Министерство науки и высшего образования Российской Федерации
Камчатский государственный университет имени Витуса Беринга

Ю. В. МАРАПУЛЕЦ

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ В WINAPI

Издание второе, исправленное и дополненное

Рекомендовано
Дальневосточным региональным учебно-методическим центром (ДВ РУМЦ) в качестве учебного пособия для студентов направления подготовки бакалавров 01.03.02 "Прикладная математика и информатика" вузов региона

Петропавловск-Камчатский
2019

УДК 681.306
ББК 32.973-018.2
М25

Рецензент
Р. И. Паровик,
кандидат физико-математических наук,
декан физико-математического факультета КамГУ им. Витуса Беринга

Марапулец Ю. В.

М25 Системное программирование в WINAPI. Издание второе, исправленное и дополненное / Ю. В. Марапулец. — Петропавловск-Камчатский: КамГУ им. Витуса Беринга, 2019. —197 с.

ISBN 978-5-7968-0673-9

Целью предложенного пособия является систематизированное изложение принципов и приемов системного программирования в современных операционных системах в соответствии с рабочей программой дисциплины «Системное программирование» для студентов направления подготовки бакалавров 01.03.02 «Прикладная математика и информатика». В качестве базового языка использован язык программирования высокого уровня C++. В книге подробно рассмотрены основы построения базовых элементов современных операционных систем. Особое внимание уделено принципам разработки программ в операционных системах семейства Windows в среде WINAPI.

Учебное пособие предназначено для студентов, изучающих программирование, а также для самостоятельного изучения принципов программирования в среде WINAPI. Во втором издании приведены дополнительные знания, описывающие новые направления развития технологий системного программирования, устранены выявленные ошибки и опечатки.

Издание второе, исправленное и дополненное рекомендовано учебно-методическим советом ФГБОУ ВО «Камчатский государственный университет имени Витуса Беринга» в качестве учебного пособия для студентов, обучающихся по направлению подготовки «Прикладная математика и информатика».

Рекомендовано Дальневосточным региональным учебно-методическим центром (ДВ РУМЦ) в качестве учебного пособия для студентов направления подготовки бакалавров 01.03.02 «Прикладная математика и информатика» вузов региона.

УДК 681.306
ББК 32.973-018.2

ISBN 978-5-7968-0673-9

Введение

Любая современная ПЭВМ состоит из ***аппаратных средств*** и ***программного обеспечения***. Программное обеспечение выполняет функцию посредника между пользователями и ПЭВМ, расширяет возможности аппаратуры вычислительной машины, являясь логическим ее продолжением. Использование развитого программного обеспечения позволяет увеличить производительность вычислительных комплексов, автоматизировать многочисленные рутинные информационные процессы в различных областях человеческой деятельности, повысить производительность труда разработчиков различных систем автоматизированной обработки информации, сократить общие сроки разработок и т. д.

Все программное обеспечение ПЭВМ можно разделить на три составные части:
Системные программы.
Инструментальные системы (среды разработки для программистов).
Прикладные программы.

Системные программы представляют собой комплекс управляющих и обрабатывающих программ, описаний и инструкций, обеспечивающих функционирование вычислительной системы, а также предоставляющих пользователю ПЭВМ определенные услуги. Программы системного программного обеспечения различаются по функциональному назначению и характеру исполнения. Можно выделить следующие виды системных программ:

– ***операционные системы***;

– ***драйверы***, предназначенные для расширения возможностей операционных систем по управлению аппаратными средствами ПЭВМ (клавиатура, монитор, мышь и т. д.);

– ***программы-оболочки***, которые обеспечивают более удобный и наглядный, по сравнению с реализуемым в операционной системе, интерфейс. Наиболее популярными программами-оболочками являются Norton Commander, XTree Pro Gold и PC Shell—для DOS; Far, Windows и Total Commander — для WINDOWS. В целом программы-оболочки обеспечивают:

- работу с файлами и каталогами;
- просмотр файлов различных форматов;
- редактирование файлов различных форматов;
- создание пользовательских меню;
- выдачу сведений о системе и размещении информации на дисках;
- доступ к пользовательскому интерфейсу ОС;
- автоматическое восстановление состояния оболочки после завершения работы прикладной программы;

– ***операционные оболочки***, среди которых наиболее популярными являются WINDOWS ver. 1.0–3.1, DES Qview и Ensemble для — ОС DOS; Presentation Manager — для OS/2 ver. 1.0–2.0; Motif и Ten/Plus — для UNIX. Операционные оболочки в отличие от программ-оболочек не только дают пользователю более наглядные средства для выполнения часто используемых действий, но и представляют новые возможности для запускаемых программ:

- графический интерфейс;
- одновременное выполнение нескольких программ для однозадачных систем;
- расширенные средства для обмена информацией между программами;

– ***вспомогательные программы-утилиты***, которые предоставляют пользователям различные необходимые услуги:

- обслуживание магнитных и оптических дисков;
- обслуживание файлов и каталогов;
- создание и обновление архивов;
- предоставление пользователю информации об аппаратных и программных средствах ПЭВМ и их диагностику;
- шифрование информации;

- защиту от компьютерных вирусов;
- автономную печать (программы-спулеры);
- управление памятью и т. д.

Операционной системой (ОС) называют комплекс программ, обеспечивающий управление ***ресурсами*** ПЭВМ и ***процессами***, использующими эти ресурсы при вычислениях. При этом под «ресурсом» следует понимать любой логический или физический компонент ПЭВМ и предоставляемые им возможности. Далее понятия«процесс»и«ресурс» будут рассмотрены более подробно. Управление ресурсами сводится к выполнению следующих функций:

- упрощению доступа к ресурсам;
- распределению ресурсов между процессами.

Реализация первой функции позволяет «спрятать» аппаратные особенности ПЭВМ и тем самым предоставить в распоряжение пользователей и программистов виртуальную машину с существенно облегченным управлением. Функция распределения ресурсов присуща только операционным системам, которые обеспечивают одновременное выполнение (или по крайней мере одновременное хранение в ОЗУ) нескольких программ, которые принято называть ***вычислительными процессами*** (или просто «процессами»).

Понятие **процесс** является одним из основных при рассмотрении операционных систем. Отдельный процесс — это выполнение конкретной программы с ее данными на процессоре. Концептуально процесс рассматривается в двух аспектах: во-первых, он является носителем данных, и, во-вторых, он (одновременно) выполняет операции, связанные с их обработкой.

Термин **ресурс** обычно применяется по отношению к повторно используемым, относительно стабильным и часто недостающим объектам, которые запрашиваются, используются и освобождаются процессами в период их активности. Другими словами, ***ресурсом называется всякий объект, который может распределяться внутри системы***. Ресурсы могут быть разделяемыми, когда несколько процессов могут их использовать одновременно (в один и тот же момент времени) или параллельно (в течение некоторого интервала времени процессы используют ресурс попеременно), а могут быть и неделимыми (рис.1) |2|.

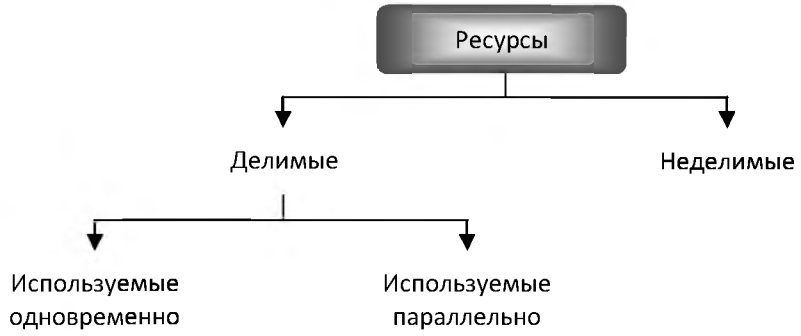

Рис. 1. Классификация ресурсов

При разработке первых систем ресурсами считались ***процессорное время***, ***память***, ***каналы ввода/вывода*** и ***периферийные устройства***. Однако очень скоро понятие ресурса стало гораздо более универсальным и общим. Различного рода ***программные*** и ***информационные ресурсы*** также могут быть определены для системы как объекты, которые могут разделяться и распределяться и доступ к которым необходимо соответствующим образом контролировать. ***В настоящее время понятие ресурса превратилось в абстрактную структуру с целым рядом атрибутов, характеризующих способы доступа к этой структуре и ее физическое представление в системе*** [2]. Более того, помимо системных ресурсов стали толковать как ресурс и такие объекты, как ***сообщения*** и ***синхросигналы***, которыми обмениваются задачи.

В первых вычислительных системах любая программа могла выполняться только после полного завершения предыдущей, поскольку все подсистемы и устройства компьютера управлялись исключительно центральным процессором. Центральный процессор осуществлял и выполнение

вычислений, и управление операциями ввода/вывода данных. Соответственно, пока осуществлялся обмен данными между оперативной памятью и внешними устройствами, процессор не мог выполнять вычисления. Введение в состав вычислительной машины специальных контроллеров позволило совместить во времени (распараллелить) операции вывода полученных данных и последующие вычисления на центральном процессоре. Однако все равно процессор продолжал часто и долго простаивать, дожидаясь завершения очередной операции ввода/вывода. Поэтому было предложено организовать так называемый ***мультипрограммный*** (***мультизадачный***) режим работы вычислительной системы. Суть его заключается в том, что пока одна программа (один вычислительный процесс или задача) ожидает завершения очередной операции ввода/вывода, другая программа (а точнее, другая задача) может быть поставлена (рис. 2, 3) |2|.

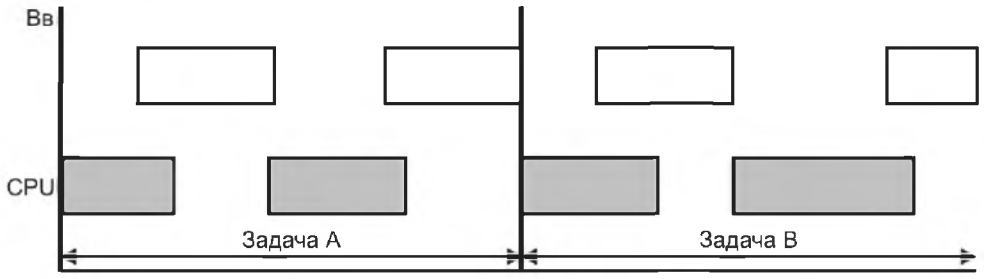

Рис. 2. Пример выполнения двух программ в однопрограммном режиме

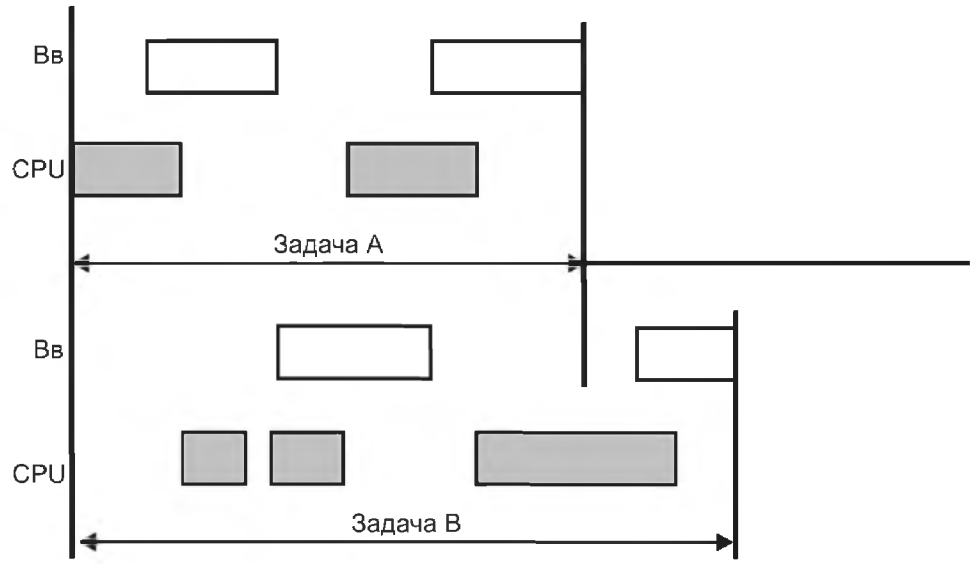

Рис. 3. Пример выполнения двух программ в мультипрограммном режиме

Из рис. 2, 3 видно, что благодаря совмещению во времени выполнения двух программ общее время выполнения двух задач получается меньше, чем если бы мы выполняли их по очереди. Из этих же рисунков видно, ***что время выполнения каждой задачи в общем случае становится больше***, чем, если бы мы выполняли каждую из них как единственную.

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме (всякое разделение ресурсов замедляет работу одного из участников за счет дополнительных затрат времени на ожидание освобождения ресурса).

Общая схема выделения ресурсов такова. При необходимости использовать какой-либо ресурс задача обращается к ***супервизору операционной системы*** посредством специальных вызовов (команд, директив) и сообщает о своем требовании. При этом указывается вид ресурса и, если надо, его объем (например, количество адресуемых ячеек оперативной памяти, количество дорожек или секторов на системном диске, устройство печати и объем выводимых данных и т. п.). Директива обращения к операционной системе передает ей управление, переводя процессор в привилегированный режим работы, если такой существует.

Ресурс может быть выделен задаче, обратившейся к супервизору с соответствующим запросом, если [2]:

- он свободен и в системе нет запросов от задач более высокого приоритета к этому же ресурсу;
- текущий запрос и ранее выданные запросы допускают совместное использование ресурсов;
- ресурс используется задачей низшего приоритета и может быть временно отобран (разделяемый ресурс).

Получив запрос, операционная система либо удовлетворяет его и возвращает управление задаче, выдавшей данный запрос, либо, если ресурс занят, ставит задачу в очередь к ресурсу, переводя ее в состояние ожидания (блокируя). После окончания работы с ресурсом задача опять с помощью специального вызова супервизора сообщает операционной системе об отказе от ресурса или операционная система забирает ресурс сама. Супервизор операционной системы освобождает ресурс и проверяет, имеется ли очередь к освободившемуся ресурсу. Если очередь есть, то в зависимости от принятой ***дисциплины обслуживания*** и приоритетов заявок он выводит из состояния ожидания задачу, ждущую ресурс, и переводит ее в состояние готовности к выполнению. После этого управление либо передается данной задаче, либо возвращается к той, которая только что освободила ресурс.

При выдаче запроса на ресурс задача может указать, хочет ли она владеть ресурсом монопольно или допускает совместное использование с другими задачами. Например, с файлом можно работать монопольно, а можно и совместно с другими задачами. Если в системе имеется некоторая совокупность ресурсов, то управлять их использованием можно на основе определенной стратегии. Стратегия подразумевает четкую формулировку целей, следуя которым можно добиться эффективного распределения ресурсов.

При организации управления ресурсами всегда требуется принять решение о том, что в данной ситуации выгоднее: быстро обслуживать отдельные наиболее важные запросы, предоставлять всем процессам равные возможности либо обслуживать максимально возможное количество процессов и наиболее полно использовать ресурсы.

Рассмотрим кратко **основные виды ресурсов вычислительной системы** и способы их разделения (см. рис. 1). Прежде всего, одним из важнейших ресурсов является сам ***процессор***, точнее, ***процессорное время***. Обычно процессорное время делится попеременно (видимость параллельности).

Вторым видом ресурсов вычислительной системы можно считать память. ***Оперативная память*** может быть разделена и одновременным способом, и попеременно(в зависимости от размера свободной памяти). Когда говорят о *внешней памяти* (например, память на магнитных дисках), то собственно *память и доступ к ней считаются разными видами ресурса*. Каждый из этих ресурсов может предоставляться независимо от другого. Но для полной работы с внешней памятью необходимо иметь оба этих ресурса. Собственно внешняя память может разделяться одновременно, а доступ к ней — попеременно. Если говорить о *внешних устройствах*, то они, как правило, *могут разделяться параллельно*, если используются механизмы прямого доступа. Если же устройство работает с последовательным доступом, то оно не может считаться разделяемым ресурсом.

Очень важным видом ресурсов являются ***программные модули***. Прежде всего следует рассматривать системные программные модули, поскольку именно они обычно и рассматриваются как программные ресурсы и могут быть распределены между выполняющимися процессами.

Программные модули могут быть *однократно* и *многократно* (или повторно) используемыми. Однократно используемыми называют такие программные модули, которые могут быть правильно выполнены только один раз. Это означает, что в процессе своего выполнения они могут испортить себя: повреждаются либо часть кода, либо исходные данные, от которых зависит ход вычислений. Очевидно, что ***однократно используемые программные модули*** являются ***неделимым ресурсом***. Более того, их обычно вообще не распределяют как ресурс системы. Системные однократно используемые программные модули используются, как правило, только на этапе загрузки ОС. При этом следует иметь в виду тот очевидный факт, что собственно двоичные файлы, которые обычно хранятся на системном диске и в которых записаны эти модули, не портятся, а потому могут быть повторно использованы при следующем запуске ОС.

Повторно используемые программные модули, в свою очередь, могут быть ***непривилегированными***, ***привилегированными*** и ***реентерабельными***.

Привилегированные программные модули работают в так называемом привилегированном режиме, т. е. при отключенной системе прерываний, так, что никакие внешние события не могут нарушить естественный порядок вычислений. В результате программный модуль выполняется до своего конца, после чего он может быть вновь вызван на исполнение из другой задачи (другого вычислительного процесса). Таким образом, такой модуль будет выступать как попеременно разделяемый ресурс. Структура привилегированных программных модулей изображена на рис. 4 [2]. Здесь в первой секции программного модуля выключается система прерываний. В последней секции, напротив, эта система включается.

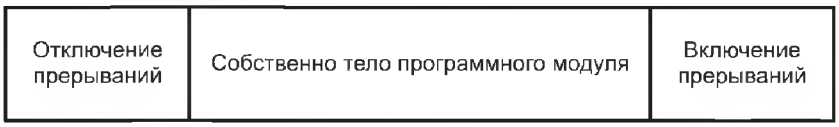

Рис. 4. Структура привилегированного программного модуля

Непривилегированные программные модули — это обычные программные модули, которые могут быть прерваны во время своей работы. Следовательно, в общем случае их нельзя считать разделяемыми, потому что если после прерывания его запустить еще раз по требованию другого вычислительного процесса, то промежуточные результаты для прерванных вычислений могут быть потеряны. В противоположность этому ***реентерабельные программные модули*** допускают повторное многократное прерывание своего исполнения и повторный их запуск по обращению из других задач. Для этого реентерабельные программные модули должны быть созданы таким образом, чтобы было обеспечено сохранение промежуточных вычислений для прерываемых вычислений и возврат к ним, когда вычислительный процесс возобновляется с прерванной ранее точки (рис.5) [2].

Рис. 5. Структура реентерабельного программного модуля

Основная идея построения и работы реентерабельного программного модуля (см. рис. 5) заключается в том, что в первой (головной) своей части с помощью обращения из системной привилегированной секции осуществляется запрос на получение в системной области памяти блока ячеек, необходимого для размещения всех текущих (промежуточных) данных. При этом на вершину стека помещается указатель на начало области данных и ее объем. Все текущие переменные программного модуля в этом случае располагаются в системной области памяти. Поскольку в конце привилегированной секции система прерываний включается, то во время работы центральной (основной) части реентерабельного модуля возможно ее прерывание. Если прерывание не возникает, то в третьей (заключительной) секции осуществляется запрос на освобождение использованного блока системной области памяти. Если же во время работы центральной секции возникает

прерывание и другой вычислительный процесс обращается к тому же самому реентерабельному программному модулю, то для этого нового процесса вновь заказывается новый блок памяти в системной области памяти и на вершину стека записывается новый указатель. Очевидно, что возможно многократное повторное вхождение в реентерабельный программный модуль до тех пор, пока в области системной памяти, выделяемой специально для реентерабельной обработки, есть свободные ячейки, число которых достаточно для выделения нового блока.

Еще одним видом ресурсов являются **информационные ресурсы**, которые могут существовать как в виде переменных, находящихся в оперативной памяти, так и в виде файлов. Если процессы используют данные только для чтения, то такие информационные ресурсы можно разделять, в других случаях необходимо специальным образом организовывать работу с такими данными.

В настоящее время существует большое разнообразие операционных систем. В целом их можно ***классифицировать*** по следующим признакам [4]:

– по количеству пользователей, одновременно обслуживаемых системой;

– числу процессов, которые могут одновременно выполняться под управлением ОС;

– типу доступа пользователя к ПЭВМ;

– типу средств вычислительной техники, для управления ресурсами которых система предназначена.

В соответствии с первым признаком различают ***однопользовательские*** и ***многопользовательские*** ОС. Многопользовательские системы поддерживают одновременную работу на ЭВМ нескольких пользователей за различными терминалами. Второй признак делит ОС на ***однозадачные*** и ***многозадачные***. В соответствии с третьим признаком ОС делятся:

– ***на системы с пакетной обработкой***, когда из программ, подлежащих выполнению, формируется пакет, который предъявляется ПЭВМ (в данном случае пользователи непосредственно с ОС не взаимодействуют);

– ***системы разделения времени***, обеспечивающие одновременный диалоговый (интерактивный) доступ к ЭВМ нескольких пользователей через терминалы. Ресурсы ЭВМ при этом выделяются каждому пользователю по очереди в соответствии с той или иной дисциплиной обслуживания. Данный тип ОС предназначен для наиболее эффективного использования ресурсов ЭВМ;

– ***системы реального времени***, которые должны обеспечивать гарантированное время ответа на внешние события. Такие ОС служат для управления внешними по отношению к ЭВМ процессами и объектами.

По четвертому признаку ОС делятся на ***однопроцессорные***, ***многопроцессорные***, ***сетевые*** и ***распределенные***.

Следует отметить, что ОС не могут предоставить пользователям возможности, которыми не обладает ПЭВМ. Они в состоянии только эффективно использовать аппаратные средства компьютера. Наиболее популярными ОС, созданными в различное время для ПЭВМ, являются CP/M, DOS, UNIX, OS/2, WINDOWS, LINUX.

Предлагаемый курс «Системное программирование» посвящен главной задаче современного программирования — изучению принципов и получения опыта разработки программ как составных элементов операционных систем. Основные примеры программного кода построены на функциях WIN API для операционных систем семейства Windows как наиболее популярных в настоящее время в Российской Федерации.

Глава 1
Принципы разработки программ в современных ОС

§ 1.1. Операционные системы для ПЭВМ

Первоначально для IBM-совместимых ПЭВМ были разработаны следующие классы операционных систем:

- ОС семейства CP/M;
- ОС семейства DOS;
- ОС семейства OS/2;
- ОС семейства UNIX.

В дальнейшем, с развитием ПЭВМ, ОС CP/M перестала использоваться на этапе перехода к архитектуре IBM PC, поддержка OS/2 закончилась к концу 90-х годов, а остальные классы операционных систем получили свое развитие в более совершенных современных системах. Так, на смену Dos пришла операционная система Windows различных модификаций, на смену UNIX — LINUX, QNX и др. Рассмотрим более подробно основные классы ОС.

1.1.1. ОС CP/M

Система CP/M (Control Program for Microcomputers — управляющая программа для микрокомпьютеров) была разработана в 1974 г. фирмой Digital Research и исторически является одной из первых ОС для ПЭВМ. Она предназначена для управления ресурсами 8-разрядных персональных компьютеров на основе МП 8080. Это однозадачная ОС, состоящая из нескольких компонентов, что позволяет достаточно легко адаптировать ее к архитектурным особенностям той или иной машины путем перекодирования только одного компонента, а именно BIOS.

Развитием CP/M явилась система CP/M-86, предназначенная для ПЭВМ класса XT. Дальнейшее совершенствование ОС CP/M привело к появлению многозадачной системы CCP/M-86, а затем и многопользовательской ОС MP/M-86. В рамках данного семейства ОС было создано большое число программ для ПЭВМ, включая трансляторы языков Бейсик, Паскаль, Си, Фортран, Кобол, Ада, а также текстовые редакторы, СУБД, графические пакеты и др.

Достоинство систем данного класса состояло в предельной простоте, малой потребной емкости ОЗУ для работы (20 Кбайт), а также возможности быстрой настойки на разные конфигурации ПЭВМ. Однако следует отметить, что представители семейства CP/M были довольно примитивны и имели слабый командный язык наряду с простейшей файловой системой. Поэтому на 16-разрядных ПЭВМ они нашли весьма ограниченное применение и дальнейшего развития не получили.

1.1.2. ОС DOS

Первый представитель ОС ***семейства DOS*** — система MS-DOS (Microsoft Disk Operating System— дисковая операционная система фирмы Microsoft) была выпущена в 1981 г. в связи с появлением ПЭВМ IBM PC и очень напоминала систему CP/M. Каждая новая версия DOS появлялась, как правило, в связи с созданием новых аппаратных средств. Номер версии состоит из двух чисел, разделенных точкой. Первое число обозначает основную редакцию, второе — ее модификацию. Так, DOS 2.1 сильно отличается от DOS 1.1, но очень похожа на DOS 2.0. Первые версии DOS были не русифицированы, и только в начале декабря 1989 г. фирма IBM зарегистрировала для СНГ кодовую страницу с номером 866, разработанную СП «Диалог» и фирмой Microsoft при участии фирм IBM и HP, а также зарезервировала номера 867 и 868 для других языков Содружества. Далее перейдем к рассмотрению наиболее существенных особенностей различных версий DOS:

– DOS 1.00. Появилась в связи с созданием IBM PC. Подобна CP/M, но предназначена для МП 8088. Поддерживает только односторонние 133-миллиметровые 8-секторные (160-Кбайт) НГМД;

– DOS 1.05. Устраняет ряд ошибок, обнаруженных в DOS 1.00;DOS 1.10. Была стандартом более года. Дополнительно к предыдущей версии поддерживает двухсторонние 133-миллиметровые 8-секторные (320-Кбайт) НГМД;

– DOS 2.00. Появилась в связи с созданием IBM PC XT. Поддерживает НЖМД емкостью до 10 Мбайт. Дополнительно к предыдущей версии ОС обслуживает 133-миллиметровые 9-секторные односторонние (180-Кбайт) и двухсторонние (360-Кбайт) НГМД. Поддерживает древовидную файловую структуру. Реализует концепции стандартного ввода / вывода, перенаправления ввода / вывода и фильтров. Обрабатывает следующие новые команды: FC (только MS-DOS), BACKUP, RESTORE, TREE, CD, MD, RD, PATH и др. Имеет расширенный язык командных файлов за счет новых команд GO TO, IF, ECHO и др. Реализует возможность подключения (установки) внешних драйверов устройств. Обеспечивает фоновую печать по команде PRINT. Поддерживает видеосистему CGA;

– DOS 2.10. Создана для IBM PCjr. Основана на DOS 2.00 и устраняет обнаруженные в ней ошибки;

– DOS 3.00. Появилась в связи с созданием IBM PC AT. За счет указания маршрута поиска позволяет выполнять программы из файлов, которые находятся не в рабочем каталоге. Поддерживает НЖМД емкостью до 20 Мбайт. Дополнительно к предыдущим версиям обслуживает двухсторонние 133-миллиметровые 15-секторные (1,2-Мбайт) НГМД. Обрабатывает новые команды ATTRIB, LABEL, SELECT, KEYBxx, SHARE, GRAFTABL, COUNTRY. Поддерживает виртуальный диск в ОЗУ;

– DOS 3.10. Имеет некоторые сетевые средства. Поддерживает новые команды JOIN и SUBST;

– DOS 3.20.Создана для IBM PC Convertible. Дополнительно поддерживает 89-миллиметровые(720-Кбайт) НГМД. Обрабатывает новые команды REPLACE и XCOPY. Поддерживает усовершенствованные команды ATTRIB, COMMAND, FORMAT, SELECT, GRAPHICS, SHELL. Препятствует непреднамеренному форматированию жесткого диска. Поддерживает драйвер DRIVER.SYS для создания фиктивных дисководов;

– DOS 3.30. Появилась в связи с созданием семейства PS/2 и способна функционировать на моделях семейства PC. Поддерживает концепцию разбиения жестких дисков любого объема на логические диски размером до 32 Мбайт каждый, которые можно использовать одновременно (все они доступны DOS). Дополнительно обслуживает 89-миллиметровые(1,44-Мбайт) НГМД. Содержит усовершенствованные средства для поддержки национальных языков (введено понятие кодовой страницы). Поддерживает новые команды APPEND, CALL, CHCP, FASTOPEN и NLSFUNC, а также усовершенствованные команды DATE, TIME, ATTRIB, BACKUP, FDISK, RESTORE и XCOPY. Реализует усовершенствованный язык командных файлов;

– DOS 4.00. Поддерживает логические диски на винчестере размером свыше 32 Мбайт. Использует отображаемую память для буферов ОС и структур данных команды FASTOPEN (требуется EMS 4.0). Позволяет задействовать для размещения резидентных программ первые 64 Кбайт расширенной памяти (HMA-память). Обеспечивает расширенную поддержку национальных языков. Обрабатывает новую команду MEM, а также усовершенствованные команды APPEND, ATTRIB, BACKUP, COUNTRY, MODE, FASTOPEN, FDISK, GRAPHICS, GRAFTABL, NLSFUNC, REPLACE, SELECT, TREE, DEL и др. Содержит усовершенствованные драйверы устройств ANSI.SYS, DISPLAY.SYS, DRIVER.SYS и PRINTER.SYS. Наконец-то полностью поддерживает все режимы работы видеосистем EGA и VGA;

– DOS 4.01. Содержит графическую оболочку MS-DOS Shell, поддерживающую манипулятор «мышь»;

– DOS 5.00. Обеспечивает размещение своего ядра, а также драйверов и резидентных программ в верхней памяти. Способна работать с 89-миллиметровыми (2,88 Мбайт) НГМД. Непосредственно (без загрузки SHARE) поддерживает логические диски на винчестере размером свыше 32 Мбайт. Обрабатывает новые команды DELOLDOS, DOSKEY, EXPAND, LOADHIGH, MIRROR, SETVER, UNDELETE и UNFORMAT, а также усовершенствованные команды DIR, FORMAT, SYS и др. Имеет встроенную справочную систему. Содержит улучшенную систему программирования Basic. Отличается высокой надежностью в работе;

– DOS 6.0. Включает средства сжатия информации на дисках (DOUBLE SPACE), Программы создания резервных копий, антивирусную программу и другие усовершенствования.

– DOS 6.20. Содержит усовершенствованную версию средств сжатия информации на дисках;

– DOS 6.21. Версия с изъятой по судебному решению (иск фирмы Stack Electronics) программой динамического сжатия дисков (DOUBLE SPACE);

– DOS 6.22. Содержит «подправленную» версию DOUBLESPACE, не нарушающую патент.

ОС семейства DOS являются однозадачными, но имеют и некоторые элементы многозадачности. В частности, можно организовать фоновую печать на принтере, а также разместить в ОЗУ несколько резидентных программ и активизировать их при необходимости.

Все версии DOS совместимы снизу вверх (т. е. программа, разработанная для младшей версии, в подавляющем большинстве случаев будет работать и под управлением более старшей версии ОС). ОС семейства DOS могли работать на всех классахIBM-совместимых ПЭВМ. Для операционной системы DOS было написано огромное количество прикладных программ, многие из которых продолжают использоваться и в настоящее время.

В целом ОС семейства DOS обладают следующими характерными чертами:

– поддержкой командных файлов, что обеспечивает возможность создания пользовательских макрокоманд;

– поддержкой иерархической (древовидной) файловой структуры;

– возможностью не только последовательного, но и прямого доступа к содержимому файлов;

– трактовкой на логическом уровне устройств ввода / вывода как файлов, что унифицирует средства обмена информацией с любыми ПУ и файлами;

– наличием конвейеров (средств передачи вывода одной программы или команды на вход другой) и возможностью перенаправления ввода / вывода на уровне командного языка;

– некоторыми средствами поддержки сетей ЭВМ;

– модульностью структуры, упрощающей перенос системы на другие типы ПЭВМ;

– небольшим потребным объемом оперативной памяти для работы (около 60 Кбайт) и внешней памяти для хранения системных файлов;

– возможностью создания в памяти виртуальных дисков, что ускоряет обмен информацией;

– возможностью запуска фоновых задач;

– поддержкой национальных алфавитов и соглашений.

Наряду с достоинствами ОС семейства DOS имеют и ряд недостатков. Наиболее существенные из них — полное отсутствие средств защиты от несанкционированного доступа к ресурсам ПЭВМ и к самой ОС, однозадачность, неудобный интерфейс пользователя, для улучшения которого необходимо применять программы оболочки (Norton Commander) и операционные оболочки (первые версии Windows). В настоящее время с переходом на 32-разрядную адресацию памяти DOS используется только в ОС Windows 9x как средство поддержки ранее выпущенного программного обеспечения. Дальнейшим развитием ОС семейства DOS явилось создание ОС Windows линеек NT и 9x, которые мы рассмотрим далее.

1.1.3. ОС OS/2

В связи с созданием в 1987 г. нового семейства ПЭВМ PS/2 фирмой IBM совместно с Microsoft была разработана ***многозадачная ОС второго поколения, названная OS/2*** (Operating System/2 — операционная система/2). Она создавалась как преемник DOS и имела схожий с последней пользовательский интерфейс. Система OS/2 управляет МП 80286 в защищенном режиме, а поэтому может применяться только на ПЭВМ с МП 80286 и выше. Первые версии этой системы позволяли программам использовать физическую память размером до 16 Мбайт и виртуальную до 0,5 Гбайт на каждую задачу. С целью выполнения программ, разработанных для DOS, в рамках OS/2 может быть запущена эта ОС в качестве подзадачи. Система OS/2 обеспечивала одновременную работу до 12 программ, но только одну программу DOS.

Первоначально были выпущены три основные версии OS/2:

– MS OS/2 Standard Edition — стандартная версия для ПЭВМ типа PC AT, AT-386, AT-486 и PS/2 с такими же МП; как раз эта версия разработана IBM совместно с Microsoft;

– IBM OS/2 Standard Edition, предназначенная только для моделей семейства PS/2 и IBM-совместимых ПЭВМ с шиной MCA, снабженных МП 80286 — 486; она была разработана корпорацией IBM и ориентирована именно на шину MCA (в отличие от предыдущей версии);

– OS/2 Extended Edition— расширенная версия, применяемая также только на MCA-ПЭВМ и созданная фирмой IBM; она содержит систему управления базой данных DB2, а также другие программные продукты, поддерживаемые IBM.

При этом каждая версия имела две редакции:

– 1.0 — стандартный продукт;

– 1.1 — ОС, включающая графическую интерфейсную систему PM, поддерживающая файлы и логические диски на винчестере размером более 32 Мбайт, а также содержащая текстовый редактор и ряд систем программирования.

В редакциях 1.2 (IBM) и 1.21 (Microsoft) реализована концепция загружаемой (устанавливаемой) файловой системы (Installable File System—IPS) и несколько изменен «внешний вид» PM. Реализация концепции IPS обеспечивала возможность подключения во время загрузки ОС дополнительной файловой системы:

– файловая система высокой производительности(High Performance File System — HPFS), рассчитанная на быстрый доступ к сверхбольшим файлам, базам данных, а также к большому числу файлов в каталогах;

– UNIX — совместимая файловая система;

– файловая система для доступа к данным на оптических дисках(типа CD-ROM).

Стандартная же файловая система OS/2 совместима с файловой системой DOS. Первоначально пользовательский интерфейс OS/2 был похож на DOS, однако интерфейсная система PM превращала его в графический (как, впрочем, и Windows для DOS).

К дополнительным достоинствам OS/2 следует отнести:

– поддержку динамической компоновки программных модулей в единую исполняемую программу, т. е. формирование программы во время ее выполнения, а не предварительно. Динамическая компоновка позволяет включить в программу только требуемые для данного конкретного выполнения модули;

– поддержку виртуальной памяти.

Несмотря на широкую рекламу OS/2 ее разработчиками — фирмами IBM и Microsoft, пользователи отнеслись к ней с большой осторожностью, что объясняется следующими обстоятельствами:

– консерватизмом пользователей, имеющим объективные причины, связанные с необходимостью приобретения новой ОС и ее освоения;

– невысокой реактивностью(быстродействием) OS/2;

– большим объемом ОЗУ, необходимым для ее работы (сама OS/2 занимает около 500 Кбайт, при этом ПЭВМ должна иметь память объемом не менее 2 Мбайт), а также невозможностью хранения системы на гибком диске, а следовательно, и загрузки с него;

– недоработками в самой ОС;

– небольшим количеством имеющихся системных и прикладных программных средств для этой ОС (если же использовать ПО, разработанное для DOS, то его эффективность заметно упадет вследствие функционирования одновременно двух ОС).

Анализ этих причин показывает, что главной причиной малого распространения OS/2 в то время явился низкий уровень развития технических средств самих ПК. В связи с неудачами OS/2 «в недрах»IBM и Microsoft сформировались определенные пути их преодоления. Они оказались различными, что привело к разрыву отношений между двумя бывшими союзниками.

Компания IBM продолжала делать ставку на OS/2 и уже в 1991 г. выпустила 32-разрядную OS/2 версии 2.0. Далее IBM заключила союз с компанией Apple и создала объектно-ориентированную ОС нового поколения, которая получила название OS/2 Warp 3.0. Эта операционная система имела уже полностью графический многооконный интерфейс и на несколько лет опередила другие операционные системы в данном направлении. Дальнейшим развитием ОС была версия OS/2 Warp 4.0. Политика же фирмы Microsoft была направлена в область развития ОС DOS в совокупности с интерфейсной системой Windows, которая принесла Microsoft небывалый финансовый успех.

К сожалению, в настоящее время ОС OS/2 Warp используется редко и только для высокопрофессионального применения. Главной причиной этого является упадок компании IBM, а так же давление фирмы Microsoft на производителей программного обеспечения различного назначения и, как следствие этого, неразвитое ПО для ОС OS/2.

1.1.4. ОС UNIX

По мере развития технических средств ПЭВМ растет и популярность ОС *семейства UNIX*. Первый представитель этого семейства был разработан в 1969 г. для мини-ЭВМ К. Томпсоном при участии Д. М. Ритчи, М. Д. Макилроя и Дж. Ф. Осанна — сотрудников американской фирмы Bell Laboratories, являющейся филиалом американской корпорации AT&T. Интересен тот факт, что работы по ОС UNIX начались вопреки желаниям администрации этой фирмы. Созданная система получилась настолько удачной, что впоследствии стала стандартом "de facto" для ЭВМ промышленных и научных организаций. Основные концепции, заложенные в UNIX, воплощаются во многие новые ОС.

Основой работы UNIX является ядро операционной системы. Это обычный выполняемый файл, лежащий в директории, типа /unix, или /stand/unix, или /vmunix, или /vmlinuz (в зависимости от конкретной реализации). При старте системы он целиком грузится в память, постоянно там находится и выполняет все системные функции. В ядре находятся драйверы устройств, возможно, и ненужные, подпрограммы управления системными ресурсами, таблицы текущих процессов и открытых файлов, системные вызовы, т. е. обработчики системно-зависимых функций. Аналогом последних является 21 прерывание в MS-DOS или многочисленные экспортируемые функции в DLL в Windows.

Любые операции с дисками кэшируются в памяти — это так называемый буферный кэш. Процессы, т. е. все выполняемые программы, записываются в таблице процессов. В целях экономии памяти в UNIX всегда загружается только одна копия выполняемой программы; также имеются разделяемые библиотеки (аналогом является DLL), позволяющие иметь только одну копию некоторых функций. Каждый процесс имеет свое виртуальное адресное пространство. Для того чтобы обеспечить возможность поддерживать программы, требующие оперативной памяти, которая превосходит их физический объем, UNIX поддерживает swapping. Область памяти программы делится на три части: собственно код программы, статические и динамические данные. Можно потребовать по возможности не выгружать программу после завершения, установив бит «навязчивости». В этом случае при последующей загрузке программа будет запущена почти мгновенно.

Файловая система UNIX представляет собой одно большое дерево. Каждый раздел диска имеет свое дерево, а все такие деревья сцепляются в одно. Корнем является директория "/". В файловую систему входит и директория /dev, в которой находится описание физических и логических устройств, таких как жесткие диски, принтер, TTY и некоторые другие. Поскольку для быстроты файловая система кэшируется, то выключение питания может привести к разрушению файловой системы.

Большинство UNIX-подобных систем являются многопользовательскими и обладают следующими характерными чертами:

- поддержка иерархической файловой структуры с монтируемыми дисковыми томами;
- наличие конвейеров и средств перенаправления ввода / вывода;
- наличие средств коммуникации в локальных и других вычислительных сетях;
- поддержка широкого разнообразия периферийных устройств идентичным, с файловой трактовкой, образом;
- наличие множества полезных стандартных и дополнительных утилит;
- наличие встроенных инструментальных систем;
- применение средств парольной защиты;
- высокая мобильность вследствие модульности ОС и использование для ее разработки языка программирования C/C++;

– открытость для модификаций и расширений;
– эффективные средства электронной почты и передачи данных;
– поддержка виртуальной памяти со страничным запросом.

Спустя довольно непродолжительное время после своего возникновения UNIX стала развиваться по двум направлениям. Первое (коммерческое) поддерживалось USL (UNIX System Laboratory), второе (некоммерческое) было проектом, осуществляемым в Berkley. Первое направление доросло до SVR, второе — до BSD. Однако USL засудила BSD за использование фрагментов кода с копирайтом AT&T, вследствие чего многие коммерческие UNIX перешли на линию SVR. Рассмотрим основные современные типы UNIX для PC:

1) некоммерческие:
– Net BSD;
– Free BSD 2.1 — наиболее подходит для работы в сети. В нем полностью заново переделан код ядра, где мог бы возникнуть конфликт с USL;
– Linux — самый популярный среди бесплатных UNIX;
– Qnx Realtime Platform;
2) коммерческие:
– BSD/OS 2.0 BSD i/386 — недорогой;
– Solaris — копия с SUN;
– SCO UNIX.

Рассмотрим подробнее двух представителей семейства ОС UNIX, которым в последние годы уделяется наибольшее внимание — Linux и Qnx.

1.1.5. ОС LINUX

Linux — свободно распространяемая версия UNIX, первоначально разработанная Линусом Торвальдсом в Университете Хельсинки (Финляндия).Linux был создан с помощью многих UNIX-программистов и энтузиастов из Internet, тех, кто имеет достаточно навыков и способностей развивать систему. Ядро Linux не использует коды AT&T или какого-либо другого частного источника, и большинство программ Linux разработаны в рамках проекта GNU из Free Software Foundation в Cambridge, Massachusetts. Следует отметить, что в разработку Linux внесли лепту программисты всего мира.

Вообще Linux создавался Линусом Торвальдсом как хобби. Его вдохновила операционная система Minix — маленькая UNIX-система, созданная Andy Tanenbaum. Впервые Linux обсуждался по компьютерной сети в рамках USENET newsgroupcomp.os.minix. В этих обсуждениях прежде всего принимали участие пользователи Minix из учебных и научных заведений, которым хотелось чего-то большего, чем Minix.

Раннее развитие Linuxбыло связано прежде всего с проблемой переключения задач в защищенном режиме для 80386. Все писалось на ассемблере. Относительно появления Linux версии 0.01 никогда не делалось никаких официальных заявлений. Исходные тексты 0.01 не давали даже нормального выполняемого кода: они фактически состояли лишь из набора заготовок для ядра и молчаливо предполагали, что вы имеете доступ к Minix-машине, чтобы иметь возможность компилировать их и совершенствовать. Первая «официальная» версияLinux 0.02 была объявлена Торвальдсом 5 октября 1991 г. В это время Linux уже мог выполнять *bash* (the GNU Bourne Again Shell) и *gcc* (the GNU C compiler), хотя работал он крайне ненадежно. После версии 0.03 Торвальдс скачком перешел в нумерации к версии 0.10, так как над проектом работало уже довольно много программистов-энтузиастов. После нескольких последовавших пересмотров версий Линус присвоил очередной версии номер 0.95, чтобы тем самым отразить свое впечатление о том, что скоро возможна уже «официальная» версия (обычно программам не дают номер версии 1.0 до тех пор, пока она не будет теоретически завершена и отлажена).

Сегодня Linux — это полноценная ОС семейства UNIX, способная работать с X Windows, TCP/IP, Emacs, UUCP, mail и USENET. Практически все важнейшие программные пакеты были поставлены и на Linux, т. е. для Linux теперь доступны и коммерческие пакеты. Все большее разно-

образие оборудования поддерживается по сравнению с первоначальным ядром. Linux поддерживает большинство свойств, присущих другим реализациям UNIX, а также ряд тех, которых больше нигде нет.Linux — это полная многозадачная многопользовательская операционная система (точно также, как и другие версии UNIX).Linux достаточно хорошо совместима с рядом стандартов для UNIX (насколько можно говорить о стандартизации UNIX) на уровне исходных текстов, включая IEEE POSIX.1, System V и BSD. Она создавалась, имея в виду такую совместимость. Большинство свободно распространяемых по сети Internet программ для UNIX может быть откомпилировано для LINUX практически без особых изменений. Кроме того, все исходные тексты для Linux, включая ядро, драйверы устройств, библиотеки, пользовательские программы и инструментальные средства, распространяются свободно.

Другие специфические внутренние черты Linux включают: контроль работ по стандарту POSIX (используемый оболочками, такими как *csh* и *bash*), псевдо терминалы (*pty*), поддержку национальных и стандартных клавиатур динамически загружаемыми драйверами клавиатур. Linuxтакже поддерживает виртуальные консоли (*virtual consoles*), которые позволяют «переключать экраны» на консоли в текстовом режиме.

Ядро ОС может само эмулировать команды 387-FPU, так что системы без сопроцессора способны выполнять программы, на него рассчитывающие (т. е. с плавающей точкой).

Linux поддерживает различные типы файловых систем для хранения данных. Некоторые файловые системы, такие как файловая система ext2fs, были созданы специально для Linux. Поддерживаются также другие типы файловых систем, такие как Minix-1 и Xenix, ISO 9660 CD-ROM для работы с дисками CD-ROM MS-DOS на жестком диске. Реализована файловая система MS-DOS, позволяющая прямо обращаться к файлам MS-DOS на жестком диске.

Linux обеспечивает полный набор протоколов TCP/IP для сетевой работы. Это включает драйверы устройств для многих популярных карт Ethernet, SLIP (Serial Line Internet Protocol, обеспечивающих доступ по TCP/IP при последовательном соединении), PLIP (*Parallel Line Internet Protocol*), PPP (*Point-to-Point Protocol*), NFS (*Network File System*) и т. д. Поддерживается весь спектр клиентов и услуг TCP/IP, таких как FTP, telnet, NNTP и SMTP.

Ядро Linux сразу создано с учетом специального защищенного режима для процессоров Intel 80386 и 80486. В частности, Linux использует парадигму описания памяти в защищенном режиме и другие новые свойства процессоров. Ядро Linux поддерживает загрузку только нужных страниц, т. е. с диска в память загружаются те сегменты программы, которые действительно используются. Возможно использование одной страницы, физически один раз загруженной в память, несколькими выполняемыми программами.

Для увеличения объема доступной памяти Linux осуществляет также разбиение диска на страницы, т. е. на диске может быть выделено до 256 Мбайт «пространства для свопинга» (*swap space*). Когда системе нужно больше физической памяти, то она с помощью свопинга выводит неактивные страницы на диск. Это позволяет выполнять более объемные программы и обслуживать одновременно больше пользователей. Однако свопинг не исключает наращивания физической памяти, поскольку он снижает быстродействие, увеличивает время доступа.

Ядро ОС также поддерживает универсальный пул памяти для пользовательских программ и дискового кэша. При этом для кэша может использоваться вся память, и наоборот, кэш уменьшается при работе больших программ. Выполняемые программы используют динамически связываемые библиотеки, т. е. выполняемые программы могут совместно использовать библиотечную программу, представленную одним физическим файлом на диске (иначе, чем это реализовано в механизме разделяемых библиотек SunOS). Это позволяет выполняемым файлам, особенно тем, которые многократно используют библиотечные функции, занимать меньше места на диске. Есть также статические связываемые библиотеки для тех, кто желает пользоваться отладкой на уровне объектных кодов или иметь «полные» выполняемые программы, которые не нуждаются в разделяемых библиотеках. В Linux разделяемые библиотеки динамически связываются во время выполнения, позволяя программисту заменять библиотечные модули своими собственными.

Для обеспечения отладки ядро Linux выдает дампы памяти для «посмертного» анализа. Использование дампа и динамических отладчиков позволяет определить причины краха программы.

Наиболее популярной современной версией Linux является Ubuntu, которая ориентирована на удобство и простоту использования. Изначально ОС Ubuntu создавалась как временное ответвление от ОС Debian, как и в ней, в Ubuntu в основном используется свободное программное обеспечение. В настоящее время Ubuntu финансируется Марком Шаттлвортом и основанной им компанией Canonical. Она включает использование утилиты sudo, которая позволяет пользователям выполнять администраторские задачи, не запуская потенциально опасную сессию суперпользователя и имеет развитую интернационализацию. ОС Ubuntu при установке на жёсткий диск требуется от пяти гигабайт свободного пространства, а для корректной работы рекомендуется от 512 мегабайт оперативной памяти.

1.1.6. ОС QNX

Другой ОС, являющейся «дальним родственником» семейства UNIX, является ***Qnx*** — операционная система реального времени, существующая уже более 20 лет. Разработкой данной системы занимается канадская фирма QNX Software Systems, Ltd.

Первая операционная система реального времени QNX 2 была разработана по заказу Министерства обороны США и очень долго и успешно применялась во многих отраслях. Позже ее сменила QNX 4, пришедшая в начале 90-х гг. и в Россию уже как доступная всем полнофункциональная коммерческая операционная система. Эта система, поражающая своей компактностью и возможностями — с графической оболочкой Photon, браузером Voyager, средствами соединения с Internet и пакетом документации, загружаемая всего с одной (!) дискеты емкостью 1,44 Мбайта — завоевала умы и сердца очень многих пользователей и разработчиков, особенно если учесть, что работать с ней можно было на компьютере с процессором 80386 и с 4 Мбайтами памяти. К сожалению, эта система отпугивала пользователей высокой для «домашнего» компьютера стоимостью.

И вот, наконец, свершилось — 24 апреля 2000 г. компания QSSL анонсирует появление системы нового уровня, бесплатной для конечного пользователя и разработчика при соблюдении условий ее некоммерческого использования. Названная QNX Realtime Platform, эта система целиком и полностью оправдывает свое название — в ней реализовано лучшее из того опыта, что был накоплен фирмой за два десятилетия. По своим возможностям она превышает своего собрата QNX 4, уже называемого «младшим». По встраиваемости это система бьет все возможные рекорды — для нормальной работы можно ограничиться только ядром, запускающим один единственный процесс, далее работающий с нитями (в QNX 4 в любом случае необходимо запускать менеджер процессов). Ядро же системы QNX RtP занимает всего 32 килобайта.

В настоящий момент эта система встает в один ряд с Linux, FreeBSD и BeOS. В ней реализована компактная графическая оболочка Photon micro GUI 2.0, которая гораздо меньше громоздкой X-Windows и намного более дружественная. К тому же здесь есть встроенная поддержка запуска оригинальных X-приложений — пакет XPhoton. Под Photon существует специальный пакет для визуальной разработки собственных приложений — Photon Application Builder, который предоставляет для разработчика ПО значительно более удобный и дружественный интерфейс, чем Delphi или Visual C++.

1.1.7. ОС WINDOWS

Вскоре после появления в середине 1981 г. IBM PC стало очевидно, что господствующей операционной системой для PC (включая совместимые) должна стать MS-DOS. Ранние версии MS-DOS обеспечивали для пользователя интерфейс командной строки, отображая такие команды, как DIR и TYPE, которые могли загружать выполняемые программы в оперативную память и предлагали для этих программ определенный интерфейс для доступа к файлам, считывания информации с клавиатуры, отображения на принтере и на экране дисплея (только в символьном режиме).

Из-за ограниченных возможностей программного и аппаратного обеспечения псевдографическая среда пробивала себе дорогу медленно. Компьютеры Apple показали возможную альтернативу, когда в январе 1983 г.появилась скандально известная ОС Lisa, а затем в январе

1984 г.Apple, разработав Macintosh, создала образцовую графическую среду (несмотря на постепенную утрату этой моделью компьютера своих позиций на рынке), все еще рассматриваемую как эталон, на который равняются создатели любой другой графической оболочки.

О работе над Windows корпорация Microsoft заявила в ноябре 1983 г. (позже, чем появилась Lisa, но раньше, чем Macintosh) и реализовала ее двумя годами позже, в ноябре 1985 г. В течение двух следующих лет Microsoft Windows версии 1.0 претерпела несколько модернизаций, необходимых для удовлетворения требований международного рынка. Кроме этого, появились дополнительные драйверы для новых дисплеев и принтеров.

Windows версии 2.0 была создана в ноябре 1987 г. Эта версия содержала несколько изменений пользовательского интерфейса. Наиболее важное из этих изменений касалось использования перекрывающихся окон вместо окон, расположенных рядом, что было характерно для Windows версии 1.x. Кроме того, Windows версии 2.0 содержала также улучшенный интерфейс клавиатуры и манипулятора «мышь», а также отчасти окон меню и диалога.

В то время для Windows требовались только процессоры Intel 8086 или 8088, работающие в реальном режиме, при этом доступ осуществлялся к 1 Мбайту оперативной памяти. Windows/386 (созданная вскоре после Windows 2.0) использовала виртуальный режим процессора Intel 80386 для запуска нескольких одновременно работающих с оборудованием программ MS-DOS в окнах. Для симметрии Windows версии 2.1 назвали Windows/286.

Windows версии 3.0 появилась 22 марта 1992 г. Здесь были объединены ранние версии Windows/286 и Windows/386. Главным изменением в Windows 3.0 была поддержка защищенного режима процессоров Intel 80286, 80386 и 80486. Это позволило Windows и ее приложениям получить доступ к 16 Мбайтам оперативной памяти. «Оболочка» программ Windows для запуска программ и поддержки файлов была полностью переделана. Windows 3.0 — это первая версия Windows, которая стала «родной» для множества пользовательских машин в домах и офисах.

Windows версии 3.1 появилась в апреле 1992 г. В нее были включены такие важные свойства, как технология TrueType для шрифтов (что дало возможность масштабировать шрифты для Windows), multimedia (звук и музыка), OLE и диалоговые окна общего пользования. Кроме этого,Windows 3.1 работала только в защищенном режиме, требовала процессора 80286 или 80386 и по крайней мере одного мегабайта оперативной памяти. После Windows 3.1 компания Microsoft выпустила в продажу несколько других вариантов ОС, включая Windows for Workgroups 3.1 и 3.11, известную как Windows 3.11. С точки зрения разработчика прикладных программ, все эти варианты мало чем отличались друг от друга.

Windows NT, появившаяся в июле 1993 г., стала первой версией Windows, поддерживающей 32-разрядную модель программирования для процессоров Intel 80386 и 80486, а также Pentium. Windows NT имела сплошное плоское (flat) 32-разрядное адресное пространство и 32-разрядные целые. В то время многие программисты возлагали на новую версию Windows большие надежды, однако появление NT стало для них сильным разочарованием. Новая операционная система от Microsoft оказалась громоздкой и тяжеловесной. Она требовала, чтобы компьютер был оснащен как минимум 16 Мбайт оперативной памяти (в то время лишь немногие обладали компьютерами с объемом ОЗУ более 8 Мбайт). Установить NT могли лишь счастливые обладатели устройств чтения компакт-дисков. Тогда это было дорогое удовольствие, и приобрести привод CD-ROM могли лишь избранные. Даже если компьютер и обладал необходимой конфигурацией, зачастую, установив NT, программисты сталкивались с двумя основными проблемами этой операционной системы: отсутствием поддержки многих аппаратных устройств и недостаточным количеством прикладных программ. Таким образом, сразу после появления NT круг поклонников этой операционной системы был относительно узким. Большинство пользователей продолжало использовать Windows 3.1. Поставщики программного и аппаратного обеспечения не испытывали особого желания начать широкомасштабную поддержку NT. Конечно, в рабочей среде NT могли использоваться многие программы, созданные для Windows 3.1, но далеко не все. Следует отметить, что новая операционная система Microsoft стоила значительно дороже, чем Windows 3.1, кроме того, дополнительные финансовые вложения могли потребоваться и для модернизации компьютера. При создании

Windows NT компания Microsoft включила в состав этой операционной системы модуль Windows-On-Windows (WOW), благодаря которому в NT можно было запускать приложения Windows 3.1, однако инженеры Microsoft были неприятно удивлены. Оказалось, что огромное количество существовавших на тот момент программ в процессе своей работы использовали ошибки и недокументированные возможности интерфейса Win16 API. Такие программы зачастую отказывались работать в среде WOW, так как этот модуль в точности соответствовал документации.

Тем не менееMicrosoft приняла решение не отказываться от своего детища и продолжать работу над улучшением новой операционной системы(как в случае с первыми версиями Windows). Через некоторое время было выпущено несколько новых, улучшенных версий NT. Все же руководство компании понимало, что для достижения успеха необходимо привлечь в мир NT сторонних независимых разработчиков программного обеспечения.

Решением проблемы стала операционная система Windows 95 (первоначально условно названная Chicago), которая появилась в августе 1995 г. Вместо того чтобы выпустить на рынок Windows 4.0, компания Microsoft решила удивить мир совершенно новым продуктом. Руководство компании страстно желало, чтобы покупатели думали о Windows 95 как о Windows NT Lite, т. е.как об облегченной версии NT. Однако на самом деле Windows 95 больше напоминала Windows 3.1, которую накачали нелегальными стероидами. Основным козырем Windows 95 было то обстоятельство, что эта операционная система обладала программным интерфейсом, сходным с Windows NT. Таким образом, сторонние разработчики программ могли без дополнительных усилий создавать приложения, с одинаковым успехом работающие как в Windows 95, так и в NT. Компания Microsoft рассчитывала, что это приведет к стремительному увеличению количества программных продуктов для NT. Чтобы подобный замысел сработал, необходимо было обеспечить массовый переход пользователей Windows 3.1 на использование Windows 95. В результате при проектировании Windows 95 были применены некоторые весьма спорные дизайнерские решения, которые сложно обосновать, если только не принимать во внимание необходимость обеспечения полной совместимости с Windows 3.1.

В процессе разработки Windows 95 компания Microsoft решила ни в коем случае не повторять подход, использованный при создании WOW. Дело в том, что пользователи домашних компьютеров очень неохотно идут на обновление операционной системы, если у них есть хотя бы малейшие сомнения в том, что их любимые программы смогут нормально работать после установки новой ОС. Исходя из этого было принято решение в точности воспроизвести Win 16 API (программный интерфейс Windows 3.1) в составе Windows 95. При этом, чтобы лишний раз не «напрягаться», Microsoft решила не разрабатывать новый Win16, а использовать уже существующий. В этом случае можно было не опасаться проблем, связанных с совместимостью: фактически все существующее программное обеспечение Windows 3.1 сможет работать в Windows 95, так как обе эти операционные системы, по большому счету, обладают одинаковой начинкой. Мало того, в Windows 95 смогут работать старые драйверы аппаратных устройств. Конечно, Microsoft внесла в систему некоторые изменения. Кое-какие компоненты были полностью переписаны в 32-битном коде (например, отображение шрифтов и управление памятью). Но в целом система почти полностью сохранила свое внутреннее строение.

Но почему тогда Windows 9x называют 32-битной операционной системой? Потому, что в ее состав входит интерфейс Win32 API. Этот интерфейс преобразует вызовы Win32 в вызовы Win16. Данный подход позволяет разрабатывать программы, которые работают как в Windows 95, так и в NT, но при этом многие элементы NT отсутствуют в Windows 95 или функционируют по-другому. Кроме того, распределение памяти в Windows 95 существенно отличается от NT. Это сделано для того, чтобы обеспечить совместимость с программами Win16.

Стратегия, избранная Microsoft, сработала великолепно. После появления Windows 95 количество 32-битных прикладных программ стало увеличиваться с невероятной скоростью. Модернизация программ для Windows 3.1 таким образом, чтобы они смогли работать с новым интерфейсом Win 95/NT, была легкой работой, и многие программисты решили этим воспользоваться, вливаясь в общий поток желающих попасть на новый рынок.

Тем временем индустрия аппаратного обеспечения не стояла на месте. Компьютеры развивались почти столь же стремительно, как и операционные системы. Вскоре процессоры класса Pentium стали общей нормой, а цены на микросхемы оперативной памяти снижались с каждым

днем. Многие высокопроизводительные видеокарты и принтеры оснащались собственными ОЗУ, а компьютеры с объемом оперативной памяти 64 Мбайт и более перестали быть редкостью. Такие компьютеры могли с легкостью обеспечить работу NT, и многие пользователи переключились на использование NT 3.5 или 3.51. Снижение цен на многопроцессорные материнские платы также стало причиной того, что многие обратили внимание на NT. Ведь эта операционная система способна поддерживать работу с несколькими процессами, а Windows 95 всегда использует только один процессор.

Повышению популярности Windows NT также способствовал Интернет. Windows NT — неплохой сетевой сервер, а компания Microsoft включила в комплект поставки этой операционной системы все необходимое для того, чтобы организовать работу полноценного узла Всемирной сети. Единственной серьезной проблемой NT было то, что, с точки зрения пользователя, версия NT 3.51 внешне сильно напоминала Windows 3.1. Привыкшие к удобствам Windows 95 пользователи желали иметь дело с таким же простым в использовании интерфейсом, средствами Plug and Play и другими возможностями, доступными только в среде Windows 95, но отсутствующими в NT. Проблема была решена с появлением Windows NT 4.0.

Параллельно с Windows NT 4.0 на рынок потребительских ОС была выпущена Windows 98, которая явилась продолжением развития линейки ОС Windows 95. Наиболее существенным отличием данной ОС от Windows 95 явилось наличие встроенных средств связи с Internet. К 2000 г. компания Microsoft выпустила на рынок два своих новых продукта. В продолжение линейки NT была выпущена Windows 2000, а в продолжение и завершение линейки 9x — Windows Millenium. К 2001 г. подавляющее большинство пользователей домашних компьютеров продолжали использовать Windows 95 или Windows 98 (Windows Mellenium не была принята большинством пользователей ПК, в первую очередь в связи с частыми отказами системы). Основная часть корпоративных пользователей, разработчиков и квалифицированных пользователей перешла с Windows NT на Windows 2000, которая обладала следующими основными особенностями:

– была рассчитана на рабочие станции и серверы, а также на применение в центрах обработки данных;

– отказоустойчивость — плохо написанные программы не могли привести к краху системы;

– защищенность (несанкционированный доступ к ресурсам, например файлам или принтерам, управляемым этой системой, был невозможен);

– обладала богатым набор средств и утилит для администрирования системы в масштабах организации;

– ядро Windows 2000 было написано в основном на C и C++, поэтому система легко переносилась на процессоры с другими архитектурами;

– поддерживала многопоточность и мультипроцессорную обработку, обеспечивая высокую масштабируемость системы;

– имела высокоэффективную подсистему управления памятью с широкими возможностями;

– позволяла расширять функциональность за счет динамически подключаемых библиотек (DLL);

– файловая система Windows 2000 давала возможность отслеживать, как пользователи манипулируют с данными на своих ПЭВМ.

Как уже было сказано, Windows 98 являлась операционной системой потребительского класса. Она обладала многими возможностями Windows 2000, но некоторые ключевые из них не поддерживала, а именно:

– Windows 98 не являлась отказоустойчивой (приложение вполне способно привести к краху системы);

– Windows 98 была менее защищена, поддерживала аппаратные платформы только с одним процессором;

Но почему вообще на момент начала XXI в. существовало ядро Windows 98? Ответ очень прост — Windows 98 была более дружественна к пользователю, чем Windows 2000.

Компания Microsoft активно продолжала работать над созданием операционной системы, более дружественной к пользователю, и в результате в 2001 г. объявила о выпуске ОС ***Windows XP*** (от англ. *experience* — опыт), известной также под кодовым наименованием ***Microsoft Codename Whistler***. Первоначально в планы корпорации Microsoft входила разработка двух независимых опе-

рационных систем нового поколения. Первый проект получил рабочее название ***Neptune*** — эта ОС должна была стать очередным обновлением ***Windows Millennium Edition***, новой системой линейки Windows 9X. Второй проект, называвшийся ***Odyssey***, предполагал создание ОС на платформе Windows NT, которая должна была прийти на смену ***Windows 2000***. Однако руководство Microsoft посчитало нецелесообразным рассредоточивать ресурсы на продвижение двух разных ОС, вследствие чего оба направления разработок были объединены в один проект — ***Microsoft Whistler***. Главной задачей при разработке данной операционной системы было сохранение всех основных особенностей Windows 2000 и создание при этом более дружественного интерфейса для пользователя. Следует отметить, что компании Microsoft это удалось. С учетом вложения компанией Microsoft более миллиарда долларов в рекламную компанию **Windows XP** в 2002–2003 гг. ОС Windows XP в 2003–2005 гг. побила все рекорды продаж в области ОС. К 2006 г. Windows 9x осталась только на устаревших компьютерах, которые по техническим данным не позволяли установить более современную систему. ***Windows XP*** для настольных ПК и рабочих станций выпускалась в трех модификациях: ***Home Edition*** — для домашних персональных компьютеров, ***Professional Edition*** — для офисных ПК и, наконец, ***Microsoft Windows XP 64bit Edition*** — для персональных компьютеров, собранных на базе 64-битных процессоров.

Удивляет демократичность при установке ОС, вплоть до выбора типа файловой системы (FAT32 или NTFS), хотя в данном случае при выборе FAT32 меняется главная сущность линейки NT — защищенность на уровне файловой системы. Несмотря на ряд спорных решений, компания Microsoft, создав Windows XP, достигла главного — смогла сдвинуть рядового пользователя домашних систем с Windows 98 в Windows XP.

В 2003 г. компанией Microsoft была анонсирована новая операционная система Windows Server 2003, созданная как операционная система нового поколения для серверов и основанная на концепции ***Microsoft Windows.NET.*** Данная система пришла на смену Windows 2000 Server, Advanced Server и Datacenter Server.

В этом же 2003 г.Microsoft объявляет о начале работы над ОС нового поколения — **Microsoft Windows Longhorn** (позднее система получила новое название — **Wista)**. Главная особенность системы — новая файловая система, созданная Microsoft в качестве замены для FAT32 и NTFS. Это ***Windows Future Storage***, или, говоря коротко, ***WinFS***. Главная особенность файловой системы — абстрагирование пользователя и приложений от физического расположения информации. В каком-то смысле она работает наподобие обычной базы данных: где бы ни находился нужный файл, для нахождения и доступа к нему требуется лишь выдача операционной системы запроса, характеризующего искомый объект. Длина файла, его имя, специфические признаки — все это может служить в качестве компонентов такого запроса. Результат — максимально быстрый доступ к данным вне зависимости от их типов и местонахождения, а также замечательное свойство: все файлы, доступные с одного компьютера, можно увидеть сразу, а не разбросанными по папкам. Попросту говоря, запустив MP3-плейер, можно увидеть сразу все поддерживаемые им файлы, доступные с данной машины, без необходимости искать их по директориям. Новыми являлись также поисковая система и новый пользовательский интерфейс, известный под названием Plex. Это в первую очередь две панели: одна — привычная панель задач, вторая — опциональная панелька, дополняющая первую для большего удобства работы с системой, а также использование векторной графики для прорисовки иконок.

Следует отметить, что выпуск новой тяжеловесной системы от Microsoft откладывался более трех лет, и недаром. Множество недоработок, многие спорные решения привели к тому, что данная система стала главным «долгостроем» от Microsoft. И наконец, только в начале 2007 г., после многократного переноса сроков выпуска, компания Microsoft выпустила ОС Windows Vista. В связи с неоднозначностью решений, принятых корпорацией Microsoft в ходе разработки данной ОС, многие издания весьма негативно высказывались об этой ОС. Например, ОС Windows Vista заняла первое место в конкурсе «Провал года», проводимом сайтом Pwnie award за 2008 год. С учетом такого «провального» выхода новой ОС, компания Microsoft срочно приступила к разработке новой ОС и меньше, чем через три года, 22 октября 2009 года в продажу поступила Windows 7. В линейке Windows NT система имела номер версии 6.1 (Windows 2000 — 5.0, Windows XP — 5.1, Windows Server 2003 — 5.2, Windows Vista и Windows Server 2008 — 6.0).

Windows 7 оказалась очень удачной ОС. С июля 2011 до марта 2017 года она занимала лидирующее положение по количеству пользователей в мире. В 2017 году компания Microsoft объявила, что поддержка Windows 7 будет прекращена 14 января 2020 года. Платные обновления расширенной безопасности (ESU), продаваемые отдельно для каждого устройства, будут предлагаться до января 2023 года. На смену Windows 7 пришла Windows 8, которая в отличие от своих предшественников, использовала новый интерфейс под названием Metro. Вместо меню «Пуск» в данном интерфейсе используется «активный угол», нажатие на который открывает стартовый экран. Прокрутка в Metro-интерфейсе идет горизонтально. Плитки на стартовом экране можно перемещать и группировать. В зависимости от разрешения экрана система автоматически определяет количество строк для плиток. Интерфейс Metro был ориентирован на сенсорный экран, но не исключал использования на несенсорных ПК. Основная идея данного интерфейса заключалась в широком использовании ОС Windows 8 на различных устройствах: от смартфонов и планшетов до настольных ПК. Но данная идея подвергалась критике все время использования Windows 8. Попытка же заставить пользователей настольных ПК с Windows и на мобильных устройствах перейти на данную ОС не увенчалась успехом. Слухи о новой ОС кодовым названием «Windows Blue» появились сразу же после появления Windows 8. 2 апреля 2014 года Windows 8.1 Update была представлена официально. Все пользователи лицензионных копий Windows 8 бесплатно получили возможность обновиться до Windows 8.1. Версия ядра NT новой ОС — 6.3 (в Windows 8 используется ядро NT 6.2). В Windows 8.1 произошло улучшение интерфейса Metro, так же появилось улучшение в области энергопотребления. Основная поддержка Windows 8.1 Update прекращена 9 января 2018 года. Расширенная поддержка Windows 8.1 Update будет действовать до 10 января 2023 года. В декабре 2013 года технический обозреватель Мэри Джо Фоли сообщил, что Microsoft работает над обновлением для Windows 8 под кодовым именем «Threshold», основной целью проектирования которой было создание единой платформы приложений и инструмента разработки для Windows, Windows Phone и Xbox One, которые используют единое ядро — Windows NT. Windows «Threshold» была официально представлена во время мероприятия 30 сентября 2014 года под названием Windows 10. В компании Microsoft было заявлено, что Windows 10 станет самой «всеобъемлющей платформой» Microsoft, обеспечивающей единую унифицированную платформу для ПК, ноутбуков, планшетов, смартфонов и устройств «все-в-одном». Кроме этого, с учетом критики сенсорного интерфейса Windows 8 от пользователей, использовавших клавиатуры и мыши, в Windows 10 были предприняты шаги по восстановлению механизма пользовательского интерфейса из Windows 7. Windows 10 — операционная система для персональных компьютеров и рабочих станций, разработанная корпорацией Microsoft в рамках семейства Windows NT (ядро NT 6.3). После Windows 8.1 система получила номер 10, минуя 9. Серверный аналог Windows 10 — Windows Server 2016. Официально Windows 10 была выпущена 29 июля 2015 года. С апреля 2017 г. Windows 10 стала занимать первое место в мире среди операционных систем, используемых для доступа к сети Интернет, опередив предыдущего лидера — Windows 7. Будет ли эта операционная система настолько популярна, как ее знаменитые предшественники — ОС Windows XP и Windows 7 покажет время.

В заключение анализа ОС для настольных ПК и рабочих станций от Microsoft хочется отметить, что в настоящее время огромное количество компаний, занимающихся разработкой программных приложений, «уставших» от постоянной гонки Microsoft по пути создания новых операционных систем, переключается на ОС семейства Linux, интерфейс которых стал значительно проще и более «дружественным» к пользователю.

§ 1.2. Основы программирования в ОС Windows

Рассмотрев в предыдущем параграфе многообразие операционных систем для персональных компьютеров, можно сделать вывод о том, что программирование в конкретной операционной системе — это процесс настолько сложный, основанный на специфических особенностях данной

ОС, что переход программиста на новую операционную систему практически соизмерим с переходом на новый язык программирования. Поэтому в настоящее время каждый программист является узким специалистом в области программирования в конкретной ОС, и даже более — в конкретном узком направлении разработки для данной ОС. При этом общие принципы построения современных ОС существенно не отличаются друг от друга. Данное учебное пособие посвящено особенностям программирования базовых элементов операционных систем, и поскольку в пределах курса невозможно качественно рассмотреть особенности технологий разработки системных элементов для многих ОС, сосредоточимся в практическом плане на операционных системах, получивших в настоящее время наибольшее распространение: ОС, разработанных компанией Microsoft:Windows линеек 9x и NT (2000, XP). Таким образом, в пособии первоначально будут рассматриваться особенности построения базовых элементов основных современных операционных систем, а далее в более практическом плане — особенности разработки этих технологий в ОС Windows. В качестве базового языка программирования выбран язык C++ и инструментальная система Visual C++ по следующим причинам [4]:

– ранее при изучении основ программирования в качестве базового был выбран именно этот язык и инструментальная система; таким образом, время, выделенное на изучение курса, максимально будет потрачено на изучение основ программирования базовых элементов ОС, а не на изучение нового языка;

– операционные системы семейства Windows сами были созданы на языке программирования C/C++, поэтому программирование их элементов на данном языке в наименьшей степени вызовет проблемы совместимости этих элементов;

– бóльшая часть документации, публикаций и других материалов о программировании для Windows ориентирована на использование C++, и фактически вся документация Microsoft рассчитана на программистов, знакомых именно с этим языком программирования, который в настоящее время "de facto" стал системным для многих ОС, в том числе для Windows;

– в настоящее время большинство системных программистов во всем мире работают именно на C/C++.

Вместе с тем в рабочей среде Windows используются многие другие популярные среды разработки, в частности получившая в последнее время широкое распространение среда программирования Java (в первую очередь, при программировании для Internet). Компания Borland поддерживает собственную среду Delphi, которая основана на языке Pascal (существует также C++ Builder, которая использует похожий подход к созданию приложений, но ориентирована на использование C++). Не стоит забывать и про Visual Basic (VB). Начиная с пятой версии, этот продукт поддерживает полноценную компиляцию, технологию ActiveX, а также прямое обращение к API.

В курсе лекций, посвященных программированию, мы уже научились создавать различные приложения в среде Windows,в частности приложения на базе однооконного и многооконного интерфейса, обрабатывать сообщения клавиатуры и мыши, создавать различные ресурсы, предоставляемые ОС, использовать цветовые палитры, создавать подключаемые библиотеки и т. д. Все это поможет рассмотреть программирование базовых системных элементов ОС в более качественной форме. Первоначально рассмотрим кратко основы программирования в ОС семейства Windows, общие черты и различия Windows линеек 9x и NT.

1.2.1. Принципы взаимодействия ОС Windows с прикладными программами

Благодаря интерфейсу вызова функций в Windows доступ к системным ресурсам осуществляется через целый ряд системных функций. Совокупность таких функций называется прикладным программным интерфейсом, или API (Application Programming Interfase). Для взаимодействия с Windows приложение запрашивает функции API, с помощью которых реализуются все необходимые системные действия, такие как выделение памяти, вывод на экран, создание окон и т. п.

Поскольку API состоит из большого числа функций, может сложиться впечатление, что при компиляции каждой программы, написанной для Windows, к ней подключается код довольно значительного объема. В действительности это не так. Функции API содержатся в ***библиотеках***

динамической загрузки (Dynamic Link Libraries, или DLL), которые загружаются в память только в тот момент, когда к ним происходит обращение, т. е. при выполнении программы. Рассмотрим, как осуществляется механизм динамической загрузки.

Динамическая загрузка обеспечивает ряд существенных преимуществ. Во-первых, поскольку практически все программы используют API-функции, то благодаря DLL-библиотекам существенно экономится дисковое пространство, которое в противном случае занималось бы большим количеством повторяющегося кода, содержащегося в каждом из исполняемых файлов. Во-вторых, изменения и улучшения в Windows-приложениях сводятся к обновлению только содержимого DLL-библиотек. Уже существующие тексты программ не требуют перекомпиляции.

В настоящее время наибольшее распространение получила версия API, которую назвали Win32. Данная версия API пришла на смену версии Win16, используемой вWindows 3.1. Фактически 32-разрядная Win32, используемая в операционных системах 9x, является надмножеством для Win16 (т. е. фактически включает в себя этот интерфейс), так как большинство функций имеет то же название и применяется аналогичным образом. Однако будучи в принципе похожими, оба интерфейса все же отличаются друг от друга: Win32 поддерживает 32-разрядную линейную адресацию, тогда как Win16 работает только с 16-разрядной сегментированной моделью памяти. Это привело к тому, что некоторые функции были модифицированы таким образом, чтобы принимать 32-разрядные аргументы и возвращать 32-разрядные значения. Часть из них пришлось изменить с учетом 32-разрядной архитектуры. Была реализована поддержка потоковой многозадачности, новых элементов интерфейса и прочих нововведений Windows.

Так как Win32 поддерживает полностью 32-разрядную адресацию, то логично, что целые типы данных (*intergers*) также объявлены 32-разрядными. Это означает, что переменные типа *int* и *unsigned*будут иметь длину 32 бита, а не 16, как в Windows 3.1. Если же необходимо использовать переменную или константу длиной 16 бит, то они должны быть объявлены как *short*(далее будет показано, что для этих типов определены независимые *typedef*-имена.). Следовательно, при переносе программного кода из 16-разрядной среды необходимо убедиться в правильности использования целочисленных элементов, которые автоматически будут расширены до 32 битов, что может привести к появлению побочных эффектов. Другим следствием 32-разрядной адресации является то, что указатели больше не нужно объявлять как *near* и *far*. Любой указатель может получить доступ к любому участку памяти. В Windows 9x константы *near* и *far*объявлены (с помощью директивы *#define*) пустыми.

Существенные изменения коснулись также функций API, которые работают с символьными строками. Внутренние механизмы Windows 9x (не полностью), NT и Windows 2000 используют символы в формате UNICODE (16-битное значение символов). Но многие прикладные программы продолжают использовать ANSI, т. е. стандарт, в котором каждый символ кодируется при помощи 8 бит. Чтобы решить эту проблему, каждая функция, работающая со строками, реализована в двух вариантах. В первом варианте в качестве аргумента воспринимается строка в формате ANSI (внутри функции эта строка автоматически преобразуется в UNICODE). Второй вариант функции напрямую работает со строками в формате UNICODE.

Одним из подмножеств API является GDI (Graphics Device Interfase — интерфейс графического устройства). GDI — это та часть Windows, которая обеспечивает поддержку аппаратно-независимой графики. Благодаря функциям GDI Windows-приложение может выполняться на различных ПЭВМ.

Еще одной особенностьюWindows является многозадачность, причем поддерживаются два типа многозадачности: основанная на процессах и основанная на потоках. Далее идеология многозадачности будет рассмотрена более подробно.

Во многих операционных системах взаимодействие между системой и программой инициализирует программа. Например, в DOS программа запрашивает разрешение на ввод и вывод данных. Другими словами, не-Windows-программы сами вызывают операционную систему. Обратного процесса не происходит. В Windows все совершенно наоборот: именно система вызывает программу. Это осуществляется следующим образом: программа ожидает получения сообщения от Windows. Когда это происходит, то выполняется некоторое действие. После его завершения программа ожидает следующего сообщения.

Windows может посылать программе сообщения множества различных типов. Например, каждый раз при щелчке мышью в окне активной программы посылается соответствующее сообщение. Другой тип сообщений посылается, когда необходимо обновить содержимое активного окна. Сообщения посылаются также при нажатии клавиши, если программа ожидает ввода с клавиатуры. Необходимо запомнить одно: по отношению к программе сообщения появляются случайным образом. Вот почему Windows-программы похожи на программы обработки прерываний: невозможно предсказать, какое сообщение появится в следующий момент.

Функция окна. Все Windows-программы должны содержать специальную функцию, которая не используется в самой программе, но вызывается операционной системой. Эту функцию обычно называют ***функцией окна***, или ***процедурой окна***. Она вызывается Windows, когда системе необходимо передать сообщение в программу. Именно через нее осуществляется взаимодействие между программой и системой. Функция окна передает сообщение в своих аргументах. Согласно терминологии Windows те функции, которые вызываются системой, называются ***функциями обратного вызова***. Таким образом, функция окна является функцией обратного вызова. Помимо принятия сообщения от Windows функция окна должна вызывать выполнение действия, указанного в сообщении. Конечно, программа не обязана отвечать на все сообщения, посылаемые Windows. Поскольку их могут быть сотни, то большинство сообщений обычно обрабатывается самой системой, а программе достаточно поручить Windows выполнить действия, предусмотренные по умолчанию.

Цикл сообщений. Как объяснялось выше, Windows взаимодействует с программой, посылая ей сообщения. Все приложения Windows должны организовать так называемый цикл сообщений. В этом цикле каждое необработанное сообщение должно быть извлечено из очереди сообщений данного приложения и передано назад в Windows, которая затем вызывает функцию окна программы с данным сообщением в качестве аргумента.

Класс окна. Как будет показано далее, каждое окно в Windows-приложении характеризуется определенными атрибутами, называемыми классом окна (здесь понятие «класс» не идентично используемому в C++. Оно скорее означает стиль или тип.) В традиционной программе класс окна должен быть определен и зарегистрирован прежде, чем будет создано окно. При регистрации необходимо сообщить Windows, какой вид должно иметь окно и какую функцию оно выполняет. В то же время регистрация класса окна еще не означает создания самого окна. Для этого требуется выполнить дополнительные действия.

Структура Windows-программ отличается от структуры программ других типов. Это вызвано двумя обстоятельствами: во-первых, способом взаимодействия между программой и Windows, описанным ранее; во-вторых, правилами, которым следует подчиняться для создания стандартного интерфейса Windows-приложения (т. е. чтобы сделать программу «похожей» на Windows-приложение).

Цель Windows — дать человеку, который хотя бы немного знаком с системой, возможность сесть за компьютер и запустить любое приложение без предварительной подготовки. Для этого Windows предоставляет дружественный интерфейс пользователя, который необходимо поддерживать всем программистам, создающим программное обеспечение в данной операционной системе.

1.2.2. Типы данных в Windows

В Windows-программах не слишком широко применяются стандартные типы данных из C или C++, такие как *int* или *char**. Вместо них используются типы данных, определенные в различных библиотечных (*header*) файлах. Наиболее часто используемыми типами являются HANDLE, HWND, BYTE, WORD, DWORD, UNIT, LONG, BOOL, LPSTR и LPCSTR [11].

Тип HANDLE обозначает 32-разрядное целое, используемое в качестве дескриптора. Есть несколько похожих типов данных, но все они имеют ту же длину, что и HANDLE, и начинаются с литеры *H*. ***Дескриптор*** — это просто число, определяющее некоторый ресурс.

Тип HWND обозначает 32-разрядное целое — дескриптор окна. В программах, использующих библиотеку MFC, дескрипторы применяются не столь широко, как это имеет место в традиционных программах.

Тип BYTE обозначает 8-разрядное беззнаковое символьное значение.

Тип WORD — 16-разрядное беззнаковое короткое целое.

Тип DWORD — беззнаковое длинное целое.

Тип UINT — беззнаковое 32-разрядное целое.

Тип LONG эквивалентен типу *long*.

Тип BOOL обозначает целое и используется, когда значение может быть либо истинным, либо ложным.

Тип LPSTR определяет указатель на строку.

Тип LPCSTR определяет константный (*const*) указатель на строку.

1.2.3. Графический и консольный интерфейсы

В Windows поддерживаются два типа приложений: основанные на графическом интерфейсе GUI (*graphical user interface*) и консольные CUI (*console user interface*). У приложений первого типа внешний интерфейс чисто графический: создаются окна, меню, диалоговые окна и т. д. Почти все стандартные программы Windows и большинство пользовательских программ являются GUI-приложениями. Приложения консольного типа работают в текстовом режиме: они не формируют окна, не обрабатывают сообщения, но на экране тоже размещаются в окне (правда, черного цвета) и могут вызывать диалоговые окна.

