А.В.Гордеев

А.Ю.Молчанов

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Электронный вариант книги издательства

«Питер»

Санкт–Петербург

Челябинск

ЮУрГУ

Каф. Автоматика и управление

2002

Предисловие

Настоящий учебник предназначается в первую очередь студентам технических вузов, хотя может быть востребован и обычными подготовленными пользователями, желающими углубить свои познания в области системного программного обеспечения.

Согласно Государственному образовательному стандарту, по которому ведётся обучение студентов, поступивших в вузы до 2000 г., в рамках дисциплины «Системное программное обеспечение», относящейся к обязательным специальным дисциплинам учебного плана по направлению «Информатика и вычислительная техника», студент должен изучить следующие обязательные разделы: *«…назначение, функции и структура операционной системы (ОС); понятие процесса; управление процессами, способы диспетчеризации процессов; понятие ресурса, виды ресурсов, управление ресурсами; управление памятью; устройства, виды устройств, драйверы устройств, устройства в MS-DOS; файловая система на диске, структура логического диска в MS-DOS; синхронизация процессов, семафоры, сообщения, использование семафоров для решения задач взаимоисключения и синхронизации; тупики, способы борьбы с тупиками; загрузка и настройка ОС, файлы конфигурирования MS-DOS, основное команды MS-DOS; обзор современных ОС; трансляторы; формальные языки и грамматики, типы грамматик; вывод цепочек; конечный и магазинный автоматы, распознаватели и преобразователи, построение автомата по заданной грамматике; структура компиляторов и интерпретаторов, лексический, синтаксический и семантический анализаторы, генератор кода; распределение памяти, виды переменных; статическое и динамическое связывание; загрузчики; функции загрузчика; настраивающий и динамический загрузчики; подключение библиотек».*

В новой редакции Государственного образовательного стандарта, который вступил в силу в 2000 г. и относится к поколению студентов, поступивших в вузы осенью 2000 г. и позже, несколько изменено основное содержание этой дисциплины. В частности, в стандарте записано, что в рамках дисциплины «Системное программное обеспечение» должны изучаться следующие обязательные разделы:

«...пользовательский интерфейс операционной среды; управление задачами; управление памятью; управление вводом/выводом; управление файлами; пример современной операционной системы; программирование в операционной среде; ассемблеры; мобильность программного обеспечения; макроязыки; трансляторы; формальные языки и грамматики, типы грамматик; вывод цепочек; конечный и магазинный автоматы, распознаватели и преобразователи, построение автомата по заданной грамматике; структура компиляторов и интерпретаторов, лексический, синтаксический и семантический анализаторы, генератор кода; распределение памяти, виды переменных; статическое и динамическое связывание; загрузчики; функции загрузчика; настраивающий и динамический загрузчики; подключение библиотек».

Таким образом, мы можем констатировать, что в основном в дисциплине «Системное программное обеспечение» должно уделяться внимание операционным системам, средам и системам программирования. Именно в таком ключе в основном и строится настоящий учебник, поскольку предполагается, что его читателями, прежде всего, будут студенты, обучающиеся по специальностям, относящимся к направлению «Информатика и вычислительная техника». Учебный материал, ставший основой для настоящего учебника, уже в течение нескольких лет изучается студентами специальности 22.01.00 в Санкт-Петербургском государственном университете аэрокосмического приборостроения. Другими словами, по существу, в основу учебника лег расширенный конспект лекций по дисциплине «Системное программное обеспечение». Эта дисциплина изучается в течение двух семестров. В первом семестре рассматриваются операционные системы (принципы их построения и функционирования, вопросы создания параллельных взаимодействующих задач, выполняющихся в мультизадачных операционных системах), а во втором – формальные грамматики, трансляторы и системы программирования. Поэтому книга разбита на две крупные части. Эти две части связаны между собой не потому, что так построен план изучения дисциплины. Материал, рассматриваемый в каждой из частей учебника, тесно связан с вопросами, изучаемыми в другой её части. Таким образом, детальное изучение материала любой части книги требу-

ет, по крайней мере, знакомства с основными понятиями всего учебника. Так, например, изучение структуры и технических аспектов работы компиляторов невозможно без знания принципов распределения памяти, которые относятся к вопросам построения операционных систем. Именно поэтому эти два крупных раздела объединены авторами в одну книгу и совместно представляются вниманию читателей.

Помимо общетеоретических вопросов в учебнике рассмотрены и отдельные практические вопросы, описаны конкретные реализации различных системных программ.

В первой части учебника, прежде всего, излагаются основные понятия ОС, принципы их построения и функционирования. В последние годы практически повсеместно ПК работают под управлением современных 32-битовых ОС, использующих аппаратные возможности микропроцессоров для создания и организации эффективных и защищённых вычислений. Мы посчитали необходимым рассмотреть в первой части учебника эти вопросы.

Наиболее популярными ОС являются системы Windows 95/98, Windows NT 4.0, начинается переход к Windows ME и семейству ОС Windows 2000 компании Microsoft. По этим ОС имеется огромное количество самых разнообразных публикаций, в том числе и учебных материалов, объём которых порой очень велик. В то же время по остальным ОС публикаций существенно меньше. Поэтому в первой части настоящего учебного пособия мы в качестве примеров операционных систем и сред кратко рассматриваем такие ОС, как OS/2 Warp, UNIX и Linux, QNX. Естественно, что отдельные вопросы иллюстрируются и на примере популярных ОС Windows 95/98 и Windows NT 4.0.

Во второй части учебника рассматриваются как общие вопросы, связанные с построением трансляторов, так и методы их практической реализации от примитивных распознавателей текста до законченных систем программирования. Практическая реализация компиляторов и интерпретаторов рассматривается с точки зрения современных широко распространенных языков программирования высокого уровня, таких как Pascal, C и C++.

Кратко рассматриваются также практические вопросы построения прикладных программ на основе архитектуры «клиент–сервер» и трехзвенной архитектуры, ориентированной на работу с серверами баз данных и серверами приложений. Эти вопросы затрагиваются не с точки зрения технологии их реализации (такие сведения можно найти в появившейся сейчас специализированной литературе) в той или иной ОС, а со стороны методов разработки соответствующих прикладных программ с помощью той или иной системы программирования. Для более детального знакомства с конкретными реализациями читатели могут воспользоваться приводимыми авторами ссылками на литературные источники или на соответствующие технические публикации в глобальной сети Интернет.

Таким образом, данный учебник должен быть полезен не только тем, кто хочет детально изучить системное программное обеспечение, но и тем, кто собирается сам разработать отдельные компоненты, в том числе отдельные системные утилиты, распознаватели и интерпретаторы команд, компоновщик или транслятор с некоторого языка, создать комплекс параллельно исполняющихся взаимодействующих программ. В учебнике рассматриваются вопросы, которые полезно знать в любом случае, если разработчик прикладной программы имеет дело с некоторым входным языком (которым может быть далеко не только язык программирования, но и любой другой язык команд, в том числе заданный пользователем или определенный в некоторой прикладной области).

Примеры систем программирования, которые рассматриваются в этом учебнике, предназначены для работы в среде перечисленных выше операционных систем. Фрагменты программ, приведенные в книге, написаны на языках программирования высокого уровня Pascal и C.

В этом коротком предисловии мы, как авторы этой книги, хотим ещё высказать самые теплые слова благодарности своим родным и близким за их долгое терпение, доброжелательность и сердечную заботу в течение всего времени подборки материалов, написания книги и её «бесконечного» улучшения, благодаря которым только и мог появиться на свет этот труд, вынужденно оторвавший нас на некоторое время от домашних забот и хлопот. Хочется поблагодарить и Гордеева В. А. за

конструктивную критику и помощь в подготовке отдельных программ. Авторы признательны также сотрудникам издательства «Питер» Васильеву А. В. и Ваулиной Е. Ю. за их терпеливое и внимательное отношение в процессе подготовки текста книги, его обработки и корректуры. Наше взаимное уважение и сотрудничество позволили довести учебный материал книги «до ума», хотя это и не всегда удавалось сделать в заранее оговоренные сроки.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу электронной почты comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

Подробную информацию о наших книгах вы найдете на Web-сайте издательства http://www.piter.com.

Часть 1
Операционные системы и среды

В англоязычной технической литературе термин System Software (системное программное обеспечение) означает программы и комплексы программ, являющиеся общими для всех, кто совместно использует технические средства компьютера, и применяемые как для автоматизации разработки (создания) новых программ, так и для организации выполнения программ уже существующих. С этой точки зрения программное обеспечение может быть разделено на следующие пять групп:

1 Операционные системы (ОС).

2 Системы управления файлами (СУФ).

3 Интерфейсные оболочки для взаимодействия пользователя с ОС и программные среды.

4 Системы программирования.

5 Утилиты.

Рассмотрим вкратце эти группы системных программ.

