшее нуля. Это значение, однако, можно использовать только для сравнения, так как абсолютная величина возвращенного значения не равна новому значению изменяемой переменной.

Приложение MutexSDI

Для демонстрации способор работы с объектами Mutex мы переделали исходные тексты приложения MultiSDI, в котором синхронизация задач выполнялась при помощи критических секций. В новом приложении MutexSDI вместо критической секции мы использовали объект Mutex. Кроме того, в приложении MutexSDI демонстрируется способ обнаружения работающей копии приложения, основанный на том, что имена объектов Mutex являются глобальными (так же, как и имена объектов-событий, рассмотренных нами ранее).

Исходный текст приложения приведен в листинге 4.3. Так как он сделан из исходного текста приложения MultiSDI, мы опишем только основные отличия.

Листинг 4.3. Файл mutexsdi/mutexsdi.c

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <process.h>
#include <stdio.h>
#include "resource.h"
#include "afxres.h"
#include "mutexsdi.h"

HINSTANCE hInst;
char szAppName[]   = "MutexMultiSDI";
char szAppTitle[]  = "Multithread SDI Application with Mutex";

// Имя объекта Mutex
char szMutexName[] = "$MyMutex$MutexMultiSDI$";

// Идентификатор объекта Mutex
HANDLE hMutex;

// Признак завершения всех задач
BOOL fTerminate = FALSE;

// Массив идентификаторов запущенных задач
HANDLE hThreads[3];

// -------------------------------------------------------
// Функция WinMain
// -------------------------------------------------------
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
 WNDCLASSEX wc;
 HWND hWnd;
 MSG msg;

 // Сохраняем идентификатор приложения
 hInst = hInstance;

 // Создаем объект Mutex
 hMutex = CreateMutex(NULL, FALSE, szMutexName);
 if(hMutex == NULL)
 {
  MessageBox(NULL, "CreateMutex Error",
      szAppTitle, MB_OK | MB_ICONEXCLAMATION);
  return 0l;
 }

 // Преверяем, не было ли это приложение запущено ранее
 if(GetLastError() == ERROR_ALREADY_EXISTS)
 {
  MessageBox(NULL, "MutexSDI already started",
      szAppTitle, MB_OK | MB_ICONEXCLAMATION);
```

```
  return 0l;
}

// Регистрируем класс окна
memset(&wc, 0, sizeof(wc));
wc.cbSize = sizeof(WNDCLASSEX);
wc.hIconSm = LoadImage(hInst,
  MAKEINTRESOURCE(IDI_APPICONSM),
  IMAGE_ICON, 16, 16, 0);
wc.style = 0;
wc.lpfnWndProc = (WNDPROC)WndProc;
wc.cbClsExtra  = 0;
wc.cbWndExtra  = 0;
wc.hInstance = hInst;
wc.hIcon = LoadImage(hInst,
  MAKEINTRESOURCE(IDI_APPICON),
  IMAGE_ICON, 32, 32, 0);
wc.hCursor = LoadCursor(NULL, IDC_ARROW);
wc.hbrBackground = (HBRUSH) (COLOR_WINDOW + 1);
wc.lpszMenuName = MAKEINTRESOURCE(IDR_APPMENU);
wc.lpszClassName = szAppName;
if(!RegisterClassEx(&wc))
  if(!RegisterClass((LPWNDCLASS)&wc.style))
              return FALSE;

// Создаем главное окно приложения
hWnd = CreateWindow(szAppName, szAppTitle,
   WS_OVERLAPPEDWINDOW,
   CW_USEDEFAULT, 0, CW_USEDEFAULT, 0,
   NULL, NULL, hInst, NULL);
if(!hWnd) return(FALSE);

// Отображаем окно и запускаем цикл
// обработки сообщений
ShowWindow(hWnd, nCmdShow);
UpdateWindow(hWnd);
  while(GetMessage(&msg, NULL, 0, 0))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
  return msg.wParam;
}

// ----------------------------------------------------
// Функция WndProc
// ----------------------------------------------------
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam,
        LPARAM lParam)
{
  switch(msg)
  {
    HANDLE_MSG(hWnd, WM_CREATE,     WndProc_OnCreate);
    HANDLE_MSG(hWnd, WM_DESTROY,    WndProc_OnDestroy);
    HANDLE_MSG(hWnd, WM_PAINT,      WndProc_OnPaint);
    HANDLE_MSG(hWnd, WM_COMMAND,    WndProc_OnCommand);

    default:
      return(DefWindowProc(hWnd, msg, wParam, lParam));
  }
}

// ----------------------------------------------------
// Функция WndProc_OnCreate
// ----------------------------------------------------
BOOL WndProc_OnCreate(HWND hWnd,
                      LPCREATESTRUCT lpCreateStruct)
{
  // Сбрасываем флаг завершения задач
  fTerminate = FALSE;
```

```
  // Запускаем три задачи, сохраняя их идентификаторы
  // в массиве
  hThreads[0] = (HANDLE)_beginthread(PaintEllipse,
    0, (void*)hWnd);
  hThreads[1] = (HANDLE)_beginthread(PaintRect,
    0, (void*)hWnd);
  hThreads[2] = (HANDLE)_beginthread(PaintText,
    0, (void*)hWnd);

  return TRUE;
}

// ----------------------------------------------------
// Функция WndProc_OnDestroy
// ----------------------------------------------------
#pragma warning(disable: 4098)
void WndProc_OnDestroy(HWND hWnd)
{
  // Устанавливаем флаг завершения задач
  fTerminate = TRUE;

  // Дожидаемся завершения всех трех задач
  WaitForMultipleObjects(3, hThreads, TRUE, INFINITE);

  // Перед завершением освобождаем идентификатор
  // объекта Mutex
  CloseHandle(hMutex);

  // Останавливаем цикл обработки сообщений, расположенный
  // в главной задаче
  PostQuitMessage(0);
  return 0L;
}

// ----------------------------------------------------
// Функция WndProc_OnPaint
// ----------------------------------------------------
#pragma warning(disable: 4098)
void WndProc_OnPaint(HWND hWnd)
{
  HDC hdc;
  PAINTSTRUCT ps;
  RECT rc;
  DWORD dwRetCode;

  // Ожидаем, пока объект Mutex не перейдет в
  // отмеченное состояние
  dwRetCode = WaitForSingleObject(hMutex, INFINITE);

  // Если не было ошибок, выполняем рисование
  if(dwRetCode == WAIT_OBJECT_0)
  {
    // Перерисовываем внутреннюю область окна
    hdc = BeginPaint(hWnd, &ps);
    GetClientRect(hWnd, &rc);
    DrawText(hdc, "SDI Window", -1, &rc,
      DT_SINGLELINE | DT_CENTER | DT_VCENTER);
    EndPaint(hWnd, &ps);

    // Переводим объект Mutex в неотмеченное состояние
    ReleaseMutex(hMutex);
  }

  return 0;
}
// ----------------------------------------------------
// Функция WndProc_OnCommand
// ----------------------------------------------------
#pragma warning(disable: 4098)
void WndProc_OnCommand(HWND hWnd, int id,
  HWND hwndCtl, UINT codeNotify)
{
```

```
  switch (id)
  {
    case ID_FILE_EXIT:
    {
      // Завершаем работу приложения
      PostQuitMessage(0);
      return 0L;
        break;
    }

    case ID_HELP_ABOUT:
    {
      MessageBox(hWnd,
        "Multithread SDI Application with Mutex\n"
        "(C) Alexandr Frolov, 1996\n"
        "Email: frolov@glas.apc.org",
        szAppTitle, MB_OK | MB_ICONINFORMATION);
                  return 0L;
                  break;
    }
    default:
      break;
  }
  return FORWARD_WM_COMMAND(hWnd, id, hwndCtl, codeNotify,
    DefWindowProc);
}

// ------------------------------------------------------
// Функция задачи PaintEllipse
// ------------------------------------------------------
void PaintEllipse(void *hwnd)
{
  HDC hDC;
  RECT rect;
  LONG xLeft, xRight, yTop, yBottom;
  short nRed, nGreen, nBlue;
  HBRUSH hBrush, hOldBrush;
  DWORD dwRetCode;

  srand((unsigned int)hwnd);
  while(!fTerminate)
  {
    // Ожидаем, пока объект Mutex не перейдет в
    // отмеченное состояние
    dwRetCode = WaitForSingleObject(hMutex, INFINITE);

    // Если не было ошибок, выполняем рисование
    if(dwRetCode == WAIT_OBJECT_0)
    {
      hDC = GetDC(hwnd);

      nRed   = rand() % 255;
      nGreen = rand() % 255;
      nBlue  = rand() % 255;

      GetWindowRect(hwnd, &rect);

      xLeft   = rand() % (rect.left   + 1);
      xRight  = rand() % (rect.right  + 1);
      yTop    = rand() % (rect.top    + 1);
      yBottom = rand() % (rect.bottom + 1);

      hBrush = CreateSolidBrush(RGB(nRed, nGreen, nBlue));
      hOldBrush = SelectObject(hDC, hBrush);
      Ellipse(hDC, min(xLeft, xRight), min(yTop, yBottom),
                max(xLeft, xRight), max(yTop, yBottom));
      SelectObject(hDC, hOldBrush);
      DeleteObject(hBrush);
      ReleaseDC(hwnd, hDC);

      ReleaseMutex(hMutex);
    }
```

```
    // Если ожидание было отменено или произошла ошибка,
    // прерываем цикл
    else
    {
      break;
    }
    Sleep(100);
  }
}

// ------------------------------------------------------
// Функция задачи PaintRect
// ------------------------------------------------------
void PaintRect(void *hwnd)
{
  HDC hDC;
  RECT rect;
  LONG xLeft, xRight, yTop, yBottom;
  short nRed, nGreen, nBlue;
  HBRUSH hBrush, hOldBrush;
  DWORD dwRetCode;

  srand((unsigned int)hwnd + 1);
  while(!fTerminate)
  {
    dwRetCode = WaitForSingleObject(hMutex, INFINITE);
    if(dwRetCode == WAIT_OBJECT_0)
    {
      hDC = GetDC(hwnd);
      nRed   = rand() % 255;
      nGreen = rand() % 255;
      nBlue  = rand() % 255;
      GetWindowRect(hwnd, &rect);
      xLeft   = rand() % (rect.left   + 1);
      xRight  = rand() % (rect.right  + 1);
      yTop    = rand() % (rect.top    + 1);
      yBottom = rand() % (rect.bottom + 1);
      hBrush = CreateSolidBrush(RGB(nRed, nGreen, nBlue));
      hOldBrush = SelectObject(hDC, hBrush);
      Rectangle(hDC, min(xLeft, xRight), min(yTop, yBottom),
                max(xLeft, xRight), max(yTop, yBottom));
      SelectObject(hDC, hOldBrush);
      DeleteObject(hBrush);
      ReleaseDC(hwnd, hDC);
      ReleaseMutex(hMutex);
    }
    else
    {
      break;
    }
    Sleep(100);
  }
}

// ------------------------------------------------------
// Функция задачи PaintText
// ------------------------------------------------------
void PaintText(void *hwnd)
{
  HDC hDC;
  RECT rect;
  LONG xLeft, xRight, yTop, yBottom;
  short nRed, nGreen, nBlue;
  DWORD dwRetCode;

  srand((unsigned int)hwnd + 2);
  while(!fTerminate)
  {
    dwRetCode = WaitForSingleObject(hMutex, INFINITE);
    if(dwRetCode == WAIT_OBJECT_0)
    {
      hDC = GetDC(hwnd);
```

```
    GetWindowRect(hwnd, &rect);
    xLeft   = rand() % (rect.left   + 1);
    xRight  = rand() % (rect.right  + 1);
    yTop    = rand() % (rect.top    + 1);
    yBottom = rand() % (rect.bottom + 1);
    nRed   = rand() % 255;
    nGreen = rand() % 255;
    nBlue  = rand() % 255;
    SetTextColor(hDC, RGB(nRed, nGreen, nBlue));
    nRed   = rand() % 255;
    nGreen = rand() % 255;
    nBlue  = rand() % 255;
    SetBkColor(hDC, RGB(nRed, nGreen, nBlue));
    TextOut(hDC, xRight - xLeft,
      yBottom - yTop,"TEXT", 4);
    ReleaseDC(hwnd, hDC);
    ReleaseMutex(hMutex);
   }
   else
   {
    break;
   }
   Sleep(100);
  }
}
```

В области глобальных переменных определен массив szMutexName, в котором хранится имя объекта Mutex, используемого нашим приложением. Идентификатор этого объекта после его создания будет записан в глобальную переменную hMutex.

Функция WinMain создает объект Mutex, вызывая для этого функцию CreateMutex, как это показано ниже:

```
hMutex = CreateMutex(NULL, FALSE, szMutexName);
if(hMutex == NULL)
{
 . . .
 return 0l;
}
```

В качестве атрибутов защиты передается значение NULL. Так как второй параметр функции CreateMutex равен FALSE, объект Mutex после создания будет находиться в отмеченном состоянии.

Если при создании объекта функция CreateMutex не вернула признак ошибки (значение NULL), приложение проверяет, действительно ли был создан новый объект Mutex или же функция вернула идентификатор для уже существующего в системе объекта с таким же именем. Проверка выполняется с помощью функции GetLastError:

```
if(GetLastError() == ERROR_ALREADY_EXISTS)
{
 MessageBox(NULL, "MutexSDI already started",
    szAppTitle, MB_OK | MB_ICONEXCLAMATION);
 return 0l;
}
```

Если эта функция вернула значение ERROR_ALREADY_EXISTS, значит была запущена копия этого приложения, которая и создала объект Mutex с именем szMutexName. В этом случае приложение выводит сообщение и завершает свою работу.

В нашем приложении четыре задачи могут рисовать в главном окне. Это главная задача процесса и три задачи, созданные главной задачей.

Главная задача выполняет рисование при обработке сообщения WM_PAINT. Обработкой этого сообщения занимается функция WndProc_OnPaint.

Перед тем как получить контекст отображения и приступить к рисованию, эта функция вызывает функцию WaitForSingleObject, пытаясь стать владельцем объекта Mutex:

```
dwRetCode = WaitForSingleObject(hMutex, INFINITE);
```

Если никакая другая задача в данный момент не претендует на этот объект и, следовательно, не собирается рисовать в окне приложения, функция WaitForSingleObject немедленно возвращает управление с кодом возврата, равным WAIT_OBJECT_0. После этого функция WndProc_OnPaint получает контекст отображения, выполняет рисование, освободает контекст отображения и затем переводит объект Mutex в неотмеченное состояние, вызывая функцию ReleaseMutex:

```
ReleaseMutex(hMutex);
```

После этого другие задачи, выполняющие ожидание для объекта Mutex, могут продолжить свою работу.

В приложении определены три функции задач, выполняющих рисование в окне приложения. Синхронизация всех этих функций между собой и с главной задачей процесса выполняется одинаковым способом. Все эти задачи

выполняют в цикле ожидание события для объекта Mutex с идентификатором hMutex:

```
while(!fTerminate)
{
  dwRetCode = WaitForSingleObject(hMutex, INFINITE);
  if(dwRetCode == WAIT_OBJECT_0)
  {
    hDC = GetDC(hwnd);
     . . .
    Ellipse(hDC, min(xLeft, xRight), min(yTop, yBottom),
              max(xLeft, xRight), max(yTop, yBottom));
     . . .
    ReleaseDC(hwnd, hDC);
    ReleaseMutex(hMutex);
  }
  else
    break;
  Sleep(100);
}
```

Если ни одна задача не владеет объектом Mutex, функция WaitForSingleObject возвращает значение WAIT_OBJECT_0. При этом задача выполняет рисование в окне приложения. В том случае когда какая-либо другая задача владеет объектом Mutex, данная задача перейдет в состояние ожидания. Если при ожидании произошла ошибка, цикл завершает свою работу.

После рисования задача переводит объект Mutex в неотмеченное состояние с помощью функции ReleaseMutex.

Кратко перечислим другие файлы, имеющие отношение к приложению MutexSDI.

В файле mutexsdi.h (листинг 4.4) находятся прототипы функций, определенных в приложении.

Листинг 4.4. Файл mutexsdi/mutexsdi.h

```
LRESULT WINAPI
WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam);
BOOL WndProc_OnCreate(HWND hWnd,
  LPCREATESTRUCT lpCreateStruct);
void WndProc_OnDestroy(HWND hWnd);
void WndProc_OnPaint(HWND hWnd);
void WndProc_OnCommand(HWND hWnd, int id,
  HWND hwndCtl, UINT codeNotify);

void PaintEllipse(void *hwnd);
void PaintRect(void *hwnd);
void PaintText(void *hwnd);
```

Файл resource.h (листинг 4.5) создается автоматически и содержит определения констант для файла ресурсов приложения.

Листинг 4.5. Файл mutexsdi/resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by MutexSDI.RC
//
#define IDR_APPMENU                 102
#define IDI_APPICON                 103
#define IDI_APPICONSM               104
#define ID_FILE_EXIT                40001
#define ID_HELP_ABOUT               40003
#define ID_FORMAT_BOLD              40010
#define ID_FORMAT_ITALIC            40011
#define ID_FORMAT_UNDERLINE         40012
#define ID_FORMAT_PARAGRAPH_LEFT    40014
#define ID_FORMAT_PARAGRAPH_RIGHT   40015
#define ID_FORMAT_PARAGRAPH_CENTER  40016
#define ID_EDIT_DELETE              40021
#define ID_FILE_SAVEAS              40024
#define ID_EDIT_SELECTALL           40028
#define ID_SETPROTECTION_PAGENOACCESS   40035
#define ID_SETPROTECTION_PAGEREADONLY   40036
#define ID_SETPROTECTION_PAGEREADWRITE  40037
#define ID_SETPROTECTION_PAGEGUARD      40038
#define ID_MEMORY_READ              40039
#define ID_MEMORY_WRITE             40040
#define ID_MEMORY_LOCK              40041
#define ID_MEMORY_UNLOCK            40042
```

```
// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NEXT_RESOURCE_VALUE        121
#define _APS_NEXT_COMMAND_VALUE         40043
#define _APS_NEXT_CONTROL_VALUE         1000
#define _APS_NEXT_SYMED_VALUE           101
#endif
#endif
```

И, наконец, файл ресурсов приложения mutexsdi.rc приведен в листинге 4.6.