Все Windows-программы, написанные на языке C++, начинают выполнение с вызова функции входа. В среде Win32 существует четыре модификации таких функций:

```
int WINAPI WinMain(HINSTANCE hInstance, // дескриптор, присваиваемый запущенному приложению
        HINSTANCE hPrevInstance,   // для совместимости с win16, в win32 не используется
        LPSTR lpCmdLine,   // указатель на командную строку, если приложение так запущено
        int nCmdShow);   // значение, которое может быть передано в функцию ShowWindow()

int WINAPI wWinMain(HINSTANCE hInstance,
        HINSTANCE hPrevInstance,
        LPSTR lpCmdLine,
        int nCmdShow);

int __cdecl main (int argc,      // количество аргументов, переданных в командной строке
        char *argv[],       // массив размером argc с указателями на ANSI-строки.
                            // Каждый элемент массива указывает на один из аргументов командной
                              строки
        char *envp[]);      // массив указателей на ANSI-строки. Каждый элемент массива
                            // указывает на строку-переменную окружения (область памяти,
                            // выделенная в адресном пространстве процесса)

int __cdecl wmain(int argc,      // количество аргументов, переданных в командной строке
        wchar_t *argv[],    // массив размером argc с указателями на UNICODE-строки.
                            // Каждый элемент массива указывает на один из аргументов командной
                              строки
        wchar_t *envp[]);   // массив указателей на UNICODE-строки. Каждый элемент массива
                            // указывает на строку-переменную окружения (область памяти, выделенная
                            // в адресном пространстве процесса)
```

Первые две функции используются при разработке приложения с графическим интерфейсом (GUI), две другие — при консольном (CUI). Приставка *w* перед названием функции показывает, что функция используется для работы с Unicode-строками, обычные функции работают с ANSI-строками. На самом деле входная функция не вызывается операционной системой, вместо этого происходит обращение к стартовой функции из библиотеки C/C++. Она инициализирует библиотеку C/C++, а также обеспечивает корректное создание любых объявленных глобальных и статических объектов до того, как начнется выполнение программного кода. В табл. 1.1 показано, в каких случаях реализуются различные функции входа [7].

Таблица 1.1

Тип приложения	Функция входа	Стартовая функция, встраиваемая в исполняемый файл
GUI-приложение, работающее с ANSI-символами и строками	*WinMain()*	*WinMainCRTStartup()*
GUI-приложение, работающее с Unicode-символами и строками	*wWinMain()*	*wWinMainCRTStartup()*
CUI-приложение, работающее с ANSI-символами и строками	*Main()*	*mainCRTStartup()*
CUI-приложение, работающее с Unicode-символами и строками	*Wmain()*	*wmainCRTStartup()*

В целом все стартовые функции выполняют следующие однотипные задачи:

– считывают указатель на полную командную строку нового процесса;

– считывают указатель на переменные окружения нового процесса;

– инициализируют глобальные переменные из библиотеки C/C++;

– инициализируют кучу (динамически распределяемую область памяти), используемую С-функциями выделения памяти (*malloc, calloc)* и другими процедурами низкоуровневого ввода / вывода;

– вызывают конструкторы всех глобальных и статических объектов классов C++.

Закончив эти операции, стартовая функция обращается к функции входа.

1.2.4. Создание элементарного графического окна

Рассмотрим, как создать элементарное графическое окно в Windows. Как уже указывалось, главной функцией для разработки такой программы будет WinMain(). Сразу после входа в WinMain() создается и регистрируется класс главного окна приложения. Для этого необходимо заполнить структуру WNDCLASS. Для создания окна используются функции CreateWindow() непосредственно для его создания, ShowWindow() для визуализации и UpdateWindow() для перерисовки. Все эти функции описаны в файле *windows.h*. Рассмотрим структуру WNDCLASS и наиболее значимую функцию CreateWindow() более подробно.

Структура WNDCLASS состоит из 10 полей:

```
typedef struct tagWNDCLASS
{
   UINT cbSize;              // размер структуры в байтах
   UINT style;               // стиль класса окна
   WNDPROC lpfnWndProc;      // указатель на функцию окна
   HINSTANCE hInstance;      // дескриптор приложения, которое запускает окно
   HICON hIcon;              // дескриптор пиктограммы
   HCURSOR hCursor;          // дескриптор курсора
   HBRUSH hbrBackground;     // дескриптор кисти для закраски фона
   LPCTSTR lpszMenuName;     // указатель на строку с именем меню
   LPCTSTR lpszClassName;    // указатель на строку с именем класса
   HICON hIconSm;            // дескриптор малой пиктограммы
} WNDCLASS;
```

Функция CreateWindow() получает на входе 11 параметров:

```
HWND CreateWindow (
   LPCTSTR lpClassName,      // зарегистрированное имя класса
   LPCTSTR lpWindowName,     // имя окна
   DWORD dwStyle,            // стиль окна
   int x,                    // x — координата позиции окна
   int y,                    // y — координата позиции окна
   int nWidth,               // ширина окна
   int nHeight,              // высота окна
   HWND hWndParent ,         // указатель родительского окна
```

```
    HMENU hMenu,                    // дескриптор меню. NULL. если меню нет
    HINSTANCE hInstance,            // дескриптор приложения
    LPVOID lParam );                // данные. которые могут быть переданы для создания окна
                                    // NULL. если данные не передаются
```

Для примера рассмотрим исходный текст программы, которая создает элементарное окно в Windows:

```
#include <windows.h>                // функция для управления параметрами создания и уничтожения
                                    главного окна
#include <windowsx.h>               // запуск всех дополнительных элементов окна (если они есть)
                                    производится в ней
LRESULT CALLBACK WndProc(HWND. UINT. WPARAM, LPARAM);       // главная функция
int WINAPI WinMain(HINSTANCE hInstance,
    HINSTANCE hPrevInstance,
    LPSTR lpCmdLine,
    int nCmdShow)
{
                                    // описываем параметры. которые необходимы для создания окна
    HWND hwnd;                                  // указатель класса окна
    MSG msg;                                    // Структура MSG содержит информацию
                                                //о сообщениях из очереди потока сообщений.
    WNDCLASS w;                                 // структура, определяющая класс окна
    memset(&w, 0. sizeof(WNDCLASS));            // выделение памяти для класса окна
    w.style = 0;                                // стиль окна по умолчанию
    w.lpfnWndProc = WndProc;                    // функция окна
    w.hInstance = hInstance;                    // дескриптор приложения
    w.hbrBackground = GetStockBrush(WHITE_BRUSH);       // цвет для заполнения окна
    w.lpszClassName = "API Windows",            // имя класса окна
    RegisterClass(&w);                          // Регистрируем класс окна
                                                // Создаем окно
    hwnd = CreateWindow("API Windows",          // имя класса
        "API Windows",                          //название окна
        WS_OVERLAPPEDWINDOW,                    // стиль окна
        10,                                 // x—координата
        10,                                 // y— координата
        600,                                    // ширина
        480,                                    // высота
        NULL,                                   // нет родительского окна
        NULL,                                   // нет меню
        hInstance,                              // дескриптор приложения
        NULL);                                  // не передаем данных
    ShowWindow (hwnd, nCmdShow);                // активация окна
    UpdateWindow (hwnd);                        // перерисовка
                                                // Цикл обработки сообщений
    while(GetMessage(&msg, NULL, 0, 0))
    {
    TranslateMessage(&msg);                     // разрешить использование клавиатуры
    DispatchMessage(&msg);                      // вернуть управление Windows
    }
                                                //возвращаемое значение для функции WndProc()
    return msg.wParam;
}

LRESULT CALLBACK WndProc(HWND hwnd, UINT Message,
        WPARAM wparam, LPARAM lparam)
{
    if (Message = = WM_DESTROY)
    {
    PostQuitMessage(0);
```

```
        return 0;
        }
        return DefWindowProc(hwnd, Message. wparam. lparam);
}
```

Компиляция и компоновка проекта приводят к созданию *exe*-файла, а его запуск, в свою очередь, к появлению окна, представленного на рис.1.1.

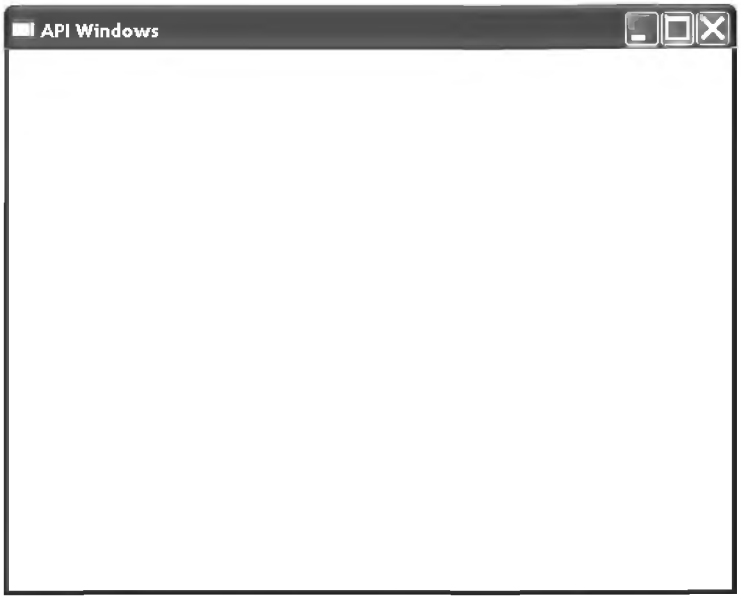

Рис.1.1. Результат выполнения программы

§ 1.3. Принципы разработки динамических библиотек

1.3.1. Основные положения

Динамически подключаемые библиотеки (DLL — *dynamic link libraries*) представляют собой важное средство операционной системы ОС Windows. DLL-библиотеки позволяют подключать предварительно созданные модули к операционной системе, что обеспечивает для приложений общий код, создает модульное представление часто используемых функций и способствует расширяемости. ОС Windows сама состоит в основном из DLL-библиотек. Любое приложение, где применяется Win32API, использует функциональные возможности DLL. Использование динамических библиотек — это способ осуществления модульности в период выполнения программы. DLL позволяет упростить и саму разработку программного обеспечения. Вместо того чтобы каждый раз перекомпилировать огромные EXE-программы, достаточно перекомпилировать лишь отдельный динамический модуль. Кроме того, доступ к динамической библиотеке возможен сразу из нескольких исполняемых модулей, что делает многозначность более гибкой. Структура DLL-модуля практически такая же, как и EXE-модуля. Динамическая библиотека —это еще и возможность разработки приложений на разных языках. Динамическая библиотека, написанная, скажем, на Visual C++, может вызываться из программ, написанных на любых языках. Следует также отметить, что использование динамических библиотек экономит дисковое пространство, так как процедура из такой библиотеки помещается в модуль лишь один раз в отличие от процедур, помещаемых в модули из статических библиотек.

Динамическая библиотека проецируется на адресное пространство вызывающего процесса. В результате она становится частью этого процесса. Если динамическая библиотека резервирует

динамическую память, то этот блок памяти принадлежит процессу. При удалении библиотеки из памяти блок остается и удаляется только процессом.

При создании DLL необходимо указывать экспортируемые функции. Это делается следующим образом [7]:

```
_declspec(dllexport) int Func(char * s);
```

В программе, вызывающей эту функцию, следует ввести следующую строку:

```
_declspec(dllimport) int Func(char * s);
```

Для дальнейшего изучения DLL введем понятие *«связывание»*. Во время трансляции связываются имена, указанные в программе как внешние, с соответствующими именами из библиотек. Такое связывание называется ранним (или статическим). Напротив, в случае с динамической библиотекой связывание происходит во время выполнения модуля. Такое связывание называется поздним (или динамическим). При этом позднее связывание может происходить в автоматическом режиме в начале запуска программы и при помощи специальных API-функций (см. далее) по желанию программиста. При этом говорят о явном и неявном связывании (рис. 1.2).

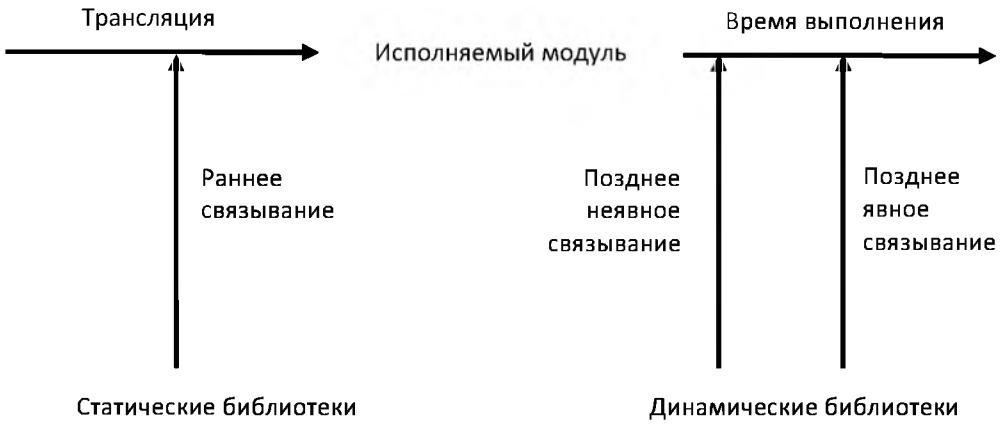

Рис. 1.2. Иллюстрация механизма связывания

В среде Windows практикуются два механизма связывания: по символьным и порядковым номерам. В первом случае функция, определенная в динамической библиотеке, идентифицируется по ее имени, во втором — по порядковому номеру.

Динамическая библиотека может содержать ресурсы. Так, файлы шрифтов представляют собой динамические библиотеки, единственным содержимым которых являются ресурсы. Надо сказать, что динамическая библиотека как бы становится продолжением вашей программы, загружаясь в адресное пространство процесса. Соответственно данные процесса доступны из динамической библиотеки и наоборот — данные динамической библиотеки доступны для процесса.

В любой динамической библиотеке следует определить точку ввода (процедура ввода). При загрузке и выгрузке динамической библиотеки автоматически вызывается процедура ввода. Следует отметить, что каким бы способом ни была загружена динамическая библиотека (явно или неявно), выгрузка динамической библиотеки из памяти будет происходить автоматически при закрытии процесса или потока. В принципе процедура входа может быть использована для некоторой начальной инициализации динамических переменных. Довольно часто эта процедура остается пустой. Для реализации уведомляющей точки входа в DLL потребуется поместить код функции, основанный на прототипе DLLEntryPoint():

```
BOOL DLLEntryPoint(HINSTANCE hInstDLL, DWORD dwNotification,
    LPVOID lpReserved)
```

Имя функции DLLEntryPoint() служит просто заполнителем. По умолчанию модуль подключения ищет функцию с именем **DLLMain()**. При определении точки входа в DLL функции можно присвоить любое имя. При вызове процедуры входа в нее помещаются три параметра:

- идентификатор DLL-модуля;
- причина вызова;
- резерв.

Рассмотрим подробнее второй параметр процедуры ввода, который может принимать четыре возможных значения:

• DLL_PROCESS_ATTACH — сообщает, что динамическая библиотека загружена в адресное пространство вызывающего процесса;

• DLL_THREAD_ATTACH — сообщает, что текущий процесс создает новый поток. Такое сообщение посылается всем динамическим библиотекам, загруженным к этому времени процессом;

• DLL_PROCESS_DETACH — сообщает, что динамическая библиотека выгружается из адресного пространства процесса;

• DLL_THREAD_DETACH —сообщает, что некий поток, созданный данным процессом, в адресное пространство которого загружена данная динамическая библиотека, уничтожается.

1.3.2. Главная функция DllMain()

Большинство библиотек DLL — просто коллекции практически независимых друг от друга функций, экспортируемых в приложения и используемых в них. Кроме функций, предназначенных для экспортирования, в каждой библиотеке DLL должна быть функция DllMain(). Эта функция предназначена для инициализации и очистки DLL. Она пришла на смену функциям LibMain и WEP, применявшимся в предыдущих версиях Windows. Структура простейшей функции DllMain() может выглядеть следующим образом:

```
BOOL WINAPI DllMain (HINSTANCE hInstDll, DWORD dwReason, LPVOID lpReserved)
{
BOOL bAllWentWell=TRUE ;
switch (dwReason)
{
    case DLL_PROCESS_ATTACH:          // Инициализация процесса.
        break;
    case DLL_ THREAD_ATTACH:          // Инициализация потока.
        break;
    case DLL_THREAD_DETACH:           // Очистка структур потока.
        break ;
    case DLL_PROCESS_DETACH:          // Очистка структур процесса.
        break;
}
if(bAllWentWell)
   return TRUE ;
else
   return FALSE;
}
```

Функция DllMain() вызывается в нескольких случаях. Причина ее вызова определяется параметром *dwReason*, который может принимать одно из следующих значений. При первой загрузке библиотеки DLL процессом вызывается функция DllMain() с параметром *dwReason*, равным DLL_PROCESS_ATTACH. Каждый раз при создании процессом нового потока (кроме первичного) DllMain() вызывается с параметром *dwReason*, равным DLL_THREAD_ATTACH.

По окончании работы процесса с DLL функция DllMain() вызывается с параметром *dwReason*, равным DLL_PROCESS_DETACH. При уничтожении потоков (кроме первичного) *dwReason* будет равен DLL_THREAD_DETACH.

Все операции по инициализации и очистке для процессов и потоков, в которых нуждается DLL, необходимо выполнять на основании значения *dwReason*, как было показано в предыдущем примере. Инициализация процессов обычно ограничивается выделением ресурсов, совместно используемых потоками, в частности загрузкой разделяемых файлов и инициализацией библиотек. Инициализация потоков применяется для настройки режимов, свойственных только данному потоку, например для инициализации локальной памяти.

В состав DLL могут входить ресурсы, не принадлежащие вызывающему эту библиотеку приложению. Если функции DLL работают с ресурсами DLL, было бы, очевидно, полезно сохранить дескриптор hInst и использовать его при загрузке ресурсов из DLL. Указатель *lpReserved* зарезервирован для внутреннего использования Windows. Следовательно, приложение не должно претендовать на него, можно лишь проверить его значение. Если библиотека DLL была загружена динамически, оно будет равно NULL. При статической загрузке этот указатель будет ненулевым. В случае успешного завершения функция DllMain() должна возвращать TRUE. В случае возникновения ошибки возвращается FALSE и дальнейшие действия прекращаются.

1.3.3. Экспортирование функций из DLL

Чтобы приложение могло обращаться к функциям динамической библиотеки, каждая из них должна занимать строку в таблице экспортируемых функций DLL. Есть два способа занести функцию в эту таблицу на этапе компиляции. Можно экспортировать функцию из DLL, поставив в начале ее описания модификатор _declspec (*dllexport*). В языке C++ по умолчанию производится добавка к именам функций таким образом, чтобы различать функции с разным количеством и типом параметров. Поэтому для того, чтобы можно было вызывать функцию из другого модуля по ее обычному имени, ее следует объявить как extern "C".

Структура простейшей экспортируемой функции в этом случае может выглядеть следующим образом:

```
extern "C" __declspec (dllexport) int MyFunction(...)
{
  ...
  return 0;
}
```

Кроме этого, для экспорта функций можно использовать файлы определения модуля **.def*. Синтаксис файлов с расширением *.def* в достаточно прямолинеен, главным образом потому, что сложные параметры, использовавшиеся в ранних версиях Windows, в Win32 более не применяются. Как станет понятно из следующего простого примера, *.def*-файл содержит имя и описание библиотеки, а также список экспортируемых функций:

```
MyDLL.def
LIBRARY       "MyDLL"
DESCRIPTION   'MyDLL — пример DLL-библиотеки'
EXPORTS       MyFunction    @1
```

В строке экспорта функции можно указать ее порядковый номер, поставив перед ним символ @. Этот номер будет затем использоваться при обращении к функции GetProcAddress(), которая будет описана далее. На самом деле компилятор присваивает порядковые номера всем экспортируемым объектам. Однако способ, которым он это делает, отчасти непредсказуем, если не присвоить эти номера явно. В строке экспорта можно использовать параметр NONAME. Он запрещает компилятору включать имя функции в таблицу экспортирования DLL:

```
MyFunction    @1 NONAME
```

Иногда это позволяет сэкономить много места в файле DLL. Приложения, использующие библиотеку импортирования для неявного подключения DLL, не «заметят» разницы, поскольку при неявном подключении порядковые номера используются автоматически. Приложениям, загружающим библиотеки DLL динамически, потребуется передавать в GetProcAddress() порядковый номер, а не имя функции.

В отличие от статических библиотек, которые, по существу, становятся частью кода приложения, динамические библиотеки в 16-разрядных версиях Windows работали с памятью несколько иначе. Под управлением Win16API память DLL размещалась вне адресного пространства задачи. Размещение динамических библиотек в глобальной памяти обеспечивало возможность совместного использования их различными задачами.

В Win32API библиотека DLL располагается в области памяти загружающего ее процесса. Каждому процессу предоставляется отдельная копия «глобальной» памяти DLL, которая реинициализируется каждый раз, когда ее загружает новый процесс. Это означает, что динамическая библиотека не может использоваться совместно, в общей памяти, как это было в Win16.

И все же, выполнив ряд замысловатых манипуляций над сегментом данных DLL, можно создать общую область памяти для всех процессов, использующих данную библиотеку. Допустим, имеется массив целых чисел, который должен использоваться всеми процессами, загружающими данную DLL. Это можно запрограммировать следующим образом:

```
#pragma data_seg(".myseg")
int shared Ints[11];
// другие переменные общего пользования
#pragma data_seg()
#pragma comment (lib, "msvcrt" "–SECTION:.myseg, rws");
```

Все переменные, объявленные между директивами *#pragma data_seg*(), размещаются в сегменте *.myseg*. Директива *#pragma comment*() — не обычный комментарий. Она дает указание библиотеке выполняющей системы пометить новый раздел как разрешенный для чтения, записи и совместного доступа.

Для демонстрации принципа разработки DLL создадим простую динамическую библиотеку, содержащую только простую функцию DllMain() и одну внешнюю функцию, которая получает строку символов и выводит ее в виде окна с сообщением:

```
#include <windows.h>
BOOL WINAPI DllMain (HINSTANCE hInctance, DWORD fdReason, PVOID pvReserved)
{
   return TRUE;
}
extern "C" __declspec (dllexport) int MyFunction(const char *str)
{
   MessageBox (NULL, str, "Function from DLL", MB_OK);
   return 1;
}
```

Компиляция данного проекта приведет к созданию двух необходимых в дальнейшем файлов — с расширениями *.dll* и *.lib*.

1.3.4. Подключение DLL

Практически невозможно создать приложение Windows, в котором не использовались бы библиотеки DLL. В DLL содержатся все функции Win32API и несчетное количество других функций операционных систем Win32. Вообще говоря, DLL — это просто наборы функций, собранные в библиотеки. Однако в отличие от своих статических родственников (файлов *.lib*) динамические библиотеки не присоединены непосредственно к выполняемым файлам с помощью редактора связей. В выполняемый файл занесена только информация об их местонахождении. В момент выполнения программы загружается вся библиотека целиком. Благодаря этому разные процессы могут пользоваться совместно одними и теми же библиотеками, находящимися в памяти. Такой подход позволяет сократить объем памяти, необходимый для нескольких приложений, использующих много общих библиотек, а также контролировать размеры EXE-файлов.

Однако если библиотека используется только одним приложением, то лучше сделать ее обычной, статической. Конечно, если входящие в ее состав функции будут использоваться только в одной программе, можно просто вставить в нее соответствующий файл с исходным текстом.

В случае использования динамических библиотек, как было сказано ранее, приложение может активизировать размещенные в DLL процедуры двумя методами: поздними неявным и явным связываниями. Рассмотрим их более подробно.

1. ***Позднее неявное связывание***. В данном случае приложение реализует динамическое связывание во время загрузки за счет указания имен процедур из DLL непосредственно в исходном коде. Компоновщик вставляет ссылки на эти процедуры при их обнаружении в связанной с приложением библиотеке импорта либо через раздел IMPORTS файла определений модуля для данного приложения. Во время выполнения приложения загрузчик Windows помещает DLL-библиотеки в память и разрешает эти ссылки. При запуске приложение пытается найти все файлы DLL, неявно подключенные к приложению, и поместить их в область оперативной памяти, занимаемую данным процессом. Поиск файлов DLL операционной системой осуществляется в следующей последовательности:

- каталог, в котором находится EXE-файл;
- текущий каталог процесса;
- системный каталог Windows.

Если библиотека DLL не обнаружена, приложение выводит диалоговое окно с сообщением о ее отсутствии и путях, по которым осуществлялся поиск. Затем процесс отключается. Если нужная библиотека найдена, она помещается в оперативную память процесса, где и остается до его окончания. Теперь приложение может обращаться к функциям, содержащимся в DLL. Для импорта функций библиотеки необходимо использовать строку *__declspec (dllimport)*.

Эта форма динамического связывания наиболее проста, однако при некоторых условиях она может создавать проблемы. Например, если приложение ссылается на процедуру из DLL таким способом, в случае его реализации DLL-библиотека должна существовать даже тогда, когда приложение никогда не обращается к процедуре. Кроме того, на этапе компиляции приложение должно знать имена всех процедур, которые будут связываться во время загрузки. Для этого используется специальный файл с расширением *.lib*. Однако .lib-файл, используемый при неявном связывании DLL, — это не обычная статическая библиотека. Такие *.lib*-файлы называются библиотеками импортирования (*import libraries*). В них содержится не сам код библиотеки, а только ссылки на все функции, экспортируемые из файла DLL, в котором все и хранится. В результате библиотеки импортирования, как правило, имеют меньший размер, чем DLL-файлы. Создаются такие файлы обычно средой разработки одновременно с *.dll*. Имя *.lib*-файла определяется среди прочих параметров редактора связей в командной строке или на вкладке "Link" диалогового окна "Project Settings" среды Developer Studio. В библиотеках языка С имена функций не расширяются, поэтому если необходимо подключить библиотеку на С к приложению на С++, все функции из этой библиотеки придется объявить как внешние в формате С:

```
extern "C" __declspec (dllimport) int MyOldCFunction(int myParam);
```

Рассмотрим простой пример подключения библиотеки с поздним неявным связыванием. В качестве Dll используем приведенную выше. Исходный код приложения следующий:

```
#include <windows.h>
extern "C" __declspec (dllimport) int MyFunction (const char *str);
int WINAPI WinMain (HINSTANCE hInstance, HINSTANCE hPrevInstance, LPSTR lpCmdLine, int nCmdShow)
{
    int iCode=MyFunction ("Hello!");
    return 0;
}
```

2. ***Позднее явное связывание***. Хотя данный тип связывания преодолевает ограничения во время загрузки, это метод требует большой работы над приложением. Вместо указания DLL-процедур на стадии компиляции приложение использует функции LoadLibrary(), LoadLibraryEx(), GetProcAddress() и FreeLibrary(), чтобы задать во время выполнения имена DLL-библиотек и процедур, на которые будут выполняться ссылки.

Кроме того, позднее явное связывание позволяет приложению поддерживать функциональные возможности, недопустимые на этапе создания приложения. Например, текстовый процессор может предоставлять внутри DLL подпрограммы преобразования файлов различного формата. При использовании динамического связывания во время выполнения возможно добавление новых DLL-библиотек, содержащих подпрограммы преобразования новых форматов, которых не было на момент создания приложения. Поскольку приложение во время выполнения определяет, какие DLL-библиотеки преобразования форматов существуют, в результате установки новых DLL приложение получит возможность использования новых процедур преобразования. Для извлечения имен существующих DLL приложение применяет функции FindFirstFile() и FindNextFile(). Затем оно может загрузить каждую DLL, получить адреса процедур преобразования и поместить эти адреса в структуру, которая впоследствии будет задействована в процессе преобразования файлов.

Первое, что необходимо сделать при динамической загрузке DLL, — это поместить модуль библиотеки в память процесса. Данная операция выполняется с помощью функции LoadLibrary(), имеющей единственный аргумент *lpFileName* — имя загружаемого модуля.

```
HMODULE LoadLibrary(LPCTSTR lpLibFileName);          // имя загружаемого модуля
```

Соответствующий фрагмент программы должен выглядеть следующим образом:

```
HINSTANCE hMyDll;
if ( (hMyDll=:: LoadLibrary ("MyDLL")) ==NULL) { / * не удалось загрузить DLL */ }
else { /* приложение имеет право пользоваться функциями DLL через hMyDll */ }
```

Стандартным расширением файла библиотеки Windows считает *.dll*, если не указать другое расширение. Если в имени файла указан и путь, то только он будет использоваться для поиска файла. В противном случае Windows будет искать файл по той же схеме, что и в случае неявно подключенных DLL, начиная с каталога, из которого загружается *exe*-файл, и продолжая в соответствии со значением PATH.

Когда Windows обнаружит файл, его полный путь будет сравнен с путем библиотек DLL, уже загруженных данным процессом. Если обнаружится тождество, вместо загрузки копии приложения возвращается дескриптор уже подключенной библиотеки.

Если файл обнаружен и библиотека успешно загрузилась, функция LoadLibrary() возвращает ее дескриптор, который используется для доступа к функциям библиотеки.

Расширенная функция LoadLibraryEx() получает три параметра:

```
HMODULE LoadLibraryEx(LPCTSTR lpLibFileName,      // имя загружаемого модуля
        HANDLE hFile,                             // зарезервирован, должен быть нулевым
        DWORD dwFlags);                           // флаги, определяющие опции загрузки модуля
```

Перед тем как использовать функции библиотеки, необходимо получить их адрес. Для этого сначала следует воспользоваться директивой *typedef* для определения типа указателя на функцию и определить переменную этого нового типа, например:

```
// тип PFH_MyFunction будет объявлять указатель на функцию,
// принимающую указатель на символьный буфер и выдающую значение типа int

typedef int (WINAPI *PFH_MyFunction) (char *);
...
PFH_MyFunction pfhMyFunction;
```

Затем следует получить дескриптор библиотеки, при помощи которого и определить адреса функций, например адрес функции с именем MyFunction:

```
hMyDll=::LoadLibrary("MyDLL");
pfhMyFunction=(PFH_MyFunction)::GetProcAddress(hMyDll, "MyFunction");
...
int iCode=(*pfhMyFunction) ("Hello");
```

В обращении к функции GetProcAddress() следует указывать имя библиотеки и имя функции. Допускается указание имени процедуры в виде ASCII-строки либо порядкового номера (в этом

случае для создания библиотеки должен использоваться def-файл, второй параметр GetProcAddress() приводится к виду MAKEINTRESOURCE(N)). Синтаксис функции GetProcAddress():

```
FAR PROC GetProcAddress(HMODULE hDLLibrary, LPCSTR lpszProcName);
```

При успешном завершении функция возвращает адрес точки входа запрашиваемой процедуры. В противном случае возвращается NULL. Параметры функции:

- hDLLibrary — дескриптор DLL, возвращаемый функцией LoadLibrary();
- lpszProcName — это либо указатель на завершаемую нулевым символом строку, идентифицирующую процедуру, либо ее порядковый номер.

После завершения работы с библиотекой динамической компоновки ее можно выгрузить из памяти процесса с помощью функции FreeLibrary():

```
::FreeLibrary(hMyDll);
```

Рассмотрим часть исходного кода программы подключения DLL с поздним явным связыванием. В качестве динамической библиотеки используем приведенную ранее (см. с. 32):

```
...
typedef int (WINAPI *PFH_MyFunction) (char *);
HINSTANCE hMyDll = LoadLibrary("MyDll.dll");
if (hMyDll )
{
   PFH_MyFunction pfhMyFunction =
(PFH_MyFunction)GetProcAddress(hMyDll, "MyFunction");
   if (pfhMyFunction )
            int iCode = (*pfhMyFunction)("Hello!");
   else
            MessageBox(NULL, "Cannot find function!", NULL,
                     MB_OK | MB_ICONASTERISK);
   FreeLibrary(hMyDll);
}
else
   MessageBox(NULL, "DLL could not be loaded", NULL,
                     MB_OK | MB_ICONASTERISK);
...
```

Глава 2
Разработка программного кода, учитывающего многозадачную архитектуру современных ОС

§ 2.1. Общие принципы организации многозадачности

2.1.1. Основные понятия и определения

Как уже было сказано, понятие процесса введено для реализации идей многозадачности. Необходимо различать **системные управляющие процессы**, представляющие работу супервизора операционной системы и занимающиеся распределением и управлением ресурсов, от всех других. Для системных управляющих процессов в большинстве операционных систем ресурсы распределяются изначально и однозначно. Эти процессы управляют ресурсами системы, поэтому обычно их не принято называть задачами. Термин же «задача» обычно употребляют только по отношению к процессам пользователей. Но это справедливо не для всех ОС. Например, в так называемых микроядерных ОС (в качестве примера можно привести ОС реального времени QNX) большинство управляющих программных модулей самой системы и даже драйверы имеют статус высокоприоритетных процессов, для выполнения которых необходимо выделить соответствующие ресурсы. Аналогично и в UNIX-системах выполнение системных программных модулей тоже имеет статус системных процессов, которые получают ресурсы для своего исполнения.

Если обобщать и рассматривать не только обычные ОС общего назначения, но и, например, ОС реального времени, то следует сказать, что ***процесс*** может находиться в ***активном*** и ***пассивном*** состояниях. В ***активном состоянии*** процесс может участвовать в конкуренции за использование ресурсов вычислительной системы, а в ***пассивном*** он только известен системе, но в конкуренции не участвует (хотя его существование в системе и сопряжено с предоставлением ему оперативной и/или внешней памяти).В свою очередь, ***активный процесс*** может быть в одном из следующих состояний [2, 5]:

– ***выполнения*** — все затребованные процессом ресурсы выделены. В этом состоянии в каждый момент времени может находиться только один процесс, если речь идет об однопроцессорной вычислительной системе;

– ***готовности к выполнению*** — ресурсы могут быть предоставлены, тогда процесс перейдет в состояние выполнения;

– ***блокирования*** или ***ожидания*** — затребованные ресурсы не могут быть предоставлены или не завершена операция ввода / вывода.

В большинстве операционных систем последнее состояние, в свою очередь, подразделяется на множество состояний ожидания, соответствующих определенному виду ресурса, из-за отсутствия которого процесс переходит в заблокированное состояние.

В обычных ОС, как правило, ***процесс появляется при запуске какой-нибудь программы***. Система создает для нового процесса соответствующий ***дескриптор*** (описатель) процесса, и он начинает выполняться. Поэтому пассивного состояния в обычных ОС не существует. В ОС реального времени (ОСРВ) ситуация иная. Обычно при проектировании системы реального времени уже заранее известен состав программ, которые должны будут выполняться. Известны и многие их параметры, которые необходимо учитывать при распределении ресурсов (например, объем памяти, приоритет, средняя длительность выполнения, открываемые файлы, используемые устройства и т. п.). Поэтому для них заранее заводят дескрипторы задач с тем, чтобы впоследствии не тратить драгоценное время на организацию дескриптора и поиск для него необходимых ресурсов. Таким образом, в ОСРВ многие процессы (задачи) могут находиться в состоянии бездействия, что показано на рис.2.1 пунктиром [2].

За время своего существования процесс может неоднократно совершать переходы из одного состояния в другое. Это обусловлено обращениями к операционной системе с запросами ресурсов и выполнения системных функций, которые предоставляет операционная система, взаимодейст-

вием с другими процессами, появлением сигналов прерывания от таймера, каналов и устройств ввода/вывода, а также других устройств. Возможные переходы процесса из одного состояния в другое отображены в виде графа состояний на рис. 2.1. Рассмотрим эти переходы более подробно.

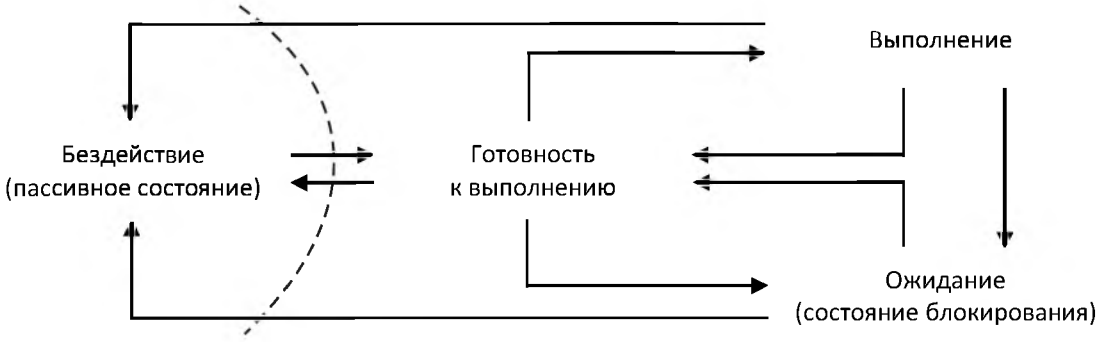

Рис. 2.1. Граф состояний процесса

Процесс из ***состояния бездействия*** может перейти в ***состояние готовности*** в следующих случаях [2]:

– по команде пользователя. Имеет место в тех диалоговых операционных системах, где программа может иметь статус задачи, а не просто быть исполняемым файлом и только на время исполнения получать статус задачи;

– при выборе из очереди планировщиком, что характерно для операционных систем, работающих в пакетном режиме;

– по вызову из другой задачи;

– по прерыванию от внешнего инициативного устройства;

– при наступлении запланированного времени запуска программы.

Последние два способа запуска задачи, при которых процесс из состояния бездействия переходит в состояние готовности, характерны для ОСРВ.

Процесс, который может исполняться, как только ему будет предоставлен процессор, ***находится в состоянии готовности***. Считается, что такому процессу уже выделены все необходимые ресурсы, за исключением процессора.

Из ***состояния выполнения*** процесс может выйти по одной из следующих причин [2, 5]:

– процесс завершается, при этом он посредством обращения к супервизору передает управление операционной системе и сообщает о своем завершении. В результате этих действий супервизор либо переводит его в список бездействующих процессов, либо уничтожает. В состояние бездействия процесс может быть переведен принудительно: по команде пользователя или путем обращения к супервизору операционной системы из другой задачи с требованием остановить данный процесс;

– процесс переводится супервизором операционной системы в состояние готовности к исполнению в связи с появлением более приоритетной задачи или в связи с окончанием выделенного ему кванта времени;

– процесс блокируется (переводится в состояние ожидания) либо вследствие запроса операции ввода/вывода (которая должна быть выполнена прежде, чем он сможет продолжить исполнение), либо в силу невозможности предоставить ему ресурс, запрошенный в настоящий момент, а также по команде оператора на приостановку задачи или по требованию через супервизор от другой задачи.

При наступлении соответствующего события (завершилась операция ввода/вывода, освободился затребованный ресурс, в оперативную память загружена необходимая страница виртуальной памяти и т. д.) процесс деблокируется и переводится в состояние ***готовности к исполнению***.

Таким образом, движущей силой, меняющей состояния процессов, являются ***события***. К одному из основных видов событий относятся ***прерывания***, которые будут рассмотрены далее.

Для того чтобы операционная система могла управлять процессами, она должна располагать всей необходимой для этого информацией. С этой целью на каждый процесс заводится специальная

информационная структура, называемая ***дескриптором процесса*** (описателем задачи, блоком управления задачей). В общем случае дескриптор процесса содержит следующую ***информацию*** [2]:

– идентификатор процесса (так называемый PID — process identificator);

– тип (или класс) процесса, который определяет для супервизора некоторые правила предоставления ресурсов;

– приоритет процесса, в соответствии с которым супервизор предоставляет ресурсы. В рамках одного класса процессов в первую очередь обслуживаются более приоритетные процессы;

– переменную состояния, которая определяет, в каком состоянии находится процесс (готов к работе, в состоянии выполнения, ожидание устройства ввода / вывода и т. д.);

– защищенную область памяти (или адрес такой зоны), в которой хранятся текущие значения регистров процессора, если процесс прерывается, не закончив работы. Эта информация называется ***контекстом задачи***;

– информацию о ресурсах, которыми процесс владеет и / или имеет право пользоваться (указатели на открытые файлы, информация о незавершенных операциях ввода / вывода и т. п.);

– место (или его адрес) для организации общения с другими процессами;

– параметры времени запуска (момент времени, когда процесс должен активизироваться, и периодичность этой процедуры);

– в случае отсутствия системы управления файлами — адрес задачи на диске в ее исходном состоянии и адрес на диске, куда она выгружается из оперативной памяти, если ее вытесняет другая.

Дескрипторы задач, как правило, постоянно располагаются в оперативной памяти с целью ускорить работу супервизора, который организует их в списки (очереди) и отображает изменение состояния процесса перемещением соответствующего описателя из одного списка в другой. Для каждого состояния ОС ведет соответствующий список задач, находящихся в этом состоянии. Для состояния ожидания может быть не один список, а столько, сколько различных видов ресурсов могут вызывать состояние ожидания.

В некоторых операционных системах количество дескрипторов определяется жестко и заранее (на этапе генерации варианта операционной системы или в конфигурационном файле, который используется при загрузке ОС), в других системах может выделять по мере необходимости участки памяти под новые дескрипторы. Например, в OS/2 максимально возможное количество дескрипторов задач определяется в конфигурационном файле CONFIG.SYS, а в Windows NT оно в явном виде не задается.

В ОСРВ чаще всего количество процессов фиксируется и, следовательно, целесообразно заранее определять (на этапе генерации или конфигурирования ОС) количество дескрипторов. Для использования таких ОС в качестве систем общего назначения обычно количество дескрипторов берется с некоторым запасом, и появление новой задачи связывается с заполнением этой информационной структуры. Поскольку дескрипторы процессов постоянно располагаются в оперативной памяти, то их количество не должно быть очень большим. При необходимости иметь большое количество задач один и тот же дескриптор может в разное время предоставляться для разных задач, но это сильно снижает скорость реагирования системы.

Для более эффективной обработки данных в системах реального времени целесообразно иметь постоянные задачи, полностью или частично всегда существующие в системе независимо от того, поступило на них требование или нет [2]. Каждая постоянная задача обладает некоторой собственной областью оперативной памяти (ОЗУ-резидентные задачи) независимо от того, выполняется задача в данный момент или нет. Эта область, в частности, может использоваться для хранения данных, полученных задачей ранее. Данные могут храниться в ней и тогда, когда задача находится в состоянии ожидания или даже в состоянии бездействия.

Для аппаратной поддержки работы операционных систем с этими информационными структурами (дескрипторами задач) в процессорах могут быть реализованы соответствующие механизмы. Так, например, в микропроцессорах Intel 80×86, начиная с поколения 80286, имеется специальный регистр TR (task register), указывающий местонахождение TSS (сегмента состояния задачи), в котором при переключении с задачи на задачу автоматически сохраняется содержимое регистров процессора.

Как уже было сказано, ***процесс*** — отдельная задача, для которой операционная система выделяет необходимые ресурсы, такие как виртуальная память, процессорное время и т. д. Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно

используя все ресурсы вычислительной системы, конкурируют друг с другом. В общем случае процессы просто никак не связаны между собой и могут принадлежать даже разным пользователям, разделяющим одну вычислительную систему.

Однако желательно иметь еще и возможность задействовать внутренний параллелизм, который может быть в самих процессах. Такой внутренний параллелизм встречается достаточно часто, и его использование позволяет ускорить их решение. Например, некоторые операции, выполняемые приложением, могут требовать для своего исполнения достаточно длительного использования центрального процессора. В этом случае при интерактивной работе с приложением пользователь вынужден долго ожидать завершения заказанной операции и не может управлять приложением до тех пор, пока операция не выполнится до самого конца. Такие ситуации встречаются достаточно часто, например, при обработке больших изображений в графических редакторах. Если же программные модули, исполняющие такие длительные операции, оформлять в виде самостоятельных «подпроцессов» (**потоков**, **тредов**), которые будут выполняться параллельно с другими «подпроцессами», то у пользователя появляется возможность параллельно выполнять несколько операций в рамках одного приложения (процесса). Операционная система не создает для потоков полноценную ***виртуальную машину***. Эти задачи ***не имеют своих собственных ресурсов*** — они развиваются в том же виртуальном адресном пространстве, могут пользоваться теми же файлами, виртуальными устройствами и иными ресурсами, что и данный процесс. Единственное, что им необходимо иметь в самостоятельном пользовании — это ***процессорный ресурс***. В однопроцессорной системе потоки разделяют между собой процессорное время так же, как это делают обычные процессы, а в мультипроцессорной системе могут выполняться одновременно, если не встречают конкуренции из-за обращения к иным ресурсам.

Главное, что обеспечивает ***многопоточность*** — это возможность параллельно выполнять несколько видов операций в одной прикладной программе. Особенно эффективно можно использовать многопоточность для выполнения распределенных приложений, например многопоточный сервер может параллельно выполнять запросы сразу нескольких клиентов. В операционной системе OS/2, одной из первых среди ОС, используемых на ПК, была введена многопоточность. В середине 90-х гг. для этой ОС было создано очень большое количество приложений, в которых использование механизмов многопоточной обработки приводило к существенному увеличению скорости выполнения вычислений.

Каждый процесс всегда состоит по крайней мере из одного потока, и только если имеется внутренний параллелизм, программист может разделить один поток на несколько параллельных. Потоки выполняются строго последовательно и имеют свой собственный программный счетчик и стек. Потоки, как и процессы, могут порождать потоки-потомки. Подобно традиционным процессам, т. е. процессам, состоящим из одного потока, каждый поток может находиться в ***одном из активных состояний***. ***Пока один поток заблокирован (или просто находится в очереди готовых к исполнению задач), другой поток того же процесса может выполняться***. Потоки разделяют процессорное время по тем же принципам, как это делают обычные процессы, т. е. в соответствии с механизмами, заложенными в различных дисциплинах диспетчеризации.

2.1.2. Планирование и диспетчеризация

Операционная система выполняет следующие основные функции, связанные с управлением процессами и потоками (задачами) [2]:

- создание и удаление;
- планирование процессов и диспетчеризация;
- синхронизация задач, обеспечение их средствами коммуникации.

Система управления задачами обеспечивает прохождение их через компьютер. В зависимости от состояния процесса ему должен быть предоставлен тот или иной ресурс. Например, новый процесс необходимо разместить в основной памяти, следовательно, ему необходимо выделить часть адресного пространства. Новый порожденный поток текущего процесса необходимо включить в общий список задач, конкурирующих между собой за ресурсы центрального процессора.

Создание и удаление задач осуществляется по соответствующим запросам от пользователей или от самих задач. Задача может породить новую задачу. При этом между процессами появляются «родственные» отношения. Порождающая задача называется ***предком***, ***родителем***, а порожденная — ***потомком*** или ***дочерней задачей***. ***Родитель*** может приостановить или удалить свою ***дочернюю задачу***, тогда как ***потомок*** не может управлять ***предком***.

Основным подходом к организации того или иного метода управления процессами, обеспечивающего эффективную загрузку ресурсов или выполнение каких-либо иных целей, является ***организация очередей*** процессов и ресурсов. Очевидно, что на распределение ресурсов влияют конкретные потребности тех задач, которые должны выполняться параллельно, т. е. можно столкнуться с ситуациями, когда невозможно эффективно распределять ресурсы с тем, чтобы они не простаивали. Например, всем выполняющимся процессам требуется некоторое устройство с последовательным доступом. Но поскольку оно не может распределяться между параллельно выполняющимися процессами, то процессы вынуждены будут очень долго ждать своей очереди. Таким образом, недоступность одного ресурса может привести к тому, что длительное время не будут использоваться и многие другие ресурсы.

Если же запустить задачи, которые не будут конкурировать между собой за неразделяемые ресурсы при параллельном выполнении, то процессы смогут выполняться быстрее, при этом имеющиеся в системе ресурсы будут использоваться более эффективно. Таким образом, актуальной является задача подбора такого множества процессов, чтобы при выполнении они как можно реже конфликтовали из-за имеющихся в системе ресурсов. Такая задача называется ***планированием*** вычислительных процессов [2].

Задача планирования процессов возникла очень давно — в первых пакетных ОС при планировании пакетов задач, которые должны были выполняться на компьютере и оптимально использовать его ресурсы. В настоящее время актуальность этой задачи не так велика. На первый план уже очень давно вышли задачи динамического (или краткосрочного) планирования, т. е. текущего, наиболее эффективного распределения ресурсов, возникающего практически при каждом событии. Задачи динамического планирования стали называть ***диспетчеризацией*** [2].

Очевидно, что планирование осуществляется гораздо реже, чем задача текущего распределения ресурсов между уже выполняющимися процессами и потоками. Первая операция выполняется раз в несколько минут, вторая может запускаться каждые 30 или 100 мс.

При рассмотрении ***стратегий планирования*** идет речь, как правило, о краткосрочном планировании, т. е. о ***диспетчеризации***. Долгосрочное планирование, как было отмечено ранее, заключается в подборе таких вычислительных процессов, которые бы меньше всего конкурировали между собой за ресурсы вычислительной системы.

Стратегия планирования определяет, какие процессы планируются на выполнение для того, чтобы достичь поставленной цели. Известно большое количество различных стратегий выбора процесса, которому необходимо предоставить центральный процессор. Среди них, прежде всего, можно назвать следующие стратегии [2]:

– по возможности заканчивать вычислительные процессы в том же самом порядке, в котором они были начаты;

– отдавать предпочтение более коротким процессам;

– предоставлять всем процессам одинаковые услуги, в том числе одинаковое время ожидания.

Когда говорят о ***диспетчеризации***, то всегда в явном или неявном виде имеют в виду понятие потока. Если ОС не поддерживает механизм потоков, то можно использовать понятие процесса. Известно большое количество правил (дисциплин диспетчеризации), в соответствии с которыми формируется очередь готовых к выполнению задач.

Различают два больших класса дисциплин обслуживания — ***бесприоритетные*** и ***приоритетные***. При ***бесприоритетном*** обслуживании выбор задачи производится в некотором заранее установленном порядке без учета их относительной важности и времени обслуживания. При реализации приоритетных дисциплин обслуживания отдельным задачам предоставляется преимущественное право попасть в состояние исполнения. Перечень дисциплин обслуживания и их классификация приведены на рис.2.2.

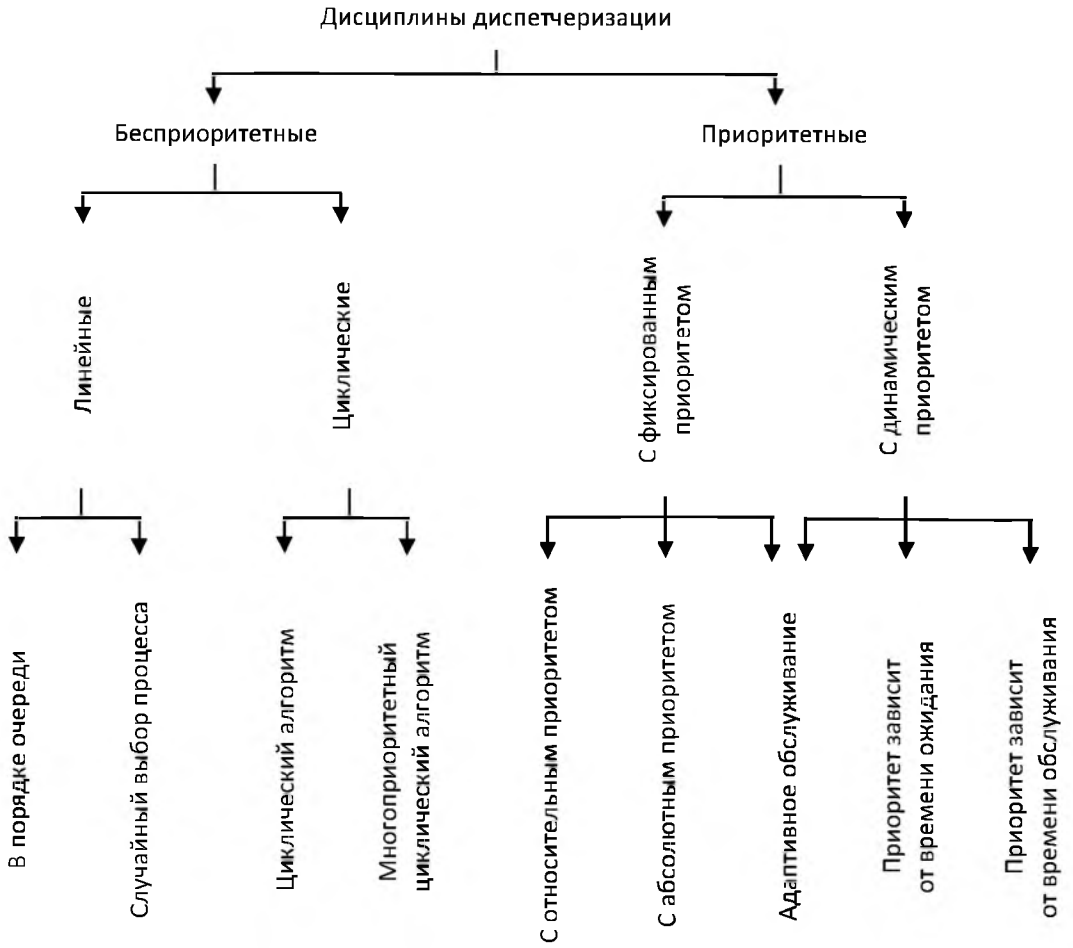

Рис. 2.2. Дисциплины диспетчеризации

Диспетчеризация с ***динамическими приоритетами*** требует дополнительных расходов на вычисление значений приоритетов исполняющихся задач, поэтому во многих ОСРВ используются методы диспетчеризации на основе статических (постоянных) приоритетов. При этом следует отметить, что динамические приоритеты позволяют реализовать гарантии обслуживания задач. Рассмотрим кратко некоторые наиболее часто используемые дисциплины диспетчеризации [2].

Самой простой в реализации является ***дисциплина FCFS*** (*first come — first served*), согласно которой задачи обслуживаются «в порядке очереди», т. е. в порядке их появления. Те задачи, которые были заблокированы в процессе работы (попали в какое-либо из состояний ожидания, например, из-за операций ввода/вывода), после перехода в состояние готовности ставятся в эту очередь готовности перед теми задачами, которые еще не выполнялись. Другими словами, образуются две очереди (рис. 2.3): одна очередь образуется из новых задач, а вторая — из ранее выполнявшихся задач, но попавших в состояние ожидания. Такой подход позволяет реализовать стратегию обслуживания «по возможности заканчивать вычисления в порядке их появления». Эта дисциплина обслуживания не требует внешнего вмешательства в ход вычислений, при ней не происходит перераспределения процессорного времени.

К ***достоинствам*** этой дисциплины можно отнести прежде всего простоту реализации и малые расходы системных ресурсов на формирование очереди задач.

Однако эта дисциплина приводит к тому, что ***при увеличении загрузки вычислительной системы растет и среднее время ожидания обслуживания***, причем короткие задания (требующие небольших затрат машинного времени) вынуждены ожидать столько же, сколько и трудоемкие задания. Избежать этого недостатка позволяют дисциплины ***SJN*** и ***SRT***.

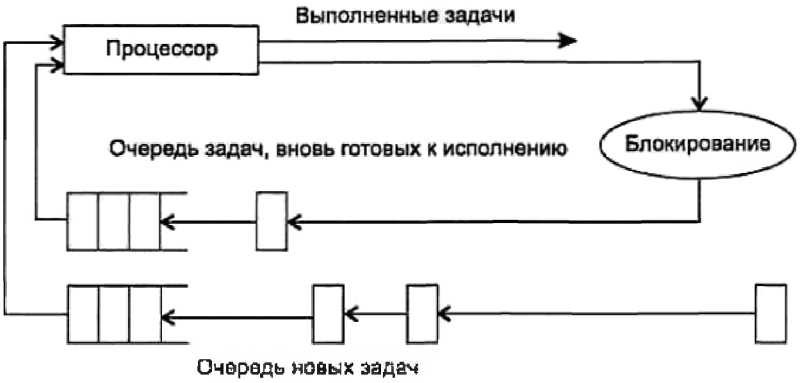

Рис.2.3. Дисциплина диспетчеризации FCFS

Дисциплина обслуживания SJN (*shortest job next*, что означает «следующим будет выполняться кратчайшее задание») требует, чтобы для каждого задания была известна оценка в потребностях машинного времени. Необходимость сообщать ОС характеристики задач, в которых описывались бы потребности в ресурсах вычислительной системы, привела к тому, что были разработаны соответствующие языковые средства. Язык ***JCL*** (*job control language* — язык управления заданиями) был одним из наиболее известных. Пользователи вынуждены были указывать предполагаемое время выполнения, и для того чтобы они не злоупотребляли возможностью указать заведомо меньшее время выполнения (с целью получить результаты раньше других), ввели подсчет реальных потребностей. Диспетчер задач сравнивал заказанное время и время выполнения и в случае превышения указанной оценки в данном ресурсе ставил данное задание не в начало, а в конец очереди. Еще в некоторых ОС в таких случаях использовалась система штрафов, при которой в случае превышения заказанного машинного времени оплата вычислительных ресурсов осуществлялась уже по другим расценкам.

Дисциплина обслуживания SJN предполагает, что имеется только одна очередь заданий, готовых к выполнению. ***Задания, которые в процессе своего исполнения были временно заблокированы*** (например, ожидали завершения операций ввода/вывода), вновь попадают в конец очереди заданий, готовых к выполнению, наравне с вновь поступающими. Это приводило к тому, что задания, которым требуется очень немного времени для своего завершения, вынуждены ожидать процессор наравне с длительными работами, что не всегда хорошо. Для устранения этого недостатка и была предложена ***дисциплина SRT*** (*shortest remaining time* — следующее задание требует меньше всего времени для своего завершения).

Все три указанные дисциплины обслуживания могут использоваться для пакетных режимов обработки, когда пользователь не вынужден ожидать реакции системы, а просто сдает свое задание и через несколько часов получает результаты вычислений. Для интерактивных вычислений желательно прежде всего обеспечить приемлемое время реакции системы и равенство в обслуживании, если система является мульти терминальной. Если же это однопользовательская система, но с возможностью мультипрограммной обработки, то желательно, чтобы те программы, с которыми производится непосредственная работа, имели лучшее время реакции, нежели фоновые задания. При этом желательно, чтобы некоторые приложения, выполняясь без непосредственного участия пользователя (например, программа получения электронной почты, использующая модем и коммутируемые линии для передачи данных), гарантированно получали бы необходимую часть процессорного времени. Для решения подобных проблем используется дисциплина обслуживания, называемая ***RR*** (*round robin* — круговая, карусельная), и приоритетные методы обслуживания.

Дисциплина обслуживания RR предполагает, что каждая задача получает процессорное время порциями (говорят, ***квантами времени, q***). После окончания кванта времени q задача снимается с процессора и он передается следующей задаче. Снятая задача ставится в конец очереди задач, готовых к выполнению. Эта дисциплина обслуживания иллюстрируется на рис. 2.4. Для оптимальной работы системы необходимо правильно выбрать закон, по которому кванты времени выделяются задачам.

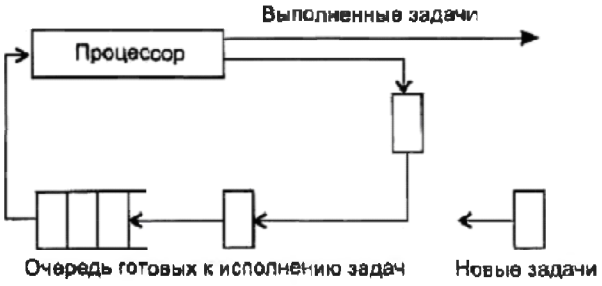

Рис.2.4. Карусельная (*round robin*) дисциплина диспетчеризации

Величина кванта времени q выбирается как компромисс между приемлемым временем реакции системы на запросы пользователей (с тем чтобы их простейшие запросы не вызывали длительного ожидания) и накладными расходами на частую смену контекста задач. Очевидно, что при прерываниях ОС вынуждена сохранить достаточно большой объем информации о текущем (прерываемом) процессе, поставить дескриптор снятой задачи в очередь, загрузить контекст задачи, которая теперь будет выполняться. Если величина q велика, то при увеличении очереди готовых к выполнению задач реакция системы станет плохой. Если же величина мала, то относительная доля накладных расходов на переключения между исполняющимися задачами станет большой и это ухудшит производительность системы. В некоторых ОС есть возможность указывать в явном виде величину q либо диапазон ее возможных значений.

Дисциплина диспетчеризации RR — это одна из самых распространенных дисциплин. Однако бывают ситуации, когда ОС не поддерживает в явном виде дисциплину карусельной диспетчеризации. Например, в некоторых ОС реального времени используется диспетчер задач, работающий по принципам абсолютных приоритетов (процессор предоставляется задаче с максимальным приоритетом, а при равенстве приоритетов он действует по принципу очередности). Другими словами, снять задачу с выполнения может только появление задачи с более высоким приоритетом. Поэтому если нужно организовать обслуживание задач таким образом, чтобы все они получали процессорное время равномерно и равноправно, то системный оператор может сам организовать такую дисциплину. Для этого достаточно всем пользовательским задачам присвоить одинаковые приоритеты и создать одну высокоприоритетную задачу, которая не должна ничего делать, но которая, тем не менее, будет по таймеру через указанные интервалы времени планироваться на выполнение. Эта задача снимет с выполнения текущее приложение — оно будет поставлено в конец очереди, и поскольку этой высокоприоритетной задаче на самом деле ничего делать не надо, то она тут же освободит процессор и из очереди готовности будет взята следующая задача.

Рассмотренные дисциплины диспетчеризации процессов могут быть разбиты на два класса: ***вытесняющие*** (*preemptive*) и ***невытесняющие*** (*non preemptive*) [5]. В большинстве современных ОС для мощных вычислительных систем, а также и в ОС для ПК, ориентированных на высокопроизводительное выполнение приложений (Windows NT, OS/2, Linux), реализована ***вытесняющая многозадачность***.

Диспетчеризация без перераспределения процессорного времени называется ***«невытесняющая многозадачность»*** (non-preemptive multitasking). Это такой способ диспетчеризации процессов, при котором активный процесс выполняется до тех пор, пока он сам, что называется, по собственной инициативе не отдаст управление диспетчеру задач для выбора из очереди другого, готового к выполнению процесса или потока. Дисциплины обслуживания ***FCFS, SJN, SRT*** относятся к ***невытесняющим***.

Диспетчеризация с перераспределением процессорного времени между задачами называется ***«вытесняющая многозадачность»*** (*preemptive multitasking*). Это такой способ, при котором решение о переключении процессора с выполнения одного процесса на выполнение другого процесса принимается диспетчером задач, а не самой активной задачей. При вытесняющей многозадачности механизм диспетчеризации задач целиком сосредоточен в операционной системе, и программист может писать свое приложение, не заботясь о том, как оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции: определяет момент сня-

тия с выполнения текущей задачи, сохраняет ее контекст в дескрипторе задачи, выбирает из очереди готовых задач следующую и запускает ее на выполнение, предварительно загрузив ее контекст. Дисциплина ***RR*** и многие другие, построенные на ее основе, относятся к ***вытесняющим***.

Одна из проблем, которая возникает при выборе подходящей дисциплины обслуживания, — это гарантия обслуживания. Дело в том, что при некоторых дисциплинах, например при использовании дисциплины абсолютных приоритетов, низкоприоритетные процессы оказываются обделенными многими ресурсами, и прежде всего процессорным временем. Возникает реальная дискриминация низкоприоритетных задач, и ряд таких процессов, имеющих к тому же большие потребности в ресурсах, могут очень длительное время откладываться или, в конце концов, вообще быть не выполнены.Поэтому вопрос гарантии обслуживания является очень актуальным. Существуют различные ***дисциплины диспетчеризации***, учитывающие жесткие временные ограничения, но не существует дисциплин, которые могли бы предоставить больше процессорного времени, чем может быть в принципе выделено.

Планирование с учетом жестких временных ограничений легко реализовать, организуя очередь готовых к выполнению процессов в порядке возрастания их временных ограничений. Основным недостатком такого простого упорядочения является то, что процесс (за счет других процессов) может быть обслужен быстрее, чем это ему реально необходимо. Для того чтобы избежать этого, проще всего процессорное время выделять все-таки квантами. С учетом сказанного ***гарантировать обслуживание можно следующими тремя способами*** [2]:

– выделять минимальную долю процессорного времени некоторому классу процессов, если по крайней мере один из них готов к исполнению. Например, можно отводить 20 % от каждых 10 мс процессам реального времени, 40 % от каждых 2 с — интерактивным процессам и 10 % от каждых 5 мин — фоновым процессам;

– выделять минимальную долю процессорного времени некоторому конкретному процессу, если он готов к выполнению;

– выделять столько процессорного времени некоторому процессу, чтобы он мог выполнить свои вычисления к сроку.

Для сравнения алгоритмов диспетчеризации обычно используются следующие ***критерии*** [2]:

1. Загрузка центрального процессора (*CPU utilization*). В большинстве персональных систем средняя загрузка процессора не превышает 2–3 %, доходя в моменты выполнения сложных вычислений и до 100 %. В реальных системах, где компьютеры выполняют очень много работы, например в серверах, загрузка процессора колеблется в пределах 15–40 % для легко загруженного процессора и до 90–100 % для сильно загруженного процессора.

Пропускная способность (*CPU throughput*) процессора, которая может измеряться количеством процессов, выполняемых в единицу времени.

Время оборота (*turnaround time*). Для некоторых процессов важным критерием является полное время выполнения, т. е. интервал от момента появления процесса во входной очереди до момента его завершения. Это время названо временем оборота и включает время ожидания во входной очереди, время ожидания в очереди готовых процессов, время ожидания в очередях к оборудованию, время выполнения в процессоре и время ввода / вывода.

Время ожидания (*waiting time*). Под временем ожидания понимается суммарное время нахождения процесса в очереди готовых процессов.

Время отклика (*response time*). Для интерактивных программ важным показателем является время отклика или время, прошедшее от момента попадания процесса во входную очередь до момента первого обращения к терминалу. Очевидно, что простейшая стратегия краткосрочного планировщика должна быть направлена на максимизацию средних значений загруженности и пропускной способности, времени ожидания и времени отклика.

Правильное планирование процессов сильно влияет на производительность всей системы. Можно выделить следующие главные причины, приводящие к уменьшению производительности системы [2]:

– накладные расходы на переключение процессора. Они определяются не только переключениями контекстов задач, но и перемещениями страниц виртуальной памяти, а также необходимостью обновления данных в КЭШе;

– переключение на другой процесс в тот момент, когда текущий процесс выполняет критическую секцию, а другие процессы активно ожидают входа в свою критическую секцию. В этом случае потери будут особенно велики.

В случае использования мультипроцессорных систем применяются следующие методы повышения производительности системы [2]:

– совместное планирование, при котором все потоки одного приложения одновременно выбираются для выполнения процессорами и одновременно снимаются с них;

– планирование, при котором находящиеся в критической секции задачи не прерываются, а активно ожидающие входа в критическую секцию задачи не выбираются до тех пор, пока вход в секцию не освободится;

– планирование с учетом так называемых советов программы во время ее выполнения. Например, в известной своими новациями ОС Mach имелись два класса таких советов (*hints*): указания о снятии текущего процесса с процессора, а также указания о том процессе, который должен быть выбран взамен текущего.

При выполнении программ, реализующих какие-либо задачи контроля и управления (что характерно прежде всего для систем реального времени), может случиться такая ситуация, когда одна или несколько задач не могут быть решены в течение длительного промежутка времени из-за возросшей нагрузки в вычислительной системе. Потери, связанные с невыполнением таких задач, могут быть больше, чем потери от невыполнения программ с более высоким приоритетом. При этом оказывается целесообразным временно изменить приоритет «аварийных» задач, для которых истекает отпущенное для них время обработки. После выполнения этих задач их приоритет восстанавливается. Поэтому почти в любой ОСРВ имеются ***средства для изменения приоритета программ***. Есть такие средства и во многих ОС, которые не относятся к классу ОСРВ. Введение механизмов ***динамического изменения приоритетов*** позволяет реализовать более быструю реакцию системы на короткие запросы пользователей (что очень важно при интерактивной работе), но при этом гарантировать выполнение любых запросов.

Класс задач, имеющих самые высокие значения приоритета, называется ***критическим*** (*time critical*). Этот класс предназначается для задач, которые принято называть задачами реального времени, т. е. для них должен быть обязательно предоставлен определенный минимум процессорного времени. Наиболее часто встречающимися задачами, которые относят к этому классу, являются задачи коммуникаций (например, задача управления последовательным портом, принимающим биты с коммутируемой линии, к которой подключен модем, или задачи управления сетевым оборудованием). Если такие задачи не получат управление в нужный момент времени, то сеанс связи может прерваться.

Следующий класс задач имеет название ***приоритетного***. Поскольку к этому классу относят задачи, которые выполняют по отношению к остальным задачам роль сервера, то его еще иногда называют ***серверным***. Приоритет таких задач должен быть выше — это будет гарантией того, что запрос на некоторую функцию со стороны обычных задач выполнится сразу, а не станет дожидаться, пока до него дойдет очередь на фоне других пользовательских приложений.

Большинство задач относят к обычному классу — его еще называют ***регулярным*** или ***стандартным***. По умолчанию система программирования порождает задачу, относящуюся именно к этому классу.

Наконец, существует еще класс фоновых задач, называемый в OS/2 ***остаточным***. Программы этого класса получают процессорное время только тогда, когда нет задач из других классов, которым сейчас нужен процессор. В качестве примера такой задачи можно привести программу проверки электронной почты.

Внутри каждого из рассмотренных классов задачи, имеющие одинаковый уровень приоритета, выполняются в соответствии с дисциплиной ***RR***. Переход от одного потока к другому происходит либо по окончании отпущенного ему кванта времени, либо по системному прерыванию, передающему управление задаче с более высоким приоритетом (таким образом, система вытесняет задачи с более низким приоритетом для выполнения задач с более высоким приоритетом и может обеспечить быструю реакцию на важные события).

§ 2.2. Принципы разработки многопоточного приложения в ОС Windows

2.2.1. Основы многозадачности и многопоточности в Windows

В Windows каждый процесс имеет свое собственное виртуальное адресное пространство (4 Гбайт). Процесс состоит из кода, данных и других системных ресурсов, таких как открытые файлы, каналы (*pipes*), синхронизирующие объекты. Однако процесс — статический объект, который сам по себе действия не производит. Поток (*thread*) — базовый объект, которому операционная система распределяет время центрального процессора. Поток выполняет команды программы с учетом заданного ему маршрута. Каждый процесс представляет собой один начальный поток, который иногда называют ***первичным потоком***.

Первичный поток способен создать вторичные потоки. Все потоки, принадлежащие одному процессу, имеют совместный доступ к его ресурсам. Все они работают под управлением команд одной и той же программы, обращаются к одним и тем же глобальным переменным, записывают информацию в одну и ту же область памяти и имеют доступ к одним и тем же объектам. В целом следует отметить, что программа может выполнять поставленные задачи и без организации потоков, однако в данном случае для запуска «дочернего» процесса необходимо временно приостанавливать основной процесс, что приводит к замедлению выполнения программы в целом. Дополнительные потоки создаются в первую очередь в том случае, когда программа должна выполнять асинхронные операции, работает одновременно с несколькими окнами.

Организация многозадачности в MS Windows различается в линейках 9x и NT. В Windows 9x реализована ***приоритетная многозадачность***. В данном случае каждому активному потоку предоставляется определенный промежуток времени работы процессора. По истечении данного промежутка управление автоматически передается следующему потоку. Это не дает программам возможность полностью захватывать ресурсы процессора.

Windows NT использует ***вытесняющую многозадачность***. Выполнение всех процессов строго контролируется операционной системой. Последняя выделяет каждой из программ некоторое количество процессорного времени и периодически производит переключение между запущенными на компьютере программами. Обратившись к специальному системному вызову, вы можете как бы приостановить (*sleep*) выполнение программы, однако если вы этого не сделаете, то со временем операционная система сделает это за вас. Подвисание одной из программ не приведет к подвисанию всей системы.

В общем и целом вытесняющая многозадачность, которая ранее рассмотрена более подробно, выглядит привлекательней. Однако за все приходится платить, — в первую очередь расплачиваться приходится программистам, которые разрабатывают приложения, предназначенные для работы в среде с приоритетной многозадачностью. Представьте себе, что на компьютере работают несколько программ, использующих один и тот же файл журнала ошибок. Если вам необходимо сделать запись в этом файле, то в Windows 9x вы можете без лишних сложностей открыть файл, записать в него данные, а затем закрыть его. Если при этом вы ни разу не обратились к специальным системным функциям, то можете быть уверены, что в то время, пока вы работали с файлом, ни одна другая программа не обратилась к этому же файлу (так как фактически в ходе работы с файлом на компьютере работает только одна ваша программа, а все остальные программы находятся в состоянии ожидания).

В среде Windows NT все не так просто. Предположим, что один из потоков открыл файл и начал запись, но в этот момент операционная система передала управление другому потоку. Что произойдет, если другой поток попытается открыть тот же самый файл? Либо этого сделать не удастся, либо другой поток откроет файл в режиме совместного доступа (*for sharing*), что может не соответствовать вашим ожиданиям. Даже если входе работы с файлом операционная система не осуществила передачу управления другому потоку, получить доступ к файлу может попытаться поток, работающий на другом процессоре. В результате вы столкнетесь с той же проблемой. На системном уровне каждый поток представляет собой объект, созданный системным менеджером объектов. Аналогично остальным системным объектам поток содержит данные (атрибуты) и методы (функции). Схематически объект-поток представлен на рис. 2.5 [12].

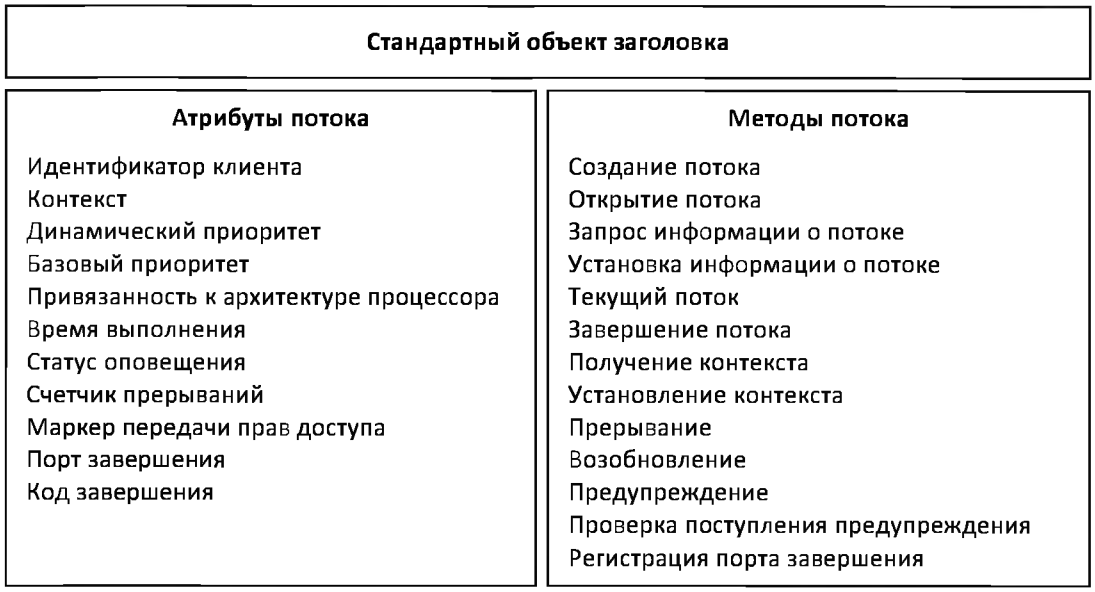

Рис.2.5. Схема объекта потока

Для большинства методов потока имеются соответствующие API-функции Win32. Windows защищает свои внутренние структуры от прямого вмешательства пользовательских программ. В отличие от более привилегированных программ, функционирующих на уровне ядра операционной системы, пользовательские программы не могут прямо анализировать или изменять параметры системных объектов. Все операции с ними выполняются посредством функций Win32 API. Windows предоставляет дескриптор, идентифицирующий объект. При выполнении операций с объектом его дескриптор передается в качестве аргумента одной из API-функций. Свои дескрипторы имеют потоки, процессы, семафоры, файлы и другие объекты. Внутренняя структура объектов доступна только менеджеру объектов. Функция, создающая поток, возвращает дескриптор нового объекта. С помощью этого дескриптора можно выполнить следующие операции:

- повысить или понизить плановый приоритет потока;
- приостановить поток и возобновить его выполнение;
- прекратить выполнение потока;
- определить код завершения потока.

В ОС Windows потоки, процессы, семафоры и исключающие семафоры могут иметь несколько разных дескрипторов. Завершив работу с объектом, необходимо вызвать функцию CloseHandle, которая, закрыв последний дескриптор, сама уничтожит объект. В целом в Windows каждый процесс не может одновременно поддерживать более 65 536 открытых дескрипторов.

Работа с потоками не сводится только к их запуску и остановке. Необходимо обеспечить совместное функционирование потоков. Для организации эффективного взаимодействия между несколькими потоками необходимо производить контроль за их временными параметрами. Контроль осуществляется [12]:

- установлением приоритетов;
- синхронизацией.

Приоритет потока определяет, насколько часто данный поток получает доступ к центральному процессору. Синхронизация регулирует порядок обращения потоков к общим ресурсам. Когда системная программа-планировщик останавливает один поток и ищет другой, который должен быть запущен следующим, она отдает предпочтение потокам, имеющим наиболее высокий приоритет. Обработчики системных прерываний всегда имеют более высокий приоритет по сравнению с пользовательскими процессами. Каждому процессу присущ собственный приоритет. Базовый плановый приоритет потока определяется на основе приоритета процесса, который является владельцем этого потока. Всего различают 32 уровня приоритета — от 0 до 31. При этом приоритеты уровня от 0 до 15 называются переменными приоритетами, от 16 до 31 — фиксированными приоритетами. Схема наследования приоритетов потока показана на рис.2.6 [12].

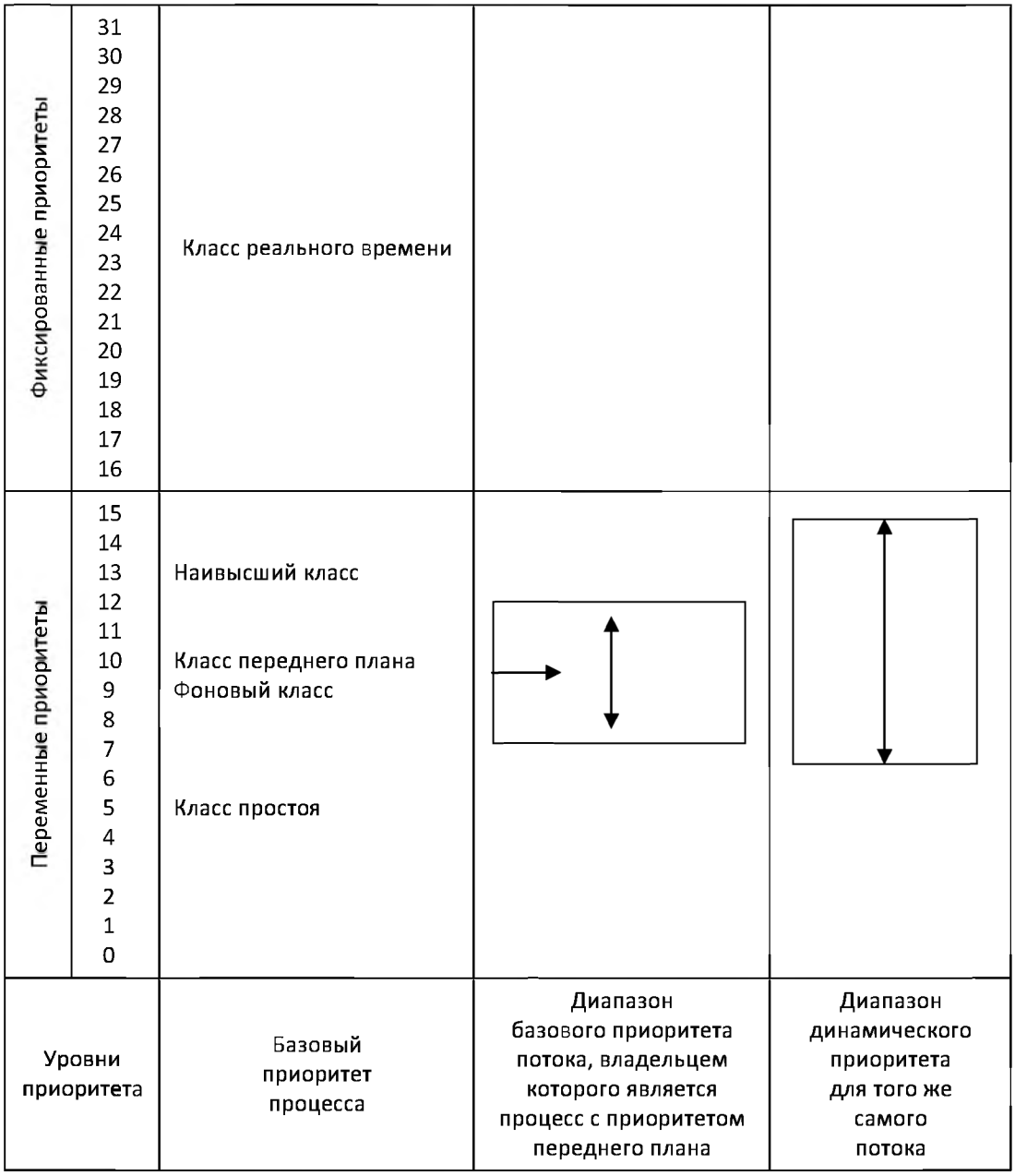

Рис. 2.6. Схема наследования приоритета потоков от исходного приоритета процесса

Среди атрибутов объекта потока различают базовый и динамический приоритеты. При вызове команды для изменения приоритета потока меняется его базовый приоритет, который не может быть выше или ниже приоритета процесса-владельца более чем на два уровня. Операционная система способствует «продвижению» потока для его выполнения. Для этого система поддерживает динамический приоритет потоков, которые выполняют важные задачи. Например, если процесс выводит информацию в окно либо считывает данные с диска, временно повышается приоритет всех потоков такого процесса. Эти временные приращения в сумме с базовым приоритетом образуют динамический приоритет процесса. Планировщик определяет очередность выполнения потоков на основании их динамического приоритета. Со следующим тактом процесса приращение приоритета начинает уменьшаться на один уровень, постепенно достигая уровня базового приоритета.

Выбирая поток, который будет выполняться следующим, программа-планировщик начинает просмотр очереди заданий с потоков, имеющих наивысший приоритет, выполняет их, а затем переходит к остальным потокам. Однако иногда в очереди заданий содержатся не все созданные в системе потоки, поскольку некоторые из них могут быть приостановлены или заблокированы. В каждый момент времени поток может находиться в одном из шести состояний:

• **Ready** (готов) — поставлен в очередь и ожидает выполнения;

• **Standby** (ближайший) — готов быть выполненным следующим;

• **Running** (выполнение) — находится в режиме выполнения и взаимодействует с центральным процессором;

• **Waiting** (ожидание) — не выполняется, ожидая сигнала выполнения;

• **Transition** (промежуточное) — будет выполняться после того, как система загрузит его контекст;

• **Terminated** (завершен) — выполнение завершено, однако объект не удален.

Когда программа-планировщик выбирает из очереди готовый к выполнению поток, она загружает его ***контекст***. В состав контекста входит набор значений регистров процессора, стек ядра, блок параметров окружения потока и пользовательский стек в адресном пространстве процесса, который является владельцем данного потока. Если часть контакта была записана на диск, поток проходит в промежуточное состояние и ожидает, пока система соберет все составные части контекста.

Чтобы потоки могли надежно работать, их необходимо синхронизировать. Представьте себе, что один поток создает кисть, а затем создает несколько потоков, которые вместе используют эту кисть для выполнения графических операций. Первый поток не должен уничтожать кисть до тех пор, пока другие потоки не завершат операции рисования. Или представьте себе, что один поток принимает данные, введенные пользователем, и записывает их в файл, а другие потоки считывают данные из этого файла и образовывают введенный текст. Считывание не должно происходить в тот момент, когда идет запись. В обоих случаях надо принять меры по координации последовательности операций в нескольких потоках.