1 *Под операционной систе*мой (ОС) обычно понимают комплекс управляющих и обрабатывающих программ, который, с одной стороны, выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами, а с другой – предназначен для наиболее эффективного использования ресурсов вычислительной системы и организации надёжных вычислений. Любой из компонентов прикладного программного обеспечения обязательно работает под управлением ОС. На рис.1 изображена обобщённая структура программного обеспечения вычислительной системы.

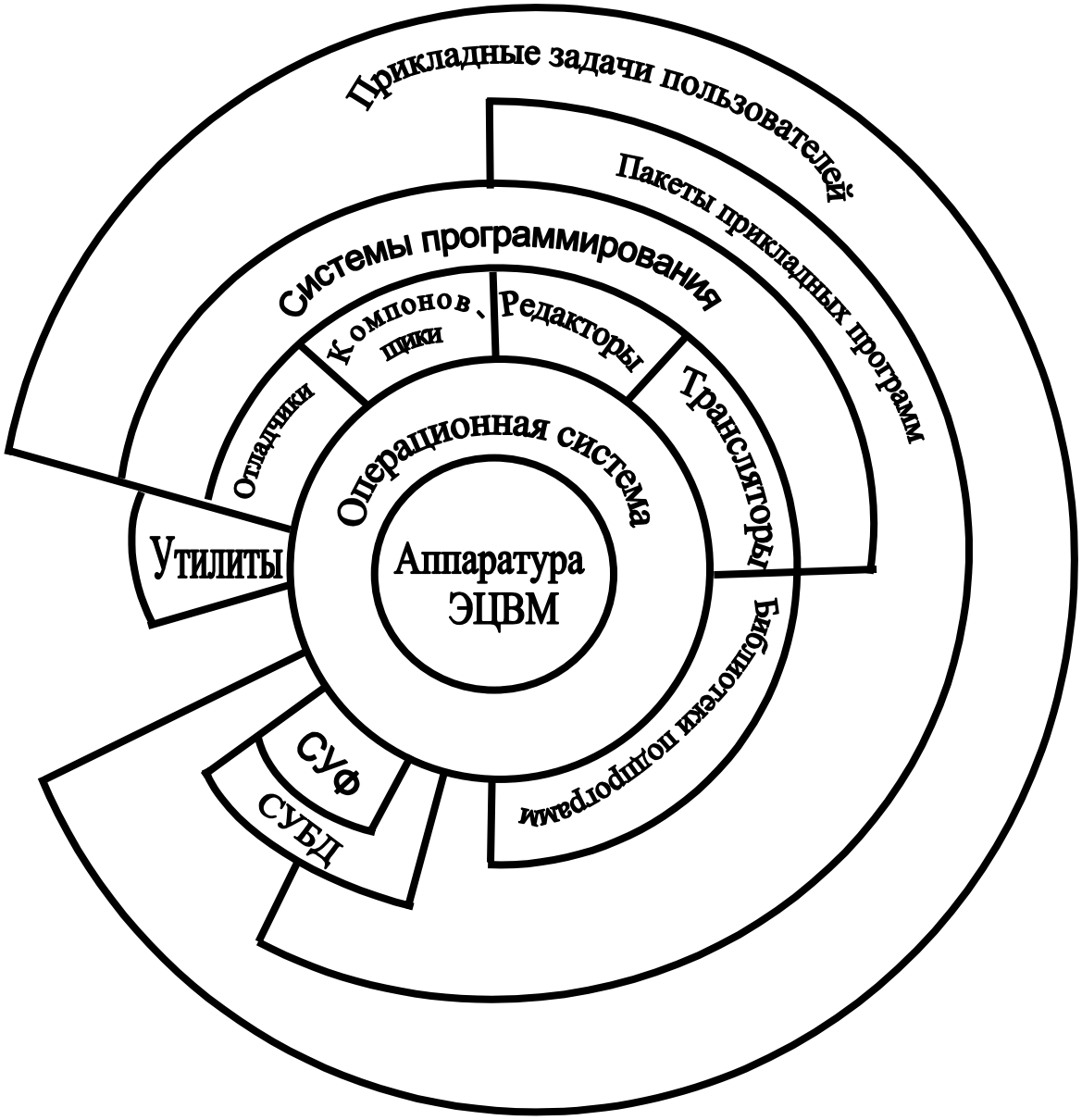

Рис.1 Обобщённая структура программного обеспечения вычислительной системы

Видно, что ни один из компонентов программного обеспечения, за исключением самой ОС, не имеет непосредственного доступа к аппаратуре компьютера. Даже пользователи взаимодействуют со своими программами через интерфейс ОС. Любые их команды, прежде чем попасть в прикладную программу, сначала проходят через ОС.

Основными функциями, которые выполняют ОС, являются следующие:

♦приём от пользователя (или от оператора системы) заданий или команд, сформулированных на соответствующем языке – в виде директив (команд) оператора или в виде указаний (своеобразных команд) с помощью соответствующего манипулятора (например, с помощью «мыши»), – и их обработка;

♦приём исполнение программных запросов на запуск, приостановку, остановку других программ;

♦загрузка в оперативную память подлежащих исполнению программ;

♦инициация программы (передача ей управления, в результате чего процессор исполняет программу);

♦идентификация всех программ и данных;

♦обеспечение работы систем управления файлами (СУФ) и/или системы управления базами данных (СУБД), что позволяет резко увеличить эффективность всего программного обеспечения;

♦обеспечение режима мультипрограммирования, то есть выполнения двух или более программ на одном процессоре, создающее видимость их одновременного исполнения;

♦обеспечение функций по организации и управлению всеми операциями ввода/вывода;

♦удовлетворение жёстким ограничениям на время ответа в режиме реального времени (характерно для соответствующих ОС);

♦распределение памяти, а в большинстве современных систем и организация виртуальной памяти;

♦планирование и диспетчеризация задач в соответствии с заданными стратегией и дисциплинами обслуживания;

♦организация механизмов обмена сообщениями и данными между выполняющимися программами;

♦защита одной программы от влияния другой; обеспечение сохранности данных;

♦предоставление услуг на случай частичного сбоя системы;

♦обеспечение работы систем программирования, с помощью которых пользователи готовят свои программы.

2 Назначение *системы управления файлами* – организация более удобного доступа к данным, организованным как файлы. Именно благодаря системе управления файлами вместо низкоуровневого доступа к данным с указанием конкретных физических адресов нужной нам записи используется логический доступ с указанием имени файла и записи в нём. Как правило, все современные ОС имеют соответствующие системы управления файлами. Однако выделение этого вида системного программного обеспечения в отдельную категорию представляется целесообразным, поскольку ряд ОС позволяет работать с несколькими файловыми системами (либо с одной из нескольких, либо сразу с несколькими одновременно). В этом случае говорят о монтируемых файловых системах (дополнительную систему управления файлами можно установить), и в этом смысле они самостоятельны. Более того, можно назвать примеры простейших ОС, которые могут работать и без файловых систем, а значит, им необязательно иметь систему управления файлами, либо они могут работать с одной из выбранных систем. ***Надо, однако, понимать, что любая система управления файлами (СУФ) не существует сама по себе – она разработана для работы в конкретной ОС и с конкретной файловой системой***. Можно сказать, что всем известная файловая система FAT (file allocation table)[1] имеет множество реализаций как система управления файлами, например, FAT-16 для самой MS-DOS, super-FAT для OS/2, FAT 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 соответствую-

[1] Здесь и далее без указания на источник заимствования приводятся английские эквиваленты слов и словосочетаний

щая система управления файлами; и эта система управления файлами будет работать только в той ОС, для которой она и создана.

Для удобства взаимодействия с ОС могут использоваться дополнительные интерфейсные оболочки. Их основное назначение – либо расширить возможности по управлению ОС, либо изменить встроенные в систему возможности. В качестве классических примеров интерфейсных оболочек и соответствующих операционных сред выполнения программ можно назвать различные варианты графического интерфейса X Window в системах семейства UNIX (например, K Desktop Environment в Linux), PM Shell или Object Desktop в OS/2 с графическим интерфейсом Presentation Manager; наконец, можно указать разнообразные варианты интерфейсов для семейства ОС Windows компании Microsoft, которые заменяют Explorer и могут напоминать либо UNIX с его графическим интерфейсом, либо OS/2, либо MAC OS. Следует отметить, что о семействе ОС компании Microsoft с общим интерфейсом, реализуемым программными модулями с названиями Explorer (в файле system.ini, который находится в каталоге Windows, имеется строка SHELL=EXPLORER.EXE), всё же можно сказать, что заменяемой в этих системах является только интерфейсная оболочка, в то время как сама операционная система остаётся неизменной; она интегрирована в ОС. Другими словами, операционная среда определяется программными интерфейсами, то есть API (application program interface). Интерфейс прикладного программирования (API) включает в себя управление процессами, памятью и вводом/выводом.