Листинг 4.6. Файл mutexsdi/mutexsdi.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 2 resource.
//
#include "afxres.h"
////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

////////////////////////////////////////////////////////////
// English (U.S.) resources
#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

////////////////////////////////////////////////////////////
// Menu
//

IDR_APPMENU MENU DISCARDABLE
BEGIN
    POPUP "&File"
    BEGIN
        MENUITEM "E&xit",                       ID_FILE_EXIT
    END
    POPUP "&Help"
    BEGIN
        MENUITEM "&About...",                   ID_HELP_ABOUT
    END
END

#ifdef APSTUDIO_INVOKED
////////////////////////////////////////////////////////////
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END
```

```
#endif    // APSTUDIO_INVOKED

//////////////////////////////////////////////////////////////
// Icon
//

// Icon with lowest ID value placed first to ensure
// application icon
// remains consistent on all systems.
IDI_APPICON             ICON    DISCARDABLE     "mutexsdi.ico"
IDI_APPICONSM           ICON    DISCARDABLE     "mutexssm.ico"

//////////////////////////////////////////////////////////////
// String Table
//

STRINGTABLE DISCARDABLE
BEGIN
    ID_FILE_EXIT            "Quits the application"
END

#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
// Generated from the TEXTINCLUDE 3 resource.
//
//////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Синхронизация с использованием семафоров

Последнее средство, предназначенное для синхронизации задач, которое мы рассмотрим в нашей книге, это объекты-семафоры. В отличие от объектов Mutex, которые используются для организации последовательного доступа задач к ресурсу, семафоры позволяют обеспечить доступ к ресурсу для заранее определенного, ограниченного приложением количества задач. Все остальные задачи, пытающиеся получить доступ сверх установленного лимита, будут переведены при этом в состояние ожидания до тех пор, пока какая либо задача, получившая доступ к ресурсу раньше, не освободит ресурс, связанный с данным семафором.

Примером области применения семафоров может послужить программное обеспечение какого-либо аппаратного устройства, с которым может работать только ограниченное количество задач. Все остальные задачи, пытающиеся получить доступ к этому устройству, должны быть переведены в состояние ожидания и пребывать в нем до тех пор, пока не завершит свою работу одна из задач, уже получившая доступ к устройству.

Ресурс, доступ к которому управляется семафором, может быть не только физическим устройством, но и чисто логическим.

Поясним это на примере.

Раньше в нашей книге мы приводили исходные тексты приложения MultiMDI. В этом приложении пользователь мог создавать MDI-окна, для каждого из которых запускалась отдельная задача, выполняющая рисование эллипсов произвольной формы и цвета. Сколько окон создаст пользователь, столько и будет запущено задач. Все эти задачи будут работать параллельно.

Предположим теперь, что нам нужно изменить приложение таким образом, чтобы пользователь мог по-прежнему создавать любое количество MDI-окон и соответствующих им задач, но чтобы при этом в каждый момент работало только ограниченное количество задач, например, не больше трех. Для этого мы создаем семафор, котрый будет использован всеми запускаемыми задачами и который будет допускать работу только трех задач.

В этом случае если пользователь, например, создаст пять MDI-окон, в трех из них будут хаотически отображаться эллипсы, а содержимое остальных не будет перерисовываться, так как соответствующие задачи будут находиться в состоянии ожидания.

Если теперь в нашем новом варианте приложения пользователь закроет MDI-окно, в котором выполняется рисование, то одно из двух “спящих” MDI-окон должно “проснутся”. При этом в нем начнется процесс рисования эллипсов. Если закрыть еще одно активное окно, процесс рисования должен начаться во втором “спящем” окне.

Таким образом, сколько бы MDI-окон не создал пользователь, эллипсы будут отображаться только в трех из них, так как только три задачи будут работать, а остальные будут находиться в состоянии ожидания.

Для реализации такого приложения нам нужен объект синхронизации, который имеет в своем составе счетчик задач, получивших доступ к этому объекту. Этим объектом синхронизации может послужить семафор. Что же касается приложения, ведущего себя подобным образом, то далее в этой главе мы приведем его исходные тексты.

Как работает семафор

В отличие от железнодорожного семафора, который может быть либо открыт, разрашая движение, либо закрыт, запрещая его, семаформы в операционной системе Microsoft Windows NT действуют более сложным образом.

Так же как и объект Mutex, семафор может находиться в отмеченном или неотмеченном состоянии. Приложение выполняет ожидание для семафора при помощи таких функций, как WaitForSingleObject или WaitForMultipleObject (точно также, как и для объекта Mutex). Если семафор находится в неотмеченном состоянии, задача, вызвавшая для него функцию WaitForSingleObject, находится в состоянии ожидания. Когда же состояние семафора становится отмеченным, работа задачи возобновляется. В такой логике работы для вас, однако, нет ничего нового.

В отличие от объекта Mutex, с каждым семафором связывается счетчик, начальное и максимальные значения которого задаются при создании семафора. Значение этого счетчика уменьшается, когда задача вызывает для семафора функцию WaitForSingleObject или WaitForMultipleObject, и увеличивается при вызове другой, специально предназначенной для этого функции.

Если значение счетчика семафора равно нулю, он находится в неотмеченном состоянии. Если же это значение больше нуля, семафор переходит в отмеченное состояние.

Пусть, например, приложение создало семафор, указав для него максимальное значение счетчика, равное трем. Пусть начальное значение этого счетчика также будет равно трем.

Если в этой ситуации несколько запускаемых по очереди задач будут выполнять с помощью функции WaitForSingleObject ожидание семафора, то первые три запущенные задачи будут работать, а все остальные перейдут в состояние ожидания. Это связано с тем, что первые три вызова функции WaitForSingleObject приведут к последовательному уменьшению значения счетчика семафора до нуля, в результате чего семафор переключится в неотмеченное состояние.

Задача, запущенная четвертой, вызовет функцию WaitForSingleObject для неотмеченного семафора, в результате чего она будет ждать. Точно также, задачи, запущенные после запуска четвертой задачи, будут выполнять ожидание для того же семафора.

Как долго продлится ожидание?

До тех пор, пока одна из первых трех задач не освободит семафор, увеличив его счетчик на единицу вызовом специальной функции. Сразу после этого будет запущена одна из задач, ожидающих наш семафор.

На рис. 4.4 мы показали последовательное изменение счетчика семафора (обозначенного символом N) при запуске первых трех задач. Так как счетчик больше нуля, семафор открыт, то есть находится в отмеченном состоянии и поэтому функция WaitForSingleObject не будет выполнять ожидание.

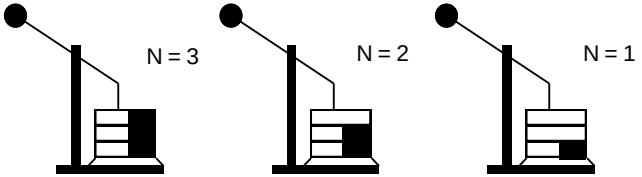

Рис. 4.4. Изменение значения счетчика семафора от трех до единицы

После запуска четвертой задачи, выполняющей ожидание семафора, значение счетчика уменьшится до нуля. При этом семафор будет закрыт, то есть перейдет в неотмеченное состояние (рис. 4.5).

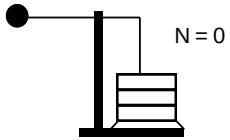

Рис. 4.5. После запуска четвертой задачи семафор закрывается

Функции для работы с семафорами

Рассмотрим функции программного интерфейса операционной системы Microsoft Windows NT, предназначенные для работы с семафорами.

Создание семафора

Для создания семафора приложение должно вызвать функцию CreateSemaphore, прототип которой приведен ниже:

```
HANDLE CreateSemaphore(
  LPSECURITY_ATTRIBUTES lpSemaphoreAttributes, // атрибуты
                                               // защиты
  LONG lInitialCount,  // начальное значение счетчика семафора
  LONG lMaximumCount,  // максимальное значение
                       // счетчика семафора
  LPCTSTR  lpName);    // адрес строки с именем семафора
```

Этой функцией пользоваться достаточно просто.

В качестве атрибутов защиты вы можете передать значение NULL.

Через параметры lInitialCount и lMaximumCount передается, соответственно, начальное и максимальное значение счетчика, связанного с

создаваемым семафором. Начальное значение счетчика lInitialCount должно быть больше или равно нулю и не должно превосходить максимальное значение счетчика, передаваемое через параметр lMaximumCount.

Имя семафора указывается аналогично имени рассмотренного нами ранее объекта Mutex с помощью параметра lpName.

В случае удачного создания семафора функция CreateSemaphore возвращает его идентификатор. В случае возникновения ошибки возвращается значение NULL, при этом код ошибки можно узнать при помощи функции GetLastError.

Так как имена семафоров доступны всем приложениям в системе, возможно возникновение ситуации, когда приложение пытается создать семафор с уже использованным именем. При этом новый семафор не создается, а приложение получает идентификатор для уже существующего семафора. Если возниклав такая ситуация, функция GetLastError, вызванная сразу после функции CreateSemaphore, возвращает значение ERROR_ALREADY_EXISTS.

Уничтожение семафора

Для уничтожения семафора вы должны передать его идентификатор функции CloseHandle. Заметим, что при завершении процесса все созданные им семафоры уничтожаются автоматически.

Открывание семафора

Если семафор используется только для синхронизации задач, созданных в рамках одного приложения, вы можете создать безымянный семафор, указав в качестве параметра lpName функции CreateSemaphore значение NULL. В том случае, когда необходимо синхронизовать задачи разных процессов, следует определить имя семафора. При этом один процесс создает семафор с помощью функции CreateSemaphore, а второй открывает его, получая идентификатор для уже существующего семафора.

Существующий семафор можно открыть функцией OpenSemaphore, прототип которой приведен ниже:

```
HANDLE OpenSemaphore(
  DWORD   fdwAccess,         // требуемый доступ
  BOOL    fInherit,          // флаг наследования
  LPCTSTR lpszSemaphoreName ); // адрес имени семафора
```

Флаги доступа, передаваемые через параметр fdwAccess, определяют требуемый уровень доступа к семафору. Этот параметр может быть комбинацией следующих значений:

Значение	*Описание*
SEMAPHORE_ALL_ACCESS	Указаны все возможные флаги доступа
SEMAPHORE_MODIFY_STATE	Возможно изменение значение счетчика семафора функцией ReleaseSemaphore
SYNCHRONIZE	Полученный идентификатор можно будет использовать в любых функциях ожидания события

Параметр fInherit определяет возможность наследования полученного идентификатора. Если этот параметр равен TRUE, идентификатор может наследоваться дочерними процессами. Если же он равен FALSE, наследование не допускается.

Через параметр lpszSemaphoreName вы должны передать функции адрес символьной строки, содержащей имя семафора.

Если семафор открыт успешно, функция OpenSemaphore возвращает его идентификатор. При ошибке возвращается значение NULL. Код ошибки вы можете определить при помощи фукнции GetLastError.

Увеличение значения счетчика семафора

Для увеличения значения счетчика семафора приложение должно использовать функцию ReleaseSemaphore:

```
BOOL ReleaseSemaphore(
  HANDLE hSemaphore,        // идентификатор семафора
  LONG   cReleaseCount,     // значение инкремента
  LPLONG lplPreviousCount); // адрес переменной для записи
                            // предыдущего значения счетчика семафора
```

Функция ReleaseSemaphore увеличивает значение счетчика семафора, идентификатор которого передается ей через параметр hSemaphore, на значение, указанное в параметре cReleaseCount.

Заметим, что через параметр cReleaseCount вы можете передавать только положительное значение, большее нуля. При этом если в результате увеличения новое значение счетчика должно будет превысить максимальное значение, заданное при создании семафора, функция ReleaseSemaphore возвращает признак ошибки и не изменяет значение счетчика.

Предыдущее значение счетчика, которое было до использования функции ReleaseSemaphore, записывается в переменную типа LONG. Адрес этой переменной передается функции через параметр lplPreviousCount.

Если функция ReleaseSemaphore завершилась успешно, она возвращает значение TRUE. При ошибке возвращается значение FALSE. Код ошибки в этом случае можно определить, как обычно, при помощи функции GetLastError.

Функция используется обычно для решения двух задач.

Во-первых, с помощью этой функции задачи освобождают ресурс, доступ к которому регулируется семафором. Они могут делать это после использования ресурса или перед своим завершением.

Во-вторых, эта функция может быть использована на этапе инициализации мультизадачного приложения. Создавая семафор с начальным значением счетчика, равным нулю, главная задача блокирует работу задач, выполняющих ожидание этого семафора. После завершения инициализации главная задача с помощью функции ReleaseSemaphore может увлеичить значение счетчика семафора до максимального, в результате чего известное количество ожидающих задач будет активизировано.

Уменьшение значения счетчика семафора

В программном интерфейсе операционной системы Microsoft Windows NT нет фукнции, специально предназначенной для уменьшения значения счетчика семафора. Этот счетчик уменьшается, когда задача вызывает функции ожидания, такие как WaitForSingleObject или WaitForMultipleObject. Если задача вызывает несколько раз функцию ожидания для одного и того же семафора, содержимое его счетчика каждый раз будет уменьшаться.

Определение текущего значения счетчика семафора

Единаственная возможность определения текущего значения счетчика семафора заключается в увеличении этого значения функцией ReleaseSemaphore. Значение счетчика, которое было до увеличения, будет записано в переменную, адрес которой передается функции ReleaseSemaphore через параметр lplPreviousCount.

Заметим, что в операционной системе Microsoft Windows NT не предусмотрено средств, с помощью которых можно было бы определить текущее значение семафора, не изменяя его. В частности, вы не можете задать функции ReleaseSemaphore нулевое значение инкремента, так как в этом случае указанная функция просто вернет соответствующий код ошибки.

Приложение SEMMDI

Приложение SEMMDI создано на базе приложения MultiMDI и демонстрирует использование семфафоров для ограничения количества работающих задач, запущенных для MDI-окон.

На рис. 4.6 показано главное окно приложения SEMMDI. Мы запустили это приложение в среде операционной системы Microsoft Windows 95, так как она также является мультизадачной операционной системой и может работать с семафорами и другими объектами синхронизации. Разумеется, что в среде Microsoft Windows NT приложение SEMMDI также будет работать.

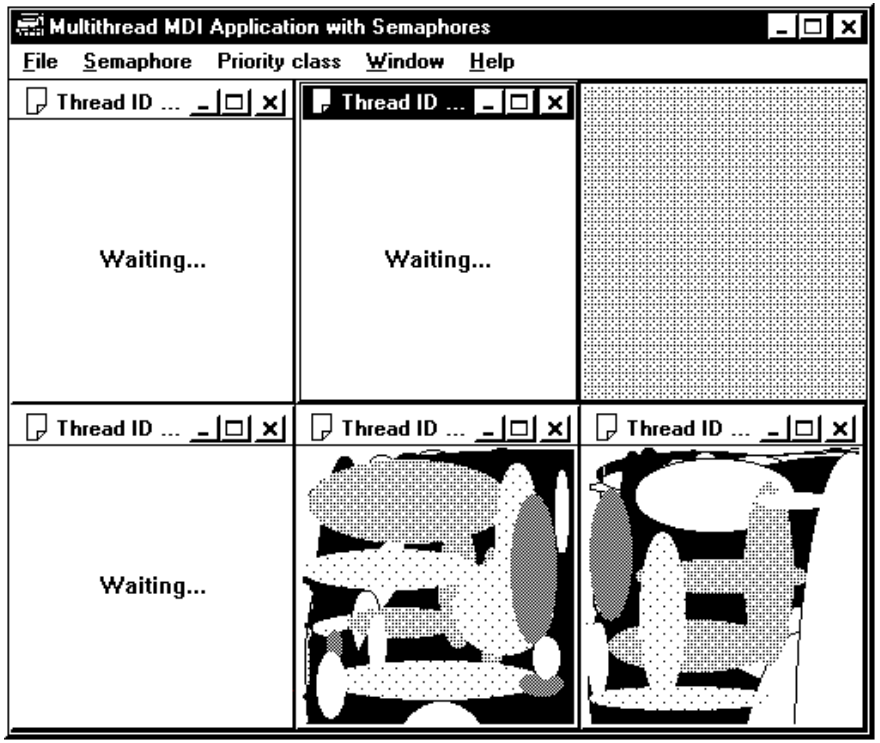

Рис. 4.6. Главное окно приложения SEMMDI, запущенного в среде операционной системы Microsoft Windows 95

Выбирая из меню File строку New, вы можете создавать новые MDI-окна. В первых двух созданных вами окнах начнется процесс рисования эллипсов случайных размеров и цвета. В остальных окнах отображается только строка Waiting.

Если теперь закрыть одно из двух работающих окон (в которых идет рисование эллипсов), одно из ожидающих окон “просыпается” и в нем начинается процесс рисования. Таким образом, сколько бы вы ни создали MDI-окон, рисование будет выполняться только в двух из них.

Выбирая из меню Semaphore строку Increment, вы можете увеличивать значение семафора на единицу. При этом максимальное количество окон, в которых идент процесс рисования, также будет увеличиваться.

Заметим, что ожидающие окна нельзя удалить. Все остальные возможности приложения MultiMDI сохранены. В частности, если сделать

щелчок правой клавишей мыши в MDI-окне, на экране появится плавающее меню, с помощью которого можно выполнять над этим окном различные операции.

Исходные тексты приложения

Главный файл исходных текстов приложения SEMMDI представлен в листинге 4.7. Заметим, что для сборки проекта необходимо использовать мультизадачный вариант библиотеки времени выполнения.