Одно из возможных решений заключается в создании глобальной переменной типа Boolean, которую один из потоков будет использовать с целью информирования других потоков о том, что объект занят. Например, поток, записывающий данные в файл, может присвоить переменной ***bDone*** значение ***TRUE***, а потоки, считывающие данные из файла, будут циклически просматривать эту переменную до тех пор, пока ее значение не изменится. Такая схема вполне работоспособна, однако циклический просмотр флага несколькими потоками занимает много времени процессора. Вот почему в Win32 поддерживается набор синхронизирующих объектов [9, 12]. Перечислим их.

Объект типа ***исключающий семафор*** функционирует подобно узкой двери, «пропуская», т. е. давая допуск, одновременно по одному потоку.

Объект типа ***семафор*** функционирует подобно многостворчатой двери, ограничивая количество потоков, которые могут проходить через него одновременно.

Объект типа ***события*** передает глобальный сигнал, воспринимаемый любым потоком, для которого он адресован.

Объект типа ***критический раздел*** аналогичен исключающему семафору, но работает только в пределах одного процесса.

Все указанные объекты являются системными и создаются менеджером объектов. Хотя каждый синхронизирующий объект координирует определенный вид взаимодействия, все они функционируют сходным образом. Поток, который должен выполнить определенную синхронизируемую операцию, ожидает ответа от одного из этих объектов и осуществляет свои функции только после получения на то разрешения. Программа-планировщик удаляет ожидающие объекты из очереди запуска, с тем чтобы они не занимали времени центрального процессора. После получения соответствующего сигнала планировщик возобновляет работу потока. Место и способ получения сигнала зависят от конкретного объекта.

Исключающие семафоры, обычные семафоры и события позволяют координировать работу потоков, принадлежащих разным процессам, в то время как критические разделы воспринимаются лишь потоками одного процесса. Если один процесс создает другой процесс, то дочерний процесс часто наследует дескрипторы имеющихся синхронизирующих объектов. Объекты критических разделов не могу быть унаследованы.

С позиции фундаментальности синхронизирующий объект, подобно другим системным объектам, представляет собой структуру данных. Синхронизирующие объекты могут находиться в двух состояниях: при наличии сигнала и при отсутствии такового. Потоки взаимодействуют с синхро-

низирующими объектами путем установки сигнала или путем его ожидания. Поток, находящийся в состоянии ожидания, блокируется и не выполняется. При наличии сигнала ожидающий поток завладевает объектом, выключает сигнал, выполняет определенные синхронизирующие операции, а затем опять включает сигнал и освобождает объект.

Потоки могут ожидать разрешения не только от исключающих и обычных семафоров, событий и критических разделов, но и от других объектов. Иногда возникает ситуация, когда необходимо ожидать разрешения от процесса, другого потока, таймера или файлового объекта. Все эти объекты имеют свое предназначение, но подобно синхронизирующим объектом они способны давать сигналы разрешения. Процессы и потоки сигнализируют о своем завершении, объекты-таймеры — об истечении определенного интервала времени, а файловые объекты — о завершении операций чтения или записей файлов. Потоки могут ожидать появления любого из этих сигналов.

2.2.2. API-функции для реализации механизма многопоточности

В Win32 API определены следующие функции работы с процессами и потоками (табл. 2.1) [4].

Таблица 2.1

Название функции	Выполняемое действие
1	*2*
AttachThreadInput	Переключение механизмов ввода с одной нити на другую
CommandLineToArgvW	Производит разбор командной строки в Unicode
CreateProcess	Создает процесс
CreateRemoteThread	Создает поток в адресном пространстве другого процесса
CreateThread	Создает поток
ExitProcess	Завершает процесс и все его потоки
ExitThread	Завершает поток
FreeEnvironmentStrings	Освобождает память области переменных среды
GetCommandLine	Возвращает указатель на командную строку
GetCurrentProcess	Возвращает описатель (*handle*) текущего процесса
GetCurrentProcessId	Возвращает идентификатор текущего процесса
GetCurrentThread	Возвращает описатель (*handle*) текущего потока
GetCurrentThreadId	Возвращает идентификатор текущего потока
GetEnvironmentStrings	Возвращает строку переменных среды
GetEnvironmentVariable	Возвращает значение указанной переменной среды
GetExitCodeProcess	Возвращает код завершения процесса
GetExitCodeThread	Возвращает код завершения потока
GetPriorityClass	Возвращает класс приоритета процесса
GetProcessAffinityMask	Сообщает, на каких процессорах разрешено исполнение процесса
GetProcessShutdownParameters	Сообщает параметры поведения процесса при завершении работы системы
GetProcessTimes	Возвращает временные характеристики процесса
GetCurrentProcess	Сообщает версию Windows, для которой предназначен процесс
GetProcessWorkingSetSize	Возвращает характеристики доступного процессу адресного пространства
GetStartupInfo	Возвращает параметры процесса, полученные им при создании
GetThreadPriority	Сообщает приоритет указанного потока
GetThreadTimes	Возвращает временные характеристики указанного потока
OpenProcess	Возвращает описатель (*handle*) указанного процесса
ResumeThread	Уменьшает счетчик задержаний потока (или запускает его)
SetEnvironmentVariable	Устанавливает значение указанной переменной среды
SetPriorityClass	Устанавливает класс приоритета процесса

Окончание табл. 2.1

1	2
SetProcessShutdownParameters	Устанавливает параметры поведения процесса при завершении работы системы
SetThreadAffinityMask	Устанавливает, на каких процессорах разрешено исполнение потока
SetThreadPriority	Устанавливает приоритет указанного потока
Sleep	Задерживает исполнение потока на указанное количество миллисекунд
SleepEx	Задерживает исполнение до наступления события ввода/вывода или на время
SetProcessWorkingSetSize	Устанавливает характеристики доступного процессу адресного пространства
SuspendThread	Приостанавливает исполнение указанного потока
TerminateProcess	Завершает указанный процесс
TerminateThread	Завершает указанный поток
TlsAlloc	Распределяет индекс локальной памяти потока (*thread local storage TLS*)
TlsFree	Освобождает индекс TLS
TlsGetValue	Возвращает данные, размещенные в TLS с указанным индексом
TlsSetValue	Помещает данные в TLS с указанным индексом
WaitForInputIdle	Ждет, пока не начнется ввод для указанного процесса
WinExec	Выполняет указанное приложение

Подробное описание функций приведено в Win32 Programmer`s Reference. Далее рассмотрим только некоторые основные функции [4].

Функция CreateProcess()создает новый процесс и его первичный поток. Новый процесс исполняет указанный исполняемый файл. Формат функции:

```
BOOL CreateProcess(LPCTSTR lpApplicationName,      // имя исполняемого файла
                   LPTSTR lpCommandLine, // командная строка
                   LPSECURITY_ATTRIBUTES lpProcessAttributes, //атрибуты защиты процесса
                   LPSECURITY_ATTRIBUTES lpThreadAttributes,  // атрибуты защиты потока
                   BOOL bInheritHandles,                      // флаг наследования описателей
                   DWORD dwCreationFlags,                     // флаги создания
                   LPVOID lpEnvironment,                 // указатель блока переменных среды
                   LPCTSTR lpCurrentDirectory,                // текущий каталог
                   LPSTARTUPINFO lpStartupInfo,               // блок начальных параметров
                   LPPROCESS_INFORMATION lpProcessInformation );      // указатель
                                                  // структуры, описывающей порожденный процесс
```

Функция возвращает TRUE в случае успеха, FALSE — в случае неудачи и имеет следующие параметры:

- lpApplicationName — указатель на строку, содержащую имя исполняемой программы. Имя может быть полное. Если оно неполное, то поиск файла производится в текущем каталоге. Параметру может быть присвоено значение NULL. В этом случае в качестве имени файла выступает первая выделенная пробелами лексема из строки lpCommandLine;
- lpCommandLine — указатель командной строки. Если параметр lpApplicationName имеет значение NULL, то имя исполняемого файла выделяется из lpCommandLine, а поиск исполняемого файла производится в соответствии с правилами, действующими в системе;
- lpProcessAttributes — указатель на структуру, описывающую параметры защиты процесса. Если параметру присвоено значение NULL, то устанавливаются атрибуты «по умолчанию»;
- lpThreadAttributes — указатель на структуру, описывающую параметры защиты первичного потока. Если параметру присвоено значение NULL, то устанавливаются атрибуты «по умолчанию»;
- bInheritHandles — определяет, будет ли порожденный процесс наследовать описатели (*handles*) объектов родительского процесса. Например, если родительский процесс *AB*, то он получил описатель процесса *B* и может им манипулировать. Если теперь он порождает процесс *C* с параметром bInheritHandles, равным TRUE, то и процесс *C* сможет работать с описателем процесса *B*;

• dwCreationFlags — определяет некоторые дополнительные условия создания процесса и его класс приоритета;

• lpEnvironment —указатель на блок переменных среды порожденного процесса. Если этот параметр равен NULL, то порожденный процесс наследует среду родителя. Иначе он должен указывать на завершающийся нулем блок строк, каждая из которых завершается нулем (аналогично DOS);

• lpCurrentDirectory — указатель на строку, содержащую полное имя текущего каталога порожденного процесса. Если этот параметр равен NULL, то порожденный процесс наследует каталог родителя;

• lpStartupInfo — указатель на структуру STARTUPINFO, которая определяет параметры главного окна порожденного процесса;

• lpProcessInformation — указатель на структуру, которая будет заполнена информацией о порожденном процессе после возврата из функции.

Пример. Программа, запускающая Microsoft Word:

```
#include<windows.h>
#include <conio.h>
#include <stdio.h>
int main()
{
        PROCESS_INFORMATION pi;
        STARTUPINFO si;
        ZeroMemory(&si, sizeof(si));
        si.cb = sizeof(si);
        printf("Press any key to start WinWord --");
        getch();
        CreateProcess(NULL, "WinWord", NULL, NULL, FALSE, 0, NULL, NULL, &si, &pi);
        return 0;
}
```

Функция CreateThread() создает новый поток в адресном пространстве процесса. Формат функции [8]:

```
HANDLE CreateThread(LPSECURITY_ATTRIBUTES lpThreadAttributes,
                                        // атрибуты защиты потока
        DWORD dwStackSize,  // размер стека в байтах
        LPTHREAD_START_ROUTINE lpStartAddress, //указатель на функцию потока
        LPVOID lpParameter,             // аргумент, передаваемый в функцию потока
        BDWORD dwCreationFlags,         // флаги управления созданием потока
        LPDWORD lpThreadId );           // область памяти для возвращения
                                        //идентификатора потока
```

Функция возвращает описатель порожденного потока. Параметры:

• lpThreadAttributes — указатель на структуру, описывающую параметры защиты потока. Если параметру присвоено значение NULL, то устанавливаются атрибуты «по умолчанию»;

• dwStackSize — устанавливает размер стека, который отводится потоку. Если параметр равен нулю, то устанавливается стек, равный стеку первичного потока;

• lpStartAddress — адрес функции, которую будет исполнять поток. Функция имеет один 32-битный аргумент и возвращает 32-битное значение;

• lpParameter — параметр, передаваемый в функцию, которую будет исполнять поток;

• dwCreationFlags — дополнительный флаг, который управляет созданием потока. Если этот параметр равен CREATE_SUSPENDED, то поток после порождения не запускается на исполнение до вызова функции ResumeThread;

• lpThreadId — указатель на 32-битную переменную, которой будет присвоено значение уникального идентификатора потока.

Пример. Программа, порождающая поток:

```
#include<stdio.h>
#include <conio.h>
#include <windows.h>
DWORD WINAPI Output( LPVOID Param )
{
        while(TRUE)
        {
                printf("A");
                Sleep(100);
        }
        return(0);
}
int main()
{
        HANDLE hThread;
        DWORD ThreadId;
        hThread = CreateThread(NULL, 0, Output, NULL, 0, &ThreadId);
        getch();
        TerminateThread(hThread, 0);
        return(0);
}
```

Потоки, обладающие высоким приоритетом, занимают бóльшую часть времени центрального процессора, раньше завершают свою работу и способны быстрее реагировать на действия пользователя. Если всем потокам будет присвоен одинаково высокий приоритет, ничего хорошего не выйдет. Дело в том, что если нескольким потокам будет присвоен один и тот же приоритет (не имеет значения, высокий или низкий), то программа-планировщик выделит им одинаковое время работы центрального процессора и сама идея приоритетов утратит смысл. Один поток сможет быстрее реагировать на сигналы только в том случае, если будут замедлены другие потоки. Это же правило в равной степени применимо и к процессам. Старайтесь ограничивать приоритет всех потоков и процессов низким или средним уровнем и присваивайте им высокий приоритет только по мере необходимости [8, 12].

Приведенные далее функции проверяют или изменяют базовый приоритет потока:

```
BOOL SetThreadPriority(HANDLE hThread, int iPriority);      // дескриптор потока
int GetThreadPriority (HANDLE hThread);                     // новый уровень приоритета
```

Функция ***SetThreadPriority*** возвращает значение ***TRUE*** в случае успешного завершения потока, а значение ***FALSE*** — при возникновении ошибки. Функция ***GetThreadPriority*** возвращает значение, определяющее приоритет. Для обозначения возможных значений приоритета в обеих функциях используется набор констант:

THREAD_PRIORITY_LOWEST	На два уровня ниже приоритета процесса
THREAD_PRIORITY_BELOW_NORMAL	На один уровень ниже приоритета процесса
THREAD_PRIORITY_NORMAL	Тот же уровень приоритета, что и у процесса
THREAD_PRIORITY_ABOVE_NORMAL	На один уровень выше приоритета процесса
THREAD_PRIORITY_HIGHEST	На два уровня выше приоритета процесса
THREAD_PRIORITY_TIME_CRITICAL	Уровень 15 (для обычных пользовательских процессов)
THREAD_PRIORITY_IDLE	Уровень 1 (для обычных пользовательских процессов)

Прерванный поток приостанавливает свое выполнение и не учитывается при распределении времени центрального процессора. Поток остается в таком состоянии до тех пор, пока другой поток не возобновит его выполнение. Остановку потока можно произвести, в частности, в том случае, если пользователь прерывает выполнение определенной задачи. До тех пор, пока задание не будет отменено, поток можно перевести в состояние ожидания. Если пользователь решит продолжить работу, поток возобновит выполнение с той точки, где он был остановлен. Для приостановки и возобновления выполнения потоков служат функции:

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

Один и тот же поток можно последовательно остановить несколько раз, не возобновляя его выполнения, однако каждой последовательной команде ***SuspendThread*** должна соответствовать ответная команда ***ResumeThread***. Система отсчитывает количество отмененных команд с помощью счетчика прерываний. Каждая команда ***SuspendThread*** инкрементирует значения счетчика, а каждая команда ***ResumeThread*** декрементирует его. Обе функции возвращают предыдущее значение счетчика в виде параметра типа ***DWORD***. Поток возобновит свое выполнение только в том случае, если счетчик примет значение ***0***.

Поток способен остановить себя, но он не в состоянии самостоятельно возобновить свое выполнение. Однако он может на нужное время перенести себя в режим ожидания. Команда Sleep задерживает выполнение потока, удаляя его из очереди программы-планировщика, до тех пор, пока не пройдет заданный интервал времени. Интерактивные потоки, которые выводят определенную информацию для пользователя, часто делают короткие паузы, чтобы дать ему время для ознакомления с результатами. Применение режима ожидания предпочтительнее задействования «пустого» цикла, поскольку в этом случае не используется время центрального процессора.

Для осуществления паузы в течение заданного времени поток вызывает следующие функции:

```
VOID Sleep (DWORD dwMilliseconds);
DWORD SleepEx(DWORD dwMilliseconds,        // продолжительность паузы
BOOL bAlertable);                          // TRUE — возобновить работу
                                           // при завершении операции ввода / вывода
```

Расширенная функция SleepEx обычно работает совместно с функциями фонового ввода/вывода и может использоваться для инициации команд чтения или записи, не требуя их завершения. Эти операции выполняются в фоновом режиме. По завершении операции система извещает об этом пользователя, обращаясь к предусмотренной в программе процедуре обратного вызова. Фоновый ввод/вывод, или ***перекрывающийся ввод/вывод*** (далее будет рассмотрен более подробно), чаще всего применяется в интерактивных программах, которые должны реагировать на команды пользователя, не прерывая работы со сравнительно медленными устройствами, например с накопителем на магнитной ленте или с сетевым диском.

Параметр bAlertable функции SleepEx имеет тип Boolean. Если этот параметр равен TRUE, система может преждевременно возобновить выполнение потока при условии, что перекрывающаяся операция ввода/вывода завершилась до истечения заданного времени ожидания. В случае досрочного прерывания функция SleepEx возвращает значение WAIT_IO_COMPLETION по истечении указанного времени — значение 0.

По специальному запросу поток возвращает свои дескриптор и идентификатор. Указанные далее функции позволяют получить информацию о текущем потоке:

```
DWORD GetCurrentThreadId (VOID);
HANDLE GetCurrentThread (VOID);
```

Результирующее значение функции GetCurrentThreadId совпадает со значением параметра lpIDThread после выполнения функции CreateThread и однозначно идентифицирует поток в системе. Хотя идентификатор потока нужен лишь для незначительного количества функций Win32API, он может использоваться с целью мониторинга системных потоков без необходимости поддерживать открытый дескриптор для каждого потока. Открытый дескриптор защищает поток от уничтожения.

Дескриптор, полученный в результате выполнения функции GetCurrentThread, служит для тех же целей, что и дескриптор, возвращенный функцией CreateThread. Несмотря на то, что он может использоваться аналогично другим дескрипторам, на самом деле этот параметр является псевдодескриптором. Псевдодескриптор — это специальная константа, которая всегда интерпретируется системой особым образом, подобно тому, как одиночная точка (.) в DOS всегда указывает на текущий каталог, а параметр *this* в C++ определяет текущий объект. Константа-псевдодескриптор, полученная в результате выполнения функции GetCurrentThread, указывает на текущий поток. В отличие от настоящих дескрипторов псевдодескриптор не может передаваться другим потокам. Чтобы получить настоящий переносимый дескриптор потока, необходимо выполнить следующие действия:

```
HANDLE hThread;
hThread = DuplicateHandle(GetCurrentProcess(),     // процесс-источник
          GetCurrentThread(),                      // исходный дескриптор
          GetCurrentProcess(),                     // целевой процесс
          &hThread,                                // новый дублирующийся дескриптор
          0,                                       // привилегии доступа (подавляемые
                                                   // последним параметром)
          FALSE,                                   // дочерние объекты не унаследуют дескриптор
          DUPLICATE_SAME_ACCESS);                  // привилегии доступа копируются у исходного
                                                   //дескриптора
```

Хотя функция CloseHandle не влияет на псевдодескрипторы, дескриптор, созданный с помощью функции DuplicateHandle, является настоящим и в конце концов должен быть закрыт. Применение псевдодескрипторов значительно ускоряет работу функции GetCurrentThread, поскольку в этом случае подразумевается, что поток имеет полный доступ сам к себе и функция возвращает результат, не заботясь о мерах безопасности.

По аналогии с тем, как Windows-программа завершается по достижении конца функции WinMain, поток обычно прекращает свое существование при достижении конца функции, в которой был начат. Когда он достигает конца стартовой функции, система автоматически вызывает команду ExitThread, имеющую такой синтаксис:

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

Хотя операционная система вызывает функцию ExitThread автоматически, при необходимости досрочного завершения потока вы можете вызвать эту функцию явным образом:

```
DWORD ThreadFunction (LPDWORD lpdwParam)
{
HANDLE hThread = CreateThread (<параметры>);  // далее следуют стандартные операции инициализации;
                                              // проверка наличия ошибочных условий
if (<условие возникновения ошибки>)
{
        ExitThread (ERROR_CODE);              // прекратить работу потока
}
                                              // ошибки нет. работа продолжается
        return (SUCCESS_CODE);                // эта строка программы
                                              // заставляет систему вызвать функцию ExitThread
}
```

Параметры ERROR_CODEи SUCCESS_CODEопределяются по вашему усмотрению. В нашем простом примере поток нетрудно прервать с помощью команды *return*:

```
if (<условие возникновения ошибки>)
{
        return (ERROR_CODE);                  // прекратить работу потока
}
```

В данном случае команда *return* приводит к тому же результату, что и функция ExitThread, так как при ее выполнении осуществляется неявный вызов последней. Эта функция особенно полезна при необходимости прервать поток из любой подпрограммы, вызываемой внутри функции типа ThreadFunction.

Когда поток завершается с помощью оператора return, 32-разрядный код завершения автоматически передается функции ExitThread. После прекращения работы потока код его завершения может быть получен с помощью следующей функции:

```
BOOL GetExitCodeThread(HANDLE hThread, LPDWORD lpdwExitCode);
                  // один поток вызывает эту функцию для получения кода завершения другого потока
```

Функция GetExitCodeThread возвращает значение FALSE в том случае, если определить код завершения помешала ошибка.

Независимо от того, как вызывается функция ExitThread (явно или неявно в результате выполнения оператора return), она удаляет поток из очереди программы-планировщика и уничтожает его стек. Однако сам объект при этом сохраняется. Поэтому даже после прекращения выполнения

потока вы можете запросить его код завершения. По возможности дескриптор потока следует закрывать явно (с помощью функций CloseHandle), с тем чтобы поток не занимал лишний объем памяти. При закрытии последнего дескриптора система автоматически уничтожает поток. Система не может уничтожить выполняющийся поток, даже если закрыты все его дескрипторы. В этом случае поток будет уничтожен сразу же после завершения выполнения. Если по завершении процесса остаются незакрытые дескрипторы, система закрывает их автоматически и удаляет все «подвешенные» объекты, которые не принадлежат ни одному процессу.

С помощью команды ExitThread поток может остановить себя самостоятельно в том месте программы, где это необходимо. Кроме того, один поток способен по своему усмотрению мгновенно остановить другой поток:

```
BOOL TerminateThread (HANDLE hThread, DWORD dwExitCode);
                                    // с помощью вызова этой функции один поток может остановить другой
```

Поток не в состоянии защитить себя от прерывания. Имея соответствующий дескриптор, любой объект может мгновенно остановить поток вне зависимости от его текущего состояния (конечно, в том случае, если дескриптор разрешает полный доступ к потоку). Если при вызове функции CreateThread использовать набор атрибутов безопасности, заданный по умолчанию, то результирующий дескриптор обеспечит полные привилегии доступа к созданному потоку.

Функция TerminateThread не уничтожает стек потока, а только возвращает код его завершения. Функции ExitThread и TerminateThread переводят объект в сигнальное состояние, что служит признаком возможности запуска других потоков, ожидавших его завершения. После выполнения любой из двух указанных функций поток продолжает существовать в сигнальном состоянии до тех пор, пока не будут закрыты все его дескрипторы.

2.2.3. Синхронизация потоков

При работе с потоками необходимо иметь возможность координировать их действия. Часто координация действий подразумевает определенный порядок выполнения операций. Кроме функций, предназначенных для создания потоков и изменения их планового приоритета, Win32API содержит функции, которые переводят потоки в режим ожидания сигналов от определенных объектов, например от файлов или процессов. Кроме того, эти функции обеспечивают поддержку некоторых специальных объектов, в частности семафоров и исключающих семафоров.

Проиллюстрировать применение синхронизирующих объектов лучше всего на примере функций, ожидающих сигнала от объекта. С помощью одного набора обобщенных команд можно организовать ожидание сигналов от процессов, семафоров, исключающих семафоров, событий и некоторых других объектов. Следующая функция ожидает поступления сигнала от указанного объекта:

```
DWORD WaitForSingleObject (HANDLE hObject,          // объект, сигнал от которого ожидается
    DWORD dwMilliseconds);                          // максимальное время ожидания
```

Функция WaitForSingleObject позволяет приостановить выполнение потока до тех пор, пока не поступит сигнал от заданного объекта. Кроме того, в этой команде указывается максимальное время ожидания. Чтобы обеспечить бесконечное ожидание, в качестве временного интервала следует задать значение INFINITE. Если объект уже доступен или если он подает сигнал в течение заданного времени, функция WaitForSingleObject возвращает значение 0 и выполнение потока возобновляется. Но если заданный интервал времени прошел, а объект не подал сигнала, функция возвращает значение WAIT_TIMEOUT.

Для того чтобы заставить поток ожидать сигналы сразу от нескольких объектов, воспользуйтесь функцией WaitForMultipleObjects. Эта функция возвратит управление потоку при поступлении сигнала либо от одного из указанных объектов, либо от всех объектов вместе. В программе, управляемой событиями, должен быть задан массив объектов:

```
DWORD WaitForMultipleObjects(DWORD dwNumObjects,      // количество ожидаемых объектов
    LPHANDLE lpHandles,          // массив дескрипторов
    BOOL bWaitAll,               // TRUE — ожидание сигналов сразу от всех объектов;
                                 // FALSE — ожидание сигнала от любого из объектов
    DWORD dwMilliseconds);       // максимальный период ожидания
```

Результирующее значение WAIT_TIMEOUT опять-таки говорит о том, что заданный интервал времени прошел, а сигнал от объектов не поступил. Если флаг bWaitAll имеет значение FALSE, соответствующее ожиданию сигнала от любого из указанных объектов, в случае успешного завершения функция WaitForMultipleObjects возвращает код, который указывает, от какого из элементов массива lpHandles поступил сигнал (первый элемент массива соответствует значению 0, второй — значению 1 и т. д.). Если флаг bWaitAll имеет значение TRUE, функция не возвращает результата до тех пор, пока не будут установлены флаги всех объектов, т. е. пока не завершите выполнения всех потоков.

Две расширенные версии функций ожидания содержат дополнительный флаг статуса оповещения, который позволяет возобновить выполнение потока, если в течение периода ожидания были завершены асинхронные операции чтения или записи. Работу этих функций можно представить так, как будто они просят «разбудить» их в одном из трех случаев: если становится доступным указанный объект, если заканчивается заданный период времени, если завершилось выполнение фоновой операции ввода / вывода:

```
DWORD WaitForSingleObjectEx (HANDLE hObject,
                                        // объект, сигнал от которого ожидается
        DWORD dwMilliseconds,           // максимальное время ожидания
        BOOL bAlertable);               // TRUE— прекращение ожидания
                                        // при завершении операции ввода / вывода
DWORD WaitForMultipleObjectsEx(DWORD dwNumObjects,
                                        // количество ожидаемых объектов
        LPHANDLE lpHandles,             // массив дескрипторов
        BOOL bWaitAll,                  // TRUE — ожидание сигналов сразу от всех объектов;
                                        // FALSE — ожидание сигнала от любого из объектов
        DWORD dwMilliseconds,           // максимальный период ожидания
        BOOL bAlertable);               // TRUE — прекращение ожидания
                                        // при завершении операции ввода / вывода
```

При успешном выполнении функции ожидания объект, сигнал от которого ожидался, обычно определенным образом изменяется. Например, если поток ожидал и получил сигнал от исключающего семафора, функция восстанавливает несигнальное состояние исключающего семафора, чтобы остальные потоки знали о том, что он занят. Кроме того, функции ожидания декрементируют значение счетчика семафора и сбрасывают информацию о некоторых событиях.

Функции ожидания не изменяют состояния указанного объекта до тех пор, пока не поступит сигнал от одного или нескольких других объектов. В частности, поток не захватывает исключающий семафор сразу же после поступления сигнала от него, а ожидает сигналов от других объектов. Кроме того, в течение времени ожидания исключающий семафор снова может быть захвачен другим потоком, который еще больше продлит состояние ожидания.

Конечно, ожидать поступления сигнала от объекта можно лишь в том случае, если этот объект уже создан. Начнем с создания ***исключающих семафоров*** и ***семафоров***, поскольку для работы с ними существуют параллельные API-команды, позволяющие создавать и уничтожать эти объекты, захватывать и освобождать их, а также получать их дескрипторы.

Функциям, создающим ***исключающие семафоры*** и ***семафоры***, нужно указать требуемые привилегии доступа и начальные параметры создаваемого объекта (можно также указать его имя, но это необязательно) [12]:

```
HANDLE CreateMutex (LPSECURITY_ATTRIBUTES lpsa,  // необязательные атрибуты безопасности
        BOOL bInitialOwner,                      // TRUE — создатель хочет
                                                 // завладеть полученным объектом
        LPTSTR lpszMutexName )                   // имя объекта

HANDLE CreateSemaphore(
        LPSECURITY_ATTRIBUTES lpsa,              // необязательные атрибуты безопасности
        LONG lInitialCount,                      // исходное значение счетчика (обычно 0)
        LONG lMaxCount,                          // максимальное значение
                                                 // счетчика (ограничивает число потоков)
        LPTSTR lpszSemName);                     // имя семафора (может иметь значение NULL)
```

Если в качестве атрибута безопасности задано значение NULL, то результирующий дескриптор получит все привилегии доступа и не будет наследоваться дочерними процессами. Имена объектов являются необязательными, однако они становятся полезными в ситуации, когда несколько процессов управляют одним и тем же объектом.

Если флагу bInitialOwner присвоить значение TRUE, то поток сразу после создания объекта завладеет им. Созданный исключающий семафор не станет подавать сигналов до тех пор, пока поток не освободит его.

В отличие от исключающего семафора, который может принадлежать только одному потоку, неисключающий семафор остается в сигнальном состоянии до тех пор, пока его счетчик захватов не получит значения iMaxCount. Если другие потоки в этот момент попытаются завладеть семафором, то они будут приостановлены до тех пор, пока счетчик захватов не будет декрементирован до значения ниже максимального.

Пока семафор (или исключающий семафор) существует, поток взаимодействует с ним посредством операций захвата и освобождения. Для захвата любого объекта поток вызывает функцию WaitForSingleObject (или одну из ее разновидностей). Завершив выполнение задачи, которая синхронизировалась захваченным объектом, поток освобождает этот объект с помощью одной из следующих функций:

```
BOOL ReleaseMutex(HANDLE hMutex);

BOOL ReleaseSemaphore(HANDLE hSemaphore,
    LONG lRelease,              // величина, на которую
                                // инкрементируется значение счетчика
                                // при освобождении объекта (обычно 1)
    LPLONG lplPrevious);        // переменная, которой присваивается
                                // предыдущее значение счетчика
```

При освобождении семафора или исключающего семафора значение счетчика захватов инкрементируется. Значение счетчика, превышающее 0, воспринимается системой как сигнал объекта ожидающим его потокам.

Освободить исключающий семафор может только тот поток, который завладел им. Однако любой поток может вызвать функцию ReleaseSemaphore, которая инкрементирует значение счетчика захватов обычного семафора вплоть до его максимального значения. Изменение значения счетчика дает возможность в процессе выполнения программы задать произвольным образом количество потоков, которые могут завладеть семафором. Обратите внимание, что функция CreateSemaphore позволяет при создании нового семафора присвоить его счетчику значение меньше максимального. Например, при разработке нового семафора его счетчику можно задать начальное значение 0. Такой прием позволит заблокировать все потоки до тех пор, пока программа не произведет инициализацию, а затем не увеличит значение счетчика с помощью команды ReleaseSemaphore.

Не забывайте вовремя освобождать синхронизирующие объекты. Не задав максимального времени ожидания и забыв освободить исключающий семафор, вы заблокируете все ожидающие его потоки.

Поток может ожидать несколько сигналов от одного и того же объекта, не будучи заблокированным, однако после завершения каждого из процессов ожидания необходимо выполнять операцию освобождения. Это требование справедливо для семафоров, исключающих семафоров и критических разделов.

Событие представляет собой объект, который создается программой при необходимости информировать потоки о выполнении определенных действий. В простейшем случае (ручной сброс) событие переключает свое состояние с помощью команд SetEvent (сигнал включен) и ResetEvent (сигнал выключен). Когда сигнал включается, его получают все потоки, которые ожидают появления соответствующего события. Если сигнал выключается, все такие потоки блокируются. В отличие от семафоров и исключающих семафоров события данного типа изменяют свое состояние только при подаче соответствующей команды каким-нибудь потоком.

События целесообразно использовать при условии, что поток должен выполняться только после того, как программа обновит свое окно или пользователь введет определенную информацию [12]. Далее представлены основные функции, предназначенные для работы с событиями:

```
HANDLE CreateEvent(LPSECURITY_ATTRIBUTES lpsa,
                                    // привилегии доступа(по умолчанию = NULL)
        BOOL bManualReset,          // TRUE — событие должно быть сброшено вручную
        BOOL bInitialState,         // TRUE — создание события в сигнальном состоянии
        LPTSTR lpszEventName);      //имя события (допускается значение NULL)
BOOL SetEvent (HANDLE hEvent);
BOOL ResetEvent (HANDLE hEvent);
```

При установке параметра bInitialState функция CreateEvent создает событие, которое сразу же будет находиться в сигнальном состоянии. Функции SetEvent и ResetEvent в случае успешного завершения возвращают значение TRUE, при возникновении ошибки — значение FALSE.

Параметр bManualReset функции CreateEvent позволяет создать событие, сбрасываемое не вручную, а автоматически. Автоматически сбрасываемое событие переходит в несигнальное состояние сразу же после выполнения функции SetEvent. Для таких событий функция ResetEvent является избыточной. Кроме того, перед автоматическим сбросом по каждому сигналу событие освобождает только один поток. Автоматически сбрасываемые события целесообразно применять в таких программах, где один основной поток подготавливает данные для других, вспомогательных потоков. При готовности нового набора данных основной поток устанавливает событие, по которому освобождается один вспомогательный поток. Остальные вспомогательные потоки продолжают ожидать подготовки новых данных.

Наряду с выполнением операций установки и сброса события можно сгенерировать импульсное событие:

```
BOOL PulseEvent (hEvent);
```

Импульсное событие включает сигнал на короткий промежуток времени. Применение этой функции для события, сбрасываемого вручную, позволяет оповестить о нем все ожидающие потоки, а затем сбросить событие. Вызов функции для события, сбрасываемого автоматически, дает возможность оповестить только один ожидающий поток. Если не было ни одного ожидающего потока, то никакой другой поток не будет оповещен. С другой стороны, установка автоматического события позволит оставить сигнал включенным до тех пор, пока не появится ожидающий его поток. После оповещения потока событие сбрасывается автоматически.

Семафоры, исключающие семафоры и события могут совместно использоваться несколькими процессами, которые необязательно должны быть связаны друг с другом. Путем совместного задействования синхронизирующих объектов процессы могут координировать свои действия по аналогии с тем, как это делают потоки. Существуют три механизма совместного использования. Первый из них — это наследование, при котором один процесс создает новый процесс, получающий копии всех дескрипторов родительского процесса. При этом копируются только те дескрипторы, которые при создании были помечены как доступные для наследования.

Два других метода сводятся к созданию второго дескриптора существующего объекта с помощью вызова функций. Какая из функций будет вызвана, зависит от имеющейся информации. При наличии дескрипторов как исходного процесса, так и процесса назначения следует вызывать функцию DuplicateHandle при наличии только имени объекта — одну из функций Openxxx. Две программы могут заранее определить имя совместно используемого объекта. Кроме того, одна из программ способна передать другой это имя посредством совместно используемой области памяти функций DDEML (DDE Management Library — библиотека управления динамическим обменом данных) или канала:

```
BOOL DuplicateHandle(HANDLE hSourceProcess,   // процесс, которому принадлежит исходный объект
        HANDLE hSource,                       // дескриптор исходного объекта
        HANDLE hTargetProcess,                // процесс, который хочет создать копию дескриптора
        LPHANDLE lphTarget,                   // переменная для записи копии дескриптора
        DWORD fdwAccess,                      // запрашиваемые привилегии доступа
```

```
        BOOL bInherit,                    // может ли наследоваться копия дескриптора?
        DWORD fdwOptions);                // дополнительные операции, например
                                          // закрытие исходного дескриптора
HANDLE OpenMutex(DWORD fdwAccess,         // запрашиваемые привилегии доступа
        BOOL bInherit,                    // TRUE — дочерний процесс может
                                          // наследовать этот дескриптор
        LPTSTR lpszName);                 // имя исключающего семафора

HANDLE OpenSemaphore(DWORD fdwAccess,     // запрашиваемые привилегии доступа
        BOOL bInherit,                          // TRUE — дочерний процесс может
                                                // наследовать этот дескриптор
        LPTSTR lpszName);                       // имя семафора

HANDLE OpenEvent(DWORD fdwAccess,         // запрашиваемые привилегии доступа
        BOOL bInherit,                          // TRUE— дочерний процесс может
                                                // наследовать этот дескриптор
        LPTSTR lpszName);                       // имя события
```

Используемый в этом примере тип данных LPTSTR — это обобщенный текстовый тип, который компилируется по-разному в зависимости от того, какой стандарт — Unicode или ASCII— поддерживается приложением.

Семафоры, исключающие семафоры и объекты событий будут сохраняться в памяти до тех пор, пока не завершатся все использующие их процессы или пока с помощью функции CloseHandle не будут закрыты все дескрипторы соответствующего объекта:

```
BOOL CloseHandle(hObject);
```

Критический раздел представляет собой объект, выполняющий те же функции, что и исключающий семафор, но в отличие от последнего критический раздел не может наследоваться. Оба объекта доступны только для одного процесса. Преимущество критических разделов перед исключающими семафорами состоит в том, что они проще в управлении и гораздо быстрее работают [12].

Терминология, принятая для функций, которые используются при работе с критическими разделами, отличается от терминологии, разработанной для функций управления семафорами, исключающими семафорами и событиями, однако сами функции выполняют одни и те же операции. В частности, принято говорить не о создании критического раздела, а о его инициализации. Процесс не ожидает критического раздела, а входит в него, и не освобождает критический раздел, а покидает его. К тому же вы не закрываете дескриптор, а удаляете объект:

```
VOID InitializeCriticalSection (LPCRITICAL_SECTION lpcs);
VOID EnterCriticalSection(LPCRITICAL_SECTION lpcs);
VOID LeaveCriticalSection(LPCRITICAL_SECTION lpcs);
VOID DeleteCriticalSection(LPCRITICAL_SECTION lpcs);
```

Переменная типа LPCRITICAL_SECTION содержит указатель (а не дескриптор) критического раздела. Функция InitializeCriticalSection должна получить указатель на пустой объект (&cs), который можно создать следующим образом:

```
CRITICAL_SECTION cs;
```

2.2.4. Использование классов MFC для создания потоков

Способ создания потоков с помощью функций библиотеки MFC заключается в создании класса, порожденного от класса ***CWinThread***. Схема этого процесса выглядит следующим образом:

```
IMPLEMENT_DYNCREATE(CThreadExample, CWinThread)      // Класс CThreadExample

CThreadExample::CThreadExample()
{
...                                                  // инициализация переменных-членов класса
}
```

```
CThreadExample::~CThreadExample()
{
}

BOOL CThreadExample::InitInstance()
{
                                        // TODO: здесь следует выполнить инициализацию потока
    ...
                                        //здесь должны выполняться операции инициализации,
                                        //не связанные с переменными, например создание
                                        // экземпляров других объектов класса
    return TRUE;
}
int CThreadExample::ExitInstance()
{
                                        // TODO: здесь выполняются все операции
                                        // очистки для потока
...
    return CWinThread::ExitInstance();
}

BEGIN_MESSAGE_MAP(CThreadExample, CWinThread)

                                        //{{AFX_MSG_MAP(CThreadExample)
                                        // ПРИМЕЧАНИЕ — Мастер ClassWizard будет добавлять /
                                        // удалять в этом месте макросы обработки сообщений
                                        // }}AFX_MSG_MAP
END_MESSAGE_MAP()
```

Объект класса CWinThread представляет поток выполнения в рамках приложения. Хотя основной поток выполнения приложения обычно задается объектом класса, порожденного от CWinApp, сам класс CWinApp является производным от класса CWinThread.

Для обеспечения безопасности потоков в MFC-приложениях должны применяться классы, являющиеся производными от класса CWinThread. Библиотека MFC использует переменные — члены этого класса. Потоки, созданные с помощью функции_beqinthreadex, не могут использовать ни одной из API-функций библиотеки MFC.

Поддерживаются два основных типа потоков, а именно ***рабочие*** и ***интерфейсные***. Для рабочих потоков не нужно создавать цикл обработки сообщений. Такие потоки могут выполнять фоновые вычисления в электронной таблице без взаимодействия с пользователем и не должны реагировать на сообщения.

В отличие от рабочих интерфейсные потоки обрабатывают сообщения, полученные от системы (или от пользователя). Для них необходима специальная процедура обработки сообщений. Интерфейсные потоки создаются на базе класса CWinApp или непосредственно класса CWinThread.

Объект класса CWinThread обычно существует в течение всего времени существования потока, однако такой способ функционирования можно изменить, присвоив переменной-члену m_bAutoDelete значение ***FALSE***.

Потоки создаются с помощью функции AfxBeqinThread. Для создания интерфейсного потока функции AfxBeqinThread следует передать указатель на класс CRuntimeClass объекта, производного от класса CWinThread. В случае рабочих потоков функция AfxBeqinThread вызывается с указанием управляющей функции и параметра, передаваемого последней.

Как для рабочих, так и для интерфейсных потоков можно указать дополнительные параметры, изменяющие приоритет, размер стека, флаги создания и атрибуты безопасности потока. Функция AfxBeqinThread возвращает указатель на новый объект класса CWinThread.

В качестве альтернативного варианта можно определить, а затем создать объект, производный от класса CWinThread,вызвав функцию CreateThread данного класса. В этом случае производный объект может многократно использоваться при последовательных созданиях и уничтожениях потока.

Глава 3
Принципы разработки программного кода для обработки прерываний и исключений

§ 3.1. Система обработки прерываний

Обладая способностью переключаться от одного потока выполнения к другому, ядро операционной системы должно к тому же реагировать на ***прерывания*** (*interrupts*) и ***исключения*** (*exceptions*). Речь идет о сигналах, которые возникают в системе и заставляют процессор прерывать свою работу и переключаться на обработку возникшей ситуации. Рассмотрим эти два механизма более подробно.

Прерывания представляют собой ***механизм***, позволяющий координировать параллельное функционирование отдельных устройств вычислительной системы и реагировать на особые состояния, возникающие при работе процессора. Таким образом, ***прерывание — это принудительная передача управления от выполняемой программы к системе*** (а через нее — к соответствующей программе обработки прерывания), происходящая при возникновении определенного события.

Идея прерываний была предложена в середине 50-х гг.Можно без преувеличения сказать, что она внесла наиболее весомый вклад в развитие вычислительной техники. Основная цель введения прерываний — реализация асинхронного режима работы и распараллеливание работы отдельных устройств вычислительного комплекса.

Механизм прерываний реализуется аппаратно-программными средствами. Первоначально рассмотрим в общих чертах аппаратную часть системы прерываний. Сигналы аппаратных прерываний, возникающие в устройствах, входящих в состав компьютера или подключенных к нему, поступают в процессор не непосредственно, а через два контроллера прерываний, один из которых называется ведущим, а второй ведомым (рис. 3.1). В прежних моделях машин контроллеры представляли собой отдельные микросхемы; в современных компьютерах они входят в состав многофункциональной микросхемы периферийного контроллера.

Рис.3.1. Аппаратная часть системы прерываний

Два контроллера используются для увеличения допустимого числа внешних устройств. Дело в том, что каждый контроллер прерываний может обслуживать сигналы лишь от восьми устройств. Для обслуживания большего количества устройств контроллеры можно объединять, образуя из них веерообразную структуру. В современных машинах устанавливают два контроллера, увеличивая тем самым возможное число входных устройств до 15 (7 — у ведущего, 8 — у ведомого контроллера).

К входным выводам IRQ1...IRQ7 и IRQ8...IRQ15 (IRQ — это сокращение от Interrupt Request, т. е. запрос прерывания) подключаются выводы устройств, на которых возникают сигналы прерываний. Выход ведущего контроллера подключается к входу INT микропроцессора, а выход ведомого — к входу IRQ2 ведущего. Основная функция контроллеров — передача сигналов запросов прерываний от внешних устройств на единственный вход прерываний процессора. При этом кроме сигнала INT контроллеры передают в процессор по линиям данных номер вектора, который

образуется в контроллере путем сложения базового номера, записанного в одном из его регистров, с номером входной линии, по которой поступил запрос прерывания. Номера базовых векторов заносятся в контроллеры автоматически в процессе начальной загрузки компьютера. Очевидно, что номера векторов аппаратных прерываний однозначно связаны с номерами линий, или уровнями IRQ, а через них — с конкретными устройствами компьютера. На рис. 3.1 обозначены основные устройства компьютера, работающие в режиме прерываний.

Процессор, получив по линии INT сигнал прерывания, выполняет последовательность стандартных действий, которую можно назвать процедурой прерывания. Смысл процедуры прерывания заключается в том, чтобы сохранить состояние текущей (прерываемой) программы и передать управление обработчику, соответствующему возникшему прерыванию.

Структуры систем прерывания (в зависимости от аппаратной архитектуры) могут быть самыми разными, но все они имеют одну общую особенность: прерывание непременно влечет за собой изменение порядка выполнения команд процессором. Механизм обработки прерываний независимо от архитектуры вычислительной системы включает следующие элементы [2]:

1. Установление факта прерывания (прием сигнала на прерывание) и идентификация прерывания (в операционных системах иногда осуществляется повторно, на шаге 4).

2. Запоминание состояния прерванного процесса. Состояние процесса определяется прежде всего значением счетчика команд (адресом следующей команды, который, например, в i80×86 определяется регистрами CS и IP — указателем команды), содержимым регистров процессора и может включать также спецификацию режима (например, режим пользовательский или привилегированный) и другую информацию.

3. Управление аппаратно передается подпрограмме обработки прерывания. В простейшем случаев счетчик команд заносится начальный адрес подпрограммы обработки прерываний, а в соответствующие регистры — информация из слова состояния. В более развитых процессорах, например в том же i80286 и последующих 32-битовых микропроцессорах начиная с i80386, осуществляется достаточно сложная процедура определения начального адреса соответствующей подпрограммы обработки прерывания и не менее сложная процедура инициализации рабочих регистров процессора.

4. Сохранение информации о прерванной программе, которую не удалось спасти на шаге 2 с помощью действий аппаратуры. В некоторых вычислительных системах предусматривается запоминание довольно большого объема информации о состоянии прерванного процесса.

5. Обработка прерывания. Эта работа может быть выполнена той же подпрограммой, которой было передано управление на шаге 3, но в ОС чаще всего она реализуется путем последующего вызова соответствующей подпрограммы.

6. Восстановление информации, относящейся к прерванному процессу (этап, обратный шагу 4).

7. Возврат в прерванную программу.

Шаги 1–3 реализуются аппаратно, шаги 4–7 — программно. На рис. 3.2 показано, что при возникновении запроса на прерывание естественный ход вычислений нарушается и управление передается программе обработки возникшего прерывания. При этом сохраняется адрес той команды, с которой следует продолжить выполнение прерванной программы. После выполнения программы обработки прерывания управление возвращается прерванной ранее программе посредством занесения в указатель команд сохраненного адреса команды. Такая схема используется только в самых простых программных средах. В мультипрограммных операционных системах обработка прерываний происходит по более сложным схемам, о чем будет рассказано ниже. Итак, к ***главным функциям механизма прерываний*** относятся:

- распознавание, или классификация, прерываний;
- передача управления соответственно обработчику прерываний;
- корректное возвращение к прерванной программе.

Переход от прерываемой программы к обработчику и обратно должен выполняться как можно быстрей. Одним из быстрых методов является использование таблицы, содержащей перечень всех допустимых для компьютера прерываний и адреса соответствующих обработчиков. Для корректного возвращения к прерванной программе перед передачей управления обработчику прерываний содержимое регистров процессора запоминается либо в памяти с прямым доступом, либо в системном стеке (*systems tack*).

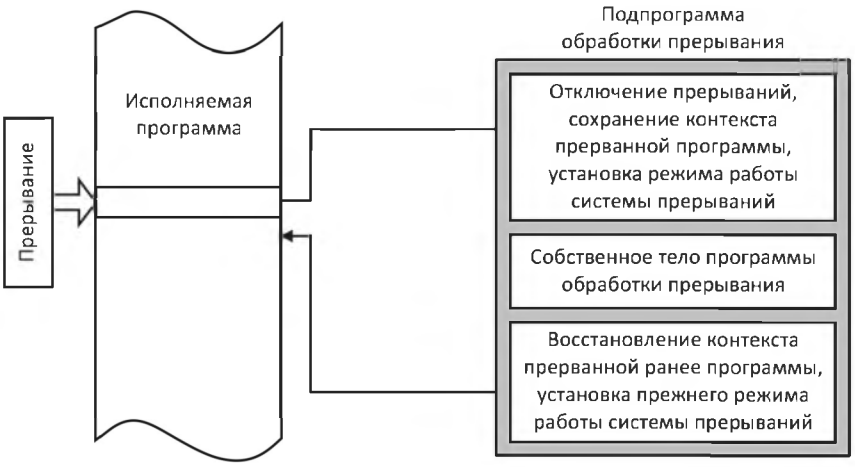

Рис.3.2. Обработка прерывания

Прерывания, возникающие при работе вычислительной системы, можно разделить на два основных класса: внешние (их иногда называют асинхронными) и внутренние (синхронные).

Внешние прерывания вызываются асинхронными событиями, которые происходят вне прерываемого процесса, например:

– прерывания от таймера;

– прерывания от внешних устройств (прерывания по вводу / выводу);

– прерывания по нарушению питания;

– прерывания с пульта оператора вычислительной системы;

– прерывания от другого процессора или другой вычислительной системы.

Внутренние прерывания вызываются событиями, которые связаны с работой процессора и синхронны с его операциями. Примерами являются следующие запросы на прерывания:

– при нарушении адресации (в адресной части выполняемой команды указан запрещенный или несуществующий адрес, обращение к отсутствующему сегменту или странице при организации механизмов виртуальной памяти);

– при наличии в поле кода операции незадействованной двоичной комбинации;

– при делении на нуль;

– при переполнении или исчезновении порядка;

– при обнаружении ошибок четности, ошибок в работе различных устройств аппаратуры средствами контроля.

Кроме того, могут существовать прерывания при обращении к супервизору ОС: в некоторых компьютерах часть команд может использовать только ОС, а не пользователи. Соответственно в аппаратуре предусмотрены различные режимы работы, и пользовательские программы выполняются в режиме, в котором эти привилегированные команды не исполняются. При попытке использовать команду, запрещенную в данном режиме, происходит внутреннее прерывание и управление передается супервизору ОС. К привилегированным командам относятся и команды переключения режима работа центрального процессора.

Наконец, существуют собственно ***программные прерывания***. Эти прерывания происходят по соответствующей команде прерывания, т. е. по этой команде процессор осуществляет практически те же действия, что и при обычных внутренних прерываниях. Данный механизм был специально введен для того, чтобы переключение на системные программные модули осуществлялось не просто как переход в подпрограмму, а точно таким же образом, как и обычное прерывание. Этим обеспечивается автоматическое переключение процессора в привилегированный режим с возможностью исполнения любых команд.

Сигналы, вызывающие прерывания, формируются вне процессора или в самом процессоре, и они могут возникать одновременно. Выбор одного из них для обработки осуществляется на основе приоритетов, приписанных каждому типу прерывания. Очевидно, что прерывания от схем контроля процессора должны обладать наивысшим приоритетом(если аппаратура работает неправильно, то не имеет смысла продолжать обработку информации). На рис. 3.3 изображен обычный порядок (приоритеты) обработки прерываний в зависимости от типа прерываний [2].

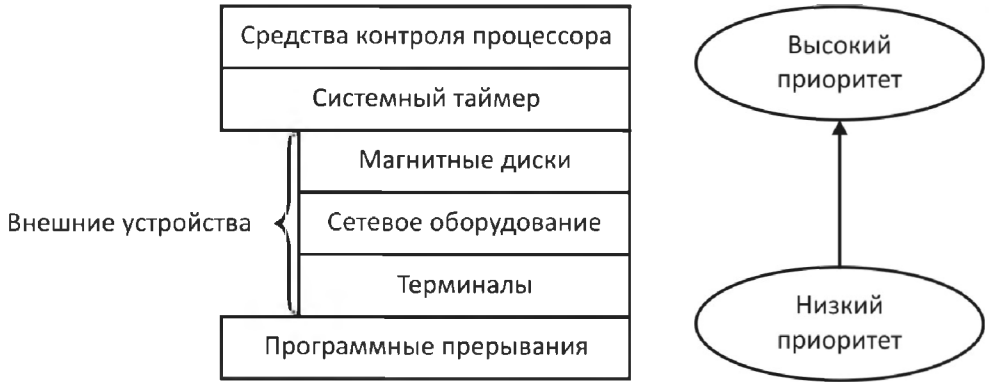

Рис.3.3. Распределение прерываний по уровням приоритета

Учет приоритета может быть встроен в технические средства, а также определяться операционной системой, т.е. кроме аппаратно-реализованных приоритетов прерывания большинство вычислительных машин и комплексов допускают программно-аппаратное управление порядком обработки сигналов прерывания. Второй способ, дополняя первый, позволяет применять различные ***дисциплины обслуживания прерываний.***

Наличие сигнала прерывания не обязательно должно вызывать прерывание исполняющейся программы. Процессор может обладать средствами защиты от прерываний, к которым относятся отключение системы прерываний и маскирование (запрет) отдельных сигналов прерывания. Программное управление этими средствами (существуют специальные команды для управления работой системы прерываний) позволяет операционной системе регулировать обработку сигналов прерывания, заставляя процессор обрабатывать их сразу по приходу, откладывать их обработку на некоторое время или полностью игнорировать. Обычно операция прерывания выполняется только после завершения выполнения текущей команды. Поскольку сигналы прерывания возникают в произвольные моменты времени, то на момент прерывания может существовать несколько сигналов прерывания, которые обрабатываются только последовательно. Чтобы обработать сигналы прерывания в разумном порядке, им присваиваются приоритеты. Сигнал с более высоким приоритетом обрабатывается в первую очередь, обработка остальных сигналов прерывания откладывается.

Программное управление специальными регистрами маски (маскирование сигналов прерывания) позволяет реализовать различные дисциплины обслуживания [2]:

– с ***относительными приоритетами***, т.е. обслуживание не прерывается даже при наличии запросов с более высокими приоритетами. После окончания обслуживания данного запроса обслуживается запрос с наивысшим приоритетом. Для организации такой дисциплины необходимо в программе обслуживания данного запроса наложить маски на все остальные сигналы прерывания или просто отключить систему прерываний;

– с ***абсолютными приоритетами***, т.е. всегда обслуживается прерывание с наивысшим приоритетом. Для реализации этого режима необходимо на время обработки прерывания замаскировать все запросы с более низким приоритетом. При этом возможно многоуровневое прерывание, т.е. прерывание программ обработки прерываний. Число уровней прерывания в этом режиме изменяется и зависит от приоритета запроса;

– по ***принципу стека***, или, как иногда говорят, ***по дисциплине LCFS*** (*last come first served* — последним пришел, первым обслужен), т.е. запросы с более низким приоритетом могут прерывать обработку прерывания с более высоким приоритетом. Для этого необходимо не накладывать маски ни на один сигнал прерывания и не выключать систему прерываний.

Следует особо отметить, что для правильной реализации последних двух дисциплин нужно обеспечить полное маскирование системы прерываний при выполнении шагов 1–4 и 6–7. Это необходимо для того, чтобы не потерять запрос и правильно его обслужить. Многоуровневое прерывание должно происходить на этапе собственно обработки прерывания, а не на этапе перехода с одного процесса на другой.

Управление ходом выполнения задач со стороны ОС заключается в организации реакций на прерывания, организации обмена информацией (данными и программами), предоставлении необходимых ресурсов, динамике выполнения задачи, организации сервиса. Причины прерываний определяет ОС (модуль, который называют супервизором прерываний). Она же выполняет действия, необходимые при данном прерывании и в данной ситуации. Поэтому в состав любой ОС реального времени прежде всего входят программы управления системой прерываний, контроля состояний задач и событий, синхронизации задач, средства распределения памяти и управления ею, а уже потом средства организации данных (с помощью файловых систем и т. д.). Следует, однако, заметить, что современная ОС реального времени должна вносить в аппаратно-программный комплекс нечто большее, нежели просто обеспечение быстрой реакции на прерывания.

Как мы уже знаем, при появлении запроса на прерывание система прерываний идентифицирует сигнал, и если прерывания разрешены, управление передается на соответствующую подпрограмму обработки. На рис. 3.2 видно, что в подпрограмме обработки прерывания имеются ***две служебные секции***: первая секция, где осуществляется сохранение контекста прерванной задачи, который не смог быть сохранен на 2-м шаге, и последняя, заключительная секция, в которой, наоборот, осуществляется восстановление контекста. Для того чтобы система прерываний не среагировала повторно на сигнал запроса на прерывание, она обычно автоматически «закрывает» (отключает) прерывания, поэтому необходимо потом в подпрограмме обработки прерываний вновь включать систему прерываний. Установка рассмотренных режимов обработки прерываний (с относительными и абсолютными приоритетами и по правилу LCFS) осуществляется в конце первой секции подпрограммы обработки. Таким образом, на время выполнения центральной секции (в случае работы в режимах с абсолютными приоритетами и по дисциплине LCFS) прерывания разрешены. На время работы заключительной секции подпрограммы обработки система прерываний должна быть отключена и после восстановления контекста вновь включена. Поскольку эти действия необходимо выполнять практически в каждой подпрограмме обработки прерываний, во многих операционных системах первые секции подпрограмм обработки прерываний выделяются в специальный системный программный модуль, называемый ***супервизором прерываний***.

Супервизор прерываний прежде всего сохраняет в дескрипторе текущей задачи рабочие регистры процессора, определяющие контекст прерываемого вычислительного процесса. Далее он определяет ту подпрограмму, которая должна выполнить действия, связанные с обслуживанием настоящего (текущего) запроса на прерывание. Наконец, перед тем как передать управление этой подпрограмме, супервизор прерываний устанавливает необходимый режим обработки прерывания. После выполнения подпрограммы обработки прерывания управление вновь передается супервизору, на этот раз уже на тот модуль, который занимается диспетчеризацией задач. И уже диспетчер задач, в свою очередь, в соответствии с принятым режимом распределения процессорного времени (между выполняющимися процессами) восстановит контекст той задачи, которой будет решено выделить процессор. Рассмотренная схема представлена на рис. 3.4 [2], из которого следует: нет непосредственного возврата в прерванную ранее программу из самой подпрограммы обработки прерывания. Для прямого возврата достаточно сохранить в стеке адрес возврата, что и делает аппаратура процессора. При этом стек легко обеспечивает возможность возврата в случае вложенных прерываний, поскольку он всегда реализует дисциплину LCFS (*last come — first served*).

Однако если бы контекст процессов сохранялся просто в стеке, как это обычно реализуется аппаратурой, а не в дескрипторах задач, то не было бы возможности гибко подходить к выбору той задачи, которую нужно передать процессору после завершения работы подпрограммы обработки прерывания.

Рассмотрим, как производится обработка прерываний в ОС Windows [4]. В этой операционной системе каждому прерыванию назначается определенный приоритет, называемый ***уровнем запроса прерывания*** (*interrupt request level*, IRQL). IRQL назначается источнику прерывания. Например, мышь имеет IRQL, который присваивается поступающим от нее сигналам. Системный таймер также генерирует собственные прерывания, которым назначается другой IRQL.

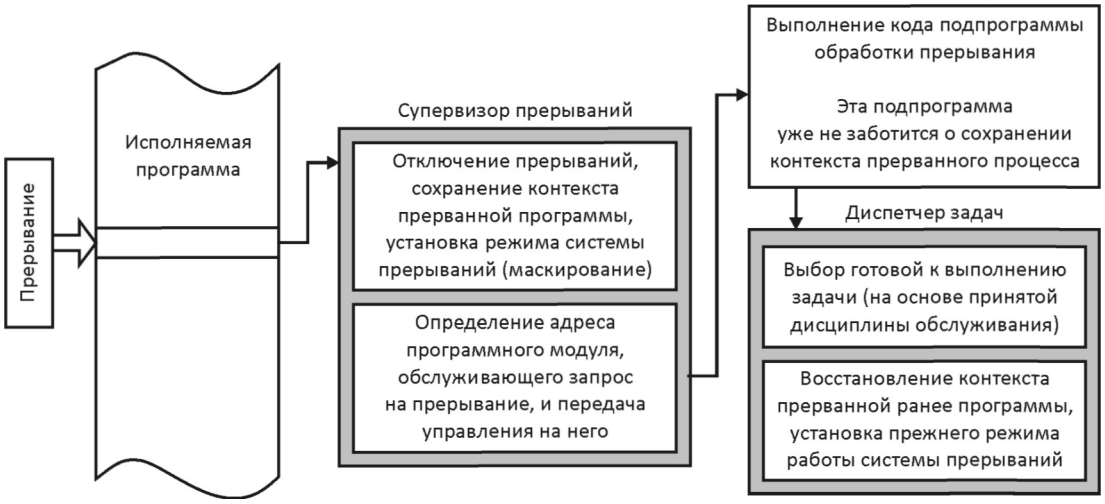

Рис. 3.4. Обработка прерывания при участии супервизоров ОС

Центральный процессор также имеет свой IRQL, который изменяется по мере работы системы. Изменение IRQL центрального процессора позволяет системе блокировать прерывания с более низким приоритетом. Приоритет процессора могут изменить только системные сервисы, работающие на уровне ядра, например перехватчик. Потоки, функционирующие на пользовательском уровне, не имеют таких привилегий. Заблокированным прерываниям внимание не уделяется до тех пор, пока какой-нибудь поток явным образом не понизит уровень прерывания центрального процессора. Если процессор работает при самом низком значении IRQL, происходит нормальное выполнение потока и разрешается обработка всех прерываний. Когда перехватчик вызывает ***процедуру обслуживания прерывания*** (*interrupt service routine*, ISR), он сначала назначает центральному процессору тот же IRQL, который имеет перехваченный сигнал. На время выполнения ISR прерывания более низкого уровня, например сигналы от устройств ввода данных, блокируются, чтобы не отвлекать систему от выполнения критических операций, таких как процедуры восстановления при отключении питания. При понижении IRQL процессора заблокированные прерывания разблокируются и начинают поочередно обрабатываться обычным образом. Время от времени центральный процессор переключается на самый низкий IRQL, чтобы возобновить выполнение прерванных потоков.

Прежде чем приступить к обработке прерывания, перехватчик должен найти в системе соответствующую подпрограмму. Он ищет такие подпрограммы в ***таблице распределения прерываний*** (*interrupt dispatch table*, IDT). Указанная таблица содержит 32 записи, по одной для каждого уровня запроса прерывания. Каждая из таких записей указывает на подпрограмму обработки прерывания или на последовательную цепочку таких подпрограмм в том случае, если несколько устройств используют один и тот же IRQL.

При загрузке в систему новый драйвер устройства записывает в IDT собственный обработчик. Драйверы делают это путем создания и подключения ***объекта-прерывания***, т. е. структуры, содержащей всю информацию, необходимую ядру для дополнения IDT. С помощью объекта-прерывания драйверы получают возможность зарегистрировать свои обработчики прерываний, ничего не зная ни об аппаратном устройстве, ни о структуре таблицы распределения прерываний.

§ 3.2. Общие принципы обработки исключений

По мере выполнения программы ее нормальная работа по различным причинам может нарушаться. В частности, это может быть связано с тем, что центральный процессор наталкивается на недопустимый адрес памяти, пользователь прерывает выполнение программы комбинацией «горячих» клавиш [Ctrl + C], отладчик произвольным образом останавливает программу и запускает ее вновь. Ошибка может быть связана и с вводом неправильного значения при выполнении операции с плавающей запятой. Эти и другие исключительные ситуации способны возникать как на поль-

зовательском уровне, так и на уровне ядра операционной системы, как в RISC-процессорах, так и в процессорах Intel. О возникновении подобных ситуаций может сигнализировать и аппаратное, и программное обеспечение. Любой язык программирования должен содержать средства обработки исключений для унификации процесса обработки, выполняемой в различных ситуациях. Windows включает встроенные низкоуровневые механизмы структурированной обработки исключений [12].

Исключения в значительной степени аналогичны прерываниям, и в первую очередь тем, что оба сигнала заставляют центральный процессор передать управление специальной части операционной системы. Однако исключения и прерывания — это не одно и то же. Прерывания происходят асинхронно, часто в результате определенных аппаратных событий, например нажатия клавиш или поступления данных через последовательный порт. Программа не имеет возможности контролировать такие прерывания и они могут происходить в любой момент. С другой стороны, исключения возникают синхронно как результат выполнения определенных операторов программы. Исключения часто служат сигналом, свидетельствующим о наличии ошибочных условий. Обычно они воспроизводятся путем повторного запуска программы с тем же контекстом.

Исключения часто генерируются в том случае, если какая-то часть программы обнаруживает ошибку, но не может обработать ее самостоятельно. Так, система генерирует исключение в ответ на ошибки доступа к памяти и ошибки типа деления на ноль. Однако не все исключения порождаются ошибочными ситуациями. Windows генерирует специальное исключение для вызова определенного системного сервиса. При обработке этого исключения ядро передает управление той части операционной системы, которая предоставляет запрашиваемый сервис.

Любое порожденное исключение должно быть обработано если не самой программой, то операционной системой, а точнее, должна откликнуться определенная подпрограмма, которая обработает и снимет исключение. Следовательно, обработка исключений заключается в создании специальных блоков программного кода, которые запускаются при возникновении исключительных ситуаций. Каждое приложение должно содержать несколько маленьких обработчиков, защищающих различные части программы от всевозможных исключений.

В процессе поиска соответствующего блока программного кода, предназначенного для конкретной исключительной ситуации, система сначала просматривает текущую процедуру, затем возвращается назад по стеку вызовов, просматривает другие активные отложенные процедуры того же процесса и, наконец, переходит к системным обработчикам исключений. Если процесс, породивший исключение, находится под защитой отладчика, последний также получает шанс обработать исключение.

Средства обработки ошибок в различных подсистемах и языках программирования несколько отличаются друг от друга. Например, WOW-подсистема (Windows on Win32 — защищенная подсистема, выполняющаяся внутри процесса виртуальной DOS-машины) должна обрабатывать все исключения непосредственно, поскольку клиенты Winl6 не имеют возможности делать это самостоятельно. Кроме того, в различных языках программирования может сильно различаться синтаксис операторов обработки исключений. Термин ***«структурированная обработка исключений»*** подразумевает, что язык содержит определенную управляющую структуру, связанную с исключениями.

Программисты, впервые столкнувшиеся с проблемой структурированной обработки исключений, часто ошибочно полагают, что больше нет необходимости проверять код завершения каждой команды. На самом деле ошибка — это далеко не то же самое, что исключение. Функция может завершиться без генерации исключения. Рассмотрим в качестве примера следующие строки программного кода, реализованного на API-функциях для ОС Windows:

```
hBrush = CreateSolidBrush(RGB(255, 0, 0));
hOldBrush = SelectObject(hDC, hBrush);
Rectangle (hDC, 0, 0, 100, 100);
```

Если первая команда выполняется некорректно и возвращает для создаваемой кисти значение NULL, то функция SelectObject() также не может быть выполнена. Третья команда все же рисует прямоугольник, однако закрашивает его не тем цветом, который нужен. Исключения при этом не генерируются. Единственный способ защиты от подобных ошибок заключается в проверке возвращаемого значения. Вот еще один пример:

```
HANDLE hMemory;
char *pData;
hMemory =GlobalAlloc(GHND, 1000);
pData = (char *)GlobalLock (hMemory);
```

В случае возникновения ошибки, связанной с выделением памяти, переменная hMemory принимает значение NULL, функция GlobalLock() не выполняется и переменная pData также получает значение NULL. Однако ни одна из этих ошибок не порождает исключения. Но следующая строка программного кода при попытке произвести запись по неправильному адресу генерирует исключение:

```
pData[0] = "a";                    // порождает исключение, если pData = NULL
```

Исключение представляет собой разновидность ошибки, которая не может быть обработана самой командой. Если функция GlobalAlloc не обнаруживает достаточно места для своего выполнения, она просто возвращает значение NULL. Но если оператору присваивания некуда передавать значение, то он не выполняет никаких действий и даже не возвращает код ошибки. При этом порождается исключение, и если процесс не может его обработать, операционная система должна закрыть данный процесс.

Часто бывает трудно провести черту между ошибками и исключениями. Различие между ними порой зависит от конкретной реализации. Для того чтобы распознавать команды, которые могут порождать исключения, необходимо иметь некоторый опыт. Вы должны знать всевозможные типы исключений и определить, какие операции могут их породить. Например, ошибка в операторе присваивания приводит к исключению типа ***нарушение прав доступа***. Список возможных исключений обычно изменяется в зависимости от конкретного компьютера, однако имеется ряд исключений, которые определяются на уровне ядра Windows [12]:

- несоответствие типов данных (*data-type misalignment*);
- прерывание отладчика (*debugger breakpoint*);
- пошаговая отладка (*debugger single-step*);
- деление на ноль в операции с плавающей запятой (*floating-point divide by zero*);
- переполнение и потеря разрядов в операции с плавающей запятой (*floating-point overflow and under flow*);
- зарезервированный операнд в операции с плавающей запятой (*floating-point reserved operand*);
- нарушение защиты страницы (*guard-page violation*);
- недопустимый оператор (*illegal instruction*);
- деление на ноль в операции с целыми числами (*integer divide by zero*);
- переполнение в операции с целыми числами (*integer overflow*);
- нарушение прав доступа к памяти (*memory-access violation*);
- ошибка чтения страницы (*page-read error*);
- превышение квоты страничного файла (*paging file quota exceeded*);
- привилегированный оператор (*privileged instruction*).

Используя функцию DWORD GetExceptionCode(void) можно определить тип возникшего исключения. Функция возвращает код исключения. В табл. 3.1 приведены наименования основных кодов.

Таблица3.1

EXCEPTION_ACCESS_VIOLATION	Попытка обращения к ячейке памяти, доступ к которой запрещен (например, память не выделена)
EXCEPTION_FLT_DIVIDE_BY_ZERO	Деление на ноль с плавающей точкой
EXCEPTION_INT_DIVIDE_BY_ZERO	Деление на ноль с фиксированной точкой
EXCEPTION_INT_OVERFLOW	Целочисленное переполнение
EXCEPTION_PRIV_INSTRUCTION	Попытка исполнения привилегированной команды

§ 3.3. Средства обработки исключений в Visual C++

Блок программного кода на языке C++ всегда должен начинаться ключевыми словами *try* и *catch*. Блок *try* помечает фрагмент программы, который может породить исключение, а блок *catch* содержит программу, запускающуюся при наличии исключения. С точки зрения программиста, подобные синтаксические структуры удобны для отделения программного кода, предназначенного для обработки исключительных ситуаций, от кода, выполняющего обычные задачи. В Visual C++ реализован механизм обработки исключений, который основан на схеме, предложенной ANSI-комитетом по стандартизации языка C++. В соответствии с этой схемой перед вызовом функции, которая может стать причиной возникновения исключительной ситуации, должен быть инициирован обработчик исключений. Если функция приводит к возникновению ненормальной ситуации, генерируется исключение и управление передается обработчику.

В среде Visual C++ обработка исключений поддерживается с помощью нескольких механизмов, а именно:

- функций, обеспечивающих структурированную обработку исключений;
- классов, отвечающих за обработку определенных типов исключений;
- макрокоманд, позволяющих осуществить структуризацию обработчиков исключений приложения;
- функций, предназначенных для генерации исключений различных типов.

Все эти механизмы позволяют генерировать исключения нужных типов и при необходимости прекращать выполнение программы.

Как уже было сказано, основу любого механизма обработки исключений в среде Visual C++ составляют операторы *try* и *catch*. Структура *try* / *catch* отделяет подпрограмму обработки исключения от программного кода, который выполняется в нормальных ситуациях:

```
try                                          // начало блока try
{
<операторы основной программы>               // код, в котором могут
// генерироваться исключения
}
catch( <фильтр> )                            // начало блока обработки исключений
{
<код подпрограммы обработки исключений>      // код, который выполняется
// при наличии исключения
}
```

В любой своей реализации механизм структурированной обработки исключений связывает блок защищаемого программного кода с подпрограммой обработки исключений. Если исключение происходит при выполнении защищаемого блока, управление передается фильтрующему выражению. Последнее обычно определяет тип исключения и принимает решение о том, как продолжать его обработку. Исключения, прошедшие через фильтр, поступают в подпрограмму-обработчик. Если фильтр блокирует исключение, подпрограмма-обработчик не вызывается и система продолжает поиск процедуры, которая возьмет на себя обработку этого исключения.

Фильтрующее выражение может быть достаточно сложным. При необходимости оно может даже вызывать отдельную функцию. Иногда фильтр сам выполняет обработку исключения, вследствие чего блок catch остается пустым. Во многих случаях для обработки исключительных ситуаций применяются MFC-класс CException и производные от него классы. Каждый из классов этой группы предназначен для обработки определенных типов исключений (табл. 3.2).

Эти исключения предназначены для использования в блоках try/catch. Обычно каждый производный класс служит для перехвата исключений определенного типа, но можно сначала использовать класс CException, предназначенный для перехвата исключений всех типов, а затем метод CObject::IsKindOf(), определяющий тип исключения (и соответствующий производный класс). Однако не следует забывать, что метод CObject::IsKindOf() применим только к тем классам, которые были объявлены с помощью макрокоманды IMPLEMENT_DYNAMIC, разрешающей динамическую проверку типов. При объявлении всех классов, производных от класса CException, также необходимо использовать эту макрокоманду.

Таблица 3.2

Класс	Описание
CMemoryException	Исключение, вызванное нехваткой памяти
CNotSupportedException	Запрос недопустимой операции
CArchiveException	Исключение при работе с архивами
CFileException	Исключение при выполнении операций с файлами
CResourceException	Ресурс не обнаружен или не может быть создан
COleException	OLE-исключение
CDBException	Исключение, возникающее при выполнении операций с базами данных; генерируется ODBC-классами библиотеки MFC
COleDispatchException	Ошибка диспетчеризации (OLE-автоматизации)
CUserException	Исключение, связанное с невозможностью найти заданный ресурс
CDaoException	Исключение, возникающее при работе с объектами доступа к данным (Data Access Object, DAO); генерируется DAO-классами библиотеки MFC
CInternetException	Исключение, связанное с работой в Internet; генерируется Internet-классами библиотеки MFC

Для получения подробной информации об исключении любого из классов, производных от CException, можно воспользоваться функциями GetErrorMessage() и ReportError().

В ответ на исключение система обычно пытается сначала запустить фильтр исключений, а затем подпрограмму-обработчик. Однако вы можете реализовать обработку нескольких различных исключений, организовав последовательность блоков catch, как показано в следующем фрагменте:

```
try
{
...                                  // блок программы, порождающий исключение
}
catch (CMemoryException *e)          // исключение типа нехватки памяти
{
...
}
catch (CFileException *e)            // исключение при выполнении операции с файлом
{
...
}
catch (CArchiveException *e)         // исключение при выполнении
{                                    //архивации / сериализации

...
}
catch (CNotSupportedException *e)
                                     // отклик на запрос
{                                    // сервиса, который не поддерживается
...
}
catch (CResourceException *e)        // исключение при выделении ресурса
{
...
}
catch(CDaoException *e )             // исключение при обращении к базе данных (DAO-классы)
{
...
}
catch (CDBException *e)              // исключение при обращении к базе данных (ODBC-классы)
```

```
{
...
}
catch(COleException *e)              // OLE-исключения
{
...
}
catch (COleDispatchException *e)     // исключение при выполнении диспетчеризации
                                     // (OLE-автоматизации)
{
...
}
catch (CUserException *e)            // пользовательское исключение
{
...
}
catch (CException *e)                // данный блок должен следовать после предыдущих блоков.
{                                    // иначе компилятор выдаст код ошибки
...
}
catch (...)                          // перехват всех исключений!!!
{
...
}
```

В этом примере все классы, производные от класса CException, включены в блоки *catch*. Поскольку CException представляет собой абстрактный базовый класс, вы не можете напрямую создавать производные от него классы — пользовательские классы могут быть порождены только от классов, производных от CException. Если вы хотите создать собственное исключение типа CException, воспользуйтесь в качестве модели одним из производных классов. Кроме того, при создании производного класса необходимо указать макрокоманду IMPLEMENT_DYNAMIC, организующую поддержку динамической проверки типов.

Наряду с тем, что вы можете явно указать производные от CException классы в операторах *catch*, в качестве аргумента этого оператора можно задать и многоточие: *catch* (...). При этом блок *catch* будет перехватывать исключения всех типов, в том числе исключения языка С и исключения, порожденные системой и приложениями. К ним относятся также исключения, связанные с нарушением защиты памяти, делением на ноль, недопустимыми операциями с плавающей запятой. Следует обратить внимание на то, что подобно обработчику CException, который должен следовать после всех подпрограмм обработки, основанных на классах CException, оператор *catch* (...) всегда должен быть последним обработчиком в своем блоке TRY.

Задача поиска процедуры обработки исключения становится более сложной, когда процесс находится в состоянии отладки, поскольку операционная система передает управление отладчику еще до начала поиска обработчиков, находящихся в самой программе. Обычно отладчик использует этот режим для обработки пошаговых и контрольных исключений, что дает возможность пользователю проанализировать контекст программы перед возобновлением ее выполнения.

Если отладчик принимает решение не обрабатывать исключения на стадии раннего оповещения, система возвращается к процессу и ищет обработчик исключений внутри него самого. При условии, что процесс также отвергает исключение, система дает отладчику повторную возможность обработать эту, теперь уже более серьезную, ситуацию. Но если отладчик отвергает исключение повторно, система берет управление на себя и вырабатывает собственный ответ, который обычно заключается в приказе уничтожить процесс.

Вместо того чтобы непосредственно применять блоки TRY / CATCH, можно структурировать свою программу с помощью макрокоманд обработки исключений. В этом случае необходимо начать с макрокоманды TRY, которая устанавливает блок TRY, отмечающий потенциально опасный фрагмент программы.

Исключения, порожденные в блоке TRY, обрабатываются в блоках CATCH и AND_CATCH. Допускается и рекурсивная обработка: исключения могут быть переданы внешнему блоку TRYпутем их игнорирования или в результате вызова макрокоманды THROW_LAST.

Все блоки CATCH должны завершаться макрокомандой END_CATCH или END_CATCH_ALL. Если в программе нет блоков CATCH, блок TRY сам должен завершаться указанными макрокомандами.

Макрокоманды THROW и THROW_LAST генерируют заданное исключение, прерывают выполнение программы и передают управление блоку CATCH. Если таковой отсутствует, управление передается модулю библиотеки MFC, который отображает сообщение об ошибке и завершает свою работу. Макрокоманда THROW_LAST передает исключение назад внешнему блоку CATCH.

Если исключение перехвачено одной из макрокоманд, объект CException удаляется автоматически. Если же оно перехвачено с использованием блока CATCH, объект CException невозможно удалить автоматически, поэтому в программе для этого должны быть предусмотрены специальные меры.

Глава 4
Принципы разработки программного кода, учитывающего организацию памяти в современных ОС

§ 4.1. Основы организации памяти

Оперативная память (ОП) — один из важнейших ресурсов ОС, без которого невозможна работа на персональном компьютере. Поэтому изучение механизмов использования памяти различными задачами и самой операционной системой по значимости аналогично изучению механизмов многопоточности, событий, прерываний и т. п. С появлением первых поколений ЭВМ в них было введено понятие ***оперативной памяти***, которую часто называют ***физической памятью***. Оперативная память представляет собой упорядоченное множество ячеек, которые пронумерованы, т. е. к каждой из них можно обратиться, указав ее порядковый номер (адрес). Количество ячеек физической памяти ограниченно и фиксированно. Центральный процессор извлекает из ОП команды, данные и помещает в нее результаты вычислений. Одновременно с этим, по мнению программистов, ОП — это область, с которой можно работать при помощи некоторого набора логических имен (переменных), и они могут быть символьными или числовыми. Следует отметить, что множество переменных неупорядочено, хотя отдельные элементы могут располагаться в некотором порядке, например элементы массива. Имена переменных и входных точек программных модулей составляют пространство имен.

Операционная система должна связать каждое указанное пользователем имя с физической ячейкой памяти, т. е. осуществить отображение пространства имен на физическую память компьютера. В общем случае это отображение осуществляется в два этапа, представленных на рис.4.1 [2]: сначала системой программирования, а затем операционной системой, управляющей соответствующими аппаратными средствами. В современных ОС переход с первого этапа на второй осуществляется через промежуточный ***виртуальный*** или логический адрес. При этом можно сказать, что множество всех допустимых значений виртуального адреса для некоторой программы определяет ее ***виртуальное адресное пространство,*** или ***виртуальную память***. Виртуальное адресное пространство зависит прежде всего от архитектуры процессора, системы программирования и практически не зависит от объема реальной физической памяти, установленной в персональном компьютере.

Любая система программирования осуществляет трансляцию и компоновку программы, используя библиотечные программные модули. В результате работы системы программирования полученные виртуальные адреса могут иметь как двоичную, так и символьно-двоичную форму, т. е. некоторые программные модули и их переменные получают какие-то числовые значения, а другие, адреса для которых в данный момент не могут быть определены, имеют по-прежнему символьную форму, и их окончательная привязка к физическим ячейкам будет осуществлена на этапе загрузки программы в память перед ее непосредственным выполнением.

Одним из частных случаев отображения пространства имен на физическую память [2] является ***тождественность виртуального адресного пространства физической памяти***. При этом нет необходимости осуществлять второе отображение. В данном случае говорят, что система программирования генерирует ***абсолютную двоичную программу***, в которой все двоичные адреса таковы, что программа может исполняться только в том случае, если ее виртуальные адреса будут точно соответствовать физическим. Часть программных модулей любой операционной системы обязательно должна быть абсолютной двоичной программой. Эти программы размещаются по фиксированным адресам и с их помощью можно далее реализовывать размещение остальных программных модулей, подготовленных системой программирования, на различных свободных физических адресах.

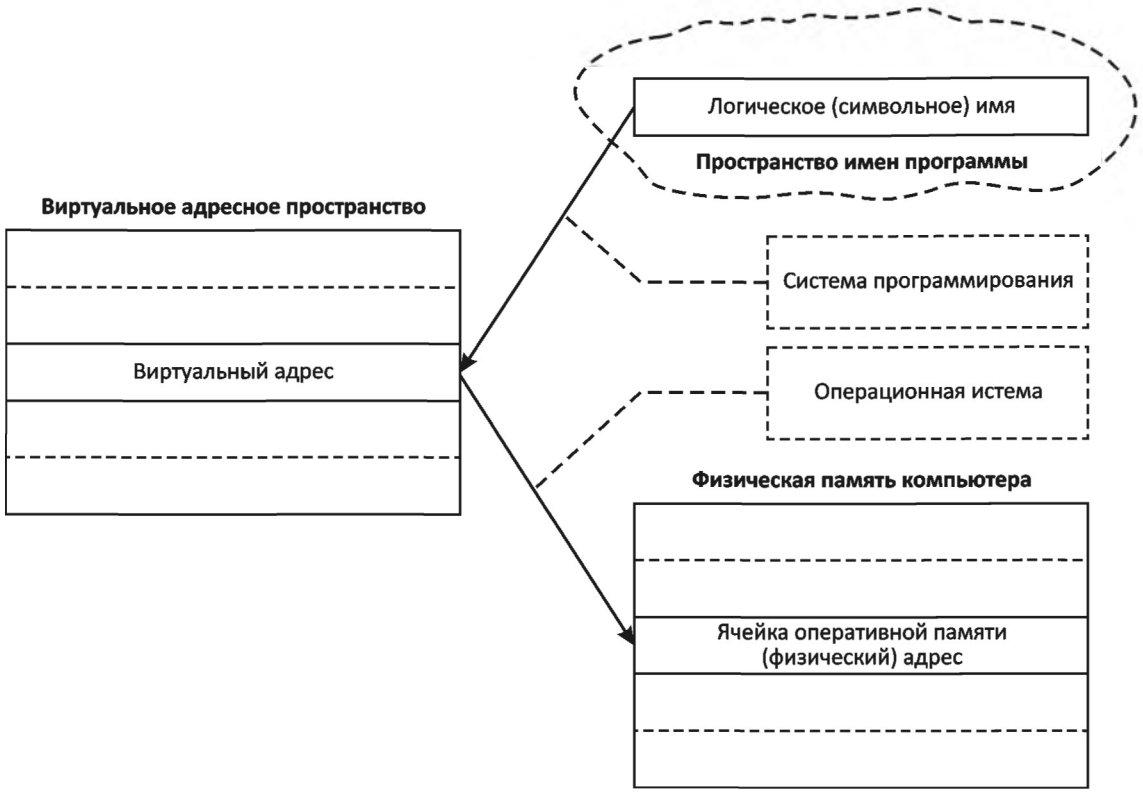

Рис.4.1. Связь переменной с физической памятью

Если рассматривать общую схему двухэтапного отображения адресов, то с позиции соотношения объемов упомянутых адресных пространств можно отметить наличие следующих трех ситуаций [2]:

– объем виртуального адресного пространства программы V_V меньше объема физической памяти V_P;

– $V_V = V_P$;

– $V_V > V_P$.