Ряд операционных систем могут организовывать выполнение программ, созданных для других ОС. Например, в OS/2 можно выполнять как программы, созданные для самой OS/2, так и программы, предназначенные для выполнения в среде MS-DOS и Windows 3.x. Соответствующая операционная среда организуется в операционной системе в рамках отдельной виртуальной машины. Аналогично, в системе Linux можно создать условия для выполнения некоторых программ, написанных для Windows 95/98. Определёнными возможностями исполнения программ, созданных для иной операционной среды, обладает и Windows NT. Эта система позволяет выполнять некоторые программы, созданные для MS-DOS, OS/2 1.x, Win-

dows 3.x. Правда, в своём последнем семействе ОС Windows 2000 разработчики решили отказаться от поддержки возможности выполнения DOS–программ.

Наконец, к этому классу системного программного обеспечения следует отнести и эмуляторы, позволяющие смоделировать в одной операционной системе какую-либо другую машину или операционную систему. Так, известна система эмуляции WMWARE, которая позволяет запустить в среде Linux любую другую ОС, например, Windows. Можно, наоборот, создать эмулятор, работающий в среде Windows, который позволит смоделировать компьютер, работающий под управлением любой ОС, в том числе и под Linux.

Таким образом, термин *операционная среда* означает соответствующий интерфейс, необходимый программам для обращения к ОС с целью получить определённый сервис[1]– выполнить операцию ввода/вывода, получить или освободить участок памяти и т. д.

3 *Система программирования* на рис.1 представлена такими компонентами, как транслятор с соответствующего языка, библиотеки подпрограмм, редакторы, компоновщики и отладчики. Не бывает самостоятельных (оторванных от ОС) систем программирования. Любая система программирования может работать только в соответствующей ОС, под которую она и создана, однако при этом она может позволять разрабатывать программное обеспечение и под другие ОС. Например, одна из популярных систем программирования на языке C/C++ фирмы Watcom для OS/2 позволяет создавать программы и для самой OS/2, и для DOS, и для Windows.

В том случае, когда создаваемые программы должны работать совсем на другой аппаратной базе, говорят о кросс-системах. Так, для ПК на базе микропроцессоров семейства i80x86 имеется большое количество кросс-систем, позволяющих создавать программное обеспечение для различных микропроцессоров и микроконтроллеров.

4 Наконец, под *утилитами* понимают специальные системные программы, с помощью которых можно как обслуживать саму операционную систему, так и под-

[1] *Сервис* (service)–обслуживание, выполнение соответствующего запроса

готавливать для работы носители данных, выполнять перекодирование данных, осуществлять оптимизацию размещения данных на носителе и производить некоторые другие работы, связанные с обслуживанием вычислительной системы. К утилитам следует отнести и программу разбиения накопителя на магнитных дисках на разделы, и программу форматирования, и программу переноса основных системных файлов самой ОС. К утилитам относятся также и известные комплексы программ фирмы Symantec, носящие имя Питера Нортона (создателя этой фирмы и соавтора популярного набора утилит для первых IBM PC). Естественно, что утилиты могут работать только в соответствующей операционной среде.

ГЛАВА 1 Основные понятия

Понятие операционной среды

Операционная система выполняет функции управления вычислительными процессами в вычислительной системе, распределяет ресурсы вычислительной системы между различными вычислительными процессами и образует программную среду, в которой выполняются прикладные программы пользователей. Такая среда называется операционной.

Любая программа имеет дело с некоторыми исходными данными, которые она обрабатывает, и порождает в конечном итоге некоторые выходные данные, результаты вычислений. Очевидно, что в абсолютном большинстве случаев исходные данные попадают в оперативную память (с которой непосредственно работает процессор, выполняя вычисления по программе) с внешних (периферийных) устройств. Аналогично и результаты вычислений, в конце концов, должны быть выведены на внешние устройства. Следует заметить, что программирование операций ввода/вывода относится, пожалуй, к наиболее сложным и трудоёмким задачам. Дело в том, что при создании таких программ без использования современных систем программирования, как говорится, «по старинке», нужно знать не только архитектуру процессора (его состав, назначение основных регистров, систему команд процессора, форматы данных и т. п.), но и архитектуру подсистемы ввода/вывода (соответствующие интерфейсы, протоколы обмена данными, алгоритм работы кон-

троллера устройства ввода/вывода и т. д.). Именно поэтому развитие системного программирования и самого системного программного обеспечения пошло по пути выделения наиболее часто встречающихся операций и создания для них соответствующих программных модулей, которые можно в дальнейшем использовать в большинстве вновь создаваемых программ.

Например, в далекие пятидесятые годы, на заре развития вычислительных систем, при разработке первых систем программирования прежде всего создавали программные модули для подсистемы ввода/вывода, а уже затем – вычисления часто встречающихся математических операций и функций. Благодаря этому при создании прикладных программ программисты могли просто обращаться к соответствующим функциям ввода/вывода и иным функциям и процедурам, что избавляло их от необходимости каждый раз создавать все программные компоненты «с нуля» и от необходимости знать во всех подробностях особенности работы контроллеров ввода/вывода и соответствующих интерфейсов.

Следующий шаг в автоматизации создания готовых к выполнению машинных двоичных программ заключался в том, что транслятор с алгоритмического языка более высокого уровня, нежели первые ассемблеры, уже сам мог подставить вместо высокоуровневого оператора типа READ или WRITE все необходимые вызовы к готовым библиотечным программным модулям. Состав и количество библиотек систем программирования постоянно увеличивались. В конечном итоге возникла ситуация, когда при создании двоичных машинных программ программисты могут вообще не знать многих деталей управления конкретными ресурсами вычислительной системы, а должны только обращаться к некоторой программной подсистеме с соответствующими вызовами и получать от неё необходимые функции и сервисы. Эта программная подсистема и есть операционная система (ОС), а набор её функций, сервисов и правила обращения к ним как раз и образуют то базовое понятие, которое мы называем операционной средой. Таким образом, можно сказать, что термин операционная среда означает, прежде всего, соответствующие интерфейсы, необходимые программам и пользователям для обращения к ОС с целью получить определенные сервисы.

Можно спросить: а чем отличаются *системные программные модули,* реализующие основные системные функции, от тех программных модулей, что пишутся прикладными программистами? Ответ простой: тем, что эти модули, как правило, используются всеми прикладными программами. Поэтому нет особого смысла на этапе создания машинной двоичной программы (которую и исполняет процессор) присоединять соответствующие системные программные модули к телу программы. Выгоднее просто обращаться к этим программным модулям, указывая их адреса и передавая им необходимые параметры, поскольку они уже и так находятся в основной памяти, ибо нужны всем. Другими словами, эти основные системные программные модули входят в состав самой ОС.

Параллельное существование терминов «операционная система» и «операционная среда» вызвано тем, что ОС в общем случае может поддерживать несколько операционных сред. Например, операционная система OS/2 Warp может выполнять следующие программы:

♦ так называемые *«нативные*[1]*»* программы, созданные с учётом соответствующего «родного» 32-битового программного интерфейса этой ОС;

♦ 16-битовые программы, созданные для систем OS/2 первого поколения;

♦ 16-битовые приложения, разработанные для выполнения в операционной среде MS-DOS или PC DOS;

♦ 16-битовые приложения, созданные для операционной среды Windows 3.x;

♦ саму операционную оболочку Windows 3.x и уже в ней – созданные для неё программы.

Операционная среда может включать несколько интерфейсов: пользовательские и программные. Если говорить о пользовательских, то, например, система Linux имеет для пользователя как интерфейсы командной строки (можно использовать различные «оболочки» – shell), интерфейс наподобие Norton Commander – Midnight Commander, так и графические интерфейсы – X-Window с различными менеджерами окон – KDE, Gnome и т. д. Если же говорить о программных интерфейсах, то в той же ОС Linux программы могут обращаться как к операционной

[1] *Native* – родной.

системе за соответствующими сервисами и функциями, так и к графической подсистеме (если она используется). С точки зрения архитектуры процессора (и всего ПК в целом) двоичная программа, созданная для работы в среде Linux, использует те же команды и форматы данных, что и программа, созданная для работы в среде Windows NT. Однако в первом случае мы имеем обращение к одной операционной среде, а во втором – к другой. И программа, созданная для Windows непосредственно, не будет выполняться в Linux; однако если в ОС Linux организовать полноценную операционную среду Windows, то наша Windows-программа сможет быть выполнена. Можно сказать, что операционная среда – это то системное программное окружение, в котором могут выполняться программы, созданные по правилам работы этой среды.

Понятия вычислительного процесса и ресурса

Понятие *«вычислительный процесс»* (или просто – *«процесс»*) является одним из основных при рассмотрении операционных систем. Как понятие процесс является определенным видом абстракции, и мы будем придерживаться следующего неформального определения, приведенного в работе [37]. Последовательный процесс (иногда называемый *«задачей»*[1]) – это выполнение отдельной программы с её данными на последовательном процессоре. Концептуально процессор рассматривается в двух аспектах: во-первых, он является носителем данных и, во-вторых, он (одновременно) выполняет операции, связанные с их обработкой.