Листинг 4.7. Файл semmdi/semmdi.c

```
#define STRICT
#include <windows.h>
#include <windowsx.h>
#include <stdio.h>
#include "resource.h"
#include "afxres.h"
#include "semmdi.h"

// Имена классов окна
char const szFrameClassName[] = "MDISemAppClass";
char const szChildClassName[] = "MDISemChildAppClass";

// Заголовок окна
char const szWindowTitle[] =
  "Multithread MDI Application with Semaphores";

HINSTANCE hInst;

HWND hwndFrame;  // окно Frame Window
HWND hwndClient; // окно Client Window
HWND hwndChild;  // окно Child Window

// Структура, которая создается для каждого дочернего окна
typedef struct _CHILD_WINDOW_TAG
{
  // Признак активности задачи
  BOOL fActive;

  // Флаг ожидания семафора
  BOOL fWaiting;

  // Критическая секция для рисования в окне
  CRITICAL_SECTION csChildWindowPaint;

  // Идентификатор задачи
  HANDLE hThread;
} CHILD_WINDOW_TAG;

typedef CHILD_WINDOW_TAG *LPCHILD_WINDOW_TAG;

// Семафор для ограничения количества работающих задач
HANDLE hSemaphore;

// Имя семафора
char const lpSemaphoreName[] =
 "$MyVerySpecialSemaphoreName$For$SemMDI$Application$";

// ====================================
// Функция WinMain
// ====================================
int APIENTRY
WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
        LPSTR lpCmdLine, int nCmdShow)
{
  MSG  msg;         // структура для работы с сообщениями
  hInst = hInstance; // сохраняем идентификатор приложения

  // Инициализируем приложение
  if(!InitApp(hInstance))
    return FALSE;

  // Создаем семафор для ограничения количества
  // задач, работающих одновременно
```

```
hSemaphore = CreateSemaphore(NULL, 2, 2, lpSemaphoreName);
if(hSemaphore == NULL)
{
  MessageBox(NULL,
    "Ошибка при создании семафора",
     szWindowTitle, MB_OK | MB_ICONEXCLAMATION);
  return FALSE;
}

// Создаем главное окно приложения - Frame Window
hwndFrame = CreateWindow(
  szFrameClassName,    // имя класса окна
  szWindowTitle,       // заголовок окна
  WS_OVERLAPPEDWINDOW, // стиль окна
  CW_USEDEFAULT, 0,    // задаем размеры и расположение
  CW_USEDEFAULT, 0,    // окна, принятые по умолчанию
  0,                   // идентификатор родительского окна
  0,                   // идентификатор меню
  hInstance,           // идентификатор приложения
  NULL);   // указатель на дополнительные параметры

// Если создать окно не удалось, завершаем приложение
if(!hwndFrame)
  return FALSE;

// Рисуем главное окно
ShowWindow(hwndFrame, nCmdShow);
UpdateWindow(hwndFrame);

// Запускаем цикл обработки сообщений
while(GetMessage(&msg, NULL, 0, 0))
{
  // Трансляция для MDI-приложения
  if(!TranslateMDISysAccel(hwndClient, &msg))
  {
    TranslateMessage(&msg);
    DispatchMessage(&msg);
  }
 }

 // Освобождаем идентификатор семафора
 CloseHandle(hSemaphore);
 return msg.wParam;
}

// ====================================
// Функция InitApp
// Выполняет регистрацию класса окна
// ====================================

BOOL InitApp(HINSTANCE hInstance)
{
  ATOM aWndClass; // атом для кода возврата
  WNDCLASS wc;    // структура для регистрации

  // Регистрируем класс для главного окна приложения
  // (для окна Frame Window)
  memset(&wc, 0, sizeof(wc));
  wc.lpszMenuName  = "APP_MENU";
  wc.style         = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc   = (WNDPROC)FrameWndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon(hInstance, "APP_ICON");
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_APPWORKSPACE + 1);
  wc.lpszClassName = (LPSTR)szFrameClassName;
  aWndClass = RegisterClass(&wc);

  if(!aWndClass)
    return FALSE;
```

```
  // Регистрируем класс окна для
  //дочернего окна Document Window
  memset(&wc, 0, sizeof(wc));
  wc.lpszMenuName  = 0;
  wc.style         = CS_HREDRAW | CS_VREDRAW;
  wc.lpfnWndProc   = (WNDPROC)ChildWndProc;
  wc.cbClsExtra    = 0;
  wc.cbWndExtra    = 0;
  wc.hInstance     = hInstance;
  wc.hIcon         = LoadIcon(hInstance, "APPCLIENT_ICON");
  wc.hCursor       = LoadCursor(NULL, IDC_ARROW);
  wc.hbrBackground = (HBRUSH)(COLOR_WINDOW + 1);
  wc.lpszClassName = (LPSTR)szChildClassName;
  aWndClass = RegisterClass(&wc);

  if(!aWndClass)
    return FALSE;

  return TRUE;
}

// ====================================
// Функция FrameWndProc
// ====================================

LRESULT CALLBACK
FrameWndProc(HWND hwnd, UINT msg,
             WPARAM wParam, LPARAM lParam)
{
  HWND   hwndChild;
  HANDLE hThread;
  DWORD  dwIDThread;
  CHAR   szBuf[255];
  LPCHILD_WINDOW_TAG lpMyWndTag;

  // Структура для создания окна Client Window
  CLIENTCREATESTRUCT clcs;

  // Указатель на структуру для хранения
  // состояния дочернего окна
  LPCHILD_WINDOW_TAG lpTag;

  switch (msg)
  {
    // ПРи создании окна Frame Window создаем
    // окно Client Window, внутри которого будут создаваться
    // дочерние окна Document Window
    case WM_CREATE:
    {
      clcs.hWindowMenu = GetSubMenu(GetMenu(hwnd), 2);
      clcs.idFirstChild = 500;

      // Создаем окно Client Window
      hwndClient = CreateWindow(
        "MDICLIENT",    // имя класса окна
        NULL,           // заголовок окна
        WS_CHILD | WS_CLIPCHILDREN | WS_VISIBLE |
          WS_HSCROLL | WS_VSCROLL,
        0, 0, 0, 0,
        hwnd,         // идентификатор родительского окна
        (HMENU)1,     // идентификатор меню
        hInst,        // идентификатор приложения
        (LPSTR)&clcs);// указатель на дополнительные параметры

      break;
    }

    // Обработка сообщений от главного меню приложения
    case WM_COMMAND:
    {
      switch (wParam)
```

```
{
 // Создание нового окна Document Window
 case CM_FILENEW:
 {
  hwndChild = CreateMDIWindow(
   (LPSTR)szChildClassName,    // класс окна
   "MDI Child Window",         // заголовок окна
   0,                          // дополнительные стили
   CW_USEDEFAULT, CW_USEDEFAULT, // размеры окна
   CW_USEDEFAULT, CW_USEDEFAULT, //  Document Window
   hwndClient,
   hInst,           // идентификатор приложения
   0);              // произвольное значение

  // Получаем память для структуры, в которой будет
  // хранится состояние окна
  lpTag = malloc(sizeof(CHILD_WINDOW_TAG));

  // Устанавливаем признак активности
  lpTag->fActive = 1;

  // Инициализируем критическую секцию
  InitializeCriticalSection(
    &(lpTag->csChildWindowPaint));

  // Устанавливаем адрес структуры состояния в
  // памяти окна
  SetWindowLong(hwndChild, GWL_USERDATA, (LONG)lpTag);

  // Создаем задачу для дочернего окна
  hThread = CreateThread(NULL, 0,
   (LPTHREAD_START_ROUTINE)ThreadRoutine,
   (LPVOID)hwndChild, 0,(LPDWORD)&dwIDThread);

  if(hThread == NULL)
  {
   MessageBox(hwnd,"Ошибка при создании задачи",
              szWindowTitle, MB_OK | MB_ICONEXCLAMATION);
  }

  // Сохраняем идентификатор созданной задачи
  lpTag->hThread = hThread;

  // Отображаем идентификатор задачи в заголовке
  // дочернего окна
  sprintf(szBuf, "Thread ID = %lX", dwIDThread);
  SetWindowText(hwndChild, szBuf);

  break;
 }

 // Увеличиваем содержимое счетчика семафора
 // на единицу
 case ID_SEMAPHORE_INCREMENT:
 {
  ReleaseSemaphore(hSemaphore, 1, NULL);
  break;
 }

 // Размещение окон Document Window рядом друг с другом
 case CM_WINDOWTILE:
 {
  SendMessage(hwndClient, WM_MDITILE, 0, 0);
  break;
 }

 // Размещение окон Document Window с перекрытием
 case CM_WINDOWCASCADE:
 {
  SendMessage(hwndClient, WM_MDICASCADE, 0, 0);
  break;
 }
```

```
// Размещение пиктограм минимизированых окон
// Document Window в нижней части окна Client Window
case CM_WINDOWICONS:
{
 SendMessage(hwndClient, WM_MDIICONARRANGE, 0, 0);
 break;
}

// Уничтожение всех окон Document Window
case CM_WINDOWCLOSEALL:
{
 HWND hwndTemp;

 ShowWindow(hwndClient, SW_HIDE);

 while(TRUE)
 {
  // Получаем идентификатор дочернего окна
  // для окна Client Window
  hwndTemp = GetWindow(hwndClient, GW_CHILD);
  // Если дочерних окон больше нет, выходим из цикла
  if(!hwndTemp)
   break;

  // Пропускаем окна-заголовки
  while(hwndTemp && GetWindow(hwndTemp, GW_OWNER))
   hwndTemp = GetWindow(hwndTemp, GW_HWNDNEXT);

  // Удаляем дочернее окно Document Window
  if(hwndTemp)
  {
   // Завершаем задачу, запущенную для окна
   lpMyWndTag =
     (LPCHILD_WINDOW_TAG)GetWindowLong(
     hwndTemp, GWL_USERDATA);
   lpMyWndTag->fActive = 0;

   // Посылаем сообщение, в ответ на которое
   // окно будет удалено
   SendMessage(hwndClient, WM_MDIDESTROY,
    (WPARAM)hwndTemp, 0);
  }
  else
   break;
 }

 // Отображаем окно Client Window
 ShowWindow(hwndClient, SW_SHOW);

 break;
}

// Устанавливаем классы приоритета процесса
case ID_PRIORITYCLASS_REALTIME:
{
 SetPriorityClass(GetCurrentProcess(),
  REALTIME_PRIORITY_CLASS);
 break;
}
case ID_PRIORITYCLASS_HIGH:
{
 SetPriorityClass(GetCurrentProcess(),
  HIGH_PRIORITY_CLASS);
 break;
}
case ID_PRIORITYCLASS_NORMAL:
{
 SetPriorityClass(GetCurrentProcess(),
  NORMAL_PRIORITY_CLASS);
 break;
}
```

```
  case ID_PRIORITYCLASS_IDLE:
  {
    SetPriorityClass(GetCurrentProcess(),
      IDLE_PRIORITY_CLASS);
    break;
  }

  case CM_HELPABOUT:
  {
    MessageBox(hwnd,
      "Демонстрация использования мультизадачности\n"
      "в MDI-приложениях\n"
      "(C) Alexandr Frolov, 1996\n"
      "Email: frolov@glas.apc.org",
      szWindowTitle, MB_OK | MB_ICONINFORMATION);
    break;
  }

  // Завершаем работу приложения
  case CM_FILEEXIT:
  {
    DestroyWindow(hwnd);
    break;
  }

  default:
    break;
}

// Определяем идентификатор активного окна
// Document Window
hwndChild =
  (HWND)LOWORD(SendMessage(hwndClient,
  WM_MDIGETACTIVE, 0, 0l));

// Если это окно, посылаем ему сообщение WM_COMMAND
    if(IsWindow(hwndChild))
      SendMessage(hwndChild, WM_COMMAND, wParam, lParam);

    return DefFrameProc(
      hwnd, hwndClient, msg, wParam, lParam);
  }

  case WM_DESTROY:
  {
    PostQuitMessage(0);
    break;
  }

  default:
    break;
  }
  return DefFrameProc(hwnd, hwndClient, msg, wParam, lParam);
}

// ====================================
// Функция ChildWndProc
// ====================================

LRESULT CALLBACK
ChildWndProc(HWND hwnd, UINT msg,
             WPARAM wParam, LPARAM lParam)
{
  HDC hdc;
  PAINTSTRUCT ps;
  RECT rc;
  LPCHILD_WINDOW_TAG lpMyWndTag;
  HMENU hmenuPopup;
  POINT pt;
  CHAR szBuf[256];

  switch (msg)
```

```
{
 case WM_PAINT:
 {
  // Получаем адрес структуры состояния окна
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(hwnd, GWL_USERDATA);

  // Входим в критическую секцию
  EnterCriticalSection(&(lpMyWndTag->csChildWindowPaint));

  // Перерисовываем внутреннюю область дочернего окна
  hdc = BeginPaint(hwnd, &ps);
  GetClientRect(hwnd, &rc);

  if(lpMyWndTag->fWaiting)
    DrawText(hdc, "Waiting...", -1, &rc,
      DT_SINGLELINE | DT_CENTER | DT_VCENTER);
  else
    DrawText(hdc, "Child Window. Running", -1, &rc,
      DT_SINGLELINE | DT_CENTER | DT_VCENTER);

  EndPaint(hwnd, &ps);

  // Выходим из критической секции
  LeaveCriticalSection(&(lpMyWndTag->csChildWindowPaint));
  break;
 }

 case WM_CLOSE:
 {
  // Сбрасываем признак активности задачи
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(hwnd, GWL_USERDATA);

  // Если задача находится в состоянии ожидания
  // семафора, запрещаем удаление окна
  if(lpMyWndTag->fWaiting)
    return 0;

  // Если задача не ожидает семафор, завершаем
  // задачу и разрешаем удаление окна
  else
    lpMyWndTag->fActive = 0;

  break;
 }

// Когда пользователь нажимает правую кнопку мыши
// в дочернем окне, отображаем плавающее меню
case WM_RBUTTONDOWN:
{
  pt.x = LOWORD(lParam);
  pt.y = HIWORD(lParam);
  ClientToScreen(hwnd, &pt);

  hmenuPopup = GetSubMenu(
    LoadMenu(hInst, "IDR_POPUPMENU"), 0);
  TrackPopupMenu(hmenuPopup,
    TPM_CENTERALIGN | TPM_LEFTBUTTON,
    pt.x, pt.y, 0, hwnd, NULL);
  DestroyMenu(hmenuPopup);
  break;
}

// Обрабатываем команды, поступающие от плавающего меню
case WM_COMMAND:
{
  switch (wParam)
  {
    // Приостановка выполнения задачи
    case ID_THREADCONTROL_SUSPEND:
    {
```

```
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);

  // Входим в критическую секцию
  EnterCriticalSection(
     &(lpMyWndTag->csChildWindowPaint));

  SuspendThread(lpMyWndTag->hThread);

  // Выходим из критической секции
  LeaveCriticalSection(
    &(lpMyWndTag->csChildWindowPaint));

  break;
}

// Возобновление выполнения задачи
case ID_THREADCONTROL_RESUME:
{

  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);
  ResumeThread(lpMyWndTag->hThread);
  break;
}

// Изменение относительного приоритета
case ID_THREADCONTROL_PRIORITYLOWEST:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);
  SetThreadPriority(lpMyWndTag->hThread,
    THREAD_PRIORITY_LOWEST);
  break;
}
case ID_THREADCONTROL_PRIORITYNORMAL:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);
  SetThreadPriority(lpMyWndTag->hThread,
    THREAD_PRIORITY_NORMAL);
  break;
}
case ID_THREADCONTROL_PRIORITYHIGHEST:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);
  SetThreadPriority(lpMyWndTag->hThread,
    THREAD_PRIORITY_HIGHEST);
  break;
}

// Определение и отображение относительного приоритета
case ID_THREADCONTROL_GETPRIORITY:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
    hwnd, GWL_USERDATA);

  strcpy(szBuf, "Thread priority: ");
  switch (GetThreadPriority(lpMyWndTag->hThread))
  {
    case THREAD_PRIORITY_LOWEST:
    {
      strcat(szBuf, "THREAD_PRIORITY_LOWEST");
      break;
    }
```

```
    case THREAD_PRIORITY_BELOW_NORMAL:
    {
      strcat(szBuf, "THREAD_PRIORITY_BELOW_NORMAL");
      break;
    }
    case THREAD_PRIORITY_NORMAL:
    {
      strcat(szBuf, "THREAD_PRIORITY_NORMAL");
      break;
    }
    case THREAD_PRIORITY_ABOVE_NORMAL:
    {
      strcat(szBuf, "THREAD_PRIORITY_ABOVE_NORMAL");
      break;
    }
    case THREAD_PRIORITY_HIGHEST:
    {
      strcat(szBuf, "THREAD_PRIORITY_HIGHEST");
      break;
    }
   }

          MessageBox(hwnd, szBuf,
    szWindowTitle, MB_OK | MB_ICONINFORMATION);
   break;
}

// Удаление задачи
case ID_THREADCONTROL_KILLTHREAD:
{
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(
     hwnd, GWL_USERDATA);
  TerminateThread(lpMyWndTag->hThread, 5);
  break;
}

      // Уничтожение дочернего окна
      case ID_THREADCONTROL_CLOSEWINDOW:
      {
        lpMyWndTag =
          (LPCHILD_WINDOW_TAG)GetWindowLong(
          hwnd, GWL_USERDATA);

        // Если задача не находится в состоянии ожидания
        // семафора, удаляем окно и завершаем работу
        // задачи, запущенной для него
        if(!lpMyWndTag->fWaiting)
        {
          SendMessage(hwndClient, WM_MDIDESTROY,
            (WPARAM)hwnd, 0);

          lpMyWndTag->fActive = 0;
        }
        break;
      }
      default:
        break;
    }
    break;
   }
   default:
    break;
  }
  return DefMDIChildProc(hwnd, msg, wParam, lParam);
}

// ====================================
// Функция ThreadRoutine
// Задача, которая выполняется для каждого
// дочернего окна
// ====================================
```

```
DWORD ThreadRoutine(HWND hwnd)
{
  LONG lThreadWorking = 1L;
  HDC hDC;
  RECT rect;
  LONG xLeft, xRight, yTop, yBottom;
  short nRed, nGreen, nBlue;
  HBRUSH hBrush, hOldBrush;
  LPCHILD_WINDOW_TAG lpMyWndTag;

  // Получаем указатель на структуру параметров окна
  lpMyWndTag =
    (LPCHILD_WINDOW_TAG)GetWindowLong(hwnd, GWL_USERDATA);

  // Если произошла ошибка, завершаем работу задачи
  if(lpMyWndTag == NULL)
  {
    return 0;
  }

  // Устанавливаем флаг ожидания семафора
  lpMyWndTag->fWaiting = TRUE;

  // Выполняем ожидание семафора. Если оно завершилось
  // с ошибкой, завершаем работу задачи
  if(WAIT_FAILED ==
    WaitForSingleObject(hSemaphore, INFINITE))
  {
    return 0;
  }

  // Сбрасываем флаг ожидания семафора
  lpMyWndTag->fWaiting = FALSE;

  srand((unsigned int)hwnd);

  while(TRUE)
  {
    if(!lpMyWndTag->fActive)
      break;

    EnterCriticalSection(&(lpMyWndTag->csChildWindowPaint));

    hDC = GetDC(hwnd);
    nRed   = rand() % 255;
    nGreen = rand() % 255;
    nBlue  = rand() % 255;

    GetWindowRect(hwnd, &rect);
    xLeft   = rand() % (rect.left   + 1);
    xRight  = rand() % (rect.right  + 1);
    yTop    = rand() % (rect.top    + 1);
    yBottom = rand() % (rect.bottom + 1);

    hBrush = CreateSolidBrush(RGB(nRed, nGreen, nBlue));
    hOldBrush = SelectObject(hDC, hBrush);

    Ellipse(hDC, min(xLeft, xRight), min(yTop, yBottom),
              max(xLeft, xRight), max(yTop, yBottom));

    SelectObject(hDC, hOldBrush);
    DeleteObject(hBrush);
    ReleaseDC(hwnd, hDC);

    LeaveCriticalSection(&(lpMyWndTag->csChildWindowPaint));
    Sleep(1);
  }

  DeleteCriticalSection(&(lpMyWndTag->csChildWindowPaint));
  free(lpMyWndTag);
```

```
  // Перед завершением работы задачи увеличиваем
  // на единицу счетчик семафора, разрешая работу
  // других задач, находящихся в состоянии ожидания
  if(hSemaphore != NULL)
    ReleaseSemaphore(hSemaphore, 1, NULL);

  return 0;
}
```

Файл semmdi.h (листинг 4.8) содержит прототипы функций, определенных в приложении SEMMDI.