Первая ситуация, при которой $V_V < V_P$, в настоящее время практически не встречается, но тем не менее это реальное соотношение. Скажем, не так давно 16-разрядные мини-ЭВМ имели систему команд, в которых пользователи-программисты могли адресовать до 2^{16} = 64 Кбайт адресов (обычно в качестве адресуемой единицы выступала ячейка памяти размером 1 байт). Старшие модели этих мини-ЭВМ могли иметь объем оперативной памяти в несколько мегабайт. Обращение к памяти столь большого объема осуществлялось с помощью специальных регистров, содержимое которых складывалось с адресом операнда (или команды), извлекаемым и/или определяемым из поля операнда (или из указателя команды). Соответствующие значения в эти специальные регистры, выступающие как базовое смещение в памяти, заносила операционная система. Для одной задачи в регистр заносилось одно значение, а для второй (третьей, четвертой и т. д.) задачи, размещаемой одновременно с первой, но в другой области памяти, заносилось соответственно другое значение. Вся ***физическая память, таким образом, разбивалась на разделы объемом по 64 Кбайт***, и на каждый такой раздел осуществлялось отображение своего виртуального адресного пространства.

Ситуация, когда $V_V = Vp$, еще совсем недавно (80–90-е гг.XX в.) встречалась достаточно часто. Она была особенно характерна для недорогих вычислительных комплексов. Для этого случая имелось большое количество способов распределения оперативной памяти.

Наконец, в настоящее время наиболее вероятной является ситуация $V_V > V_P$. Для этого случая реализовано несколько способов распределения памяти, отличающихся как сложностью, так и эффективностью. Все эти способы можно разделить на две большие группы — ***непрерывные*** и ***разрывные*** [2, 5]. В первом случае каждая задача размещается в одной непрерывной области памяти, во втором разбивается на несколько не связанных между собой областей. Рассмотрим наиболее известные способы распределения памяти.

§ 4.2. Способы распределения памяти

4.2.1. Простое непрерывное распределение памяти

Это самое простое решение, согласно которому вся память условно может быть разделена на три участка:

- область, занимаемая операционной системой;
- область, в которой размещается исполняемая задача;
- свободная область памяти.

Изначально являясь самой первой схемой, она продолжает и сегодня быть достаточно распространенной в элементарных системах. При простом непрерывном распределении предполагается, что ОС не поддерживает мультипрограммирование, поэтому не возникает проблема распределения памяти между несколькими задачами. Программные модули, необходимые для всех программ, располагаются в области самой ОС, а вся оставшаяся память может быть предоставлена задаче. Эта область памяти является непрерывной, что облегчает работу системы программирования.

Для того чтобы выделить наибольший объем памяти для задач, ОС строится таким образом, что постоянно в оперативной памяти располагается только самая нужная ее часть. Эту часть стали называть ***ядром операционной системы***. Остальные модули ОС являются диск-резидентными, т. е. загружаются в оперативную память только по необходимости и после своего выполнения вновь выгружаются.

Такая схема распределения памяти влечет за собой ***два вида потерь вычислительных ресурсов***[2]: потерю процессорного времени, потому что центральный процессор простаивает, пока задача ожидает завершения операций ввода/вывода, и потерю самой оперативной памяти, потому что далеко не каждая программа использует всю память, а режим работы в этом случае однопрограммный. Однако это очень недорогая реализация, которая позволяет отказаться от многих дополнительных функций операционной системы, в частности от такой сложной проблемы, как защита памяти.

Классическим примером для данного случая является распределение памяти в ОС DOS. Как известно, MS-DOS — это однопрограммная ОС. В ней, конечно, можно организовать запуск резидентных или TSR-задач, но в целом она предназначена для выполнения только одного вычислительного процесса.

В IBM PC изначально использовался 16-разрядный микропроцессор i8086/88, который за счет введения сегментного способа адресации позволял адресовать память объемом до 1 Мбайт. В последующих ПК (IBM PC AT, AT386 и др.) поддерживалась совместимость с первыми, поэтому при работе с MS-DOS прежде всего рассматривают первый мегабайт. Таким образом, вся память в соответствии с архитектурой IBM PC условно может быть разбита на три части |ОС|.

В ***самых младших адресах памяти*** (первые 1024 ячейки) размещается таблица векторов прерываний. Это связано с аппаратной реализацией процессора i8086/88, на котором был реализован ПК. В последующих процессорах, начиная с i80286, адрес таблицы прерываний определяется через содержимое соответствующего регистра, но для обеспечения полной совместимости с первым процессором при включении или аппаратном сбросе в этот регистр заносятся нули. Следует отметить, что в случае использования современных микропроцессоров i80x86 векторы прерываний можно разместить и в другой области.

Вторая часть памяти отводится для размещения программных модулей самой MS-DOSи программ пользователя. Эта область памяти называется ***Conventional Memory*** (основная, стандартная память). В младших адресах основной памяти размещается то, что можно назвать ядром ОС: системные переменные, основные программные модули, блоки данных для буферизации операций ввода/вывода. Для управления устройствами, драйверы которых не входят в базовую подсистему ввода/вывода, загружаются так называемые ***загружаемые,*** или ***инсталлируемые,*** драйверы, перечень которых определяется в специальном конфигурационном файле CONFIG.SYS. В случае использования ПК с объемом ОП более 1 Мбайта и наличия в памяти драйвера HIMEM.SYS возможно размещение данных за пределами первого мегабайта. Эта область памяти получила название ***HMA*** (*high memory area* — область высокой памяти).

Наконец, ***третья часть адресного пространства*** отведена для постоянных запоминающих устройств и функционирования некоторых устройств ввода/вывода. Эта область памяти получила название ***UMA*** (*upper memory areas* — область верхней памяти).

Для того чтобы предоставлять больше памяти программам пользователя, в MS-DOS применено то же решение, что и во многих других простейших ОС — командный процессор COMMAND.COM состоит из двух частей. Первая часть является резидентной и размещается в области ядра. Вторая часть — транзитивная — размещается в области старших адресов раздела памяти, выделяемой для программ пользователя. Если программа пользователя перекрывает собой область, в которой была расположена транзитивная часть командного процессора, то последний при необходимости восстанавливает ее в памяти после выполнения программы, так как далее управление возвращается резидентной части COMMAND.COM.

4.2.2. Распределение памяти с перекрытием (оверлейные структуры)

Если есть необходимость создать программу, логическое (и виртуальное) адресное пространство которой должно быть больше, чем свободная область памяти, или даже больше, чем весь возможный объем оперативной памяти, то используется распределение с перекрытием. Этот способ распределения предполагает, что вся программа может быть разбита на части-сегменты. Каждая оверлейная программа имеет одну ***главную часть*** (*main*) и ***несколько сегментов*** (*segment*), причем в памяти машины одновременно могут находиться только ее главная часть и один или несколько неперекрывающихся сегментов.

Пока в оперативной памяти располагаются выполняющиеся сегменты, остальные находятся во внешней памяти. После того как текущий (выполняющийся) сегмент завершит свое выполнение, возможны два варианта: либо он сам (если данный сегмент не нужно сохранить во внешней памяти в его текущем состоянии) обращается к ОС с указанием, какой сегмент должен быть загружен в память следующим, либо он возвращает управление главному сегменту задачи (в модуль *main*), а тот обращается к ОС с указанием, какой сегмент сохранить (если это нужно), а какой сегмент загрузить в оперативную память, и вновь отдает управление одному из сегментов, располагающихся в памяти. Простейшие схемы сегментирования предполагают, что в памяти в каждый конкретный момент времени может располагаться только один сегмент (вместе с модулем *main*). Более сложные схемы, используемые в больших вычислительных системах, позволяют располагать сразу несколько сегментов. В некоторых вычислительных комплексах могли существовать отдельно сегменты кода и сегменты данных. В отличие от сегментов данных, которые необходимо сохранять в любом случае, сегменты кода, как правило, не претерпевают изменений в процессе своего исполнения, поэтому при загрузке нового сегмента кода на место отработавшего последний можно не сохранять во внешней памяти.

Первоначально программисты сами должны были включать в тексты своих программ соответствующие обращения (***вызовы***) к ОС и тщательно планировать, какие сегменты могут находиться в оперативной памяти одновременно, чтобы их адресные пространства не пересекались. В более современных системах программирования вызовы стали вставляться компиляторами в код программы автоматически, если в этом возникает необходимость.

4.2.3. Распределение памяти разделами

Для организации мультипрограммного режима необходимо обеспечить одновременное расположение в оперативной памяти нескольких задач (целиком или их частями). Самая простая схема распределения памяти между несколькими задачами предполагает, что память, незанятая ядром ОС, может быть ***разбита на несколько непрерывных частей*** (зон, разделов). Разделы характеризуются именем, типом, границами (как правило, указываются начало раздела и его длина). Разбиение памяти на несколько ***непрерывных разделов*** может быть ***фиксированным*** (***статическим***) либо ***динамическим.*** Рассмотрим эти утверждения более подробно.

Разбиение всего объема оперативной памяти на несколько разделов может осуществляться ***единовременно,*** т. е. в процессе генерации варианта ОС, который потом эксплуатируется, или по мере необходимости. Пример разбиения памяти на разделы с фиксированными границами приведен на рис.4.2.

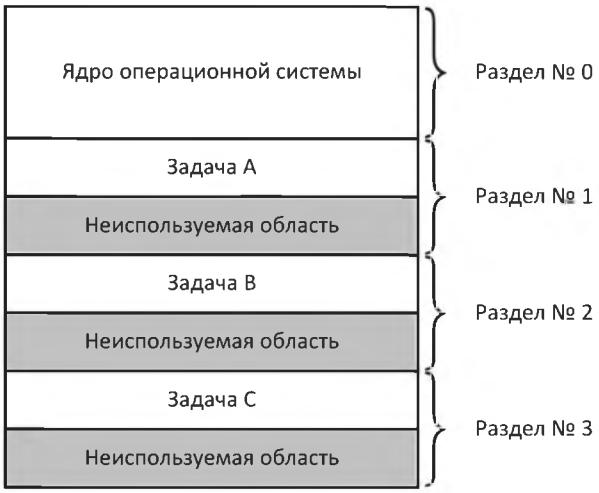

Рис. 4.2. **Распределение памяти разделами с фиксированными границами**

В каждом разделе на данный момент времени может располагаться по одной программе (задаче). В этом случае по отношению к каждому разделу можно применить все те методы разработки программ, которые используются для однопрограммных систем. Возможно использование оверлейных структур, что позволяет создавать более сложные программы и в то же время поддерживать идеологию мультипрограммирования. Первые многозадачные ОС, а в более поздний период — недорогие вычислительные системы строились по подобной схеме, так как она является несложной и обеспечивает возможность параллельного выполнения программ. Иногда в одном разделе размещалось несколько небольших программ, которые постоянно в нем и находились. Такие программы назывались ***ОЗУ-резидентными***.

При небольшом объеме памяти и, следовательно, небольшом количестве разделов увеличить количество параллельно выполняемых приложений (особенно когда эти приложения интерактивны и во время своей работы фактически не используют процессорное время, а в основном ожидают операций ввода/вывода) можно за счет идеологии ***свопинга*** (*swapping*). При ***свопинге*** задача может быть целиком выгружена на магнитный диск (перемещена во внешнюю память), а на ее место загружается либо более привилегированная, либо просто готовая к выполнению другая задача, находившаяся на диске в приостановленном состоянии.

Основным недостатком такого способа распределения памяти является наличие достаточно большого объема неиспользуемой памяти, которая может быть в каждом из разделов. Поскольку разделов несколько, то и неиспользуемых областей получается несколько, поэтому такие потери стали называть ***фрагментацией памяти***. В отдельных разделах потери памяти могли быть достаточно значительными, но использовать фрагменты свободной памяти при таком способе распределения практически невозможно.

Чтобы избавиться от фрагментации, можно размещать задачи в оперативной памяти плотно (***динамически***), одну за другой, выделяя ровно столько памяти, сколько задача требует.

Одной из первых ОС, реализовавшей такой способ распределения памяти, была ОС MVT (*multi programming with avariable number of tasks*) — мультипрограммирование с переменным числом задач. Эта ОС была одной из самых распространенных при эксплуатации больших ЭВМ класса IBM 360/370. В этой системе ***специальный планировщик*** (диспетчер памяти) ведет список адресов свободной оперативной памяти. При появлении новой задачи диспетчер памяти просматривает этот список и выделяет для задачи раздел, объем которого либо равен необходимому, либо чуть больше, если память выделяется не ячейками, а некими дискретными единицами. При этом модифицируется список свободной памяти. При освобождении раздела диспетчер памяти пытается объединить освобождающийся раздел с одним из свободных участков, если таковой является смежным.

При этом список свободных участков может быть упорядочен либо по адресам, либо по объему. Выделение памяти под новый раздел может осуществляться одним из трех методов [2]:

- первый подходящий участок;
- самый подходящий участок;
- самый неподходящий участок.

В ***первом случае*** список свободных областей упорядочивается по адресам (например, по их возрастанию). Диспетчер памяти просматривает этот список и выделяет задаче раздел в той области, которая первой подойдет по объему. В этом случае если подобный фрагмент имеется, то в среднем необходимо просмотреть половину списка. При освобождении раздела также необходимо просмотреть половину списка. Таким образом,***«первый подходящий» метод приводит к тому, что память для небольших задач преимущественно будет выделяться в области младших адресов, следовательно, это увеличивает вероятность того, что в области старших адресов будут образовываться фрагменты достаточно большого объема.***

Метод ***«самый подходящий»*** предполагает, что список свободных областей упорядочен по возрастанию объема этих фрагментов. В таком случае при просмотре списка для нового раздела будет использован фрагмент свободной памяти, объем которой наиболее точно соответствует требуемому. Требуемый раздел будет определяться по-прежнему в результате просмотра в среднем половины списка. В результате оставшийся фрагмент оказывается настолько малым, что в нем уже вряд ли удастся разместить какой-либо еще раздел, при этом данный фрагмент попадет в самое начало списка. Поэтому в целом такую дисциплину ***нельзя назвать эффективной***.

Как ни странно, ***самым эффективным методом*** является последний, по которому для нового раздела выделяется ***«самый неподходящий»*** фрагмент свободной памяти. Для этой дисциплины список свободных областей упорядочивается по убыванию объема свободного фрагмента. Очевидно, что если есть такой фрагмент памяти, то он сразу же и будет найден, и поскольку этот фрагмент является самым большим, то, скорее всего, после выделения из него раздела памяти для задачи оставшаяся область памяти еще сможет быть использована в дальнейшем.

Очевидно, что при любом из указанных методов вследствие того, что задачи появляются и завершаются в произвольные моменты времени и при этом имеют разные объемы, в памяти всегда будет наблюдаться сильная фрагментация. При этом возможны ситуации, когда из-за сильной фрагментации памяти диспетчер задач не сможет образовать новый раздел, хотя суммарный объем свободных областей будет больше, чем необходимо для задачи. В этой ситуации необходимо организовать так называемое ***уплотнение памяти (дефрагментацию)***. Для уплотнения все вычисления приостанавливаются, и диспетчер памяти корректирует свои списки, перемещая разделы в начало памяти (или, наоборот, в область старших адресов). ***Недостатком такого решения является потеря времени на дефрагментацию*** и, что самое главное, невозможность при этом выполнять сами вычислительные процессы.

Следует отметить, что данный способ распределения памяти применялся достаточно длительное время, так как в нем выделяется для задач непрерывное адресное пространство, что существенно упрощает создание систем программирования и их работу.

В современных ОС на смену различным ***непрерывным способам*** распределения памяти пришли ***разрывные способы***. Идея заключается в предложении размещать задачу не в одной непрерывной области памяти, а в нескольких областях. Это требует для реализации соответствующей аппаратной поддержки — нужно иметь относительную адресацию. Если указывать адрес начала текущего фрагмента программы и величину смещения относительно этого начального адреса, то можно выбрать необходимую нам переменную или команду. Таким образом, ***виртуальный адрес можно представить состоящим из двух полей***. Первое поле будет указывать часть программы (с которой сейчас осуществляется работа) для определения местоположения этой части в памяти, а второе поле виртуального адреса позволит найти нужную нам ячейку относительно найденного адреса. Программист может либо самостоятельно разбивать программу на фрагменты, либо автоматизировать эту задачу и возложить ее на систему программирования. Рассмотрим наиболее распространенные разрывные способы распределения памяти.

4.2.4. Сегментное распределение памяти

Для реализации этого способа программу необходимо разбивать на части и уже каждой части в отдельности выделять физическую память. Естественным механизмом разбиения программы на части является разбиение ее на ***логические элементы — сегменты***. В принципе каждый программный модуль может быть воспринят как отдельный сегмент, и тогда вся программа будет представлять собой множество сегментов. Каждый сегмент размещается в памяти как до определенной степени самостоятельная единица. Логически обращение к элементам программы в этом случае будет представляться как указание имени сегмента и смещения относительно начала данного сегмента. Физически имя (или порядковый номер) сегмента будет соответствовать некоторому адресу, с которого этот сегмент начинается при его размещении в памяти, и смещение должно прибавляться к этому базовому адресу.

Преобразование имени сегмента в его порядковый номер осуществляет ***система программирования***, а ***операционная система*** размещает сегменты в физической памяти и для каждого сегмента получает информацию о его начале. Таким образом, ***виртуальный адрес для данного способа состоит из двух полей: номера сегмента и смещения относительно начала сегмента***. Каждый сегмент, размещаемый в памяти, имеет соответствующую информационную структуру, часто называемую ***дескриптором сегмента***. Именно операционная система строит для каждого исполняемого процесса соответствующую таблицу дескрипторов сегментов и при размещении каждого из сегментов в оперативной или внешней памяти в дескрипторе отмечает его текущее местоположение. Если сегмент задачи в данный момент находится в оперативной памяти, то об этом делается пометка в дескрипторе. Как правило, для этого используется ***«бит присутствия»*** (*present bit*). ***В этом случае в поле «адрес» диспетчер памяти записывает адрес физической памяти***, с которого сегмент начинается, а ***в поле «длина сегмента»*** (*limit*) ***указывается количество адресуемых ячеек памяти***. Это поле используется не только для того, чтобы размещать сегменты без наложения один на другой, но и для того, чтобы проконтролировать, не обращается ли код исполняющейся задачи за пределы текущего сегмента. В случае превышения длины сегмента вследствие ошибок программирования мы можем говорить о нарушении адресации и с помощью введения специальных аппаратных средств генерировать сигналы прерывания, которые позволят обнаружить подобные ошибки. Если ***бит присутствия*** в дескрипторе указывает, что сейчас этот сегмент находится не в оперативной, а во внешней памяти (например, на жестком диске), то названные поля адреса и длины используются для указания адреса сегмента в координатах внешней памяти.

Помимо информации о местоположении сегмента в дескрипторе сегмента, как правило, содержатся данные о его типе (сегмент кода или сегмент данных), правах доступа к этому сегменту (можно или нельзя его модифицировать, предоставлять другой задаче), отметка об обращениях к данному сегменту (информация о том, как часто этот сегмент используется), на основании которых можно принять решение о предоставлении места, занимаемого текущим сегментом, другому сегменту.

При передаче управления следующей задаче ОС должна занести в соответствующий регистр адрес таблицы дескрипторов сегментов для этой задачи. Сама ***таблица дескрипторов сегментов***, в свою очередь, также представляет собой сегмент данных, который обрабатывается диспетчером памяти операционной системы. При таком подходе появляется возможность размещать в оперативной памяти не все сегменты задачи, а только те, с которыми в настоящий момент происходит работа. Если требуемого сегмента в оперативной памяти нет, то возникает прерывание, при этом управление передается через диспетчер памяти программе загрузки сегмента. Пока происходит поиск сегмента во внешней памяти и загрузка его в оперативную, диспетчер памяти определяет подходящее для сегмента место. Возможно, что свободного места нет — тогда принимается решение о выгрузке какого-нибудь сегмента и его перемещение во внешнюю память. Если при этом еще остается время, то процессор передается другой готовой к выполнению задаче. После загрузки необходимого сегмента процессор вновь передается задаче, вызвавшей прерывание из-за отсутствия сегмента. Всякий раз при считывании сегмента в оперативную память в таблице дескрипторов сегментов необходимо установить адрес начала сегмента и признак присутствия сегмента.

При поиске свободного места обычно используются правила «первого подходящего» и«самого неподходящего». Если свободного фрагмента в памяти достаточного объема нет, но тем не менее сумма свободных фрагментов превышает требования по памяти для нового сегмента, то

может быть применена процедура дефрагментации памяти. В идеальном случае размер сегмента должен быть достаточно малым, чтобы его можно было разместить в случайно освобождающихся фрагментах оперативной памяти, но и достаточно большим, чтобы содержать логически законченную часть программы с целью минимизировать межсегментные обращения.

Для решения проблемы замещения (определения того сегмента, который должен быть либо перемещен во внешнюю память, либо просто замещен новым) используются следующие правила [2]:

– ***FIFO*** (*first in — first out* — первый пришедший первым и выбывает);

– ***LRU*** (*least recently use*d — последний из недавно использованных, дольше всего неиспользуемый);

– ***LFU*** (*least frequently used* — используемый реже всех остальных);

– ***random*** — (случайный выбор сегмента).

Первая и последняя дисциплины являются самыми простыми в реализации, но они не учитывают, насколько часто используется тот или иной сегмент, и, следовательно, диспетчер памяти может выгрузить или расформировать тот сегмент, к которому в самом ближайшем будущем произойдет обращение.

Алгоритм FIFO ассоциирует с каждым сегментом время, когда он был помещен в память. Для замещения выбирается наиболее старый сегмент. Учет времени необязателен, когда все сегменты в памяти связаны в FIFO-очередь и каждый помещаемый в память сегмент добавляется в хвост этой очереди. Алгоритм учитывает только время нахождения сегмента в памяти, но не учитывает фактического использования сегментов. Например, первые загруженные сегменты программы могут содержать переменные, используемые на протяжении работы всей программы. Это приводит к немедленному возвращению к только что замещенному сегменту.

Для реализации дисциплин LRU и LFU необходимо, чтобы процессор имел дополнительные аппаратные средства. В принципе достаточно, чтобы при обращении к дескриптору сегмента для получения физического адреса, с которого сегмент начинает располагаться в памяти, соответствующий бит обращения менял свое значение (скажем, с нулевого, которое установила ОС, в единичное). Тогда диспетчер памяти может время от времени просматривать таблицы дескрипторов исполняющихся задач и собирать для соответствующей обработки статистическую информацию об обращениях к сегментам. В результате можно составить список, упорядоченный либо по длительности неиспользования (для дисциплины LRU), либо по частоте использования (для дисциплины LFU).

Важнейшей проблемой, которая возникает при организации мультипрограммного режима, является ***защита памяти***. Для того чтобы выполняющиеся приложения не смогли испортить саму ОС и другие вычислительные процессы, необходимо, чтобы доступ к таблицам сегментов с целью их модификации был обеспечен только для кода самой ОС. Для этого код ОС должен выполняться в некотором привилегированном режиме, из которого можно осуществлять манипуляции с дескрипторами сегментов, тогда как выход за пределы сегмента в обычной прикладной программе должен вызывать прерывание по защите памяти. Каждая прикладная задача должна иметь возможность обращаться только к своим собственным сегментам.

Остановимся на основных недостатках сегментного способа распределения памяти. Во-первых, по сравнению с непрерывными способами произошло ***увеличение времени на загрузку программы и ее сегментов***. Для получения доступа к искомой ячейке памяти необходимо сначала найти и прочитать дескриптор сегмента, а уже потом, используя данные из него, вычислить конечный физический адрес сегмента. Для того чтобы уменьшить эти потери, используется ***кэширование***, т. е. расположение дескрипторов выполняемых задач в кэш-памяти микропроцессора. Вторым недостатком является ***наличие фрагментации памяти***, хотя и существенно меньшей, чем при непрерывном распределении.

Кроме этого, имеются ***потери памяти и процессорного времени на размещение и обработку дескрипторных таблиц***, так как на каждую задачу необходимо иметь свою таблицу дескрипторов сегментов, а при определении физических адресов необходимо выполнять операции сложения. Уменьшение этих недостатков реализовано в страничном способе распределения памяти, который будет рассмотрен далее.

Примером использования сегментного способа организации виртуальной памяти является операционная система для ПК OS/2 ver. 1, созданная для процессора i80286. В этой ОС в полной мере использованы аппаратные средства микропроцессора, который специально проектировался для поддержки сегментного способа распределения памяти.

OS/2 v. 1 поддерживала распределение памяти, при котором выделялись сегменты программы и сегменты данных. Система позволяла работать как с именованными, так и с неименованными сегментами. Имена разделяемых сегментов данных имели ту же форму, что и имена файлов. Процессы получали доступ к именованным разделяемым сегментам, используя их имена в специальных системных вызовах. OS/2 v. 1 допускала разделение программных сегментов приложений и подсистем, а также глобальных сегментов данных подсистем. Сегменты, которые активно не использовались, могли выгружаться на жесткий диск. Система восстанавливала их, когда в этом возникала необходимость. Вообще вся концепция системы OS/2 была построена на понятии разделения памяти: процессы почти всегда разделяют сегменты с другими процессами. В этом состояло ее существенное отличие от систем типа UNIX, которые обычно разделяют только реентерабельные программные модули между процессами.

4.2.5. Страничное распределение памяти

В этом способе все фрагменты (части) задачи имеют одинаковый размер и длину, кратную степени двойки. Вследствие этого операции сложения можно заменить операциями слияния, что существенно уменьшает вычислительную нагрузку. Одинаковые части задачи называют ***страницами*** и говорят, что память разбивается на физические страницы, а программа — на виртуальные страницы. Часть виртуальных страниц задачи может размещаться в оперативной памяти, часть — во внешней. Обычно место во внешней памяти, в качестве которой в абсолютном большинстве случаев выступают жесткие диски, называют ***файлом подкачки*** или ***страничным файлом***(*paging file*). Иногда этот файл называют ***swap-файлом***, тем самым подчеркивая, что записи этого файла — страницы, которые замещают друг друга в оперативной памяти. В некоторых ОС выгруженные страницы располагаются не в файле, а в специальном разделе дискового пространства. В UNIX-системах для этих целей выделяется специальный раздел, но кроме него могут быть использованы и файлы, выполняющие те же функции, если объема выделяемого раздела недостаточно.

Разбиение всей оперативной памяти на страницы одинаковой величины, причем ***величина каждой страницы выбирается кратной степени двойки***, приводит к тому, что ***вместо одномерного адресного пространства памяти можно говорить о двумерном***. Первая координата адресного пространства — это номер страницы, а вторая координата — номер ячейки внутри выбранной страницы (его называют индексом). Таким образом, физический адрес определяется парой (P_P, i), а виртуальный адрес — парой (P_V, i), где P_V — номер виртуальной страницы, P_P — номер физической страницы, i —индекс ячейки внутри страницы. ***Количество бит, отводимое под индекс, определяет размер страницы, а количество бит, отводимое под номер виртуальной страницы, —объем возможной виртуальной памяти, которой может пользоваться программа*** [2]. Отображение, осуществляемое системой во время исполнения, сводится к отображению P_V в Pp и приписыванию к полученному значению битов адреса, задаваемых величиной i. При этом нет необходимости ограничивать число виртуальных страниц числом физических, т. е. не поместившиеся страницы можно размещать во внешней памяти, которая в данном случае служит расширением оперативной. Для отображения виртуального адресного пространства задачи на физическую память, как и в случае с сегментным способом организации, каждой задаче необходимо иметь ***таблицу страниц*** для трансляции адресных пространств. Для описания каждой страницы диспетчер памяти ОС заводит соответствующий ***дескриптор***, который отличается от дескриптора сегмента прежде всего тем, что в нем нет необходимости иметь поле длины, так как все страницы имеют одинаковый размер. По номеру виртуальной страницы в таблице дескрипторов страниц текущей задачи находится соответствующий элемент (дескриптор). Если бит присутствия имеет единичное значение, значит, данная страница сейчас размещена в оперативной, а не во внешней памяти, и в дескрипторе записан номер физической страницы, отведенной под данную виртуаль-

ную. Если же бит присутствия равен нулю, то в дескрипторе записан адрес виртуальной страницы, расположенной в данный момент во внешней памяти. Таким достаточно сложным механизмом и ***осуществляется трансляция виртуального адресного пространства на физическую память***.

Защита страничной памяти, как и в случае с сегментным механизмом, основана на контроле уровня доступа к каждой странице. Как правило, возможны следующие уровни доступа: ***только чтение***; ***чтение и запись***; ***только выполнение***. В этом случае каждая страница снабжается соответствующим кодом уровня доступа. При трансформации логического адреса в физический сравнивается значение кода разрешенного уровня доступа с фактически требуемым. При их несовпадении работа программы прерывается.

При обращении к виртуальной странице, не оказавшейся в данный момент в оперативной памяти, возникает прерывание и управление передается диспетчеру памяти, который должен найти свободное место. Обычно предоставляется первая же свободная страница. Если свободной физической страницы нет, то диспетчер памяти по одной из упомянутых дисциплин замещения (LRU, LFU, FIFO, *random*) определит страницу, подлежащую расформированию или сохранению во внешней памяти. На ее место он и разместит новую виртуальную страницу, к которой было обращение из задачи. Для использования дисциплин LRU и LFU в процессоре должны быть реализованы соответствующие аппаратные средства.

Если объем физической памяти небольшой и даже часто требуемые страницы не удается разместить в оперативной памяти, возникает так называемая ***пробуксовка***. Другими словами, ***пробуксовка*** —это ситуация, при которой загрузка нужной нам страницы вызывает перемещение во внешнюю память той страницы, с которой мы тоже активно работаем. Чтобы не допускать этого, можно ***увеличить объем оперативной памяти, уменьшить количество параллельно выполняемых задач*** либо попробовать использовать ***более эффективные дисциплины замещения***.

В абсолютном большинстве современных ОС используется дисциплина замещения страниц LRU как самая эффективная. Так, именно эта дисциплина используется в OS/2 и Linux. Но в ОС от компании Microsoft, например Windows NT, разработчики, желая сделать систему максимально независимой от аппаратных возможностей процессора, пошли на отказ от этой дисциплины и применили правило FIFO. Для того чтобы хоть как-нибудь сгладить ее неэффективность, была введена***«буферизация»*** тех страниц, которые должны быть записаны в файл подкачки на диск или просто расформированы. Принцип буферизации прост. Прежде чем замещаемая страница действительно будет перемещена во внешнюю память или просто расформирована, она помечается как кандидат на выгрузку. Если в следующий раз произойдет обращение к странице, находящейся в таком «буфере», то страница никуда не выгружается и уходит в конец списка FIFO. В противном случае страница действительно выгружается, а на ее место в «буфере» попадает следующий «кандидат». Величина такого «буфера» не может быть большой, поэтому эффективность страничной реализации памяти в Windows NT намного ниже, чем у других современных ОС, и явление пробуксовки проявляется при достаточно большом объеме оперативной памяти. В системе Windows NT файл с выгруженными виртуальными страницами носит название PageFile.sys. Следует отметить, что в современных версиях Windows NT Microsoft все же реализовала дисциплину замещения LRU. Более подробно распределение памяти в ОС семейства Windows будет рассмотрено далее.

Как и в случае с сегментным способом организации виртуальной памяти, страничный способ приводит к тому, что без специальных аппаратных средств он будет существенно замедлять работу вычислительной системы. Поэтому обычно используется кэширование страничных дескрипторов. Наиболее эффективным способом кэширования является использование ***ассоциативного кэша***. Именно такой ассоциативный кэш и создан в 32-разрядных микропроцессорах i80x86 начиная с i80386, который стал поддерживать страничный способ распределения памяти. В этих микропроцессорах имеется кэш на 32 страничных дескриптора. Поскольку размер страницы в них равен 4 Кбайт, возможно быстрое обращение к 128 Кбайт памяти.

Рассмотрев страничный способ организации памяти, можно сделать следующие выводы. ***Основным достоинством*** страничного способа, по сравнению с рассмотренными ранее, является ***минимально возможная фрагментация***. Так как на каждую задачу может приходиться по одной незаполненной странице, становится очевидным, что память можно использовать достаточно эффективно.

Недостатками такого ***способа*** является, во-первых, ***наличие существенных дополнительных накладных расходов*** для страничной трансляция виртуальной памяти. В данном случае таблицы страниц нужно тоже размещать в памяти. Кроме того, эти таблицы нужно обрабатывать — именно с ними работает диспетчер памяти.

Второй существенный недостаток страничной адресации заключается в том, что ***программы разбиваются на страницы случайно***, без учета логических взаимосвязей, имеющихся в коде. Это приводит к усложнению организации разделения программных модулей между выполняющимися процессами. Для исправления этого недостатка предложен сегментно-страничный способ, который будет рассмотрен позже.

Несмотря на перечисленные недостатки, все современные ОС для ПК используют именно этот способ распределения памяти.

4.2.6. Сегментно-страничное распределение памяти

В данном способе распределения памяти, как и в сегментном, программа разбивается на логически законченные части — сегменты и виртуальный адрес содержит указание на номер соответствующего сегмента. Вторая составляющая виртуального адреса — смещение относительно начала сегмента состоит из двух полей: виртуальной страницы и индекса. Другими словами, получается, что виртуальный адрес теперь состоит из трех компонентов: сегмента, страницы, индекса.

Очевидно, что этот способ организации виртуальной памяти вносит еще бóльшую задержку доступа к памяти. Необходимо сначала вычислить адрес дескриптора сегмента и прочитать его, затем вычислить адрес элемента таблицы страниц этого сегмента и извлечь из памяти необходимый элемент и уже только после этого можно приписать к номеру физической страницы номер ячейки в странице (индекс). Задержка доступа к искомой ячейке получается по крайней мере в три раза больше, чем при простой прямой адресации. Чтобы избежать этого, вводится кэширование, причем кэш, как правило, строится по ассоциативному принципу.

Главным достоинством сегментно-страничного способа является возможность размещать сегменты в памяти целиком. Сегменты разбиты на страницы, все страницы конкретного сегмента обязательно загружаются в память. Это позволяет уменьшить обращения к отсутствующим страницам. Страницы исполняемого сегмента при этом находятся в памяти в случайных местах, так как диспетчер памяти манипулирует страницами, а не сегментами. Таким образом, наличие сегментов облегчает реализацию разделения программных модулей между параллельными процессами. В этом случае возможна и динамическая компоновка задачи. Выделение же памяти страницами позволяет минимизировать фрагментацию.

Главный недостаток этого способа — еще большая потребность в вычислительных ресурсах. Сегментно-страничное распределение памяти достаточно сложно реализовать, используется оно редко, обычно в дорогих и мощных вычислительных системах. Возможность реализации сегментно-страничного способа организации памяти заложена и в семействе 32-разрядных микропроцессоров i80x86, но вследствие слабой аппаратной поддержки, трудностей при создании систем программирования и ОС он практически ***не используется в ПК***.

§ 4.3. Организация памяти в ОС Windows

В Win32API используется плоская 32-разрядная модель памяти. Каждому процессу выделяется «личное» (*private*) изолированное адресное пространство, размер которого составляет 4 Гб. Это пространство разбивается на регионы, при этом нижние 2 Гб этого пространства отведены процессу для свободного использования, а верхние 2 Гб зарезервированы для использования операционной системой. На рис. 4.3 представлена архитектура памяти Windows NT, а на рис. 4.4, для сравнения и выявления особенностей и отличий Windows NT, — Windows 9x.

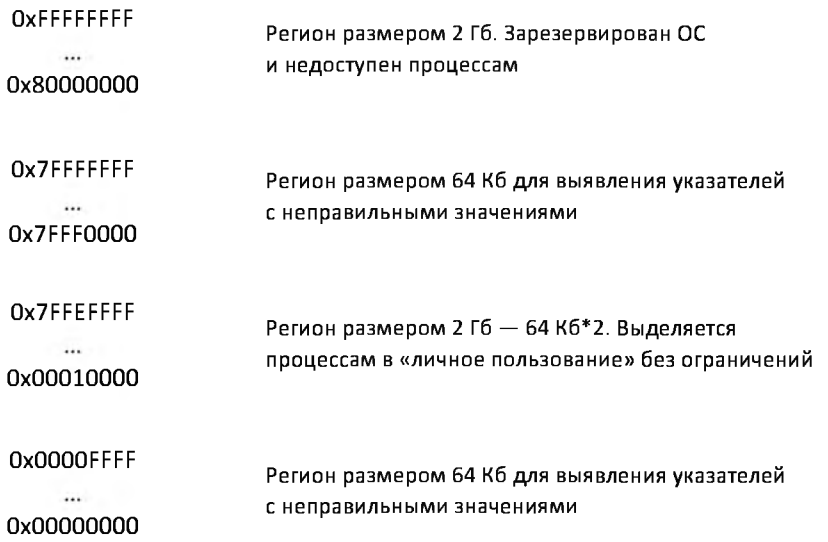

Рис. 4.3. Архитектура памяти Windows NT

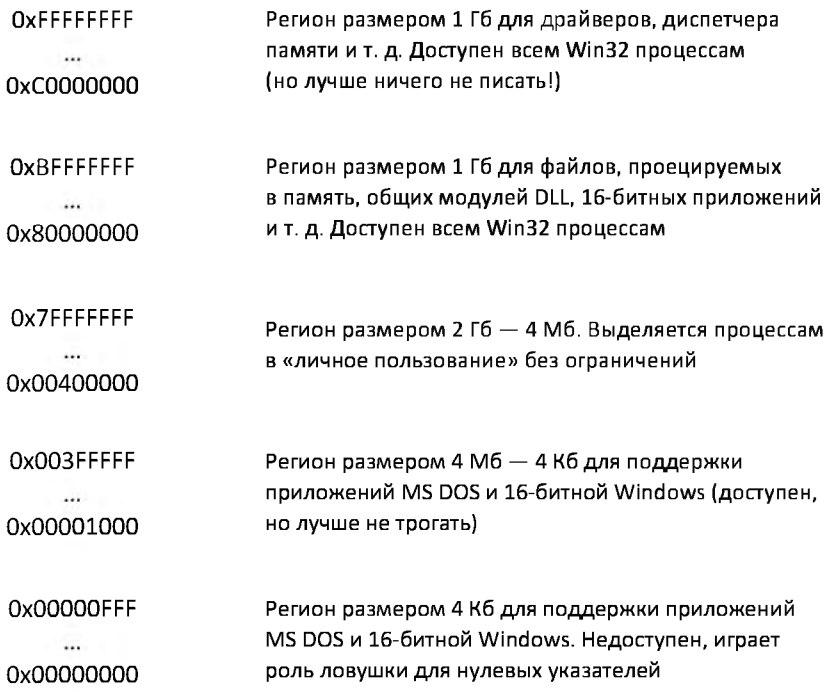

Рис.4.4. Архитектура памяти Windows 9x

Как видно из данных рисунков, код системы Windows NT лучше защищен от процесса пользователя, чем код Windows 9x. Это обусловливает большую устойчивость ОС к ошибкам в прикладной программе.

Используемые прикладными программами Windows 32-разрядные адреса для доступа к коду и данным не являются 32-разрядными физическими адресами, которые микропроцессор использует для адресации физической памяти. Поэтому адрес, который используется приложением, является виртуальным адресом (*virtual address*).

API-функции Windows, работающие в логическом адресном пространстве объемом до 2 Гб, поддерживаются менеджером виртуальной памяти — VMM (*Virtual Memory Manager*), который, в свою очередь, оптимизирован для работы на современных 32-разрядных процессорах. Для того чтобы аппаратное обеспечение системы могло использовать 32-разрядную адресацию памяти,

Windows обеспечивает отображение физических адресов в виртуальном адресном пространстве и поддерживает страничную организацию памяти. На этой основе VMM формирует собственные механизмы и алгоритмы управления виртуальной памятью.

Как виртуальной реальности, так и виртуальной оперативной памяти не существует, однако у системы возникает полная иллюзия ее наличия: 2 Гб адресного пространства, к которому может обращаться любая программа, представляют собой виртуальную память. На самом деле в системе нет 2 Гб физической памяти, однако каким-то образом программы могут использовать весь диапазон адресов. Очевидно, присутствует некий фоновый «переводчик», молчаливо преобразующий каждое обращение к памяти в реальный физический адрес. Благодаря такому преобразованию реализуется большинство возможностей менеджера виртуальной памяти (VMM). VMM — это часть операционной системы Windows, которая отвечает за преобразование ссылок на виртуальные адреса памяти в ссылки на реальную физическую память. Менеджер виртуальной памяти принимает решения о том, где размещать каждый объект памяти и какие страницы памяти записать на диск. Он также выделяет каждому выполняемому процессу отдельное адресное пространство, отказываясь преобразовывать виртуальные адреса одного процесса в адреса физической памяти, используемой другим процессом. VMM поддерживает иллюзию «идеализированного» адресного пространства объемом 2 Гб (подобно GDI, который создает видимость того, что каждая программа выводит графические данные в координатном пространстве «идеализированного» логического устройства). Система преобразует логические адреса памяти или логические координаты в физические и предотвращает конфликты между программами из-за попыток одновременного использования одного и того же ресурса. Рассмотрим более подробно механизм образования виртуальной памяти.

Физическая память делится на страницы (*pages*) размером 4096 байт (4 Кб). Следовательно, каждая страница начинается с адреса, в котором младшие 12 бит нулевые. Машина, оснащенная 8 Мб памяти, содержит 2048 страниц. Операционная система Windows хранит набор таблиц страниц (каждая таблица сама по себе представляет страницу) для преобразования виртуального адреса в физический.

Каждый процесс, выполняемый в Windows, имеет свою собственную страницу каталога (*directory page*) таблиц страниц, которая содержит до 1024 32-разрядных дескрипторов таблиц страниц. Физический адрес страницы каталога таблиц страниц хранится в регистре CR3 микропроцессора. Содержимое этого регистра изменяется при переключении Windows управления между процессами. Старшие 10 бит виртуального адреса определяют один из 1024 возможных дескрипторов в каталоге таблиц страниц. В свою очередь, старшие 20 бит дескриптора таблицы страниц определяют физический адрес таблицы страниц (младшие 12 бит физического адреса равны нулю). Каждая таблица страниц содержит, в свою очередь, до 1024 32-разрядных дескрипторов страниц. Выбор одного из этих дескрипторов определяется содержимым средних 10 битов исходного виртуального адреса. Старшие 20 бит дескриптора страницы определяют физический адрес начала страницы, а младшие 12 бит виртуального адреса определяют физическое смещение в пределах этой страницы. На рис. 4.5 представлена схема организации виртуального адресного пространства.

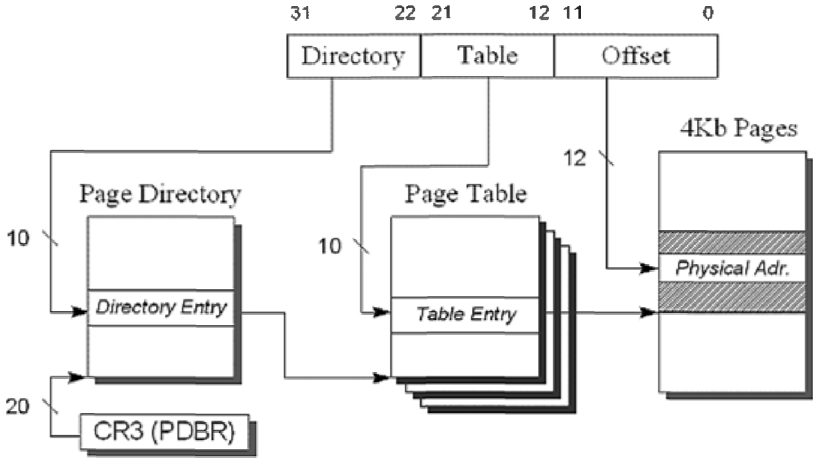

Рис. 4.5. Схема организации виртуального адресного пространства

Очевидно, это сложно понять с первого раза, поэтому проиллюстрируем процесс еще раз в символьной форме [6]. Вы можете представить 32-разрядный виртуальный адрес (с которым оперирует программа) в виде 10-разрядного индекса в таблице каталога таблиц страниц (*d*), 10-разрядного индекса в таблице страниц (*p*), 12-разрядного смещения (*o*):

dddd-dddd-ddpp-pppp-pppp-oooo-oooo-oooo

Для каждого процесса микропроцессор хранит в регистре CR3 (*r*) старшие 20 бит физического адреса таблицы каталога таблиц страниц:

rrrr-rrrr-rrrr-rrrr-rrrr

Начальный физический адрес каталога таблиц страниц определяется как

rrrr-rrrr-rrrr-rrrr-rrrr-0000-0000-0000

Необходимо запомнить, что каждая страница имеет размер 4 Кб и начинается с адреса, у которого 12 младших бит нулевые. Сначала микропроцессор получает физический адрес:

*rrrr-rrrr-rrrr-rrrr-rrrr-dddd-dddd-dd*00

По этому адресу содержится другое 20-разрядное значение (*t-table*):

tttt-tttt-tttt-tttt-tttt,

соответствующее начальному физическому адресу таблицы страниц:

tttt-tttt-tttt-tttt-tttt-0000-0000-0000

Затем, микропроцессор осуществляет доступ по физическому адресу:

*tttt-tttt-tttt-tttt-tttt-pppp-pppp-pp*00

Здесь хранится 20-битная величина, являющаяся основой для физического адреса начала страницы памяти (*f-pageframe*):

ffff-ffff-ffff-ffff-ffff

Результирующий 32-разрядный физический адрес получается в результате комбинирования основы физического адреса страницы и 12-разрядного смещения виртуального адреса:

ffff-ffff-ffff-ffff-ffff-oooo-oooo-oooo

Это и есть результирующий физический адрес.

Преимущества разделения памяти на страницы огромны. Во-первых, приложения изолированы друг от друга. Никакой процесс не может случайно или преднамеренно использовать адресное пространство другого процесса, так как он не имеет возможности его адресовать без указания соответствующего значения регистра CR3 этого процесса, которое устанавливается только внутри ядра Windows.

Во-вторых, такой механизм разделения на страницы решает одну из основных проблем в многозадачной среде — объединение свободной памяти. При более простых схемах адресации в то время как множество программ выполняются и завершаются, память может стать фрагментированной. В том случае, если память сильно фрагментирована, программы не могут выполняться из-за недостатка непрерывной памяти, даже если общего количества свободной памяти вполне достаточно. При использовании разделения на страницы нет необходимости объединять свободную физическую память, поскольку страницы необязательно должны быть расположены последовательно. Все управление памятью производится с помощью манипуляций с таблицами страниц. Потери связаны только собственно с самими таблицами страниц и с их размером 4 Кб.

В-третьих, в 32-битных дескрипторах страниц существует еще 12 бит, кроме тех, которые используются для адреса страницы. Один из этих битов показывает возможность доступа к конкретной странице (он называется битом доступа — *accessed bit*); другой показывает, была ли произведена запись в эту страницу (он называется битом мусора — *dirty bit*). Windows может использовать

эти биты для того, чтобы определить, можно ли сохранить эту страницу в файле подкачки для освобождения памяти. Еще один бит — бит присутствия (*present bit*) показывает, была ли страница сброшена на диск и нужно ли ее подкачать обратно в память.

Другой бит (чтения/записи) показывает, разрешена ли запись в данную страницу памяти. Этот бит обеспечивает защиту кода от «блуждающих» указателей. Например, если включить в программу для Windows оператор:

```
*(int*) WinMain = 0;
```

то на экран будет выведено следующее окно-сообщение: "*This program has performedan illegal operation and will be shut down*" («Эта программа выполнила недопустимую операцию и будет завершена»). Этот бит не препятствует компилированной и загруженной в память программе быть запущенной на выполнение. Приведем несколько замечаний по поводу управления памятью в Windows 9x.

Виртуальные адреса имеют разрядность 32 бита. Программа и данные имеют адреса в диапазоне от 0×00000000 до 0×7FFFFFFF. Сама Windows использует адреса от 0×80000000 до 0×FFFFFFFF. В этой области располагаются точки входа в динамически подключаемые библиотеки.

Общее количество свободной памяти, доступной программе, определяется как количество свободной физической памяти плюс количество свободного места на жестком диске, доступного для свопинга страниц. Как правило, при управлении виртуальной памятью Windows использует алгоритм LRU (*least recently used*) для определения того, какие страницы будут сброшены на диск. Бит доступа и бит мусора помогают осуществить эту операцию. Страницы кода не должны сбрасываться на диск: поскольку запись в его страницы запрещена, они могут быть просто загружены из файла с расширением .EXE или из динамически подключаемой библиотеки.

Организацией свопинга занимается VMM. При генерации системы на диске образуется специальный файл свопинга, куда записываются те страницы, которым не находится места в физической памяти. Процессы могут захватывать память в своем 32-битном адресном пространстве и затем использовать ее. При обращении потока к ячейке памяти могут возникнуть три различные ситуации [12]:

- страница существует и находится в памяти;
- страница существует и выгружена на диск;
- страница не существует.

При этом VMM использует алгоритм организации доступа к данным, представленный на рис.4.6 [8].

Запуск на исполнение EXE-модуля происходит следующим образом: EXE-файл проецируется на память. При этом он не переписывается в файл подкачки. Просто элементы каталога и таблиц страниц настраиваются так, чтобы они указывали на EXE-файл, лежащий на диске. Затем передается управление на точку входа программы. При этом возникает исключение, обрабатывая которое стандартным образом VMM загружает в память требуемую страницу и программа начинает исполняться. Такой механизм существенно ускоряет процедуру запуска программ, так как загрузка страниц EXE-модуля происходит по мере необходимости. Образно говоря, вначале программа начинает исполняться, а потом загружается в память. Если программа записана на дискете, то она перед началом исполнения переписывается в файл подкачки.

Для поддержания иллюзии огромного адресного пространства менеджеру виртуальной памяти необходимо знать, как правильно организовать данные. Все операции распределения памяти, которые процесс выполняет в выделенном ему диапазоне виртуальных адресов, записываются в виде дерева дескрипторов виртуальных адресов — VAD (*Virtual Address Descriptor*). Каждый раз при выделении программе памяти VMM создает дескриптор виртуального адреса (VAD) и добавляет его к дереву (рис.4.7) [12]. VAD содержит информацию о запрашиваемом диапазоне адресов, статусе защиты всех страниц в указанном диапазоне, а также о том, могут ли дочерние процессы наследовать объекты, которые находятся в данном диапазоне адресов. Если поток использует адрес, который не определен ни в одном дескрипторе, менеджер виртуальной памяти воспринимает его как адрес, который никогда не резервировался, вследствие чего возникает ошибка доступа.

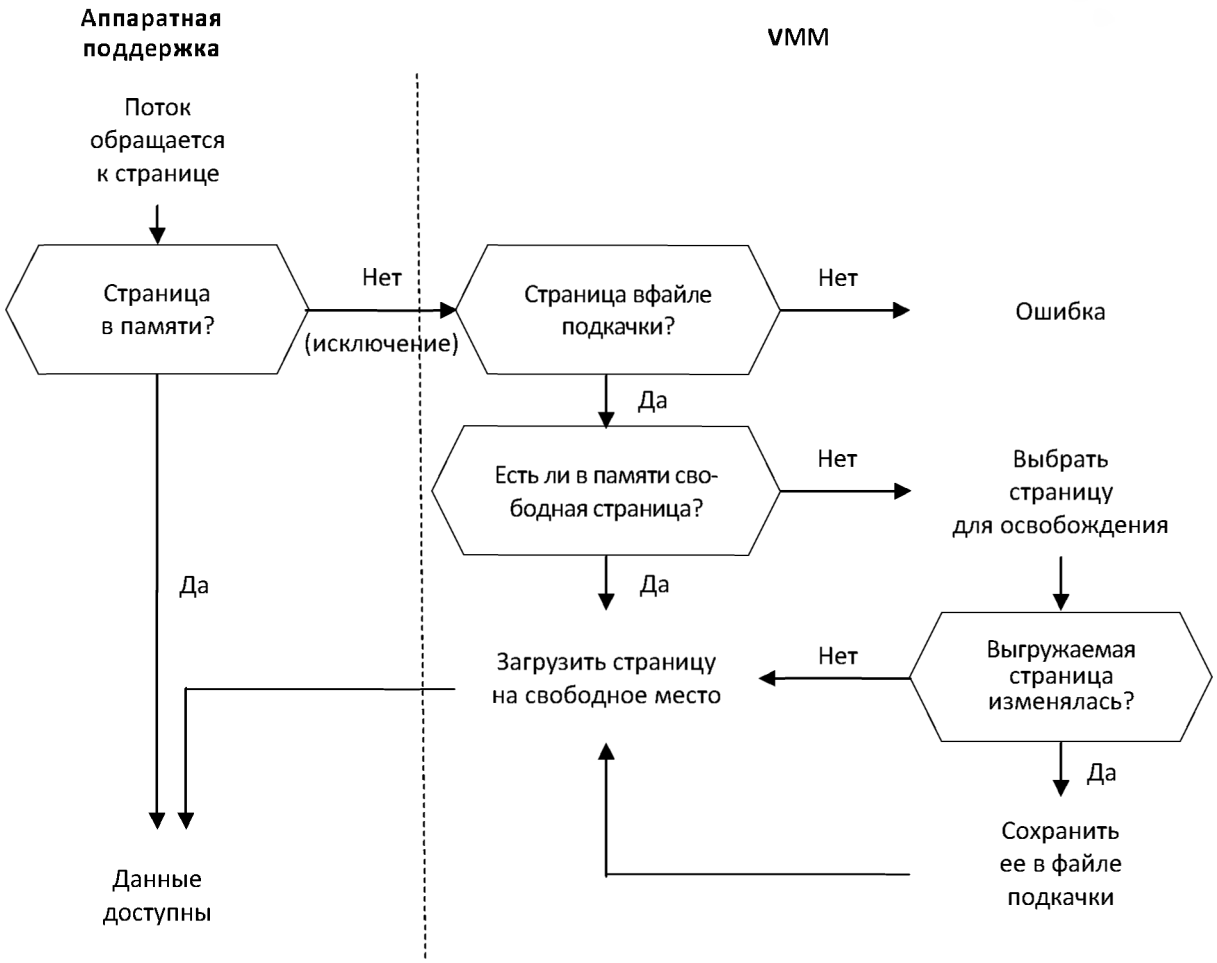

Рис. 4.6. Алгоритм организации доступа к данным

Намного проще формировать VAD, чем создавать таблицу страниц и заполнять ее адресами действительных страничных блоков. Кроме того, объем выделенной памяти не влияет на скорость проведения операции. Резервирование 2 Кб происходит не быстрее, чем выделение 2 Мб: по каждому запросу создается один дескриптор. Если поток использует зарезервированную память, VMM закрепляет за ним страничные блоки, копируя информацию из дескриптора в новую запись таблицы страниц.

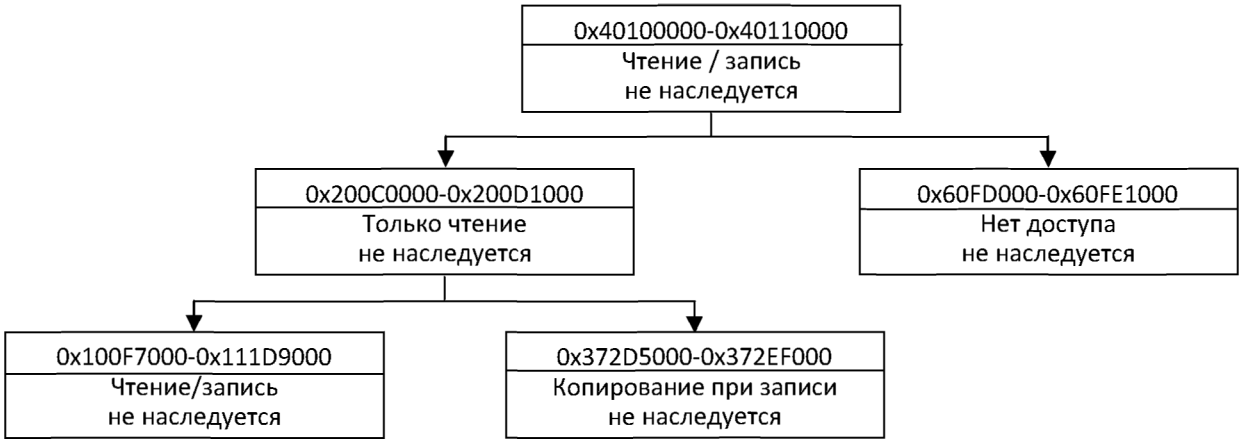

Рис. 4.7. Дерево дескрипторов виртуальных адресов (VAD)

Диспетчер управления памятью (VMM) является составной частью ядра операционной системы. Приложения не могут получить к нему прямой доступ.

§ 4.4. Интерфейсы API-функций для разработки программ с выделением памяти в ОС Windows

Для управления памятью прикладным программам предоставляются различные интерфейсы (API), схема которых приведена на рис.4.8 [4], где:

- ***Virtual Memory API*** — набор функций, позволяющих приложению работать с виртуальным адресным пространством: назначать физические страницы блоку адресов и освобождать их, устанавливать атрибуты защиты;
- ***Memory Mapped File API*** — набор функций, позволяющих работать с файлами, отображаемыми в память: новый механизм, предоставляемый Win32API для работы с файлами и взаимодействия процессов;
- ***Heap Memory API*** — набор функций, позволяющих работать с динамически распределяемыми областями памяти (кучами);
- ***Local, Global Memory API*** — набор функций работы с памятью, совместимых с 16-битной Windows (следует избегать их использования);
- ***CRT Memory API*** — функции стандартной библиотеки языка "С" периода исполнения (*run time*).

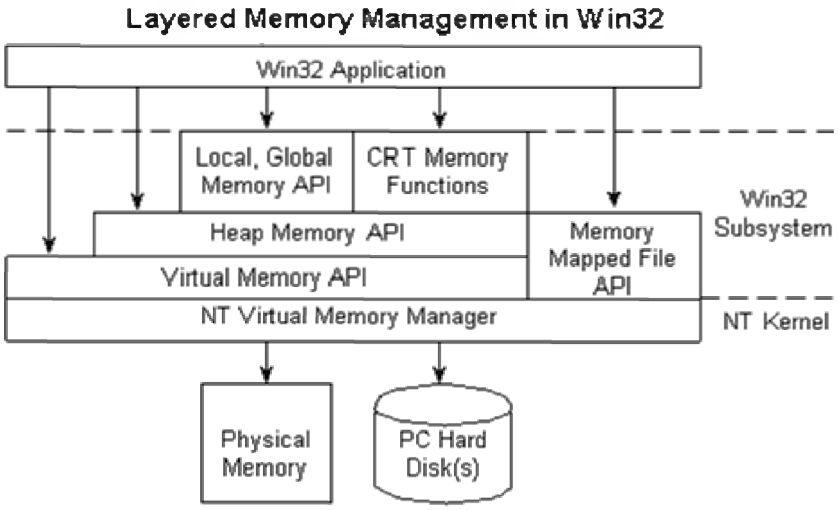

Рис.4.8. Интерфейсы API-функций для управления памятью

Рассмотрим более подробно первых три интерфейса API, поскольку они применяются в настоящее время наиболее часто.

4.4.1. API-функции для программ с выделением виртуальной памяти

Блок адресов в адресном пространстве процесса может находиться в одном из трех состояний [4]:

1) ***выделен (committed)***— блоку адресов назначена физическая память либо часть файла подкачки;

2) ***зарезервирован (reserved)***— блок адресов помечен как занятый, но физическая память не распределена;

3) ***свободен (free)***— блок адресов не выделен и не зарезервирован.

Резервирование и выделение памяти производится блоками. Начальный адрес блока должен быть выровнен на границу 64 Кбайт (округляется вниз), а размер кратен размеру страницы (округляется вверх). При выделении память обнуляется.

Для резервирования региона памяти в адресном пространстве процесса или для ее выделения используется функция ***VirtualAlloc***, а для освобождения — функция ***VirtualFree***.

Функция VirtualAlloc является базовой при выполнении операций управления виртуальным адресным пространством. Параметры этой функции определяют, какой объем памяти необходимо

выделить, в каком месте адресного пространства должен располагаться выделенный фрагмент, надо ли закреплять за ним физическую память и какой вид защиты следует установить. Функция возвращает адрес выделенного региона или NULL в случае неудачи:

```
LPVOID VirtualAlloc (LPVOID lpvAddress,        // адрес для размещения нового блока
    DWORD dwSize,                              // размер нового блока
    DWORD fdwAllocationType,                   // зарезервировать адреса или закрепить
                                               // физическую память
    DWORD fdwProtect);                         // нет доступа, только чтение
                                               // или чтение/запись
```

Функция VirtualAlloc сначала пробует найти область свободных адресов размером dwSize байтов, которая начинается с адреса lpvAddress. Для этого она просматривает дерево VAD. Если необходимая область памяти свободна, функция возвращает значение lpvAddress. В противном случае она просматривает все адресное пространство и ищет свободный блок памяти достаточного размера. При обнаружении такого блока функция возвращает его начальный адрес, иначе —значение NULL.

Аргумент fdwAllocationType может принимать значение MEM_RESERVE, либо MEM_COMMIT, либо оба значения одновременно. Для резервирования определенного интервала адресов функция VirtualAlloc создает новый VAD, который отмечает используемую область. Однако эта функция не выделяет физическую память, из-за чего невозможно использовать зарезервированные адреса. При попытке чтения или записи в зарезервированные страницы возникает ошибка доступа к памяти. С другой стороны, никакая другая команда выделения памяти не может использовать ранее зарезервированные адреса. Например, функции GlobalAlloc и malloc не могут разместить новые объекты в области, которая пересекается с зарезервированным адресным пространством. Попытка заставить функцию VirtualAlloc зарезервировать все доступное адресное пространство (1 Гб) приведет к конфликту: последующие операции выделения памяти не будут выполняться, даже если указанная функция не задействовала никакой физической памяти.

Память не может быть закреплена, если она не зарезервирована. Комбинация флагов MEM_RESERVE и MEM_COMMIT позволяет одновременно зарезервировать и закрепить указанную область памяти. Часто программисты вызывают функцию VirtualAlloc сначала с флагом MEM_RESERVE для резервирования большой области памяти, а затем несколько раз подряд с флагом MEM_COMMIT для поэтапного закрепления отдельных фрагментов.

Флаг fdwProtect определяет, каким образом использовать определенную страницу или диапазон страниц. Для резервируемой памяти этот флаг должен иметь значение PAGE_NOACCESS. При закреплении памяти устанавливается флаг PAGE_READONLY или PAGE_READWRITE. Другие программы не могут читать информацию из адресного пространства вашего процесса, поэтому режим доступа только для чтения обеспечивает защиту вашей программы от ошибок, которые могут привести к случайному повреждению важной информации. Уровни защиты применимы к отдельным страницам. Различные страницы в одной области памяти могут иметь разные значения флага защиты. Например, вы можете применить флаг PAGE_READONLY ко всему блоку, а затем временно изменять уровень защиты отдельных страниц, разрешая доступ к ним для записи. Защитить от записи только часть страницы невозможно, поскольку флаги устанавливаются для целых страниц. В табл. 4.1 приведены возможные значения, которые может принимать флаг fdwProtect.

Функция VirtualAlloc не может зарезервировать более 1 Гб памяти, поскольку процесс контролирует только нижнюю половину своего адресного пространства объемом 2 Гб. В действительности объем контролируемой памяти еще меньше из-за свободных областей (по 64 Кб каждая) на границах адресного пространства процесса. Кроме того, функция VirtualAlloc резервирует память фрагментами по 64 Кб, а закрепляет ее фрагментами объемом в одну страницу. При резервировании памяти функция VirtualAlloc округляет аргумент lpvAddress до ближайшего значения, кратного 64 Кб. При закреплении памяти функция VirtualAlloc осуществляет одно из двух действий. Если аргумент lpvAddress имеет значение NULL, функция VirtualAlloc округляет значение аргумента dwsize до ближайшей границы между страницами. Если значение аргумента lpvAddress не равно NULL, она закрепляет все страницы, содержащие хотя бы один байт информации в диапа-

зоне адресов от lpvAddress до lpvAddress + dwSize. Например, если при выделении двух байтов памяти заданный адрес пересекает границу двух страниц, закрепляются две целые страницы. В большинстве систем Windows 98 размер страницы составляет 4 Кб, но если вы хотите проверить это значение, вызовите функцию GetSystemInfo.

Таблица 4.1

Значение флага	Выполняемое действие
PAGE_READONLY	Допускается только чтение
PAGE_READWRITE	Допускается чтение и запись
PAGE_EXECUTE	Допускается только исполнение
PAGE_EXECUTE_READ	Допускается исполнение и чтение
PAGE_EXECUTE_READWRITE	Допускается исполнение, чтение и запись
PAGE_GUARD	Дополнительный флаг защиты, который комбинируется с другими флагами. При первом обращении к странице этот флаг сбрасывается и возникает исключение STATUS_GUARD_PAGE. Этот флаг используется для контроля размеров стека с возможностью его динамического расширения
PAGE_NOCACHE	Запрещает кэширование страниц. Может быть полезен при разработке драйверов устройств (например, данные в видеобуфер должны переписываться сразу, без кэширования)

По завершении процесса система автоматически освобождает использованную им память. Освободить память, не дожидаясь окончания процесса, позволяет функция VirtualFree:

```
BOOL VirtualFree (LPVOID lpvAddress,        // адрес освобождаемого блока
      DWORD dwSize,                         // размер освобождаемого блока
      DWORD fdwFreeType);                   // перевести в резерв или освободить
```

Функция VirtualFree отменяет закрепление набора страниц, оставляя их адреса зарезервированными, или полностью освобождает память, занимаемую этими страницами. Закрепление можно отменять маленькими блоками, содержащими как зарезервированные, так и закрепленные страницы.

При освобождении зарезервированных адресов необходимо очищать весь выделенный блок, причем все страницы этого блока должны находиться в одинаковом состоянии — либо в закрепленном, либо в зарезервированном. Аргумент lpvAddress должен содержать базовый адрес, возвращенный функцией VirtualAlloc. Значение аргумента dwSize игнорируется, поскольку сразу освобождается весь выделенный диапазон. Аргумент dwSize учитывается лишь при отмене закрепления отдельных фрагментов. Аргумент fdwFreeType может принимать следующие значения:

- MEM_DECOMMIT — освободить выделенную память;
- MEM_RELEASE — освободить зарезервированный регион. При использовании этого флага параметр dwSize должен быть равен нулю.

В программах, где используются команды для работы с виртуальной памятью, должен быть предусмотрен механизм «сбора мусора», обеспечивающий освобождение страниц, которые становятся пустыми. В качестве такого механизма может применяться низкоприоритетный поток, который время от времени просматривает выделенную область и ищет пустые страницы.

Для изменения атрибутов защиты регионов используются функции VirtualProtect и VirtualProtectEx, причем первая позволяет изменять атрибуты защиты в адресном пространстве текущего процесса, а вторая — произвольного. Рассмотрим подробнее функцию VirtualProtect:

```
BOOL VirtualProtect (LPVOID lpvAddress,     // адрес защищаемого блока
      DWORD dwSize,                         // размер защищаемого блока
      DWORD fdwNewProtect,                  // новые флаги защиты
      PDWORD pfdwOldProtect);               // переменная, в которую
                                            // записываются старые флаги
```

Аргументы lpvAddress и dwSize служат для указания диапазона адресов защищаемой памяти. Параметры fdwNewProtect и pfdwOldProtect содержат по одному из флагов защиты: PAGE_NOACCESS, PAGE_READONLY или PAGE_READWRITE. Эти флаги применяются к целым страницам. При установке такого флага изменяется уровень защиты всех страниц, включенных в заданный диапазон, даже тех, что входят в него частично. Параметр pfdwOldProtect возвращает предыдущее состояние первой страницы заданного диапазона.

Функция VirtualProtect работает только с закрепленными страницами. Если какая-либо страница в заданном диапазоне не закреплена, эта функция возвращает ошибку. Однако необязательно, чтобы все страницы диапазона имели одинаковые флаги защиты.

Основное достоинство защиты страниц заключается в их предохранении от ошибок вашей собственной программы. В ряде случаев необходима информация об определенном блоке памяти. Например, перед записью данных в страницу целесообразно проверить, закреплена ли она. Функция VirtualQuery заполняет поля структуры MEMORY_BASIC_INFORMATION информацией о заданном блоке памяти:

```
DWORD VirtualQuery (LPVOID lpvAddress,          // адрес описываемой области
        PMEMORY_BASIC_INFORMATION pmbiBuffer,   // адрес буфера описания
        DWORD dwLength);                        // размер буфера описания

typedefstruct _MEMORY_BASIC_INFORMATION
{
    PVOID BaseAddress;                  // базовый адрес группы страниц
    PVOID AllocationBase;               // адрес наибольшего выделенного блока
    DWORD AllocationProtect;            // первоначальный уровень защиты выделенного блока
    DWORD Regionsize;                   // размер группы страниц в байтах
    DWORD State;                        // закреплена, зарезервирована, свободна
    DWORD Protect;                      // уровень защиты группы
    DWORD Type;                         // тип страниц (всегда MEM_PRIVATE)
} MEMORY_BASIC_INFORMATION;

typedef MEMORY_BASIC_INFORMATION *PMEMORY_BASIC_INFORMATION;
```

Параметр lpvAddress функции VirtualQuery служит для указания произвольного адреса. Любой заданный адрес может принадлежать двум выделенным блокам. Он может являться частью большого диапазона зарезервированных страниц или же входить в меньший блок страниц, которые были закреплены и вновь зарезервированы или защищены. Блок состоит из последовательного набора страниц с одинаковыми атрибутами.

В поле BaseAddress функция VirtualQuery возвращает адрес первой страницы меньшего блока, который содержит ячейку с адресом, заданным аргументом lpvAddress. В поле AllocationBase возвращается адрес большего зарезервированного блока страниц, содержащего данную ячейку. Значение параметра AllocationBase совпадает со значением, возвращаемым функцией VirtualAlloc. Если функция VirtualAlloc применила к указанному диапазону определенный флаг защиты, он может быть возвращен в виде значения поля AllocationProtect (PAGE_NOACCESS, PAGE_READONLY или PAGE_READWRITE).

В остальных полях описывается меньшая подгруппа страниц: указываются ее размер, текущее состояние и флаги защиты. В последнем поле всегда возвращается значение MEM_PRIVATE, которое свидетельствует о том, что другие процессы не могут совместно использовать заданный блок памяти. Наличие данного поля указывает, что впоследствии Microsoft может рассмотреть вопрос о применении других механизмов для совместного использования памяти различными процессами.

Хотя функции GlobalMemoryStatus и GetSystemInfo не входят в набор команд, предназначенных для работы с виртуальной памятью, они возвращают полезную информацию о памяти. Функция GlobalMemoryStatus определяет размер и свободный объем физической памяти, страничного файла и текущего адресного пространства. Функция GetSystemInfo наряду с другой информацией возвращает размер системной физической страницы, а также младший и старший виртуальные адреса, доступные для процессов и DLL-файлов. Обычно эти значения составляют 4 Кб, 0×00010000 и 0×7FFEFFFF соответственно.

Выделенные страницы можно заблокировать в памяти, т. е. запретить их вытеснение в файл подкачки. Для этих целей служит пара функций VirtualLock и VirtualUnlock. Заблокированная страница не может быть перекачана на диск при выполнении программы. Однако если ваша программа в данный момент не выполняется, на диск могут быть перекачаны все страницы, включая заблокированные. В сущности, блокировка выступает гарантией того, что страница будет постоянной и неотъемлемой частью рабочего набора программы. При перегрузке операционной системы менеджер рабочих наборов может ограничить количество страниц, блокируемых одним процессом. Максимальное количество блокируемых страниц для любого процесса колеблется от 30 до 40 и зависит от объема системной памяти и рабочего набора приложения.

Блокировка страниц является нежелательной, поскольку она препятствует работе менеджера виртуальной памяти и затрудняет организацию физической памяти. Как правило, все страницы блокируются только драйверами устройств и другими компонентами системного уровня. Программы, которые должны очень быстро реагировать на системные сигналы, блокируют только часть страниц, чтобы реакция на неожиданный системный сигнал не задерживалась из-за медленного выполнения операций чтения с диска.

```
BOOL VirtualLock (LPVOID lpvAddress,          // начало блокируемой области
      DWORD dwSize);                          // размер блокируемой области

BOOL VirtualUnlock (LPVOID lpvAddress,        // начало области, с которой снимается
                                              // блокировка
      DWORD dwSize);                          // размер области, с которой снимается
                                              // блокировка
```

В системе отсутствует счетчик, определяющий, сколько раз блокировалась виртуальная память. Необязательно, чтобы каждой функции VirtualLock соответствовала функция VirtualUnlock. Например, можно заблокировать три последовательные страницы с помощью трех различных функций, а затем отменить их блокировку с помощью одной функции. Все три страницы предварительно должны быть заблокированы. Однако диапазон, заданный функцией VirtualUnlock, может не совпадать с диапазоном, заданным функцией VirtualLock.

Перед блокировкой память должна быть закреплена. По завершении процесса операционная система автоматически отменяет блокировку всех страниц, которые остались заблокированными. Функция VirtualFree освобождает все страницы, даже заблокированные.

4.4.2. API-функции для программ с проецированием файлов

Как и виртуальная память, проецируемые файлы позволяют резервировать регион адресного пространства и передавать ему физическую память. Различие между этими механизмами состоит в том, что в последнем случае физическая память не выделяется из системного страничного файла, а берется из файла, уже находящегося на диске. Как только файл спроецирован в память, к нему можно обращаться так, как будто он в нее целиком загружен.

Данный механизм может использоваться в следующих случаях [9, 12]:

- для запуска исполняемых файлов (exe) и динамически связываемых библиотек (dll);
- для работы с файлами;
- для одновременного использования одной области данных двумя процессами.

Рассмотрим подробнее каждый из перечисленных механизмов.

Запуск исполняемых файлов и библиотек. Ранее рассматривалась функция CreateProcess для запуска процессов. При использовании данной функции VMM применяется для выполнения следующих действий:

- создание адресного пространства процесса (размером 4Гб);
- резервирование в адресном пространстве процесса региона размером, достаточным для размещения исполняемого файла. Начальный адрес региона определяется в заголовке EXE-модуля. Обычно он равен 0×00400000, но может быть изменен при построении файла параметром /BASE компоновщика;

– отображение исполняемого файла на зарезервированное адресное пространство. Тем самым VMM распределяет физические страницы не из файла подкачки, а непосредственно из EXE-модуля;

– отображение на адресное пространство процесса необходимых ему динамически связываемых библиотек. Информация о необходимых библиотеках находится в заголовке EXE-модуля. Желательное расположение региона адресов описано внутри библиотеки. Visual C++, например, устанавливает по умолчанию адрес 0×10000000. Этот адрес может также изменяться параметром /BASE компоновщика. Если при загрузке выясняется, что данный регион занят, то система попытается переместить библиотеку в другой регион адресов на основе настроечной информации, содержащейся в DLL-модуле. Однако эта операция снижает эффективность системы, и кроме того, если настроечная информация удалена при компоновке библиотеки параметром /FIXED, то загрузка становится вообще невозможной.

При одновременном запуске нескольких приложений Win32 отображает один и тот же исполняемый файл и библиотеки на адресные пространства различных процессов. При этом возникает проблема независимого использования процессами статических переменных и областей данных. Кроме того, изменение данных исполняющейся программой не должно приводить к изменению EXE-файла. Win32 откладывает решение этой проблемы на максимально возможный срок (*Lazy Evaluation*). При этом используется классический механизм отложенного копирования (*copy-on-write* — копирование при попытке записи). Все страницы адресного пространства процесса получают атрибут защиты PAGE_WRITECOPY. При попытке записи в такую страницу возникает исключение нарушения защиты и VMM копирует страницу для обратившегося процесса. В дальнейшем эта страница будет выгружаться в файл подкачки. После копирования происходит рестарт команды, вызвавшей исключение.

Работа с файлами данных, проецируемых в память. Проецирование файла данных в адресное пространство процесса предоставляет мощный механизм работы с файлами. Спроецировав файл на адресное пространство процесса, программа получает возможность работать с ним как с массивом.

Рассмотрим основные функции для проецирования файлов. Для создания объекта ядра «файл» используется функция CreateFile(). Эта функция аналогична функции open() из CRT-библиотеки. Более подробно она будет рассмотрена в следующей главе. Функцией CreateFileMapping() создается объект ядра «проецируемый файл»:

```
HANDLE CreateFileMapping (HANDLE hFile,          // дескриптор файла,
                                                 // открытого функцией CreateFile()
    LPSECURITY_ATTRIBUTES lpAttributes,          // структура SECURITY_ATTRIBUTES
    DWORD fdwProtect,                            // нет доступа, только чтение
                                                 // или чтение / запись
    DWORD dwMaximumSizeHigh,                     // максимальный размер проецируемого
                                                 // файла в верхнем разряде DWORD
    DWORD dwMaximumSizeLow,                      // максимальный размер проецируемого
                                                 // файла в нижнем разряде DWORD
                                                 // если оба параметра равны 0,
                                                 // максимальный размер равен размеру файла
    LPCTSTRlpName);                              // имя проецируемого файла,
                                                 // если NULL — то без имени
```

Если файл был создан ранее, то он может быть открыт функцией OpenFileMapping().

```
HANDLE OpenFileMapping (DWORD dwDesiredAccess,   // режим доступа к отображаемому объекту
    BOOL bInheritHandle,                         // флаг наследования дескрипторов
    LPCTSTR lpName);                             // имя открываемого проецируемого файла
```

Для отображения «проецируемого файла» или его части на адресное пространство процесса применяется функция MapViewOfFile(), для открепления — UnmapViewOfFile():

```
LPVOID MapViewOfFile (HANDLE hFileMappingObject,  // дескриптор
                                                  // проецируемого файла
    DWORD dwDesiredAccess,                        // режим доступа к отображаемому объекту
    DWORD dwFileOffsetHigh,                       // смещение для начала отображения файла
```

```
                                        // в верхнем разряде DWORD
DWORD dwFileOffsetLow,                  // смещение для начала отображения файла
                                        // в нижнем разряде DWORD
                                        // Комбинация высоких и низких смещений
                                        // должна определить смещение в пределах файла,
                                        //которое соответствует степени детализации
                                        // распределения памяти в системе
SIZE_T dwNumberOfBytesToMap);           // количество байтов файла для отображения
BOOL UnmapViewOfFile (LPCVOID lpBaseAddress); //указатель на базовый адрес,
                                        // возвращается функцией MapViewOfFile()
```

Проецирование файла в память выполняется в три этапа:

1) с помощью функции CreateFile() создается объект ядра «файл»;

2) с помощью функции CreateFileMapping() создается объект ядра «проецируемый файл»,при этом используется описатель файла (*handle*), возвращенный функцией CreateFile(). Теперь файл готов к проецированию;

3) с помощью функции MapViewOfFile() производится отображение объекта «проецируемый файл» или его части на адресное пространство процесса.

Для открепления файла от адресного пространства процесса используется функция UnmapViewOfFile(), а для уничтожения объектов «файл» и «проецируемый файл» — функция CloseHandle(). Последовательность кода работы с проецированными файлами следующая:

```
HANDLE hFile, hFileMapping;
PVOID pMassive;
hFile = CreateFile("File Name", ...);
hFileMapping= CreateFileMapping(hFile, ...);
CloseHandle(hFile) ;
pMassive = MapViewOfFile(hFileMapping, ...);
/* Работа с массивом pMassive */
UnmapViewOfFile (pMassive);
CloseHandle(hFileMapping) ;
```

Взаимодействие процессов через общую область данных. Когерентность. Два процесса могут совместно использовать объект «проецируемый файл». При этом при помощи функции MapViewOfFile() каждый процесс проецирует этот объект на свое адресное пространство и использует эту часть адресного пространства как разделяемую область данных. Схема, описывающая данный механизм, представлена на рис. 4.9.

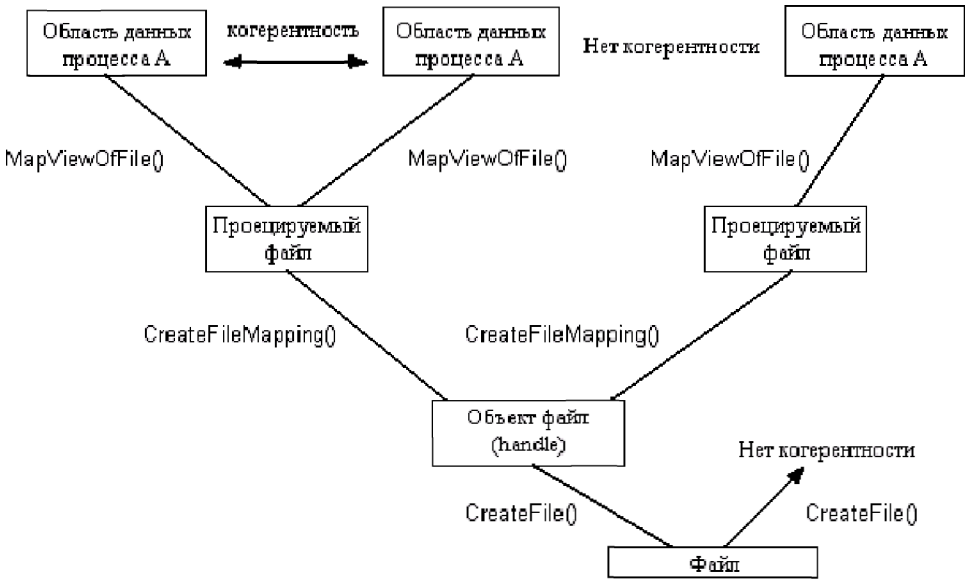

Рис. 4.9. Схема механизма взаимодействия процессов через общую область данных

Общий механизм таков: один процесс создает объект «проецируемый файл» с помощью функции CreateFileMapping() и порождает другой процесс, передавая ему в наследство описатель этого объекта. Дочерний процесс может пользоваться этим описателем наравне с родительским. Проблема состоит только в том, как сообщить дочернему процессу, какой из переданных ему в наследство описателей является описателем «проецируемого файла». Это можно сделать любым способом. Например, передачей параметров при запуске процесса, через переменные среды, передачей сообщения в главное окно процесса и так далее.

Общая область данных может быть создана не только путем проецирования файла, но и путем проецирования части файла подкачки. Для этого в функцию CreateFileMapping() необходимо передать в качестве параметра не описатель ранее открытого файла, а–1. В этом случае необходимо задать размеры выделяемой области. Кроме того, в параметре lpName можно задать имя объекта, которое является глобальным в системе. Если это имя задается в системе впервые, то процессу выделяется новая область данных, а если имя было уже задано, то именованная область данных предоставляется для совместного использования.

Если один процесс меняет разделяемую область данных, то она меняется и для другого процесса. Операционная система обеспечивает когерентность разделяемой области данных для всех процессов. Но для обеспечения когерентности процессы должны работать с одним объектом — «проецируемый файл», а не с одним файлом.

4.4.3. API-функции для программ с выделением динамических областей

Динамическая область памяти, или ***куча (heap)***, представляет собой блок памяти, из которого программа при необходимости выделяет себе более мелкие фрагменты. 16-разрядные Windows-программы выделяют память как из глобальной, так и из локальной динамической области. Последняя работает быстрее, но ее объем ограничен 64 Кб.

Современные ОС семейства Windows со своим плоским адресным пространством аннулируют разницу между понятиями «***глобальный***» и «***локальный***», а также «***дальний***» и «***ближний***», превращая всю память в одну нераздельную динамическую область.