В качестве примеров можно назвать следующие процессы (задачи): прикладные программы пользователей, утилиты и другие системные обрабатывающие программы. Процессами могут быть редактирование какого-либо текста, трансляция исходной программы, её компоновка, исполнение. Причем трансляция какой-нибудь исходной программы является одним процессом, а трансляция следующей исходной программы – другим процессом, поскольку, хотя транслятор как объ-

[1]В концепции, которая получила наибольшее распространение в 70-е годы, *задача (task)* – это совокупность связанных между собой и образующих единое целое программных модулей и данных, требующая ресурсов вычислительной системы для своей реализации. В последующие годы задачей стали называть единицу работы, для выполнения которой предоставляется центральный процессор. Вычислительный процесс может включать в себя несколько задач.

единение программных модулей здесь выступает как одна и та же программа, но данные, которые он обрабатывает, являются разными.

Определение концепции процесса преследует цель выработать механизмы распределения и управления ресурсами. Понятие ресурса, так же как и понятие процесса, является, пожалуй, основным при рассмотрении операционных систем. 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. 1.1).

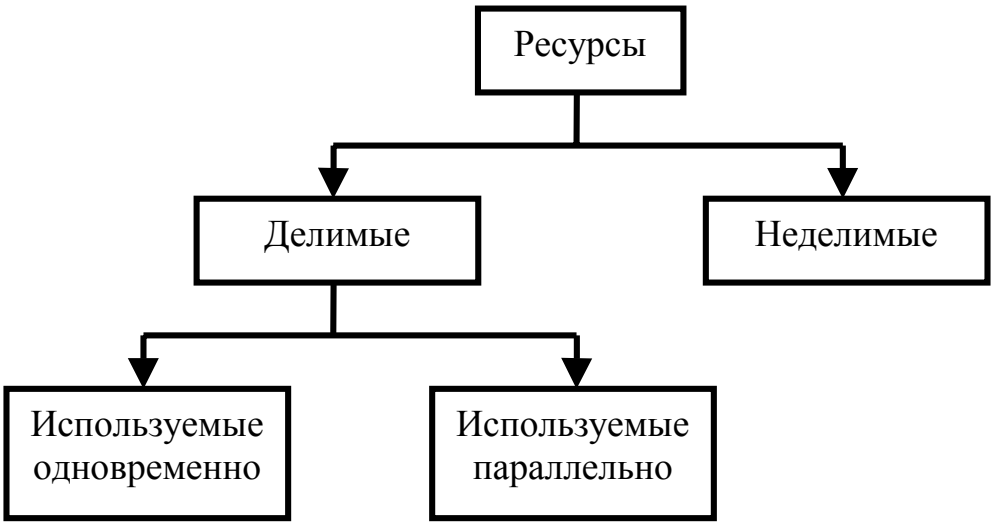

Рис. 1.1. Классификация ресурсов

При разработке первых систем ресурсами считались процессорное время, память, каналы ввода/вывода и периферийные устройства [49, 89]. 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 кото-

рым необходимо соответствующим образом контролировать. В настоящее время понятие ресурса превратилось в абстрактную структуру с целым рядом атрибутов, характеризующих способы доступа к этой структуре и её физическое представление в системе. Более того, помимо системных ресурсов, о которых мы сейчас говорили, как ресурс стали толковать и такие объекты, как сообщения и синхросигналы, которыми обмениваются задачи.

В первых вычислительных системах любая программа могла выполняться только после полного завершения предыдущей. Поскольку эти первые вычислительные системы были построены в соответствии с принципами, изложенными в известной работе Яноша Джон фон Неймана, 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 очереднойоперации 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, как мы теперь говорим) ожидает завершения очереднойоперации ввода/вывода, другая программа (а точнее, другая задача) может быть поставлена на решение (рис. 1.2).

Из рис. 1.2, на котором в качестве примера изображена такая гипотетическая ситуация, видно, что благодаря совмещению во времени выполнения двух программ общее время выполнения двух задач получается меньше, чем если бы мы выполняли их по очерёди (запуск одной только после полного завершения другой). Из этого же рисунка 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 счёт дополнительных затрат времени на ожидание освобождения ресурса).

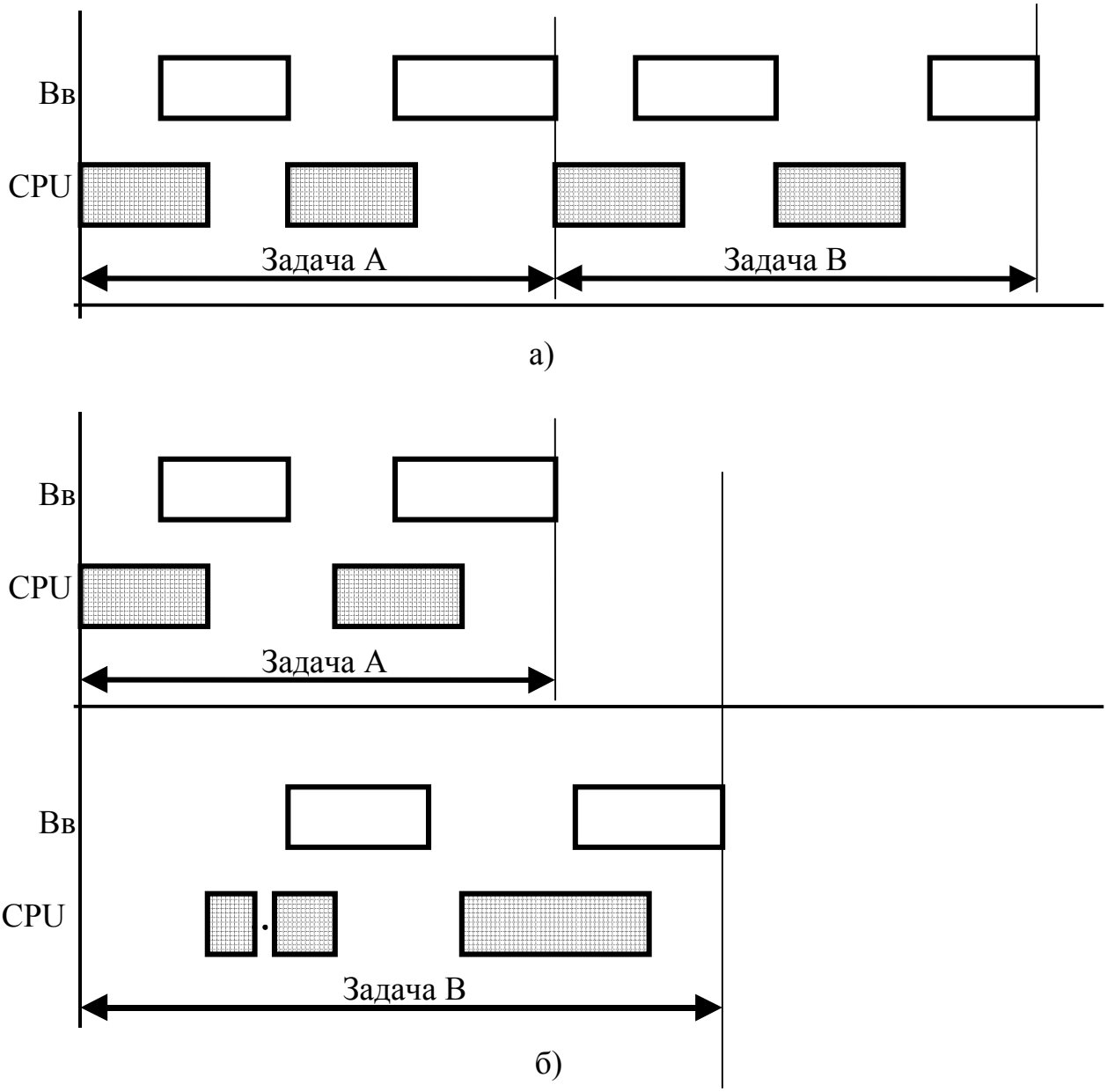

Рис. 1.2. Пример выполнения двух программ: а – однопрограммный режим; б – мультипрограммный режим

Как мы уже отмечали, операционная система поддерживает *мультипрограммирование* (многопроцессность) и старается эффективно использовать ресурсы путём организации к ним очерёдей запросов, составляемых тем или иным способом.