Листинг 4.8. Файл semmdi/semmdi.h

```
// Прототипы функций
BOOL InitApp(HINSTANCE);
LRESULT CALLBACK FrameWndProc(HWND, UINT, WPARAM, LPARAM);
LRESULT CALLBACK ChildWndProc(HWND, UINT, WPARAM, LPARAM);
DWORD ThreadRoutine(HWND hwnd);
VOID AddThreadToList(HANDLE hThread);
```

Файл resource.h (листинг 4.9), создаваемый автоматически, содержит определение констант для ресурсов приложения SEMMDI.

Листинг 4.9. Файл semmdi/resource.h

```
//{{NO_DEPENDENCIES}}
// Microsoft Developer Studio generated include file.
// Used by Mdiapp.rc
//
#define CM_HELPABOUT                100
#define CM_FILEEXIT                 101
#define CM_FILENEW                  102
#define CM_WINDOWTILE               103
#define CM_WINDOWCASCADE            104
#define CM_WINDOWICONS              105
#define CM_WINDOWCLOSEALL           106
#define ID_THREADCONTROL_SUSPEND    40001
#define ID_THREADCONTROL_RESUME     40002
#define ID_THREADCONTROL_PRIORITYLOWEST 40003
#define ID_THREADCONTROL_PRIORITYNORMAL 40004
#define ID_THREADCONTROL_PRIORITYHIGHEST 40005
#define ID_THREADCONTROL_GETPRIORITY    40006
#define ID_THREADCONTROL_KILLTHREAD     40007
#define ID_THREADCONTROL_CLOSEWINDOW    40008
#define ID_PRIORITYCLASS_REALTIME       40009
#define ID_PRIORITYCLASS_NORMAL         40010
#define ID_PRIORITYCLASS_HIGH           40011
#define ID_PRIORITYCLASS_IDLE           40012
#define ID_SEMAPHORE_INCREMENT          40013

// Next default values for new objects
//
#ifdef APSTUDIO_INVOKED
#ifndef APSTUDIO_READONLY_SYMBOLS
#define _APS_NO_MFC                 1
#define _APS_NEXT_RESOURCE_VALUE    102
#define _APS_NEXT_COMMAND_VALUE     40015
#define _APS_NEXT_CONTROL_VALUE     1000
#define _APS_NEXT_SYMED_VALUE       101
#endif
#endif
```

Ресурсы приложения SEMMDI определены в файле mdiapp.rc, который приведен в листинге 4.10.

Листинг 4.10. Файл semmdi/mdiapp.rc

```
//Microsoft Developer Studio generated resource script.
//
#include "resource.h"

#define APSTUDIO_READONLY_SYMBOLS
////////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 2 resource.
//
```

```
#include "afxres.h"

//////////////////////////////////////////////////////////////
#undef APSTUDIO_READONLY_SYMBOLS

//////////////////////////////////////////////////////////////
// English (U.S.) resources

#if !defined(AFX_RESOURCE_DLL) || defined(AFX_TARG_ENU)
#ifdef _WIN32
LANGUAGE LANG_ENGLISH, SUBLANG_ENGLISH_US
#pragma code_page(1252)
#endif //_WIN32

//////////////////////////////////////////////////////////////
//
// Menu
//

APP_MENU MENU DISCARDABLE
BEGIN
  POPUP "&File"
  BEGIN
    MENUITEM "&New",          CM_FILENEW
    MENUITEM SEPARATOR
    MENUITEM "E&xit",         CM_FILEEXIT
  END
  POPUP "&Semaphore"
  BEGIN
    MENUITEM "&Increment",    ID_SEMAPHORE_INCREMENT
  END
  POPUP "Priority class"
  BEGIN
    MENUITEM "&Realtime",     ID_PRIORITYCLASS_REALTIME
    MENUITEM "&High",         ID_PRIORITYCLASS_HIGH
    MENUITEM "&Normal",       ID_PRIORITYCLASS_NORMAL
    MENUITEM "&Idle",         ID_PRIORITYCLASS_IDLE
  END
  POPUP "&Window"
  BEGIN
    MENUITEM "&Tile",         CM_WINDOWTILE
    MENUITEM "&Cascade",      CM_WINDOWCASCADE
    MENUITEM "Arrange &Icons", CM_WINDOWICONS
    MENUITEM "Close &All",    CM_WINDOWCLOSEALL
  END
  POPUP "&Help"
  BEGIN
    MENUITEM "&About...",     CM_HELPABOUT
  END
END

IDR_POPUPMENU MENU DISCARDABLE
BEGIN
  POPUP "&Thread Control"
  BEGIN
    MENUITEM "&Suspend",      ID_THREADCONTROL_SUSPEND
    MENUITEM "&Resume",       ID_THREADCONTROL_RESUME
    MENUITEM SEPARATOR
    MENUITEM "Priority &Lowest", ID_THREADCONTROL_PRIORITYLOWEST

    MENUITEM "Priority &Normal", ID_THREADCONTROL_PRIORITYNORMAL

    MENUITEM "Priority &Highest", ID_THREADCONTROL_PRIORITYHIGHEST

    MENUITEM SEPARATOR
    MENUITEM "&Get Priority", ID_THREADCONTROL_GETPRIORITY
    MENUITEM SEPARATOR
    MENUITEM "&Kill Thread", ID_THREADCONTROL_KILLTHREAD
    MENUITEM "&Close Window", ID_THREADCONTROL_CLOSEWINDOW
  END
END
```

```
//////////////////////////////////////////////////////////////
//
// Icon
//

// Icon with lowest ID value placed first to ensure
// application icon
// remains consistent on all systems.
APP_ICON            ICON    DISCARDABLE     "multimdi.ico"
APPCLIENT_ICON      ICON    DISCARDABLE     "mdicl.ico"

#ifdef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
//
// TEXTINCLUDE
//

1 TEXTINCLUDE DISCARDABLE
BEGIN
    "resource.h\0"
END

2 TEXTINCLUDE DISCARDABLE
BEGIN
    "#include ""afxres.h""\r\n"
    "\0"
END

3 TEXTINCLUDE DISCARDABLE
BEGIN
    "\r\n"
    "\0"
END

#endif    // APSTUDIO_INVOKED
#endif    // English (U.S.) resources
//////////////////////////////////////////////////////////////
#ifndef APSTUDIO_INVOKED
//////////////////////////////////////////////////////////////
//
// Generated from the TEXTINCLUDE 3 resource.
//
//////////////////////////////////////////////////////////////
#endif    // not APSTUDIO_INVOKED
```

Определения и глобальные переменные

Приложение SEMMDI сделано на базе описанного ранее приложения MultiMDI, поэтому здесь мы приведем сокращенное описание определений, глобальных переменных и функций.

Для каждого дочернего MDI-окна в приложении MultiMDI мы создавали структуру типа CHILD_WINDOW_TAG, в которой хранилась такая информация, как признак активности задачи, запущенной для этого окна, критическая секция для рисования в окне, а также идентификатор задачи, запущенной для окна. Создавая приложение SEMMDI, мы добавили в эту структуру поле fWaiting:

```
typedef struct _CHILD_WINDOW_TAG
{
  BOOL fActive;
  BOOL fWaiting;
  CRITICAL_SECTION csChildWindowPaint;
  HANDLE hThread;
} CHILD_WINDOW_TAG;
typedef CHILD_WINDOW_TAG *LPCHILD_WINDOW_TAG;
```

В поле fWaiting записывается значение TRUE, когда соответствующая задача находится в состоянии ожидания семафора. Проверяя содержимое этого поля, мы запрещаем пользователю удалять окна, если соответствующие им задачи находятся в состоянии ожидания семафора.

Описание функций приложения

Приведем описание функций, которые изменились по сравнению с приложением MultiMDI.

Функция WinMain

Дополнительно к действиям, выполняемым этой функцией в приложении MultiMDI, новый вариант функции WinMain создает семафор с именем lpSemaphoreName:

```
hSemaphore = CreateSemaphore(NULL, 2, 2, lpSemaphoreName);
if(hSemaphore == NULL)
{
 MessageBox(NULL, "Ошибка при создании семафора",
   szWindowTitle, MB_OK | MB_ICONEXCLAMATION);
 return FALSE;
}
```

Начальное и максимальное значение счетчика семафора равно двум, поэтому одновременно будут работать не более двух задач, выполняющих рисование эллипсов в MDI-окнах, созданных пользователем. Если при создании семафора произошла ошибка, на экране появляется сообщение, после чего работа приложения завершается.

После завершения цикла обработки сообщений наше приложение освобождает идентификатор созданного ранее семафора, вызывая для этого функцию CloseHandle:

```
CloseHandle(hSemaphore);
```

Функция FrameWndProc

Эта функция обрабатывает сообщения для главного окна приложения Frame Window, в частности, сообщения, которые поступают от главного меню приложения.

Обработчик сообщения WM_COMMAND получает управление, когда пользователь выбирает строки главного меню приложения.

Обработка сообщений от меню Semaphore

Если из меню Semaphore выбрать строку Increment, содержимое счетчика семафора будет увеличено на единицу:

```
case ID_SEMAPHORE_INCREMENT:
{
 ReleaseSemaphore(hSemaphore, 1, NULL);
 break;
}
```

Для этого используется функция ReleaseSemaphore.

Обработка сообщений от меню Window

Меню Window в приложении SEMMDI, как и в приложении MultiMDI, предназначено для управления дочерними MDI-окнами. Выбирая строки этого меню, пользователь может упорядочить расположение окон одним из двух способов, упорядочить расположение пиктограмм минимизированных окон, а также закрыть все дочерние MDI-окна.

Функция ChildWndProc

Функция ChildWndProc обрабатывает сообщения, поступающие в дочерние MDI-окна.

Обработка сообщения WM_PAINT

Обработчик сообщения WM_PAINT получает управление, когда возникает необходимость перерисовать внутреннюю область дочернего MDI-окна. При перерисовке с помощью критической секции выполняется синхронизация главной задачи приложения и задачи дочернего окна.

Если задача, запущенная для дочернего MDI-окна, находится в состоянии ожидания, в окне отображается строка Waiting, а если эта задача работает - строка Child Window. Running. Для выбора нужной строки обработчик сообщения WM_PAINT проверяет содержимое поля fWaiting структуры параметров окна типа CHILD_WINDOW_TAG:

```
if(lpMyWndTag->fWaiting)
 DrawText(hdc, "Waiting...", -1, &rc,
   DT_SINGLELINE | DT_CENTER | DT_VCENTER);
else
 DrawText(hdc, "Child Window. Running", -1, &rc,
   DT_SINGLELINE | DT_CENTER | DT_VCENTER);
```

Обработка сообщения WM_CLOSE

Это сообщение поступает в функцию дочернего окна при уничтожении последнего.

Обработчик сообщения WM_CLOSE проверяет флаг ожидания в поле fWaiting структуры параметров окна типа CHILD_WINDOW_TAG. Если задача не находится в состоянии ожидания, обработчик сбрасывает признак активности задачи, в результате чего задача завершает свою работу:

```
if(lpMyWndTag->fWaiting)
 return 0;
else
 lpMyWndTag->fActive = 0;
```

В том случае, когда задача находится в состоянии ожидания, обработчик сообщения WM_CLOSE возвращает нулевое значение, запрещая удаление окна.

Обработка сообщения WM_COMMAND

Сообщение WM_COMMAND поступает в функцию дочернего MDI-окна, когда пользователь выбирает строки плавающего меню.

С помощью плавающего меню вы можете удалить дочернее MDI-окно, завершив работу соответствующей задачи. При этом задача не должна находится в состоянии ожидания семафора:

```
case ID_THREADCONTROL_CLOSEWINDOW:
{
 lpMyWndTag = (LPCHILD_WINDOW_TAG)GetWindowLong(
        hwnd, GWL_USERDATA);
 if(!lpMyWndTag->fWaiting)
 {
  SendMessage(hwndClient, WM_MDIDESTROY, (WPARAM)hwnd, 0);
  lpMyWndTag->fActive = 0;
 }
 break;
}
```

Как видно из приведенного выше фрагмента исходного текста, для определения возможности удаления окна обработчик сообщения WM_COMMAND проверяет флаг ожидания в поле fWaiting структуры параметров окна типа CHILD_WINDOW_TAG.

Функция задачи ThreadRoutine

Задача ThreadRoutine запускается для каждого вновь создаваемого дочернего MDI-окна. Ее функция получает параметр - идентификатор этого дочернего окна, который используется ей для выполнения рисования. Другие параметры, нужные для работы функции задачи ThreadRoutine, извлекаются из структуры типа CHILD_WINDOW_TAG.

Перед началом своей работы функция задачи ThreadRoutine устанавливает флаг ожидания в поле fWaiting структуры типа CHILD_WINDOW_TAG:

```
lpMyWndTag->fWaiting = TRUE;
```

Затем она выполняет ожидание семафора, вызывая для этого функцию WaitForSingleObject:

```
if(WAIT_FAILED == WaitForSingleObject(hSemaphore, INFINITE))
{
 return 0;
}
```

Если пользователь создал не более двух окон, вызов функции WaitForSingleObject не приведет к задержке в выполнении задачи. Если же окон создано больше, задача перейдет в состояние ожидания.

Когда задача вновь возобновит свою работу (в результате того, что пользователь закроет одно из активных окон), флаг ожидания сбрасывается:

```
lpMyWndTag->fWaiting = FALSE;
```

5 РАБОТА С ФАЙЛАМИ

Операционная система Microsoft Windows NT имеет очень развитые средства работы с файлами. Она способна выполнять операции над файлами, расположенными в нескольких файловых системах, таких как FAT (старая добрая файловая система MS-DOS), HPFS (файловая система, созданная операционной системой IBM OS/2), NTFS (файловая система Microsoft Windows NT) и CDFS (еще одна файловая система Microsoft Windows NT, предназначенная для доступа к накопителю CD-ROM). При необходимости можно разработать собственную файловую систему и подключить ее к Microsoft Windows NT.

Файловая система FAT используется операционной системой Microsoft Windows NT для записи файлов на дискеты или на жесткий диск. Заметим, что в среде этой операционной системы вы можете отфоматировать дискеты только в формате FAT.

С помощью команды FORMAT или приложения Disk Administrator (пиктограмма которого находится в группе Administrative Tools) вы сможете отформатировать разделы жесткого диска для работы с файловыми системами FAT или NTFS.

Что же касается файловой системы HPFS, то возможность работы с ней в среде Microsoft Windows NT оставлена исключительно для совместимости с операционной системой IBM OS/2. В среде Microsoft Windows NT вы не сможете создать новый раздел HPFS. Однако если по каким-либо причинам нежелательно удалять существующий раздел в этом формате (например, вы работаете попеременно с операционными системами Microsoft Windows NT и IBM OS/2), то в среде Microsoft Windows NT вы будете иметь доступ к файлам, расположенным в разделе HPFS. Кстати, операционная система IBM OS/2 Warp версии 3.0 не предоставляет доступ к разделам NTFS.

В этой главе мы расскажем вам о преимуществах файловой системы NTFS над другими перечисленными выше файловыми системами и кратко опишем наиболее важные функции программного интерфейса операционной системы Microsoft Windows NT, предназначенные для работы с файлами. Вопросы использования файлов, отображаемых в виртуальную память, мы отложим до следующего тома “Библиотеки системного программиста”, посвященного Microsoft Windows NT. Там же вы найдете исходные тексты приложений, работающих с файлами.

Преимущества файловой системы NTFS

Прежде чем мы перечислим достоинства файловой системы NTFS, сделаем краткий обзор недостатков других файловых систем, созданных для различных операционныъх систем персональных компьютеров.