Даже при наличии большого непрерывного адресного пространства иногда целесообразно работать с динамической областью памяти меньшего размера. Резервирование и закрепление виртуальной памяти имеет очевидные преимущества при работе с большими динамическими или разреженными структурами. А как быть с алгоритмами, которые предполагают выделение большого количества мелких блоков памяти? Команды для работы с динамической областью памяти позволяют создавать в адресном пространстве программы одну или несколько локальных куч и выделять из них более мелкие блоки памяти.

Команды для работы с динамической областью памяти удобны тем, что позволяют сосредоточить выделенные блоки в небольшом диапазоне адресного пространства. Группировка выделенных блоков выполняется по следующим причинам [12]:

– она позволяет отделить и защитить группу связанных блоков. Программа, создающая большое количество маленьких блоков одинакового размера, гораздо эффективнее упаковывает память, если блоки следуют последовательно;

– если все узлы связанного списка находятся в одной куче, а узлы двоичного дерева — в другой, то ошибка одного алгоритма в меньшей степени скажется на работе другого алгоритма;

– объекты памяти, работающие совместно, могут быть сгруппированы, что сводит к минимуму подкачку страниц. Несколько адресов, оказавшихся на одной странице памяти, можно прочитать с помощью одной дисковой операции.

Для получения дескриптора кучи «по умолчанию»применяется функция

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

Используя возвращаемый этой функцией дескриптор, можно осуществлять работу с кучей. Память, выделяемая из кучи, ничем не отличается от любой другой памяти. Вы можете самостоятельно организовать работу с кучей, прибегнув к командам управления виртуальной памятью, ведь именно так поступает Windows. Для создания кучи нужно задать ее начальный и максимальный размеры:

```
HANDLE HeapCreate (DWORD dwOptions,          // флаг выделения кучи
    DWORD dwInitialSize,                     // начальный размер кучи
    DWORD dwMaximumSize);                    // максимальный размер кучи
```

«За кулисами» подсистема Win32 реагирует на эту функцию, резервируя блок памяти максимального размера и закрепляя страницы, которые определяют начальный размер кучи. Последующие выделения памяти приводят к увеличению или уменьшению объема кучи. Если для очередного выделения памяти потребуются новые страницы, команды для работы с кучей автоматически закрепят их. Страницы остаются закрепленными до тех пор, пока куча не будет уничтожена или пока сама программа не завершится.

Система не может управлять содержимым локальной кучи, уплотнять кучу или перемещать объекты внутри нее. Поэтому не исключено, что после многократного выделения и освобождения большого количества мелких объектов куча станет фрагментированной. Если при очередном выделении памяти куча достигает максимального размера, все последующие выделения не совершаются. Однако если аргумент dwMaximumSize равен 0, размер кучи ограничивается объемом доступной памяти.

Параметр dwOptions позволяет установить один-единственный флаг — HEAP_NO_SERIALIZE. По умолчанию без этого флага (значение аргумента равно 0) куча не допускает взаимодействия посредством потоков, которые совместно используют дескрипторы памяти. Сериализованная куча препятствует одновременному выполнению нескольких операций с одним дескриптором. Один из потоков блокируется до тех пор, пока другой поток не завершит выполнение своей операции. Сериализация несколько снижает быстродействие. Если в программе реализован только один поток, а также если только один из нескольких потоков программы обращается к куче или если программа самостоятельно обеспечивает защиту кучи (например, путем создания исключающего семафора или критического раздела), то сериализация кучи не требуется.

Функции ***HeapAlloc***, ***HeapReAlloc*** и ***HeapFree*** осуществляют выделение, повторное выделение и освобождение блоков памяти из кучи. Все эти функции в качестве одного из аргументов принимают дескриптор, возвращенный функцией HeapCreate:

```
LPSTR HeapAlloc (HANDLE hHeap,           // дескриптор локальной кучи
    DWORD dwFlags,                       // управляющие флаги
    DWORD dwBytes);                      // количество выделяемых байтов
```

Функция HeapAlloc возвращает указатель блока необходимого размера. Ей могут быть переданы два управляющих флага:

– HEAP_GENERATE_EXCEPTIONS — определяет, как команда будет обрабатывать ошибки. Если флаг не установлен, функция HeapAlloc сообщает об ошибке, возвращая значение NULL. Если флаг установлен, в ответ на любую ошибку функция порождает исключение;

– HEAP_ZERO_MEMORY — дает функции HeapAlloc указание инициализировать новый выделенный блок, заполняя его нулями. При успешном выполнении функция выделяет столько памяти, сколько требуется, или немного больше, чтобы достичь ближайшей границы страницы.

Чтобы узнать точный размер любого блока, вызовите функцию ***HeapSize***. Наряду с байтами блока при каждом выделении будет задействовано несколько дополнительных байтов, необходимых для поддержки внутренней структуры динамической области памяти. Точный размер этого дополнительного кусочка памяти варьируется и составляет в среднем около 16 байтов. Вы должны знать это, потому что без учета подобных «накладных расходов» не сможете выделить из кучи столько блоков, сколько предполагали. Если создать динамическую область памяти объемом 2 Мб и попытаться выделить из нее два блока размером по 1 Мб каждый, то второй блок, скорее всего, не будет выделен.

Изменить размер блока после его выделения позволяет функция HeapReAlloc:

```
LPSTR HeapReAlloc (HANDLE hHeap,         // дескриптор локальной кучи
    DWORD dwFlags,                       // флаги, влияющие на перераспределение памяти
    LPSTR lpMem,                         // адрес блока памяти, размер которого изменяется
    DWORD dwBytes);                      // новый размер выделенного блока памяти
```

Кроме двух флагов, которые функция HeapAlloc использовала для обнуления памяти и генерации исключений, параметр dwFlags функции HeapReAlloc принимает еще один флаг — HEAP_REALLOC_IN_PLACE_ONLY. (Microsoft заботится о том, чтобы мы все понимали, объединив целых пять слов в имени одной-единственной константы.) Этот флаг предотвращает перемещение выделенного блока в более свободную область памяти, выполняемое функцией HeapReAlloc. Если соседние блоки мешают расширению данного блока до требуемого размера, то в результате выполнения функции HeapReAlloc возникает ошибка. Флаг HEAP_REALLOC_IN_PLACE_ONLY обычно применяется в сочетании с другими флагами.

Если выделенный блок памяти больше не нужен, освободите его с помощью функции HeapFree. Если же отсутствует необходимость и в самой куче, освободите ее посредством функции HeapDestroy:

```
BOOL HeapFree (HANDLE hHeap,          // дескриптор локальной кучи
    DWORD dwFlags,                    // не используется (должен быть равен нулю)
    LPSTR lpMem);                     // адрес освобождаемого блока памяти

BOOL HeapDestroy(HANDLE hHeap);
```

Освобождение блока памяти не приводит к отмене закрепления страниц, которые он занимал, однако это пространство становится доступным для последующих выделений из той же динамической области памяти. Функция HeapDestroy освобождает все страницы кучи независимо от наличия в ней выделенных блоков. После выполнения функции HeapDestroy дескриптор hHeap становится недействительным (неопределенным).

Еще один набор функций предназначен для проверки правильности указателей (табл. 4.2).

Таблица 4.2

Функция	Аргумент	Вид проверки
IsBadCodePtr	Указатель функции	Проверяет возможность чтения начала функции
IsBadReadPtr	Указатель блока памяти	Проверяет возможность чтения заданного диапазона адресов
IsBadStringPtr	Указатель строки	Проверяет возможность чтения всех байтов до конца заданной строки
IsBadWritePtr	Указатель блока памяти	Проверяет возможность записи заданного диапазона адресов

Каждая функция из этого набора получает виртуальный адрес и возвращает значение TRUE, если процесс не имеет определенных привилегий доступа.

Глава 5
Принципы разработки программного кода для организации ввода / вывода в современных ОС

§ 5.1. Основы организации ввода / вывода в ПЭВМ

Обеспечение операций ввода / вывода, наравне с обеспечением непосредственно вычислительных операций является одной из основных задач любой операционной системы. Под операциями ввода / вывода в целом понимается обмен данными между памятью и устройствами, внешними по отношению к памяти и процессору, такими как жесткие диски, монитор, клавиатура, мышь, таймер и т. д. Для обеспечения этой возможности используются аппаратные и программные средства. В основе реализации операций ввода / вывода лежит рассмотренный ранее механизм прерываний, который обеспечивает параллельность работы центрального процессора с устройствами ввода / вывода и другими запущенными процессами. Вообще для процессора ввод / вывод информации также является процессом.

Следует отметить, что запись или чтение большого количества информации из адресного пространства ввода / вывода (например, с жесткого диска) приводит к увеличению количества операций ввода / вывода, что, в свою очередь, повышает нагрузку на центральный процессор. Для освобождения процессора от операций последовательного вывода данных из оперативной памяти или последовательного ввода в нее реализован механизм прямого доступа внешних устройств к памяти — ПДП (*Direct Memory Access* — DMA). Для технической реализации этого метода применяется специализированный контроллер прямого доступа к памяти, имеющий несколько спаренных линий — каналов DMA, которые могут подключаться к различным устройствам. В отличие от прерываний, где один номер прерывания мог соответствовать нескольким устройствам, каналы DMA всегда находятся в монопольном владении устройств. В общем виде система ввода / вывода персонального компьютера представлена на рис.5.1 [3].

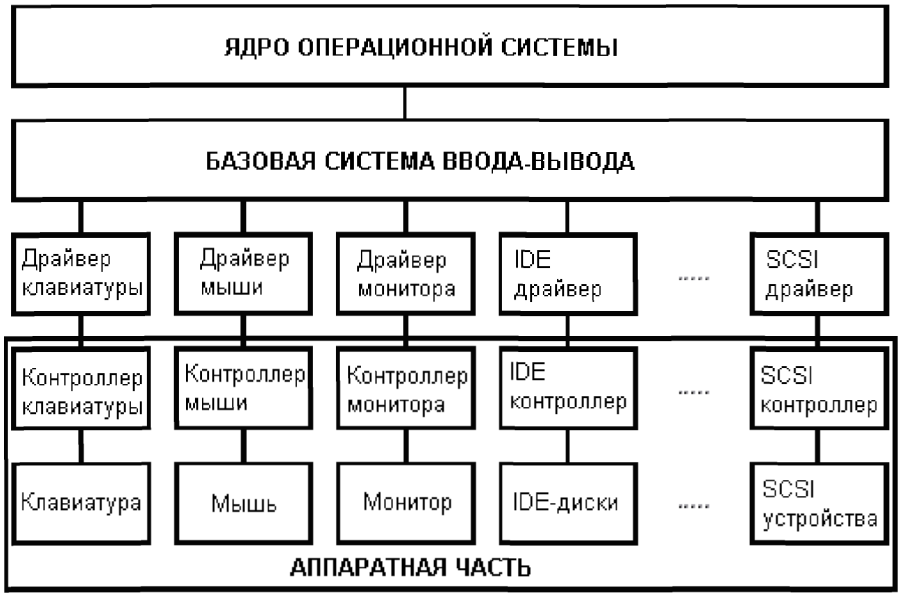

Рис.5.1. Система ввода / вывода ПК

Два нижних уровня этой системы составляют аппаратную часть, т. е. сами устройства, непосредственно выполняющие операции, и их контроллеры, служащие для организации совместной работы устройств и остальной вычислительной системы. Следующий уровень составляют драйверы устройств ввода / вывода, скрывающие от разработчиков операционных систем особенности функционирования конкретных устройств и обеспечивающие четко определенный интерфейс между

аппаратными средствами и следующим уровнем — базовой подсистемой ввода / вывода, которая, в свою очередь, предоставляет механизм взаимодействия между драйверами устройств и ядром операционной системы.

Управление вводом / выводом осуществляется следующим образом. Центральный процессор посылает устройству управления команду выполнить некоторое действие для устройства ввода / вывода. Устройство управления исполняет команду, транслируя сигналы центрального процессора в сигналы, понятные устройству ввода / вывода. К сожалению, быстродействие устройства ввода / вывода намного меньше быстродействия центрального процессора, поэтому сигнал готовности приходилось ожидать достаточно долго, постоянно опрашивая соответствующую линию интерфейса на наличие или отсутствие нужного сигнала. В первых ОС в это время центральный процессор простаивал. В современных ОС во время ожидания сигнала готовности от устройства ввода / вывода центральный процессор переключается на выполнение другой программы. При появлении сигнала готовности генерируется прерывание от соответствующего устройства ввода / вывода (рис.5.2)[2].

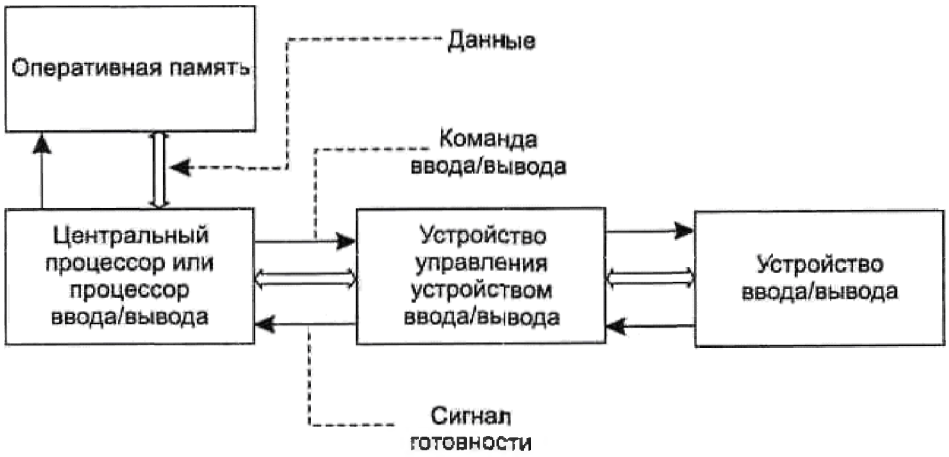

Рис.5.2. Механизм управления вводом / выводом

Все системные вызовы, связанные с осуществлением операций ввода / вывода, по способам реализации взаимодействия процесса и устройства ввода / вывода можно разбить на три группы.

К первой, наиболее привычной для большинства программистов группе относятся ***блокирующиеся системные вызовы***. Как следует из самого названия, применение такого вызова приводит к блокировке инициировавшего его процесса, т. е. процесс переводится операционной системой из состояния исполнения в состояние ожидания. Завершив выполнение всех операций ввода / вывода, предписанных системным вызовом, операционная система переводит процесс из состояния ожидания в состояние готовности. После того как процесс будет снова выбран для исполнения, в нем произойдет окончательный возврат из системного вызова.

Ко второй группе относятся ***неблокирующиеся системные вызовы***. В данном случае системный вызов возвращается немедленно, выполнив предписанные ему операции ввода / вывода полностью, частично или не выполнив совсем (в зависимости от текущей ситуации: состояния устройства, наличия данных и т. д.). В качестве примера такого вызова можно привести периодическую проверку на поступление информации с клавиатуры при выполнении трудоемких расчетов.

К третьей группе относятся ***асинхронные системные вызовы***. Процесс, использовавший асинхронный системный вызов, никогда в нем не блокируется. Системный вызов инициирует выполнение необходимых операций ввода / вывода и немедленно возвращается, после чего процесс продолжает свою регулярную деятельность. Об окончании завершения операции ввода / вывода операционная система впоследствии информирует процесс изменением значений некоторых переменных, передачей ему сигнала или сообщения или каким-либо иным способом.

Необходимо четко понимать разницу между неблокирующимися и асинхронными вызовами. Неблокирующийся системный вызов для выполнения операции чтения вернется немедленно, при этом может быть прочитано как запрошенное количество байтов, так и меньшее их количество или вообще ничего. Асинхронный системный вызов для этой операции также вернется немедленно, но требуемое количество байтов все равно будет прочитано в полном объеме.

Для обеспечения операций ввода/вывода достаточно часто используются механизмы ***буферизации*** и ***кэширования***. Рассмотрим их более подробно.

Под буфером обычно понимается некоторая область памяти для запоминания информации при обмене данных между двумя устройствами, двумя процессами или процессом и устройством. Существуют три основные причины, приводящие к использованию буферов при выполнении операции ввода/вывода[2].

Первая причина буферизации — разные скорости приема и передачи информации, которыми обладают участники обмена. Вторая причина — это разные объемы данных, которые могут быть приняты или получены участниками обмена единовременно. Третья причина буферизации связана с необходимостью копирования информации из приложений, осуществляющих ввод/вывод, в буфер ядра операционной системы и обратно.

Под словом «кэш» (англ. *cash* — наличные) обычно понимают область быстрой памяти, содержащую копию данных, расположенных где-либо в более медленной памяти, и предназначенную для ускорения работы вычислительной системы. В современных системах для этой задачи обычно используется кэш-память центрального процессора. Не следует смешивать понятия буферизации и кэширования. Буфер обычно содержит единственный набор данных, существующий в системе, в то время как кэш, по определению, содержит копию данных, существующих где-нибудь еще. Функции буферизации и кэширования необязательно должны быть локализованы в базовой подсистеме ввода/вывода. Они могут быть частично реализованы в драйверах и даже в контроллерах устройств, скрытно по отношению к базовой подсистеме.

С точки зрения программиста, наиболее часто используемой операцией ввода/вывода является ввод/вывод на жесткие диски. В этом случае обычно речь идет о работе с файлами, т. е. о файловом вводе/выводе. Рассмотрим принципы размещения данных на жестком диске более подробно.

§ 5.2. Общие принципы размещения данных на магнитных дисках

Под ***файлом*** понимают набор данных, организованных в виде совокупности записей одинаковой структуры [2]. В любой ОС работу с файлами обеспечивает ***файловая система***— набор спецификаций и соответствующее им программное обеспечение, которые отвечают за создание, уничтожение, организацию, чтение, запись, модификацию и перемещение файловой информации, а также за управление доступом к файлам и за управление ресурсами, которые используются файлами. Именно файловая система определяет способ организации данных на диске или на каком-нибудь ином носителе данных. В качестве примера можно привести файловую систему FAT, реализация для которой имеется в абсолютном большинстве ОС, работающих в современных ПК.

Как правило, все современные ОС имеют соответствующие ***системы управления файлами***, которые выполняют несколько задач по работе с файлами[2]. Во-первых, через систему управления файлами связываются по данным все системные обрабатывающие программы. Во-вторых, с помощью этой системы решаются проблемы централизованного распределения дискового пространства и управления данными. В-третьих, благодаря использованию той или иной системы управления файлами пользователям предоставляются следующие возможности:

– создание, удаление, переименование (и другие операции) именованных наборов данных (именованных файлов) из своих программ или посредством специальных управляющих программ, реализующих функции интерфейса пользователя с его данными и активно использующих систему управления файлами;

– работа с недисковыми периферийными устройствами как с файлами;

– обмен данными между файлами, устройствами, файлом и устройством;

– работа с файлами с помощью обращений к программным модулям системы управления файлами;

– защита файлов от несанкционированного доступа.

В некоторых ОС может быть несколько систем управления файлами, что обеспечивает им возможность работать с несколькими файловыми системами. С одной стороны, очевидно, что системы управления файлами, будучи компонентом ОС, не зависят от этой ОС, поскольку активно используют соответствующие вызовы API. С другой стороны, системы управления файлами сами дополняют API новыми вызовами. Можно сказать, что основное назначение файловой системы и соответствующей ей системы управления файлами — организация удобного доступа к данным, организованным как файлы. Следует различать понятия ***«файловая система»*** и ***«система управления файлами»***.

Под термином ***«файловая система»***понимаются прежде всего принципы доступа к данным, организованным в файлы. Этот же термин часто используют и по отношению к конкретным файлам, расположенным на том или ином носителе данных. Термин ***«система управления файлами»*** употребляется по отношению к конкретной реализации файловой системы, т. е. это комплекс программных модулей, обеспечивающих работу с файлами в конкретной операционной системе.

Следует заметить, что любая система управления файлами не существует сама по себе — она разработана для работы в конкретной ОС. В качестве примера можно сказать, что всем известная файловая система FAT (*file allocation table*) имеет множество реализаций как система управления файлами [2]. Так, система, получившая это название и разработанная для первых персональных компьютеров, называлась просто FAT (сейчас ее называют FAT-12). Ее создавали для работы с дискетами, и некоторое время она использовалась при работе с жесткими дисками. Потом ее усовершенствовали для работы с жесткими дисками большего объема, и эта новая реализация получила название FAT-16. Это название файловой системы мы используем и по отношению к системе управления файлами самой MS-DOS. Реализацию же системы управления файлами для OS/2, которая использует основные принципы системы FAT, называют super-FAT.Основное ее отличие от других систем — возможность поддерживать для каждого файла расширенные атрибуты. Есть версия системы управления файлами с принципами FAT и для Windows 95/98, для Windows NT и т. д. Другими словами, для работы с файлами, организованными в соответствии с некоторой файловой системой, для каждой ОС должна быть разработана соответствующая система управления файлами. Эта система управления файлами будет работать только в той ОС, для которой она и создана, но при этом она позволит работать с файлами, созданными с помощью системы управления файлами другой ОС, работающей по тем же основным принципам файловой системы.

Для того чтобы загружать ОС с жесткого диска, а далее с ее помощью организовать работу той или иной системы управления файлами, были приняты специальные системные соглашения о структуре диска, которые далее будут рассмотрены более подробно. В любом случае в начале магнитного диска всегда располагается информация о его логической организации и простейшая программа, с помощью которой можно находить и загружать программы загрузки той или иной ОС.

Информация на магнитных дисках размещается и передается блоками. Каждый такой блок называется ***сектором***(*sector*).Секторы расположены на концентрических дорожках поверхности диска. Каждая ***дорожка*** (*track*) образуется при вращении магнитного диска под зафиксированной в некотором предопределенном положении головкой чтения / записи. Жесткий диск (раньше он назывался НЖМД—накопитель на жестких магнитных дисках) содержит как один диск, так и более, но обычно под термином «жесткий диск» понимают весь пакет магнитных дисков.

Группы дорожек одного радиуса, расположенных на поверхностях магнитных дисков, образуют так называемые ***цилиндры*** (*cylinder*). Современные жесткие диски могут иметь по несколько десятков тысяч цилиндров, в то время как на поверхности дискеты число дорожек (цилиндров) составляет, как правило, всего 80 (рис.5.3) [3].

Каждый сектор состоит из ***поля данных*** и ***поля служебной информации***, ограничивающей и идентифицирующей его. Размер сектора, точнее, емкость поля данных устанавливается контроллером или драйвером. Пользовательский интерфейс DOS поддерживает единственный размер сектора — 512 байт. BIOS же предоставляет возможность работы с секторами размером 128, 256, 512 или 1024 байт. Таким образом, если управлять контроллером жесткого диска непосредственно, а не через программный интерфейс более высокого уровня, можно обрабатывать секторы и с другими размерами. Однако в большинстве современных ОС размер сектора выбирается равным 512 байт аналогично DOS.

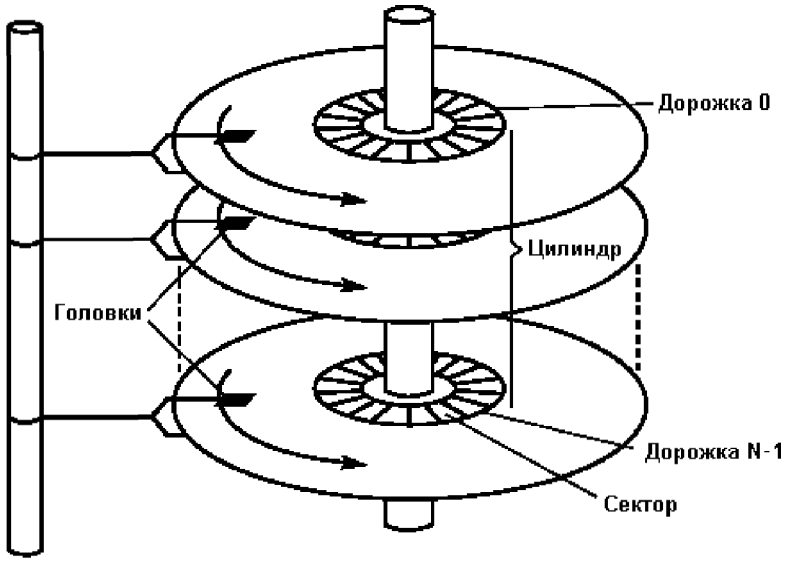

Рис.5.3. Структура хранения информации на жестком диске

При работе диска набор пластин вращается вокруг своей оси с высокой скоростью, подставляя по очереди под ***магнитные головки*** (*head*) соответствующих дорожек все их секторы. Физический адрес сектора на диске определяется с помощью трех координат [*c*–*h*–*s*], где *c* — номер цилиндра, *h* — номер рабочей поверхности диска (магнитной головки), *s* — номер сектора на дорожке. Номер цилиндра *c* лежит в диапазоне 0...*C*–1, где *C* — количество цилиндров. Номер рабочей поверхности диска *h* принадлежит диапазону 0...*N*–1, где *N* — число магнитных головок в накопителе. Номер сектора на дорожке *s* указывается в диапазоне 1...*S*, где *S* — количество секторов на дорожке. Например, триада [1–0–2] адресует сектор 2 на дорожке 0 (обычно верхняя рабочая поверхность) цилиндра 1.

Обмен информацией между оперативной памятью и дисками физически осуществляется только секторами. Вся совокупность физических секторов на винчестере представляет его неформатированную емкость.

При планировании использования жесткого диска естественным параметром является время, которое потребуется для выполнения очередного запроса. Время, необходимое для чтения или записи сектора, можно разделить на две составляющие: время обмена информацией между магнитной головкой и компьютером, которое обычно не зависит от положения данных и определяется скоростью их передачи (*transfer speed*), и время, необходимое для позиционирования головки над заданным сектором — время позиционирования (*positioning time*). Время позиционирования, в свою очередь, состоит из времени, необходимого для перемещения головок на нужный цилиндр — времени поиска (*seek time*), и времени, которое требуется для того, чтобы нужный сектор довернулся под головку, т. е. для задержки на вращение (*rotational latency*). Времена поиска пропорциональны разнице между номерами цилиндров предыдущего и планируемого запросов и их легко сравнивать. Задержка на вращение определяется довольно сложными соотношениями между номерами цилиндров и секторов предыдущего и планируемого запросов, скоростями вращения диска и перемещения головок. Без знания соотношения этих скоростей сравнение становится невозможным. Поэтому естественно, что набор параметров планирования сокращается до времени поиска различных запросов, определяемого текущим положением головки и номерами требуемых цилиндров, а разницей в задержках на вращение пренебрегают.

Существует несколько алгоритмов поиска необходимого сектора [3]. Простейший из них — ***First Come First Served (FCFS)*** — первым пришел, первым обслужен. Все запросы организуются в очередь FIFO и обслуживаются в порядке поступления. Алгоритм прост в реализации, но может приводить к достаточно длительному общему времени обслуживания запросов. Более правильным было бы первоочередное обслуживание запросов, данные для которых лежат рядом с текущей позицией головок, а уж затем далеко отстоящих. Алгоритм ***Short Seek Time First (SSTF)*** — короткое время поиска первым — как раз и исходит из этой позиции. Для очередного обслуживания выби-

рается запрос, данные для которого лежат наиболее близко к текущему положению магнитных головок. Естественно, что при наличии равноудаленных запросов решение о выборе между ними может приниматься исходя из различных соображений, например по алгоритму FCFS. Следует отметить, что этот алгоритм похож на алгоритм SJN для планирования процессов, если за аналог оценки времени выполнения процесса выбирать расстояние между текущим положением головки и положением, необходимым для удовлетворения запроса.

Еще один вид алгоритмов — алгоритмы сканирования (SCAN, C-SCAN, LOOK, C-LOOK). В простейшем из алгоритмов сканирования — SCAN головки постоянно перемещаются от одного края диска до другого, по ходу дела обслуживая все встречающиеся запросы. По достижении другого края направление движения меняется, и все повторяется снова. Если есть информация, что обслужен последний попутный запрос в направлении движения головок, то можно не доходить до края диска, а сразу изменить направление движения на обратное. Полученная модификация алгоритма SCAN получила название LOOK.

Допустим, что к моменту изменения направления движения головки в алгоритме SCAN, т. е. когда головка достигла одного из краев диска, у этого края накопилось большое количество новых запросов, на обслуживание которых будет потрачено достаточно много времени (не забывайте, что надо не только перемещать головку, но еще и передавать прочитанные данные). Тогда запросы, относящиеся к другому краю диска и поступившие раньше, будут ждать обслуживания несправедливо долго. Для сокращения времени ожидания запросов применяется другая модификация алгоритма SCAN — циклическое сканирование. Когда головка достигает одного из краев диска, она без чтения попутных запросов перемещается на другой край, откуда вновь начинает движение в прежнем направлении. Этот алгоритм получил название C-SCAN. По аналогии с C-SCAN реализован и алгоритм C-LOOK.

Жесткий диск может быть разбит на несколько ***разделов*** (*partition*), которые в принципе могут использоваться затем либо одной ОС, либо различными ОС. Причем самым главным является то, что на каждом разделе может быть организована своя файловая система. Однако для организации даже одной единственной файловой системы необходимо определить, по крайней мере, один раздел.

Разделы диска могут быть двух типов — ***primary*** (обычно этот термин переводят как *первичный*) и ***extended*** (*расширенный*). Максимальное число primary-разделов равно четырем. При этом на диске обязательно должен быть по крайней мере один primary-раздел. Если primary-разделов несколько, то только один из них может быть активным. Именно загрузчику, расположенному в активном разделе, передается управление при включении компьютера и загрузке операционной системы. Остальные primary-разделы в этом случае считаются невидимыми, скрытыми (*hidden*).

Согласно спецификациям на одном жестком диске может быть только один ***extended***-раздел, который, в свою очередь, может быть разделен на большое количество подразделов — ***логических дисков*** (*logical*). В этом смысле термин «первичный» следует признать не совсем удачным переводом слова *primary*: это слово можно перевести и как «простейший, примитивный». В данном случае становится понятным и логичным термин «*extended*».

Один из primary-разделов должен быть ***активным***, и именно с него должна загружаться программа загрузки операционной системы, или так называемый ***менеджер загрузки***, назначение которого—загрузить программу загрузки ОС из какого-нибудь другого раздела и уже с ее помощью загружать операционную систему. Поскольку до загрузки ОС система управления файлами работать не может, то следует использовать для указания упомянутых загрузчиков исключительно абсолютные адреса в формате [*c–h–s*].

По физическому адресу [0–0–1] на винчестере располагается ***главная загрузочная запись*** (*master boot record* — MBR), содержащая ***внесистемный загрузчик*** (*non-system bootstrap* — NSB), а также ***таблицу разделов*** (*partition table* — PT). Эта запись занимает ровно один сектор и размещается в памяти, начиная с адреса 0:7C00h, после чего управление передается коду, содержащемуся в этом первом секторе магнитного диска. Таким образом, в самом первом (стартовом) секторе физического жесткого диска находится не обычная запись *boot record*, как на дискете, а *master boot record*.

MBR является основным средством загрузки с жесткого диска, поддерживаемым BIOS. В MBR находятся три важных элемента:

– программа начальной загрузки (*non-system bootstrap*). Именно она запускается BIOS после успешной загрузки в память первого сектора с MBR. Она, очевидно, не превышает 512 байт, и ее хватает только на то, чтобы загрузить следующую, чуть более сложную программу, обычно стартовый сектор операционной системы, и передать ей управление;

– таблица описания разделов диска (*partition table*), которая располагается в MBR по смещению 0x1BE и занимает 64 байта;

– сигнатура MBR. Последние два байта MBR должны содержать число AA55h. По наличию этой сигнатуры BIOS проверяет, что первый блок был загружен успешно. Использование такой сигнатуры выбрано не случайно. Ее успешная проверка позволяет установить, что все линии передачи данных могут передавать и нули, и единицы.

Таблица ***partition table*** описывает размещение и характеристики имеющихся на винчестере разделов. Можно сказать, что эта таблица разделов — одна из наиболее важных структур данных на жестком диске. Если она повреждена, то не только не будет загружаться операционная система (или одна из операционных систем, установленных на винчестере), но перестанут быть доступными и данные, расположенные на винчестере, особенно если жесткий диск был разбит на несколько разделов. Упрощенно структура MBR представлена в табл. 5.1 [2].

Таблица 5.1

Смещение (Offset)	Размер (Sise), байт	Содержимое (Contents)
0	446	Программа анализа Partition Table и загрузки System Bootstrap с активного раздела жесткого диска
+1BEh	16	Partition 1 entry (Описатель раздела)
+1CEh	16	Partition 2 entry
+1DEh	16	Partition 3 entry
+1EEh	16	Partition 3 entry
+1FEh	16	Сигнатура (AA55h)

Из табл. 5.1 следует, что в начале этого сектора располагается программа анализа таблицы разделов и чтения первого сектора из активного раздела диска. Сама таблица *partition table* располагается в конце MBR, и для описания каждого раздела в этой таблице отводится по 16 байтов. Первым байтом в элементе раздела идет флаг активности раздела boot indicator (0 — неактивен, 128 (80H) — активен). Он служит для определения, является ли раздел системным загрузочным и есть ли необходимость производить загрузку операционной системы с него при старте компьютера. Активным может быть только один раздел. За флагом активности раздела следует байт номера головки, с которой начинается раздел. За ним следует два байта, означающие соответственно номер сектора и номер цилиндра загрузочного сектора, где располагается первый сектор загрузчика операционной системы. Затем следует кодовый идентификатор SystemID (длиной в один байт), указывающий на принадлежность данного раздела к той или иной операционной системе и установке на нем соответствующей файловой системы. В табл. 5.2 приведены некоторые наиболее известные идентификаторы [2].

За байтом кода операционной системы расположен байт номера головки конца раздела, за которым идут два байта — номер сектора и номер цилиндра последнего сектора данного раздела. В табл. 5.3 представлен формат элемента таблицы разделов [2].

Номер сектора и номер цилиндра секторов в разделах занимают соответственно по 6 и 10 бит. На рис.5.4 представлен формат записи, содержащей номера сектора и цилиндра.

Загрузчик *non-system bootstrap* служит для поиска с помощью *partition table* активного раздела, копирования в оперативную память компьютера загрузчика system bootstrap из выбранного раздела и передачи ему управления, что позволяет осуществить загрузку ОС. Вслед за сектором MBR размещаются собственно разделы (рис.5.5).

Таблица 5.2

System ID	Тип раздела	System ID	Тип раздела
00	Empty («пустой» раздел)	41	PPC PreP Boot
01	FAT12	42	SFS
02	XENIX root	4D	QNX4.x
03	XENIX usr	4E	QNX 4.x 2nd part
04	FAT16 (<32 Мбайт)	4F	QNX 4.x 3nd part
05	Extended	50	OnTrack DM
06	FAT16	51	OnTrack DM6 Aux
07	HPFS/NTFS	52	CP/M
08	AIX	53	OnTrack DM6
09	AIX bootable	54	OnTrack DM6
0A	OS/2 Boot Manager	55	EZ Drive
0B	Win95 FAT32	56	Golden Bou
0C	Win95 FAT32 LBA	5C	Priam Edisk
0E	Win95 FAT16 LBA	61	Speed Stor
0F	Win95 Extended	64	Novell Netware
10	OPUS	65	Novell Netware
11	Hidden FAT12	75	PC/IX
12	Compaq diagnost	80	Old Minix
14	Hidden FAT16 (<32 Мбайт)	82	Linux swap
16	Hidden FAT16	83	Linux native
17	Hidden HPFS/NTFS	84	OS/2 hidden C:
18	AST Windows swap	85	Linux Extended
1B	Hidden Win95 Fat	86	NTFS volume set
1C	Hidden Win95 Fat	A5	BSD/386
1E	Hidden Win95 Fat	A6	Open BSD
24	NEC DOS	A7	Next Step
3C	Partition Magic	EB	Be OS
40	Venix 80286		

Таблица 5.3

Название записи элемента Partition Table	Длина, байт
Флаг активности раздела	1
Номер головки начала раздела	1
Номер сектора и номер цилиндра загрузочного сектора раздела	2
Кодовый идентификатор операционной системы	1
Номер головки конца раздела	1
Номер сектора и цилиндра последнего сектора раздела	2
Младшее и старшее двухбайтовое слово относительного номера начального сектора	4
Младшее и старшее двухбайтовое слово размера раздела в секторах	4

Биты номера цилиндра								Биты номера сектора							
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Рис.5.4. Формат записи с номером сектора и цилиндра

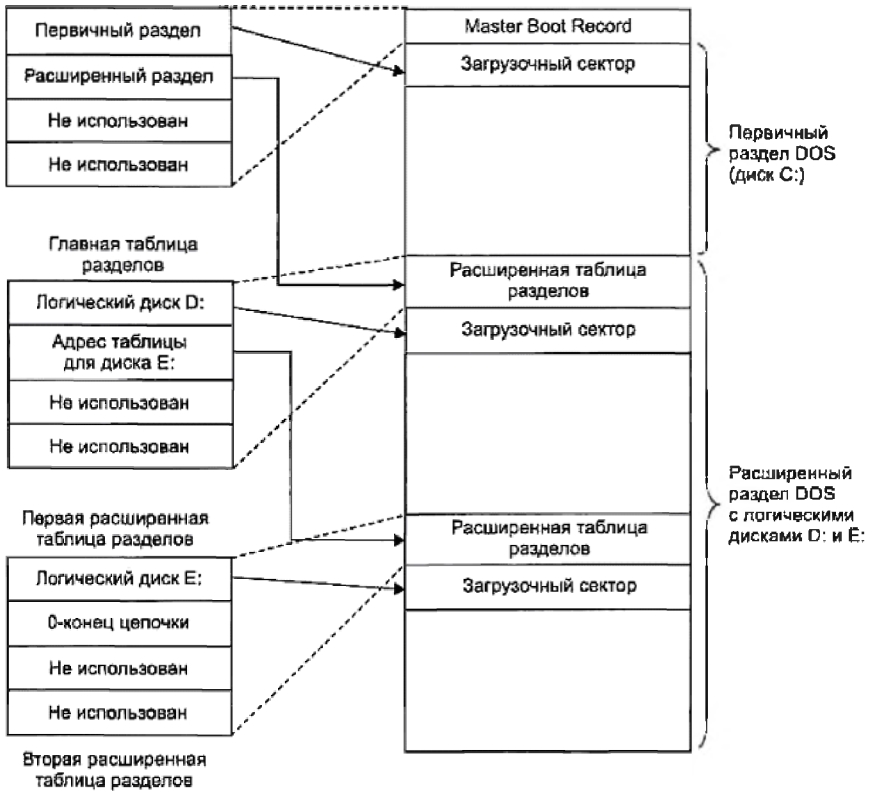

Рис.5.5. Разбиение диска на разделы

В процессе начальной загрузки сектора MBR, содержащего таблицу *partition table*, работают программные модули BIOS. Начальная загрузка считается корректно выполненной только в том случае, когда таблица разделов содержит допустимую информацию.

В MS-DOS в первичном разделе может быть сформирован только один логический диск, а в расширенном — любое их количество. Каждый логический диск «управляется» своим логическим приводом. Каждому логическому диску на винчестере соответствует своя (относительная) ***логическая*** нумерация. Физическая же адресация жесткого диска — сквозная. Первичный раздел DOS включает только ***системный логический диск*** без каких-либо дополнительных информационных структур. Расширенный раздел DOS содержит вторичную запись MBR (*secondary* MBR, SMBR), в состав которой вместо *partition table* входит таблица логического диска (LDT, *logical disk table*), аналогичная ей. Таблица LDT описывает размещение и характеристики раздела, содержащего единственный логический диск, а также может специфицировать следующую запись SMBR. Следовательно, если в расширенном разделе DOS создано *K* логических дисков, то он содержит *K* экземпляров SMBR, связанных в список. Каждый элемент этого списка описывает соответствующий логический диск и ссылается (кроме последнего) на следующий элемент списка.

На основании сказанного процесс загрузки ОС состоит из следующих этапов [2]. Первоначально вызывается прерывание BIOSINT 19h, что инициализирует процедуру начальной загрузки (*bootstrap loader*). Эта процедура определяет первое готовое устройство из списка разрешенных и доступных (гибкий или жесткий диск, CD-ROM, ZIP-drive, сетевой адаптер и др.) и пытается загрузить с него в ОП программу первоначальной загрузки — *non-system bootstrap*. Далее этот загрузчик определяет на диске активный раздел, загружает с него загрузчик ОС — *system bootstrap* и передает ему управление. И, наконец, загрузчик ОС загружает необходимые файлы операционной системы и передает ей управление. Далее ОС выполняет инициализацию своих программных и аппаратных средств. При этом, как правило, добавляются новые сервисы, вызываемые через механизм программных прерываний, а также расширяются или обновляются некоторые сервисы BIOS.

§ 5.3. Принципы разработки программного кода для файлового ввода / вывода

5.3.1. API-функции для организации ввода / вывода

Обычно если программист хочет открыть файл, то он использует один из стандартных вызовов библиотеки языка программирования (например, в языке С это функция *fopen*). В большинстве языков программирования предусмотрены достаточно удобные высокоуровневые средства работы с файлами. Однако в некоторых ситуациях требуется открыть файл и работать с ним на уровне операционной системы, не используя высокоуровневые функции. Например, прямое обращение к операционной системе может потребоваться в случае, если вы намерены использовать асинхронный ввод / вывод.

Системная функция, с помощью которой осуществляется открытие файла, называется ***CreateFile***. В зависимости от флагов, которые программист передает в качестве параметров, она может либо действительно создать новый файл, либо открыть уже существующий. В любом случае вызов CreateFile() создает дескриптор файла и возвращает его вызвавшей программе, которая может использовать этот дескриптор для дальнейшей работы с файлом. Описание аргументов функции приведено в табл. 5.4 [10]:

```
HANDLE CreateFile(LPCTSTR lpFileName,
        DWORD dwDesiredAccess,
        DWORD dwShareMode,
        LPSECURITY_ATTRIBUTES lpSecurityAttributes,
        DWORD dwCreationDispostion,
        DWORD dwFlagsAndAttributes,
        HANDLE hTemplateFile);
```

Таблица 5.4

Аргумент	Описание
lpFileName	Указатель на имя файла или устройства
DesiredAccess	Устанавливает вид доступа к объекту. Используются флаги GENERIC_READ (чтение), GENERIC_WRITE (запись) или оба при помощи оператора логического сложения. Значение 0 указывает на необходимость проверки возможности доступа
dwShareMode	Набор битовых флагов, указывающий на режим совместного доступа к объекту. Если значение dwShareMode равно 0, совместный доступ к объекту запрещен. Флаги:FILE_SHARE_DELETE — совместное удаление FILE_SHARE_READ — совместное чтение FILE_SHARE_WRITE — совместная запись
lpSecurityAttributes	Указатель на структуру SECURITY_ATTRIBUTES, которая определяет, будет ли создаваемый дескриптор наследоваться дочерними процессами. Если аргумент lpSecurityAttributes имеет значение NULL, дескриптор не будет наследоваться
dwCreationDisposition	Указывает, каким образом следует создать (или открыть) файл. Допускается использовать следующие значения: CREATE_NEW (создать новый файл; если файл существует, функция не срабатывает), CREATE_ALWAYS (создать новый файл; если файл существует, он перезаписывается), OPEN_EXISTING (открыть файл; если файл не существует, функция не срабатывает), OPEN_ALWAYS (открыть файл; если файл не существует, он создается) или TRUNCATE_EXISTING (открыть файл и сделать его равным нулю; если файл не существует, функция не срабатывает)
dwFlagsAndAttributes	Набор атрибутов и флагов, которыми должен обладать файл. Например, если требуется, чтобы новый файл был скрытым, используйте значение FILE_ATTRIBUTE_HIDDEN. Другие возможные значения флагов перечислены в табл. 5.5
hTemplateFile	Содержит дескриптор с доступом GENERIC_READ. Это дескриптор шаблонного файла, атрибуты которого (включая расширенные) будут присвоены создаваемому файлу

Наиболее интересный аргумент функции CreateFile() — предпоследний, шестой аргумент, который является набором флагов доступа к файлу и атрибутов файловой системы. Используя различные комбинации флагов (табл. 5.5), можно настроить доступ к файлу самым причудливым образом. Например, можно приказать системе удалить файл сразу же после того, как он будет закрыт, или объявить, что файл будет использоваться для ***перекрывающегося*** (*overlapped*) доступа (перекрывающийся доступ — базовый метод осуществления асинхронного ввода / вывода). Помимо прочего флаги позволяют управлять механизмом кэширования.

Таблица 5.5

Флаг	Значение
FILE_FLAG_WRITE_THROUGH	Приказывает Windows осуществлять немедленную запись данных на диск. При этом возможно использование кэша
FILE_FLAG_NO_BUFFERING	Приказывает системе открыть файл без использования кэширования или буферизации, что дает максимальную производительность
FILE_FLAG_RANDOM_ACCESS	Оповещает систему о том, что доступ к файлу осуществляется случайным образом. Система может использовать это обстоятельство для оптимизации кэширования файла
FILE_FLAG_SEQUENTIAL_SCAN	Оповещает систему о том, что доступ к файлу осуществляется последовательно от начала к концу файла. Система может использовать это обстоятельство для оптимизации кэширования файла
FILE_FLAG_OVERLAPPED	Приказывает системе инициализировать объект для перекрывающегося ввода / вывода
FILE_FLAG_DELETE_ON_CLOSE	Приказывает системе уничтожить файл сразу же после того, как все его дескрипторы будут закрыты
FILE_FLAG_BACKUP_SEMANTICS	Указывает на то, что файл предназначен для операций резервного копирования или восстановления из резервной копии. Операционная система разрешает вызывающему процессу любой доступ к файлу при условии, что вызывающий процесс обладает привилегиями SE_BACKUP_NAME и SE_RESTORE_NAME
FILE_FLAG_POSIX_SEMANTICS	Доступ к файлу осуществляется в соответствии с правилами POSIX. При этом разрешается использовать несколько различных файлов, имена которых отличаются только регистром букв. Такие файлы разрешается создавать только в системах, поддерживающих подобную схему именования файлов
FILE_FLAG_OPEN_REPARSE_POINT	Подавляет поведение, свойственное для точек грамматического разбора (*reparse points*) файловой системы NTFS. Когда файл открывается, вызывающему процессу возвращается его дескриптор независимо от того, работоспособен ли фильтр, контролирующий точку грамматического разбора, или нет. Этот флаг не может использоваться совместно с флагом CREATE_ALWAYS
FILE_FLAG_OPEN_NO_RECALL	Информирует систему о том, что вызывающее приложение запрашивает данные, хранящиеся в файле, однако файл может продолжать оставаться на удаленном носителе данных. Этот флаг используется удаленными системами хранения данных или совместно с системой Hierarchical Storage Management

Еще одной особенностью функции CreateFile() является обслуживание сменных носителей информации. Если носитель данных в настоящий момент недоступен (гибкий диск вытащили из дисковода), то операционная система выведет на экран диалоговое окно с сообщением об ошибке. Если вы хотите избежать этого (обычно, убедившись в отсутствии гибкого диска, приложение принимает меры самостоятельно), необходимо обратиться к функции SetErrorMode() и передать ей в качестве параметра флаг SEM_FAILCRITICALERRORS. В этом случае при отсутствии гибкого диска в дисководе система не будет отображать на экране каких-либо сообщений об ошибках.

Обладая дескриптором файла, можно выполнить чтение или запись данных при помощи соответственно функций ReadFile() и WriteFile(). Эти функции работают стандартным образом: в качестве параметров они принимают дескриптор файла, указатель на буфер, длину буфера и указатель на переменную, в которой будет сохранено количество прочитанных или записанных байт. Если используется перекрывающийся ввод/вывод, то в качестве одного из аргументов необходимо передать указатель на структуру OVERLAPPED. Для обычных файлов указатель на эту структуру всегда равен NULL:

```
BOOL ReadFile(HANDLE hFile;              // дескриптор файла
        LPVOID lpBuffer;                 // буфер для временного хранения прочитанных данных
        DWORD dwBytesToRead;             // количество байтов, которые должны быть прочитаны
        LPDWORD lpdwBytesRead;           // возвращает количество прочитанных байтов
        LPOVERLAPPED lpOverlapped); // поддержка асинхронного ввода / вывода
BOOL WriteFile(HANDLE hFile,             // дескриптор файла
        CONSTVOID *lpBuffer,             //указывает данные, которые должны быть записаны в файл
        DWORD dwBytesToWrite,            // количество записываемых байтов
        LPDWORD lpdwBytesWritten,        // возвращает количество записанных байтов
        LPOVERLAPPED lpOverlapped); // задает поддержку асинхронного ввода / вывода
```

Чтобы закрыть файл, используется функция CloseHandle(). Эту функцию можно использовать не только для закрытия дескрипторов файлов. С ее помощью можно закрыть любой другой дескриптор:

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

Для примера далее приведен исходный код простой программы, использующей файловые операции для отображения на экране содержимого одного или нескольких файлов. Вызов CreateFile() открывает файл для чтения. После этого вызов GetStdHandle() возвращает дескриптор стандартного вывода. Затем при помощи функций ReadFile() и WriteFile() блоками по 4 Кбайт происходит передача содержимого файла с жесткого диска в стандартный поток вывода. Передача продолжается до тех пор, пока программа не обнаружит конец файла:

```
#include<windows.h>
void MB(char *s)                         // Для удобства использования MessageBox
{
        MessageBox(NULL, s, NULL, MB_OK | MB_ICONSTOP);
}
void docat(char *fname)                  // основная подпрограмма
{
        HANDLE f=CreateFile (fname, GENERIC_READ, 0, NULL, OPEN_EXISTING, 0, NULL);
        HANDLE out=GetStdHandle(STD_OUTPUT_HANDLE);
        if (f==INVALID_HANDLE_VALUE)
        {
                MB(«Не могу открыть файл»);
                exit(1);
        }
char buf[4096];
unsigned long n;
do
{
        unsigned long wct;
        if (!ReadFile(f, buf, sizeof(buf), &n, NULL))
                break;
        if (n)
        WriteFile(out, buf, n, &wct, NULL);
}
while (n==sizeof(buf));                  // Если EOF, это условие не выполняется
        CloseHandle(f);
}
void main(int argc, char *argv[])
{
        if (argc==1)
        {
```

```
                                        // утилита cat – Ошибки фактически не обрабатываются
                                        // Любая ошибка вызывает аварийное завершение программы
            MB("Usage: cat FILENAME [FILENAME ...]");
            Exit(9);
        }
                                        // Обработать все указанные файлы
        while (--argc) docat(*++argv);
        exit(0);
}
```

Вот так происходит работа с файлами на уровне Win32API. Обычно при обращении к ReadFile() или WriteFile() работа программного потока приостанавливается до того момента, когда операция чтения / записи будет завершена.

Благодаря механизмам асинхронного ввода / вывода программа может осуществлять ввод / вывод данных и одновременно с этим выполнять какие-либо другие полезные действия, например сложные математические вычисления. Следует напомнить, что ***асинхронный ввод / вывод*** — это механизм, позволяющий программе осуществлять обмен данными с каким-либо устройством ввода / вывода и одновременно с этим выполнять какую-либо другую полезную работу [1]. Например, программа отдает команду на чтение данных из файла, а затем, не ожидая завершения чтения данных, приступает к выполнению математических вычислений. Когда данные из файла перемещаются в оперативную память, программа завершает вычисления и приступает к обработке прочитанных данных, причем это лишь один из множества возможных сценариев использования асинхронного ввода / вывода.

Если вы разрабатываете программу, принимающую данные из нескольких разных источников, вам также может потребоваться асинхронный ввод / вывод. Предположим, что вы разрабатываете терминальную программу, которая читает данные как с клавиатуры, так и из последовательного порта. По мере поступления данных программа должна немедленно осуществлять их обработку. В этой ситуации также чрезвычайно удобно использовать механизмы асинхронного ввода / вывода.

5.3.2. Механизмы асинхронного ввода / вывода

Асинхронный ввод / вывод можно реализовать несколькими разными способами. Проще всего создать ***новый программный поток*** и осуществлять весь ввод / вывод средствами этого потока. Вместо этого можно использовать ***перекрывающийся ввод / вывод*** или ***порты завершения ввода / вывода***. Еще одной технологией, с помощью которой можно реализовать асинхронный ввод / вывод, является ***технология проецирования файлов***, рассмотренная ранее.

Использование потоков. Если вы не хотите, чтобы работа программного потока приостановилась для выполнения процедур ввода / вывода, создайте второй программный поток и поручите выполнение ввода / вывода ему. Тогда первый программный поток сможет продолжить работу в то время, когда второй поток будет выполнять медлительные процедуры, связанные с вводом / выводом. Например, для реализации терминальной программы вы можете использовать два потока: один читает команды, вводимые с клавиатуры, а второй следит за поступлением команд через последовательный порт. Оба потока записывают любые принятые ими данные в циклический буфер. Третий поток проверяет содержимое каждого из буферов и выполняет действия в соответствии с поступающими командами. При этом скорость поступления данных через последовательный порт не повлияет на скорость обработки команд, вводимых с клавиатуры.

При реализации подобного простого подхода обычно возникает проблема: третий поток должен постоянно следить за изменением содержимого циклических буферов, чтобы вовремя обнаружить появление в любом из них нового символа. Таким образом, для его работы требуется дополнительное процессорное время. Более сложная реализация может предусматривать использование ***событий*** (рассмотрены ранее при изучении многопоточности). Когда один из потоков ввода принимает символ, он переводит специально предназначенное для этого событие в сигнальное состояние. При этом третий поток может использовать для слежения за состоянием события функцию WaitForMultipleObjects(), т. е. он не будет расходовать слишком много процессорного времени. Для

слежения за состоянием консоли можно использовать непосредственно дескриптор консоли. Этот дескриптор переходит в сигнальное состояние в момент, когда в потоке ввода консоли появляются данные, ожидающие чтения. Если вместо события программа следит за состоянием дескриптора консоли, можно ограничиться только двумя потоками.

Если терминальная программа обладает собственным циклом обработки системных сообщений (к консольным приложениям это обычно не относится), вы можете использовать другой подход. Потоки, осуществляющие ввод / вывод, могут передавать основному окну программы сообщения, определенные пользователем. Основная программа будет реагировать на эти сообщения так же, как она реагирует на любые системные сообщения.

Все перечисленные стратегии очень просты в реализации, однако они недостаточно эффективны. Создание потоков — относительно ресурсоемкая операция. Кроме того, максимальное количество потоков, одновременно работающих в системе, ограниченно. Конечно, это не будет проблемой для простой консольной программы, однако если ваше приложение должно поддерживать интенсивный асинхронный ввод / вывод, то применение для этой цели большого количества потоков может оказаться неприемлемым.

Перекрывающийся ввод / вывод. Значительно более эффективным способом реализации асинхронного ввода / вывода в Windows является перекрывающийся ввод / вывод. В Windows эта разновидность ввода / вывода поддерживается в отношении фактически всех типов файлов. В частности, вы можете применить перекрывающийся ввод / вывод в отношении дисковых файлов, коммуникационных портов, именованных каналов(*named pipes*) и сетевых сокетов. В общем и целом если в отношении чего-либо можно применить операции ReadFile() и WriteFile(), то значит, при этом допускается использовать перекрывающийся ввод / вывод.

Прежде чем вы сможете осуществлять в отношении файла операции перекрывающегося ввода / вывода, необходимо открыть файл при помощи функции CreateFile() с использованием флага FILE_FLAG_OVERLAPPED. Если этот флаг не указать, перекрывающийся ввод / вывод будет невозможен. Если флаг установлен, вы не сможете использовать файл обычным способом. При обмене данными с файлом будет использоваться перекрывающийся ввод / вывод.

Выполняя перекрывающийся ввод / вывод при помощи функций ReadFile() и WriteFile(), в качестве одного из аргументов вы передаете этим функциям указатель на структуру OVERLAPPED. Первые два 32-битных слова этой структуры зарезервированы системой для внутреннего использования. Вторые два слова могут содержать 64-битное смещение, указывающее на позицию файла, в которой будет осуществляться чтение или запись данных.Ввод / вывод выполняется асинхронно, поэтому не существует гарантий, что данные из файла будут извлекаться (или записываться) последовательно байт за байтом. Таким образом, при выполнении перекрывающегося ввода / вывода не существует понятия текущей позиции. При выполнении любой операции вы просто указываете смещение в файле. Если вы работаете с потоками данных (например, последовательный порт или сокет), понятие смещения теряет смысл, поэтому система игнорирует соответствующее поле структуры OVERLAPPED. Последнее поле структуры OVERLAPPED — это дескриптор события. Впрочем, его можно не указывать (присваивать NULL).

Когда происходит обращение к функции асинхронного ввода / вывода, система может завершить процедуру ввода / вывода немедленно. Например, вы можете приказать системе выполнить чтение одного байта из последовательного устройства и записать этот байт в 30-байтный буфер, ожидающий ввода. В этом случае функция ввода / вывода завершает свою работу так, словно вы и не использовали при этом механизмов перекрывающегося ввода / вывода. Если функция вернула ненулевое значение, значит, операция ввода / вывода успешно завершена, а передача данных произошла фактически точно так же, как если бы вы не использовали перекрывающегося ввода / вывода.

Особенности асинхронного ввода / вывода проявляются в случае, если функция возвращает значение FALSE. Нулевое значение, возвращенное функцией ввода / вывода, означает, что либо процедура ввода / вывода находится в стадии выполнения, либо произошла какая-либо ошибка. Чтобы уточнить, чем, собственно, завершился вызов функции, необходимо вызвать функцию GetLastError(). Если этот вызов вернул значение ERROR_IO_PENDING, значит, запрос на ввод / вывод либо ожидает своей очереди, либо находится в стадии выполнения. Любое другое значение означает, что произошла ошибка.

Если вы обратились к функции ввода / вывода и получили подтверждение того, что запрос на ввод / вывод находится в процессе исполнения, то рано или поздно захотите узнать, когда же все-таки осуществление ввода / вывода завершится. Если вы осуществляете чтение данных, то не сможете использовать данные из буфера чтения до того момента, пока не завершится операция чтения. Если вы осуществляете запись, то рано или поздно должны убедиться в том, что процедура записи успешно завершена.

Наиболее простой способ проверить текущее состояние запроса на ввод / вывод — это воспользоваться функцией GetOverlappedResult():

```
BOOL GetOverlappedResult(HANDLE hFile,          // дескриптор файла или устройства
    LPOVERLAPPED lpOverlapped,                  // поддержка асинхронного ввода / вывода
    LPDWORD lpNumberOfBytesTransferred,         // количество переданных байт
    BOOL bWait );                               // флаг ожидания
```

В качестве аргументов этой функции необходимо передать дескриптор файла, указатель на структуру OVERLAPPED, использованную при обращении к функции ввода / вывода, указатель на переменную, в которую будет занесено количество переданных байт, и флаг, определяющий, должен ли вызов ждать завершения процедуры ввода / вывода или ему следует немедленно вернуть управление вызывающей программе, чтобы сообщить ей о текущем состоянии запроса на ввод / вывод. Если этот флаг имеет значение TRUE и операция ввода / вывода все еще не завершена, вызов будет ожидать до тех пор, пока операция ввода / вывода завершится или произойдет ошибка. Если флаг ожидания имеет значение FALSE, вызов возвращает ошибку (нулевое значение).

Если вызов вернул нулевое значение, а флаг ожидания равен FALSE, то необходимо обратиться к GetLastError(). Если этот вызов вернул значение ERROR_IO_INCOMPLETE, значит, процедура ввода / вывода все еще не завершена. Любое другое значение свидетельствует о том, что в ходе выполнения ввода / вывода произошла ошибка. Чтобы прервать выполнение операции ввода / вывода, следует использовать функцию ***CanselIo***:

```
BOOL CancelIo(HANDLE hFile);    // дескриптор файла
```

Эта функция отменяет выполнение любых запросов на ввод / вывод, инициированных для некоторого дескриптора файла текущим потоком. Вызов CanselIo() работает только в отношении процедур перекрывающегося ввода / вывода. Очевидно, что если поток инициировал традиционную процедуру ввода / вывода, то он не сможет обратиться к каким-либо другим вызовам до тех пор, пока процедура ввода / вывода не будет завершена. Если вы прервали выполнение запроса на ввод / вывод при помощи вызова CanselIo(), то соответствующая операция ввода / вывода завершается с сообщением об ошибкеRROR_OPERATION_ABORTED (это значение можно получить при помощи вызова GetLastError()).

В случае если было инициировано слишком большое количество запросов на асинхронный ввод / вывод, функция ReadFile() может завершиться возвратом значения ERROR_INVALID_USER_BUFFER или ERROR_NOTENOUGH_MEMORY.

В некоторых ситуациях для осуществления операций перекрывающегося ввода / вывода удобнее использовать специально предназначенные для этого функции ReadFileEx() и WriteFileEx(). Эти функции не могут использоваться для обычного ввода / вывода. В качестве ***дополнительного аргумента*** каждый из этих вызовов принимает указатель на функцию, которая будет вызвана в момент завершения операции ввода / вывода:

```
BOOL ReadFileEx(HANDLE hFile,
    LPVOID lpBuffer,
    DWORD nNumberOfBytesToRead,
    LPOVERLAPPED lpOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
BOOL WriteFileEx(HANDLE hFile,
    LPCVOID lpBuffer,
    DWORD nNumberOfBytesToWrite,
    LPOVERLAPPED lpOverlapped,
    LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine);
```

Однако для того чтобы произошло обращение к функции завершения, поток, обратившийся к ReadFileEx() или WriteFileEx(), должен находиться в «настороженном» состоянии. Чтобы перейти в «настороженное» состояние, необходимо использовать вызовы WaitForSingleObject() или SleepEx(). Находясь в «настороженном» состоянии, поток не выполняет никаких действий, а просто ждет, когда в результате завершения операции ввода / вывода произойдет обращение к функции завершения.

Все эти особенности перекрывающегося ввода / вывода несколько обескураживают. Перекрывающийся ввод / вывод удобен во многих ситуациях, однако он не настолько асинхронный, как этого хотелось бы программистам. Чтобы реализовать действительно асинхронный ввод / вывод, следует создать отдельный программный поток и в его рамках использовать обычные функции ввода / вывода. В качестве альтернативы при разработке некоторых программ можно использовать порты завершения ввода / вывода (о них речь пойдет чуть позже).

Если традиционная функция чтения данных в процессе осуществления операции ввода встречает символ EOF, то она тем или иным образом оповещает об этом вызывающую программу. В частности, традиционный вызов ReadFile() устанавливает количество прочитанных байт равным нулю. Если символ EOF встречается в процессе выполнения перекрывающегося вызова ReadFile(), то вызов возвращает ошибку. В этом случае значение, возвращаемое функцией GetLastError(), будет равно ERROR_HANDLE_EOF. Символ EOF может встретиться непосредственно при обращении к функции ReadFile() или позже, в процессе выполнения операции ввода / вывода.

Порты завершения ввода / вывода. Перекрывающийся ввод / вывод обладает массой ограничений. Фактически при использовании перекрывающегося ввода / вывода для обмена данными с некоторым объектом (файлом или устройством) используется отдельный программный поток. Например, если вы планируете использовать перекрывающийся ввод / вывод при разработке сетевого сервера, обслуживанием каждого из клиентов будет заниматься отдельный программный поток. Такая схема будет отлично работать в случае, если число клиентов небольшое. Однако вряд ли удастся использовать подобный подход для обеспечения работы высокопроизводительного сервера, обслуживающего одновременно десятки тысяч клиентов. Этого можно достичь, если использовать ***порты завершения ввода / вывода*** (I/O *completion ports*) [1].

Порт завершения ввода / вывода напоминает очередь. В эту очередь заносятся уведомления о том, что та или иная процедура ввода / вывода завершена. Любой поток может проверить очередь и отреагировать на любое из этих уведомлений. Прежде чем приступить к использованию порта завершения ввода / вывода, необходимо создать его при помощи вызова CreateIoCompletionPort():

```
HANDLE CreateIoCompletionPort (HANDLE FileHandle,      // дескриптор файла
    HANDLE ExistingCompletionPort,                     // дескриптор создаваемого
                                                       // (или открываемого) порта завершения
    ULONG_PTR CompletionKey,                           // ключ завершения,
                                                       // вставляемый в каждый пакет
    DWORD NumberOfConcurrentThreads);                  //количество подключаемых потоков
```

В качестве одного из аргументов эта функция принимает дескриптор файла, открытого для перекрывающегося ввода / вывода. Как только дескриптору файла ставится в соответствие порт завершения ввода / вывода, при успешном завершении любой операции ввода / вывода в очередь порта завершения будет внесено уведомление о завершении этой операции. Вызов CreateIoCompletionPort() можно использовать не только для создания нового порта завершения, но также и для назначения уже существующему порту другого дескриптора файла.

При обращении к функции CreateIoCompletionPort() вы можете указать ключ — 32-битное значение, которое система добавляет в каждое из уведомлений об успешном осуществлении операций ввода / вывода, связанных с указанным файлом. Последний аргумент функции CreateIoCompletionPort() служит для передачи этой функции максимально допустимого количества потоков, которые будут реагировать на появление уведомлений о завершении ввода / вывода. Обычно значение этого аргумента равно нулю.

Любой поток может получить информацию о завершении ввода / вывода при помощи функции GetQueuedCompletionStatus(). Эта функция возвращает количество байт, переданных в процессе ввода / вывода, ключ завершения (32-битное число, установленное при обращении к CreateIoCompletion) и структуру OVERLAPPED, использованную при инициализации процедуры ввода / вывода:

```
BOOL GetQueuedCompletionStatus(HANDLE CompletionPort,      //дескриптор порта
    LPDWORD lpNumberOfBytes,                               // количество переданных байт
    PULONG_PTR lpCompletionKey,                            //указатель на ключ завершения,
                                                           // (если он объявлен ранее)
    LPOVERLAPPED *lpOverlapped,                            //указатель на Overlapped
    DWORD dwMilliseconds);                                 // время ожидания пакета
```

При создании порта завершения ввода / вывода вы можете не ставить ему в соответствие каких-либо дескрипторов файлов. В этом случае порт завершения может использоваться в качестве механизма связи между несколькими потоками одного процесса. При помощи функции PostQueuedCompletionStatus() любой поток может поместить в очередь порта завершения уведомление о завершении ввода / вывода. При этом необходимо указать количество переданных байт, ключ завершения и указатель на структуру OVERLAPPED, которая будет возвращена ожидающему потоку вызовом функции GetQueuedCompletionStatus():

```
BOOL PostQueuedCompletionStatus(HANDLE CompletionPort,
    DWORD dwNumberOfBytesTransferred,
    ULONG_PTR dwCompletionKey,
    LPOVERLAPPED lpOverlapped);
```

Безусловно, во многих ситуациях вы можете обойтись и без портов завершения ввода / вывода, однако очень часто этот механизм оказывается весьма удобным.

Как уже было сказано, еще одной «экзотической технологией», поддерживаемой ОС Windows для асинхронного ввода / вывода, являются ***файлы, отображаемые в память***. Менеджер виртуальной памяти Windows позволяет программе работать с файлом таким образом, будто этот файл полностью загружен в оперативную память компьютера. На самом деле это не так. Менеджер виртуальной памяти загружает в оперативную память компьютера только фрагменты файла (страницы). Чтобы отобразить файл на оперативную память, необходимо его открыть при помощи функции CreateFile() и передать дескриптор файла CreateFileMapping(). Подробно этот механизм был рассмотрен ранее.

В результате анализа всех возможностей файлового ввода / вывода возникает вопрос: обязаны ли программисты всегда использовать CreateFile() и остальные связанные с этим функции для того, чтобы работать с файлами в Windows? Конечно, нет. Если вы ориентируетесь на традиционные методики работы с файлами, то можете продолжать использование функции fopen(), MFC-класса CFile, потоков C++ или другого традиционного метода, к которому привыкли. Однако если вы желаете воспользоваться специальными возможностями Windows (такими, как, например, перекрывающийся ввод / вывод), то придется применить системный вызов CreateFile() и связанные с этим дальнейшие действия, так как большинство библиотек компиляторов не поддерживают технологий, являющихся особенностью ОС Windows.

Глава 6
Принципы разработки программного кода для работы с реестром ОС Windows

§ 6.1. Структура и особенности реестра Windows

Реестр (*registry*) — понятие сравнительно новое. Приложения, работавшие в среде Windows 3.x и DOS, обычно создавали файлы инициализации, в которые записывалась информация о параметрах конфигурации, пользовательских настройках, флагах состояния и других характеристиках приложения. Так, файл Win.INI содержит данные о конфигурации системы, файл Reg.DAT — о связях между приложениями и расширениями документов, а также о OLE-объектах, файл System.INI — об аппаратной конфигурации. В других INI-файлах хранятся сведения о параметрах отдельных приложений.

Все файлы инициализации записываются в ASCII-формате, и доступ к ним можно осуществить с помощью любого текстового редактора. Конечно, разработчики операционной системы и приложений вовсе не стремятся к тому, чтобы пользователи самостоятельно «ковырялись» в файлах инициализации. В идеале эти файлы должны изменяться только приложениями. Следует отметить, что файлы инициализации часто портятся, могут быть случайно удалены. Порой самые невинные изменения одного из них приводят к необратимым последствиям, нарушая работу всей системы. Короче говоря, файлы инициализации — это вечная головная боль программистов.

С появлением Windows NT (версия 3.1) на смену множеству разрозненных файлов инициализации, в том числе System.INI, Win.INI, Protocol.INI, Config.SYS и Autoexec.BAT, пришел единый системный реестр, хотя даже сегодня некоторые приложения все еще записывают свои параметры в INI-файлы [12]. При запуске приложения многие из этих параметров записываются в реестр, т. е. хранятся в двух форматах. Следовательно, при перезагрузке системы конфигурацию, которая была нарушена в результате повреждения или ошибочного удаления INI-файла, можно восстановить.

Если говорить честно, системный реестр — это тоже головная боль программистов, хотя и не такая сильная, как файлы инициализации. Разработчику приложений просто необходимо знать, как используется реестр. В данном параграфе рассмотрим методы создания разделов реестра, записи и чтения данных о конфигурации системы.

Реестр Windows представляет собой иерархическую структурированную базу данных, в которой содержится информация о всех системных настройках, путях, переменных, а также параметрах приложений, ранее располагавшаяся в разрозненных файлах инициализации [12].

В Windows 9x физически реестр располагался в двух файлах в каталоге **\SystemRoot\:** System.dat и User.dat. Первый файл — System.dat содержит информацию, общую для всех пользователей данного компьютера: параметры устройств, настройки некоторых программ и др. Второй файл — User.dat является специфичным для каждого пользователя, сохраняя его настройки. Оба файла имеют атрибут «Скрытый». Если на компьютере зарегистрировано несколько пользователей, то в папке **\SystemRoot\Profiles** хранятся в отдельных каталогах «личные» файлы каждого: содержимое Рабочего стола, главное меню и др., а также часть User.dat системного реестра. В папке **\SystemRoot** тоже хранится файл User.dat, содержащий настройки системы по умолчанию (т. е. те, которые будут загружены, если в ответ на запрос имени пользователя и пароля нажать клавишу Esc). При загрузке компьютера после ввода имени пользователя файл User.dat из каталога этого пользователя вместе с System.dat и User.dat из каталога Windows загружается в память и «склеивается» с ними в единое целое. В Windows NT (в версиях до Windows NT 4) реестр располагается в папках **\SystemRoot\System32\config** и в **\UserProfile**Ntuser.dat. Реестр Windows NT старше версии 4.0 имеет более сложную конфигурацию, которая будет рассмотрена позже.

Формат базы данных имеет ряд преимуществ по сравнению с форматом простого списка. Данные реестра защищены как от непреднамеренной порчи, так и от случайного удаления. Кроме того,

поскольку вся системная информация находится в одном месте, вы можете легко и быстро получить доступ к интересующим вас данным, не занимаясь утомительным поиском в нескольких разрозненных источниках [12]. В Windows NT первых версий каждый раз после успешной загрузки компьютера автоматически создавались резервные копии файлов System.DAT и User.DAT под именами System.DA0 и User.DA0. Эти файлы могли быть использованы для восстановления поврежденных файлов реестра. К сожалению, в Windows 9x эта операция поддерживалась. В современных Windows восстановление реестра существенно упрощено и выполняется с помощью специальной утилиты.

Напрямую данные системного реестра недоступны. Их нельзя просматривать и редактировать так же просто, как записи в INI-файлах. Для работы с реестром предназначен редактор реестра — RegistryEditor (утилита RegEdit.EXE или RegEd32.EXE).

Windows не инсталлирует утилиту RegEdit или RegEd32 в программную группу и не создает для нее ярлыка в стартовом меню для быстрого доступа. Но это не упущение разработчиков Microsoft, а специальная мера защиты реестра от неквалифицированных пользователей. Некорректные операции с реестром могут необратимым образом повредить всю систему.

Прежде чем изменять системную информацию, необходимо сделать резервную копию реестра. Для этой цели в меню ***Registry*** имеется команда ***ExportRegistryFile***,осуществляющая резервное копирование всего реестра. Выберите удобное место и легко распознаваемое имя для записи резервной копии реестра, например корневой каталог и имя Registry.REG. Не следует изменять расширение REG, иначе вы столкнетесь с трудностями при восстановлении данных. Имейте в виду, что файл записывается в ASCII-формате и будет занимать около 4–5 Мб.

Теперь, если изменение системной информации приведет к возникновению каких-либо проблем, вы всегда сможете восстановить первоначальный реестр, либо импортировав резервную копию с помощью редактора реестра, либо просто выполнив двойной щелчок на имени экспортированного файла. В случае серьезной ошибки поврежденные файлы системного реестра (System.DAT и User.DAT) можно заменить их резервными копиями, восстановив тем самым исходную конфигурацию.

Несмотря на необходимость соблюдать все меры предосторожности, иногда все-таки возникает потребность в прямом доступе к системному реестру. Простой просмотр сложной и многокомпонентной иерархической структуры может оказаться непростой и довольно утомительной задачей. Единственным альтернативным методом поиска определенной записи (точное расположение которой неизвестно) является применение находящихся в меню ***Edit*** команд ***Find*** и ***FindNext*** с указанием искомой строки или какого-либо другого известного значения.

С точки зрения разработчика, более важно не уметь редактировать реестр, а знать, каким образом он используется приложениями, т. е. как они записывают данные в реестр и читают из него системную информацию. Для этого необходимо разобраться со структурой реестра.

Реестр представляет собой иерархическую систему разделов и подразделов. Нагляднее будет представить структуру реестра в виде дерева, разделы и подразделы которого аналогичны каталогам и подкаталогам файловой системы (именно таким образом реестр представлен в окне ***Registry Editor***). ***Раздел*** (*key*) любого уровня может содержать один или несколько фрагментов информации (параметров), подобно тому, как любой каталог может содержать один или несколько файлов. В разделе могут также содержаться подразделы.

Основу структуры реестра Windows 9x и NT первых поколений составляют шесть разделов верхнего уровня (табл. 6.1), предоставляющих доступ ко всем остальным подразделам.

Windows XP в отличие от своих предшественниц не имеет ограничения по размеру реестра. Реестр Windows XP содержит пять основных разделов:

- **HKEY_CLASSES_ROOT**— хранится информация о зарегистрированных классах, расширениях документов;
- **HKEY_CURRENT_USER** — хранится информация о текущей пользовательской конфигурации, внешнем виде рабочего стола, сетевых настройках;
- **HKEY_LOCAL_MACHINE** — хранится информация о системной и аппаратной конфигурации;
- **HKEY_USERS** — хранится информация обо всех зарегистрированных пользователях;

• **HKEY_CURRENT_CONFIG** — текущая аппаратная конфигурация.

Таблица 6.1

Раздел	Содержимое
HKEY_CLASSES_ROOT	Информация о зарегистрированных классах, расширениях документов и т. д. На самом деле данный раздел является ссылкой на подраздел HKEY_LOCAL_MACHINE\SOFTWARE\Classes
HKEY_CURRENT_USER	Информация, о текущей пользовательской конфигурации, внешнем виде рабочего стола, сетевых настройках и т. д. Реально этот раздел представляет собой ссылку на подраздел HKEY_USERS для текущего пользователя
HKEY_LOCAL_MACHINE	Информация о системной (аппаратной) конфигурации, в том числе о глобальных настройках приложений, поддерживаемых устройствах, схемах подключения устройств и т. д.
HKEY_USERS	Информация обо всех пользователях, зарегистрированных в локальной системе (см. раздел HKEY_CURRENT_USER)
HKEY_CURRENT_CONFIG	Ссылка на подраздел HKEY_LOCAL_MACHINE\Config\xxxxxx, где *xxxxxx*— числовой параметр, соответствующий текущей аппаратной конфигурации
HKEY_DYN_DATA	Динамические данные для устройств типа Plug-and-Play и VdX-драйверов виртуальных устройств

Элементы реестра начиная с Windows 2000 хранятся в виде атомарной структуры. Реестр разделяется на составные части (ульи — *hives*), или кусты. Ульи хранятся на диске в виде файлов. Некоторые ульи, такие как HKLM\HARDWARE, не сохраняются в файлах, а создаются при каждой загрузке, т. е. являются изменяемыми (*volatile*). При запуске системы реестр собирается из ульев в единую древовидную структуру с корневыми разделами. В табл. 6.2 перечислены ульи реестра и их местоположение на диске (данные справедливы для NT старше версии 4.0).

Таблица 6.2

Улей	Расположение
HKLM\SYSTEM	SystemRoot\system32\config\system
HKLM\SAM	SystemRoot\system32\config\SAM
HKLM\SECURITY	SystemRoot\system32\config\SECURITY
HKLM\SOFTWARE	SystemRoot\system32\config\software
HKLM\HARDWARE	Изменяемый улей
HKLM\SYSTEM\Clone	Изменяемый улей
HKU\<SID пользователя>	USER PROFILE\ntuser.dat
HKU\<SID пользователя>Classes	USER PROFILE\Local Settings\Application Data\Microsoft\Windows\UsrClass.dat
HKU\.DEFAULT	SystemRoot\system32\config\default

Кроме данных файлов имеется ряд вспомогательных со следующими расширениями:

• **LOG** — журнал транзакций, в котором регистрируются все изменения реестра;

• **SAV** — копии ульев в том виде, в котором они находились после завершения текстовой фазы установки.

Реестр является настоящей базой данных, поэтому в нем используется технология восстановления, похожая на технологию в NTFS. Уже упомянутые LOG-файлы содержат журнал транзакций, который хранит все изменения. Благодаря этому реализуется атомарность реестра, т. е. в данный момент времени в реестре могут быть либо старые значения, либо новые, даже после сбоя.

Как видим, в отличие от NTFS здесь обеспечивается сохранность не только структуры реестра, но и данных. К тому же реестр поддерживает такие параметры NTFS, как управление избирательным доступом и аудит событий — система безопасности пронизывает всю современную NT .

Внутри корневого раздела находятся разделы и подразделы, которые аналогичны каталогам и подкаталогам жесткого диска. Раздел может содержать информацию или данные. Раздел и подраздел могут содержать 0, 1 или несколько ***параметров***, параметр по умолчанию, а также 0 или несколько подразделов. Каждый параметр имеет имя, тип и значение. Три части параметра реестра всегда располагаются в определенном порядке:

[RegistrySizeLimit] [REG_DWORD] [0x8000000]. Имя, тип данных, значение.

Записи реестра, называемые ***параметрами***, могут содержать данные различных типов (табл. 6.3).

Таблица 6.3

Тип данных	Описание
REG_BINARY	Необработанные двоичные данные. Большинство сведений об аппаратных компонентах хранится в виде двоичных данных и выводится в редакторе реестра в шестнадцатеричном формате
REG_DWORD	Данные, представленные целым числом. Многие параметры служб и драйверов устройств имеют этот тип и отображаются в двоичном, шестнадцатеричном или десятичном форматах
REG_SZ	Текстовая строка в формате, удобном для восприятия человеком. Значениям, представляющим собой описания компонентов, обычно присваивается именно этот тип данных
REG_EXPAND_SZ	Расширяемая строка данных. Эта строка представляет собой текст, содержащий переменную, которая может быть заменена при вызове со стороны приложения
REG_MULTI_SZ	Многострочное поле. Значения, которые фактически представляют собой списки текстовых строк в формате, удобном для восприятия человеком, обычно имеют именно этот тип данных; строки разделены символом NULL
REG_FULL_RESOURCE_DESCRIPTOR	Последовательность вложенных массивов, разработанная для хранения списка ресурсов аппаратного компонента или драйвера
REG_LINK	Символьная ссылка в кодировке Unicode
REG_RESOURCE_LIST	Список ресурсов драйверов устройств
REG_NONE	Тип значения не определен

Среди них чаще всего используются данные, записанные в двоичном, десятичном, шестнадцатеричном и текстовом форматах. Однако используемый тип данных оказывает незначительное влияние на операции чтения и записи благодаря применению API-функций RegQueryValueEx() и RegSetValueEx(), о которых мы поговорим далее.

§ 6.2. API-функции для работы с реестром Windows

Windows API содержит 26 функций, предназначенных для работы с реестром (включая 5 функций, которые служат для совместимости с Windows 3.x). Из всех этих функций в приложениях обычно используется не более трех-четырех. Все API-функции, применяемые для выполнения операций с реестром, перечислены в табл. 6.4.

API-функции RegSetKeySecurity(), RegGetKeySecurity(), а также параметры безопасности некоторых других функций можно применять только в среде Windows NT.

Таблица 6.4

Функция	Выполняемое действие
1	*2*
RegCloseKey	Закрывает (освобождает) дескриптор указанного раздела без обновления реестра: см. также функцию RegFlushKey (для всех версий)
RegConnectRegistry	Устанавливает связь с заранее определенным дескриптором реестра на другом (удаленном или сетевом) компьютере. Для выполнения операций с локальным реестром не нужна (для всех версий)
RegCreateKey	Создает заданный раздел или, если таковой уже существует, открывает его
RegCreateKeyEx	Создает заданный раздел или, если таковой уже существует, открывает его (для Windows 98/95/NT)
RegDeleteKey	Удаляет заданный раздел и все его подразделы (для Windows 98/95). Удаляет заданный раздел, но не удаляет его подразделы (для Windows NT)
RegDeleteValue	Удаляет значение именованного параметра из заданного раздела реестра (для всех версий)
RegEnumKey*	Составляет список подразделов заданного открытого раздела
RegEnumKeyEx	Составляет список подразделов заданного открытого раздела (для Windows 98/95/NT)
RegEnumValue	Составляет список параметров, которые содержатся в заданном открытом разделе (для Windows 98/95/NT)
RegFlushKey	Записывает все атрибуты заданного раздела в реестр (для всех версий)
RegGetKeySecurity	Возвращает дескриптор безопасности заданного открытого раздела (для Windows NT)
RegLoadKey	Создает подраздел раздела HKEY_USER или HKEY_LOCAL_MACHINE, прежде чем копировать в этот подраздел информацию из указанного файла (для всех версий)
RegNotifyChangeKeyValue	Извещает вызывающую функцию об изменении атрибутов или содержимого открытого раздела. Не извещает об удалении раздела (для всех версий)
RegOpenKey*	Открывает указанный раздел, но не создает его
RegOpenKeyEx	Открывает указанный раздел, но не создает его (для Windows 98/95/NT)
RegQueryInfoKey**	Читает информацию об открытом подразделе, в том числе о его размере, номере, классе, атрибуте безопасности и т. д. (для Windows 98/95/NT)
RegQueryMultipleValues	Возвращает тип и данные для списка имен параметров, связанных с открытым разделом (для всех версий)
RegQueryValue*	Читает значение, связанное с безымянным параметром указанного открытого раздела
RegQueryValueEx**	Читает значение, связанное с безымянным параметром указанного открытого раздела (для Windows 98/95/NT)
RegReplaceKey	Заменяет файл, в котором хранятся копии разделов и подразделов, новым файлом. При перезапуске системы разделы и подразделы получат значения параметров, заданные в новом файле (для Windows 98/95/NT)
RegRestoreKey	Читает данные реестра из указанного файла, копируя их в подразделы заданного раздела (для всех версий)
RegSaveKey	Сохраняет указанный раздел, подразделы и параметры в файле (для всех версий)
RegSetKeySecurity	Устанавливает атрибут безопасности открытого раздела (для Windows NT)

Окончание табл. 6.4

1	2
RegSetValue*	Связывает безымянный параметр (текстовый) с определенным разделом
RegSetValueEx**	Сохраняет данные в заданном параметре открытого раздела и может присвоить разделу дополнительное значение и информацию о типе (для Windows 98/95/NT)
RegUnLoadKey	Выгружает (удаляет) из реестра указанный раздел, все его подразделы и параметры (для Windows 98/95/NT)

* Функция обеспечивает совместимость с Windows 3.x.
** Функции, которые чаще всего используются приложениями.