Это требование достигается поддерживанием в памяти более одного процесса, ожидающего процессор, и более одного процесса, готового использовать другие ресурсы, как только последние станут доступными. Общая схема выделения ресурсов такова. При необходимости использовать какой-либо ресурс (оперативную память, устройство ввода/вывода, массив данных и т. п.) задача обращается к супервизору операционной системы – её центральному управляющему модулю, который может состоять из нескольких модулей, например: супервизор ввода/вывода, супервизор прерываний, супервизор программ, диспетчер задач и т. д. – посредством специальных вызовов (команд, директив) и сообщает о своём требовании. При этом указывается вид ресурса и, если надо, его объём (например, количество адресуемых ячеек оперативной памяти, количество дорожек или секторов на системном диске, устройство печати и объём выводимых данных и т. п.).

Директива обращения к операционной системе передаёт ей управление, переводя процессор в привилегированный режим работы (см. раздел «Прерывания», глава 1), если такой существует. Не все вычислительные комплексы имеют два (и более) режима работы: привилегированный (режим супервизора), пользовательский, режим эмуляции какого-нибудь другого компьютера и т. д.

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

♦ 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 её в состояние ожидания (блокируя). очередь к ресурсу может быть организована несколькими способами, но чаще всего это осуществляется с помощью списковой структуры.

После окончания работы с ресурсом задача опять с помощью специального вызова супервизора (посредством соответствующей директивы) сообщает операционной системе об отказе от ресурса, или операционная система забирает ресурс сама, если управление возвращается супервизору после выполнения какой-либо системной функции. Супервизор операционной системы, получив управление по этому обращению, освобождает ресурс и проверяет, имеется ли очередь к освободившемуся ресурсу. Если очередь есть – в зависимости от принятой *дисциплины обслуживания* (правила обслуживания)[1] и приоритетов заявок он выводит из состояния ожидания задачу, ждущую ресурс, и переводит её в состояние готовности к выполнению. После этого управление либо передаётся данной задаче, либо возвращается 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 [37].

Диаграмма состояний процесса

Необходимо различать системные управляющие процессы, представляющие работу супервизора операционной системы и занимающиеся распределением и управлением ресурсов, от всех других процессов: системных обрабатывающих процессов, которые не входят в ядро операционной системы, и процессов пользо-

[1] Например, дисциплина «последний пришедший обслуживается первым» определяет обслуживание в порядке, обратном очерёдности поступления соответствующих запросов

вателя. Для системных управляющих процессов в большинстве операционных систем ресурсы распределяются изначально и однозначно. Эти процессы управляют ресурсами системы, за использование которых существует конкуренция между всеми остальными процессами. Поэтому исполнение системных управляющих программ не принято называть процессами. *Термин задача можно употреблять только по отношению к процессам пользователей и к системным обрабатывающим процессам*. Однако это справедливо не для всех ОС. Например, в так называемых «микроядерных» (см. главу 5 «Архитектура операционных систем и интерфейсы прикладного программирования») ОС (в качестве примера можно привести ОС реального времени QNX фирмы Quantum Software systems) большинство управляющих программных модулей самой ОС 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 можно сказать, что процесс может находиться в активном и пассивном (не активном) состоянии. 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:

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

♦ *готовности к выполнению* – ресурсы могут быть предоставлены, тогда процесс перейдёт 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. ОС организует (порождает или выделяет) для нового процесса соответствующий дескриптор (см. об этом дальше) процесса, и процесс (задача) начинает развиваться (выполняться). Поэтому пассивного состояния не существует. В ОС реального времени (ОСРВ) ситуация иная. Обычно при проектировании системы реального времени уже заранее бывает известен состав программ (задач), которые должны будут выполняться. Известны и многие их параметры, которые необходимо учитывать при распределении ресурсов (например, объём 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 мы и отобразили на рис. 1.3, отделив это состояние от остальных состояний пунктиром.

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, взаимодействием 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. 1.3. Рассмотрим эти переходы из одного состояния в другое более подробно.

Процесс из состояния бездействия может перейти в состояние готовности в следующих случаях:

♦ по команде оператора (пользователя). Имеет место в тех диалоговых операционных системах, где программа может иметь статус задачи (и при этом являться

пассивной), а не просто быть исполняемым файлом и только на время исполнения получать статус задачи (как это происходит в большинстве современных ОС для ПК);

♦ при выборе из очерёди планировщиком (характерно для операционных систем, работающих в пакетном режиме);

♦ по вызову из другой задачи (посредством обращения к супервизору один процесс может создать, инициировать, приостановить, остановить, уничтожить другой процесс);

♦ по прерыванию от внешнего инициативного[1] устройства (сигнал о свершении некоторого события может запускать соответствующую задачу);

♦ при наступлении запланированного времени запуска программы.

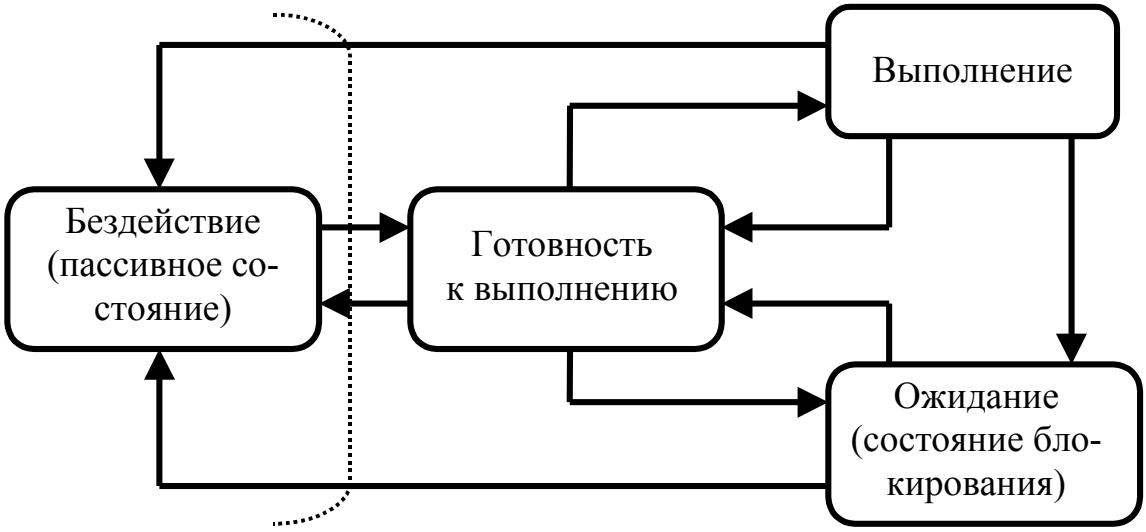

Рис. 1.3. Граф состояний процесса

Последние два способа запуска задачи, при которых процесс из состояния бездействия переходит в состояние готовности, характерны для операционных систем реального времени.

Процесс, который может исполняться, как только ему будет предоставлен процессор, а для диск-резидентных задач в некоторых системах – и оперативная па-

[1] Устройство называется «*инициативным*», если по сигналу запроса на прерывание от него должна запускаться некоторая задача

мять, 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:

♦ процесс завершается, при этом он посредством обращения к супервизору передаёт управление операционной системе и сообщает о своем завершении. В результате этих действий супервизор либо переводит его в список бездействующих процессов (процесс переходит в пассивное состояние), либо уничтожает (уничтожается, естественно, не сама программа, а именно задача, которая соответствовала исполнению некоторой программы). В состояние бездействия процесс может быть переведен принудительно: по команде оператора (действие этой и других команд оператора реализуется системным процессом, который «транслирует» команду в запрос к супервизору с требованием перевести указанный процесс в состояние бездействия), 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 (причиной перевода в состояние ожидания может быть и отсутствие сегмента или страницы в случае организации механизмов виртуальной памяти, см. раздел «Сегментная, страничная и сегментно-страничная организация памяти» в главе 2), 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. Один из основных видов событий – это прерывания.

Реализация понятия последовательного процесса в ОС

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:

♦ 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.);

♦ защищённую 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 её исходном состоянии и адрес на диске, куда она выгружается из оперативной памяти, если её вытесняет другая (для *диск-резидентных* задач, которые постоянно

находятся во внешней памяти на системном магнитном диске и загружаются в оперативную память только на время выполнения).

Описатели 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 (за исключением состояния выполнения для однопроцессорной системы) операционная система ведет соответствующий список задач, находящихся в этом состоянии. Однако для состояния ожидания может быть не один список, а столько, сколько различных видов ресурсов могут вызывать состояние ожидания. Например, состояний ожидания завершения операции ввода/вывода может быть столько, сколько устройств ввода/вывода имеется в системе.

В некоторых операционных системах количество описателей определяется жестко и заранее (на этапе генерации варианта операционной системы или в конфигурационном файле, который используется при загрузке ОС), в других – по мере необходимости система может выделять участки памяти под новые описатели. Например, в OS/2 максимально возможное количество описателей задач определяется в конфигурационном файле CONFIG.SYS, а в Windows NT оно в явном виде не задается. Справедливости ради стоит заметить, что в упомянутом файле указывается количество не процессов, а именно задач, и под задачей в данном случае понимается как процесс, так и поток этого же процесса, называемый потоком или тредом (см. следующий раздел). Например, строка в файле CONFIG.SYS

THREADS=1024

указывает» что всего в системе может параллельно существовать и выполняться до 1024 задач, включая вычислительные процессы и их потоки.