Операционная система MS-DOS и файловая система FAT

Все вы хорошо знаете недостатки файловой системы FAT, разработанной на заре развития операционной системы MS-DOS. Однако в те времена жесткие диски персональных компьютеров имели объем 10 - 40 Мбайт, и файловая система FAT в целом неплохо подходила для работы с такими дисками и дискетами.

Один из недостатков файловой системы FAT, наиболее очевидный для пользователей, заключается в жестких ограничениях на имена файлов и каталогов. Имя должно состоять не более чем из 8 символов, плюс еще три символа расширения имени. Так как расширение имени всегда используется для обозначения типа документа (например, txt - текстовый файл, doc - файл документа Microsoft Word), пользователь был вынужден изобретать восьмисимвольные имена, отражающие содержимое файла. Если пользователь работает с десятками или сотнями документов (что совсем не редкость), ему нужно иметь незаурядную фантазию, чтобы суметь придумать для всех документов осмысленные имена.

Другой, менее очевидный недостаток, заключается в том что информация о каждом файле хранится в нескольких удаленных друг от друга местах диска. Для того чтобы прочесть файл, операционная система должна сначала найти его имя и номер первого распределенного файлу кластера в каталоге, затем ей следует обратиться к таблице размещения файлов FAT, чтобы получить список кластеров, распределенных файлу, и, наконец, прочитать кластеры, которые находятся совсем в другом месте диска.

Если с диском работает только одна программа, накладные расходы на перемещения блока магнитных головок между этими областями диска, возможно, не приведут к заметному снижению производительности (особенно при использования кэширования диска). Однако в мультизадачной среде, когда несколько программ пытаются получить доступ одновременно к нескольким разным файлам, указанные накладные расходы будут заметнее.

У файловой системы FAT есть еще одна большая проблема - увеличение фрагментации диска и файлов при интенсивной работе с файлами. Фрагментация приводит к тому, что файл как бы размазывается по диску. Для чтения такого файла нужно много времени, так как перемещение головок выполняется относительно медленно. Поэтому пользователям операционной системы очень хорошо знакомы утилиты дефрагментации, такие как Microsoft Defrag и Norton Speedisk.

Файловой системе FAT и средствам работы с диском на разных уровнях в среде операционной системы MS-DOS мы посвятили 19 том “Библиотеки системного программиста”, который называется “MS-DOS для программиста. Часть вторая”. Вы найдете в этой книге подробное изложение принципов построения файловой системы FAT.

Операционная система Microsoft Windows версии 3.1

Операционная система Microsoft Windows версии 3.1 и более ранних версий не внесла ничего нового в теорию и практику файловых систем, ограничившись использованием все той же системы FAT. Более того, для доступа к файлам процессор переключался из защищенного в реальный режим или в режим виртуального процессора 8086, а затем вызывалось прерывание MS-DOS с номером 21h.

Постоянные переключения режимов работы процессора приводили к снижению производительности. И хотя для подавляющего большинства дисковых контроллеров IDE можно было включить режим 32-разрядного доступа к диску, исключающий вызов модулей BIOS (работающих только в реальном режиме), заметного влияния на быстродействие системы этот режим не оказывал. К тому же, для большинства контроллеров SCSI 32-разрядный доступ вообще нельзя было использовать.

В программном интерфейсе операционной системы Microsoft Windows версии 3.1 были предусмотрены несколько функций для работы с файлами. Мы описали их в 113 томе “Библиотеки системного программиста”, который называется “Операционная система Microsoft Windows 3.1 для программиста. Часть третья”. Это такие функции, как OpenFile, _lopen, _lclose, _lcreat, _lread, _lwrite, _hread, _hwrite, _llseek.

Все перечисленные выше функции, а также функции стандартной библиотеки времени выполнения тансляторa Microsoft Visual C++ можно использовать в приложениях Microsoft Windows NT. Это сделано для обеспечения совместимости на уровне исходных текстов. Однако лучше работать с новыми функциями программного интерфейса Microsoft Windows NT, обладающими намного более широкими возможностями.

Операционная система Microsoft Windows for Workgroups

Известная своими средствами работы в сети операционная система Microsoft Windows for Workgroups версии 3.11 имела одно усовершенствование, заметно увеличивающее скорость работы с файлами. Эта операционная система позволяла устанавливать режим 32-разрядного доступа не только к диску, но и к файлам, полностью исключая необходимость переключения процессора из защищенного режима работы в режим виртуального процессора 8086. И хотя по-прежнему 32-разрядный драйвер диска, поставляющийся в составе Microsoft Windows for Workgroups версии 3.11, мог работать далеко не со всеми дисковыми контроллерами, многие фирмы, изготавляющие такие контроллеры, продавали 32-разрядные драйверы, позволяющие воспользоваться преимуществами доступа к диску и файлам в 32-разрядном режиме.

Однако доступ к дискам и файлам в защищенном режиме работы процессора - это все, что изменилось в файловой системе. Операционная система Microsoft Windows for Workgroups, так же как и предыдущие версии Microsoft Windows, работает с файловой системой FAT.

Операционная система Microsoft Windows 95

При разработке операционной системы Microsoft Windows 95 была создана новая модификация файловой системы, которая получила название VFAT. Ее также называют файловой системой с таблицей размещения файлов защищенного режима - Protected mode FAT.

Несмотря на свое название, в файловой системе VFAT используется самый обычный формат таблицы размещения файлов. Этого, однако, не скажешь о формате дескрипторов файлов, расположенных в каталогах. В этот формат были внесены добавления, позволяющие пользователям указывать имена файлов и каталогов размером до 260 символов.

Каким образом это было достигнуто?

Для каждого файла или каталога, имеющего длинное имя, было задействовано несколько дополнительных дескрипторов, расположенных рядом. При этом имя (наряду с другими дополнительными атрибутами) было разбито на несколько частей и хранилось в нескольких дополнительных дескрипторах.

Однако самое интересное, что при создании файловой системы VFAT была обеспечена совместимость со старыми программами MS-DOS, которые могли работать только с короткими именами “в стандарте 8.3”. Для этого во всех дополнительных дескрипторах были установлены атрибуты “скрытый”, “системный”, “метка тома”, благодаря чему программы MS-DOS пропускали такие дескрипторы, не обращая на них внимания. В то же время для каждого файла или каталога с длинным именем файловая система VFAT создавала дескриптор специального вида, содержащий альтернативное (алиасное) имя. Альтернативное имя состоит из первых шести символов длинного имени, из которого убраны пробелы, символа “тильда” (~) и числа. Например, для имени The Mircosoft Network создается альтернативное имя THEMIC~1.

В результате пользователи могли работать с длинными именами как в приложениях Microsoft Windows 95, так и в программах MS-DOS (хотя, конечно, альтернативное имя в ряде случаев может мало напоминать исходное длинное имя).

Разработчики операционной системы Microsoft Windows 95 добивались максимальной совместимости с приложениями Microsoft Windows 3.1 и программами MS-DOS. Поэтому для работы с дисковыми устройствами (особенно экзотическими) в ряде случаев можно использовать драйверы

реального режима, загружаемые с помощью файла config.sys. Это очень удобно, так как вы можете пользоваться устройством, для которого в составе Microsoft Windows 95 пока нет специального драйвера. Однако при использовании драйверов реального режима в процессе обращения к устройству происходит переключение процессора из защищенного режима в режим виртуального процессора 8086, что снижает производительность системы.

Файловая система HPFS

Высокопроизводительная файловая система HPFS (High Performance File System), использованная в операционной системе IBM OS/2, лишена большинства недостатков файловой системы FAT.

Эта файловая система оптимизирована для мультизадачной среды и ускоряет одновременную работу программ с файлами, расположенными на дисках большого объема.

Специальный алгоритм размещения файлов значительно уменьшает вредное влияние фрагментации файлов, уменьшающей общую производительность системы. При размещении файла для него подбирается подходящий непрерывный свободный участок диска и оставляется некоторый запас свободного пространства. Когда файл расширяется, для него выделяются в первую очередь секторы, относящиеся к зарезервированному для этого файла участку диска. Если же размер файла увеличился значительно, для него может быть выделен еще один или несколько свободных участков.

При использовании HPFS пользователь может указывать имена файлов размером до 254 символов, причем имя может состоять из заглавных и прописных букв, а также пробелов и некоторых других символов, например, символов “.” (в произвольном количестве).

В дополнение к таким атрибутам файлов, как “только читаемый”, “скрытый”, “системный” и “архивированный”, IBM OS/2 хранит для каждого файла набор расширенных атрибутов. Это тип файла, комментарий и ключевые слова для поиска, пиктограмма для визуального представления файла и т. д.

В распоряжении программиста имеются многочисленные функции программного интерфейса IBM OS/2, с помощью которых можно выполнять операции с файлами, в том числе и многозадачные. Например, можно запустить операцию чтения или записи фрагмента файла как отдельную задачу, которая будет выполняться автономно от запустившей ее задачи. Есть средства и для работы с расширенными атрибутами файлов, для создания, удаления и переименования файлов и каталогов, а также другие необходимые функции.

Заметим, что файловая система HPFS не содержит никаких средств разграничения доступа. Однако если она используется совместно с файл-сервером IBM Lan Server, в операционную систему IBM OS/2 добавляется специальный драйвер, обеспечивающий такое разграничение.

Другое ограничение файловой системы HPFS заключается в том, что виртуальным машинам DOS, работающим под управлением IBM OS/2, недоступны каталоги и файлы с длинными именами. Причина заключается в том, что в этой операционной системе не предусмотрено никакого механизма, обеспечивающего генерацию коротких альтернативных имен, как это сделано в операционных системах Microsoft Windows 95 и Microsoft Windows NT.

Подробнее об операционной системе IBM OS/2 вы можете прочитать в 20 томе “Библиотеки системного программиста”, который называется “Операционная система IBM OS/2 Warp” и в 25 томе этой же серии с названием “Программирование для операционной системы IBM OS/2 Warp”.

Основные характеристики файловой системы NTFS

Теперь, после такого краткого обзора наиболее популярных файловых систем, мы может сказать, что файловая система NTFS вобрала в себя самое лучшее из всего, что было создано к настоящему моменту в этой области. Обладая практически такой же высокой производительностью, как файловая система FAT, файловая система NTFS допускает использование длинных имен, имеет намного более высокую надежность и средства разграничения доступа, мощные средства поиска файлов в каталогах, основанные на использовании B-деревьев, и не требует выполнения периодической дефрагментации диска.

Что касается имен файлов, то в файловой системе NTFS допускается указывать имена размером до 255 символов в кодировке UNICODE, когда каждый символ представляется двумя байтами. Кодировку UNICODE мы рассмотрим подробнее в одном из следующих томов “Библиотеки системного программиста”, посвященного операционной системе Microsoft Windows NT. Сейчас мы только скажем, что она позволяет сохранить имена файлов при их копировании в системы, использующие другие национальные языки.

В именах файлов и каталогов можно использовать строчные либо прописные буквы, при этом файловая система не делает между ними различий. В итоге имена MyFile, MYFILE и myfile означают одно и то же. В режиме совместимости со стандартом POSIX, однако, операционная система Microsoft Windows NT будет считать все перечисленные выше имена разными. Рассмотрение стандарта POSIX, предназначенного для совместимости с UNIX-программами (точнее говоря, приложения UNIX, отвечающие стандарту POSIX, будет легче переносить на платформу Microsoft Windows NT), выходит за рамки нашей книги.

Использование B-деревьев при поиске имен в каталогах вместо обычного последовательного пребора значительно сокращает время открывания файлов, особенно если каталог содержит очень много файлов (последнее не редкость, так как современные приложения состоят из десятков, если не сотен файлов).

Стоит специально отметить, что в отличие от HPFS, файловая система NTFS содержит специальные средства, предназначенные для повышения устойчивости к различным аварийным ситуациям, таким, например, как внезапное отключение электропитания или аварийный останов операционной системы. Эти средства используют логику обработки транзакций, в результате чего операции, которые в результате аварии не успели завершиться, будут отменены с восстановлением файловой системы в исходное состояние.

Файловая система NTFS практически не имеет ограничений на макисмальный размер файла. Так, если файловые системы FAT и HPFS позовляет создавать файлы размером не более 2^{32} байта, файловая система NTFS может работать с файлами размером до 2^{64} байта. Максимальный размер пути, который в FAT составлял 64 байта, в NTFS не ограничен.

Для совместимости с программами MS-DOS, запущенными под управлением операционной системы Microsoft Windows NT, файловая система NTFS для всех файлов и каталогов создает короткие альтернативные имена, аналогично тому, как это делает операционная система Microsoft Windows 95. Если операционная система Microsoft Windows NT Server используется в качестве файл-сервера сети, то альтернативные имена позволяют получить доступ ко всем файлам и каталогам сервера из рабочих станций MS-DOS, неспособных работать с длинными именами напрямую.

Ко всему прочему добавим, что файловая система NTFS в операционной системе Microsoft Windows NT версии 3.51 позволяет выполнять динамическую компрессию файлов, аналогично тому как это делает драйвер DriveSpace или Stacker в операционной системе MS-DOS.

Функции для работы с файлами

В этом разделе мы рассмотрим основные функции программного интерфейса операционной системы Microsoft Windows NT, предназначенные для работы с файлами и каталогами.

Универсальная функция CreateFile

Можно было бы подумать, что функция CreateFile предназначена только для создания файлов, аналогично функции _lcreat из программного интерфейса Microsoft Windows версии 3.1, однако это не так. Во-первых, с помощью функции CreateFile можно выполнять не только создание нового файла, но и открывание существующего файла или каталога, а также изменение длины существующего файла. Во-вторых, эта функция может выполнять операции не только над файлами, но и над каналами передачи данных, трубами (pipe), дисковыми устройствами и консолями. В этом томе мы, однако, ограничимся лишь файлами.

Прототип функции CreateFile мы привели ниже:

```
HANDLE CreateFile(
  LPCTSTR lpFileName,     // адрес строки имени файла
  DWORD  dwDesiredAccess, // режим доступа
  DWORD  dwShareMode, // режим совместного использования файла
  LPSECURITY_ATTRIBUTES lpSecurityAttributes, // дескриптор
                                  // защиты
  DWORD  dwCreationDistribution, // параметры создания
  DWORD  dwFlagsAndAttributes,   // атрибуты файла
  HANDLE hTemplateFile);  // идентификатор файла с атрибутами
```

Через параметр lpFileName вы должны передать этой функции адрес строки, содержащей имя файла, который вы собираетесь создать или открыть. Строка должна быть закрыта двоичным нулем.

Параметр dwDesiredAccess определяет тип доступа, который должен быть предоставлен к открываемому файлу. Здесь вы можете использовать логическую комбинацию следующих констант:

Константа	*Описание*
0	Доступ запрещен, однако приложение может определять атрибуты файла или устройства, открываемого при помощи функции CreateFile
GENERIC_READ	Разрешен доступ на чтение
GENERIC_WRITE	Разрешен доступ на запись

С помощью параметра dwShareMode задаются режимы совместного использования открываемого или создаваемого файла. Для этого параметра вы можете указать комбинацию следующих констант:

Константа	*Описание*
0	Совместное использование файла запрещено
FILE_SHARE_READ	Другие приложения могут открывать файл с помощью функции CreateFile для чтения
FILE_SHARE_WRITE	Аналогично предыдущему, но на запись

Через параметр lpSecurityAttributes необходимо передать указатель на дескриптор защиты или значение NULL, если этот дескриптор не используется. В наших приложениях мы не работаем с дескриптором защиты.

Параметр dwCreationDistribution определяет действия, выполняемые функцией CreateFile, если приложение пытается создать файл, который уже существует. Для этого параметра вы можете указать одну из следующих констант:

Константа	*Описание*
CREATE_NEW	Если создаваемый файл уже существует, функция CreateFile возвращает код ошибки
CREATE_ALWAYS	Существующий файл перезаписывается, при этом содержимое старого файла теряется
OPEN_EXISTING	Открывается существующий файл. Если файл с указанным именем не существует, функция CreateFile возвращает код ошибки
OPEN_ALWAYS	Если указанный файл существует, он открывается. Если файл не существует, он будет создан
TRUNCATE_EXISTING	Если файл существует, он открывается, после чего длина файла устанавливается равной нулю. Содержимое старого файла теряется. Если же файл не существует, функция CreateFile возвращает код ошибки

Параметр dwFlagsAndAttributes задает атрибуты и флаги для файла.

При этом можно использовать любые логические комбинации следующих атрибутов (кроме атрибута FILE_ATTRIBUTE_NORMAL, который можно использовать только отдельно):

Атрибут	*Описание*
FILE_ATTRIBUTE_ARCHIVE	Файл был архивирован (выгружен)
FILE_ATTRIBUTE_COMPRESSED	Файл, имеющий этот атрибут, динамически сжимается при записи и восстанавливается при чтении. Если этот атрибут имеет каталог, то для всех расположенных в нем файлов и каталогов также выполняется динамическое сжатие данных
FILE_ATTRIBUTE_NORMAL	Остальные перечисленные в этом списка атрибуты не установлены
FILE_ATTRIBUTE_HIDDEN	Скрытый файл
FILE_ATTRIBUTE_READONLY	Файл можно только читать
FILE_ATTRIBUTE_SYSTEM	Файл является частью операционной системы

В дополнение к перечисленным выше атрибутам, через параметр dwFlagsAndAttributes вы можете передать любую логическую комбинацию флагов, перечисленных ниже:

Флаг	*Описание*
FILE_FLAG_WRITE_THROUGH	Отмена промежуточного кэширования данных для уменьшения вероятности потери данных при аварии
FILE_FLAG_NO_BUFFERING	Отмена промежуточной буферизации или кэширования. При использовании этого флага необходимо выполнять чтение и запись порциями, кратными размеру сектора (обычно 512 байт)
FILE_FLAG_OVERLAPPED	Выполнение чтения и записи асинхронно. Во время асинхронного чтения или записи приложение может продолжать обработку данных
FILE_FLAG_RANDOM_ACCESS	Указывает, что к файлу будет выполняться произвольный доступ. Флаг предназначен для оптимизации кэширования
FILE_FLAG_SEQUENTIAL_SCAN	Указывает, что к файлу будет выполняться последовательный доступ от начала файла к его концу. Флаг предназначен для оптимизации кэширования
FILE_FLAG_DELETE_ON_CLOSE	Файл будет удален сразу после того как приложение закроет его идентификтор. Этот флаг удобно использовать для временных файлов
FILE_FLAG_BACKUP_SEMANTICS	Файл будет использован для выполнения операции выгрузки или восстановления. При этом выполняется проверка прав доступа
FILE_FLAG_POSIX_SEMANTICS	Доступ к файлу будет выполняться в соответствии со спецификацией POSIX

И, наконец, последний параметр hTemplateFile предназначен для доступа к файлу шаблона с расширенными атрибутами для создаваемого файла. Этот параметр мы рассматривать не будем для экономии места. При необходимости вы найдете всю информацию по этому вопросу в документации, поставляемой вместе с SDK.