Рассмотрим определение некоторых основных функций работы с реестром. Функция RegQueryInfoKey() определена следующим образом:

```
LONG RegQueryInfoKey (HKEY hKey,          // дескриптор запрашиваемого раздела
      LPTSTR lpClass,                     // строка описания класса
      LPDWORD lpcbClass,                  // размер буфера строки
      LPDWORD lpReserved,                 // зарезервирован
      LPDWORD lpcSubKeys,                 // количество подразделов
      LPDWORD lpcbMaxSubKeyLen,           // размер самого длинного имени подраздела
      LPDWORD lpcbMaxClassLen,            // размер самой длинной строки описания класса
      LPDWORD lpcValues,                  // количество параметров раздела
      LPDWORD lpcbMaxValueNameLen,        // размер самого длинного имени параметра
      LPDWORD lpcbMaxValueLen,            // размер самого длинного значения параметра
      LPDWORD lpcbSecurityDescriptor,     // длина дескриптора безопасности
      PFILETIME lpftLastWriteTime);       // последнее время записи
```

В табл. 6.5 описаны аргументы функции RegQueryInfoKey. Единственным аргументом, необходимым для любого запроса, является hKey, задающий раздел реестра, информацию о котором вы хотите получить.

Таблица 6.5

Тип данных	Аргумент	Описание
HKEY	hKey	Дескриптор запрашиваемого раздела (обязательный аргумент)
LPTSTR	lpClass	Строка описания класса, связанного с данным разделом
LPDWORD	lpcbClass	Размер строкового буфера класса (должен задаваться вместе с аргументом lpClass)
LPDWORD	lpReserved	Зарезервирован; всегда должен иметь значение NULL
LPDWORD	lpcSubKeys	Количество подразделов, которые находятся в запрашиваемом разделе
LPDWORD	lpcbMaxSubKeyLen	Размер самого длинного имени подраздела
LPDWORD	lpcbMaxClassLen	Размер самой длинной строки описания для классов, связанных с подразделами
LPDWORD	lpcValues	Количество параметров, содержащихся в данном разделе
LPDWORD	lpcbMaxValueNameLen	Размер самого длинного из имен параметров, содержащихся в данном разделе
LPDWORD	lpcbMaxValueLen	Наибольшая длина параметра среди параметров, содержащихся в данном разделе
LPDWORD	lpcbSecurityDescriptor	Длина дескриптора безопасности (только для Windows NT)
PFILETIME	lpftLastWriteTime	Последнее время записи (только для Windows NT)

Посредством функции RegEnumKey можно осуществить циклический запрос имен подразделов. Указанная функция определяется следующим образом:

```
LONG RegEnumKey (HKEY hKey,          // дескриптор запрашиваемого раздела
        DWORD dwIndex,                   // номер запрашиваемого подраздела
        LPTSTR lpName,                   // адрес буфера для имени подраздела
        DWORD cbName);                   // размер буфера для имени подраздела
```

Каждый раз при вызове функции RegEnumKey со следующим значением номера dwIndex в переменной lpName возвращается имя другой подстроки. Когда список подстрок заканчивается, функция RegEnumKey выдает код ошибкиERROR_NO_MORE_ITEMS, свидетельствующей о том, что подразделов больше нет. Порядок, в котором эти элементы возвращаются, не имеет значения. До тех пор, пока мы не зададим индекс, для которого нет подраздела с таким номером, функция будет возвращать значение ERROR_SUCCESS, а в массиве lpName будет храниться имя соответствующего подраздела. При получении любого результирующего значения, кроме ERROR_SUCCESS и ERROR_NO_MORE_ITEMS, должно выводиться сообщение об ошибке.

Функция RegQueryValueEx(), производящая чтение информации из раздела реестра, определяется следующим образом:

```
LONG RegQueryValueEx (HKEY hKey,     //дескриптор раздела для чтения
        LPCTSTR lpValueName,             // адрес имени параметра
        DWORD lpReserved,                // зарезервирован
        DWORD lpType,                    //тип данных
        BYTE lpData,                     // значение параметра
        DWORD lpcbData);                 // размер значения параметра
```

Функция RegSetValueEx, записывающая информацию в реестр, определяется следующим образом:

```
LONG RegSetValueEx (HKEY hKey,       //дескриптор раздела для записи
        LPCTSTR lpValueName,             // адрес имени параметра
        DWORD lpReserved,                // зарезервирован
        DWORD dwType,                    // флаг типа параметра
        CONST BYTE *lpData,              // адрес значения параметра
        DWORD cbData);                   // размер значения параметра
```

Рассмотрим аргументы функций RegQueryValueEx() и RegSetValueEx():

– hKey — идентифицирует текущий открытый раздел или один из следующих предопределенных дескрипторов: HKEY_CLASSES_ROOT, HKEY_CURRENT_USER, HKEY_LOCAL_MACHINE или HKEY_USERS;

– lpValueName — указывает строку, содержащую имя записываемого параметра. Если ранее данный параметр отсутствовал, то теперь он добавляется в текущий раздел; если он имеет значениеNULLили указывает на пустую строку, а переменная dwType имеет тип REG_SZ, то параметру lpValueName присваивается имя (*default*) (в Windows 3.x с помощью функции RegSetValue);

– lpReserved — зарезервирован, должен иметь значение NULL;

– lpType — указывает на переменную, которая может иметь один из типов данных, представленных в табл. 6.3;

– dwType — идентифицирует тип записываемых данных. Все возможные идентификаторы типов перечислены в табл. 6.3.

– lpData — указывает на буфер, содержащий данные, которые должны быть записаны (этот параметр может иметь значение NULL, если данные не требуются);

– cbData — содержит размер (в байтах) буфера, заданный параметром lpData. При использовании типов данных REG_SZ, REG_EXPAND_SZ и REG_MULTI_SZ в параметре cbData нужно учитывать наличие завершающего нулевого символа. Размер буфера ограничивается доступной памятью.

Глава 7
Принципы разработки программного кода для организации безопасности в ОС Windows

§ 7.1. Технологии безопасности, реализованные в Windows

Материал, рассмотренный в данной главе посвящен средствам защиты, реализованным в Windows NT, так как болеее устарелые Windows 9x и Windows 3x таких средств практически не имели. Следует правда отметить, что последняя из них Windows 98 все же определенные средства защиты имела:

- поддержка защищенных каналов;
- поддержка интеллектуальных карточек (*smartcards*);
- встроенная поддержка API-функций, предназначенных для выполнения операций шифрования;
- встроенная поддержка алгоритма Authenticode;
- встроенная поддержка программы Microsoft Wallet (бумажник Microsoft).

Поддержка защищенных каналов осуществляется с помощью протокола PPTP (*Point to Point Tunneling Protocol*), который позволяет выполнять безопасное соединение с удаленной сетью даже посредством незащищенной сети. Такая связь устанавливается с помощью зашифрованных инкапсулированных пакетов и обеспечивает возможность вложения одного протокола в другой. Например, таким образом пользователь может подключиться к Internet с помощью протоколов TCP/IP и установить защищенное IPX-соединение со своей офисной сетью. При этом обеспечивается поддержка SSL 3.0, новой версии протокола SSL (*Secured Sockets Layer*), что повышает степень безопасности при обмене данными по сетям Internet и Intranet.

Поддержка операций с интеллектуальными карточками реализована в виде двухуровневой модели, включающей драйверы устройств считывания карточек, а также набор API-функций, предназначенных для аутентификации, записи и чтения данных. Интеллектуальные карточки выполняют, по крайней мере, три важные функции: аутентификацию пользователей при входе в систему (вместо паролей или в сочетании с ними); проведение финансовых операций по Internet и другим сетям; хранение информации о человеке(например, индивидуальные противопоказания к лекарствам или история болезни).

Поддержка API криптографии, технологии Authenticode и программы Microsoft Wallet реализована в виде модулей, которые инсталлируются при установке Internet Explorer.

В табл. 7.1 перечислены возможности подсистем безопасности, реализованные в Windows 95, Windows 98 и Windows NT.

Таблица 7.1

Технология	Windows 95	Windows 98	Windows 2000
Authenticode	Надстройка (IE4)	Есть	Есть
PPTP-клиент	Надстройка (IE4)	Есть	Есть
PPTP-сервер	Нет	Есть	Есть
Интеллектуальные карточки	Надстройка (IE4)	Есть	Есть
API криптографии	Надстройка (IE4)	Есть	Есть
Microsoft Wallet	Надстройка (IE4)	Есть	Есть
Безопасность на уровне групп	Частично	Частично	Есть
Безопасность на уровне файлов	Нет	Нет	Есть
Права и привилегии доступа для отдельных объектов	Нет	Нет	Есть

Как показано в табл. 7.1, Windows NT и Windows 98 имеют много общих технологий безопасности, в частности технологий, связанных с Internet. Однако в Windows NT они реализованы на более высоком уровне.

Хорошим примером различия между уровнями безопасности Windows NT и Windows 98 является доступ к файлам. В Windows 98 (и Windows 95) защита обеспечивается только на уровне доступа к диску и к папке. В указанных системах нельзя установить индивидуальные права доступа к отдельным файлам. Отчасти это обусловлено тем, что в среде Windows 98 существует много различных способов доступа к файлу: посредством DOS-прерываний, функций BIOS, функций Windows API, а также средств различных языков программирования (Pascal, C++ и т. д.). Из-за такого изобилия возможностей доступа к файлам возникает множество «лазеек» в системе защиты, которые очень сложно предусмотреть и устранить, не прерывая работу текущих приложений [1, 7].

Технологии безопасности реализуются с помощью специального набора API-функций, который называется API безопасности. Хотя этот набор входит в состав подсистемы Win32API, он полностью реализован только в Windows NT. С точки зрения программистов это означает, что полная безопасность может быть обеспечена лишь в этой системе. Такой вывод никого не удивит, ведь всем известно, что основное преимущество Windows NT перед Windows 98 — это встроенная разветвленная система безопасности.

В настоящее время в операционной системе Windows NT имеется около 80 API-функций, предназначенных для обеспечения безопасности (исключая API-функции для шифрования данных). Тем не менее API-функции, хотя бы частично способные работать в среде Windows 98, можно пересчитать по пальцам. Как правило, со всеми API-функциями безопасности системы Windows NT связаны функции-заглушки, которые расположены в соответствующих DLL-файлах. Поэтому приложение, разработанное для Windows NT, загружается в среде Windows 98 без ошибок компоновки. Следовательно, разрабатывая приложение для Windows NT, вы самостоятельно должны определить, какие из его возможностей должны использоваться при работе в среде Windows 98 (если приложение вообще сможет работать в этой среде).

Для каких приложений действительно необходимо обеспечить безопасность уровня системы Windows NT? Первоочередными кандидатами являются программы, выполняемые на ***сервере***. Кроме того, безопасность на уровне Windows NT необходима приложениям, имеющим доступ к собственным защищенным объектам (совместно используемая память, файлы, семафоры, системный реестр и т. д.), а также большинству приложений, применяемых в качестве ***сервисов***.

Для предотвращения операции несанкционированного доступа к определенным возможностям приложения или к определенным разделам базы данных программисту необходимо самостоятельно организовывать специальную проверку безопасности программы в следующей последовательности [1]:

– подготовка списка всех информационных объектов и/или операций, доступ к которым нельзя предоставлять без проверки полномочий пользователей;

– разработка логической схемы специальных прав и привилегий доступа, которые можно предоставлять различным пользователям и/или группам пользователей, эксплуатирующим вашу программу;

– задание в своей программе проверки безопасности везде, где осуществляется доступ к защищенным объектам или операциям;

– ограничение доступа к защищенным объектам извне (например, с помощью стандартных функций доступа к файлам). Обычно это достигается путем установки атрибута доступа к объекту «Только для администратора» и запуска приложения с привилегиями администратора, чтобы оно имело доступ к собственному объекту.

Система безопасности в Windows NT основана на модели безопасности ***для каждого пользователя***. При выполнении всех операций, которые активизируются пользователем после входа в систему (запуск приложений, доступ к принтерам и дискам, открытие и закрытие файлов), производится проверка того, обладает ли пользователь соответствующими правами и привилегиями [1, 7].

API-функции безопасности в Windows NT предоставляют возможность регистрировать события на системном уровне и на уровне приложений. Это позволяет определить, кто и когда имел доступ к различным компонентам системы. После того как пользователь вошел на защищенный

сервер, ему автоматически присваивается ***маркер доступа***. Этот маркер закреплен за пользователем все время, пока он находится в сети Windows NT. С другой стороны, каждому системному объекту соответствует ***дескриптор безопасности*** (*security descriptor* — SD), который содержит различную информацию, связанную с безопасностью данного объекта. С помощью маркера доступа и дескриптора безопасности система защиты Windows NT проверяет при обращении к объекту, имеет ли пользователь право работать с данным объектом.

В Windows NT встроенные средства безопасности реализованы только для объектов ядра операционной системы (подсистема Kernel). Поэтому, создавая такие объекты, как растровые рисунки, указатели мыши, перья, кисти, значки, метафайлы или дескрипторы окон, вы не должны принимать меры по обеспечению их защиты (с помощью API-функций безопасности). Но при создании или обращении к объектам ядра (файлы, папки, устройства хранения информации, каналы, почтовые слоты, последовательные порты, разделы реестра, консольные устройства и принтеры) необходимо предоставлять, по крайней мере, дескриптор безопасности (или указывать вместо соответствующего параметра значение NULL).

В среде Win32 постоянно приходится сталкиваться с API-функциями, предназначенными для манипулирования объектами ядра. Эти функции в качестве аргумента всегда принимают указатель на структуру SECURITY_ATTRIBUTES. Подобные функции мы уже рассматривали ранее, например при создании потоков и процессов (CreateThread()):

```
HANDLE CreateThread (
        LPSECURITY_ATTRIBUTES lpThreadAttributes,    // привилегии доступа
        DWORD dwStackSize,                           // по умолчанию равен 0
        LPTHREAD_START_ROUTINE lpStartAddress,       // указатель на стартовую функцию
        LPVOID lpParameter,                          // значение, передаваемое функции
        DWORD dwCreationFlags,                       // активное состояние или состояние ожидания
        LPDWORD lpThreadId );                        // здесь система возвращает идентификатор потока
```

Чаще всего программист применяет данную функцию, просто задав значение NULL в качестве аргумента lpSecurityAttributes. Такой подход является удачным при реализации большинства стандартных операций доступа. Это связано с тем, что при задании значения NULLиспользуется набор параметров структуры SECURITY_ATTRIBUTES, который задан по умолчанию.

Если необходимо нечто большее, чем атрибуты, заданные по умолчанию посредством значения NULL для структуры SECURITY_ATTRIBUTES, нужно вручную сформировать эту структуру.

§ 7.2. Создание структуры SECURITY_ATTRIBUTES

На первый взгляд, в структуре SECURITY_ATTRIBUTES нет ничего сложного:

```
typedef struct _SECURITY_ATTRIBUTES
{
        DWORD nLength;                  // размер структуры SECURITY_ATTRIBUTES;
        LPVOID lpSecurityDescriptor;    // дескриптор безопасности
        BOOL bInheritHandle;            // будет ли дескриптор наследоваться дочерним процессом.
}
SECURITY_ATTRIBUTES; *LPSECURITY_ATTRIBUTES;
```

Заполнить поле nLength очень просто (и очень важно!) [1]. Установить флаг bInheritHandle также несложно: он имеет значение типа Boolean. А вот второе поле, которое представляет собой указатель структуры SECURITY_DESCRIPTOR, мы рассмотрим более подробно. Если вы не хотите использовать механизмы безопасности, lpSecurltyDescriptor можно присвоить значение NULL. Если вы намерены защитить объект от несанкционированного доступа, необходимо создать структуру SECURITY_DESCRIPTOR, соответствующую этому объекту, и внести указатель на нее в структуру SECURITY_ATTRIBUTES.

Структура SECURITY_DESCRIPTOR (SD) хранит в себе информацию, связанную с защитой некоторого объекта от несанкционированного доступа. В этой структуре содержится информация о том, какие пользователи обладают правом доступа к объекту и какие действия эти пользователи могут выполнить в отношении этого объекта. Следует отметить, что внутреннее строение структуры SECURITY_DESCRIPTOR не документировано. Указатель на структуру SECURITY_DESCRIPTOR в составе структуры SECURITY_ATTRIBUTES является указателем на тип *void*. Предполагается, что если вы знаете о строении структуры SECURITY_DESCRIPTOR, то теоретически можете обойти систему безопасности.

Дескриптор безопасности объекта хранит в себе следующую информацию:

- SID (SecurityID) владельца объекта;
- SID основной группы владельца объекта;
- дискреционный список управления доступом (Discretionary Access Control List, DACL);
- системный список управления доступом (System Access Control List, SACL);
- управляющая информация (например, сведения о том, как списки ACL передают информацию дочерним дескрипторам безопасности).

Идентификатор безопасности (SID) представляет собой структуру переменной длины, которая однозначно идентифицирует пользователя или группу пользователей. Внутри идентификатора безопасности (среди прочей информации) содержится 48-разрядный уникальный код аутентифицированного лица (пользователя или группы), состоящий из значения уровня доступа, кода основного и вторичного лица.

Список DACL определяет, кто обладает (и кто не обладает) правом доступа к объекту. Список SACL определяет, информация о каких действиях вносится в файл журнала.

Но как можно создать дескриптор безопасности, если вы не знаете его структуры? Для этой цели следует использовать системный вызов InitializeSecurityDescriptor. Этой функции следует передать указатель на дескриптор безопасности SECURITY_DESCRIPTOR и значение DWORD, которое содержит номер ревизии структуры SECURITY_DESCRIPTOR. В качестве второго аргумента следует использовать значение SECURITY_DESCRIPTOR_REVISION.

Функция InitializeSecurityDescriptor инициализирует указанный вами дескриптор таким образом, что в нем отсутствует DACL, SACL, владелец и основная группа владельца, а все управляющие флаги установлены в значение FALSE. При этом дескриптор имеет абсолютный формат. Что это значит? Дескриптор в абсолютном (*absolute*) формате содержит лишь указатели на информацию, связанную с защитой объекта. В отличие от этого дескриптор в относительном (*self-relative*) формате включает в себя всю необходимую информацию, которая располагается в памяти последовательно, поле за полем. Таким образом, абсолютный дескриптор нельзя записать на диск (так как при последующем чтении его с диска все указатели потеряют смысл), а относительный дескриптор — можно.

Windows позволяет преобразовывать абсолютный дескриптор в относительную форму и обратно. Обычно это требуется лишь в случае, если вы записываете дескриптор на диск и считываете дескриптор с диска. Системные вызовы, требующие передачи указателя на дескриптор безопасности, работают только с дескрипторами в абсолютном формате.

Преобразование осуществляется при помощи функций MakeSelfRelativeSD и MakeAbsoluteSD. Преобразовать абсолютную форму в относительную несложно. Однако обратное преобразование (из относительной в абсолютную) обычно выполняется в несколько этапов. Дело в том, что при этом необходимо подготовить несколько буферов в памяти и передать указатели на них в функцию MakeAbsoluteSD.

Для работы с идентификаторами SID используются две основные функции: LookupAccountName и LookupAccountSid. Эти функции позволяют преобразовать идентификатор SID в имя учетной записи и наоборот — имя учетной записи в идентификатор SID.

Если программа работает в сетевой среде, она может обратиться к этим функциям с удаленного компьютера. Эта возможность может пригодиться, например, при разработке сервера, который обслуживает удаленных клиентов. При обращении к любой из этих функций в качестве имени компьютера можно указать NULL, и тогда вы сможете получить интересующие вас сведения относительно локальных пользователей.

Помимо имени или идентификатора SID каждая из этих функций также сообщает значение перечисляемого типа, указывающее, какому классу объектов соответствует этот SID (табл. 7.2). Переменная типа SID_NAME_USE содержит класс идентификатора SID, с которым вы имеете дело.

Таблица 7.2

Имя	Описание
SidTypeUser	Обычный пользователь
SidTypeGroup	Обычная группа
SidTypeDomain	Доменное имя
SidTypeAlias	Псевдоним
SidTypeWellKnownGroup	Хорошо известная группа
SidTypeDeletedAccount	Старая учетная запись
SidTypeUnknown	Неизвестный объект

Чтобы получить имя текущего пользователя, следует использовать функцию GetUserName. Получив имя, вы можете определить соответствующий SID при помощи функции LookupAccountSid.

Не всем идентификаторам SID соответствуют имена. Например, в процессе подключения к системе пользователю присваивается произвольный SID, идентифицирующий текущий рабочий сеанс. Этот SID не обладает соответствующим ему именем. К функциям, которые могут оказаться полезными при работе с SID, можно отнести AllocateAndInitializeSid, InitializeSid, FreeSid, CopySid, IsValidSid, GetLengthSid и EqualSid.

Элемент управления доступом (*Access Control Element* — ACE) — это запись, которая указывает на то, что некоторый пользователь (или группа) обладает определенным правом. Записи ACE бывают трех основных разновидностей: ACCESS_ALLOWED_ACE_TYPE, ACCESS_DENIED_ACE_TYPE и SYSTEM_AUDIT_ACE_TYPE. Запись ACE первого типа наделяет пользователя (или группу) некоторым правом. Запись второго типа отменяет это право в отношении пользователя (или группы). Запись третьего типа обозначает необходимость осуществления аудита при пользовании правом.

Список управления доступом (*Access Control List* — ACL) — это набор записей ACE. Дискреционный список DACL содержит записи типа ACCESS_ALLOWED_ACE_TYPE и ACCESS_DENIED_ACE_TYPE, а системный список SACL содержит записи типа SYSTEM_AUDIT_ACE_TYPE.

Каждая запись ACE, определяющая уровень доступа к объекту, обладает 32-битной маской ACCESS_MASK. Эта маска является набором бит, которые определяют, какие права предоставляет или отменяет данная запись ACE. Значение каждого бита определяется объектом, по отношению к которому применяется данная ACE. Например, набор прав, которые можно назначить в отношении семафора, отличается от набора прав, которые можно назначить в отношении файла. Все же существуют четыре права, которые можно назначить в отношении абсолютно любого защищаемого объекта — это GENERIC_READ (обобщенное чтение), GENERIC_WRITE (обобщенная запись), GENERIC_EXECUTE (обобщенное исполнение) и GENERIC_ALL (все права). Этим четырем правам всегда соответствуют одни и те же биты в маске ACCESS_MASK любой записи ACE.

Каждый системный объект или файл обладает индивидуальным списком ACL. Чтобы обеспечить защиту данных, о существовании которых система не имеет представления, необходимо самостоятельно сформировать собственный список ACL и сохранить его специальным образом. Система безопасности может использоваться для защиты объектов нескольких разных типов. Для доступа к дескриптору безопасности каждого из этих объектов следует использовать разные функции (табл. 7.3).

Таблица 7.3

Тип объекта	Примеры	Функции
Файлы (Files)	Файлы NTFS, именованные каналы	GetSecurity, SetSecurity
Пользователь (User)	Окна, меню	GetUserObjectSecurity, SetUserObjectSecurity
Ядро ОС (Kernel)	Дескрипторы процессов и потоков, объекты отображения в память и т. д.	GetKernelObjectSecurity, SetKernelObjectSecurity
Реестр (Registry)	Ключи	RegGetKeySecurity, RegSetKeySecurity
Службы (Services)	Любая служба	QueryServiceObjectSecurity, SetServiceObjectSecurity
Частный (Private)	Определяемые пользователем объекты	CreatePrivateObjectSecurity, GetPrivateObjectSecurity, SetPrivateObjectSecurity, DestroyPrivateObjectSecurity

Когда говорят о защите данных, то чаще всего имеют в виду файлы. На самом деле в некоторых ситуациях может потребоваться ограничение доступа к объектам других типов. В некоторых ситуациях может возникнуть необходимость в создании ваших собственных объектов, доступ к которым необходимо контролировать. Windows может не знать о существовании этих объектов, однако это не значит, что хранящаяся в них информация не нуждается в защите.

Представьте, например, что вы разрабатываете базу данных, содержащую информацию о заработках сотрудников. Для защиты этой информации можно использовать ваши собственные списки ACL, содержащие сведения о том, кто имеет доступ к данным о зарплате. Если база данных получает запрос от неавторизированного источника, система может внести сообщение об этом в журнал аудита, полностью запретить доступ к базе данных или передать обратившемуся клиенту все запрашиваемые им данные за исключением секретной информации, вместо которой будет передан символ звездочки.

Рассмотрим принцип построения списков ACL [1]. Список ACL — это область памяти, содержащая заголовок ACL и некоторое количество (или ни одной) записей ACE. В свою очередь, запись ACE включает в себя ACE_HEADER и одну из структур: ACCESS_ALLOWED_ACE (соответствует разрешающей записи) или ACCESS_DENIED_ACE (соответствует запрещающей записи). При заполнении списков ACE в Windows NT 4.0 (и в более ранних версиях) следовало следить за тем, чтобы записи типа ACCESS_DENIED_ACE располагались в списке раньше, чем все остальные записи. Дело в том, что запрещающие записи имеют больший вес, чем разрешающие записи. Если в списке ACL одна запись разрешает некоторому пользователю доступ к объекту, а другая запись запрещает этому же пользователю доступ к этому же объекту, то в силу вступает именно запрещающая запись. В этом случае разрешающая запись игнорируется и пользователь теряет право на доступ к объекту. В Windows NT 4.0 это условие выполнялось только в том случае, если запрещающие записи располагались в начале списка ACL.

Одной из проблем, возникающих при работе с ACL, является следующее обстоятельство: для того чтобы сформировать список ACL, вы должны вычислить его точный размер. Список ACL должен обладать размером, достаточным для того, чтобы вместить в себя все ACE. Однако разные записи ACE могут включать в себя разное количество байт. Таким образом, задача вычисления размера ACL может оказаться непростой.

Размер записи ACE зависит от ее типа и длины SID, который в ней содержится. Для записей ACE, разрешающих доступ, длина записи составляет:

```
L= sizeof(ACCESS_ALLOWED_ACE)–sizeof(ACCESS_ALLOWED_ACE.SidStart) +
+ GetLengthSid((PSID)ace.SidStart);
```

Как видно, к длине заголовка (за вычетом длины первой части SID) добавляется длина SID.

Таким образом, размер ACL равняется размеру структуры ACL плюс сумма размеров всех ACE, входящих в список. Зачастую возникает соблазн не тратить время на расчеты, а просто зарезервировать достаточно большой буфер в памяти и предположить, что выделенного места должно хватить для размещения всего списка. В общем случае такой подход не запрещается — ACL может обладать размером бóльшим, чем размер, достаточный для хранения всех его записей.

Когда вы получили достаточно объемный буфер, вы должны инициализировать ACL при помощи вызова InitializeAcl. После этого можно добавлять в список записи ACE. Для этого служат функции AddAccessDeniedAce и AddAccessAllowedAce. Каждая из этих функций принимает в качестве аргументов указатель на буфер ACL, константу ACL_REVISION, права, которые вы намерены предоставить или отменить, а также указатель на SID. Этот SID идентифицирует пользователя или группу, которым соответствует ACE. Получить SID можно при помощи вызова LookupAccountName.

Помните, что записи ACE, запрещающие доступ, следует добавлять в список в первую очередь, в противном случае WindowsNT 4.0 и более ранние версии NT будут некорректно обрабатывать взаимоисключающие ACE. Чтобы пояснить ситуацию, рассмотрим следующий простой пример. Допустим, я являюсь членом группы TECH, а также членом группы USERS. Допустим, в списке ACL содержится запись, запрещающая для группы USERS доступ к файлу. Другая запись разрешает доступ к файлу для группы TECH. В этой ситуации я не должен иметь возможности прочитать файл, так как принадлежу к группе, для которой доступ к файлу явно запрещен. Если вы не разместите все ACE, запрещающие доступ, в самом начале списка ACL, в некоторых версиях NT такая комбинация взаимоисключающих ACE будет обработана некорректно.

Когда ACL сформирован, его необходимо установить в рамках структуры SECURITY_DESCRIPTOR. Для этого служит вызов SetSecurityDescriptorDacl. Затем указатели на дескриптор безопасности, содержащий ACL, необходимо разместить в структуре SECURITY_ATTRIBUTES. Последнюю, в свою очередь, можно передать любой API- функции, элементом которой является данная структура.

§ 7.3. API-функции для обеспечения безопасности Windows

Описание API-функций безопасности, разделенных на категории [4], приведено в табл. 7.4.

Таблица 7.4

Функция		Описание
Категория	Вид	
1	*2*	*3*
Для операций с маркерами доступа	AdjustTokenGroups	Изменяет группу, которой принадлежит маркер доступа
	AdjustTokenPrivileges	Изменяет привилегии для маркера доступа
	DuplicateToken	Создает новый маркер доступа, идентичный заданному
	DuplicateTokenEx	Аналогична предыдущей, но может создавать первичный маркер доступа, используемый в функции GreateProcessAsUser
	GetTokenInformation	Возвращает данные о пользователе, группе, привилегиях, а также другую информацию, содержащуюся в маркере доступа
	SetThreadToken	Присваивает заданному потоку маркер передачи полномочий
	SetTokenInformation	Изменяет данные о пользователе, группе, привилегиях, а также другую информацию, содержащуюся в маркере доступа
	OpenProcessToken	Читает маркер доступа для заданного процесса
	OpenThreadToken	Читает маркер доступа для заданного потока

Продолжение табл. 7.4

1	2	3
Используемые в процессе передачи полномочий клиента	CreateProcessAsUser	Идентична функции CreateProcess, но создает процесс с заданным маркером доступа
	DdeImpersonateClient	Позволяет DDE-серверу принять полномочия DDE-клиента для доступа к ресурсам с использованием клиентских атрибутов безопасности
	ImpersonateLoggedOnUser	Позволяет вызываемому потоку принять полномочия заданного пользователя (взять его маркер доступа)
	ImpersonateNamedPipeClient	Позволяет серверной части именованного канала принять полномочия клиентской части
	ImpersonateSelf	Возвращает маркер доступа, соответствующий вызывающему процессу (часто используется для изменения доступа на уровне потока)
	LogonUser	Позволяет серверному приложению запросить маркер доступа заданного пользователя, чтобы зарегистрироваться в системе от его имени
	RevertToSelf	Завершает процесс передачи полномочий клиента
Для операций с дескрипторами безопасности	MakeAbsoluteSD	Создает дескриптор безопасности в абсолютном формате по образцу дескриптора в относительном формате
	MakeSelfRelativeSD	Создает дескриптор безопасности в относительном формате по образцу дескриптора в абсолютном формате
	InitializeSecurityDescriptor	Инициализирует новый дескриптор безопасности, при этом ему не присваиваются права
	IsValidSecurityDescriptor	Проверяет правильность дескриптора безопасности
	GetSecurityDescriptorControl	Возвращает информацию об уровне доступа для дескриптора безопасности
	GetSecurityDescriptorLength	Возвращает размер заданного дескриптора безопасности в байтах
	GetSecurityDescriptorDacl	Возвращает указатель на DACL для заданного дескриптора безопасности
	GetSecurityDescriptorGroup	Возвращает указатель на идентификатор безопасности основной группы для заданного дескриптора безопасности
	GetSecurityDescriptorOwner	Возвращает указатель на идентификатор безопасности владельца для заданного дескриптора безопасности
	GetSecurityDescriptorSacl	Возвращает указатель на SACL для заданного дескриптора безопасности
	SetSecurityDescriptorDacl	Обновляет информацию о DACL для заданного дескриптора безопасности
	SetSecurityDescriptorGroup	Обновляет идентификатор безопасности группы для заданного дескриптора безопасности
	SetSecurityDescriptorOwner	Обновляет идентификатор безопасности владельца для заданного дескриптора безопасности
	SetSecurityDescriptorSad	Обновляет информацию о SACL для заданного дескриптора безопасности

Продолжение табл. 7.4

1	2	3
Для операций с идентификаторами безопасности	AllocateAndInitializeSid	Выделяет и инициализирует идентификатор безопасности для 1–8 вторичных лиц
	AllocateLocallyUniqueId	Выделяет локальный уникальный идентификатор
	InitializeSid	Инициализирует структуру идентификатора безопасности для заданного количества вторичных лиц
	CopySid	Записывает в буфер копию идентификатора безопасности
	EqualPrefixSid	Выполняет логическую (True/False) проверку равенства префиксов двух идентификаторов безопасности
	EqualSid	Выполняет логическую (True/False) проверку равенства двух идентификаторов безопасности
	FreeSid	Освобождает идентификатор безопасности, выделенный функцией AllocateAndInitializeSid
	GetSidIdentifierAuthority	Возвращает указатель на идентификатор безопасности аутентифицированного пользователя или группы (в виде структуры SID_IDENTIFIER_AUTHORITY)
	GetSidLengthRequired	Возвращает размер идентификатора в байтах, который необходим для хранения информации о заданном количестве вторичных лиц
	GetSidSubAuthority	Возвращает указатель на *n*-е вторичное лицо заданного идентификатора безопасности
	GetSidSubAuthorityCount	Возвращает количество вторичных лиц, определенных в идентификаторе безопасности
	GetLengthSid	Возвращает размер заданного идентификатора безопасности в байтах
	IsValidSid	Проверяет правильность заданного идентификатора безопасности
	LookupAccountSid	Возвращает учетное имя и имя первого домена, найденного для заданного идентификатора безопасности
	LookupAccountName	Возвращает SID, соответствующий заданному учетному имени, и домен, в котором найдено имя
Для операций с записями в списках управления доступом	AddAce	Добавляет одну или несколько записей (ACE) в указанную позицию заданного списка управления доступом (ACL)
	AddAccessAllowedAce	Добавляет в ACL запись с разрешением доступа, обеспечивая таким образом доступ к указанному идентификатору безопасности
	AddAccessDeniedAce	Добавляет в ACL запись, в которой запрещен доступ; таким образом предотвращается доступ к указанному идентификатору безопасности
	AddAuditAccessAce	Добавляет запись в SACL
	DeleteAce	Удаляет *n*-ю запись из заданногоACL
	FindFirstFreeAce	Возвращает указатель первой свободной позиции в заданномACL
	GetAce	Возвращает указатель на *n*-ю запись в заданномACL

Продолжение табл. 7.4

1	2	3
Для операций со списками управления доступом	InitializeAcl	Создает новую структуру ACL
	GetAclInformation	Возвращает информацию о заданномACL (размер, значение счетчика записей и др.)
	IsValidAcl	Проверяет правильность ACL
	SetAclInformation	Задает информацию об ACL
Для проверки прав доступа	AccessCheck	Используется серверным приложением для проверки прав доступа клиента к объекту
	AccessCheckAndAuditAlarm	Выполняет функцию AccessCheck и генерирует соответствующие аудиторские сообщения
	AreAllAccessesGranted	Проверяет, были ли предоставлены заданному пользователю все необходимые права доступа
	AreAnyAccessesGranted	Проверяет, было ли предоставлено заданному пользователю хотя бы одно из необходимых прав доступа
	PrivilegeCheck	Проверяет, имеет ли заданный маркер доступа необходимые привилегии
	PrivilegedServiceAuditAlarm	Выполняет функцию PrivilegeCheck и генерирует соответствующие аудиторские сообщения
	MapGenericMask	Накладывает маску общих прав доступа на специальные или стандартные права
Для операций с привилегиями	MapGenericMask	Накладывает заданную маску общих прав доступа на специальные или стандартные права
	PrivilegeCheck	Проверяет, имеет ли заданный маркер доступа указанные привилегии
	AllocateSocallyUniqueId	Выделяет локальный уникальный идентификатор
	LookupPrivilegeDisplayName	Читает строку с названием указанной привилегии
	LookupPrivilegeName	Читает имя привилегии, связанное с заданным локальным уникальным идентификатором (LUID)
	LookupPrivilegeValue	Возвращает LUID, связанный с указанной привилегией в данной системе
	ObjectPrivilegeAuditAlarm	Генерирует аудиторские сообщения, когда указанный пользователь (маркер доступа) пытается выполнить привилегированные операции с заданным объектом
	PrivilegedServiceAuditAlarm	Генерирует аудиторские сообщения, когда указанный пользователь (маркер доступа) пытается выполнить привилегированные операции
Для операций с окнами-станциями	GetProcessWindowStation	Возвращает дескриптор окна-станции, связанного с вызываемым процессом
	SetProcessWindowStation	Назначает заданное окно-станцию вызываемому процессу
Для операций с *local service authority*	InitLsaString	Создает структуру LSA_UNICODE_STRING для имени заданной привилегии
	LsaOpenPolicy	Задает определенную политику безопасности на указанном компьютере
	LsaLookupNames	Возвращает учетное имя (и идентификатор безопасности) для заданного маркера доступа
	LsaRemoveAccountRights	Отбирает указанные привилегии у заданного пользователя (или у нескольких пользователей)

Окончание табл. 7.4

1	2	3
	LsaAddAccountRights	Предоставляет указанные привилегии заданному пользователю (или нескольким пользователям)
	LsaClose	Отменяет определенную политику безопасности
	LsaEnumerateAccountRights	Создает список прав доступа и привилегий, которые предоставлены заданной учетной записи
	LsaEnumerateAccountsWithUserRight	Создает список всех учетных записей данной системы, которым предоставлена указанная привилегия
Для получения информации, связанной с безопасностью	GetFileSecurity	Получает дескриптор безопасности указанного файла или каталога
	GetKernelObjectSecurity	Получает дескриптор безопасности указанного объекта ядра
	GetPrivateObjectSecurity	Получает дескриптор безопасности указанного приватного объекта
	GetUserObjectSecurity	Получает дескриптор безопасности указанного пользовательского объекта
	SetFileSecurity	Присваивает файлу или каталогу указанный дескриптор безопасности
	SetKernelObjectSecurity	Присваивает объекту ядра указанный дескриптор безопасности
	SetPrivateObjectSecurity	Присваивает приватному объекту указанный дескриптор безопасности
	SetUserObjectSecurity	Присваивает пользовательскому объекту указанный дескриптор безопасности
	CreatePrivateObjectSecurity	Выделяет и инициализирует дескриптор безопасности в относительном формате; этот дескриптор будет назначен новому приватному объекту
	DestroyPrivateObjectSecurity	Удаляет дескриптор безопасности заданного приватного объекта
Для аудита	ObjectOpenAuditAlarm	Генерирует аудиторские сообщения, когда пользователь предпринимает попытку получить доступ к существующему объекту или создать новый объект
	ObjectCloseAuditAlarm	Генерирует аудиторские сообщения при закрытии дескриптора объекта
	ObjectDelateAuditAlarm	Генерирует аудиторские сообщения при удалении дескриптора объекта
	ObjectPrivilegeAuditAlarm	Генерирует аудиторские сообщения, когда указанный пользователь (маркер доступа) пытается выполнить привилегированные операции с заданным объектом
	PrivilegedServiceAuditAlarm	Генерирует аудиторские сообщения, когда указанный пользователь (маркер доступа) пытается выполнить привилегированные операции
	AccessCheckAndAuditAlarm	Выполняет функцию AccessCheck и генерирует соответствующие аудиторские сообщения

Глава 8
Принципы разработки программного кода для обмена данными между процессами в ОС Windows

§ 8.1. Обмен данными посредством буфера обмена Windows

8.1.1. Структура и основные форматы буфера обмена

Буфер обмена Windows состоит из двух разных частей: утилиты просмотра Clipbrd.EXE и непосредственно буфера. Сам буфер реализован с помощью системного модуля User и обеспечивает набор функций для временного хранения информации в такой форме, которая позволяет осуществлять обмен данными между различными приложениями. Конечно, приложение-источник тоже может прочитать собственную информацию, записанную в буфер обмена, но дело в том, что эта информация является глобальной и доступна для любых программ [6, 12].

Буфер обмена предоставляет набор возможностей для временного (не дискового) хранения информации. Данные, записанные в буфер обмена, могут передаваться между различными приложениями или возвращаться в исходное приложение, из которого они попали в буфер.

Приложение-источник может копировать данные в буфер обмена в одном из предопределенных или пользовательских форматов (они будут рассмотрены позже). Буфер самостоятельно управляет выделением памяти и размещением переданных ему данных.После того как данные попадут в буфер обмена, любое приложение сможет получить к ним доступ, определить тип данных и при желании скопировать их.

Когда программа просмотра Clipbrd.EXE является активной, она регулярно запрашивает буфер обмена о том, содержатся ли в нем данные и каков их тип. Затем, если это возможно, программа копирует данные из буфера и отображает их в своем окне.

Хотя буфер обмена выполняет свои обязанности очень хорошо, ему свойственны и некоторые недостатки, а именно:

- имеется только один буфер;
- все данные, записанные в буфер, являются общедоступными;
- записанные в буфер данные могут быть разрушены.

Так как буфер обмена только один, все приложения должны использовать его совместно. Но совместное использование неизбежно связано с возможностью конфликтов. Предположим, что приложение А записало в буфер обмена растровое изображение, а затем приложение Б записало в него блок текстовых данных. Поскольку приложение Б вполне закономерно начинает с того, что очищает буфер обмена, картинка, записанная приложением А, удаляется. Если приложение В, для которого была предназначена эта картинка, уже успело ее прочитать, то все в порядке. Если же приложение В не скопировало изображение до того, как приложение Б записало в буфер обмена текст, картинка пропадает.

Общедоступность буфера обмена также может стать причиной возможных ошибок. Поскольку элемент данных, записанный в буфер, не может быть адресован какому-либо конкретному приложению, доступ к этой информации может быть ошибочно получен другим приложением, которое работает с данными того же типа [12].

Буфер обмена может содержать данные нескольких типов, записанных одним или несколькими приложениями. В этом случае проблема заключается в том, как различить эти блоки (например, несколько блоков текста, записанных из разных источников). По этой причине приложение обычно очищает буфер обмена перед тем, как записывать в него новый материал.

Эти факторы необходимо учитывать, но они не представляют серьезной проблемы. Если перечисленные недостатки накладывают дополнительные ограничения на работу программы, можно воспользоваться технологиями каналов (*pipe*), сокетов (*sockets*) или динамического обмена данными (DDE), которые будут рассмотрены далее. Эти технологии позволяют реализовать безопасный канал для обмена информацией.

Windows поддерживает 14 стандартных форматов данных буфера обмена, определенных в файле WinUser.H.К ним относятся:

CF_BITMAP	CF_OEMTEXT	CF_TEXT
CF_DIB	CF_PALETTE	CF_TIFF
CF_DIF	CF_PENDATA	CF_UNICODETEXT
CF_ENHMETAFILE	CF_RIFF	CF_WAVE
CF_METAFILEPICT	CF_SYLK	

В файле WinUser.H определен также ряд специальных форматов, или флагов формата. Кроме того, каждое приложение может предложить собственный пользовательский формат данных буфера обмена, которые будут рассмотрены далее. Но для большинства целей вполне достаточно стандартных форматов. Рассмотрим эти форматы более подробно.

Текстовые форматы. Простейший формат данных буфера обмена—это текстовый формат CF_TEXT, который состоит из набора строк ANSI-символов, заканчивающегося нулевым символом. Каждая строка завершается символами возврата каретки (0×0D) и перевода строки (0x0A). Формат CF_OEMTEXT представляет собой набор OEM-символов. Формат CF_UNICODETEXT использует 32-разрядные символы набора Unicode.

Передав текст в буфер обмена, исходное приложение не имеет возможности обратиться к этому тексту, не запросив доступа к буферу.

Формат растровых изображений. Формат CF_BITMAP служит для хранения растровых изображений путем передачи буферу обмена дескриптора изображения. Записав изображение в буфер, исходное приложение не имеет возможности обратиться к этому изображению, не запросив доступа к буферу обмена.

Форматы метафайлов. Формат CF_METAFILEPICT служит для обмена метафайлами, находящимися в памяти (а не на диске), между различными приложениями. Этот формат использует структуру METAFILEPICT, которая определена в файле WinGDI.H следующим образом:

```
typedef struct tagMETAFILEPICT
{
        LONG mm;
        LONG xExt;
        LONG yExt;
        HMETAFILE hMF;
} METAFILEPICT, FAR *LPMETAFILEPICT;
```

Первые три поля этой структуры отображают различия, наблюдаемые при передаче метафайлов посредством буфера обмена и через дисковые файлы. Первое поле (mm) указывает предпочтительный режим отображения. Второе и третье поля (xExt и yExt) задают ширину и высоту содержащегося в метафайле изображения. Поле hMF содержит дескриптор метафайла.

Формат CF_ENHMETAFILE аналогичен формату CF_METAFILEPICT, но в отличие от последнего он идентифицирует метафайл, в котором используются расширенные команды форматирования.

DIB-формат. Формат CF_DIB применяется для передачи в буфер обмена DIB-файлов (аппаратно-независимых растровых изображений). DIB-изображение записывается в виде глобального блока памяти, начинающегося с заголовка BITMAPINFO, после которого следуют данные изображения.

Передав растровое изображение в буфер обмена, исходное приложение не имеет возможности обратиться к соответствующему глобальному блоку памяти, не запросив доступа к буферу обмена.

Форматы палитры и пера. Форматы CF_PALETTE и CF_PENDATA служат для передачи в буфер обмена дескриптора цветовой палитры и дескриптора пера соответственно. Формат CF_PALETTE часто применяется в сочетании с форматом CF_DIB для записи цветовой палитры, используемой растровым изображением.

Wave-формат. Формат CF_WAVE предназначен для передачи аудиоинформации между различными приложениями.

Форматы специального назначения. Три следующих формата изначально предназначались для применения в специализированных приложениях:

– формат CF_TIFF используется для передачи данных в формате TIFF (*Tagged Image File Format*);

– формат CF_DIF служит для передачи данных в формате DIF (*Data Interchange Format*), который был предложен фирмой Software Arts и первоначально применялся в редакторе электронных таблиц VisiCalc. Теперь лицензия на данный формат принадлежит фирме Lotus. DIF-формат основан на ASCII-строках, завершающихся парой символов CR/LF (возврат каретки / перевод строки);

– формат CF_SYLK предназначен для передачи данных в формате Microsoft Symbolic Link, который первоначально применялся для обмена данных между Microsoft-приложениями Multiplan (электронные таблицы), Chart и Excel. Этот формат основан на ASCII-строках, завершающихся парой символов CR/LF (возврат каретки / перевод строки).

8.1.2. Операции с буфером обмена

Управление глобальными блоками памяти, которые содержат данные, помещенные в буфер обмена, осуществляется с помощью флагов выделения памяти. При записи информации в буфер программа выделяет блок памяти с помощью функции GlobalAlloc() и флага GHND (определенного как GMEM_MOVABLE | GMEM_ZEROINIT).

Обычно при закрытии исходного приложения выделенная ему глобальная область памяти удаляется (освобождается) операционной системой. Но если приложение вызовет функцию SetClipboardData() с указанием дескриптора глобального блока памяти, Windows возьмет этот блок в свою собственность, точнее, в собственность буфера обмена, изменив флаги выделения.

Принадлежность глобального блока памяти назначается функцией GlobalRealloc():

```
GlobalRealloc(hMem, NULL, GMEM_MODIFY | GMEM_DDESHARE);
```

После этого вызова выделенный блок памяти не принадлежит исходному приложению и доступен только через буфер обмена с помощью функции GetClipboardData(). Эта функция предоставляет вызвавшему ее приложению временный доступ к данным, записанным в буфере обмена, передавая программе дескриптор глобального блока памяти. Однако принадлежность блока данных по-прежнему сохраняется за буфером обмена, а не передается приложению. Следовательно, данные, записанные в буфере обмена, могут быть удалены только путем вызова функции EmptyClipboard.

Хотя многие ресурсы Windows предназначены для совместного использования несколькими приложениями, доступ к буферу обмена в каждый момент возможен только одной программой. Это позволяет предотвратить конфликты между приложениями.

Прежде чем приложение начнет читать, записывать или удалять содержимое буфера обмена, необходимо запросить доступ к нему с помощью функции OpenClipboard(). Эта функция возвращает значение TRUE, если буфер открыт и доступ к нему разрешен, и значение FALSE, если доступ к буферу запрещен по той причине, что в данный момент право доступа принадлежит другому приложению.

Закончив работу с буфером обмена, приложение должно вызвать функцию CloseClipboard, которая делает буфер доступным для других программ. Следует отметить, что каждый вызов функции OpenClipboard() всегда должен сопровождаться вызовом функции CloseClipboard(). Приложение должно не пытаться длительное время удерживать буфер обмена открытым, а стремиться отдать контроль над ним как можно быстрее. Для ***записи данных*** в Clipboard можно использовать универсальную функцию TransferToClipboard(), копирующую блок памяти в буфер обмена [12]:

```
BOOL TransferToClipboard(HWND hwnd, HANDLE hMemBlock, WORD FormatCB)
{
        if(OpenClipboard (hwnd))
        {
                EmptyClipboard();
                SetClipboardData(FormatCB, hMemBlock);
                CloseClipboard();
                return(TRUE);
        }
return (FALSE);
}
```

Функция TransferToClipboard() начинается с запроса на право доступа к буферу обмена. Затем в буфер копируется одиночный блок памяти. Наконец, функция закрывает буфер обмена, освобождая его для доступа другим приложениям.

Функция TransferToClipboard() имеет довольно универсальную структуру и допускает любой тип дескриптора hMemBlock. Однако для ее работы необходим параметр FormatCB, указывающий тип данных, копируемых в буфер обмена.

Термин ***блок памяти*** не подразумевает блок какого-то определенного размера. Размер блока был определен ранее функцией GlobalAlloc. Каждый блок памяти может содержать абзац текста, несколько записей или другие данные, но обязательно одного типа [12].

Как быть, если приложение должно передать в буфер обмена смешанные данные, скажем, растровое изображение, метафайл, палитру и фрагмент текста? Решение этой проблемы довольно простое. Сначала каждый фрагмент данных копируется отдельно в глобальную выделенную память с сохранением дескриптора блока памяти, например hBitmap, hMetafile, hPalette и hText. После этого открывается и освобождается буфер обмена и в него передается каждый из дескрипторов:

```
if(OpenClipboard(hwnd))
{
        EmptyClipboard();
        SetClipboardData(CF_BITMAP, hBitmap);
        SetClipboardData(CF_PALETTE, hPalette);
        SetClipboardData(CF_METAFILEPICT, hMetaFile);
        SetClipboardData(CF_TEXT, hText);
        CloseClipboard();
}
```

После этого буфер обмена закрывается и освобождается для доступа к нему других приложений. Правда, реализовать приведенный ранее подход на практике достаточно сложно. Но поскольку идентификаторы форматов данных имеют тип WORD, а все дескрипторы также могут быть приведены к единому типу HANDLE, гораздо удобнее собрать идентификаторы и дескрипторы в массивы, а функции TransferToClipboard() передать дополнительный параметр, указывающий количество записанных элементов данных:

```
if( OpenClipboard(hwnd))
{
        EmptyClipboard();
        for(i=0; i<nCount; i++ )
                SetClipboardData(cfType[i], hData[i]);
        CloseClipboard();
}
```

Прежде чем начинать ***чтение информации*** из буфера обмена, необходимо определить, какого типа данные содержатся в нем [12]. Поскольку для различных типов данных необходимо применять разные операции, приложение должно заранее знать, что именно ему придется читать, и подготовиться к определенному виду обработки. В частности, вы можете запросить данные определенного типа и посмотреть на результат. Однако такой подход не слишком элегантен, да и эффективность его не очень высока.

Более эффективный способ анализа содержимого буфера обмена заключается в использованииAPI-функции IsClipboardFormatAvailable() или EnumClipboardFormats().

Функция IsClipboardFormatAvailable() возвращает булево значение, которое сообщает, содержит ли буфер обмена данные нужного формата. Синтаксис вызова функции таков:

```
if (IsClipboardFormatAvailable (CF_xxxx))
```

Функция EnumClipboardFormats проверяет наличие данных всех возможных форматов. При первом вызове функции с параметром NULLона возвращает информацию о первом доступном формате. При каждом последующем вызове возвращаются сведения о других форматах. Таким образом, для получения списка форматов можно воспользоваться следующим алгоритмом:

```
wFormat = NULL;
OpenClipboard(hwnd);
while(wFormat = EnumClipboardFormats(wFormat))
{
...код обработки различных форматов...
}
CloseClipboard();
```

В этом списке используемые форматы будут перечислены в том порядке, в котором исходное приложение записывало в буфер обмена соответствующие данные. Это позволит запрашивающему приложению найти первый подходящий для себя формат. Исходное приложение может записывать данные в любом другом заданном порядке, например в порядке убывания степени надежности данных.

Если список форматов исчерпан, буфер обмена пуст или не был открыт, результат выполнения функции EnumClipboardFormats() будет равен нулю. Параметру wFormat можно присвоить некоторое значение, с тем чтобы повторно прочитать список, начиная с используемого в данный момент формата. Кроме того, информацию о количестве форматов данных, находящихся в буфере обмена, можно получить с помощью оператора:

```
nFormats = CountClipboardFormats();
```

Когда приложение установит, что буфер обмена содержит данные нужного типа, процесс чтения этих данных будет состоять из двух этапов:

– получение дескриптора блока памяти, соответствующего данным, которые находятся в буфере обмена;

– выполнение определенных операций с данными при помощи имеющегося дескриптора.

Первый этап выполнить очень просто с помощью функции RetrieveCB:

```
HANDLE RetrieveCB(HWND hwnd, WORD FormatCB)
{
        HANDLE hCB;
        if(! IsClipboardFormatAvailable(FormatCB))
                return(NULL);
        OpenClipboard(hwnd);
        hCB = GetClipboardData(FormatCB);
        CloseClipboard();
        return (hCB);
}
```

Содержащаяся в этом фрагменте кода обобщенная процедура возвращает нетипизированный дескриптор блока памяти, который соответствует данным, находящимся в буфере обмена. Если запрашиваемый формат данных отсутствует, функция возвращает значение NULL.

Следует отметить, что на операции, которые могут выполняться с буфером обмена, налагается несколько ограничений:

1. Прежде чем копировать данные в буфер обмена, необходимо вызвать функцию EmptyClipboard(), предназначенную для удаления текущего содержимого. Сам факт доступа к буферу обмена еще не означает, что будет получен контроль над его содержимым. Функция EmptyClipboard() позволяет стать владельцем буфера и одновременно удалить его текущее содержимое.

2. Любое приложение может получить доступ к содержимому буфера обмена, но лишь владелец буфера, т. е. приложение, вызвавшее функцию EmptyClipboard(), имеет право записывать в него данные. Поскольку владельцем буфера обмена может быть только одно приложение, данные, записанные предыдущим владельцем, удаляются, даже если таковым было это же самое приложение.

3. Хотя в буфер обмена может быть записано несколько блоков данных, передавать их нужно в течение одной операции. Буфер обмена нельзя открыть, записать в него данные, закрыть, а затем снова открыть и добавить другие данные, не удалив предыдущий фрагмент.

4. В буфере обмена одновременно может находиться только по одному элементу данных каждого типа. Это объясняется простой причиной: не существует способа разделения нескольких эле-

ментов данных одного и того же типа. Однако при наличии данных нескольких типов приложение, получившее доступ к буферу обмена, может запросить только один элемент, несколько элементов или все элементы. В этом случае данные каждого типа должны запрашиваться отдельно.

Буфер обмена может многократно открываться для запроса других элементов данных или повторного запроса того же элемента. Но в целом, запрашивая фрагмент данных из буфера обмена, лучше всего создать его локальную копию, а не повторять один и тот же запрос снова и снова. Кроме того, не забывайте: нет никакой гарантии того, что при следующем запросе тот же самый элемент данных останется неизменным, поскольку буфер обмена является общедоступным.

8.1.3. Частные форматы буфера обмена

В Windows определено несколько «частных» форматов: CF_DSPTEXT, CF_DSPBITMAP, CF_DSPMETAFILEPICT и CF_DSPENHMETAFILE. Они соответствуют форматам CF_TEXT, CF_BITMAP, CF_METAFILEPICT и CF_ENHMETAFILE, но имеют одно принципиальное ограничение: приложения, запрашивающие стандартные форматы, не смогут получить доступа к специальным форматам буфера обмена. В этой связи следует сделать несколько замечаний:

– данные, находящиеся в одном из специальных форматов, предназначаются для обмена информацией между двумя экземплярами одного и того же приложения или между двумя приложениями, специально разработанными для совместной работы;

– в процессе подобного обмена может передаваться служебная информация, например о форматировании и / или шрифтах.

Термин ***частный формат*** может ввести в заблуждение. Дело в том, что получить доступ к таковому может любое приложение. Ведь разрабатывались подобные форматы не в целях безопасности, а лишь для обеспечения закрытого обмена данными через буфер обмена [12].

Хотя два экземпляра одного и того же приложения или два связанных приложения должны понимать свои частные форматы, применение этих форматов вовсе не гарантирует, что данные поступили от другого экземпляра того же приложения или родственного приложения. Иными словами, ничто не мешает использованию одинаковых частных форматов разными, совершенно не связанными друг с другом программами.

Однако на этот случай предусмотрены специальные меры. Вы можете узнать «автора» содержимого буфера обмена с помощью функции GetClipboardOwner():

```
hwndCBOwner = GetClipboardOwner();
```

Приложение, вызвавшее функцию EmptyClipboard() для подготовки к копированию данных, становится новым владельцем буфера обмена. Другие приложения, обращающиеся к буферу для чтения информации, его владельцами не являются — они просто получают доступ. Таким образом, за содержимое буфера обмена ответственность несет только его владелец. Приложение не может записать данные в буфер обмена, не став сначала его владельцем и не удалив его предыдущее содержимое. Поэтому любое приложение, получившее доступ к информации в частном формате, может также определить принадлежность буфера обмена.

Хотя функция GetClipboardOwner() возвращает дескриптор, указывающий на владельца буфера обмена, сам по себе этот дескриптор ни о чем не говорит. Но имея дескриптор, имя класса приложения можно запросить с помощью другой функции:

```
GetClassName (hwndCBOwner, SszClassName, 16);
```

Теперь значение переменной szClassName можно сравнить с именем класса текущего приложения или списком имен классов связанных приложений и таким образом выяснить источник информации, записанной в буфере обмена.

Передача копии исходной информации в буфер обмена подразумевает дополнительный расход памяти из-за необходимости дублировать блоки данных. При некоторых обстоятельствах этот фактор может сыграть весьма существенную роль, особенно если копируемые блоки достаточно велики. Одно из возможных решений заключается в том, чтобы передавать данные в буфер обмена, не сохраняя их копии, что полностью снимает проблему.

Но есть и другое решение, которое применяется, в частности, при передаче больших объемов информации: отложенное копирование. В этом случае в буфер обмена непосредственно записывается только спецификация формата, а вместо дескриптора глобального блока памяти соответствующему параметру присваивается значение NULL:

SetClipboardData (wFormat, NULL);

Когда приложение запрашивает данные, которые идентифицируются значением NULL, а не дескриптором блока, Windows расценивает это как запрос отложенного копирования и вызывает владельца буфера обмена (приложение, поместившее в него данные) с помощью сообщения WM_RENDERFORMAT, в котором запрашиваемый формат задан посредством параметра wParam.

В ответ на сообщение WM_RENDERFORMAT приложение вызывает функцию SetClipboardData(), которой передается дескриптор глобального блока памяти и идентификатор формата (вместо вызова функций OpenClipboard() и EmptyClipboard()). Таким образом, реальные данные передаются только тогда, когда получатель готов принять их.

В буфер обмена можно поместить несколько элементов в виде комбинации обычных и отложенных данных.Когда приложение перестает быть владельцем буфера обмена, Windows посылает ему сообщение WM_DESTROYCLIPBOARD, констатирующее данный факт. В ответ на это сообщение приложение может вернуть себе право на владение буфером обмена и снова передать в него те же самые данные, однако такой способ рекомендуется применять лишь в исключительных ситуациях.

Кроме того, если приложение готово завершить свою работу, но является владельцем буфера обмена, который содержит дескрипторы данных со значением NULL, Windows посылает сообщение WM_RENDERALLFORMATS без указания каких-либо спецификаций формата. В ответ на это сообщение приложение-владелец должно либо полностью очистить буфер обмена, либо завершить отложенные вызовы.

Но в этом случае, в отличие от реакции на сообщение WM_RENDERFORMAT, приложение, прекращающее свою работу, не будет вызывать функцию SetClipboardData(), а просто очистит буфер обмена и целиком запишет в него новые данные, как если бы отложенные вызовы вообще не существовали.

Еще один частный формат данных буфера обмена объявляется следующим образом:

SetClipboardData (CF_OWNERDISPLAY, NULL);

Данные типа CF_OWNERDISPLAY всегда передаются с глобальным дескриптором памяти, имеющим значение NULL (по аналогии с форматом отложенного копирования). Но поскольку в этом случае владелец буфера обмена несет непосредственную ответственность за отображение данных, при их запросе Windows не посылает сообщение WM_RENDERFORMAT. Просто сообщение из программы просмотра должно быть передано непосредственно программе-владельцу буфера обмена.

Как было сказано ранее, идентифицировать владельца буфера можно с помощью функции GetClipboardOwner(). И наоборот, программа-владелец может определить программу просмотра с помощью функции GetClipboardViewer().

При использовании данного формата программа просмотра посылает приложению-владельцу запрос на формирование реального изображения и предоставляет ему доступ к своему окну. Программа просмотра может послать пять различных сообщений:

- **WM_ASKCBFORMATNAME** — посылается для запроса копии имени формата у владельца буфера обмена. При этом не следует забывать, что сам буфер обмена содержит лишь идентификатор CF_OWNERDISPLAY, поэтому программа просмотра имеет возможность самостоятельно принимать решение о том, нужно ли ей знать реальный тип данных. Сообщение WM_ASKCBFORMATNAME сопровождается указанием в аргументе wParam количества копируемых байтов. Аргумент lParam содержит указатель на буфер, в который должен быть отправлен результат;
- **WM_HSCROLLCLIPBOARD / WM_VSCROLLCLIPBOARD** — посылаются в том случае, когда программа просмотра содержит вертикальную или горизонтальную полосу прокрутки и связанные с ними события должны передаваться владельцу буфера обмена. В аргументе wParam хранится дескриптор окна программы просмотра, а в аргументе lParam — стандартные идентификаторы, которые сопровождают сообщения WM_HSCROLL и WM_VSCROLL;

• **WM_PAINTCLIPBOARD** — посылаются в качестве запроса на обновление окна программы просмотра, возможно, в ответ на сообщение WM_PAINT. Аргумент wParam содержит дескриптор окна программы просмотра. Аргумент lParam представляет собой глобальный дескриптор объекта, содержащего структуру PAINTSTRUCT, которая задает обновляемую область экрана. Чтобы определить, вся ли рабочая область окна требует обновления, владелец буфера обмена должен сравнить размеры области рисования, которые содержатся в поле *repaint* структуры PAINTSTRUCT, с размерами, указанными в последнем сообщении WM_SIZECLIPBOARD;

• **WM_SIZECLIPBOARD** — посылаются для указания того, что окно программы просмотра изменило свои размеры. Аргумент wParam содержит дескриптор окна просмотра. Аргумент lParam представляет собой глобальный дескриптор объекта, содержащего структуру RECT, которая задает обновляемую область.

В ответ на любое из этих сообщений владелец буфера обмена должен вызвать функцию InvalidateRect() или обновить окно просмотра и соответствующим образом установить позицию бегунков полос прокрутки.

Приложения могут также объявлять ***собственные частные форматы буфера обмена***, зарегистрировав новый формат с помощью функции RegisterClipboardFormat():

```
wFormat = RegisterClipboardFormat (lpszFormatTitle);
```

Новый идентификатор wFormat получает значение из диапазона от 0×C000 до 0×FFFFи может использоваться как параметр функций SetClipboardData() и GetClipboardData(). Чтобы другие приложения или другой экземпляр того же приложения смогли прочитать данные из буфера обмена в этом формате, им необходимо предоставить аналогичный идентификатор wFormat. Его значение может передаваться через буфер обмена в формате CF_TEXT.

В качестве альтернативы можно использовать функцию EnumClipboardFormats(), которая рассматривалась ранее. Эта функция возвращает идентификаторы всех форматов, после чего следует вызвать функцию GetClipboardFormatName(), с тем чтобы получить имя формата в виде ASCII-строки:

```
GetClipboardFormatName (wFormat, lpszBuffer, nCharCount);
```

Следует подчеркнуть, что Windows не требует никакой информации о структуре данных, которые передаются при использовании частных форматов. Интерпретация таких данных находится исключительно в компетенции самого приложения. Операционная система должна знать только имя формата и дескриптор блока памяти.

§ 8.2. Обмен данными посредством каналов

8.2.1. Общие положения и классификация каналов

Процессы, запущенные пользователем, не всегда имеют отношение друг к другу, однако способность процессов обмениваться информацией иногда значительно повышает эффективность приложения и делает возможным решение задач, которые были бы не под силу совокупности независимых процессов. Например, программа набора телефонного номера и графический редактор имеют очень мало общего, однако если графический редактор должен передавать информацию в удаленную систему, целесообразность его связи с программой набора номера уже не вызывает сомнения. Windows обеспечивает много возможностей для связи между приложениями (например, через буфер обмена, DDE, OLE и т. д.).

Каналы представляют собой удобный механизм для обмена информацией между процессами в различных программах [12]. В отличие от некоторых альтернативных механизмов с каналами не связаны формальные стандарты или протоколы передачи информации. Эта особенность каналов делает их более удобными в использовании и более гибкими по сравнению, скажем, с механизмом DDE. Однако она же ограничивает возможность применения каналов только программами, способными правильно распознавать информацию, которой они обмениваются.

Поддержка каналов была впервые введена в Windows NT, а впоследствии в Windows 95 и Windows 98 (в виде анонимных каналов). Однако только Windows NT поддерживает возможность использования именованных каналов, которые неприменимы в Windows 9x.

Канал представляет собой участок совместно используемой памяти, где процессы оставляют свои сообщения друг для друга. Канал подобен файлу, данные в который записываются одной программой, а считываются другой. Поскольку каналы предназначены для связи между процессами, Win32API предоставляет набор команд, обеспечивающих обмен информацией. Концептуально канал является гибридом файла и электронного почтового ящика. Один процесс записывает в этот файл определенную информацию, а другой просматривает ее.

Канал появляется после того, как одна из программ принимает решение создать его. Программа, которая создает канал, называется ***сервером*** канала. Другие процессы, которые называются ***клиентами***, могут связываться с каналом на другом его конце. Сервер поддерживает жизнеспособность канала. Любой процесс может служить как сервером, так и клиентом, или одновременно и сервером, и клиентом для разных каналов.

После того как канал будет создан и к нему подключится другой процесс, клиент или сервер или одновременно и клиент, и сервер начнут записывать данные на своем конце канала. Информация, записанная на одном конце канала, читается на другом его конце. Операции чтения и записи выполняются с помощью тех же команд, которые применяются для работы с любыми другими файлами: ReadFile() и WriteFile(). Обычно процесс, ожидающий сообщения, создает для их получения новый поток. Этот поток периодически вызывает функцию ReadFile() и блокируется, оставаясь в пассивном состоянии до поступления нового сообщения.

Наконец, сервер принимает решение о том, что диалог закончен, и разрывает соединение. Для уничтожения канала сервер вызывает функцию CloseHandle(). Но канал на самом деле не будет уничтожен до тех пор, пока не будут закрыты все дескрипторы, указывающие на него как со стороны клиента, так и со стороны сервера. Кроме того, сервер может подключить к старому каналу нового клиента.

Каналы имеют несколько разновидностей [12]:

- входные, выходные и дуплексные (двунаправленные);
- байтовые и каналы сообщений;
- блокируемые и неблокируемые;
- именованные и анонимные.

Большинство атрибутов канала определяется при его создании. Рассмотрим основные виды каналов более подробно.

Входные, выходные и дуплексные каналы. Названия каналов, относящихся к первой группе, указывают на направление передачи информации. ***Входные*** и ***выходные*** каналы являются однонаправленными: с одной их стороны осуществляется запись информации, с другой — чтение. Входной канал позволяет клиенту передавать, а серверу — принимать информацию; выходной, наоборот, серверу — передавать, а клиенту — принимать. ***Дуплексный*** канал позволяет передавать и принимать данные обеим сторонам.

Байтовые каналы и каналы сообщений. Характер записываемой информации определяет, в каком режиме, ***байтового канала*** или ***канала сообщений***, будет происходить чтение. Режим канала помогает системе принять решение о том, когда следует остановить операцию чтения. В режиме байтового канала чтение информации прекращается при поступлении последнего байта из канала или при заполнении его буфера чтения. В режиме сообщений чтение информации прекращается при достижении конца сообщения.

Внутри канала, работающего в режиме сообщений, система размещает в начале каждого записываемого фрагмента заголовок, хранящий сведения о длине этого фрагмента информации. Программы, которые находятся на концах канала, не видят этого заголовка, однако команда ReadFile() прекращает чтение, достигая конца очередного фрагмента данных.

Блокируемые и неблокируемые каналы. Каналы бывают ***блокируемыми*** или ***неблокируемыми***. Данный атрибут влияет на результаты выполнения операций чтения, записи и соединения. В случае ошибки выполнения любой из этих функций в неблокируемом канале возвращается код завершения, представляющий собой код ошибки. В канале, который допускает блокировку, функции не возвращают результата до тех пор, пока они не завершатся, или до тех пор, пока существует ошибка.

Именованные и анонимные каналы. Канал может быть ***именованным***, если программа-создатель снабдила его строковым значением, однозначно идентифицирующим его имя, или ***анонимным***, если с ним не связано никакое идентифицирующее строковое значение. Подобно синхронизирующим объектам (простым и исключающим семафорам), имя канала помогает другим процессам правильно его распознавать. Windows 98 не поддерживает именованных каналов — в этой операционной системе все каналы являются анонимными.

Анонимные каналы занимают меньше системных ресурсов, однако по сравнению с именованными они выполняют лишь ограниченный набор задач. В частности, они могут передавать сообщения только в одном направлении: или от клиента к серверу, или от сервера к клиенту (но не в обоих направлениях одновременно). Кроме того, анонимные каналы не работают в сетях. Клиент и сервер должны находиться на одном компьютере.

Именованные каналы, которые поддерживаются в Windows NT, выполняют некоторые задачи, недоступные для анонимных каналов. Так, они передают информацию в обоих направлениях, соединяют посредством сети процессы, происходящие на различных удаленных компьютерах. Кроме того, именованные каналы могут существовать в нескольких экземплярах и имеют дополнительные режимы работы. Способность существовать в нескольких экземплярах позволяет именованным каналам соединять один сервер с несколькими клиентами. Каждый экземпляр канала представляет собой отдельную независимую линию связи. Сообщения одного экземпляра канала не смешиваются с сообщениями другого экземпляра. Несколько экземпляров канала создаются в том случае, когда один или несколько серверов передают одно и то же идентифицирующее имя в качестве аргумента функции CreateNamedPipe().

8.2.2. API-функции для работы с каналами

Как уже было сказано, основное предназначение каналов — поддержка связи между процессами. Канал представляет собой буфер в памяти, где система сохраняет данные в период между записью их одним процессом и чтением другим. API-команды позволяют трактовать буфер как канал, или трубу, по которой информация «перетекает» из одного места в другое. Канал имеет два конца. Односторонний канал позволяет записывать данные только на одном конце и читать их на другом, т. е. вся информация «перетекает» от одного процесса к другому. Двухсторонние каналы дают возможность обоим процессам выполнять как запись, так и чтение информации, поэтому информация может «перетекать» одновременно в двух направлениях. Кроме того, при создании канала необходимо принять решение о том, будет он анонимным или именованным [12].

Создание анонимных каналов. Анонимный канал передает информацию только в одном направлении, причем оба конца канала должны находиться на одном и том же локальном компьютере. Процесс, создающий анонимный канал, получает два дескриптора: для записи и для чтения. Чтобы связаться с каким-либо процессом, сервер должен передать ему один из дескрипторов:

```
BOOL CreatePipe (PHANDLE phRead,        // переменная для дескриптора чтения (входной канал)
      PHANDLE phWrite,                  // переменная для дескриптора записи (выходной канал)
      LPSECURITY_ATTRIBUTES lpsa,       // привилегии доступа
      DWORD dwPipeSize);                // размер буфера канала (0 по умолчанию)
```

От размера буфера канала зависит, какой максимальный объем информации может находиться в канале. Передача сообщений по заполненному каналу невозможна до тех пор, пока на другом конце канала не будет прочитана старая информация.

При успешном завершении функция CreatePipe() возвращает значение TRUE и записывает два новых дескриптора в переменные, заданные параметрами типа PHANDLE. После этого процесс, создавший канал, обычно передает один из дескрипторов другому процессу. Какой из дескрипторов вы будете передавать, зависит от того, что должен делать процесс посредством данного канала: посылать (записывать) или получать (читать) информацию. Вы можете передать дескриптор дочернему процессу с помощью его командной строки или воспользовавшись стандартными дескрипторами ввода / вывода. Несвязанные процессы должны передавать друг другу дескрипторы

иными средствами, например с помощью механизма DDE или посредством совместно используемого файла. Связь по анонимному каналу проще реализовать в том случае, если процессы связаны друг с другом [12].

Создание именованных каналов. Многим системным объектам Windows NT присваиваются идентифицирующие их строковые значения. Хотя команды создания именованных каналов могут компилироваться только в среде Windows NT, приложения, использующие именованные каналы, работают и в среде Windows 98, где именованные каналы функционируют точно так же, как и анонимные. Преимущество применения именованных каналов заключается в том, что имена каналов позволяют процессам локализовать эти объекты более простым способом. Анонимные объекты распознаются только по своим дескрипторам, а дескрипторы действительны лишь в том процессе, где они были созданы. С другой стороны, любой процесс, который знает имя объекта, может с помощью операционной системы определить его место в иерархии. Система может найти по имени любой объект на другом сетевом компьютере.

Если канал имеет имя, программа-клиент не должна ожидать, когда сервер передаст ей дескриптор этого канала. Вместо этого клиент может запросить дескриптор, вызвав функцию CreateFile() или CallNamedPipe(). В любом случае клиенту достаточно знать имя канала. Родитель посредством командной строки может передать своему дочернему процессу строковое значение, содержащее имя канала. Кроме того, это значение может передаваться от одного произвольного процесса другому не связанному с ним процессу с помощью совместно используемого файла или механизма DDE. Однако чаще всего процессы, совместно использующие именованный канал, создаются одним и тем же разработчиком, поэтому они заранее знают имя канала.

Перечисленные далее функции работают только с именованными каналами (не создавайте анонимных каналов, если хотите воспользоваться операциями, которые выполняются с помощью этих функций):

CallNamedPipe()	DisconnectNamedPipe()	RevertToSelf()
ConnectNamedPipe()	GetNamedPipeHandleState()	SetNamedPipeHandleState()
CreateFile()	GetNamedPipeInfo()	TransactNamedPipe()
CreateNamedPipe()	ImpersonateNamedPipeClient()	WaitNamedPipe()

Рассмотрим основные функции более подробно. Для создания именованных каналов предназначена функция CreateNamedPipe():

```
HANDLE CreateNamedPipe (LPTSTR lpszPipeName,     // строка с именем нового канала
        DWORD fdwOpenMode,                       // доступ, перекрытие и сквозная запись
        DWORD fdwPipeMode,                       // тип, режимы чтения и ожидания
        DWORD dwMaxInstances,                    // максимальное число экземпляров
        DWORD dwOutBuf,                          // размер выходного буфера, байты
        DWORD dwInBuf,                           // размер входного буфера/байты
        DWORD dwTimeout,                         // время паузы, миллисекунды
        LPSECURITY_ATTRIBUTES lpsa);             // привилегии доступа
```

Поскольку именованные каналы имеют больше возможностей по сравнению с анонимными каналами, функция CreateNamedPipe() имеет больше параметров, чем функция CreatePipe(). Рассмотрим значения параметров этой функции.

Параметр lpszPipeName указывает на строковое значение, которое будет играть роль имени нового объекта. Операционная система записывает это имя в иерархическую структуру имен системных объектов. Строки, содержащие имя канала, должны иметь следующий синтаксис:

```
\\.\pipe\<имя канала>
```

Первый символ обратной косой черты указывает на корневой узел иерархической структуры имен системных объектов. Еще три обратные косые разделяют имена последующих узлов. Точка (.) обозначает локальный компьютер. Хотя каналы могут соединяться с клиентами других сетевых серверов, новый объект-канал всегда появляется на том локальном сервере, где он был создан.

После имени сервера указывается узел, который называется pipe. Здесь содержатся имена всех каналов, расположенных на локальном компьютере. В строке имени программно задается только один фрагмент, а именно *<имя канала>*. Эта подстрока должна содержать не более 256 сим-

волов. Регистр при наборе символов не имеет значения, поскольку имена объектов нечувствительны к нему. И серверы, и клиенты для представления локального компьютера используют точку (.), однако клиент, который хочет открыть канал на удаленном сервере, должен знать имя этого сервера. Один из способов определения имени удаленного сервера заключается в получении списка имен с помощью функций WNetOpenEnum(), WNetEnumResource() и WNetCloseEnum(), однако делается это очень медленно.

Функции CreateNamedPipe(), CreateFile(), WaitNamedPipe() и CallNamedPipe() требуют, чтобы в качестве параметра было задано имя канала.

Параметр fdwOpenMode содержит комбинацию флагов, которые задают некоторые атрибуты канала. Наиболее важным атрибутом является режим доступа, определяющий направление передачи информации в канале. Параметр fdwOpenMode должен содержать один из трех возможных флагов доступа:

- **PIPE_ACCESS_OUTBOUND** — сервер только записывает, а клиент только читает информацию;
- **PIPE_ACCESS_INBOUND** — сервер только читает, а клиент только записывает информацию;
- **PIPE_ACCESS_DUPLEX** — сервер и клиент могут как записывать, так и читать информацию.

Еще два флага этого параметра являются необязательными:

- **FILE_FLAG_WRITE_THROUGH** — запрещает буферизацию в сети;
- **FILE_FLAG_OVERLAPPED** — разрешает асинхронные операции чтения и записи.

С целью повышения эффективности работы канала при соединении с удаленным компьютером система старается не сразу же отсылать каждое сообщение. Она, наоборот, пытается накопить несколько коротких сообщений в буфере и передать их по каналу за одну операцию. Если проходит определенное время, а буфер остается заполненным лишь частично, система все же передает его содержимое. Флаг FILE_FLAG_WRITE_THROUGH запрещает системе производить буферизацию. Каждое сообщение будет отправляться немедленно, а команды записи не завершатся до тех пор, пока не будут переданы их выходные значения. Буферизацию целесообразно отключать в том случае, если вы собираетесь посылать сообщения лишь изредка.

Операции записи и чтения обычно выполняются медленно, поскольку работа осуществляется с физическими устройствами. Второй необязательный флаг, FILE_FLAG_OVERLAPPED, позволяет командам чтения и записи завершаться, в то время как инициированные ими действия будут продолжаться в фоновом режиме (асинхронный ввод/вывод, рассмотренный ранее).

Параметр fdwPipeMode содержит еще одну комбинацию флагов, позволяющую задать другой набор характеристик канала: флаг режима чтения, флаг режима ввода и флаг ожидания. Флаги режима чтения и режима ввода тесно связаны друг с другом, поэтому более удобно было бы назвать их соответственно флагом чтения и флагом записи. Они совместно определяют принципы организации и интерпретации данных, передаваемых по каналу. Оба флага допускают выбор между байтовым режимом и режимом сообщений.

К флагам ввода (режим записи) относятся PIPE_TYPE_BYTE и PIPE_TYPE_MESSAGE, а к флагам режима чтения — PIPE_READMODE_BYTE и PIPE_READMODE_MESSAGE.

В каналах, работающих в байтовом режиме, информация записывается и читается в обычном двоичном виде и воспринимается просто как последовательность байтов.

Однако иногда бывает удобным разбить передаваемую по каналу информацию на дискретные сообщения, в которых результат каждой операции записи соответствует отдельному сообщению. Каналы, работающие в режиме сообщений, автоматически и незаметно для пользователя предваряют каждое новое сообщение невидимым заголовком, задающим длину сообщения. Заголовок обеспечивает автоматическую остановку операции чтения при достижении конца сообщения. Получатель читает сообщения одно за другим точно в том виде, в котором они записывались в канал.

Так, если одна программа посылает другой длинную последовательность целых чисел, ей целесообразно выбрать байтовый режим канала, поскольку для получателя не имеет значения, сколько целых чисел будет передано в канал в течение одной операции записи. Он просто поочередно читает целые числа из канала. Однако если программа посылает команды, написанные на языке сценариев, адресат должен получать эти команды целиком, в том виде, в котором они были записаны, чтобы правильно произвести их синтаксический анализ. Поскольку разные команды могут иметь различную длину, обе программы должны использовать канал, работающий в режиме сообщений.

Режимы записи и чтения назначаются независимо друг от друга, однако не все комбинации являются допустимыми. В частности, нельзя одновременно использовать флаги PIPE_TYPE_BYTE и PIPE_READMODE_MESSAGE. Канал байтового типа осуществляет запись информации, не разделяя сообщения заголовками, поэтому на приемном конце канала эта информация не может быть разбита на отдельные сообщения. С другой стороны, комбинация флагов PIPE_TYPE_MESSAGE и PIPE_READMODE_BYTE является вполне допустимой. В этом случае программа-отправитель записывает в канал сообщения, сопровождая их заголовками, однако получатель игнорирует эти заголовки, принимая данные в виде последовательности неразделенных байтов (иными словами, получатель не воспринимает невидимые заголовки сообщений).

Наряду с флагами режимов записи и чтения параметр fdwPipeMode может содержать еще один флаг, устанавливающий для канала режим ожидания, который определяет, что произойдет с каналом, если какие-то условия временно воспрепятствуют завершению команды. Например, при попытке прочитать информацию из пустого канала некоторые команды должны забыть о незавершенной операции чтения и перейти к выполнению следующих операторов, в то время как другие программы, прежде чем продолжить работу, должны дождаться поступления считываемого сообщения.

По умолчанию каналы заставляют читающие потоки блокироваться и ожидать поступления сообщения, однако вы можете предотвратить их блокировку, добавив в параметр fdwPipeMode флаг PIPE_NOWAIT. По умолчанию указанный параметр содержит флаг PIPE_WAIT. Режим ожидания влияет как на команды записи, так и на команды чтения. Программа, пытающаяся записать информацию при заполненном буфере канала, обычно блокируется до тех пор, пока другая программа не освободит место в буфере, прочитав сообщение на другом конце канала. Режим ожидания также влияет и на работу сервера, который пытается соединиться с клиентом. Если функция ConnectNamedPipe() не обнаруживает клиента, готового к соединению, режим ожидания канала определяет, будет ли команда ожидать готовности клиента, чтобы установить с ним соединение, или же она немедленно завершится.

Параметр dwMaxInstances. Программа-сервер может открывать каналы для нескольких клиентов, однако она должна заранее знать, со сколькими клиентами ей придется поддерживать связь. Вводить новое имя канала для каждого клиента неудобно. Каким же образом клиенты будут заранее узнавать, какое имя канала они должны использовать, открывая соединение на своем конце? Чтобы разрешить эту проблему, Win32 позволяет серверу многократно создавать один и тот же канал.

Каждый раз при вызове функции CreateNamedPipe() с одним и тем же именем вы получаете новый экземпляр данного канала. Каждый такой экземпляр обеспечивает независимую линию связи с другим клиентом. Кроме того, сервер может заранее несколько раз создать один и тот же канал. Он получит несколько различных дескрипторов и будет ожидать подключения программ-клиентов на других концах каналов. Все программы-клиенты для получения своих дескрипторов используют одно и то же имя канала, однако каждая из них получает дескриптор, указывающий на другой экземпляр канала. Если после того, как все каналы будут заняты, следующая программа-клиент попытается установить соединение, она будет заблокирована до тех пор, пока сервер не разорвет связь с одним из предыдущих экземпляров канала.