В ОС реального времени 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 (что сейчас встречается редко, а в недалеком прошлом достаточно часто в качестве вычислительных систем общего назначения

приобретали мини-ЭВМ и устанавливали на них ОС реального времени) 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 (с целью ускорить работу диспетчера), 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. 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 80x86 (см. главу 3 «Особенности архитектуры микропроцессоров 180x86 для организации мультипрограммных операционных систем»), начиная с 80286, имеется специальный регистр TR (task register), указывающий местонахождение TSS (сегмента состояния задачи[1], см. раздел «Новые системные регистры микропроцессоров i80x86», глава 3), в котором при переключении с задачи на задачу автоматически сохраняется содержимое регистров процессора [2, 22, 84]. Как правило, в современных ОС для этих микропроцессоров дескриптор задачи включает в себя TSS. Другими словами, дескриптор задачи больше по размеру, чем TSS, и включает в себя такие традиционные поля, как идентификатор задачи, её имя, тип, приоритет и т. п.

[1] *TSS* (task state segment) – сегмент состояния задачи.

Процессы и треды

Понятие процесса было введено для реализации идей мультипрограммирования. Напомним, в свое время различали термины «мультизадачность» и «мультипрограммирование». Таким образом, для реализации «мультизадачности» в её исходном толковании необходимо было тоже ввести соответствующую сущность. Такой сущностью и стали так называемые «легковесные» процессы, или, как их теперь преимущественно называют, – *потоки* или *треды* (нити)[2]. Рассмотрим эти понятия подробнее.

Когда говорят о *процессах* (process), то тем самым хотят отметить, что операционная система поддерживает их обособленность: у каждого процесса имеется свое виртуальное адресное пространство, каждому процессу назначаются свои ресурсы – файлы, окна, семафоры 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. Другими словами, в случае процессов ОС считает их совершенно несвязанными и независимыми. При этом именно ОС берет на себя роль арбитра в конкуренции между процессами по поводу ресурсов.

Однако желательно иметь ещё 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 та-

[2] *Thread* – поток, нить.

кие длительные операции, оформлять в виде самостоятельных «подпроцессов» (легковесных или облегченных процессов – потоков, можно также воспользоваться термином *задача),* которые будут выполняться параллельно с другими «подпроцессами» (потоками, задачами), то у пользователя появляется возможность параллельно выполнять несколько операций в рамках одного приложения (процесса). Легковесными эти задачи называют потому, что операционная система не должна для них организовывать 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, – это процессорный ресурс. В однопроцессорной системе треды (задачи) 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. Параллельные вычисления (а, следовательно, и более эффективное использование ресурсов центрального процессора, и меньшее суммарное время выполнения задач) теперь уже часто реализуется на уровне тредов, и программа, оформленная в виде нескольких тредов в рамках одного процесса, может быть выполнена быстрее за счёт параллельного выполнения её отдельных частей. Например, если электронная таблица или текстовый процессор были разработаны с учётом возможностей многопоточной обработки, то пользователь может запросить пересчёт своего рабочего листа или слияние нескольких документов и одновременно продолжать заполнять таблицу или открывать для редактирования следующий документ.

Особенно эффективно можно использовать многопоточность для выполнения распределённых приложений; например, многопоточный сервер может параллельно выполнять запросы сразу нескольких клиентов. Как известно, операционная система OS/2 одной из первых среди ОС, используемых на ПК, ввела многопоточность. В середине девяностых годов для этой ОС было создано очень большое ко-

личество приложений, в которых использование механизмов многопоточной обработки реально приводило к существенно большей скорости выполнения вычислений.

Итак, сущность «поток» была введена для того, чтобы именно с помощью этих единиц распределять процессорное время между возможными работами. Сущность «процесс» предполагает, что при диспетчеризации нужно учитывать все ресурсы, закрепленные за ним. А при манипулировании тредами можно менять только контекст задачи, если мы переключаемся с одной задачи на другую в рамках одного процесса. Все остальные вычислительные ресурсы при этом не затрагиваются. Каждый процесс всегда состоит, по крайней мере, из одного потока, и только если имеется внутренний параллелизм, программист может «расщепить» один тред на несколько параллельных.

Потребность в потоках (threads) возникла ещё на однопроцессорных вычислительных системах, поскольку они позволяют организовать вычисления более эффективно. Для использования достоинств многопроцессорных систем с общей памятью треды уже просто необходимы, так как позволяют не только реально ускорить выполнение тех задач, которые допускают их естественное распараллеливание, но и загрузить процессорные элементы работой, чтобы они не простаивали. Заметим, однако, что желательно, чтобы можно было уменьшить взаимодействие тредов между собой, ибо ускорение от одновременного выполнения параллельных потоков может быть сведено к минимуму из-за задержек синхронизации и обмена данными.

Каждый тред выполняется строго последовательно и имеет свой собственный программный счётчик и стек. Треды, как и процессы, могут порождать треды-потомки, поскольку любой процесс состоит, по крайней мере, из одного треда. Подобно традиционным процессам (то есть процессам, состоящим из одного треда), каждый тред может находится в одном из активных состояний. Пока один тред заблокирован (или просто находится в очерёди готовых к исполнению задач), другой тред того же процесса может выполняться. Треды разделяют процессорное время

так же, как это делают обычные процессы, в соответствии с различными вариантами диспетчеризации.

Как мы уже знаем, все треды имеют одно и то же виртуальное адресное пространство своего процесса. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый тред может иметь доступ к каждому виртуальному адресу, один тред может использовать стек другого треда. Между потоками нет полной защиты, так как это, во-первых, невозможно, а во-вторых, не нужно. *Все потоки одного процесса всегда решают общую задачу одного пользователя, и механизм потоков используется здесь для более быстрого решения задачи путем её распараллеливания*. При этом программисту очень важно получить в свое распоряжение удобные средства организации взаимодействия частей одной программы. Повторим, что кроме разделения адресного пространства, все треды разделяют также набор открытых файлов, используют общие устройства, выделенные процессу, имеют одни и те же наборы сигналов, семафоры и т. п. А что у тредов будет их собственным? Собственными являются программный счетчик, стек, рабочие регистры процессора, потоки-потомки, состояние.

Вследствие того, что треды, относящиеся к одному процессу, выполняются в одном и том же виртуальном адресном пространстве, между ними легко организовать тесное взаимодействие, в отличие от процессов, для которых нужны специальные механизмы обмена сообщениями и данными. Более того, программист, создающий многопоточное приложение, может заранее продумать работу множества тредов процесса таким образом, чтобы они могли взаимодействовать наиболее выгодным способом, а не участвовать в конкуренции за предоставление ресурсов тогда, когда этого можно избежать.

Для того чтобы можно было эффективно организовать параллельное выполнение рассмотренных сущностей (процессов и тредов), в архитектуру современных процессоров включена возможность работать со специальной информационной структурой, описывающей ту или иную сущность. Для этого уже на уровне архитектуры микропроцессора используется понятие «задача» (task). Оно как бы объединяет в себе обычный и «легковесный» процессы. Это понятие и поддерживаемая

для него на уровне аппаратуры информационная структура позволяют в дальнейшем при разработке операционной системы построить соответствующие дескрипторы, как для процесса, так и для треда. Отличаться эти дескрипторы будут, прежде всего, тем, что дескриптор треда может хранить только контекст приостановленного вычислительного процесса, тогда как дескриптор процесса (process) должен уже содержать поля, описывающие тем или иным способом ресурсы, выделенные этому процессу. Другими словами, тот же task state segment (сегмент состояния задачи), подробно рассмотренный в разделе «Адресация в 32-разрядных микропроцессорах i80x86 при работе в защищённом режиме» главы 3, используется как основа для дескриптора процесса. Каждый тред (в случае использования так называемой «плоской» модели памяти – см. раздел «Поддержка страничного способа организации виртуальной памяти», глава 3 – может быть оформлен в виде самостоятельного сегмента, что приводит к тому, что простая (не многопоточная) программа будет иметь всего один сегмент кода в виртуальном адресном пространстве.

В завершение можно привести несколько советов по использованию потоков при создании приложений, заимствованных из работы [55].

1 В случае использования однопроцессорной системы множество параллельных потоков часто не ускоряет работу приложения, поскольку в каждый отдельно взятый промежуток времени возможно выполнение только одного потока. Кроме того, чем больше у вас потоков, тем больше нагрузка на систему, потраченная на переключение между ними. Если ваш проект имеет более двух постоянно работающих потоков, то такая мультизадачность не сделает программу быстрее, если каждый из потоков не будет требовать частого ввода/ вывода.