В случае успешного завершения функция CreateFile возвращает идентификатор созданного или открытого файла (или каталога). При ошибке возвращается значение INVALID_HANDLE_VALUE (а не NULL, как можно было бы предположить). Код ошибки можно определить при помощи функции GetLastError.

В том случае, если файл уже существует и были указаны константы CREATE_ALWAYS или OPEN_ALWAYS, функция CreateFile не возвращает код ошибки. В то же время в этой ситуации функция GetLastError возвращает значение ERROR_ALREADY_EXISTS.

Функция CloseHandle

Функция CloseHandle позволяет закрыть файл. Она имеет единственный параметр - идентификатор закрываемого файла. Заметим, что если мы указали функции CreateFile флаг FILE_FLAG_DELETE_ON_CLOSE, сразу после закрывания файл будет удален. Как мы уже говорили, такая методика очень удобна при работе со временными файлами.

Функции ReadFile и WriteFile

С помощью функций ReadFile и WriteFile приложение может выполнять, соответственно, чтение из файла и запись в файл. По своему назначению эти функции аналогичны функциям _lread, _lwrite, _hread и _hwrite из программного интерфейса Microsoft Windows версии 3.1.

Приведем прототипы функций ReadFile и WriteFile:

```
BOOL ReadFile(
  HANDLE  hFile,              // идентификатор файла
  LPVOID  lpBuffer,           // адрес буфера для данных
  DWORD   nNumberOfBytesToRead, // количество байт, которые
                    // необходимо прочесть в буфер
  LPDWORD lpNumberOfBytesRead,  // адрес слова, в которое
          // будет записано количество прочитанных байт
  LPOVERLAPPED lpOverlapped); // адрес структуры типа
                    // OVERLAPPED

BOOL WriteFile(
  HANDLE  hFile,              // идентификатор файла
  LPVOID  lpBuffer,       // адрес записываемого блока данных
  DWORD   nNumberOfBytesToWrite, // количество байт, которые
                    // необходимо записать
  LPDWORD lpNumberOfBytesWrite,  // адрес слова, в котором
          // будет сохранено количество записанных байт
  LPOVERLAPPED lpOverlapped); // адрес структуры типа
                    // OVERLAPPED
```

Через параметр hFile этим функциям необходимо передать идентификатор файла, полученный от функции CreateFile.

Параметр lpBuffer должен содержать адрес буфера, в котором будут сохранены прочитанные данные (для функции ReadFile), или из которого будет выполняться запись данных (для функции WriteFile).

Параметр nNumberOfBytesToRead используется для функции ReadFile и задает количество байт данных, которые должны быть прочитаны в буфер lpBuffer. Аналогично, параметр nNumberOfBytesToWrite задает функции WriteFile размер блока данных, имеющего адрес lpBuffer, который должен быть записан в файл.

Так как в процессе чтения возможно возникновение ошибки или достижение конца файла, количество прочитанных или записанный байт может отличаться от значений, заданных, соответственно, параметрами nNumberOfBytesToRead и nNumberOfBytesToWrite. Функции ReadFile и WriteFile записывают количество действительно прочитанных или записанных байт в двойное слово с адресом, соответственно, lpNumberOfBytesRead и lpNumberOfBytesWrite.

Параметр lpOverlapped используется в функциях ReadFile и WriteFile для организации аснхронного режима чтения и записи. Если запись выполняется синхронно, в качестве этого параметра следует указать значение NULL. Способы выполнения асинхронного чтения и записи мы рассмотрим позже. Заметим только, что для использования асинхронного режима файл должен быть открыт функцией CreateFile с использованием флага FILE_FLAG_OVERLAPPED. Если указан этот флаг, параметр lpOverlapped не может иметь значение NULL. Он обязательно должен содержать адрес подготовленной структуры типа OVERLAPPED.

Если функции ReadFile и WriteFile были выполнены успешно, они возвращают значение TRUE. При возникновении ошибки возвращается значение FALSE. В последнем случае вы можете получить код ошибки, вызвав функцию GetLastError.

В процессе чтения может быть достигнут конец файла. При этом количество действительно прочитанных байт (записывается по адресу lpNumberOfBytesRead) будет равно нулю. В случае достижения конца файла

при чтении ошибка не возникает, поэтому функция ReadFile вернет значение TRUE.

Функция FlushFileBuffers

Так как ввод и вывод данных на диск в операционной системе Microsoft Windows NT буферизуется, запись данных на диск может быть отложена до тех пор, пока система не освободится от выполнения текущей работы. С помощью функции FlushFileBuffers вы можете принудительно заставить операционную систему записать на диск все изменения для файла, идентификатор которого передается этой функции через единственный параметр:

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

В случае успешного завершения функция возвращает значение TRUE, при ошибке - FALSE. Код ошибки вы можете получить при помощи функции GetLastError.

Заметим, что при закрывании файла функцией CloseHandle содержимое всех буферов, связанных с этим файлом, записывается на диск автоматически. Поэтому вы должны использовать функцию FlushFileBuffers только в том случае, если запись содержимого буферов нужно выполнить до закрывания файла.

Функция SetFilePointer

С помощью функции SetFilePointer приложение может выполнять прямой доступ к файлу, перемещая указатель текущей позиции, связанный с файлом. Сразу после открывания файла этот указатель устанавливается в начало файла. Затем он передвигается функциями ReadFile и WriteFile на количество прочитанных или записанных байт, соответственно.

Функция SetFilePointer позволяет выполнить установку текущей позиции:

```
DWORD SetFilePointer(
  HANDLE hFile,              // идентификатор файла
  LONG   lDistanceToMove, // количество байт, на которое будет
                    // передвинута текущая позиция
  PLONG  lpDistanceToMoveHigh, // адрес старшего слова,
         // содержащего расстояние для перемещения позиции
  DWORD  dwMoveMethod);      // способ перемещения позиции
```

Через параметр hFile вы должны передать этой функции идентификатор файла, для которого выполняется изменение текущей позиции.

Параметр lDistanceToMove, определяющий дистанцию, на которую будет передвинута текущая позиция, может принимать как положительные, так и отрицательные значения. В первом случае текущая позиция переместится по направлению к концу файла, во втором - к началу файла.

Если вы работаете с файлами, размер которых не превышает 2^{32} - 2 байта, для параметра lpDistanceToMoveHigh можно указать значение NULL. В том случае, когда ваш файл очень большой, для указания смещения может потребоваться 64-разрядное значение. Для того чтобы указать очень большое смещение, вы должны записать старшее 32-разрядное слово этого 64-разрядного значения в переменную, и передать функции SetFilePointer адрес этой переменной через параметр lpDistanceToMoveHigh. Младшее слово смещения следует передавать как и раньше, через параметр lDistanceToMove.

Параметр dwMoveMethod определяет способ изменения текущей позиции и может принимать одно из перечисленных ниже значений:

Значение	*Описание*
FILE_BEGIN	Смещение отсчитывается от начала файла, при этом значение смещения трактуется как беззнаковая величина
FILE_CURRENT	Смещение отсчитывается от текущей позиции в файле и может принимать как положительные, так и отрицательные значения
FILE_END	Смещение отсчитывается от конца файла и трактуется как отрицательная величина

В случае успешного завершения функция SetFilePointer возвращает младшее слово новой 64-разрядной позиции в файле. Старшее слово при этом записывается по адресу, заданному параметром lpDistanceToMoveHigh.

При ошибке функция возвращает значение 0xFFFFFFFF. При этом в слово по адресу lpDistanceToMoveHigh записывается значение NULL. Код ошибки вы можете получить при помощи функции GetLastError.

Функция SetEndOfFile

При необходимости изменить длину файла (уменьшить или увеличить), вы можете воспользоваться функцией SetEndOfFile. Эта функция устанавливает новую длину файла в соответствии с текущей позицией:

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

Для изменения длины файла вам достаточно установить текущую позицию в нужное место с помощью функции SetFilePointer, а затем вызвать функцию SetEndOfFile.

Функции LockFile и UnlockFile

Так как операционная система Microsoft Windows NT является мультизадачной и допускает одновременную работу многих процессов, возможно возникновение ситуаций, в которых несколько задач попытаются выполнять запись или чтение для одних и тех же файлов. Например, два процесса могут попытаться изменить одни и те же записи файла базы данных, при этом третий процесс будет в то же самое время выполнять выборку этой записи.

Напомним, что если функции CreateFile указать режимы совместного использования файла FILE_SHARE_READ или FILE_SHARE_WRITE, несколько процессов смогут одновременно открыть файлы и выполнять операции чтения и записи, соответственно. Если же эти режимы не указаны, совместное использование файлов будет невозможно. Первый же процесс, открывший файл, заблокирует возможность работы с этим файлом для других процессов.

Очевидно, что в ряде случаев вам все же необходимо обеспечить возможность одновременной работы нескольких процессов с одним и тем же файлом. В этом случае при необходимости процессы могут блокировать доступ к отдельным фрагментам файлов для других процессов. Например, процесс, изменяющий запись в базе данных, перед выполнением изменения может заблокировать участок файла, содержащий эту запись, и затем после выполнения записи разблокировать его. Другие процессы не смогут выполнить запись или чтение для заблокированных участков файла.

Блокировка участка файла выполняется функцией LockFile, прототип которой представлен ниже:

```
BOOL LockFile(
  HANDLE hFile, // идентификатор файла
  DWORD  dwFileOffsetLow,  // младшее слово смещения области
  DWORD  dwFileOffsetHigh, // старшее слово смещения области
  DWORD  nNumberOfBytesToLockLow,   // младшее слово длины
                    // области
  DWORD  nNumberOfBytesToLockHigh); // старшее слово длины
                    // области
```

Параметр hFile задает идентификатор файла, для которого выполняется блокировка области.

Смещение блокируемой области (64-разрядное) задается при помощи параметров dwFileOffsetLow (младшее слово) и dwFileOffsetHigh (старшее слово). Размер области в байтах задается параметрами nNumberOfBytesToLockLow (младшее слово) и nNumberOfBytesToLockHigh (старшее слово). Заметим, что если в файле блокируется несколько областей, они не должны перекрывать друг друга.

В случае успешного завершения функция LockFile возвращает значение TRUE, при ошибке - FALSE. Код ошибки вы можете получить при помощи функции GetLastError.

После использования заблокированной области, а также перед завершением своей работы процессы должны разблокировать все заблокированные ранее области, вызвав для этого функцию UnlockFile:

```
BOOL UnlockFile(
  HANDLE hFile, // идентификатор файла
  DWORD  dwFileOffsetLow,  // младшее слово смещения области
  DWORD  dwFileOffsetHigh, // старшее слово смещения области
  DWORD  nNumberOfBytesToUnlockLow,   // младшее слово длины
                    // области
  DWORD  nNumberOfBytesToUnlockHigh); // старшее слово длины
                    // области
```

Заметим, что в программном интерфейсе операционной системы Microsoft Windows NT есть еще две функции, предназначенные для блокирования и разблокирования областей файлов. Эти функции имеют имена, соответственно, LockFileEx и UnlockFileEx.

Главное отличие функции LockFileEx от функции LockFile заключается в том, что она может выполнять частичную блокировку файла, например, только блокировку от записи. В этом случае другие процессы могут выполнять чтение заблокированной области.

Для экономии места мы не будем описывать эти функции в нашей книге. Всю необходимую информацию вы найдете в документации, которая поставляется вместе с SDK.

Атрибуты файла

В этом разделе мы рассмотрим функции программного интерфейса Microsoft Windows NT, с помощью которых можно определить и изменить различные атрибуты файлов.

Размер файла

Размер файла определить очень просто - достаточно вызвать функцию GetFileSize, прототип которой приведен ниже:

```
DWORD GetFileSize(
  HANDLE  hFile,          // идентификатор файла
  LPDWORD lpFileSizeHigh); // адрес старшего слова для
                    // размера файла
```

Функция GetFileSize возвращает младшее 32-разрядное слово 64-разрядного размера файла с идентификатором hFile. Старшее слово размера файла записывается в переменную типа DWORD, адрес которой передается функции через параметр lpFileSizeHigh.

Если функция завершилась без ошибок, вызванная вслед за ней функция GetLastError возвращает значение NO_ERROR. Если же произошла ошибка, функция GetFileSize возвращает значение 0xFFFFFFFF. При этом в слово, адрес которого задается параметром lpFileSizeHigh, записывается значение NULL. Код ошибки можно определить при помощи все той же функции GetLastError.

Для изменения размера файла вы можете выполнить операцию записи в него или использовать описанные выше функции SetFilePointer и SetEndOfFile.

Набор флагов файла

Так же как и MS-DOS, операционная система Microsoft Windows NT присваивает файлам при их создании различные флаги (атрибуты). Вы можете определить атрибуты файла при помощи функции GetFileAttributes:

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

В качестве единственного параметра этой функции необходимо передать полный или частичный путь к файлу. Функция вернет слово, значение которого является логической комбинацией следующих атрибутов:

Атрибут	*Описание*
FILE_ATTRIBUTE_ARCHIVE	Файл был архивирован (выгружен)
FILE_ATTRIBUTE_COMPRESSED	Файл, имеющий этот атрибут, динамически сжимается при записи и восстанавливается при чтении
FILE_ATTRIBUTE_NORMAL	Остальные перечисленные в этом списка атрибуты не установлены
FILE_ATTRIBUTE_HIDDEN	Скрытый файл
FILE_ATTRIBUTE_READONLY	Файл можно только читать
FILE_ATTRIBUTE_SYSTEM	Файл является частью операционной системы

Для установки новых атрибутов вы можете воспользоваться функцией SetFileAttributes:

```
BOOL SetFileAttributes(
  LPCTSTR lpFileName,       // адрес строки пути к файлу
  DWORD   dwFileAttributes); // адрес слова с новыми
                             // атрибутами
```

Если вам нужно изменить только один из битов слова атрибутов, необходимо вначале получить старое слово атрибутов при помощи функции GetFileAttributes, а затем, изменив в нем только нужные биты, установить новое значение слова атрибутов функцией SetFileAttributes.

Отметки времени для файла

Операционная система Microsoft Windows NT хранит для каждого файла отдельно дату и время его создания, дату и время момента последнего доступа к файлу, а также дату и время момента, когда последний раз выполнялась запись данных в файл.

Всю эту информацию вы можете получить для открытого файла при помощи функции GetFileTime, прототип которой приведен ниже:

```
BOOL GetFileTime(
  HANDLE     hFile,          // идентификатор файла
  LPFILETIME lpCreationTime,   // время создания
  LPFILETIME lpLastAccessTime, // время доступа
  LPFILETIME lpLastWriteTime); // время записи
```

Перед вызовом этой функции вы должны подготовить три структуры типа FILETIME и передать их адреса через параметры lpCreationTime, lpLastAccessTime и lpLastWriteTime. В эти структуры будет записана, соответственно, дата и время создания файла hFile, дата и время момента последнего доступа к файлу, а также дата и время момента, когда последний раз выполнялась запись данных в этот файл.

Структура FILETIME определена следующим образом:

```
typedef struct _FILETIME
{
  DWORD dwLowDateTime;  // младшее слово
  DWORD dwHighDateTime; // старшее слово
} FILETIME;
```

Согласно документации, в структуре FILETIME хранится 64-разрядное значение даты и времени в виде количества интервалов размером 100 наносекунд от 1 января 1601 года.

Что делать с таким представлением даты и времени?

В программном интерфейсе Microsoft Windows NT предусмотрен набор функций, предназначенных для преобразования этого формата времени в более привычные нам форматы и обратно, а также для сравнения значений времени в формате структуры FILETIME.

С помощью функции FileTimeToSystemTime вы можете преобразовать дату и время из формата структуры FILETIME в более удобный для использования формат, определяемый структурой SYSTEMTIME:

```
BOOL FileTimeToSystemTime(
  CONST FILETIME *lpFileTime,  // указатель на структуру
                               // FILETIME
  LPSYSTEMTIME  lpSystemTime); // указатель на структуру
                               // SYSTEMTIME
```

Структура SYSTEMTIME определена так:

```
typedef struct _SYSTEMTIME
{
  WORD wYear;        // год
  WORD wMonth;  // месяц (1 - январь, 2 - февраль, и т. д.)
  WORD wDayOfWeek;    // день недели
            // (0 - воскресение, 1 - понедельник, и т. д.)
  WORD wDay;         // день месяца
  WORD wHour;        // часы
  WORD wMinute;      // минуты
  WORD wSecond;      // секунды
  WORD wMilliseconds; // миллисекунды
} SYSTEMTIME;
```

Обратное преобразование формата времени из формата структуры SYSTEMTIME в формат структуры FILETIME можно сделать при помощи функции SystemTimeToFileTime:

```
BOOL SystemTimeToFileTime(
  CONST SYSTEMTIME *lpSystemTime, // указатель на структуру
                          //   SYSTEMTIME
  LPFILETIME  lpFileTime); // указатель на структуру FILETIME
```

Для установки новых отметок времени файла необходимо воспользоваться функцией SetFileTime:

```
BOOL SetFileTime(
  HANDLE     hFile,          // идентификатор файла
  LPFILETIME lpCreationTime,   // время создания
  LPFILETIME lpLastAccessTime, // время доступа
  LPFILETIME lpLastWriteTime); // время записи
```

Если вам нужно сравнить два значения времени в формате FILETIME, то проще всего это сделать при помощи функции CompareFileTime:

```
LONG CompareFileTime(
  CONST FILETIME *lpTime1,  // адрес первой структуры FILETIME
  CONST FILETIME *lpTime2); // адрес второй структуры FILETIME
```

Если времена и даты, записанные в обеих структурах, равны, функция CompareFileTime возвращает нулевое значение. Если первая дата и вермя больше второго, возвращается значение 1, если же меньше - возвращается отрицательное значение -1.