Параметр dwMaxInstances функции CreateNamedPipe() устанавливает верхний предел количества экземпляров одного канала, которые будут поддерживаться системой до тех пор, пока функция CreateNamedPipe() не возвратит код ошибки. Флаг PIPE_UNLIMITED_INSTANCES свидетельствует об отсутствии верхнего предела. В этом случае максимальное число экземпляров ограничивается только системными ресурсами. Значение параметра dwMaxInstances не может превышать значения PIPE_UNLIMITED_INSTANCES. В файле Winbase.H значение параметра dwMaxInstances задано равным 255.

Параметры dwOutBuf и dwInBuf задают начальные размеры буферов, в которые производится запись информации. Для выходного канала (PIPE_ACCESS_OUTBOUND) значение имеет только размер выходного буфера, для входного канала — только размер входного буфера.

Пределы, заданные параметрами буфера, достаточно гибки. Каждая операция чтения или записи заставляет операционную систему выделить место для буфера из динамической области системной памяти ядра. Значение, определяющее размер буфера, интерпретируется как квота, ограничивающая выделенный объем памяти. Когда система предоставляет место для буфера операций

записи, она анализирует заданные пределы буфера. Если размер нового буфера не превышает заданный, операция выполняется корректно. Но даже если размер созданного буфера не укладывается в заданные рамки, система все равно выделяет для него место за счет ресурсов процесса. Во избежание перегрузки ресурсов процесса каждая операция WriteFile(), вызывающая превышение выделенного размера буфера, блокируется. Записывающий поток приостанавливается до тех пор, пока поток-получатель не уменьшит объем буфера, прочитав из него достаточный объем информации.

При оценке размеров буфера необходимо принимать во внимание тот факт, что реальный размер всегда должен быть немного больше расчетного, поскольку каждое сообщение сопровождается дополнительной внутренней структурой данных размером приблизительно в 28 байтов. Точный размер этой структуры не документируется и может изменяться от версии к версии.

Итак, операционная система при необходимости динамически выделяет системную память для буфера, однако потоки, для работы которых требуется объем памяти, превышающий выделенный размер буфера, могут быть заблокированы. От увеличения размера буфера выигрывают те программы, которые часто отправляют сообщения или требуют периодического дублирования буфера.

Параметр dwTimeout имеет значение только в том случае, если для установки соединения клиент вызывает функцию WaitNamedPipe(), которая изменяет период паузы, заданный по умолчанию. По умолчанию принимается значение, установленное сервером в качестве параметра dwTimeout функции CreateNamedPipe(), однако с помощью функции WaitNamedPipe() клиент может задать другое значение.

Параметр lpsa представляет собой указатель на структуру SECURITY_ATTRIBUTES, которая вам уже знакома. Значения переменных этой структуры определяют, какие операции можно выполнять с объектом, доступ к которому осуществляется посредством нового дескриптора, и может ли новый дескриптор наследоваться дочерним процессом. Как обычно, если в этом поле будет оставлено значение NULL, результирующий дескриптор предоставит привилегии полного доступа и не станет наследоваться.

Следует отметить, что ***анонимные каналы*** имеют характеристики, которые описывают состояние именованных каналов, заданное по умолчанию, а именно: PIPE_TYPE_BYTE, PIPE_READMODE_BYTE, PIPE_WAIT, запрет асинхронного ввода/вывода и разрешение буферизации данных в сети.

При успешном выполнении функция CreateNamedPipe() возвращает соответствующий дескриптор, при наличии ошибки — дескриптор 0xFFFFFFFF,т. е.INVALID_HANDLE_VALUE). В приложениях, скомпилированных в среде Windows 8, функция CreateNamedPipe() всегда возвращает сообщение об ошибке (INVALID_HANDLE_VALUE).

Сервер, открывший именованный канал, должен ожидать, пока этот канал не будет открыт на другом конце программой-клиентом. Клиент может открыть свой конец канала несколькими различными способами, однако чаще всего для этой цели используется функция CreateFile(). Эта же функция применяется и для открытия файлов на диске. Она также работает с именованными каналами, устройствами связи и буферами ввода/вывода символьного консольного окна. С указанным набором объектов работают команды ReadFile() и WriteFile(). Применение унифицированного набора API-функций значительно упрощает задачу программирования. Описание данных функций уже приводилось, но поскольку для работы в каналах используется определенный набор флагов, рассмотрим функцию CreateFile() еще раз:

```
HANDLE CreateFile (LPCTSTR lpszName,      //имя канала (или файла)
        DWORD fdwAccess,                  //доступ для чтения/записи (должен
                                          // соответствовать атрибутам канала)
        DWORD fdwShareMode,               //для каналов обычно значение этой
                                          //переменной равно 0 (совместное
                                          //использование не допускается)
        LPSECURITY_ATTRIBUTES lpsa,       //привилегии доступа
        DWORD fdwCreate,                  //для каналов эта переменная должна иметь
                                          //значение OPEN_EXISTING
        DWORD fdwAttrsAndFlags,           //режимы сквозной записи и перекрытия
        HANDLE hTemplateFile);            //игнорируется при наличии флага OPEN_EXISTING
```

Имя канала должно соответствовать строковому значению, переданному сервером функции CreateNamedPipe(). Если соединение между программой-сервером и программой-клиентом устанавливается по сети, в этой строке вместо точки (.) указывается сетевое имя серверного компьютера.

Параметр fdwAccess определяет, с какой целью (для чтения или для записи) будет использоваться канал. Если канал был создан с флагом PIPE_ACCESS_OUTBOUND, при вызове функции CreateFile() должно быть задано значение GENERIC_READ. Для работы с входным каналом клиент должен иметь привилегии класса GENERIC_WRITE, для работы с двухсторонним каналом — привилегии класса GENERIC_READ | GENERIC_WRITE.

Параметр fdwShareMode(), как правило, имеет значение 0, предотвращая, таким образом, совместное использование канала другими процессами. Однако иногда клиент при необходимости продублировать дескриптор канала для другого клиента вынужден применять режим совместного использования. В этом случае оба клиента будут иметь дескрипторы одного и того же экземпляра того же самого канала, поэтому необходимо принять меры для синхронизации операций записи и чтения.

Атрибуты безопасности, хранящиеся в параметре lpsa, вам уже знакомы. Флаг fdwCreate должен иметь значение OPEN_EXISTING, поскольку функция CreateFile() не создает новый канал, а просто открывает уже существующий. Остальные флаги позволяют функции CreateFile() создавать новые файловые объекты, однако если параметр lpszName соответствует объекту-каналу, использование этих флагов приведет к ошибке.

Последние два параметра функции CreateFile() обычно управляют атрибутами файла, например атрибутом скрытости, архивности или доступа только для чтения, однако функция CreateFile() использует эти атрибуты только при создании новых файлов. При открытии существующего файла (флаг OPEN_EXISTING) объект сохраняет свои атрибуты. Но есть два исключения из этого правила. При открытии существующего именованного канала срабатывают два флага параметра fdwAttrsAndFlags: FILE_FLAG_WRITE_THROUGH и FILE_FLAG_OVERLAPPED. Клиент может установить флаги, которые отличаются от флагов, установленных сервером, разрешая или запрещая буферизацию в сети и асинхронный ввод / вывод в соответствии со своими потребностями.

На двух концах одного и того же канала могут быть установлены разные режимы чтения или записи, однако при открытии дескриптора для клиента функция CreateFile() всегда копирует исходные атрибуты. В то же время ***каждый процесс может изменить свой дескриптор канала*** с помощью функции SetNamedPipeHandleState():

```
BOOL SetNamedPipeHandleState(HANDLE hNamedPipe,     // дескриптор именованного канала
      LPDWORD lpdwModes,                            // флаги режима чтения и ожидания
      LPDWORD lpdwMaxCollect,                       // размер передающего буфера
      LPDWORD lpdwCollectDataTimeout);              // максимальное время, оставшееся
                                                    // до начала передачи
```

Параметр hNamedPipe представляет собой дескриптор, возвращаемый функцией CreateNamedPipe() или CreateFile().

Параметр lpdwModes, подобно параметру fdwPipeMode функции CreateNamedPipe(), содержит комбинацию флагов, которые позволяют одновременно задать несколько атрибутов. ***Параметр lpdwModes*** определяет, какой режим — байтовый или режим сообщений — будет использоваться при операциях чтения и будут ли блокироваться некоторые команды в ожидании доступа к каналу. Режим чтения задается флагом PIPE_READMODE_BYTE или PIPE_READMODE_MESSAGE. (Если задать режим сообщений для канала, который был создан с флагом PIPE_READMODE_BYTE, функция возвратит код ошибки.) Режим чтения может использоваться в сочетании с флагом PIPE_WAIT или PIPE_NOWAIT.

Два последних параметра используются только для каналов, устанавливающих связь с удаленным компьютером. Они определяют, каким образом система будет управлять буферизацией в сети при передаче информации. Эти параметры не влияют на каналы, созданные с атрибутом PIPE_FLAG_WRITE_THROUGH, который запрещает буферизацию в сети. Буферизация дает возможность системе комбинировать несколько сообщений и передавать их одновременно за один сеанс. Причем исходящие сообщения накапливаются в буфере до тех пор, пока не произойдет заполнение буфера или не закончится заданный временной интервал. Размер накопительного бу-

фера задается параметром lpdwMaxCollect, а значение временного интервала, выраженное в миллисекундах, — параметром lpdwCollectDataTimeout. Есть три функции, позволяющие ***получить информацию о канале***, не изменяя его атрибутов [12].

Первая функция является «двойником» функции SetNamedPipeHandleState(), однако она кроме той информации, которая устанавливается функцией SetNamedPipeHandleState(), позволяет получить много дополнительных сведений:

```
BOOL GetNamedPipeHandleState(HANDLE hNamedPipe,
                                    // дескриптор именованного канала
    LPDWORD lpdwModes,              // режимы чтения и ожидания
    LPDWORD lpdwInstances,          // количество экземпляров текущего канала
    LPDWORD lpcbMaxConnect,         // максимальное количество байтов,
                                    // накапливаемых до начала удаленной передачи
    LPDWORD lpdwConnectTimeout,     // максимальное время до начала удаленной передачи
    LPTSTR lpszUserName,            // пользовательское имя процесса-клиента
    DWORD dwMaxUserNameBuff);       // размер буфера для хранения пользовательского имени,
                                    // выраженный в количестве символов
```

Параметр lpdwModes может содержать флаги PIPE_READMODE_MESSAGE и PIPE_NOWAIT. Если ни один из этих флагов не установлен, применяются байтовый режим и режим ожидания, задаваемые по умолчанию.

Параметр lpdwInstances позволяет подсчитать текущее количество экземпляров канала. Иными словами, этот параметр сообщает, сколько раз сервер вызывал функцию CreateNamedPipe() с одним и тем же именем канала.

Параметры lpcbMaxConnect и lpdwConnectTimeout возвращают ту же информацию о сетевой буферизации, которая задается функцией SetNamedPipeHandleState().

Последние два параметра указанной функции дают серверу возможность получить информацию о клиенте. Они возвращают завершающиеся нулевым символом строки, которые идентифицируют пользователя, запустившего приложение-клиента, т. е. имя пользователя, введенное при регистрации. Это имя связано с определенными конфигурацией системы и привилегиями безопасности. Имя пользователя может понадобиться серверу для регистрации или отчетности, но скорее всего этот параметр введен для совместимости с OS/2, где он также используется. ***Параметр*** lpszUserName должен иметь значение NULL, если дескриптор hNamedPipe принадлежит клиенту или, иными словами, если он был создан функцией CreateFile (а не функцией CreateNamedPipe).

Любому из параметров-указателей может быть присвоено значение NULL, что даст возможность проигнорировать значение, которое устанавливается по умолчанию.

Дополнительная информация о канале может быть возвращена и посредством функции GetNamedPipeInfo(). Данная функция возвращает атрибуты, которые нельзя изменить, чем и отличается от функции GetNamedPipeHandleState(), которая возвращает атрибуты, изменяемые во время существования канала:

```
BOOL GetNamedPipeInfo(HANDLE hNamedPipe,    // дескриптор именованного канала
    LPDWORD lpdwType,                       // тип и флаги сервера
    LPDWORD lpdwOutBuf,                     // размер выходного буфера канала, байты
    LPDWORD lpdwInBuf,                      // размер входного буфера канала, байты
    LPDWORD lpdwMaxInstances);              // максимальное количество экземпляров канала
```

Параметр lpdwType может содержать либо один из флагов PIPE_TYPE_MESSAGE и PIPE_SERVER_END, либо оба флага одновременно. Если ни один из флагов не установлен, дескриптор подключается к клиентскому концу канала, который записывает информацию в байтовом режиме. Размеры входного и выходного буферов задаются функцией CreateNamedPipe().

Параметр lpdwMaxInstances возвращает заданное функцией CreateNamedPipe() значение, которое определяет максимальное количество экземпляров канала, способных существовать одновременно.

Обычно при выполнении операции чтения из канала прочитанное сообщение удаляется из буфера. Однако с помощью функции PeekNamedPipe() сообщение можно прочитать, не удаляя его из буфера. Данная функция работает как с анонимными, так и с именованными каналами:

```
BOOL PeekNamedPipe(HANDLE hPipe,          //дескриптор именованного или анонимного канала
    LPVOID lpvBuffer,                     //адрес буфера для получения данных
    DWORD dwBufferSize,                   //размер буфера, байты
    LPDWORD lpdwBytesRead,                //возвращает количество прочитанных байтов
    LPDWORD lpdwAvailable,                //возвращает полное количество байтов
    LPDWORD lpdwMessage);                 //возвращает количество непрочитанных байтов
                                          //данного сообщения
```

Параметр lpvBuffer указывает место, куда функция PeekNamedPipe() может записывать информацию, прочитанную из канала. Следует отметить, что функция PeekNamedPipe() не сможет прочитать больше байтов, чем задано параметром dwBufferSize, даже если в канале еще осталась информация.

Параметр lpdwBytesRead возвращает данные о количестве байтов, которые функция действительно прочитала, а параметр lpdwMessage — данные о количестве байтов, оставшихся в этом сообщении. ***Параметр*** lpdwMessage игнорируется, если канал находится в режиме чтения PIPE_READMODE_BYTE. В этом случае разбивка информации на сообщения не производится. (Байтовый режим чтения применяется всеми анонимными каналами.)

Параметр lpdwAvailable возвращает информацию о суммарном количестве байтов, содержащихся во всех сообщениях. Если в данный момент в буфере находится несколько сообщений, значение *lpdwAvailable может превышать сумму значений параметров *lpdwBytesRead и *lpdwMessage.

Допускается возвращение только части информации с сохранением значения NULL для остальных параметров. Например, если вам достаточно иметь информацию лишь о том, сколько байтов данных находится в буфере, всем параметрам, кроме lpdwAvailabie, можно задать значение 0 или NULL.

Если канал находится в режиме чтения сообщений, функция PeekNamedPipe() всегда останавливается, прочитав первое сообщение, даже если в буфере осталось место для нескольких сообщений. Кроме того, функция PeekNamedPipe() никогда не блокирует пустой канал, как это делает функция ReadFile() при установленном флаге PIPE_WAIT. Режим ожидания не влияет на функцию PeekNamedPipe(), которая всегда возвращает результат немедленно.

Все операции, которые выполнялись при создании канала: именованного или анонимного, блокируемого или неблокируемого, байтового канала или канала сообщений,—являлись подготовительными, обеспечивающими возможность передачи данных по этому каналу. Для операций непосредственно чтения — записи используются функции WriteFile() и ReadFile(). Данные функции, как и CreateFile(), рассмотрены ранее при изучении файлового ввода/вывода. Однако рассмотрим их еще раз с учетом специфики каналов:

```
BOOL WriteFile(HANDLE hFile,              // куда записывать (канал или файл)
    CONST VOID *lpBuffer,                 //указывает данные, которые должны быть записаны в файл
    DWORD dwBytesToWrite,                 // количество записываемых байтов
    LPDWORD lpdwBytesWritten,             // возвращает количество записанных байтов
    LPOVERLAPPED lpOverlapped);           // задает поддержку асинхронного ввода/вывода
BOOL ReadFile(HANDLE hFile,               // источник для чтения данных (канал или файл)
    LPVOID lpBuffer,                      // буфер для временного хранения прочитанных данных
    DWORD dwBytesToRead,                  // количество байтов, которые должны быть прочитаны
    LPDWORD lpdwBytesRead,                // возвращает количество прочитанных байтов
    LPOVERLAPPED lpOverlapped);           // поддержка асинхронного ввода/вывода
```

Количество байтов, которые должны быть прочитаны или записаны, необязательно должно совпадать с размером буфера, однако превышать этот размер оно не может. Если при вызове функции ReadFile() для канала, который работает в режиме сообщений, вы зададите параметру dwBytesToRead значение меньше, чем размер следующего сообщения, функция ReadFile() прочитает только часть сообщения и возвратит значение FALSE. Если после этого проанализировать причину ошибки, вызвав функцию GetLastError(), она возвратит код ошибки ERROR_MORE_DATA. Чтобы прочитать остаток сообщения, следует снова вызвать функцию ReadFile() или PeekNamedPipe(). Если функция WriteFile() записывает данные в неблокируемый канал, который работает в байтовом режиме, и видит, что его буфер почти заполнен, она все равно возвращает значение TRUE, однако значение lpdwBytesWritten окажется меньшим, чем значение dwBytesToWrite.

В зависимости от режима ожидания канала функции WriteFile() и ReadFile() могут ***блокироваться***. Функция WriteFile() может ожидать, пока заполненный канал не освободится на другом конце. Функция ReadFile() может быть заблокирована пустым каналом вплоть до момента поступления нового сообщения.

Последний параметр обеих функций представляет собой указатель на структуру OVERLAPPED. Последняя содержит дополнительную информацию, предназначенную для поддержки асинхронного или перекрывающегося ввода/вывода, который уже был рассмотрен ранее.

Другой метод реализации асинхронного ввода/вывода заключается в использовании команд ReadFileEx() и WriteFileEx(). Вместо того чтобы сигнализировать о своем завершении с помощью события, эти команды запускают специальную функцию, которая должна вызываться в конце каждой операции.

Применение асинхронного ввода/вывода представляет собой стратегический прием, позволяющий осуществлять работу с несколькими клиентами, подключенными к различным экземплярам одного канала. Операции синхронного ввода/вывода проще программировать, однако медленные команды записи и чтения часто задерживают выполнение других операций. Сервер может создавать отдельный поток для каждого клиента, однако такой подход подразумевает гораздо большую степень избыточности, чем того требует реальная ситуация.

При асинхронном вводе/выводе один и тот же поток может одновременно записывать и читать данные в различных экземплярах канала, поскольку каждая из команд немедленно возвращает управление программе и освобождает поток, а операция ввода/вывода выполняется системой в фоновом режиме. С помощью команды WaitForMultipleObjects() поток можно блокировать вплоть до завершения всех отложенных операций. Эффективность асинхронного ввода/вывода особенно ярко проявляется при медленных сетевых соединениях. Кроме того, при наличии меньшего числа синхронизируемых потоков гораздо проще обеспечить защиту программных ресурсов.

В любой момент клиент может вызвать функцию CreateFile() для открытия своего конца именованного канала. Однако при этом могут возникнуть две проблемы [12]:

– серверу иногда необходимо знать, подключился ли клиент к каналу. От операции записи в неподключенный канал пользы немного, поэтому функция CreateFile() не сообщает серверу о наличии соединения;

– если все экземпляры канала заняты, функция CreateFile() постоянно возвращает значение INVALID_HANDLE_VALUE, не устанавливая соединения. Клиент может подождать, пока не освободится один из экземпляров канала, где будет завершена операция, инициированная другим клиентом.

Короче говоря, и сервер, и клиент должны иметь возможность ***блокироваться***, ожидая условий, при которых выполнение подключения станет возможным. Для этого сервер должен вызвать функцию ConnectNamedPipe(), а клиент — функцию WaitNamedPipe():

```
BOOL ConnectNamedPipe(HANDLE hNamedPipe,      // дескриптор доступного именованного канала
      LPOVERLAPPED lpOverlapped);             // поддержка асинхронного ввода/вывода
BOOL WaitNamedPipe(LPTSTR lpszPipeName,       // указывает на строку, которая задает имя канала
      DWORD dwTimeout);                       // максимальное время ожидания, миллисекунды
```

Эти координирующие функции работают только с именованными каналами. Поскольку клиент не может создавать собственных дескрипторов анонимных каналов, он должен получить дескриптор непосредственно от сервера, а в этом случае подключение уже выполнено.

Подобно функциям ReadFile() и WriteFile() функция ConnectNamedPipe() обеспечивает асинхронную реакцию. Параметр lpOverlapped содержит дескриптор события, которое будет служить сигналом об установке связи с клиентом.

Функционирование команды ConnectNamedPipe() зависит от того, был ли канал создан с указанием флага FILE_FLAG_OVERLAPPED и находится ли он в режиме PIPE_WAIT. Проще всего эту команду использовать с каналами, которые обеспечивают возможность ожидания.

Значения TRUE и FALSE, возвращаемые функцией ConnectNamedPipe(), являются несколько неестественными. В частности, значение TRUE возвращается только в том случае, если канал

начинает свое существование в состоянии ожидания, а клиент подключается после выполнения команды соединения, но до ее завершения. Если канал уже подключен к клиенту и команда работает асинхронно (завершается без ожидания) или если она вызывается для канала, который не допускает ожидания, функция ConnectNamedPipe() всегда возвращает значение FALSE.

Функция ***ожидания клиента*** WaitNamedPipe() реально не устанавливает никакого соединения. Она возвращает значение TRUE, если канал доступен или становится доступным впоследствии, однако дескриптора существующего канала эта функция не возвращает.

Клиент обычно циклически выполняет последовательность операций ожидания и создания до тех пор, пока не будет получен действительный дескриптор канала. Функция WaitNamedPipe() считает канал доступным только в том случае, если сервер вызывает функцию ConnectNamedPipe() в ожидании связи. Две эти функции работают вместе, ***синхронизируя*** действия клиента и сервера. Однако если сервер создает новый канал, который никогда не подключался ни к одному из клиентов, функция WaitNamedPipe() возвращает значение TRUE, даже если это противоречит результату функции ConnectNamedPipe(). Гарантировать соединение с помощью функции WaitNamedPipe() можно только в том случае, если серверу известно о доступности канала. Если клиент разрывает соединение, сервер может ошибочно отреагировать на это событие. Кроме того, при немедленном подключении нового клиента сервер не может знать о наличии нового партнера. Распознавание только новых каналов и каналов, сформированных с помощью функций ConnectNamedPipe() и WaitNamedPipe(), предотвращает отключение клиентов во время процесса обмена информацией. Следует отметить, что функция WaitForSingleObject() не работает с каналами, поскольку они не имеют сигнального состояния.

Для осуществлении связи ***по дуплексным каналам*** используется две функции — TransactNamedPipe() и CallNamedPipe(). Эти функции комбинируют операции записи и чтения в единой транзакции. Транзакции особенно полезны в сетях, поскольку они сводят к минимуму количество сеансов передачи информации.

Поддержка взаимных транзакций возможна при условии, что канал соответствует следующим условиям [12]:

- является именованным;
- использует флаг PIPE_ACCESS_DUPLEX;
- относится к типу канала сообщений;
- настроен на режим чтения сообщений.

Сервер устанавливает все эти атрибуты с помощью функции CreateNamedPipe(). При необходимости клиент может подкорректировать некоторые атрибуты с помощью функции SetNamedPipeHandleState(). Режим блокировки не влияет на выполнение команд транзакций.

Отправление запроса. Функция TransactNamedPipe() отправляет запрос по каналу и ожидает ответа. Использоваться она может как клиентами, так и серверами, хотя для первых она более полезна:

```
BOOL TransactNamedPipe(HANDLE hNamedPipe,      // дескриптор именованного канала
      LPVOID lpvWriteBuf,                      // буфер для хранения передаваемой информации
      DWORD dwWriteBufSize,                    // размер буфера записи, байты
      LPVOID lpvReadBuf,                       // буфер для полученной информации
      DWORD dwReadBufSize,                     // размер буфера чтения, байты
      LPDWORD lpdwBytesRead,                   // реально прочитанное количество байтов
                                               // (возвращаемое значение)
      LPOVERLAPPED lpOverlapped);              // поддержка асинхронного ввода / вывода
```

Несмотря на обилие параметров, функция TransactNamedPipe() действует достаточно «прямолинейно». Она записывает в канал содержимое буфера lpvWriteBuf, ожидает ответа и копирует полученное сообщение в буфер lpvReadBuf.

Функция TransactNamedPipe() выдает сообщение об ошибке в том случае, если канал имеет неправильные атрибуты или если буфер чтения слишком мал для размещения всего принятого сообщения. При этом функция GetLastError() возвращает значение ERROR_MORE_DATA и вы должны закончить чтение сообщения с помощью функций ReadFile() или PeekNamedPipe().

Функция TransactNamedPipe() осуществляет одну операцию обмена информацией по каналу. Установив соединение, программа может вызывать данную команду много раз, не разрывая связи.

Функция CallNamedPipe() предназначена для тех клиентов, которые должны выполнить однократную транзакцию. Она устанавливает соединение, читает информацию из канала и записывает ее в канал, после чего разрывает связь:

```
BOOL CallNamedPipe(LPTSTR lpszPipeName,        // указатель строки с именем объекта-канала
    LPVOID lpvWriteBuf,                        // буфер для хранения передаваемой информации
    DWORD dwWriteBuf,                          // размер буфера записи, байты
    LPVOID lpvReadBuf,                         // буфер для хранения принимаемой информации
    DWORD dwReadBuf,                           // размер буфера чтения, байты
    LPDWORD lpdwRead,                          // реально прочитанное количество байтов
                                               // (возвращаемое значение)
    DWORD dwTimeout);                          // максимальное время ожидания, миллисекунды
```

В качестве ***первого параметра*** функции CallNamedPipe() должно задаваться имя уже существующего канала. Данная функция может вызываться только клиентами. Остальные параметры, точнее, бóльшая их часть описывают буферы, необходимые для двухсторонней поддержки транзакции.

Функция CallNamedPipe() комбинирует в одной операции несколько команд, а именноWaitNamedPipe(), CreateFile(), WriteFile(), ReadFile() и CloseHandle(). Последний параметр — dwTimeout задает максимальное время ожидания для команды WaitNamedPipe().

Если буфер чтения слишком мал для записи всего принятого сообщения, функция считывает максимально возможный объем информации и возвращает значение FALSE. В этом случае функция GetLastError() возвращает значение ERROR_MORE_DATA, и поскольку канал уже закрыт, оставшиеся в нем данные исчезают.

Клиенты часто посылают серверу по каналу команды или запросы на выполнение определенных ***действий от своего имени***. Так, клиент может попросить сервер прочитать информацию из файла. Поскольку клиент и сервер — это разные процессы, часто они имеют разные атрибуты безопасности. Сервер может отказаться от выполнения команд, прав на выполнение которых у клиента недостаточно. Сервер может временно (на время выполнения запроса) присвоить себе атрибуты безопасности клиента, а затем восстановить собственные атрибуты с помощью следующих функций:

```
BOOL ImpersonateNamedPipeClient(HANDLE hNamedPipe);
BOOLRevertToSelf (void);
```

Функция ImpersonateNamedPipeClient() не работает с анонимными каналами и не позволяет серверу принимать полномочия клиента с удаленного компьютера. К тому же она временно изменяет контекст безопасности вызывающего потока.

Функция RevertToSelf() завершает процесс передачи полномочий и восстанавливает исходный контекст безопасности.

Клиент ***разрывает связь с каналом*** посредством вызова функции CloseHandle(). Сервер может поступить таким же образом, однако иногда предпочтительнее отключиться от клиента, не уничтожая канал (это позволит сохранить его для использования в будущем). С помощью функции DisconnectNamedPipe() сервер в принудительном порядке прекращает диалог с клиентом и делает его дескриптор недействительным:

```
BOOL DisconnectNamedPipe(HANDLE hNamedPipe);
```

Если клиент попытается выполнить чтение или запись с помощью своего дескриптора после отключения сервера, то возникнет ошибка и ему все равно придется вызвать функцию CloseHandle().

В результате разрыва соединения все данные, оставшиеся в канале, будут утеряны. Сервер может сохранить оставшуюся информацию, вызвав функцию FlushFileBuffers():

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

Получив дескриптор именованного канала, функция FlushFileBuffers() блокируется до тех пор, пока не освободятся буферы этого канала.

При отключении канала от клиента сам объект-канал не уничтожается. Разорвав соединение, сервер должен вызвать функцию ConnectNamedPipe() в ожидании нового подключения освободившегося канала или же вызвать функцию CloseHandle() для уничтожения данного экземпляра

объекта. Клиенты, заблокированные функцией WaitNamedPipe(), при закрытии своего дескриптора канала разблокированы не будут. Ожидая подключения к новому клиенту, сервер должен отключить свой конец канала и вызвать функцию ConnectNamedPipe().

Таким образом, как и большинство из рассмотренных до сих пор объектов, каналы остаются в памяти до тех пор, пока не будут закрыты все их дескрипторы. Любой процесс должен на своем конце канала завершаться вызовом функции CloseHandle(). Если вы забудете об этом, команда ExitProcess() автоматически закроет все оставшиеся дескрипторы.

§ 8.3. Обмен данными с использованием сокетов

8.3.1. Общие положения. Виды сетевых протоколов

Для разработки программного обеспечения для сетевых технологий необходимо знать основные концепции и терминологию, используемую при работе в этой среде. Поэтому вначале рассмотрим некоторые общие вопросы. Большинство начинающих пользователей Internet быстро привыкают к тому факту, что у них есть свой адрес электронной почты и IP-адрес. Настройка компьютера для работы в сетях TCP/IP обычно подразумевает указание IP-адресов для доменного имени сервера и сетевого шлюза, а также ввод серверных имен POP и SMTP. IP-адреса задаются в виде четырехразрядной системы обозначений, где каждый байт 4-байтового адреса представлен десятичным числом и отделен точкой от соседнего байта. Такая система обозначений принята для упрощения восприятия IP-адресов человеком. Компьютеры и другое аппаратное обеспечениеInternet всегда воспринимают 4-байтовые (32-разрядные) значения непосредственно.

Поскольку каждый компьютер в Internet имеет собственный IP-адрес (по крайней мере, пока он подключен к сети), а число таких компьютеров увеличивается неудержимо быстрыми темпами, ведутся работы по внедрению новой схемы адресации. Однако решение возникающих при этом проблем связано с определенными трудностями, поэтому в ближайшее время мы будем пользоваться в своих программах 32-разрядными IP-адресами. Рассмотрим основные ***Internet-протоколы:***

- **TCP/IP** — это набор связанных протоколов, соответствующих разным уровням модели OSI (например, на нижних уровнях расположены основные пакетные протоколы TCP и UDP, обеспечивающие транспортировку данных методом последовательной двунаправленной передачи или методом однонаправленной передачи без установления соединения);
- **протокол IP** (*Internet Protocol*) выполняет в Internet роль «службы доставки», обеспечивающей функционирование других протоколов. IP использует дейтаграммы, передаваемые без установления соединения, поэтому IP-пакеты часто называют IP-дейтаграммами;
- **протокол TCP** (*Transmission Control Protocol*) представляет собой последовательный двунаправленный протокол, предусматривающий установление соединения и использующий байтовые потоки данных. Он считается вполне надежным, поскольку при приеме каждого пакета данных выполняется проверка контрольной суммы и производится запрос на повторную передачу пакета в случае ошибки;
- **протокол UDP** (*User Datagram Protocol*) в отличие от TCP считается «ненадежным», так как осуществляет передачу пакетов данных без установления соединения и, следовательно, неспособен реализовать проверку контрольных сумм. Данный протокол сравним с радиосвязью — вы можете быть уверены в том, что сигнал передан, но не можете проверить, принят ли он.

Два описанных подхода взаимно дополняют друг друга. Передача данных без установления соединения осуществляется быстрее, поскольку правильность операций передачи и приема не проверяется. Протокол TCP требует дополнительных расходов, но обеспечивает корректную передачу данных по месту назначения и устраняет ошибки, происходящие в каких-либо звеньях аппаратного обеспечения сети (в кабелях, коммутирующих устройствах и т. д.).

Высший уровень стека TCP/IP содержит протоколы, предназначенные для более специализированных целей. Эти протоколы выполняют роль различных сетевых сервисов и используются для передачи электронной почты, файлов, документов, Web-страниц и т. д. Перечислим протоколы, которые входят в данную группу:

• **SMTP** (*Simple Mail Transfer Protocol*) — используется для отправки электронной почты на сервер;

• **POP3** (*Post Office Protocol 3*) — предназначен для чтения электронной почты с сервера;

• **FTP** (*File Transfer Protocol*) — широко применяется для передачи файлов;

• **HTTP** (*Hypertext Transfer Protocol*) — используется в качестве стандартного Web-протокола;

• **Telnet** — предназначен для удаленной регистрации и интерактивного доступа к удаленным серверам.

В то время как низкоуровневые транспортные протоколы осуществляют передачу информации с помощью довольно сложных структур и пакетов, протоколы высшего уровня обычно применяют методику, которая более привычна с точки зрения человека и удобна для программистов. Эти протоколы имеют ряд общих характеристик [12]:

– устанавливают диалог между клиентом и сервером посредством двунаправленного гнезда;

– применяют числовые коды ответов;

– их пакеты начинаются с распознаваемой командной строки, состоящей из четырех символов.

Большинство программ, предназначенных для работы с Internet, используют парадигму «гнезд» (*sockets*). В старые добрые времена, когда TCP/IP был интегрирован в операционную систему Unix, появился набор функций, получивший название Берклиевского стандарта гнезд (*Berkeley sockets standard*). Эти функции позволяют создать гнездо, прослушать соединение (на стороне сервера), связать гнездо с заданными сервером и номером порта (на стороне клиента), а также передать и принять информацию через гнездо. Таким образом, при взаимодействии клиент — сервер в сети можно рассматривать каждого участника взаимодействия как конечную точку, которая и называется *socket* (или «гнездо»).

Гнездо играет роль, аналогичную выполняемой дескриптором файла, и в некоторых версиях Unix может использоваться непосредственно функциями ввода/вывода. Имеется два типа гнезд, соответствующих протоколам, которые подразумевают или отрицают потребность в установлении соединения [1, 12]:

• TCP-гнезда — соединение между гнездом и сервером устанавливается;

• UDP-гнезда — для передачи данных необходимо только указать конечный пункт назначения (номер порта по заданному IP-адресу).

Принцип взаимодействия сервера и клиентов заключается в следующем. На сервере программа взаимодействия с клиентами подключается к адресу IP, создает гнездо и находится в режиме ожидания подключения. Программа клиент тоже создает гнездо и пытается подключиться к гнезду сервера. Сервер при обнаружении попытки подключения создает новое гнездо и использует его для взаимодействия с клиентом. Первое созданное на сервере гнездо остается в режиме ожидания подключения следующего клиента. Таким способом производится взаимодействие сервера со многими клиентами.

В начале 90-х гг. консорциумом частных лиц и компаний была предложена версия функций работы с гнездами для Windows, которая стала называться ***WinSock***. Таким образом, WinSock, или Windows socket — это интерфейс API-функций, созданный для реализации приложений в сети на основе протокола TCP/IP. Обычно эти функции описываются в библиотеке WSOCK32.DLL.

Интерфейс WinSock несколько отличается от Берклеевского стандарта гнезд, так как 16-разрядные версии Windows, для которых он первоначально создавался, в отличие от Unix не были полноценно многозадачными операционными системами. Необходимо было предусмотреть асинхронный режим работы функций WinSock для исключения возможности блокировки системы в момент ожидания ответа. В UNIX можно было просто создать отдельные потоки для приложений, работающих с гнездами, чтобы обеспечить их независимость. Современные версии Windows полноценно многозадачны, поэтому в них можно использовать обычный Берклеевский стандарт гнезд.

В результате спецификация ***WinSock*** разделяет функции интерфейса на три типа [1]:

1) расширения Windows для функций Беркли (асинхронные функции, которые не блокируют системы при нахождении в режиме ожидания, могут извещать Windows-приложения о своем завершении, отправляя специальное сообщение указанному окну, при этом приложение может вызвать функцию, продолжить другие операции, а затем проверить код завершения функции в очереди сообщений);

2) функции Беркли (синхронные функции, производящие блокировку интерфейса в момент своей работы, т. е. при работе такой функции нельзя выполнять другие функции WinSock);

3) информационные функции (получение информации о наименовании доменов, службах, протоколах Internet).

Существует две версии WinSock:

- WinSock 1.1 — поддержка только TCP/IP;
- WinSock 2.0 — поддержка дополнительного программного обеспечения.

Интерфейс Winsock 2 имеет следующие преимущества |12|:

– поддерживает различные протоколы;

– позволяет создать быстрый программный код, адаптированный к конкретной конфигурации аппаратного обеспечения;

– является единственным решением при создании серверных программ, для которых существенными характеристиками (более важными, чем, скажем, время разработки) являются быстродействие и производительность;

– обеспечивает наилучшую среду для разработки кросс-платформенных программ.

Текущая версия Winsock 2 поддерживает большинство функций, заимствованных из предыдущих версий, а также содержит несколько функций, оптимизирующих многопротокольную поддержку. Хотя интерфейс Winsock чаще всего используется с протоколами TCP/IP, он также поддерживает связь посредством гнезд по некоторым другим протоколам, в том числе IPX, SPX, Banyan VINES и Apple Talk. Далее будет рассмотрено только функционирование TCP/IP, однако имейте в виду, что приложения, работающие с гнездами, несложно модифицировать для использования других протоколов. Интерфейс Winsock содержит как синхронные, так и асинхронные функции для выполнения операций с гнездами. Кроме того, в нем имеется несколько функций преобразования данных и просмотра баз данных. Функции интерфейса Winsock2 описаны в библиотеке ws2_32.dll.

8.3.2. API-функции для работы с сокетами

Список и описание функций, предназначенных для работы с гнездами, приведен в табл. 8.1. Вместе взятые они представляют собой традиционный набор ***синхронных функций*** (***функций Беркли***).

Рассмотрим параметры основных функций. Для создания гнезда используется системный вызов *socket()* |1, 12].

```
s = socket(domain, type, protocol);
```

где ***domain*** — тип протокола (для протокола IP используется параметр AF_INET (Internet протоколы)), ***type*** — тип подключения.

Используются три возможных типа, ***type***:

- **stream socket** (SOCK_STREAM) — обеспечивает последовательный, надежный, ориентированный на установление двусторонней связи поток байтов;
- **datagram socket** (SOCK_DGRAM) — поддерживает двусторонний поток данных. Не гарантируется, что этот поток будет последовательным, надежным и что данные не будут дублироваться. Важной характеристикой данного гнезда является то, что границы записи данных предопределены;
- **raw socket** (SOCK_RAW) — обеспечивает возможность пользовательского доступа к нижележащим коммуникационным протоколам, поддерживающим сокет-абстракции. Такие гнезда обычно являются датаграм-ориентированными;
- **protocol** — необязательный параметр, при использовании AF_INET принимающий значение 0.

Функция *socket* создает конечную точку для коммуникаций и возвращает файловый дескриптор, ссылающийся на гнездо или −1 в случае ошибки. Данный дескриптор используется в дальнейшем для установления связи.

Для создания гнезда типа ***stream*** с протоколом TCP, обеспечивающим коммуникационную поддержку, вызов функции *socket* должен быть следующим:

```
s = socket (AF_INET, SOCK_STREAM, 0);
```

Таблица 8.1

Функция	Описание
accept	Регистрирует подключение к заданному гнезду
AcceptEx	Регистрирует новое подключение, возвращает локальный и удаленный адрес, а также первый блок данных, отправленный клиентом
bind	Назначает гнезду локальный адрес
closesocket	Закрывает гнездо
connect	Соединяет гнездо с заданным одноранговым узлом
GetAcceptExSockaddrs	Анализирует данные, переданные функцией AcceptEx, выделяет локальный и удаленный адреса, а также первый блок данных, полученных при соединении
ioctlsocket	Читает или устанавливает параметры режима работы гнезда
listen	Переводит заданное гнездо в режим прослушивания
recv	Читает данные из заданного гнезда
recvfrom	Читает дейтаграмму и исходный адрес
select	Определяет статус одного или нескольких гнезд
send	Посылает данные в заданное (подключенное) гнездо
sendto	Посылает данные по адресу назначения
setsockopt	Устанавливает параметры гнезда
shutdown	Запрещает передачу и / или прием данных через указанное гнездо
socket	Создает гнездо, связанное с заданным провайдером транспортных услуг

Гнездо создается без имени. Пока с гнездом не будет связано имя, удаленные процессы не имеют возможности ссылаться на него, следовательно, на данном гнезде не может быть получено никаких сообщений. Коммуникационные процессы используют для данных целей ассоциации. В Internet-домене ассоциация складывается из локального и удаленного адресов, а также из локального и удаленного портов. В большинстве доменов ассоциация должна быть уникальной.

В Internet-домене связывание гнезда и имени может быть весьма сложным, но, к счастью, обычно нет необходимости специально привязывать адрес и номер порта к гнезду, так как функции *connect* и *send* автоматически свяжут данный *socket* с подходящим адресом, если это не было сделано до их вызова.

Для связывания гнезда с адресом и номером порта используют системный вызов *bind*:

```
bind(s, sockaddr, sockaddrlen);
```

где ***s*** — дескриптор узла, возвращаемый функцией socket; ***sockaddr*** — структура, которая интерпретируется поддерживаемым протоколом (IP-адрес, номер порта и т. д.); ***sockaddrlen*** — размер параметра *sockaddr* в байтах.

Рассмотрим структуру *sockaddr* более подробно:

```
struct sockaddr
{
    u_short sa_family;
    charsa_data[14];
};
```

Первый элемент структуры обозначает протокол, второй — специфическое для протокола значение. Например, версия данной структуры для Internet (*sockaddr_in*) будет иметь следующий вид:

```
struct sockaddr_in
{
    short sin_family;          //AF_INET
```

```
    u_short sin_port;
    struct sin_addr;          //4-х байтовый IP-адрес
    charsin_zero[8];
};
```

Элемент *sin_addr* представляет собой четырех байтовый IP-адрес (например, 127.0.0.1). Для преобразования данной строки в требуемую форму используется функция *inet_addr*(), которая будет рассмотрена далее.

Для установления связи со стороны клиента используется функция *connect*:

```
error = connect(s, serveraddr, serveraddrlen);
```

Данная функция инициирует установление связи на узле, используя его дескриптор ***s*** и информацию из структуры ***serveraddr***, содержащая адрес сервера и номер порта, на который надо установить связь. Если гнездо не было связано с адресом, connect автоматически вызовет системную функцию *bind*.

Функция *connect* возвращает 0, если вызов прошел успешно. Возвращенная величина −1 указывает на то, что в процессе установления связи произошла некая ошибка. В случае успешного вызова функции процесс может работать с дескриптором узла, используя функции *read* и *write*, и может закрывать канал, используя функцию *close*.

Со стороны сервера процесс установления связи сложнее. Когда сервер желает предложить один из своих сервисов, он связывает узел с общеизвестным адресом, ассоциирующимся с данным сервисом, и пассивно слушает этот узел. Для этих целей используется системный вызов *listen*:

```
error = listen(s, qlength);
```

где ***s*** — дескриптор узла; ***qlength*** — максимальное количество запросов на установление связи, которые могут стоять в очереди, ожидая обработки сервером (это количество может быть ограничено особенностями системы).

Когда сервер получает запрос от клиента и принимает решение об установлении связи, он создает новый узел и связывает его с ассоциацией, эквивалентной «узлу в режиме ожидания». Для Internet-домена это означает тот же самый номер порта. Для этой цели используется системный вызов *accept*:

```
newsock = accept(s, clientaddr, clientaddrlen);
```

Узел, ассоциированный клиентом, и узел, который был возвращен функцией *accept*, используются для установления связи между сервером и клиентом. Когда связь установлена, с помощью различных функций может начаться процесс передачи данных. При наличии связи пользователь может посылать и получать сообщения с помощью функций *read* и *write*:

```
write(s, buf, sizeof(buf));
read(s, buf, sizeof(buf));
```

Вызовы *send* и *recv* практически идентичны *read* и *write*, за исключением того, что добавляется аргумент флагов:

```
send(s, buf, sizeof(buf), flags);
recv(s, buf, sizeof(buf), flags);
```

Могут быть указаны один или более флагов с помощью следующих ненулевых значений:

- **MSG_OOB** — посылать / получать данные, характерные для гнезд типа *stream*;
- **MSG_PEEK** — просматривать данные без чтения. В этом случае данные посылаются пользователю, но сами помечаются как «непрочитанные». Следующий *read* или *recv*, вызванный на данном узле, вернет прочитанные в прошлый раз данные;
- **MSG_DONTROUTE** — посылать данные без маршрутизации пакетов (используется только процессами, управляющими таблицами маршрутизации).

Закрытие узла производится следующим образом: если узел больше не используется, процесс может закрыть его с помощью функции close, вызвав ее с соответствующим дескриптором узла:

```
close(s);
```

Если данные были ассоциированы с узлом, обещающим доставку (тип *stream*), система будет пытаться осуществить передачу этих данных. Тем не менее если по истечении довольно длительного промежутка времени данные все еще не доставлены, то они будут отброшены. Если пользовательский процесс желает немедленно прекратить любую передачу данных, он может сделать это с помощью вызова shutdown на данном узле для его закрытия. Формат вызова следующий [12]:

```
shutdown(s, how);
```

где *how* имеет одно из следующих значений:

- 0, если пользователь больше не желает читать данные;
- 1, если данные больше не будут посылаться;
- 2, если данные не будут ни посылаться, ни получаться.

Функции преобразования данных. Другой набор функций интерфейса Winsock связан с преобразованием данных из Сетевого формата в формат базового компьютера и в обратном направлении (табл.8.2). Эти функции рекомендуется применять даже в том случае, если формат данных одинаков на обоих компьютерах, поскольку они делают программный код хорошо совместимым и легко переносимым.

Таблица 8.2

Функция	Во что преобразует
htonl	32-разрядное значение из базового формата в сетевой формат TCP/IP
htons	16-разрядное значение из базового формата в сетевой формат TCP/IP
inet_addr	Строку IP-адреса с точками в длинное целое без знака
inet_ntoa	Сетевой адрес, представленный в виде длинного целого без знака, в строку IP-адреса с точками
ntohl	32-разрядное значение из сетевого формата TCP/IP в базовый формат
ntohs	16-разрядное значение из сетевого формата TCP/IP в базовый формат

Функции просмотра баз данных. Другой, меньший набор функций интерфейса Winsock состоит из функций просмотра баз данных. Эти функции (табл.8.3) предназначены для поиска IP-адресов, доменных имен, идентификаторов сервисов и номеров портов протоколов. Функции просмотра баз данных во многом работают подобно своим Unix-аналогам. На каждом базовом компьютере содержится информация о сервисах и, как правило, небольшой список доменных имен и IP-адресов. Эти данные обычно записываются в два текстовых файла, а именно HOSTS и SERVICES, и функции работы с гнездами используют их в качестве исходной информации об адресах или номерах сервисов.

Таблица 8.3

Функция	Что возвращает
gethostbyaddr	Информацию о базовом компьютере по заданному адресу
gethostbyname	Информацию о базовом компьютере по заданному имени узла
gethostname	Стандартное базовое имя локального компьютера
getpeername	Адрес одноранговового узла, подключенного к заданному гнезду
getprotobyname	Информацию о заданном протоколе (по его имени)
getprotobynumber	Информацию о протоколе
getservbyname	Информацию о сервисе по заданному имени сервиса и протоколу
getservbyport	Информацию о сервисе по заданному имени, порта и протоколу
getsockname	Локальное имя заданного гнезда
getsockopt	Текущее значение указанного параметра гнезда

Независимо от операционной системы, под управлением которой работает компьютер (Windows 98, Windows NT или Unix), файл HOSTS содержит базовые адреса и имена компьютеров, приведенные попарно, а файл SERVICES хранит информацию о «популярных» сервисах, в том числе соответствующие номера портов, перечень используемых низкоуровневых протоколов и необязательные псевдонимы. В Windows 98 эти файлы находятся в папке \WINDOWS, а в Windows NT — в папке \WINNT\SYSTEM32\DRIVERS\ETC.

Ниже приведен фрагмент стандартного файла SERVICES, заимствованного с рабочей станции Windows. Обратите внимание, что в каждой записи указано имя сервиса, номер порта и тип протокола, а также один или несколько дополнительных (необязательных) псевдонимов [12].

```
# Copyright (c) 1993-1995 Microsoft Corp. #
# Этот файл содержит номера портов для популярных сервисов, заданных
# в соответствии со стандартом RFC 1060
#
# Формат:
#
# <имя сервиса> <номер порта>/<протокол> [псевдонимы...]   [#<комментарий>] #
echo            7/tcp
echo            7/udp
discard         9/tcp    sink null
discard         9/udp    sink null
systat         11/tcp
systat         11/tcp    users
daytime        13/ftcp
daytime        13/udp
netstat        15/tcp
qotd           17/top    quote
qotd           17/udp    quote
chargen        19/tcp    ttytst source
chargen        19/udp    ttytst source
ftp-data       20/tcp
ftp            21/tcp
telnet         23/tcp
amtp           25/tcp    mail
time           37/tcp    timserver
time           37/udp    timserver
rip            39/udp    resource
name           42/tcp    nameserver
name           42/udp    nameserver
whois          43/tcp    nicname
domain         53/tcp    nameserver    # сервер доменных имен
domain         53/udp    nameserver
nameserver     53/tcp    domain        # сервер доменных имен
nameserver     53/udp    domain
mtp            57/tcp                  # использовать не рекомендуется
bootp          67/udp                  # сервер программы загрузки
tftp           69/udp
rje            77/tcp    netrjs
finger         79/tcp
link           87/tcp    ttylink.
supdup         95/tcp
hostnames      101/tcp   hostname
iso-tsap       102/tcp
dictionary     103/tcp   webster
x400           103/tcp                 # ISO-почта
x400-snd       104/tcp
csnet-ns       105/tcp
pop            109/tcp   postoffice
pop2           109/tcp                 # почтовая служба
pop3           110/tcp   postoffice
```

Далее представлен пример типичного содержимого файла HOSTS. Этот файл обычно имеет сравнительно малый размер, поскольку бóльшая часть имен переносится на сервер DNS (сервер доменных имен):

```
# Файл HOSTS, используемый в TCP/IP
102.54.94.97      rhino.acme.com          # исходный сервер
38.25.63.10       x.acme.corn             #сервер клиента x
127.0.0.1         localhost
```

В этом файле каждый IP-адрес соответствует имени базового компьютера. Обратите внимание на стандартный адрес 127.0.0.1, соответствующий локальному базовому компьютеру (согласно общепринятой договоренности адрес 127.0.0.1 всегда указывает на локальный базовый компьютер.) В современных Windows обычно еще имеется файл LMHOSTS. Он служит для тех же целей, что и файл HOSTS, однако заданные здесь базовые имена являются NetBIOS-именами. Кроме того, в этом файле могут использоваться ключевые слова, не применяемые в файле HOSTS (например, ключевое слово #DOM позволяет задать контролер домена сети Windows NT).

Иногда поиск адреса сводится к простому просмотру соответствующей записи в файле HOSTS, но чаще всего поисковый запрос передается на DNS-сервер, который либо удовлетворяет этот запрос, либо передает его дальше, DNS-серверу первого уровня. Если искомая запись не обнаружена на сервере первого уровня, запрос передается следующему по иерархии DNS-серверу. Этот процесс продолжается до тех пор, пока искомая запись не будет обнаружена или пока не закончится процесс поиска (в последнем случае появится сообщение об ошибке).

Асинхронные версии функций интерфейса Winsock. Для программирования асинхронных гнезд в Windows рекомендуется использовать WSA-версии функций работы с гнездами, которые в качестве параметра обычно принимают значение дескриптора окна и по окончании выполнения (или при необходимости передачи информации приложению) записывают в очередь сообщений этого окна данные о статусе операции или сообщение о завершении ее выполнения. Поскольку Windows поддерживает одновременную работу нескольких потоков, можно также использовать синхронные функции, создавая для них отдельный поток и освобождая основное приложение для обработки других задач на время выполнения этих функций. Список асинхронных функций приведен в табл.8.4 [12].

Таблица 8.4

Функция	Выполняемое действие
1	*2*
WSAAccept	В зависимости от определенных условий регистрирует соединение и создает группу гнезд или подключается к существующей группе
WSAAddressToString	Преобразует все адреса, содержащиеся в структуре SOCKADDR, в представление, удобное для восприятия человеком
WSAAsyncGetHostByAddr	Получает информацию о базовом компьютере по заданному адресу
WSAAsyncGetHostByName	Получает информацию о базовом компьютере по заданному имени
WSAAsyncGetProtoByName	Получает информацию о заданном протоколе по его имени
WSAAsyncGetProtoByNumber	Получает информацию о заданном протоколе по его номеру
WSAAsyncGetServByName	Получает информацию о сервисе по заданному имени сервиса и протоколу
WSAAsyncGetServByPort	Получает информацию о сервисе по заданному номеру порта и протоколу
WSAAsyncSelect	Просит Windows присылать сообщения о событиях, происходящих в указанном гнезде
WSACancelAsyncRequest	Отменяет незавершенную асинхронную операцию с гнездом
WSACleanup	Завершает работу с библиотекой Winsock.DLL
WSACloseEvent	Закрывает дескриптор объекта-события

Продолжение табл. 8.4

1	2
WSAConnect	Подключается к одноранговому узлу, осуществляет обмен данными о соединении и устанавливает необходимое качество сервиса на основании заданной спецификации потока
WSACreateEvent	Создает новый объект-событие
WSADuplicateSocket	Возвращает структуру WSAPROTOCOL_INFO, которая может использоваться для создания дескриптора нового коллективного гнезда
WSAEnumNameSpaceProviders	Читает информацию о доступных провайдерах пространства имен
WSAEnumNetworkEvents	Читает информацию о событиях, которые произошли в заданном гнезде
WSAEnumProtocols	Считывает информацию о доступных транспортных протоколах
WSAEventSelect	Указывает объект-событие, который должен быть связан с заданными сетевыми событиями (FD_xxx)
WSAGetLastError	Возвращает код ошибки последней некорректной операции, выполненной в рамках интерфейса Winsock
WSAGetOverlappedResult	Возвращает результат асинхронной операции в заданном гнезде
WSAGetQOSByName	Инициализирует структуру QoS (*Quality of Service* — качество сервиса) на основании именованного шаблона
WSAGetServiceClassInfo	Запрашивает информацию о классе сервиса у заданного провайдера пространства имен
WSAGetServiceClassNameByClassId	Возвращает обобщенное имя сервиса для заданного типа сервисов
WSAHtonl	Преобразует длинное целое без знака из базового формата в сетевой
WSAHtons	Преобразует короткое целое без знака из базового формата в сетевой
WSAInstallServiceClass	Регистрирует схему класса сервисов в пространстве имен
WSAIoctl	Задает режим работы указанного гнезда, а также конфигурирует гнезда для осуществления безопасной связи с помощью протокола SSL или PCT
WSAJoinLeaf	Подключается к концевому узлу в сеансе групповой связи, осуществляет обмен данными о соединении и устанавливает необходимое качество сервиса на основе заданной спецификации потока
WSALookupServiceBegin	Возвращает дескриптор, который может использоваться в последующих клиентских запросах для поиска нужной информации о сервисе
WSALookupServiceEnd	Освобождает дескриптор, созданный функцией
WSALookupServiceNext	Возвращает следующий набор информации о сервисе, используя заданный дескриптор
WSANtohl	Преобразует длинное целое без знака из сетевого формата в базовый
WSANtohs	Преобразует короткое целое без знака из сетевого формата в базовый
WSAProviderConfigChange	Информирует приложение об изменении конфигурации провайдера
WSARecv	Считывает данные из заданного гнезда
WSARecvDisconnect	Прекращает прием данных из заданного гнезда, возвращая данные о разрыве соединения, если таковые поступают
WSARecvEx	Аналогична функции WSARecv
WSARecvFrom	Считывает дейтаграмму и исходный адрес
WSARemoveServiceClass	Отменяет регистрацию схемы класса сервисов

Окончание табл. 8.4

1	2
WSAResetEvent	Сбрасывает заданный объект-событие в несигнальное состояние
WSASend	Посылает данные в подключенное гнездо, разрешая выполнение асинхронной операции ввода / вывода
WSASendDisconnect	Инициирует прекращение связи с заданным гнездом и передает данные о разрыве соединения
WSASendTo	Отправляет данные по указанному адресу, в случае необходимости используя асинхронные операции ввода / вывода
WSASetEvent	Устанавливает заданный объект-событие в сигнальное состояние
WSASetLastError	Устанавливает код ошибки, который возвращается при последующих вызовах функции WSAGetLastError
WSASetService	Регистрирует или отменяет регистрацию экземпляра сервиса в пределах одного или нескольких пространств имен
WSASocket	Создает гнездо, связанное с указанным провайдером транспортных услуг, создавая при необходимости группу гнезд или подключаясь к уже существующей группе
WSAStartup	Инициирует применение процессом библиотеки Winsock.DLL
WSAStringToAddress	Преобразует строку адреса в структуру адреса гнезда
WSAWaitForMultipleEvents	Завершается, если один или все указанные объекты-события переходят в сигнальное состояние или если завершается период ожидания

§ 8.4. Обмен данными по технологии динамического обмена данными

8.4.1. Общие положения

Технология ***динамического обмена данными*** (*Dynamic Data Exchange* — DDE) обеспечивает прямой обмен информацией между приложениями, установившими диалог, не делая эти данные доступными для каких-либо других программ. Кроме того, в отличие от передачи данных через буфер обмена, где совместно используется один служебный ресурс, между двумя или более DDE-приложениями может одновременно устанавливаться несколько независимых друг от друга DDE-диалогов [6, 12].

Технология DDE непригодна для распределенной обработки информации или интенсивного совместного использования данных в тех ситуациях, когда первоочередным, определяющим фактором является быстродействие. Основная черта DDE заключается в том, что с ее помощью могут обмениваться данными независимые приложения, не предназначенные специально для этой цели. Приложения должны только распознавать предлагаемые форматы данных и договориться друг с другом о способе передачи, совместимых темах, элементах и т. д.

DDE представляет собой основанную на сообщениях систему обмена данными между приложениями, которая аналогична системе внутренних сообщений Windows. DDE-сообщения также управляются операционной системой.

С помощью DDE независимые программы могут обмениваться сообщениями и данными. DDE-сообщения могут передаваться широковещательно, т. е. посылаться всем приложениям, которые находятся в режиме прослушивания, или направляться непосредственно заданным приложениям.

Трафик DDE-сообщений представляет собой диалог между двумя (или несколькими) приложениями, предусматривающий наличие протоколов общения и определенной избыточности информации, благодаря которой обеспечивается высокая степень надежности передачи данных.

DDE-приложения могут одновременно поддерживать диалог с несколькими программами или несколько диалогов с одной программой. В этом смысле DDE-диалог можно сопоставить с сеан-

сом одновременной игры в шахматы по переписке с одним или несколькими противниками, когда даже посредственный игрок может поддерживать сразу несколько «диалогов». Проведение диалога невозможно без наличия у DDE-приложения трех основных идентификаторов |12|:

1. Имя приложения или сервиса. Имя DDE-приложения подразумевает широкий диапазон информации, которая может предоставляться сервером. Некоторые серверы могут выдавать только один тип информации, в то время как другие могут предоставлять данные нескольких типов, используя, таким образом, несколько имен приложений. Во избежание недоразумений в отношении операций с библиотекой DDEML (DDE Management Library) вместо термина ***имя приложения*** применяется термин ***имя сервиса***.

2. Тема. Каждый DDE-диалог должен иметь, по крайней мере, одну ***тему*** (*topic*), хотя один и тот же диалог может «переключаться» между несколькими темами либо несколько диалогов могут использовать несколько тем. Если провести аналогию с разговором между людьми, то тема эквивалентна предмету беседы. При DDE-диалоге тема может указываться и распознаваться обеими сторонами. Имя темы обычно совпадает с именем файла.

3. Элемент. Имя ***элемента*** представляет собой идентификатор внутри темы, указывающий конкретный элемент данных, который передается в результате текущей операции обмена. Если участник диалога не распознает тему, то обмен данными прерывается и диалог нарушается, хотя и не прекращается полностью. Имя элемента может указывать на страницу документа, строку, изображение, ячейку таблицы или любой другой фрагмент данных, который обычно передается от одной программы к другой.

Пусть, например, электронная таблица поддерживает два типа сервисов: *spreadsheet* (таблица) и *chart* (диаграмма). Каждый из этих сервисов в качестве темы может использовать имена определенных файлов данных электронной таблицы. Элементами сервиса *spreadsheet* могут быть ячейки электронной таблицы, а элементами сервиса *chart*—форматы представления данных, такие как *pie* (круговая диаграмма) или *bar* (гистограмма).

Количество тем и элементов ограничивается лишь словарем, который совместно используется всеми участниками сеанса. Элементы могут представлять собой как простейшие типы данных, например целые числа, так и растровые изображения, метафайлы, массивы или структурированные записи. В роли информационного элемента может также выступать любой пользовательский тип данных, распознаваемый всеми участниками диалога.

Возможно ограничение числа сервисов на сервере до одного. В данном случае если имена сервиса и сервера одинаковы, то клиент, который знает одно имя, автоматически узнает и второе. Следовательно, он может инициировать диалог, используя заданное имя EXE-файла.

Если сервер и сервис имеют разные имена или у сервера в наличии имеется несколько сервисов, приложениям-клиентам (а также их разработчикам) для инициирования диалога нужна дополнительная информация. Альтернативным вариантом является создание специального сервиса с каким-либо легко распознаваемым именем (например, *system*), который просто сообщает приложениям-клиентам имена сервисов, доступных на этом сервере.

Кроме того, если приложение-клиент знает и имя сервера (программы), и имя сервиса, оно может запустить сервер, например с помощью функции CreateProcess, и восстановить связи из предыдущего сеанса.

DDE-диалоги состоят из ***транзакций*** трех типов: ***подключения***, ***принудительных***, ***командных*** [6, 12].

В транзакциях подключения, являющихся наиболее распространенными, клиент запрашивает элемент данных, который находится на сервере. Различают три вида подключения [12]:

– ***с необязательным ответом.*** Такой вид диалога инициируется приложением-клиентом, посылающим широковещательный запрос WM_DDE_INITIATE, в котором указывается вызываемое приложение и тип запрашиваемых данных (как первый, так и второй параметры могут быть пустыми, если приемлемыми считаются любые сервер и тема). В зависимости от обстоятельств на этот запрос может откликнуться один или несколько серверов, идентифицируя себя для установления диалога. Если сервер не соответствует запрашиваемому имени или не распознает тему, он просто не откликается, поскольку приложение-клиент ожидает поступления только подтверждающих ответов. После получения данных диалог немедленно завершается;

– ***с полуобязательным ответом.*** Такой вид диалога подразумевает, что клиент и сервер «знают» друг друга и сервер имеет новую информацию, которая, как он полагает, заинтересует клиента. Обычно клиент посылает серверу сообщение WM_DDE_ADVISE, делая запрос о необходимости обновить тему (и элемент). Сервер подтверждает получение запроса, но посылает данные только при наличии новой информации;

– ***с обязательным ответом.*** Такой вид диалога отличается от предыдущего тем, что клиент ожидает от сервера подтверждения и немедленного ответа. Если информация в данный момент отсутствует, сервер просто ответит клиенту, что данные недоступны, и будет ожидать следующего запроса. Сервер не будет передавать информацию до тех пор, пока не поступит запрос от клиента.

В процессе одного и того же диалога могут чередоваться подключения всех трех типов. Кроме того, их границы бывают настолько расплывчатыми, что порой трудно определить, где заканчивается один тип подключения и начинается другой.

Принудительные транзакции используются для передачи от клиента к серверу элемента данных, который специально не запрашивался.

Командные транзакции позволяют клиенту посылать серверу команды или последовательности команд, заставляющих его выполнять определенные действия.

Обмен данными начинается в тот момент, когда приложение-клиент инициирует диалог и получает ответ от сервера. Установив соединение, клиент и сервер обмениваются данными вплоть до того момента, пока диалог не будет прерван по инициативе одной из сторон. Обмен данными осуществляется одним или сразу несколькими из перечисленных далее способов [6]:

– клиент запрашивает данные, а сервер отвечает на запрос;

– клиент просит проинформировать его, произошли ли изменения определенных данных;

– клиент просит автоматически передать определенные элементы данных в случае их обновления;

– клиент посылает команду, а сервер ее выполняет;

– клиент посылает серверу незапрашиваемый элемент данных.

Важно помнить о том, что приложение-клиент может также выступать в роли сервера и наоборот. Любое приложение может одновременно быть и сервером, и клиентом. Различия между клиентом и сервером являются весьма искусственными и не вполне определенными, поскольку любой из них может как запрашивать данные, так и передавать их.

8.4.2. API-функции библиотеки DDEML

В своей основе DDE-диалог — это довольно сложный и запутанный процесс, включающий многочисленные ответы и подтверждения с применением разных протоколов. Поэтому для облегчения жизни программистов создана специальная библиотека — DDEML (DDE *Management Library*).

DDEML входит в состав Windows и содержит высокоуровневые API-функции, позволяющие упростить процесс DDE-диалога. Библиотека поддерживает запись о каждом диалоге с помощью структуры CONVINFO, которая содержит информацию об участниках, установивших диалог, запрашиваемых темах и элементах, используемых форматах и типах данных, а также о статусе диалога, ошибках и т. д. Следует отметить, что большинство элементов этой структуры можно просто проигнорировать, возложив все обязанности по ведению записей на DDEML. Рассмотрим основные функции данной библиотеки.

Функция DdeInitialize предназначена для задания функции обратного вызова, управляющей графиком DDE-сообщений. Когда приложение вызывает функцию DdeInitialize, DDEML уведомляет об этом другие программы, отправляя сообщение XTYP_REGISTER их функциям обратного вызова. Эти сообщения используются многими клиентами для поддержки списка доступных сервисов.

Необходимость указывать идентификатор экземпляра приложения при вызове функции DdeInitialize представляет собой дополнительное ограничение для программистов, работающих в среде Windows, поскольку этот идентификатор является локальным для потока и не наследуется дочерними процессами. Так как идентификатор экземпляра необходимо задавать в качестве параметра большинства DDEML-операций, соответствующие операции должны находиться в том же

самом потоке, в котором был осуществлен вызов функции DdeInitialize. Поток, инициализировавший DDE-сеанс, не должен прекращаться вплоть до его завершения. В противном случае вы не сможете вызвать функцию DdeUninitialize, осуществляющую корректное завершение сеанса.

После инициализации сервер должен зарегистрировать имена предлагаемых сервисов. Для этой цели служит функция DdeNameService, которая вызывается с указанием дескриптора строки, содержащей имя сервиса, и идентификатора экземпляра данного сервиса. По идентификатору экземпляра DDEML узнает о том, какая процедура обратного вызова и какой поток поддерживают данный сервис:

```
HDDEDATA DdeNameService (DWORD dwInstID,      // идентификатор экземпляра
                         HSZ hszl,            // строка с именем сервиса
                         HSZ hszRes,          // зарезервирован
                         UINT uFlags);        // флаги
```

Значение параметра dwInstID возвращается функцией DdeInitialize. Параметр uFlags может содержать следующие флаги:

- **DNS_REGISTER** — регистрация имени сервиса;
- **DNS_UNREGISTER** — отмена регистрации имени сервиса; если аргумент *hszl* имеет значение NULL, происходит отмена регистрации всех сервисов данного сервера;
- **DNS_FILTERON** — предотвращает получение сервером сообщений XTYP_CONNECT для незарегистрированных сервисов;
- **DNS_FILTEROFF** — позволяет серверу получать сообщения XTYP_CONNECT при вызове любой DDE-программой функции DdeConnect.

Возвращаемый функцией DdeNameService результат в действительности представляет собой логическое значение, причем он соответствует ошибке, а ненулевое значение — успешному выполнению функции. Но в настоящее время он имеет тип HDDEDATA, что позволяет в будущем расширить диапазон значений этого индикатора.

Если приложение поддерживает более одного сервиса, каждый из них должен регистрироваться отдельно. Использование для каждого сервиса отдельного потока также имеет ряд преимуществ (в частности, в этом случае обеспечивается неизменность дескриптора экземпляра потока на протяжении всего времени существования данного сервиса).

«Сердцем» любого DDEML-приложения является функция DdeCallback. Windows передает ей восемь параметров вызова. Функция DdeCallback аналогична функции WndProc в том, что возвращаемый ею ответ определяется приложением. Список поступающих в функцию DdeCallback сообщений с указанием типов ожидаемых ответов приведен в табл. 8.5.

Таблица 8.5

Сообщение	Ответ
XTYP_ADVSTART, XTYP_CONNECT	Логическое значение (TRUE, FALSE)
XTYP_ADVREQ, XTYP_REQUEST, XTYP_WILDCONNECT	Дескриптор блока данных (или NULL)
XTYP_ADVDATA, XTYP_EXECUTE, XTYP_POKE	Флаг: DDE_FACK, DDE_FBUSY или DDE_FNOTPROCESSED
XTYP_ADVSTOP, XTYP_DISCONNECT, XTYP_CONNECT_CONFIRM, XTYP_ERROR, XTYP_REGISTER, XTYP_UNREGISTER, XTYP_XACT_COMPLETE	Нет ответа, только уведомление

Функция DdeClientTransaction дает команду DDEML отправить сообщение серверу. Тип сообщения зависит от характера действий, запрашиваемых клиентом. В табл. 8.6 перечислены некоторые типы сообщений и соответствующие им транзакции, выполняемые сервером.

Таблица 8.6

Сообщение	Транзакция
XTYP_REQUEST	Подключение с необязательным ответом, передача одного элемента данных
XTYP_ADVSTART	Подключение с полуобязательным или обязательным ответом, передача данных и обновлений
XTYP_POKE	Запись (прием) данных от клиента
XTYP_EXECUTE	Выполнение команды или действия

8.4.3. Механизмы обработки транзакций

Обработка транзакций с необязательным ответом. Когда сервер получает сообщение XTYP_REQUEST, он читает имена темы и элемента данных из двух строковых параметров и проверяет запрашиваемый формат данных. Если сервер распознает элемент и поддерживает данный формат, он создает объект данных, в который записывает текущее значение элемента, и возвращает дескриптор этого объекта. Если сервер не поддерживает данный формат или не распознает элемент, возвращается значение NULL.Сервер, который не поддерживает транзакции данного типа, во избежание получения нежелательных сообщений XTYP_REQUEST должен установить в функции DdeInitialize флаг CBF_FAIL_REQUESTS. Значение, переданное сервером, возвращается клиенту в виде результата функции DdeClientTransaction. Типичная схема такого взаимодействия представлена на рис. 8.1 [12]. Условно его можно разбить на три этапа:

1) клиент посылает серверу запрос через DDEML;
2) сервер расшифровывает запрос и формирует пакет запрашиваемых данных;
3) клиент получает ответ и расшифровывает его.

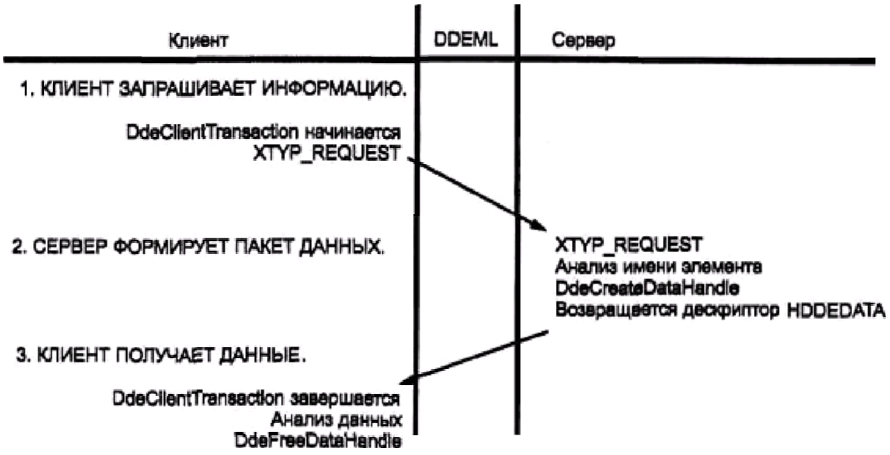

Рис.8.1. Последовательность событий, происходящих после запроса клиентом элемента данных

Когда клиент делает запрос о возможности получить извещение об обновлении данных, функция обратного вызова сервера принимает сообщение XTYP_ADVSTART. Если сервер распознает имена темы и элемента данных, а также поддерживает запрашиваемый формат, он возвращает значение TRUE, подтверждая факт установления подключения. Если одно из указанных условий не выполняется, сервер возвращает значение FALSE, предотвращающее установление подключения.

Сообщение XTYP_ADVSTART не указывает серверу, с каким ответом — обязательным или полуобязательным — будет это подключение.

Транзакции с обязательным ответом. При установлении подключения с обязательным ответом сервер не возвращает данные немедленно. Он посылает их только при последующих изменениях запрашиваемого элемента. В ответ на сообщение XTYP_ADVSTART сервер часто устанав-

ливает флаг, «напоминающий» ему о том, что поступил запрос на обновление данных. При любых изменениях сервер проверяет состояние этого флага и, если он установлен, вызывает функцию DdePostAdvise, которая уведомляет DDEML о поступлении новых данных, интересующих клиента.

Сервер не должен помнить о том, какой из клиентов послал данный запрос — DDEML решает эту задачу собственными средствами. По заданным именам темы и элемента, которые передаются сервером для идентификации доступных данных, DDEML определяет, какой из клиентов «находится на связи» и каким —обязательным или полуобязательным — является подключение.

В случае обязательного подключения DDEML немедленно отправляет серверу сообщение XTYP_ADVREQ, получает от него данные и передает их клиенту в виде сообщения XTYP_ADVATA. Этапы этого процесса можно представить следующим образом (рис. 8.2):

– инициатором сеанса выступает клиент, а сервер откликается на его запрос, посылая ответ TRUE;

– сервер информирует DDEML о наличии новых данных, а DDEML определяет, что данное подключение является обязательным, и запрашивает данные, а затем передает их приложению-клиенту. Этот этап повторяется каждый раз при получении сервером новых данных;

– клиент разрывает подключение, посылая сообщение XTYP_ADVSTOP, в ответ на которое сервер возвращает NULL.

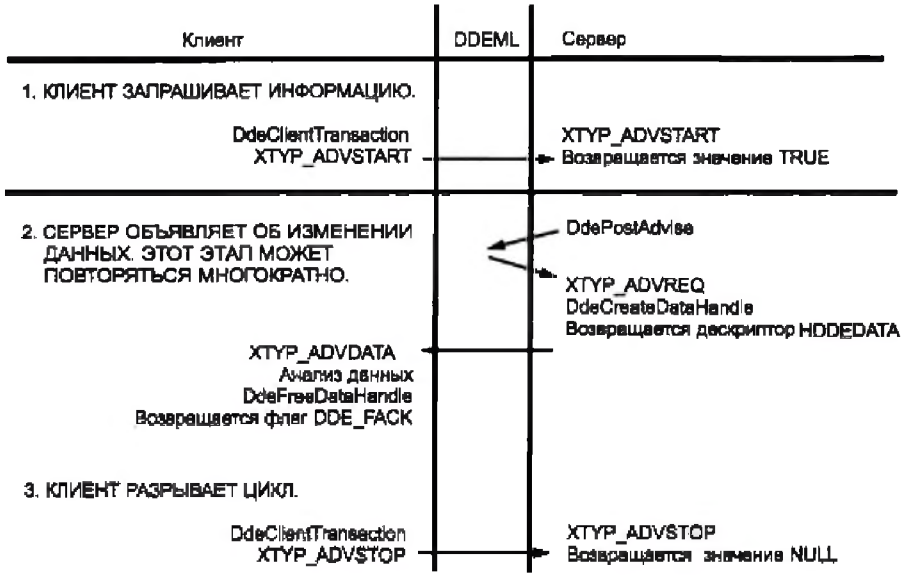

Рис.8.2. Цикл транзакции с обязательным ответом

Транзакции с полуобязательным ответом. В случае подключения с полуобязательным ответом DDEML также получает от сервера уведомление об изменении данных, но не отправляет ему сообщения с запросом данных. Вместо этого DDEML посылает клиенту сообщение XTYP_ADVDATA с пустым дескриптором блока данных.

Сервер, который не поддерживает транзакций с обязательным и полуобязательным ответом, должен установить флаг CBF_FAIL_ADVISES в функции DdeInitialize. Таким образом он предотвратит получение нежелательных сообщений XTYP_ADVSTART и XTYP_ADVSTOP.

Для получения нового значения элемента данных клиент должен выполнить транзакцию с необязательным ответом. На рис. 8.3 изображены этапы этого процесса:

– сервер объявляет о поступлении новых данных;

– DDEML информирует клиента, а клиент игнорирует это уведомление (уведомления будут посылаться клиенту всякий раз при изменении данных на сервере, однако клиент может проигнорировать их);

– сервер вновь объявляет об изменениях в запрашиваемом элементе данных;

– клиент откликается, запрашивая данные (сообщение XTYP_REQUEST);

– DDEML передает запрос серверу;

– сервер посылает запрашиваемые данные клиенту.

Рис.8.3. Цикл транзакции с полуобязательным ответом

Обработка принудительных и командных транзакций. Некоторые серверы предпочитают получать данные от клиента иным способом [12]. Клиент, который хочет передать данные серверу, должен выполнить принудительную транзакцию — послать сообщение XTYP_POKE. В ответ на это сообщение сервер проверяет тему, элемент данных и при желании читает объект данных.

Некоторые серверы также получают команды от своих клиентов, принимая их в виде дескрипторов данных. Сервер блокирует объект, анализирует содержащуюся в нем командную строку, а затем выполняет определенные действия. После анализа командной строки сервер должен освободить дескриптор данных, но при необходимости он может скопировать эту строку для дальнейшего использования. Поскольку клиенты обычно ожидают немедленного выполнения команд и требуют подтверждения, сервер по возможности должен выполнить команду из функции обратного вызова до того, как будет возвращен результат.

В ответ на принудительную или командную транзакцию сервер возвращает одно из трех значений:

- **DDE_PACK** — данные получены и подтверждены;
- **DDE_FBUSY** — сервер занят и не может обработать данные или инструкции; повторите попытку позже;
- **DDE_FNOTPROCESSED** — данные или инструкции отклонены.

Флаги, установленные в функции DdeInitialize, позволяют отфильтровывать нежелательные сообщения о принудительных (флаг CBF_FAIL_POKES) или командных (флаг CBF_FAIL_EXECUTES) транзакциях для серверов, не поддерживающих эти транзакции [12].

8.4.4. Завершение DDE-диалога

При необходимости ***прекратить DDE-диалог*** клиент или сервер вызывают либо функцию DdeDisconnect (завершить указанный диалог), либо функцию DdeDisconnectList (завершить группу диалогов). В обоих случаях конкретному приложению или всем приложениям передается сообщение XTYP_DISCONNECT. Синтаксис вызова функции DdeDisconpect имеет следующий вид:

```
DdeDisconnect (hConv);
```

где параметр hConv представляет собой дескриптор диалога, возвращаемый функцией DdeConnect.

Функция DdeDisconnectList вызывается так:

```
DdeDisconnectList (hConvList);
```

где параметр hConvList представляет собой дескриптор, возвращаемый функцией DdeConnectList.

Когда DDEML получает сообщение XTYP_DISCONNECT, все активные в данный момент транзакции текущего диалога (или диалогов) прекращаются. Согласно договоренности диалоги могут завершаться только приложениями-клиентами. Однако при особых обстоятельствах подключения могут разрываться и приложениями-серверами (например, в случае их закрытия).

Как клиенты, так и серверы должны быть готовы в любой момент получить сообщение о разрыве подключения. При этом они должны закрыть соответствующие структуры данных. В случае закрытия приложения или если оно больше не собирается устанавливать подключения к другим приложениям программа должна отменить инициализацию всех DDEML-функций обратного вызова.

При вызове функции DdeUninitialize информация о сервисе удаляется из DDEML-таблиц, все активные в данный момент диалоги прекращаются и происходит рассылка сообщения XTYP_UNREGISTER всем DDEML-функциям обратного вызова. Этапы процесса завершения диалога и отключения от DDEML показаны на рис. 8.4.

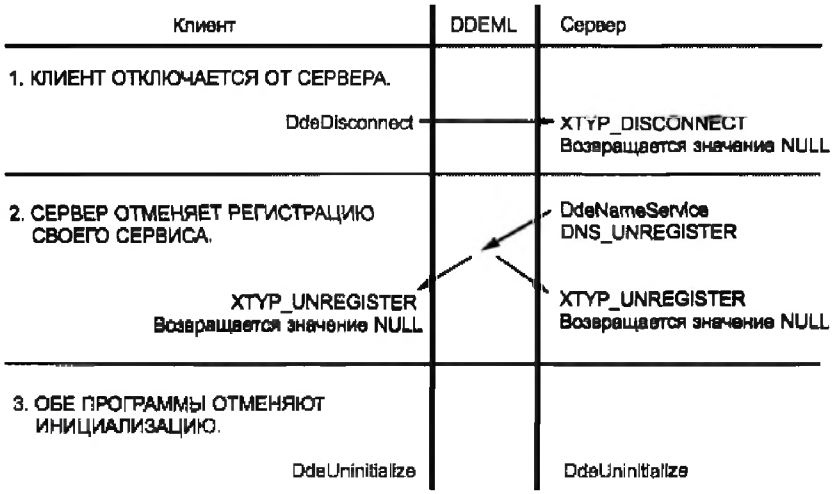

Рис.8.4. Процесс закрытия клиента и сервера

При возникновении ***ошибки*** DDE-функция возвращает значение NULL или FALSE. Более подробная информация об ошибке может быть получена с помощью функции DdeGetLastError:

```
UINT DdeGetLastError (DWORD dwInstID);
```

Поскольку функция DdeGetLastError требует задания идентификатора экземпляра приложения, она не может применяться для получения информации об ошибке в случае невыполнения функции DdeInitialize. Поэтому она возвращает собственные, расширенные коды ошибок:

- **DMLERR_NO_ERROR** — успешная инициализация;
- **DMLERR_DLL_USAGE** — программа зарегистрирована для DDE-мониторинга и не может использовать DDEML для выполнения транзакций;
- **DMLERR_INVALID_PARAMETER** — параметр содержит недопустимую информацию;
- **DMLERR_SYS_ERROR** — произошла внутренняя ошибка DDEML.

8.4.5. Синхронные и асинхронные транзакции

DDEML-клиенты могут выбирать между выполнением ***синхронных*** и ***асинхронных*** транзакций [12]. Отличие между синхронными и асинхронными операциями проявляется только на уровне программы-клиента. Относительно программы-сервера транзакции обоих типов выглядят одинаково. Преимущество асинхронных транзакций заключается в том, что они быстрее выполняются и проще программируются. Асинхронные транзакции применяются сильно загруженными

программами, которые должны обрабатывать значительный объем данных при взаимодействии с DDEML-сервером, или программами, которые вынуждены регулярно взаимодействовать с медленным сервером и хотят избежать нежелательных периодов ожидания ответа.

В случае ***синхронных*** операций приложение-клиент ожидает ответа от сервера, инициализировав транзакцию либо путем запроса информации, либо путем передачи команды для выполнения определенных операций. Если сервер заставляет клиента ожидать дольше установленного времени, функция DdeClientTransaction отменяет транзакцию и завершает свою работу.

При синхронных транзакциях необходимо принимать меры для освобождения объектов данных. DDEML передает дескриптор данных в виде результирующего значения функции DdeClientTransaction и не имеет возможности узнать о том, когда данные будут освобождены.Клиент становится владельцем объектов данных, полученных при синхронной транзакции от функции DdeClientTransaction, и в конце концов должен вызвать функцию DdeFreeDataHandle, чтобы освободить их.

В случае ***асинхронных*** операций при вызове функции DdeClientTransaction вместо длительности паузы следует задать значение TIMEOUT_ASYNC. В результате функция DdeClientTransaction немедленно возвратит значение TRUE, а также идентификатор транзакции в поле dwResult.