2 Вначале нужно понять, для чего необходим поток. Поток, осуществляющий обработку, может помешать системе быстро реагировать на запросы ввода/ вывода. Потоки позволяют программе отзываться на просьбы пользователя и устройств, но при этом сильно загружать процессор. Потоки позволяют компьютеру одновременно обслуживать множество устройств, и созданный вами поток, отвечающий за обработку специфического устройства, в качестве минимума может потребовать

столько времени, сколько системе необходимо для обработки запросов всех устройств.

3 Потокам можно назначить определенный приоритет для того, чтобы наименее значимые процессы выполнялись в фоновом режиме. Это путь честного разделения ресурсов CPU[1]. Однако необходимо осознать тот факт, что процессор один на всех, а потоков много. Если в вашей программе главная процедура передаёт нечто для обработки в низкоприоритетный поток, то сама программа становится просто неуправляемой.

4 Потоки хорошо работают, когда они независимы. Но они начинают работать непродуктивно, если вынуждены часто синхронизироваться для доступа к общим ресурсам. Блокировка и критические секции отнюдь не увеличивают скорость работы системы, хотя без использования этих механизмов взаимодействующие вычисления организовывать нельзя.

5 Помните, что память виртуальна. Механизм виртуальной памяти (см. раздел «Память и отображения, виртуальное адресное пространство», глава 2) следит за тем, какая часть виртуального адресного пространства должна находиться в оперативной памяти, а какая должна быть сброшена в файл подкачки. Потоки усложняют ситуацию, если они обращаются в одно и то же время к разным адресам виртуального адресного пространства приложения. Это значительно увеличивает нагрузку на систему, особенно при небольшом объёме кэш-памяти. Помните, что реально память не всегда «свободна», как это пишут в информационных «окошках» «О системе». Всегда отождествляйте доступ к памяти с доступом к файлу на диске и создавайте приложение с учётом вышесказанного.

6 Всякий раз, когда какой-либо из ваших потоков пытается воспользоваться общим ресурсом вычислительного процесса, которому он принадлежит, вы обязаны тем или иным образом легализовать и защитить свою деятельность. Хорошим средством для этого являются критические секции, семафоры и очерёди сообщений. Если вы протестировали свое приложение и не обнаружили ошибок синхронизации, то это ещё не значит, что их там нет. Пользователь может создать самые

[1] *CPU* (central processing unit) – центральное обрабатывающее устройство или просто процессор.

непредсказуемые ситуации. Это очень ответственный момент в разработке многопоточных приложений.

7 Не возлагайте на поток несколько функций. Сложные функциональные отношения затрудняют понимание общей структуры приложения, его алгоритм. Чем проще и менее многозначна каждая из рассматриваемых ситуаций, тем больше вероятность, что ошибок удастся избежать.

Прерывания

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 неё – к соответствующей программе обработки прерывания), происходящая при возникновении определенного события.*

Идея прерываний была предложена в середине 50-х годов и 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. 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:

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 i80x86 определяется регистрами CS и IP – указателем коман-

ды [2, 22, 84]), содержимым регистров процессора и может включать также спецификацию режима (например, режим пользовательский или привилегированный) и другую информацию.

3 Управление аппаратно передаётся подпрограмме обработки прерывания. В простейшем случае в счётчик 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 (см. раздел «Система прерываний 32-разрядных микропроцессоров i80x86», глава 3).

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 объёма 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, а шаги 4-7 – программно.

На рис. 1.4 показано, что при возникновении запроса на прерывание естественный ход вычислений нарушается и управление передаётся программе обработки возникшего прерывания. При этом средствами аппаратуры сохраняется (как правило, с помощью механизмов стековой памяти) 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 сохранённого адреса команды.

Однако такая схема используется только в самых простых программных средах. В мультипрограммных операционных системах обработка прерываний происходит по более сложным схемам, о чём будет более подробно написано ниже.

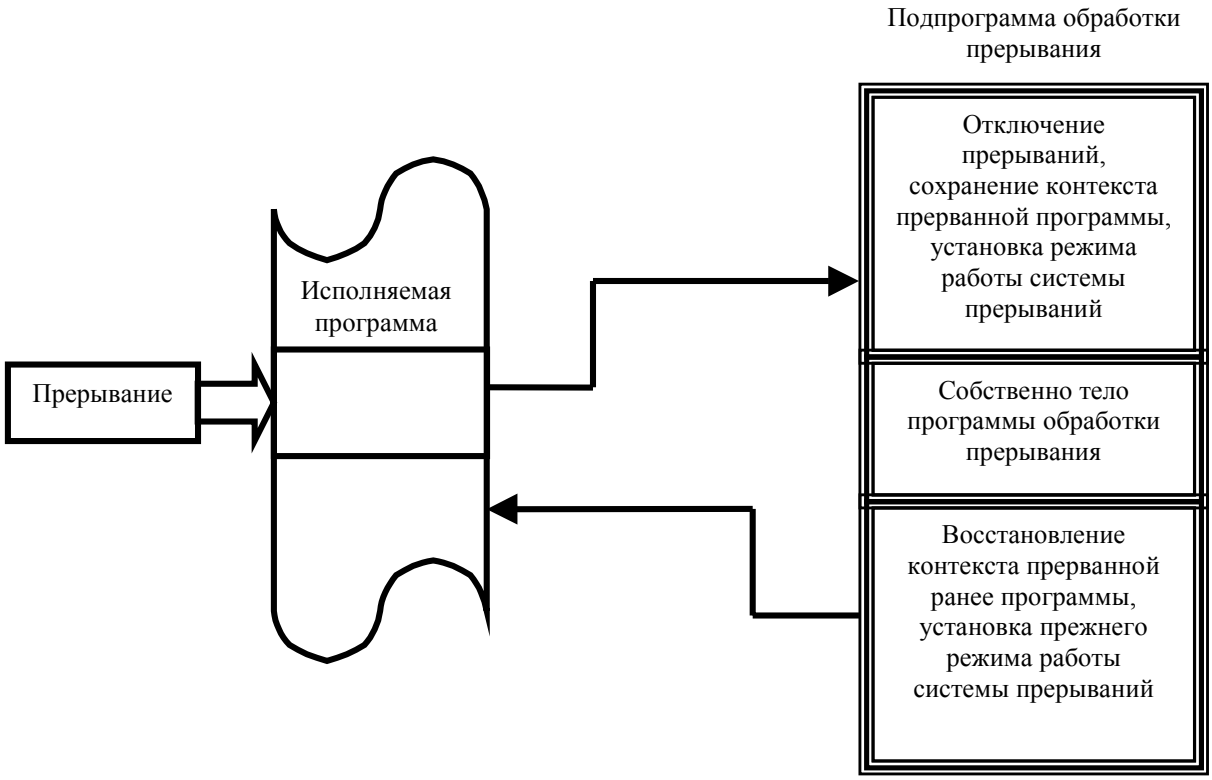

Рис. 1.4. Обработка прерывания

Итак, главные функции механизма прерываний:

- ♦ 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 – system stack.

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 являются синхронными с его операциями. Примерами являются следующие запросы на прерывания:

- ♦ при нарушении адресации (в адресной части выполняемой команды указан запрещённый 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 чётности, ошибок в работе различных устройств аппаратуры средствами контроля.

Могут ещё 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, запрещённую в данном режиме, происходит внутреннее прерывание и управление передаётся 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, то есть 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 происходило 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; 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.1.5 изображен обычный порядок (приоритеты) обработки прерываний в зависимости от типа прерываний. Учет приоритета может быть встроен в технические средства, а также определяться операционной системой, то есть 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: отключение системы прерываний, маскирование (запрет) отдельных сигналов прерывания. Программное управление этими средствами (существуют специальные команда 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 могут быть обработаны только последовательно. Чтобы обработать сигналы прерывания в разумном порядке им (как уже отмечалось) присваиваются приоритеты. Сигнал с более высоким приоритетом обрабатывается в первую очередь, обработка остальных сигналов прерывания откладывается.