Для вас могут также представлять интерес еще две функции, выполняющие преобразование формата времени. Это функции FileTimeToDosDateTime и DosDateTimeToFileTime, выполняющие, соответственно, преобразование даты и времени из формата структуры FILETIME в формат, принятый в операционной системы MS-DOS, и обратно. Описание этих функций при необходимости вы найдете в SDK.

Получение информации о файле по его идентификатору

С помощью функции GetFileInformationByHandle вы сможете получить разнообразную информацию об открытом файле по его идентификатору:

```
BOOL GetFileInformationByHandle(
  HANDLE  hFile,                // идентификатор файла
  LPBY_HANDLE_FILE_INFORMATION lpFileInformation ); // адрес
    // структуры, в которую будет записана информация о файле
```

Функция GetFileInformationByHandle записывает информацию о файле в структуру типа BY_HANDLE_FILE_INFORMATION, определенную следующим образом:

```
typedef struct _BY_HANDLE_FILE_INFORMATION
{
  DWORD    dwFileAttributes;     // атрибуты файла
  FILETIME ftCreationTime;       // время создания файла
  FILETIME ftLastAccessTime;     // время доступа к файлу
  FILETIME ftLastWriteTime;      // время записи в файл
  DWORD    dwVolumeSerialNumber; // серийный номер тома
  DWORD    nFileSizeHigh;   // размер файла (старшее слово)
  DWORD    nFileSizeLow;    // размер файла (младшее слово)
  DWORD    nNumberOfLinks;       // количество связей файла
  DWORD    nFileIndexHigh;  // системный номер файла
                  // (старшее слово)
  DWORD    nFileIndexLow;   // системный номер файла
                  // (младшее слово)
```

```
} BY_HANDLE_FILE_INFORMATION;
```

Поле nNumberOfLinks используется приложениями в стандарте POSIX.

Что же касается системного номера файла, то он отличается от идентификатора файла, полученного при помощи функции CreateFile по смыслу и значению. Системные номера файлов являются глобальными и различаются для всех файлов, открытых в системе.

Асинхронные операции с файлами

Когда вы работали с файлами в операционных системах MS-DOS или Microsoft Windows версии 3.1, то вы могли выполнять только синхронные операции с файлами. Это означает, что программа MS-DOS или 16-разрядное приложение Microsoft Windows, вызывая функции для работы с файлами, приостанавливали свою работу до тех пор, пока нужная операция (запись или чтение) не будет выполнена. Это и понятно - если приложение вызывает функцию _lread или _lwrite, она не вернет управление до тех пор, пока не будет завершена, соответственно, операция чтения или записи.

Приложение, запущенное в среде мультизадачной операционной системы Microsoft Windows NT, может совместить операции ввода или вывода с выполнением другой полезной работы. Например, процессор электронных таблиц может пересчитывать одну таблицу во время загрузки другой с диска.

Пользуясь сведениями, полученными из нашей книги, вы и сами без труда сможете организовать такую обработку, например, с помощью отдельных задач, выполняющих пересчет таблиц и загрузку таблиц с диска. Задача, работающая с диском, может открыть файл таблицы (пользуясь для этого функцией CreateFile) и затем выполнить его синхронное чтение функцией ReadFile или даже функцией _lread, если она вам больше нравится. Создавая отдельную задачу для чтения файла, вы и в самом деле сможете совместить в своем приложении выполнение файловых операций с обработкой данных.

Однако операционная система Microsoft Windows NT позволяет решить задачу совмещения файловых операций с другой работой намного проще. Для этого вы должны выполнять файловые операции асинхронно при помощи уже известных вам функций ReadFile и WriteFile.

Как это сделать?

Прежде всего, открывая или создавая файл функцией CreateFile вы должны указать флаг FILE_FLAG_OVERLAPPED. Далее, перед вызовом функций ReadFile или WriteFile вы должны подготовить структуру типа OVERLAPPED и передать ее адрес этим функциям через параметр lpOverlapped.

Структура OVERLAPPED определена следующим образом:

```
typedef struct _OVERLAPPED
{
  DWORD  Internal;     // зарезервировано
  DWORD  InternalHigh; // зарезервировано
  DWORD  Offset;       // младшее слово позиции в файле
  DWORD  OffsetHigh;   // старшее слово позиции в файле
  HANDLE hEvent;   // идентификатор события, который будет
                   // установлен в отмеченное состояние
                   // после завершения операции
} OVERLAPPED;
```

В этой структуре вы должны заполнить поля Offset, OffsetHigh и hEvent. Поля Internal и InternalHigh зарезервированы для использования операционной системой.

В поля Offset и OffsetHigh необходимо записать смещение в файле, относительно которого будет выполняться асинхронная операция записи или чтения. Если для представления смещения достаточно 32 разрядов, в поле OffsetHigh нужно записать значение NULL.

В поле hEvent нужно записать идентификатор созданного предварительно объекта-события, который будет использоваться для синхронизации задач. Если записать в это поле значение NULL, для синхронизации будет использован идентификатор файла. Остановимся на этом подробнее.

Когда функции ReadFile или WriteFile вызывается для выполнения синхронной операции, она возвращает управление только после того, как операция, соответственно, чтения или записи будет завершена. Если же эти функции вызываются в асинхронном режиме, они только инициируют процесс чтения или записи, сразу же возвращая управление, не дожидаясь завершения операции. Для выполнения операции в этом случае операционной системой неявно создается отдельная задача.

Такми образом, задача, вызвавшая функцию ReadFile или WriteFile в асинхронном режиме, продолжит свою работу до завершения файловой операции. Если же этой задаче (или другой задаче) требуется дождаться завершения файловой операции, необходимо использовать средства синхронизации, описанные нами в предыдущей главе.

Самый простой способ синхронизации заключается в том, что задача, вызывающая функции ReadFile или WriteFile в асинхронном режиме, записывает в поле hEvent структуры OVERLAPPED значение NULL. После вызова указанных выше функций, когда задаче нужно дождаться завершения выполнения асинхронной операции, она вызывает функцию WaitForSingleObject, передвая ей в качестве первого параметра идентификатор файла:

```
WaitForSingleObject(hFile, INFINITE);
```

Другой способ предполагает создание отдельного объекта-события, идентификатор которого записывается в поле hEvent структуры

OVERLAPPED при инициализации последней. В этом случае в качестве первого параметра функции WaitForSingleObject следует передать идентификатор этого объекта-события. Как только файловая операция будет завершена, объект-событие перейдет в отмеченное сосотояние.

Третий способ синхронизации заключается в использовании функции GetOverlappedResult. Эта функция обычно используется для проверки результата выполнения асинхронной файловой операции:

```
BOOL GetOverlappedResult(
  HANDLE  hFile,                  // идентификатор файла
  LPOVERLAPPED lpOverlapped,   // адрес структуры OVERLAPPED
  LPDWORD lpNumberOfBytesTransferred, // адрес счетчика байт
  BOOL    bWait);                 // флаг ожидания
```

Через параметры hFile и lpOverlapped передются, соответственно, идентификатор файла, для которого выполнялась асинхронная операция, и адрес адрес структуры OVERLAPPED, подготовленной перед выполнением операции.

В переменную, адрес которой передается функции через параметр lpNumberOfBytesTransferred, записывается количество действительно прочитанных или записанных байт данных.

Параметр bWait может принимать значения TRUE или FALSE. В первом случае функция GetOverlappedResult будет дожидаться завершения выполнения операции (вот вам еще одно средство синхронизации). Если же значение параметра bWait равно FALSE, то если при вызове функции операция еще не завершилась, функция GetOverlappedResult вернет значение FALSE (признак ошибки).

При нормальном завершении (без ошибок) функция GetOverlappedResult возвращает значение TRUE.

В программном интерфейсе операционной системы Microsoft Windows NT есть еще две функции, специально предназначенные для выполнения асинхронных операций с файлами. Это функции ReadFileEx и WriteFileEx:

```
BOOL ReadFileEx(
  HANDLE hFile,               // идентификатор файла
  LPVOID lpBuffer,            // адрес буфера
  DWORD  nNumberOfBytesToRead, // количество байт для чтения
  LPOVERLAPPED lpOverlapped,   // адрес структуры OVERLAPPED
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);//адрес
        // функции, вызываемой после завершения операции

BOOL WriteFileEx(
  HANDLE hFile,               // идентификатор файла
  LPVOID lpBuffer,            // адрес буфера
  DWORD  nNumberOfBytesToWrite, // количество байт для записи
  LPOVERLAPPED lpOverlapped,    // адрес структуры OVERLAPPED
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);//адрес
        // функции, вызываемой после завершения операции
```

Этим функциям через параметр hFile необходимо передать идентификатор файла, созданного или открытого функцией CreateFile с bcgjkmpjdfybtv флагf FILE_FLAG_OVERLAPPED.

Через параметр lpBuffer передается адрес буфера, который будет использоваться функциями ReadFileEx и WriteFileEx, соответственно, для чтения и записи. При этом параметр nNumberOfBytesToRead определяет количество байт, которые будут прочитаны функцией ReadFileEx, а параметр nNumberOfBytesToWrite - количество байт, которые будут записаны функцией WriteFileEx.

Назначение параметра lpOverlapped аналогично назначению этого же параметра в функциях ReadFile и WriteFile.

Через параметр lpCompletionRoutine вы должны передать обеим функциям адрес функции завершения, которая будет вызвана после выполнения операции. Прототип такой функции приведен ниже (имя функции может быть любым):

```
VOID WINAPI CompletionRoutine(
  DWORD dwErrorCode,            // код завершения
  DWORD dwNumberOfBytesTransfered, // количество прочитанных
                  // или записанных байт данных
  LPOVERLAPPED lpOverlapped);  // адрес структуры OVERLAPPED
```

Более подробное описание функций ReadFileEx и WriteFileEx вы найдете в документации, которая поставляется вместе с SDK.

Еще несколько операций с файлами

Очень часто приложениям нужно выполнять такие операции, как копирование, перемещение или удаление файлов. В программном интерфейсе операционной системы Microsoft Windows NT есть удобные функции, предназначенные для выполнения этих операций.

Копирование файла

Для копирования файла вы можете использовать функцию CopyFile:

```
BOOL CopyFile(
  LPCTSTR lpExistingFileName, // адрес пути
                  // существующего файла
```

```
 LPCTSTR lpNewFileName,     // адрес пути копии файла
 BOOL    bFailIfExists);    // флаг перезаписи файла
```

Через параметр lpExistingFileName вы должны передать путь к исходному файлу, кторый будет копироваться. Путь к файлу копии задается с помощью параметра lpNewFileName.

Параметр bFailIfExists определяет действия функции CopyFile, когда указанный в параметре lpNewFileName файл копии уже существует. Если значение параметра bFailIfExists равно TRUE, при обнаружении существующего файла копии функция CopyFile вернет признак ошибки (значение FALSE). Если же значение параметра bFailIfExists равно FALSE, существующий файл копии будет перезаписан.

Перемещение файла

С помощью функции MoveFile вы можете выполнить перемещение файла:

```
BOOL MoveFile(
 LPCTSTR lpExistingFileName, // адрес пути
                 // существующего файла
 LPCTSTR lpNewFileName);    // адрес пути копии файла
```

Параметры lpExistingFileName и lpNewFileName, определяющие, соответственно, пути к старому и новому месторасположению файла, могут указывать на разные дисковые устройства.

Немного большими возможностями обладает другая функция, предназначенная для перемещения файлов, - функция MoveFileEx:

```
BOOL MoveFileEx(
 LPCTSTR lpExistingFileName, // адрес пути
                 // существующего файла
 LPCTSTR lpNewFileName,     // адрес пути копии файла
 DWORD   dwFlags);          // режим копирования
```

Дополнительный параметр dwFlags, определяющий один из режимов копирования, может принимать логическую комбинацию следующих значений:

Значение	*Описание*
MOVEFILE_REPLACE_EXISTING	Перемещение с замещением существующего файла
MOVEFILE_COPY_ALLOWED	Если файл перемещается на другое устройство, для перемещения используются функции CopyFile и DeleteFile (удаление файла). Это значение не совместимо со значением MOVEFILE_DELAY_UNTIL_REBOOT
MOVEFILE_DELAY_UNTIL_REBOOT	Файл будет перемещен только после перезапуска операционной системы Microsoft Windows NT

Режим MOVEFILE_DELAY_UNTIL_REBOOT удобен для создания программ автоматической установки приложений (инсталляторов).

Удаление файла

Для удаления файла вы должны использовать функцию DeleteFile:

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

Параметр lpFileName задает путь к удаляемому файлу.

В случае успеха функция возвращает значение TRUE. При ошибке (например, при попытке удаления файла, открытого этим же или другим процессом), возвращается значение FALSE.

Работа с каталогами

Кратко перечислим несколько функций, предусмотренных в программном интерфейсе Microsoft Windows NT для работы с каталогами.

Определение текущего каталога

С помощью функции GetCurrentDirectory приложение может определить текущий каталог:

```
DWORD GetCurrentDirectory(
 DWORD  nBufferLength, // размер буфера
 LPTSTR lpBuffer);     // адрес буфера
```

Перед вызовом этой функции вы должны подготовить буфер. Через параметр lpBuffer функции GetCurrentDirectory следует передать адрес буфера, а через параметр nBufferLength - размер буфера в байтах.

В случае успеха функция возвращает размер заполненной части буфера, при ошибке - нулевое значение.

Определение системного каталога

При необходимости с помощью функции GetSystemDirectory вы можете определить путь к системному каталогу Microsoft Windows NT (например, C:\WINNT\SYSTEM32, если Microsoft Windows NT установлена на диск C:):

```
UINT GetSystemDirectory(
 LPTSTR lpBuffer, // адрес буфера
```

```
 UINT   uSize);   // размер буфера в байтах
```

Определение каталога Microsoft Windows NT

Функция GetWindowsDirectory позволяет узнать путь к каталогу, в который установлена операционная система Windows (например, C:\WINNT):

```
UINT GetWindowsDirectory(
 LPTSTR lpBuffer, // адрес буфера
 UINT   uSize);   // размер буфера в байтах
```

Изменение текущего каталога

Для изменения текущего каталога используйте функцию SetCurrentDtirecory:

```
BOOL SetCurrentDirectory(LPCTSTR lpszCurDir);
```

Через параметр lpszCurDir вы должны передать функции SetCurrentDtirecory путь к новому каталогу.

Создание каталога

Вы можете создать новый каталог при помощи функции CreateDirectory:

```
BOOL CreateDirectory(
 LPCTSTR lpPathName,  // путь к создаваемому каталогу
 LPSECURITY_ATTRIBUTES lpSecurityAttributes); // дескриптор
                                              // защиты
```

Если вам не нужно определять специальные права доступа к создаваемому каталогу (например, запретить некоторым пользователям удалять каталог), вы можете указать для параметра lpSecurityAttributes значение NULL.

Удаление каталога

Для удаления каталога следует использовать функцию RemoveDirectory:

```
BOOL RemoveDirectory(LPCTSTR lpszDir);
```

Через единственный параметр этой функции следует передать путь к удаляемому каталогу.

Изменение имени каталога

В программном интерфейсе Microsoft Windows NT нет специальной функции, предназначенной для изменения имени каталогов, однако вы можете выполнить эту операцию при помощи функций MoveFile или MoveFileEx, предназначенных для перемещения файлов. Разумеется, при этом новый и старый каталоги должны располагаться на одном и том же диске.

Просмотр содержимого каталога

Для просмотра содержимого каталогов в программном интерфейсе Microsoft Windows NT предусмотрены функции FindFirstFile, FindNextFile и FindClose. Просмотр с помощью этих функций выполняется в цикле.

Перед началом цикла вызовается функция FindFirstFile:

```
HANDLE FindFirstFile(
 LPCTSTR lpFileName,             // адрес пути для поиска
 LPWIN32_FIND_DATA lpFindFileData); // адрес структуры
            // LPWIN32_FIND_DATA, куда будет записана
            // информация о файлах
```

Через параметр lpFileName вы должны передать функции адрес строки, содержащей путь к каталогу и шаблон для поиска. В шаблоне можно использовать символы “?” и “*”.

Через параметр lpFindFileData следует передать адрес структуры типа WIN32_FIND_DATA, в которую будет записана информация о найденных файлах. Эта структура определена следующим образом:

```
typedef struct _WIN32_FIND_DATA
{
 DWORD    dwFileAttributes;      // атрибуты файла
 FILETIME ftCreationTime;        // время создания файла
 FILETIME ftLastAccessTime;      // время доступа
 FILETIME ftLastWriteTime;       // время записи
 DWORD    nFileSizeHigh;     // размер файла (старшее слово)
 DWORD    nFileSizeLow;      // размер файла (младшее слово)
 DWORD    dwReserved0;           // зарезервировано
 DWORD    dwReserved1;           // зарезервировано
 TCHAR    cFileName[MAX_PATH];    // имя файла
 TCHAR    cAlternateFileName[14]; // альтернативное имя файла
} WIN32_FIND_DATA;
```

Если поиск завершился успешно, функция FindFirstFile возвращает идентификатор поиска, который будет затем использован в цикле при вызове функции FindNextFile. При ошибке возвращается значение INVALID_HANDLE_VALUE.

Заметим, что поля cFileName и cAlternateFileName структуры WIN32_FIND_DATA содержат, соответственно, длинное имя файла и короткое, альтернативное имя файла “в формате 8.3”.