Пока приложение-клиент продолжает выполнять другие операции, DDEML осуществляет транзакцию в фоновом режиме. Затем по завершении транзакции DDEML посылает сообщение XTYP_XACT_COMPLETE функции обратного вызова приложения-клиента. В этом сообщении в параметре dwData содержится идентификатор транзакции, что позволяет клиенту определить, какой из запросов выполнен.

DDEML обеспечивает также альтернативный механизм идентификации транзакций по их сообщениям о завершении. Клиент может зарегистрировать свое значение типа DWORD, связанное с каждой асинхронной транзакцией. Функция DdeSetUserHandle принимает дескриптор диалога, идентификатор асинхронной транзакции и пользовательское значение типа DWORD, сохраняя эти параметры внутри собственной структуры данных. При получении сообщения XTYP_XACT_COMPLETE клиент читает локальный идентификатор с помощью функции DdeQueryConvInfo.

Ожидая завершения асинхронной транзакции, функция DdeClientTransaction входит в цикл и производит опрос сообщений каждого из окон, что позволяет клиенту продолжать реагировать на команды пользователя, ожидая ответа на транзакцию. Однако в течение этого периода клиент не может выполнить другую DDEML-функцию. Кроме того, попытка вызвать функцию DdeClientTransaction завершается неудачно, если у того же самого клиента в данный момент выполняется другая, синхронная транзакция.

Чтобы отменить асинхронную операцию, не дожидаясь ее завершения, следует вызвать функцию DdeAbandonTransaction. При этом DDEML освобождает все ресурсы, связанные с транзакцией, и отменяет результаты, возвращенные сервером (аналогичный процесс происходит по истечении времени, отведенного на выполнение синхронной транзакции).

При асинхронной транзакции DDEML предоставляет клиентской функции обратного вызова дескриптор объекта данных. Когда функция обратного вызова возвращает результат, DDEML резонно предполагает, что данные больше не нужны, и автоматически освобождает объект данных. Если клиент хочет сохранить данные, он должен их скопировать с помощью функции DdeGetData.

Как объяснялось ранее, обычная последовательность инициации диалога предполагает, что клиент знает заранее, какие ему нужны сервисы и/или темы. Чтобы предоставить возможность клиентам просмотреть все доступные темы, DDE-серверы иногда содержат для каждого сервиса тему, которая называется *system*. С помощью этой темы обеспечивается поддержка набора стандартных элементов. Кроме того, тема system позволяет клиенту получить информацию о конкретном сервисе [12]. Строки, идентифицирующие стандартные системные элементы, определяются в виде констант в файле DDEML.H. Три из них должны поддерживаться всеми приложениями:

• **SZDDESYS_ITEM_SYSITEMS** — список элементов, поддерживаемых сервером в теме *system* (имя элемента задается строкой "Sys Items");

• **SZDDESYS_ITEM_FORMATS** — список строк, разделенных символами табуляции и указывающих форматы буфера обмена, которые поддерживаются данным сервером (имя элемента задается строкой «Formats»);

• **SZDDESYS_ITEM_TOPICS** — список тем, поддерживаемых сервером (имя элемента задается строкой «Topics»).

Кроме этих трех элементов темы system, DDEML-сервер поддерживает в каждой теме еще один стандартный элемент:

• **SZDDE_ITEM_ITEMLIST** — список элементов темы, отличной от *system* (имя элемента задается строкой "Topic Item List").

В ответ на запрос этих элементов сервер должен соединить все имена темы, элемента или формата в одну длинную строку, используя в качестве разделителей символы табуляции. Из этой строки сервер создает объект данных и возвращает библиотеке DDEML дескриптор этого объекта в качестве результирующего значения в ответ на сообщение XTYP_REQUEST. Затем клиент выделяет данные из этого объекта, разбивает список на составные части и отображает их.

§ 8.5. Обмен данными по технологии связывания и внедрения объектов

8.5.1. Общие положения

Технология OLE (*Object Linking and Embedding* — связывание и внедрение объектов) представляет собой совершенно иной способ совместного использования данных различными приложениями по сравнению с буфером обмена и технологией DDE. В широком смысле технология OLE является формой межзадачного взаимодействия |12|. В частности, она позволяет одному приложению подключать или встраивать в себя информацию, созданную другим приложением. При этом создается так называемый ***составной документ***, который еще называют ***OLE-документом***.

Написанное вами OLE-приложение будет работать правильно даже в том случае, если оно взаимодействует с сервером, предоставляющим данные в формате, который не предусмотрен стандартами Microsoft. Технология OLE освобождает операционную систему от необходимости отслеживать все форматы данных — если сервер способен обрабатывать эти данные, любой клиент сможет получить их.

OLE представляет собой набор протоколов и функций, предложенных корпорацией Aldus в 1988 г. для упрощения создания и поддержки составных документов. ***Составным документом*** называется файл, который принадлежит одному приложению (например, текстовому редактору), но содержит данные, созданные другими приложениями (например, графическим редактором). Блоки «чужих» данных в составном документе называются ***объектами***. Приложение, которое получает объекты данных и формирует из них составные документы, называется ***OLE-клиентом***, а приложение, экспортирующее объекты для использования их другими приложениями — ***OLE-сервером***. Является приложение клиентом или сервером, зависит от его роли в конкретной схеме взаимодействия. Одно приложение может одновременно выступать в роли как клиента, так и сервера |12|.

Интегрировав технологию OLE в операционную систему Windows, Microsoft сделала большой шаг в сторону ориентации работы пользователя на документы, а не на приложения. Традиционно пользователь вызывал отдельное приложение для каждого нового документа. Работа с данными иного формата обычно предполагала выход из одного приложения и запуск другого. В среде, где работа ведется с приложениями, документ имел смысл только в той программе, в которой он был создан.

С другой стороны, в среде разработки документов несколько приложений могут «скооперироваться»и создать единый документ. Ни одно из приложений не распознает форматы всех объектов, содержащихся в документе, но при переходе от одного объекта к другому система автоматически вызывает соответствующие приложения. Редактирование различных объектов осуществляется отдельно в их «родных» приложениях, а базовый документ автоматически получает обновленные данные от всех «программ-пайщиков». Пользователь получает возможность свободно комбинировать

звук, видео, изображения, числа и тексты в едином интегрированном документе. Можно демонстрировать картинки в текстовом редакторе или прилагать видеоклипы к записям в базе данных.

Составные документы существовали в операционной системе Windows еще до появления технологии OLE, но возможности работы с ними были очень ограниченны. Пользователь мог создавать составной документ, копируя данные из буфера обмена и вставляя их в другом приложении. При этом объект данных перемещался из программы-сервера в программу-клиент. Но если сервер впоследствии редактировал объект, то операции копирования и вставки необходимо было повторять для ***всех документов***, в которые была внедрена копия этого объекта [12].

Для формирования среды разработки документов операционная система должна иметь возможность координировать работу приложений. В частности, система должна знать, какая из программ может работать с определенным форматом данных. По мере того как пользователь будет перемещаться от объекта к объекту в составном документе, система должна распознавать эти объекты и поддерживать связи с соответствующими приложениями. В Windows эти сложные операции реализованы с помощью библиотек OLE-функций. Они помогают создавать программы, осуществляющие обработку объектов практически любого типа путем невидимого взаимодействия с приложениями, создавшими эти объекты.

Объект представляет собой набор данных из одного приложения, которые трактуются как единое целое. OLE-приложения создают составные документы, комбинируя несколько объектов в одном файле. Пользователь видит одновременно все объекты составного документа в одном окне. Для программы-клиента каждый объект представляет собой «черный ящик», заполненный непонятными данными. Но она и не должна понимать эти данные, поскольку для манипуляций с объектами программа-клиент вызывает OLE-функции.

При импортировании объекта программа-клиент должна выбрать один из двух вариантов: ***связывание*** или ***внедрение*** [12]. Если документ содержит все данные объекта, то такой объект называется ***внедренным***. Если документ содержит только ссылку, указывающую на данные в другом документе, то такой объект называется ***связанным***. Оба метода дают на экране один и тот же результат, но только в связанных объектах автоматически отображаются все обновления.

Связывание объектов имеет еще одно преимущество: файл результирующего документа получается гораздо меньшим по объему. Преимущество же внедренных объектов заключается в их независимости: документ, содержащий только внедренные объекты, может легко переноситься из одной системы в другую. Поскольку внедренные объекты содержат внутри себя все необходимые данные, новая система не обязана содержать OLE-сервер. При желании пользователь может преобразовать связанные объекты во внедренные.

Как уже говорилось, OLE-приложения могут относиться к одному из двух типов: клиенты и серверы. Если вы внедряете изображение, созданное программой Paint, в документ текстового редактора WordPad, то Paint выступает в роли сервера, а WordPad в роли клиента. Редактор WordPad не обязан понимать данные, которые содержатся в графическом файле. Для отображения этих данных он вызывает OLE-функции. Если система не может самостоятельно отобразить эти данные, она вызывает сервер, т. е.Paint, который отображает данные в окне программы WordPad.

Такие приложения, как Word, WordPad, Excel и т. п., являются одновременно и клиентами, и серверами. Эти приложения могут как принимать OLE-объекты, предоставленные другими серверами, так и выступать в роли OLE-серверов по отношению к другим приложениям-клиентам. Редактор Write в Windows 3.x функционирует только как OLE-клиент и не располагает возможностями сервера. А вот программа Paint служит только OLE-сервером и неспособна исполнять роль клиента. Она может только редактировать изображения и передавать их другим приложениям.

Поскольку Paint является автономным приложением и в то же время представляет собой OLE-сервер, эта программа считается ***полным сервером***. В отличие от полных серверов ***мини-серверы*** не могут работать как автономные приложения и не содержат возможностей для открытия и сохранения файлов. Они могут только предоставлять свои услуги OLE-клиентам. Серверы любого типа могут предоставлять несколько типов услуг. Например, Quattro Pro предлагает услуги Quattro Pro Graph и Quattro Pro Notebook, а Microsoft Word — Microsoft Word Document и Microsoft Word Picture. Тип данных, экспортируемых сервером, обобщенно называется ***классом объекта***. Например, программа Paintbrush экспортирует объекты класса PBrush. Excel поддерживает классы

Excel Worksheet и Excel Chart. Серверы регистрируют свои классы в системном реестре. Каждым классом может управлять только один сервер.

Для каждого класса объектов регистрируется также набор команд. OLE-командой, или, по терминологии разработчиков, ***глаголом*** (*verb*), называется определенное действие, которое сервер может выполнять над объектом. Например, часто используются такие команды, как ***Edit*** и ***Play***. Когда пользователь выбирает объект в составном документе, приложение-клиент читает список команд для данного класса объектов и помещает соответствующие опции в меню. Пользователь манипулирует объектом, выполняя его команды. Различные объекты откликаются на разные команды.

Добавление объекта в документ-контейнер — это очень простой процесс. Из приложения-клиента пользователь выбирает тип вставляемого объекта. Например, в поле списка пользователю может предлагаться выбор изображения, электронной таблицы, видеоклипа и т. д. Этот список изменяется в зависимости от доступных серверов.

Предположим, вы работаете в Microsoft Word и решили внедрить в свой документ графическое изображение. Для этого запускается программа Paint — зарегистрированный сервер растровых изображений. Затем открывается BMP-файл, выбирается необходимая часть рисунка, которая копируется в буфер обмена. В Microsoft Word открывается файл документа и вызывается меню ***Edit***. В этом меню можно выбрать одну из трех команд: ***Paste***, ***Paste Link*** и ***Paste Special***. Все три команды предназначены для вставки графического изображения в текстовый файл. Простейшая команда***Paste***осуществляет внедрение объекта (если бы программа Paint не поддерживала механизм OLE, команда***Paste*** просто скопировала бы объект, а не внедрила его).

В составной документ можно также внедрять AVI- или WAV-файлы (видео- и звуковые вставки). Но каким образом внедренный видеоклип или звуковая вставка отображаются на экране? В таких случаях приложение создает специальное графическое представление, называемое ***пакетом***.

Пакет представляет собой значок, который указывает положение OLE-объекта в документе. При двойном щелчке на этом значке система определяет, какой тип данных содержится в пакете, и выполняет соответствующую команду. Для некоторых типов данных, таких как AVI- или WAV-файлы, представление данных на экране имеет смысл только в виде пакета. По умолчанию значок пакета передается из программы, в которой были созданы соответствующие данные.

Когда объект попадает в документ, клиент обеспечивает способы его активизации. Обычно активизация осуществляется двойным щелчком на объекте или его значке.

Так, например, при редактирования изображения в документе Microsoft Word OLE-библиотеки обеспечивают непосредственное, т. е. непосредственно в составном документе, редактирование изображения с помощью программы Paint. Для этого в меню ***Edit*** появляется подменю ***Bitmap Image Object*** с тремя командами — ***Edit***, ***Open*** и ***Convert***, которые передаются программой Paint. Для редактирования изображения с помощью программы Paint вызывается команда ***Edit***, являющаяся основной командой. При этом в окне Word появляются меню, палитра и панели инструментов программы Paint, что позволяет редактировать изображение, не покидая документа. При выборе команды***Open*** происходит открытие программы Paint, в которую загружается внедренный документ. Внесенные изменения отобразятся в документе Word при выходе из программы Paint, что послужит сигналом для обновления составного документа.

Составные документы можно передавать от одного пользователя к другому и читать на разных компьютерах с помощью одной и той же программы. Если в новой системе отсутствуют некоторые серверы, объекты все равно отображаются правильно, поскольку OLE-библиотеки самостоятельно обрабатывают стандартные форматы буфера обмена, например растровые изображения или метафайлы. Однако активизировать объекты без вызова сервера невозможно. Например, вы ничего не сможете сделать с WAV-пакетами без программы-проигрывателя.

Может показаться, что два разных механизма встраивания накладывают противоречивые требования на объекты данных. Чтобы объект отображался в любом приложении независимо от присутствия в системе исходной программы-сервера, объект должен содержать данные в некоем общем, легко распознаваемом формате, например в формате метафайла или растрового изображения.

С другой стороны, механизм OLE позволяет пользователю продолжить редактирование объекта даже после того, как объект вставлен в новый документ. Чтобы сервер мог редактировать,

объекты должны содержать данные в том формате, который используется для внутреннего представления объектов на сервере. Например, Excel не сможет продолжить редактирование ячеек электронной таблицы, которые были преобразованы в формат метафайла, но мало какая из программ-клиентов будет понимать внутренний формат данных Excel так хорошо, что отобразит электронную таблицу в системе, где Excel не установлен.

Решение этой проблемы заключается в предоставлении двух экземпляров данных для каждого OLE-объекта. Чуть позже вы увидите, как эта задача реализуется средствами сервера, но вкратце можно сказать, что почти каждый OLE-объект содержит данные в ***собственном*** (платформно-зависимом) формате в том виде, в котором эти данные были созданы сервером, а также в одном из нескольких стандартных ***форматов представления***(обычно в виде метафайла),которые могут отображаться любыми программами-клиентами.

Технология OLE организована с помощью функций высокого уровня, реализующих низкоуровневые процессы совместного использования данных. OLE-функции размещены в трех DLL-модулях. Библиотека OleCli32.DLL содержит все функции, которые используются OLE-клиентами, а библиотека OleSvr32.DLL содержит все функции, которые используются OLE-серверами. В технологии OLE 1.0 эти библиотеки обмениваются данными и командами с помощью DDE-сообщений. Третья библиотека, Shell32.DLL, поддерживает базу данных серверов и типов данных и обеспечивает правильную обработку OLE-запросов.

OLE-приложения взаимодействуют друг с другом с помощью функций OLE-библиотек. Например, если клиент хочет отредактировать изображение, он передает запрос библиотеке OleCli32.DLL, которая направляет его библиотеке OleSvr32.DLL. Последняя находит сервер и передает ему команду начать сеанс редактирования. Когда пользователь активизирует объект, библиотека OleSvr32 должна определить, какое приложение-сервер соответствует указанному формату данных. Для поиска соответствующего сервера OleSvr32 консультируется с библиотекой Shell32. Функции библиотеки Shell32 управляют системным реестром, в котором для каждого приложения записаны соответствующие форматы данных. При инициализации серверы вносят в системный реестр свое имя и другую информацию.

После выполнения операции библиотека OleSvr32 передает результаты библиотеке OleCli32, а та возвращает их клиенту. Схема взаимодействия клиента и сервера с помощью OLE-библиотек представлена на рис.8.5 [12].

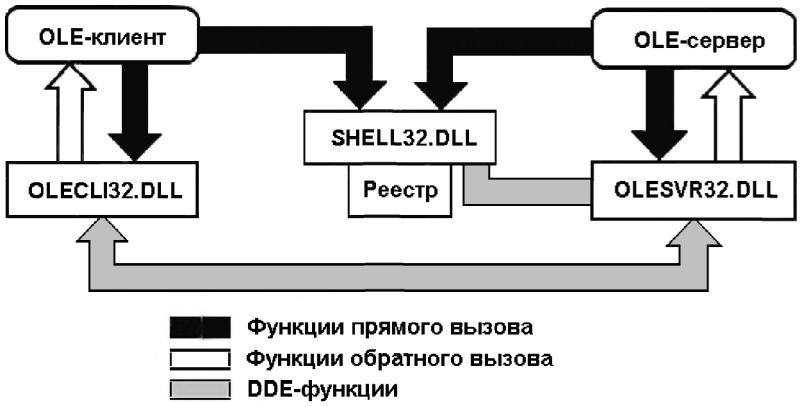

Рис.8.5. Взаимодействие клиента и сервера с помощью трех OLE-библиотек

Так же, как и DDEML, OLE-библиотеки используют в своей работе протокол DDE. OLE-команды посылают DDE-сообщения. Базовые DDE-процессы выполняются незаметно для OLE-приложения. Поскольку Microsoft разрабатывала технологии DDEML и OLE параллельно, они не опираются друг на друга.

Как уже упоминалось, при инсталляции или при первом запуске OLE-сервер регистрируется в системном реестре. Среди прочего регистрируются имя и местоположение сервера, а также типы предоставляемых им услуг. В свою очередь, OLE-клиент может найти в реестре необходимый ему сервер и услугу. Непосредственный доступ к реестру обеспечивается с помощью редактора реестра (файл RegEdit.EXE или RegEdit32.EXE в папке \WINDOWS или \WINDOWS\SYSTEM).

В качестве примера на рис. 8.6 показано окно редактора реестра после выполнения функции***Find*** из меню***Edit*** для нахождения информации о приложении Excel. В реестре имеется несколько записей для Excel, но одна из них представляет для нас особый интерес — это идентификатор класса. Он находится в разделе HKEY_CLASSES_ROOT\CLSID и выглядит следующим образом: 00020810-0000-0000-C000-000000000046. Как и для большинства остальных записей, эту информацию можно также найти в разделе HKEY_LOCAL_MACHINE\SOFTWARE\Classes\CLSID и в некоторых других разделах. Как видно на рис. 8.6, в реестр помещена информация о формате данных, опциях преобразования, стандартном расширении и значке, программном идентификаторе, а также об OLE-командах, которые поддерживаются соответствующим OLE-сервером.

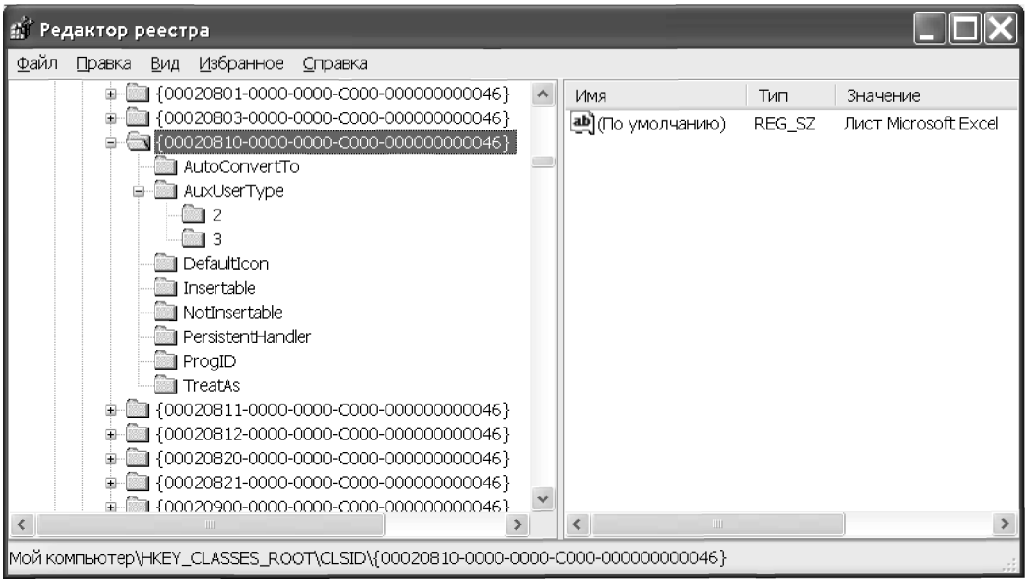

Рис.8.6. Окно редактора реестра с информацией о программе Excel

В большинстве случаев нет необходимости в прямом доступе к реестру. Существует много механизмов, обеспечивающих безопасный (хотя и ограниченный) косвенный доступ к информации, записанной в реестре. Один из них, а именно класс COleInsertDialog, будет рассмотрен позже.

Среда разработки Visual C++ содержит утилиту OLE View, которая предоставляет альтернативный способ просмотра информации об OLE-приложениях. Прежде чем OLE-клиент сможет воспользоваться услугами сервера, он должен выбрать сам сервер. Способ выбора сервера зависит от конкретной программной реализации. Например, в продуктах MS Offise в качестве OLE-серверов автоматически используются продукты, зарегистрированные для данного приложения в Windows. В других программах используются специальные настройки меню.

Полнофункциональные серверные приложения регистрируют себя автоматически при первом запуске. Если приложение было создано с помощью библиотеки MFC и мастера AppWizard, сервер автоматически регистрируется в процедуре InitInstance при выполнении метода COleTemplateServer::RegisterAll.

Для мини-серверов, которые не могут запускаться в виде независимых программ, необходимо применять иной подход. Мастер AppWizard обеспечивает регистрацию мини-сервера путем создания REG-сценария для серверного приложения.

Обычно при инсталляции приложения утилита Setup выполняет REG-сценарий. В процессе разработки можно выполнить этот сценарий непосредственно с помощью редактора реестра. Для этого в меню ***Registry*** редактора реестра выберите команду ***Import Registry File***, которая открывает диалоговое окно выбора файла. Выберите REG-сценарий для регистрации приложения.

8.5.2. Принципы разработки OLE-приложения

Раньше создание OLE-приложений было сложным и запутанным процессом, требовавшим написания сотен строк программного кода для реализации только самых элементарных клиентских операций. С помощью библиотеки MFC и мастера AppWizard (или его эквивалента в других компиляторах) процесс создания OLE-клиентов становится почти тривиальной задачей. Кроме того, эти же инструменты позволяют создавать несложные OLE-серверы.

Разработка OLE-клиента. На третьем этапе формирования заготовки программы-клиента с помощью мастера AppWizard вам будет предоставлена возможность включить в приложение поддержку составных документов (рис. 8.7). На первом этапе процесса работы с мастером вы должны выбрать поддержку многодокументного интерфейса. По умолчанию установлена опция *None*. Для реализации OLE-клиента нужно выбрать опцию ***Container***.

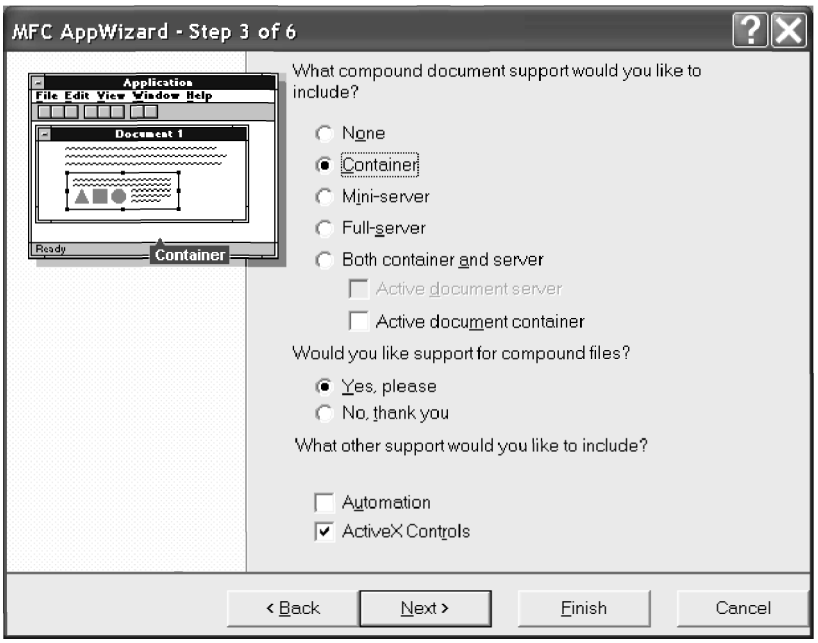

Рис.8.7. Реализация OLE-клиента

При создании OLE-клиента следует учитывать одно ограничение. Не задавайте для программы названий «OLEClient» или «OLEClient» (допускается имя «OLE_Client» с символом подчеркивания). Применение любого из указанных имен приводит к созданию класса COleClientDoc в качестве класса приложения, что вызывает конфликт с одноименным стандартным классом библиотеки, необходимым для поддержки OLE-клиентов.

По завершении работы мастера AppWizard библиотека MFC создает приложение с многодокументным интерфейсом, в котором содержится дополнительный класс, реализующий поведение OLE-контейнера — COle_ClientCntrItem.

Теперь OLE-клиент полностью готов к компиляции, компоновке и запуску. Еще более важно то, что программа готова к работе без дополнительных мер по обеспечению поддержки технологии OLE. Это совсем немало, учитывая сложность процесса создания OLE-клиента «с нуля». Если приказать мастеру AppWizard обеспечить поддержку OLE-клиента, он создаст класс COle_ClientView.

Разработка OLE-сервера. Библиотека MFC и мастер AppWizard позволяют создавать не только клиентские, но и серверные OLE-приложения (мини-серверы и полные серверы).

OLE-серверы создаются на основе трех базовых классов. Классы COleServerDoc и COleServerItem применяются всеми серверными приложениями. Класс COleTemplateServer используется полнофункциональными серверными приложениями.

Наиболее распространенным и вместе с тем простейшим в реализации типом OLE-серверов являются SDI-серверы (серверы однодокументного интерфейса). В каждом SDI-сервере имеется один объект сервера и один объект документа. В ответ на каждый запрос клиента запускается новый экземпляр сервера. Поскольку мини-серверы не поддерживают многократных соединений, мини-сервер SDI содержит только один рабочий объект. В отличие от мини-сервера полное сер-

верное приложение создает несколько объектов, когда несколько клиентов устанавливают связь с одним и тем же документом.

MDI-серверы (серверы многодокументного интерфейса) используются в том случае, когда загружать несколько экземпляров сервера нерационально или когда полное серверное приложение имеет многодокументный интерфейс. К MDI-серверам с несколькими экземплярами относятся такие приложения, как Excel и Quattro Pro, которые позволяют работать как с диаграммами (графическими объектами), так и с электронными таблицами. Каждый серверный класс содержит только один класс документа, а каждый объект сервера содержит только один объект документа. Каждый документ может предоставить несколько объектов, а каждый класс документа может поддерживать несколько классов объектов.

Для создания OLE-сервера на третьем этапе процесса работы с мастером AppWizard можно выбрать между созданием мини-сервера, полного сервера или приложения клиент/сервер (рис.8.8).

Создание мини-сервера и полного сервера, если рассматривать процесс относительно поддержки технологии OLE,выглядит совершенно одинаково. Но в процессе разработки гораздо проще тестировать полный сервер, поскольку он может работать в автономном режиме. При создании комбинированного клиент-серверного приложения происходит автоматическое создание полного сервера, а не мини-сервера, поскольку клиентская часть «настаивает» на работе приложения в автономном режиме (мини-сервер не может выступать в роли клиента, не имея пользовательского интерфейса).

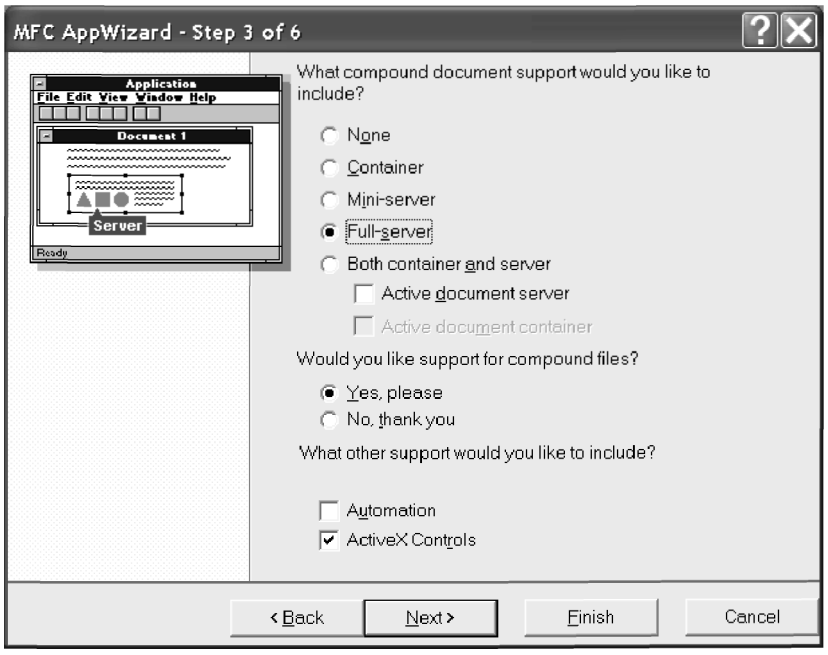

Рис.8.8. Реализация полного сервера

Сформировав заготовку программы, мастер AppWizard в дополнение к классам приложения, главного окна, окна документа и окна просмотра включает в приложение класс окна редактирования CInPlaceFrame и класс сервера CxxxSrvrItem, который является производным от ColeServerItem.

Глава 9
Принципы разработки программного кода для обработки мультимедийной информации

§ 9.1. Обзор мультимедийных устройств Windows

В операционной системе Windows поддержка технологии мультимедиа обеспечивает возможность работы с разнообразными периферийными аудио- и видеоустройствами. При этом реализуется интеграция множества различных форматов данных в единой системной среде. Средства мультимедиа можно условно разделить на три составные части: аппаратные аудио- и видеоустройства, драйверы этих устройств, обобщенные API-функции, которые переводят программные инструкции в команды для драйверов соответствующих устройств [12].

Мультимедийные операции связаны с интенсивными потоками данных. Они накладывают высокие требования на производительность центрального процессора и его возможность быстро обрабатывать большие объемы информации. Для обеспечения эффективной работы мультимедийные устройства должны поддерживать определенные стандартные функции и протоколы.

Чтобы обеспечить быстрое внедрение мультимедийных систем, Microsoft в сотрудничестве с другими разработчиками аппаратного и программного обеспечения создали спецификации Multimedia PC, которые устанавливают минимальные требования к таким системам. Аппаратным и программным продуктам, совместимым с этим стандартом, присваивается торговая марка MPC. Например, проигрыватель компакт-дисков должен передавать данные со скоростью не менее 150 Кб/с, используя при этом не более 40 % ресурсов центрального процессора. Среднее время поиска информации для такого устройства не должно превышать 1 с.

Типы мультимедийных устройств, которые поддерживает ОС Windows, представлены в табл. 9.1.

Таблица 9.1

Тип устройства	Описание
animation	Устройство воспроизведения анимации
cdaudio	Проигрыватель музыкальных компакт-дисков
dat	Цифровой магнитофон
digitalvideo	Устройство воспроизведения цифрового видео в окне (не средствами GDI)
mmovie	Анимационные файлы
other	Неопределенное MCI-устройство
overlay	Оверлейное устройство (воспроизводит аналоговое видео в окне)
scanner	Сканер изображений
sequencer	MIDI-синтезатор
videodisc	Проигрыватель лазерных дисков
waveaudio	Устройство воспроизведения оцифрованных звуков (звуковая карта)

Windows содержит набор драйверов, обеспечивающих поддержку различных аппаратных устройств и форматов данных. В недалеком прошлом эти устройства были недоступны большинству пользователей персональных компьютеров. Кроме драйверов устройств подсистема мультимедиа включает драйверы видеоадаптеров высокого разрешения, утилиты Sound Recorder (Звукозапись) и Media Player (Универсальный проигрыватель), а также несколько утилит в панели управления, предназначенных для инсталляции устройств и настройки мультимедийной среды.

Помимо аппаратного обеспечения и драйверов подсистема мультимедиа включает набор функций, образующих аппаратно-независимый интерфейс для мультимедийных программ. Так, существует единый набор команд для воспроизведения аудиофайлов с помощью любой звуковой карты. В ОС Windows функции, предназначенные для выполнения мультимедийных операций, находятся в библиотеке WinMM.DLL. Соответственно подсистема Win32, предназначенная для интерпретации и выполнения мультимедийных команд, называется WinMM. Следует отметить, что приложение, использующее команды WinMM, должно содержать файл Mmsystem.H и иметь связь с библиотекой Winmm.LIB (в среде Windows 3.1 мультимедийные функции находятся в файле Mmsystem.DLL). Таким образом, при переносе приложения из Windows 3.1 в более современные ОС семейства Windows необходимо убедиться в том, что в MAK-файле вместо Mmsystem.LIB включен файл Winmm.LIB.

Операционная система обеспечивает четыре различных способа управления мультимедийными операциями: два набора команд высокого уровня, один набор команд низкого уровня и набор команд для ввода/вывода файлов. Команды высокого и низкого уровней управляют одними и теми же мультимедийными устройствами. Первые удобнее в использовании, но вторые являются более мощными. Кроме того, поскольку все высокоуровневые команды реализованы с помощью функций низкого уровня, лучшего быстродействия можно достичь при непосредственном выполнении низкоуровневых команд. Команды низкого уровня взаимодействуют непосредственно с драйверами физических устройств. Команды высокого уровня взаимодействуют с драйверами логических устройств. В Windows есть три логических мультимедийных устройства: MIDI-синтезатор, проигрыватель компакт-дисков, проигрыватель аудиофайлов. Эти устройства преобразуют высокоуровневые команды в обращения к функциям низкого уровня, адресованные конкретным физическим устройствам. API высокого уровня, сформированный драйверами логических устройств, называется MCI (*Media Control Interface* — интерфейс управления средой). MCI-команды избавляют программиста от необходимости учета множества подробностей, связанных с управлением потоками данных, однако за счет некоторого уменьшения гибкости.

Мультимедийные команды низкого уровня необходимы только в специализированных программах. Функции высокого уровня позволяют воспроизводить MIDI-файлы, видеоклипы, записи с лазерных и компакт-дисков, а также записывать и воспроизводить звуковые файлы.

MCI поддерживает два параллельных набора функций: интерфейс команд и интерфейс сообщений. Командные строки и управляющие сообщения служат для выполнения одних и тех же операций, но первые используются в системах авторской разработки при написании пользовательских командных сценариев управления устройствами. Функция mciSendCommand() посылает управляющие сообщения, например MCI_OPEN или MCI_PLAY. Аналогичная функция mciSendString() посылает соответствующие строки, например openc:\sounds\harp.wav или playwaveaudioto 500.

Четвертый набор MCI-команд обеспечивает чтение и запись мультимедийных файлов. Мультимедийные команды ввода/вывода (Multimedia I/O — MMIO) воспринимают стандартный формат организации файлов — RIFF (он будет рассмотрен далее), а также осуществляют буферизацию, которая обеспечивает оптимальное распределение интенсивных потоков данных.

Поскольку синхронизация играет очень большую роль при выполнении мультимедийных операций, в частности при воспроизведении MIDI-файлов и координации работы различных устройств, осуществляющих одновременную обработку данных, подсистема WinMM поддерживает усовершенствованные функции таймера. Мультимедийный таймер не посылает сообщения WM_TIMER. Его работа основана на прерываниях. Центральный процессор регулярно получает сигналы прерываний от микросхемы таймера, а мультимедийный таймер во время этих прерываний активизирует функции обратного вызова, содержащиеся в вашей программе.

Сигналы таймера прерываний являются значительно более регулярными по сравнению с сообщениями, поскольку при их использовании не тратится время на ожидание в очереди сообщений приложения. Кроме того, точность мультимедийного таймера составляет 10 мс (для процессоров MIPS) или 16 мс (для процессоров Intel), а наименьший возможный интервал между вызовами функции SetTimer() составляет приблизительно 55 мс, причем даже такой интервал не всегда гарантирован из-за задержек в очереди сообщений. Временное разрешение таймера варьируется от системы к системе. Определить конкретное значение разрешения можно с помощью функции timeGetDevCaps(). Рассмотрим более подробно поддержку систем анимации и звука в WinAPI [12].

Анимация. Подсистема WinMM поддерживает воспроизведение анимации. Открыв устройство *mmovie*, можно запускать через него анимационные файлы. Функции работы с анимацией не связаны с GDI. Устройство mmovie во многом аналогично аудиоустройству, преобразующему файл данных в выходной поток. Проигрыватель видеоклипов читает данные из RIFF-файла, содержащего блоки данных в формате RMMP.

Поддерживаются такие анимационные эффекты, как персонажи, сцены, кисти, смена кадров и палитры, звуковое сопровождение, а также многое другое. Открытие устройства *mmovie* осуществляется с помощью сообщения MCI_OPEN, воспроизведение данных — с помощью сообщения MCI_PLAY, закрытие устройства — с помощью сообщения MCI_CLOSE. Существует также несколько дополнительных сообщений, специфичных для данного устройства. Например, сообщение MCI_STEPизменяет текущую позицию в файле на несколько кадров в прямом или обратном направлении. Сообщение MCI_WINDOWустанавливает и настраивает параметры окна, в котором происходит воспроизведение клипа. Но в основном анимационные MCI-команды работают аналогично командам воспроизведения звуковых файлов, и вы, зная один набор команд, легко разберетесь со вторым.

Аппаратные средства обработки звука. Первоначально интерфейс со звуковыми устройствами был введен в Windows 3.x под названием MME (*Multi Media Extension* — мультимедийное расширение). При переносе на платформу Win32 он практически не претерпел изменений. Звуковые устройства в Windows относятся к классу Multimedia/Audio. В данный класс первоначально входили три типа устройств: ***CD-DA*** (*Compact-Disk Digital Audio*), ***MIDI*** (*Musical Instrument Digital Interface*) и ***Wave*** (*waveform audio*).

Первый — это формат цифровых звукозаписей на компакт-дисках, который иногда называют ***Red Book*** — красная книга (по внешнему виду опубликованного документа, содержащего описание стандарта). Этот формат применяется при производстве музыкальных компакт-дисков. Каждая секунда такой записи занимает 176 Кб дискового пространства.

Другой, более компактный формат хранения данных представляет собой стандартный протокол взаимодействия музыкальных инструментов и компьютеров. MIDI-файлы состоят из команд воспроизведения различных музыкальных эффектов цифровыми синтезаторами, занимают гораздо меньше места на диске и позволяют создать высококачественную музыку, но для их записи необходима специальная MIDI-аппаратура.

Третий формат — формат файлов звукозаписи — позволяет записывать звук приемлемого качества без синтезатора, причем данные занимают меньше места на диске по сравнению с форматом Red Book. Этот формат обеспечивает запись звука путем дискретизации входного сигнала с определенной частотой и записи цифрового значения каждой выборки.

В Win32 дополнительно введен тип ***Aux*** — вспомогательные звуковые устройства (например, микшеры), при помощи которых реализуется управление параметрами звука, регулировки, настройки и т. п. Wave-устройства предоставляют весь необходимый сервис для записи и воспроизведения цифровых звуковых потоков в реальном времени с промежуточной буферизацией данных. Каждый из типов включает устройства ввода (*In*) и вывода (*Out*). Первые служат для записи звука от внешнего источника в приложение, вторые — для воспроизведения звука, порожденного приложением, или извлеченного из звукового файла, или полученного иным способом.

Типовой звуковой адаптер содержит стереофонические АЦП и ЦАП (аналого-цифровой и цифро-аналоговый преобразователи), микшер и управляющий цифровой процессор, координирующий работу всех узлов адаптера. Микшер расположен в аналоговой части адаптера. В его задачу входят: регулировка входных уровней различных источников звука(микрофона, линейного входа, компакт-диска, модема и т. п.), сведение всех источников в единый звуковой сигнал, поступающий на АЦП, а также регулировка выходного сигнала адаптера, снимаемого с ЦАП.

В режиме записи схема АЦП через равные интервалы времени опрашивает входной сигнал и формирует последовательность мгновенных значений амплитуды, называемых отсчетами. В зависимости от заданного режима разрядность отсчета (*sample width*) может быть разной: 8 или 16 бит — для простых адаптеров, от 18 до 24 бит — для сложных и качественных адаптеров. Чем больше разрядность отсчета, тем выше точность цифрового представления сигнала и ниже уровень шумов и помех, вносимых АЦП при оцифровке.

Частота, с которой АЦП опрашивает входной сигнал, называется частотой дискретизации (sample rate). Для точного цифрового представления сигнала частота дискретизации должна быть как минимум вдвое выше максимальной частоты сигнала. На практике обычно выбирается небольшой запас для компенсации погрешностей. Например, для представления сигналов с полосой частот до 10 кГц выбирается частота около 22 кГц.

Последовательность отсчетов, сформированная АЦП, передается управляющим процессором в основную память компьютера при помощи внепроцессорного доступа к памяти (DMA — на шине ISA, Bus Mastering — на шине PCI). После заполнения части области памяти, выделенной для обмена(обычно половины), адаптер подает сигнал аппаратного прерывания, по которому драйвер адаптера извлекает накопленные в памяти данные и переносит их в буфер программы, запросившей запись звука. После заполнения буфера программы драйвер подает ей программный сигнал, по которому программа переносит данные в нужное ей место: в другую область памяти для обработки, на диск, отображает на экране и т. п.

При воспроизведении звука происходит обратный процесс: программа записывает последовательность звуковых отсчетов в буфер и передает его драйверу, который по частям переносит данные в область памяти для DMA. Управляющий процессор адаптера последовательно извлекает из памяти отсчеты и направляет их на ЦАП, где они преобразуются в обычный электрический звуковой сигнал, который, пройдя через регуляторы микшера, попадает на выходной разъем адаптера.

Для удобства буфер обмена между процессором и звуковым адаптером делается циклическим (кольцевым). Это означает, что пока одна сторона (адаптер или ЦП) ведет запись первой половины буфера, другая сторона должна успеть прочитать данные из второй половины, и наоборот. Если быстродействия ЦП или драйвера не хватает или нарушается правильная работа системы аппаратных прерываний, то записываемый звук теряется, а воспроизводимый — зацикливается. Зацикливание короткого фрагмента воспроизводимого звука — типичный признак неверного выбора линии прерывания для адаптера или неисправности в системе прерываний.

Взаимодействие приложения с драйвером организуется в виде взаимного обмена ***потоками звуковых данных*** в реальном времени. От устройства ввода к приложению идет непрерывный поток записанного звука, от приложения к устройству вывода — непрерывный поток воспроизводимого. Приложение должно успевать принимать записываемый поток и формировать воспроизводимый, иначе в звуковых потоках возникают выпадения и помехи.

Звуковой буфер служит для переноса потоков между приложением и звуковым драйвером. Он представляет собой область памяти, в которой хранится небольшой фрагмент потока длительностью в десятки или сотни миллисекунд. Звуковые буферы создаются приложением и затем передаются драйверу: пустые — для устройств ввода, заполненные звуковыми данными — для устройств вывода. Драйвер ставит полученные буферы в очередь в порядке поступления; воспроизведение или запись данных ведется с начала очереди.

После завершения обработки каждого очередного буфера драйвер возвращает его приложению. С этого момента буфер доступен для повторного использования: он может быть заполнен новыми данными и снова передан этому же или другому звуковому устройству для постановки в очередь. Таким образом, между драйвером и приложением происходит циклическое «вращение» буферов, в которых переносятся звуковые потоки.

Для каждого звукового буфера приложение также создает заголовок (*header*) — дескриптор, куда заносятся параметры буфера и режимы его обработки. Обмен буферами между приложением и драйвером происходит в виде обмена их дескрипторами. Если к моменту завершения обработки буфера устройства вывода в очереди не имеется следующего буфера, то в выходном звуковом сигнале возникает пауза, но вывод потока не прерывается. Если программа не успевает передать драйверу очередной буфер для устройства записи, то фрагмент сигнала теряется.

Звуковые устройства делятся на ***синхронные*** и ***асинхронные***. Синхронному устройству для выполнения операций записи/воспроизведения требуются все ресурсы центрального процессора. Драйвер такого устройства, получив очередной буфер, не возвращает управления до тех пор, пока буфер не будет заполнен или проигран. В очереди драйвера синхронного устройства может находиться только один звуковой буфер. Асинхронное устройство работает независимо от центрального процессора, обрабатывая данные в выделенной области памяти и лишь изредка (один раз

в несколько десятков миллисекунд) сообщая драйверу о завершении обработки очередного фрагмента потока. Драйвер асинхронного устройства возвращает управление сразу же после получения очередного буфера, и в его очереди может находиться сколь угодно большое количество буферов.

Звуковые адаптеры, способные одновременно записывать и воспроизводить различные звуковые потоки, называются ***полнодуплексными*** (*full duplex*). Соответствующие устройства ввода и вывода в Windows могут быть открыты и использованы одновременно, при этом встречные потоки никак не влияют друг на друга.

Адаптеры, способные в каждый момент времени работать только в одном режиме (либо на запись, либо на воспроизведение), называются ***полудуплексными*** (*half duplex*). Из соответствующей пары устройств в Windows одновременно может быть открыто только одно — либо устройство ввода, либо устройство вывода. При попытке открытия второго устройства возвращается ошибка «устройство занято».

Некоторые адаптеры обладают возможностью ограниченной полнодуплексной работы, например: только в монофоническом режиме (ряд адаптеров на микросхемах ESS), только в восьмиразрядном режиме (большинство моделей Sound Blaster 16, AWE32, SB 32, AWE64) и т. п. В остальных режимах такие адаптеры работают только в полудуплексе.

При завершении обработки каждого буфера драйвер устанавливает в его заголовке флаг готовности, по которому приложение может определить, что драйвер освободил данный буфер. Однако для асинхронных устройств гораздо более эффективным способом возврата буфера является ***уведомление*** (*notification*), при котором драйвер либо вызывает заданную функцию приложения, либо активизирует событие (*event*), либо передает сообщение заданному окну или задаче (*thread*) приложения. При этом в параметрах функции или сообщения передается также указатель заголовка буфера.

Перед тем как быть переданным драйверу устройства, каждый звуковой буфер должен быть ***подготовлен*** (*prepared*). Как правило, подготовка заключается в фиксации буфера в памяти, так как большинство звуковых адаптеров пользуется внепроцессорными методами передачи данных, а эти методы требуют размещения буфера на одних и тех же адресах памяти. Передача драйверу неподготовленного буфера приводит к ошибке.

При работе со звуковыми адаптерами чаще всего используется традиционный способ цифрового кодирования PCM (Pulse Code Modulation — импульсно-кодовая модуляция (ИКМ)). Ряд мгновенных значений звуковой амплитуды, следующих друг за другом с частотой дискретизации, представляется рядом чисел выбранной разрядности, значения которых пропорциональны величине амплитуды. Именно в таком виде звуковой поток снимается с выхода АЦП или подается на вход ЦАП. Однако наряду с предельной простотой PCM обладает существенной избыточностью, передавая звук настолько точно, насколько это возможно при выбранных параметрах оцифровки. Зачастую при программировании звуковых устройств на первый план выходит задача минимизации скорости и объема звукового потока, в то время как отдельными параметрами точности и качества можно пренебречь. В таких случаях используются другие способы кодирования, когда звуковая информация представляется в виде относительных изменений амплитуды — ADPCM (*adaptive differential* PCM — адаптивная разностная ИКМ), мгновенных «снимков» спектра (Audio MPEG) и т. п. Обрабатывать звук в PCM способен любой звуковой адаптер. Иные способы кодирования аппаратно реализуются лишь в специализированных адаптерах.

Совокупность основных параметров потока: способа кодирования, частоты дискретизации, количества каналов (стерео / моно) и разрядности отсчета — называется ***форматом потока*** (*Wave Format*). Желаемый формат указывается при открытии устройства. Для смены формата требуется закрытие и повторное открытие устройства.

Главным параметром формата является способ кодирования, который называется еще признаком формата (*format tag*). Каждый способ кодирования порождает группу однотипных форматов, различающихся лишь точностью представления, а следовательно, и качеством передачи звука.

Основные частоты дискретизации (11 025, 22 050 и 44 100 Гц) в сочетаниях с различным количеством каналов (1 или 2) и различной разрядностью отсчета (8 или 16) при способе кодирования PCM образуют 12 типовых форматов. Частота 11 025 Гц (полоса звуковых частот примерно до 5 кГц)

приблизительно соответствует качеству телефонного сигнала, частота 22 050 Гц (полоса до 10 кГц) — среднего радиоприемника, частота 44 100 Гц (полоса до 20 кГц) — качественной звуковой аппаратуре.

Наименьшей единицей звукового потока является ***блок***. Соответственно размер каждого буфера, передаваемого звуковой подсистеме, должен быть кратен размеру блока, а объем данных, возвращаемый устройством ввода, всегда будет кратен размеру блока. В PCM блоком считается набор отсчетов, передаваемых за один период частоты дискретизации, а именно: один отсчет — для монофонических потоков, два—для стереофонических и т. д. Таким образом, блоки следуют друг за другом с частотой дискретизации, а отсчеты в блоках размещаются начиная с левого (нулевого) канала. Если отсчет занимает более одного байта, то байты размещаются в порядке по старшинству, в порядке возрастания, как это принято в процессорах Intel. Восьмиразрядные отсчеты в PCM представляются в виде беззнаковых целых чисел. За нуль сигнала принято «центральное» значение 128 (шестнадцатеричное — 80). Итак, предельной отрицательной амплитуде сигнала соответствует нулевое значение отсчета, а предельной положительной—значение FF. Для пересчета значений отсчетов в знаковую двуполярную форму в диапазоне от –128 до +127 из них нужно вычитать 128 (0×80) или прибавлять то же самое смещение, вычисляя по модулю 256, что дает такой же результат.

Отсчеты с разрядностью более восьми представляются в виде целых чисел со знаком в стандартном формате Intel. За нуль сигнала принято нулевое значение отсчета. Здесь может без каких-либо ограничений применяться обычная целая арифметика, например над типами ***short*** (16-разрядный) и ***long*** (32-разрядный). Если разрядность отсчета превышает 16, она может быть не кратна байту — современные звуковые адаптеры могут использовать 18-, 20- и 22-разрядные отсчеты. В таком случае отсчет выравнивается по старшей границе трех- или четырехбайтового слова, а лишние младшие разряды заполняются нулями. Подобное представление позволяет работать с отсчетами любой разрядности так же, как с 24- или 32-разрядными. От фактической разрядности отсчета зависит лишь точность полученного числа.

С момента запуска потока драйвер отслеживает текущую позицию записи или воспроизведения, которая в любой момент может быть запрошена приложением. Точность определения реальной позиции зависит от используемого устройства: она может указывать на конкретный звуковой отсчет либо на небольшую группу отсчетов, находящуюся в данный момент в обработке. Как правило, все адаптеры имеют скрытую небольшую очередь между встроенным процессором и ЦАП/АЦП, так что полученная от драйвера позиция почти всегда будет отличаться от реальной на несколько отсчетов. Драйвер отслеживает позицию путем подсчета количества звуковых блоков потока, переданных от приложения к устройству или наоборот. Если приложение не успевает своевременно передавать драйверу звуковые буферы, то вычисленная позиция будет отставать от действительного времени записи или воспроизведения.

Циклическое движение буферов может быть приостановлено и затем возобновлено с места прерывания с сохранением позиции в потоке. Поток может быть уничтожен (сброшен)—в этом случае драйвер немедленно возвращает приложению все ждущие буферы и обнуляет позицию потока. Для устройств воспроизведения один или несколько идущих подряд буферов могут быть зациклены (*looped*): в этом случае драйвер последовательно проигрывает их заданное количество раз, после чего возвращает приложению и переходит к воспроизведению следующих буферов из очереди.

Звуковая подсистема Windows допускает работу с устройством нескольких процессов (клиентов) одновременно. Многие современные и даже некоторые устаревшие звуковые устройства поддерживают более одного клиента; устройство вывода (или его драйвер) смешивает проигрываемые клиентами звуковые потоки, а устройство ввода размножает записываемый поток для всех подключенных клиентов.

Устройство, драйвер которого поддерживает не более одного клиента, не может быть повторно открыто до тех пор, пока клиент не закроет его. При попытке повторно открыть такое устройство звуковая подсистема возвращает сообщение об ошибке, сигнализирующее о том, что устройство занято.

Для упрощения реализации основных операций со звуком Windows содержит службу переназначения — ***WaveMapper***. Поскольку в ОС Windows может быть установлено более одного звукового устройства, существуют понятия стандартного системного устройства ввода и стандарт-

ного системного устройства вывода. Оба они задаются в закладке Audio-формы свойств мультимедиа. Приложение может запросить работу с конкретным звуковым устройством либо со стандартным системным. В последнем случае служба переназначения определяет нужное устройство.

Кроме трансляции запросов к нужному устройству, ***WaveMapper*** может выполнять и поиск наиболее подходящего устройства, поддерживающего требуемый формат звука.

В Win32 имеется подсистема сжатия звука (***Audio Compression Manager — ACM***), при помощи которой возможно взаимное преобразование звуковых форматов как внутри групп, так и между ними. Наряду с простыми преобразованиями (изменением частоты дискретизации, количества каналов или разрядности отсчета) ACM предоставляет широкий набор форматов сжатия: ADPCM, a-Law, mu-Law, MSN Audio, GSM, MPEG и т. п. Подсистема сжатия реализована в виде набора так называемых кодеков (ACM Codec) — специальных драйверов ACM, которые непосредственно занимаются переводом звука из одного формата в другой. Сам же ACM — это диспетчер, который взаимодействует с приложением и по запрошенным форматам активизирует нужные кодеки, снабжая их необходимыми параметрами. Служба ACM может использоваться как автономно— через собственный отдельный интерфейс, так и автоматически — службой ***Wave Mapper***.

Когда приложение открывает конкретное звуковое устройство, указывая требуемый формат потока, звуковая подсистема пытается связаться непосредственно с драйвером устройства. Однако если устройство не поддерживает требуемый формат, а обратившейся программой разрешена работа через ***Wave Mapper***, то подсистема может попытаться найти подходящий кодек ACM, который способен в реальном времени преобразовывать один из «родных» форматов устройства в формат, запрошенный приложением. Если такой кодек обнаружен, подсистема прозрачно включает его в работу между драйвером и приложением, а приложение может считать, что выбранный формат потока поддерживается непосредственно устройством.

Звуковая подсистема нумерует установленные устройства начиная с нуля. При установке нового устройства или удалении существующего нумерация изменяется, поэтому даже во время работы программы в системе могут появиться или исчезнуть звуковые устройства. Вместо номера звукового устройства может использоваться ключ (*handle*) ранее открытого устройства. Система автоматически определяет, какое именно значение передано интерфейсной функции. Как и в случае с файлами, при открытии каждого звукового устройства система возвращает его идентификатор, или ключ (*handle*), по которому затем происходит вся остальная работа с устройством. Формально идентификаторы устройств ввода и вывода имеют различные типы — ***HWAVEIN*** и ***HWAVEOUT***, однако оба они эквивалентны типу ***HWAVE***, который может использоваться для создания универсальных функций, не зависящих от типа устройства. Ключи звуковых устройств не имеют ничего общего с ключами файлов, событий, окон, задач и т. п. Системные функции ***DuplicateHandle()***, ***CloseHandle()*** и прочие к ним неприменимы.

Несмотря на то, что большая часть функций интерфейса одинакова либо симметрична для устройств ввода и вывода, разработчики Microsoft по непонятной причине разделили названия функций для устройств ввода и вывода: каждая функция имеет префикс, состоящий из типа и «ориентации» устройства: ***midiIn***, ***waveOut*** и т. п. С одной стороны, это способствует защите от ошибок, но с другой — усложняет создание универсальных функций и классов, в которых направление передачи задается параметром.

Главным недостатком звуковой подсистемы MME, реализованной в Windows 9x, является то, что подсистема и ее драйверы так и остались 16-разрядными, как и в Windows 3.x. Из-за этого каждое обращение к звуковому драйверу из Win32-приложения сопровождается двойной сменой режима исполнения (*thunking*), что приводит к дополнительным временным расходам, которые могут доходить до единиц миллисекунд. Кроме того, многие драйверы ограничивают частоту обновления кольцевого буфера, через который идет обмен между компьютером и адаптером, до нескольких десятков раз в секунду, отчего в процессе передачи звука возникает отставание (*latency*). У драйверов для адаптеров ISA это отставание может достигать десятков миллисекунд, у драйверов для адаптеров PCI оно обычно ограничивается единицами миллисекунд.

Для более оперативного вывода звука, особенно с модификацией его в реальном времени, Microsoft разработал более новый интерфейс — ***DirectSound***. Этот интерфейс призван «приблизить» аппаратуру адаптера к прикладной программе и позволяет ей практически напрямую запи-

сывать звук в системный кольцевой буфер, сводя максимальные задержки к единицам миллисекунд для любого адаптера. При работе с ***DirectSound*** программа обращается непосредственно к 32-разрядному системному драйверу адаптера (VxD), минуя переключения между 32- и 16-разрядным режимами исполнения.

В целях эффективной работы интерфейс ***DirectSound*** должен поддерживаться системным драйвером адаптера. Для устройств, драйверы которых не поддерживают ***DirectSound***, Windows эмулирует новый интерфейс «поверх» обычного MME-драйвера, но в этом случае все задержки даже возрастают из-за накладных расходов на эмуляцию.

К сожалению, Microsoft разработала спецификацию расширения ***DirectSound*** для звуковых VxD только в части воспроизведения звука, действуя прежде всего в интересах производителей игр. Запись звука через ***DirectSound*** до сих пор ведется путем эмуляции поверх MME.

Следует отметить, что звуковая подсистема Windows 3.x и 9x, равно как и подсистема удаленного доступа к сети (RAS), обладает низкой устойчивостью к ошибкам. Это чаще всего проявляется в том, что при аварийном завершении программы, открывшей звуковые устройства и работающей с ними, система не выполняет корректного закрытия (*cleanup*) используемых устройств. В результате этого в ряде случаев после такого аварийного завершения может потребоваться перезагрузка, а до тех пор незакрытые устройства будут недоступны другим приложениям. Кроме того, 16-разрядные подсистемы защищены от ошибок гораздо меньше 32-разрядных, так что серьезные ошибки в звуковых программах могут приводить к сбоям и «зависаниям» всей системы Windows.

В Windows NT все подсистемы сделаны изначально 32-разрядными, поэтому описанных проблем там не возникает, однако задержки ввода и вывода звука по-прежнему определяются частотой обновления кольцевого буфера, которая задается драйвером конкретного адаптера.

§ 9.2. Элементарные API-функции для обработки звука

Программные средства обработки звука. По общепринятой договоренности файлам звукозаписи присваивается расширение WAV. Для воспроизведения звука большинство программ используют одну из трех простейших функций, которые лучше всего подходят для работы с короткими WAV-файлами:

- MessageBeep() — воспроизводит системные звуки, занесенные в реестр в качестве сигналов предупреждений и сигналов об ошибках;
- SndPlaySound() — воспроизводит звук непосредственно из WAV-файла или из буфера в памяти;
- PlaySound() — новая функция Win32, которая во многом дублирует команду sndPlaySound, но имеет два отличия: она не воспроизводит фрагменты, находящиеся в памяти, но позволяет прослушивать звукозаписи, которые хранятся в ресурсах типа WAVE.

Функция MessageBeep() принимает единственный параметр, указывающий один из пяти системных звуков, сконфигурированных с помощью панели управления. Дополняя каждый вызов функции MessageBox() функцией MessageBeep(), можно сопровождать звуковыми сигналами все сообщения различного уровня важности. Например, если функция MessageBox() отображает значок MB_ICONHAND, в качестве аргумента MessageBeep() также должен задаваться параметр MB_ICONHAND. Воспроизводимый звук определяется записью SystemHand в системном реестре.

В табл. 9.2 приведен список допустимых значений параметра функции MessageBeep(), а также соответствующих им записей в реестре.

С помощью панели управления или редактора реестра пользователь может связать с этими сигналами любой WAV-файл. Подобно всем остальным функциям, связанным с воспроизведением звуков, функция MessageBeep() требует наличия драйвера соответствующего устройства. Обычный динамик персонального компьютера не является адекватным мультимедийным устройством.

Таблица 9.2

Значение параметра	Запись в реестре
0xFFFFFFFF	Стандартный сигнал, воспроизводимый через динамик компьютера
MB_ICONASTERISK	System Asterisk
MB_ICONEXCLAMATION	System Exclamation
MB_ICONHAND	System Hand
MB_ICONQUESTION	System Question
MB_OK	System Default

С помощью функции sndPlaySound() можно воспроизвести любой системный звук, зафиксированный в реестре и сконфигурированный с помощью панели управления (кроме пяти стандартных системных звуков могут использоваться дополнительные), или воспроизвести непосредственно WAV-файл:

```
BOOL sndPlaySound (LPCTSTR lpszSoundName,    // файл или запись реестра
    UINT uFlags);                            // флаги семейства SND_
```

Первый параметр указывает на запись в реестре, например SystemStart или SystemQuestion. Кроме того, он может содержать полный путь и имя WAV-файла. Функция sndPlaySound() требует свободного объема памяти, достаточного для полной загрузки всего файла звукозаписи. Она лучше всего работает со звуковыми файлами, размер которых не превышает 100 Кб.

В качестве второго параметра ожидается флаг, задающий параметры воспроизведения звука. Приведем несколько возможных значений этого параметра:

- **SND_MEMORY** — определяет первый параметр как указатель объекта, находящегося в памяти, а не как имя файла или системный звук;
- **SND_SYNC** — завершает воспроизведение звука и после этого возвращает управление программе;
- **SND_ASYNC** — возвращает управление программе немедленно и воспроизводит звук в фоновом режиме;
- **SND_ASYNC | SND_LOOP** — возвращает управление программе немедленно и в фоновом режиме воспроизводит звук до тех пор, пока программа не вызовет функцию sndPlaySound() со значением NULLв качестве первого параметра;
- **SND_NODEFAULT** — запрещает воспроизведение каких-либо звуков, если невозможно найти заданный файл. Обычно функция sndPlaySound() в таких случаях воспроизводит стандартный системный звук SystemDefault.

Для воспроизведения звуков, скомпилированных в виде ресурсов, лучше всего подходит функция PlaySound():

```
BOOL PlaySound (LPCTSTR lpszSoundName,    // имя файла или ресурса
    HANDLE hModule,                       // источник звукового ресурса
    DWORD dwFlags);                       // опции воспроизведения
```

Функция интерпретирует первый параметр в соответствии с установленными флагами:

- **SND_ALIAS** — воспроизводит звук, заданный в системном реестре. Первый параметр представляет собой псевдоним, записанный в реестре, например SystemAsterisk или SystemHand;
- **SND_FILENAME** — воспроизводит звук, записанный в WAV-файле, по аналогии с функцией sndPlaySound(). Первый параметр указывает имя файла;
- **SND_RESOURCE** — воспроизводит звук, содержащийся в программном ресурсе. Первый параметр представляет собой идентификатор ресурса, который возвращается макрокомандой MAKEINTRESOURCE.

Эти три флага являются взаимоисключающими. Кроме них функция PlaySound() распознает некоторые флаги, определенные для sndPlaySound(), например SND_NODEFAULT и SND_ASYNC. Флаг SND_MEMORY она не распознает.

Если в составе параметра dwFlags отсутствует флаг SND_RESOURCE, второй параметр — hModule — игнорируется. В противном случае параметр hModule идентифицирует программу, ресурсы которой включают звукозапись, указанную аргументом lpszSoundName. Данный дескриптор может быть получен с помощью функции GetModuleHandle(), LoadLibrary() или GetWindowLong().

В Windows не определено ключевое слово WAVE, которое могло бы использоваться в файлах ресурсов по аналогии с ключевыми словами ICON и BITMAP, но вы всегда можете самостоятельно определить собственный тип ресурса [12]:

```
<имя ресурса>WAVE<имя файла>                    // добавить звукозапись к ресурсам программы
```

Параметр <*имя ресурса*> представляет собой имя, используемое для нового ресурса, а параметр <*имя файла*> указывает на WAV-файл. Функция PlaySound() всегда ищет ресурсы, тип которых определен как WAVE.

Функции PlaySound(), MessageBeep() и sndPlaySound() просты в использовании, но им присущ ряд ограничений. Чтобы задавать точку начала воспроизведения, записывать и микшировать звуки, изменять уровень громкости, нужны дополнительные команды. Основным средством мультимедийного программирования является MCI-интерфейс. Приведем его описание.

MCI-операции организованы в виде командных сообщений, которые передаются устройствам. В общем случае операция начинается с открытия устройства, затем посылается команда, например MCI_PLAY или MCI_STOP, которая заставляет устройство начать воспроизведение, остановиться, начать запись, «перемотку» и т. д. Наконец, происходит закрытие устройства.

Самой важной и многоцелевой среди MCI-функций является функция mciSendCommand(). Она служит для передачи устройству одного из многочисленных сигналов:

```
MCIERROR mciSendCommand (MCIDEVICEID mciDeviceID,     // идентификатор устройства
        HINT uMessage,                                // номер сообщения
        DWORD dwFlags,                                // флаги
        DWORD dwParamBlock);                          // информационная структура
```

Первый параметр mciDeviceID представляет собой адрес конкретного устройства. При открытии устройства команда mciSendCommand возвращает его идентификатор, который сообщает Windows, куда должны быть адресованы сообщения, посылаемые последующими командами.

Второй параметр uMessage представляет собой константу, например:

- **MCI_OPEN** — открывает устройство (начало сеанса);
- **MCI_CLOSE** — закрывает устройство (конец сеанса);
- **MCI_SET** — изменяет установки устройства;
- **MCI_PLAY** — начинает воспроизведение;
- **MCI_STOP** — прерывает выполнение текущей операции;
- **MCI_RECORD** — начинает запись звука;
- **MCI_SAVE** — сохраняет записанный фрагмент в виде файла.

Третий параметр dwFlags обычно представляет собой комбинацию нескольких флагов, которые помогают Windows интерпретировать команду. Набор возможных флагов изменяется в зависимости от конкретного сообщения, но некоторые из них являются общими для всех сообщений. Например, функция mciSendCommand() обычно работает асинхронно. Когда эта команда инициирует операцию с устройством, она не ожидает завершения выполнения этой операции, а немедленно завершается, в то время как устройство продолжает выполнение операции в фоновом режиме. Если необходимо знать, когда выполнение операции закончится, следует установить флаг MCI_NOTIFY, и подсистема WinMM передаст сообщение о завершении, что может быть необходимым при закрытии устройства по завершении воспроизведения звука. Флаг MCI_WAIT инициирует синхронную работу (по команде mciSendCommand() выполнение программы останавливается до тех пор, пока устройство не завершит выполнение текущей задачи).

Последний параметр dwParamBlock также изменяется от сообщения к сообщению. Он всегда представляет собой структурированную переменную, содержащую либо информацию, которая необходима устройству для выполнения команды, либо пустые поля, которые будут заполнены устройством в результате выполнения команды.

§ 9.3. Принципы разработки программного кода для обработки формата RIFF

9.3.1. Структура формата RIFF

В целом следует отметить, что мультимедийные файлы соответствуют стандартному формату RIFF. Программистам, разрабатывающим мультимедийные приложения, необходимо разбираться в структуре RIFF-файлов и знать особенности MMIO-функций, предназначенных для записи и чтения этих файлов.

Протокол RIFF (*Resource Interchange File Format* — формат обмена файлами ресурсов) описывает файловую структуру с теговой организацией. Это означает, что файл может быть разбит на ряд нерегулярных блоков, помеченных особыми короткими строками — ***тегами***. Теги RIFF-файлов представляют собой четырех-символьные коды, например RIFF, INFO или PAL (четвертым символом тега PAL является пробел). Каждый тег начинает ***блок данных*** (*chunk*). Наиболее важные блоки начинаются с тега RIFF и могут содержать другие блоки, которые называются ***вложенными***. RIFF-файлы всегда начинаются с блока RIFF, а все остальные данные организованы в виде вложенных блоков первого блока.

Каждый блок состоит из трех частей: тега, значения размера и двоичных данных. Тег сообщает о типе последующих данных. Значение размера, имеющее тип DWORD, указывает объем данных, содержащихся в блоке. В конце данных находится тег следующего блока, если таковой имеется. Файлы звукозаписей всегда содержат не менее двух вложенных блоков: один из них предназначен для указания формата, другой — для самих данных. В некоторых блоках может быть записана информация об авторских правах и номере версии или содержаться список ***признаков*** (*cues*), т. е. позиций в файле, связанных с определенными событиями внутри других блоков или других файлов.

Блоки RIFFотличаются от большинства блоков тем, что их поля данных (т. е. раздел двоичных данных) всегда начинаются с четырех-символьного кода, обозначающего тип содержимого файла: звукозапись (WAVE), MIDI-файл (RMID), DIB-файл (RDIB), видеоклип (RMMP) или палитра (PAL).

Поскольку в RIFF-файлах содержится так много четырех-символьных кодов, существует специальная макрокоманда mmioFOURCC, предназначенная для создания этих кодов. Приведенная далее команда записывает тег RIFFв одно из полей информационной структуры:

```
MMCKINFO mmckinfo.ckid = mmioFOURCC ('R', 'I', 'F', 'F');
```

Структура MMCKINFO содержит информацию, описывающую отдельный блок данных. При чтении данных система заполняет поля описанием текущего блока. При записи данных вы заполняете информацию, которая необходима системе для сохранения-блока:

```
typedef struct _MMCKINFO /* структура блока данных RIFF-файла */
{
    FOURCC ckid;            // идентификатор блока
    DWORD cksize;           // размер блока
    FOURCC fccType;         // тип или список типов
    DWORD dwDataOffset;     // смещение блока данных в файле
    DWORD dwFlags;          // флаги, используемые MMIO-функциями
} MMCKINFO;
```

FOURCC представляет собой новый тип данных, основанный на типе DWORD (каждый символ кода записывается в один из четырех байтов). Третье поле — fccType — представляет собой тег типа содержимого, который следует после каждого тега RIFF. Поле fccType отсутствует в блоках, не имеющих идентификатора RIFF. На рис. 9.1 показана связь между родительским и вложенным блоками RIFF-файла [12].

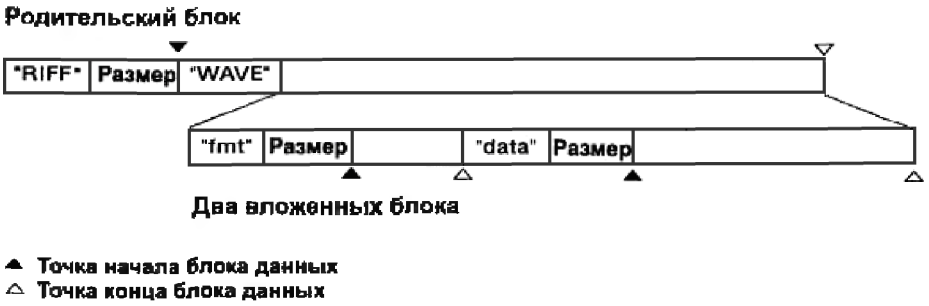

Рис. 9.1. Структура RIFF-файла

Как ранее указывалось, специализированный набор API-функций для мультимедиа содержится в подсистеме WinMM. Эти функции ориентированы на работу с блочной структурой RIFF-файлов. Кроме того, они обеспечивают буферизацию доступа к файлам. Далее рассмотрим основные функции интерфейса.

9.3.2. API-функции для обработки RIFF-файла

Функция mmioOpen() открывает файл и управляет параметрами его буфера:

```
HMMIO mmioOpen (LPTSTR lpszFilename,        // имя открываемого файла
        LPMMIOINFO lpmmioinfo,              // место для размещения информации о файле
        DWORD fdwOpen);                     // флаги
```

Первый параметр содержит имя файла, второй — информацию о его текущем состоянии. Если вы не хотите изменять настройки, заданные по умолчанию, например размер буфера ввода/вывода (8 Кб), то параметр *lpmmioinfo* должен содержать значение NULL. Третий параметр содержит набор флагов. Представим некоторые из них:

- **MMIO_READ** — допускает только чтение файла;
- **MMIO_WRITE** — допускает только запись файла;
- **MMIO_READWRITE** — допускает чтение и запись файла;
- **MMIO_CREATE** — создает новый файл;
- **MMIO_DELETE** — удаляет существующий файл;
- **MMIO_EXCLUSIVE** — предотвращает использование файла другими программами;
- **MMIO_DENYWRITE** — предотвращает изменение файла другими программами;
- **MMIO_ALLOCBUF** — обеспечивает буферизацию ввода/вывода.

В библиотеках языка С функция *fopen* начинает буферизованный ввод/вывод, а функция _open — небуферизованный ввод/вывод. Аналогичные действия можно задать с помощью флага MMIO_ALLOCBUF. Система откликается на соответствующую команду выделением буфера, размер которого задан по умолчанию (8 Кб). Чтобы увеличить или уменьшить размер буфера, нужно указать соответствующее значение в структуре MMIOINFO или вызвать функцию mmioSetBuffer().

Функция mmioOpen() возвращает дескриптор мультимедийного файла. Такие дескрипторы несовместимы с дескрипторами других файлов и их нельзя использовать с другими файловыми функциями С-библиотек или функциями подсистемы Win32.

Функции mmioRead(), mmioWrite() и mmioClose() выполняют с мультимедийными файлами операции чтения, записи и закрытия.

Ряд функций ввода/вывода специально ориентирован на выполнение операций с блоками данных. Чтобы добавить в файл новый блок, следует вызвать функцию mmioCreateChunk(). Эта команда записывает заголовок блока, включающий тег, размер, а также (для блоков RIFF и LIST) код содержимого и устанавливает позицию указателя файла на том байте, с которого начнется запись двоичных данных с помощью функции mmioWrite():

```
MMRESULT mmioCreateChunk (HMMIO hmmio,      // дескриптор RIFF-файла
        LPMMCKINFO lpmmcki,                 // описание нового блока
        UINT uOptions);                     // параметры создания
```

Для записи блоков RIFF и LIST установите флаги соответственно MMIO_CREATERIFF и MMIO_CREATELIST. Для перемещения указателя файла от блока к блоку служат функции mmioDescend() и mmioAscend(). Первая перемещает указатель на начало блока данных, вторая — на его конец:

```
MMRESULT mmioDescend (HMMIO hmmio,          // дескриптор RIFF-файла
        LPMMCKINFO lpmmcki,                 // место для записи информации о блоке
        LPMMCKINFO lpnunckiParent,          // необязательная структура родительского блока
        UINT uSearch);                      // флаги поиска
MMRESULT mmioAscend (HMMI Ohmmio,           // дескриптор RIFF-файла
        LPMMCKINFO lpmmcki,                 // место для записи информации о блоке
        UINT uReserved);                    // зарезервирован должен содержать 0
```

Функция mmioDescend() возвращает информацию о блоке в параметре lpmmcki. Кроме того, можно заставить ее выполнять поиск блоков определенного типа. Для инициации такого поиска последний параметр должен содержать значение **MMIO_FINDCHUNK**, **MMIO_FINDLIST** или **MMIO_FINDRIFF**. Поиск начинается с текущей позиции в файле и завершается в конце файла. Функция mmioAscend(), помимо перемещения к концу блока, помогает формировать новые блоки. Она вызывается после записи новых данных, выравнивая блок по четному байту и записывая объем данных в заголовок блока.

Каждая звукозапись в RIFF-файле должна содержать блок, помеченный тегом *fmt* (теги, состоящие из символов нижнего регистра, обозначают вложенные блоки.) Структура PCMWAVEFORMAT определяет содержимое этого блока:

```
/* универсальный формат файла звукозаписи (информация, общая для всех форматов) */
typedef struct waveformat_tag
{
        WORD wFormatTag;            // тип формата
        WORD nChannels;             // количество каналов (1= моно; 2 = стерео)
        DWORD nSamplesPerSec;       // частота оцифровки
        DWORD nAvgBytesPerSec;      //необходимо для оценки размера буфера
        WORD nBlockAlign;           // размер блока данных
}
WAVEFORMAT;
/* структура формата звукозаписи, специфичная для PCM-данных */
typedef struct pcmwaveformat_tag
{
        WAVEFORMAT wf;
        WORD wBitsPerSample;
}
PCMWAVEFORMAT;
```

В настоящее время для WAV-файлов определен только один формат — импульсно-кодовая модуляция PCM, поэтому поле wFormatTag структуры WAVEFORMAT должно содержать значение **WAVE_FORMAT_PCM**. В структуре PCMWAVEFORMAT к общему формату WAV-данных добавлено поле wBitsPerSample, указывающее разрядность выборки. Это поле определяет объем памяти, необходимый для записи каждой выборки. Обычно используются значения 8 и 16 битов. Монофоническая звукозапись длительностью 1 с, оцифрованная с частотой 11 кГц и разрядностью 8 битов, содержит 11 000 выборок, т. е. занимает объем примерно 11 Кб. При стереофонической звукозаписи происходит одновременная оцифровка сигналов в двух каналах. Если разрядность выборок в каждом из них равна 8 битам, то разрядность суммарной выборки составляет 16 битов. Объем стереофонической звукозаписи длительностью 1 с, частотой оцифровки 11 кГц и разрядностью 8 битов составляет 22 Кб.

§ 9.4. API-функции интерфейса DirectSound

Программно-аппаратный комплекс DirectSound представляет собой компонент технологии DirectX, обеспечивающий работу приложений с аудиоинформацией. Этот комплекс обеспечивает микширование данных с минимальной задержкой, аппаратное ускорение выполнения функций и прямой доступ к устройствам вывода аудиоинформации. Технология DirectSound построена на

нескольких интерфейсах API функций, каждый из которых предназначен для выполнения различных направлений работы со звуком. Рассмотрим основные интерфейсы комплекса DirectSound [4].

Для воспроизведения аудиоинформации DirectSound использует интерфейс ***IDirectSound***. Для записи звука средствами DirectSound используется интерфейс ***IDirectSoundCapture***. Он позволяет получить информацию о возможностях, предоставляемых установленным оборудованием для записи звука, и создать буферы, в которые будет помещаться записываемая аудиоинформация. Для создания интерфейса IDirectSoundCapture предназначена функция IDirectSoundCaptureCreate(), возвращающая адрес указателя на объект интерфейса IDirectSoundCapture. Для определения возможностей устройства записи аудиоинформации следует вызвать функцию GetCaps(), аргументом которого является объект структуры DSCCAPS. После вызова GetCaps() в объекте структуры DSCCAPS содержится информация о возможностях устройства записи аудиоинформации, включая количество используемых каналов и набор поддерживаемых стандартов формата записи. Стандартные форматы записи приведены в структуре WAVEINCAPS:

- **WAVE_FORMAT_1M08** — монофонический звук с частотой дискретизации 11,025 КГц и 8-битовым кодированием отсчета;
- **WAVE_FORMAT_1M16** — монофонический звук с частотой дискретизации 11,025 КГц и 16-битовым кодированием отсчета;
- **WAVE_FORMAT_1S08** — стереофонический звук с частотой дискретизации 11,025 КГц и 8-битовым кодированием отсчета;
- **WAVE_FORMAT_1S16** — стереофонический звук с частотой дискретизации 11,025 КГц и 16-битовым кодированием отсчета;
- **WAVE_FORMAT_2M08** — монофонический звук с частотой дискретизации 22,05 КГц и 8-битовым кодированием отсчета;
- **WAVE_FORMAT_2M16** — монофонический звук с частотой дискретизации 22,05 КГц и 16-битовым кодированием отсчета;
- **WAVE_FORMAT_2S08** — стереофонический звук с частотой дискретизации 22,05 КГц и 8-битовым кодированием отсчета;
- **WAVE_FORMAT_2S16** — стереофонический звук с частотой дискретизации 22,05 КГц и 16-битовым кодированием отсчета;
- **WAVE_FORMAT_4M08** — монофонический звук с частотой дискретизации 44,1 КГц и 8-битовым кодированием отсчета;
- **WAVE_FORMAT_4M16** — монофонический звук с частотой дискретизации 44,1 КГц и 16-битовым кодированием отсчета;
- **WAVE_FORMAT_4S08** — стереофонический звук с частотой дискретизации 44,1 КГц и 8-битовым кодированием отсчета;
- **WAVE_FORMAT_4S16** — стереофонический звук с частотой дискретизации 44,1 КГц и 16-битовым кодированием отсчета.

Для работы с буферами воспроизведения используются интерфейсы ***IDirectSoundBuffer*** и ***IDirectSound3DBuffer***. Для работы с буферами записи используется интерфейс ***IDirectSoundCaptureBuffer***. Для создания буфера записи аудиоинформации предназначен метод CreateCaptureBuffer(). Одним из аргументом данного метода является объект структуры DSCBUFFERDESC, содержащий параметры создаваемого буфера записи. После создания объекта буфера DirectSoundCapture вызов метода GetCaps() позволяет получить описание его параметров. Информация о формате данных, используемых этим буфером записи, может быть получена вызовом метода GetFormat(). Определение текущего состояния буфера записи определяется функцией GetStatus().

Для работы с расширенными возможностями звуковых карт используется интерфейс ***IKsPropertySet***. Интерфейс ***IDirectSoundNotify*** служит для отметки события о том, что при воспроизведении или при записи была достигнута определенная позиция в буфере.

Список литературы

1. *Вильямс А.* Системное программирование в Windows 2000. — СПб.: Питер, 2001. — 621 с.
2. *Гордеев А. В., Молчанов А. Ю.* Системное программное обеспечение. — СПб.: Питер, 2001. — 736 с.
3. *Карпов В. Е., Коньков К. А.* Основы операционных систем. — М.: Интернет-университет информационных технологий, 2005. — 536 с.
4. *Марапулец Ю. В.* Основы программирования в Win32 API. — Петропавловск-Камчатский: КамчатГТУ, 2004. — 148 с.
5. *Олифер В. Г., Олифер Н. А.* Сетевые операционные системы. — СПб.: Питер, 2006. — 538 с.
6. *Пэтзолд Ч.* Программирование для Windows 95. — СПб.:BHV:Санкт-Петербург, 1997. — 523 с.
7. *Рихтер Д.*Windows. Создание эффективных Win32-приложений с учетом специфики 64-разрядной версии Windows. — СПб.: Питер, 2001. — 624 с.
8. *Соломон Д., Русинович М.* Внутреннее устройство Windows 2000. — СПб.: Питер, 2004. — 746 с.
9. *Финогенов К. Г.* Win32. Основы программирования. — М.: Диалог-МИФИ, 2002. — 416 с.
10. *Харт Дж. М.* Системное программирование в среде Win32. — М.: Вильямс, 2001. — 463 с.
11. *Щупак Ю. А.* Win32 API. Эффективная разработка приложений. — СПб.: Питер, 2007. — 572 с.
12. *Эззель Б., Блейни Д.* Windows 98. Руководство разработчика. — Киев: Ирина:BHV, 1999. — 672 с.

Оглавление

Учебное издание

Марапулец Юрий Валентинович

СИСТЕМНОЕ ПРОГРАММИРОВАНИЕ В WINAPI

Издание второе, исправленное и дополненное

Авторская редакция

Федеральное государственное бюджетное образовательное учреждение высшего образования
«Камчатский государственный университет имени Витуса Беринга»
683032, Петропавловск-Камчатский, ул. Пограничная, 4
Тел. 8(415-2) 42-68-42, www.kamgu.ru

Подписано в печать 23.04.2019. Формат 60 × 84 / 8
Бумага офсетная. Печать цифровая
Гарнитура «TimesNewRoman». Усл. печ. л. 22,75. Уч.-изд. л. 16,31
Тираж 500 экз.

Отпечатано в КамГУ им. Витуса