После вызова функции FindFirstFile вы должны выполнять в цикле вызов функции FindNextFile:

```
BOOL FindNextFile(
  HANDLE hFindFile,              // идентификатор поиска
  LPWIN32_FIND_DATA lpFindFileData); // адрес структуры
                                 // WIN32_FIND_DATA
```

Через параметр hFindFile этой функции следует передать идентификатор поиска, полученный от функции FindFirstFile. Что же касается параметра lpFindFileData, то через него вы должны передать адрес той же самой структуры типа WIN32_FIND_DATA, что была использована при вызове функции FindFirstFile.

Если функция FindNextFile завершилась успешно, она возвращает значение TRUE. При ошибке возвращается значение FALSE. Код ошибки вы можете получить от функции GetLastError. В том случае, когда были просмотрены все файлы в каталоге, эта функция возвращает значение ERROR_NO_MORE_FILES. Вы должны использовать такую ситуацию для завершения цикла просмотра содержимого каталога.

После завершения цикла просмотра необходимо закрыть идентификатор поиска, вызвав для этого функцию FindClose:

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

В качестве единственного параметра этой функции передается идентификатор поиска, полученный от функции FindFirstFile.

Извещения от файловой системы

Приложение Microsoft Windows NT может динамически следить за содержимым выбранного каталога или даже дерева каталогов, получая от файловой системы извещения при изменении содержимого каталога. Механизм таких извещений основан на использования объектов-событий и функций FindFirstChangeNotification, FindNextChangeNotification, FindCloseChangeNotification.

Прежде всего приложение, которое хочет получать извещения от файловой системы, должно вызвать функцию FindFirstChangeNotification, сообщив ей путь к каталогу и события, при возникновении которых необходимо присылать извещения. Прототип функции FindFirstChangeNotification представлен ниже:

```
HANDLE FindFirstChangeNotification(
  LPCTSTR lpPathName, // адрес пути к каталогу
  BOOL bWatchSubtree, // флаг управления каталогом или деревом
  DWORD dwNotifyFilter); // флаги событий
```

Через параметр lpPathName вы должны передать функции адрес строки, содержащей путь к каталогу, за которым будет следить файловая система и при изменении в котором ваше приложение получит извещение.

Если при вызове функции флаг bWatchSubtree будет равен TRUE, будут отслеживаться изменения не только в каталоге, указанном в параметре lpPathName, но и во всех его подкаталогах. Если же значение этого флага будет равно FALSE, при изменении содержимого подкаталогов ваше приложение не получит извещение.

При помощи параметра dwNotifyFilter вы можете указать события, при возникновении которых в указанном каталоге или дереве каталогов ваше приложение получит извещение. Здесь вы можете указать комбинацию следующих констант:

Константа	*Описание изменений*
FILE_NOTIFY_CHANGE_FILE_NAME	Изменение имен файлов, расположенных в указанном каталоге и его подкаталогах, создание и удаление файлов
FILE_NOTIFY_CHANGE_DIR_NAME	Изменение имен каталогов, создание и удаление каталогов
FILE_NOTIFY_CHANGE_ATTRIBUTES	Изменение атрибутов
FILE_NOTIFY_CHANGE_SIZE	Изменение размеров файлов (после записи содержимого внутренних буферов на диск)
FILE_NOTIFY_CHANGE_LAST_WRITE	Изменение времени записи для файлов (после записи содержимого внутренних буферов на диск)
FILE_NOTIFY_CHANGE_SECURITY	Изменение дескриптора защиты

Функция FindFirstChangeNotification создает объект-событие и возвращает его идентификатор. При ошибке возвращается значение INVALID_HANDLE_VALUE.

Полученный идентификатор может быть использован в операциях ожидания, выполняемых при помощи функций WaitForSingleObject и WaitForMultipleObjects. Когда произойдет одно из событий, указанных функции FindFirstChangeNotification в параметре dwNotifyFilter, этот объект-событие перейдет в отмеченное значение.

Таким образом, приложение может создать задачу, которая вызывает функцию FindFirstChangeNotification и затем выполняет ожидание для

созданного ей объекта синхронизации. При возникновении события эта задача перейдет в активное состояние. При этом она, например, может выполнить сканирование каталога, указанного в параметре lpPathName, для обнаружения возникших изменений.

Заметим, что несмотря на свое название, функция FindFirstChangeNotification не обнаруживает изменения. Она только создает объект синхронизации, который переходит в отмеченное состояние при возникновении таких изменений.

После того как ваше приложение выполнило обнаружение и обработку изменений, оно должно вызвать функцию FindNextChangeNotification:

```
BOOL FindNextChangeNotification(HANDLE hChangeHandle);
```

В качестве параметра этой функции передается идентификатор объекта синхронизации, созданного функцией FindFirstChangeNotification. Функция FindNextChangeNotification переводит объект синхронизации в неотмеченное состояние таким образом, что его можно использовать повторно для обнаружения последующих изменений.

Если ваше приложение больше не собирается получать извещения от файловой системы, оно должно закрыть идентификатор соответствующего объекта синхронизации. Для этого следует вызвать функцию FindCloseChangeNotification, передав ей через единственный параметр идентификатор, полученный от функции FindFirstChangeNotification:

```
BOOL FindCloseChangeNotification(HANDLE hChangeHandle);
```

Информация о файловой системе

В этом разделе мы кратко опишем функции программного интерфейса Microsoft Windows NT, предназначенные для получения информации о дисковых устройствах и состоянии файловой системы.

Определение количества дисковых устройств в системе

Для того чтобы определить список установленных в системе логических дисковых устройств, вы можете вызвать функцию GetLogicalDrives:

```
DWORD GetLogicalDrives(VOID);
```

Эта функция не имеет параметров и возвращает 32-разрядное значение, каждый бит которого отвечает за свое логическое устройство. Самый младший, нулевой бит соответствует устройству с идентификатором A:, бит с номером 1 - устройству с идентификатором A:, и так далее. Если бит установлен, устройство присутствует в системе, если нет - отсутствует.

Более развернутую информацию о составе логических дисковых устройств в системе можно получить при помощи функции GetLogicalDriveStrings:

```
DWORD GetLogicalDriveStrings(
  DWORD  nBufferLength, // размер буфера
  LPTSTR lpBuffer);     // адрес буфера для записи
                        // сведений об устойствах
```

Если вызвать эту функцию с параметрами nBufferLength и lpBuffer равными, соответственно, 0 и NULL, она вернет размер буфера, необходимый для записи информации о всех логических дисковых устройствах, присутствующих в системе. После этого вы можете вызвать функцию GetLogicalDriveStrings еще раз, заказав предварительно буфер нужного размера и указав функции правильный размер буфера.

Функция GetLogicalDriveStrings заполнит буфер текстовыми строками вида:

```
a:\
b:\
c:\
```

Каждая такая строка закрыта двоичным нулем. Последняя строка будет закрыта двумя двоичными нулями.

Определение типа устройства

С помощью функции GetDriveType вы можете определить тип дискового устройства:

```
UINT GetDriveType(LPCTSTR lpRootPathName);
```

В качестве параметра функции GetDriveType нужно передать текстовую строку имени устройства, например, полученную при помощи функции GetLogicalDriveStrings.

В зависимости от типа указанного устройства функция GetDriveType может вернуть одно из следующих значений:

Значение	*Описание*
0	Тип устройства не удалось определить
1	Указанный корневой каталог не существует
DRIVE_REMOVABLE	Устройство со сменным носителем данных
DRIVE_FIXED	Устройство с несменным носителем данных
DRIVE_REMOTE	Удаленное (сетевое) устройство
DRIVE_CDROM	Устройство чтения CD-ROM
DRIVE_RAMDISK	Электронный диск (RAM-диск)

Определение параметров логического устройства

Одним из наиболее интересных параметров логического устройства является размер свободного пространства на нем. Этот параметр вместе с некоторыми другими вы можете определить при помощи функции GetDiskFreeSpace:

```
BOOL GetDiskFreeSpace(
 LPCTSTR lpRootPathName,  // адрес пути к корневому каталогу
 LPDWORD lpSectorsPerCluster,// количество секторов в кластере
 LPDWORD lpBytesPerSector,   // количество байт в секторе
 LPDWORD lpNumberOfFreeClusters, // количество свободных
                    // кластеров
 LPDWORD lpTotalNumberOfClusters); // общее количество
                     // кластеров
```

Перед вызовом этой функции вы должны подготовить несколько переменных типа DWORD и передать функции их адреса. Функция GetDiskFreeSpace запишет в эти переменные параметры логического диска, перечисленные в комментариях к прототипу функции.

Для того чтобы определить размер свободного пространства на диске в байтах, вы должны умножить значение количества свободных кластеров (записанное по адресу lpNumberOfFreeClusters) на количество секторов в кластере (записанное по адресу lpSectorsPerCluster) и на количество байт в одном секторе (которое будет записано по адресу lpBytesPerSector). Более подробно о делении диска на кластеры и секторы вы можете узнать из 19 тома “Библиотеки системного программиста”.

В программном интерфейсе Microsoft Windows NT есть еще одна функция, с помощью которой вы можете определить параметры дискового устройства. Это функция GetVolumeInformation:

```
BOOL GetVolumeInformation(
  LPCTSTR lpRootPathName,  // адрес пути к корневому каталогу
  LPTSTR  lpVolumeNameBuffer,   // буфер для имени тома
  DWORD   nVolumeNameSize, // размер буфера lpVolumeNameBuffer
  LPDWORD lpVolumeSerialNumber, // буфер для серийного номера
                    // тома
  LPDWORD lpMaximumComponentLength, // буфер для максимальной
   // длины имени файла, допустимой для данного тома
  LPDWORD lpFileSystemFlags,     // буфер для системных флагов
  LPTSTR lpFileSystemNameBuffer, // буфер для имени
                    // файловой системы
  DWORD nFileSystemNameSize);    // размер буфера
                    //   lpFileSystemNameBuffer
```

Перед использованием этой функции вы должны подготовить несколько буферов и передать функции их адреса. Функция заполнит буферы параметрами устройства, корневой каталог которого задан параметром lpRootPathName.

В буфере системных флагов, адрес которого передается функции через параметр lpFileSystemFlags, могут быть установлены следующие флаги:

Флаг	*Описание*
FS_CASE_IS_PRESERVED	Система делает различия между заглавными и прописными буквами в именах файлов при записи этих имен на диск
FS_CASE_SENSITIVE	Система делает различия между заглавными и прописными буквами
FS_UNICODE_STORED_ON_DISK	Система может работать с кодировкой Unicode в именах файлов
FS_PERSISTENT_ACLS	Система способна работать со списком контроля доступа к файлам ACL (access-control list). Такая возможность есть в файловой системе NTFS, но отсутствует в файловых системах HPFS и FAT
FS_FILE_COMPRESSION	Файловая система способна сжимать (компрессовать) отдельные файлы
FS_VOL_IS_COMPRESSED	Для тома используется автоматическая компрессия данных

Изменение метки тома

При необходимости вы можете легко изменить метку тома, вызвав для этого функцию SetVolumeLabel:

```
BOOL SetVolumeLabel(
  LPCTSTR lpRootPathName, // адрес пути
                 // к корневому каталогу тома
  LPCTSTR lpVolumeName);  // новая метка тома
```

Если параметр lpVolumeName указать как NULL, метка тома будет удалена.

Прямое управление дисковым устройством

Несмотря на то что набор функций, предназначенных для работы с дисками и файлами в среде Microsoft Windows NT, достаточно мощный и удобный, иногда его возможностей может оказаться недостаточно. Например, если вы разрабатываете систему защиты от несанкционированного копирования дискет, вам могут потребоваться средства нестандартного форматирования дискет. Вам также может потребоваться выполнять из приложения извлечение сменного носителя данных или блокировку последнего в накопителе, проверку замены сменного носителя данных или другие “необычные” операции.

Создавая аналогичную програму MS-DOS, вы можете выполнить нестандартное форматиование, например, с помощью функций BIOS или обращаясь непосредственно к портам дискового контроллера. В среде Microsoft Windows NT оба эти способа непригодны, так как функции BIOS и порты контроллера диска недоступны для обычных приложений, работающих в защищенном режиме.

Выход, однако, есть.

Он заключается в прямом обращении к драйверу дискового устройства с помощью функции управления вводом/выводом, которая называется DeviceIoControl. По своим возможностям он напоминает способ, описанный нами в 19 томе “Библиотеки системного программиста” и связанный с использованием функций GENERIC IOCTL.

Прототип функции DeviceIoControl мы привели ниже:

```
BOOL DeviceIoControl(
  HANDLE  hDevice,        // идентификатор устройства
  DWORD   dwIoControlCode, // код выполняемой операции
  LPVOID  lpInBuffer,     // буфер для входных данных
  DWORD   nInBufferSize,  // размер буфера lpInBuffer
  LPVOID  lpOutBuffer,    // буфер для выходных данных
  DWORD   nOutBufferSize, // размер буфера lpOutBuffer
  LPDWORD lpBytesReturned, // указатель на счетчик
                  // выведенных байт
  LPOVERLAPPED  lpOverlapped); // указатель на
                  // структуру OVERLAPPED
```

Через параметр hDevice вы должны передать идентификатор устройства, полученный от функции CreateFile. Для того чтобы воспользоваться этой функцией для открывания устройства, вы должны указать имя устройства следующим образом (пример приведен для диска C:):

```
hDevice = CreateFile(“\\\\.\\C:”, 0, FILE_SHARE_READ, NULL,
  OPEN_EXISTING, 0, NULL);
```

С помощью параметра dwIoControlCode можно задать один из следующих кодов операции:

Код операции	*Описание*
FSCTL_DISMOUNT_VOLUME	Размонтирование тома
FSCTL_GET_COMPRESSION	Определение состояния компрессии для каталога или файла
FSCTL_LOCK_VOLUME	Блокирование тома
FSCTL_SET_COMPRESSION	Установка состояния компрессии для каталога или файла
FSCTL_UNLOCK_VOLUME	Разблокирование тома
IOCTL_DISK_CHECK_VERIFY	Проверка замены носителя данных для устройства со сменным носителем
IOCTL_DISK_EJECT_MEDIA	Извлечение носителя данных из устройства с интерфейсом SCSI
IOCTL_DISK_FORMAT_TRACKS	Форматирование нескольких дорожек диска
IOCTL_DISK_GET_DRIVE_GEOMETRY	Получение информации о физической геометрии диска
IOCTL_DISK_GET_DRIVE_LAYOUT	Получение информации о всех разделах диска
IOCTL_DISK_GET_MEDIA_TYPES	Получение информации о среде, которую можно использовать для хранения данных в устройстве
IOCTL_DISK_GET_PARTITION_INFO	Полечение информации о разделе диска
IOCTL_DISK_LOAD_MEDIA	Загрузка носителя данных в устройство
IOCTL_DISK_MEDIA_REMOVAL	Включение или отключение механизма извлечения носителя данных
IOCTL_DISK_PERFORMANCE	Получение информации о производительности устройства
IOCTL_DISK_REASSIGN_BLOCKS	Перевод блоков диска в область резервных блоков
IOCTL_DISK_SET_DRIVE_LAYOUT	Создание разделов на диске
IOCTL_DISK_SET_PARTITION_INFO	Установка типа разделов диска

IOCTL_DISK_VERIFY	Выполнение логического форматирования
IOCTL_SERIAL_LSRMST_INSERT	Разрешение или запрещение добавления информации о состоянии линии и модема в поток передаваемых данных

Через параметр lpInBuffer вы должны передать функции DeviceIoControl адрес управляющего блока, необходимого для выполнения опрации. Формат этого блока зависит от кода выполняемой операции. В документации SDK приведены форматы управляющих блоков для всех перечисленных выше кодов операций.

В буфер, адрес которого передается через параметр lpOutBuffer, будет записан результат выполнения операции. Формат этого буфера также зависит от кода операции.

При необходимости с помощью функции DeviceIoControl вы можете выполнять асинхронные операции, подготовив структуру типа OVERLAPPED и передав ее адрес через параметр lpOverlapped. Не забудьте также при открывании устройства указать функции CreateFile флаг FILE_FLAG_OVERLAPPED.

ЛИТЕРАТУРА

1. *Фролов А.В., Фролов Г.В. Библиотека системного программиста. М.: ДИАЛОГ-МИФИ*
   Т.11 - 13. Операционная система Microsoft Windows 3.1 для программиста, 1994
   Т.14. Графический интерфейс GDI в Microsoft Windows, 1994
   Т.15. Мультимедиа для Windows, 1994
   Т.17. Операционная система Microsoft Windows 3.1 для программиста. Дополнительные главы, 1995
   Т.20. Операционная система IBM OS/2 Warp, 1995
   Т.22. Операционная система Windows 95 для программиста, 1996
2. *J. Richter. Advanced Windows NT, Microsoft Press, One Microsoft Way, Redmond, Washington, 1994*
3. *Norton D. Writing Windows Device Drivers. Addison-Wesley Publishing Company, 1992*
4. *Petzold C. Programming Windows 3.1. Microsoft Press. One Microsoft Way. Redmont, Washington, 1992*
5. *Rector B. Developing Windows 3.1 Applications with Microsoft C/C++. SAMS, 1992*
6. *Дженнингс Р. Windows-95 в подлиннике. “BHV-Санкт-Петербург”, С.-П.:, 1995*
7. *M. A. Jacobs. Не все приложения Windows 95 работают под управлением Windows NT //COMPUTER WEEK-MOSCOW, N40, с. 23*
8. *Брайан Просис. Нужно ли отказываться от FAT //PC MAGAZINE/RUSSIAN EDITION, N8,1995, с. 154*
9. *Джефф Просис. Готова ли VFAT выйти на первые роли? //PC MAGAZINE/RUSSIAN EDITION, N9,1995, с. 170*
10. *Чарльз Петцольд. Точное отображение расширенного метафайла //PC MAGAZINE/RUSSIAN EDITION, N7,1995, с. 150*
11. *Джефф Просис. Укрощение строптивых окон //PC MAGAZINE/RUSSIAN EDITION, N6,1996, с. 170*
12. *Брайан Проффит. Высокопроизводительная файловая система HPFS //PC MAGAZINE/RUSSIAN EDITION, N10,1995, с. 184*

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

СОДЕРЖАНИЕ