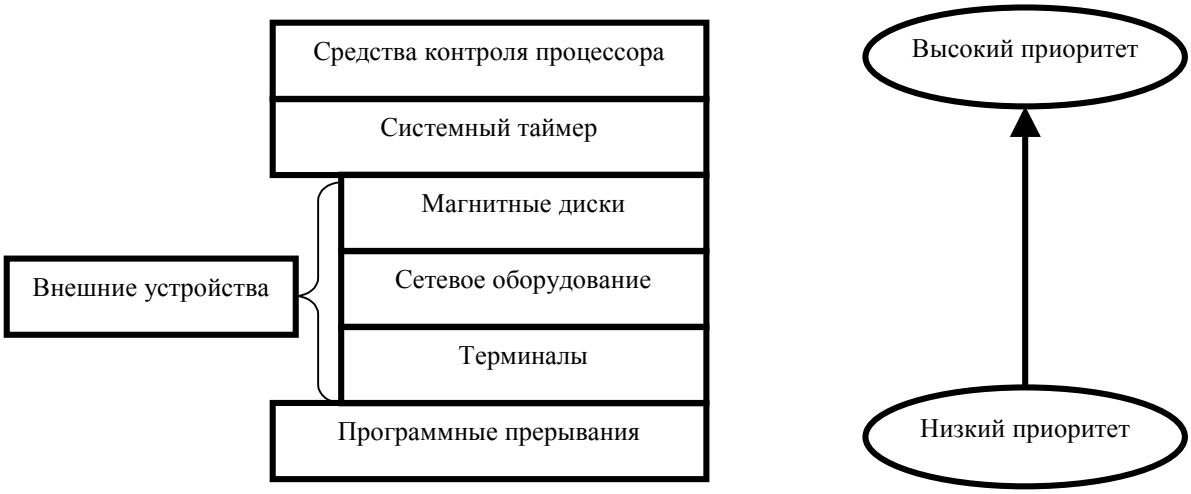

Рис. 1.5. Распределение прерываний по уровням приоритета

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

♦ с *относительными приоритетами,* то есть 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,* то есть 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, то есть 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 пришёл – первым обслужен), то есть запросы с более низким приоритетом могут прерывать обработку прерывания с более высоким приоритетом. Дли 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, в организации обмена информацией (данными и программами), предоставлении необходимых ресурсов, в динамике выполнения задачи и в организации сервиса. Причины прерываний определяет ОС (модуль, который называют супервизором прерываний), она же и 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 передаётся на соответствующую подпрограмму обработки. Из рис.1.4 видно, что в подпрограмме обработки прерывания имеются две служебные секции. Это – первая секция, в которой 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 прерыва-

ний 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 передаётся супервизору, на этот раз уже на тот модуль, который занимается диспетчеризацией задач (см. раздел «Качество диспетчеризации и гарантии обслуживания», глава 2). 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 нами схема проиллюстрирована на рис. 1.6.

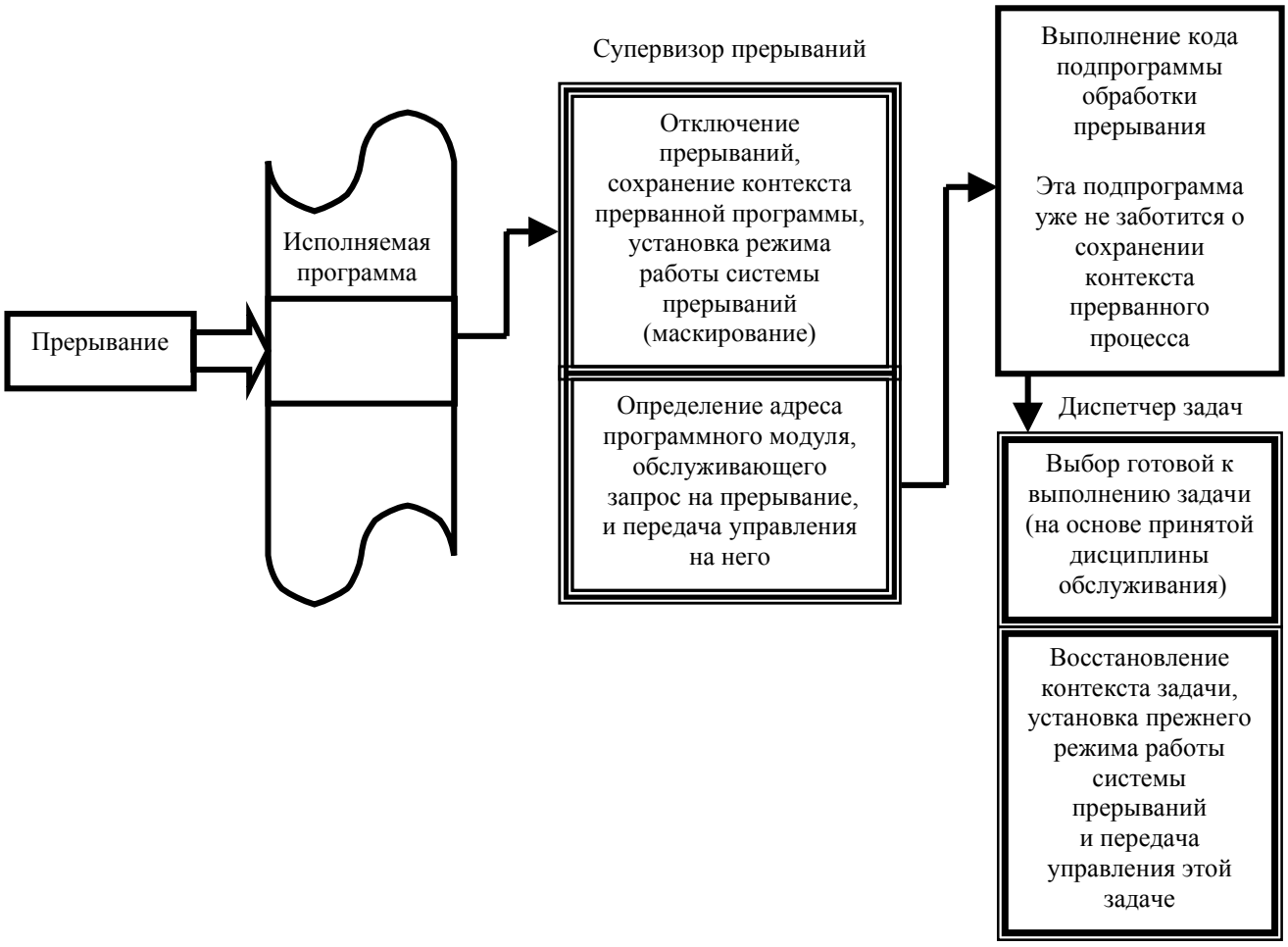

Рис. 1.6. Обработка прерывания при участии супервизоров ОС

Как мы видим из рис. 1.6, здесь нет непосредственного возврата в прерванную ранее программу непосредственно из самой подпрограммы обработки прерывания. Для прямого непосредственного возврата достаточно адрес возврата сохранить в стеке, 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 описанных выше дескрипторах задач, то у нас не было бы возможности гибко подходить к выбору той задачи, которой нужно передать процессор после завершения работы подпрограммы обработки прерывания. Естественно, что это только общий принцип. В конкретных процессорах и в конкретных ОС могут существовать некоторые отступления от рассмотренной схемы и/или дополнения к ней. Например, в современных процессорах часто имеются специальные аппаратные возможности для сохранения контекста прерываемого

процесса непосредственно в его дескрипторе, то есть дескриптор процесса (по крайней мере, его часть) становится структурой данных, которую поддерживает аппаратура.

Для полного понимания принципов создания и механизмов реализации рассматриваемых далее современных ОС необходимо знать архитектуру персональных компьютеров и, в частности, особенности системы прерывания. Этот вопрос более подробно рассмотрен в главе 4 «Управление вводом/выводом и файловые системы», посвящённом архитектуре микропроцессоров i80x86.

Основные виды ресурсов

Рассмотрим кратко основные виды ресурсов вычислительной системы и способы их разделения (см. рис. 1.1). Прежде всего, одним из важнейших ресурсов является сам процессор[1], точнее – процессорное время. Процессорное время делится попеременно (параллельно). Имеется множество методов разделения этого ресурса (см. раздел «Планирование и диспетчеризация процессов и задач», глава 2).

Вторым видом ресурсов вычислительной системы можно считать память. Оперативная память может быть разделена и одновременным способом (то есть в памяти одновременно может располагаться несколько процессов или, по крайней мере, текущие фрагменты, участвующие в вычислениях), и попеременно (в разные моменты оперативная память может предоставляться для разных вычислительных процессов). Память – очень интересный вид ресурса. Дело в том, что в каждый конкретный момент времени процессор при выполнении вычислений обращается к очень ограниченному числу ячеек оперативной памяти. С этой точки зрения желательно память разделять для возможно большего числа параллельно исполняемых процессов. С другой стороны, как правило, чем больше оперативной памяти может быть выделено для конкретного текущего процесса, тем лучше будут условия для его выполнения. Поэтому проблема эффективного разделения оперативной памяти между параллельно выполняемыми вычислительными процессами является одной

[1] Разговор о процессоре как об одном из ресурсов более характерен для мультипроцессорных систем. В случае однопроцессорных систем чаще говорят о процессорном времени.

из самых актуальных. Достаточно подробно вопросы распределения памяти между параллельно выполняющимися процессами рассмотрены в главе 2 «Управление задачами и памятью в операционных системах».

Когда говорят о внешней памяти (например, память на магнитных дисках), то собственно память и доступ[2] 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. Простыми и наглядными примерами внешних устройств, которые не могут быть разделяемыми, являются принтер и накопитель на магнитной ленте. Действительно, если допустить, что принтер можно разделять между двумя процессами, которые смогут его использовать попеременно, то результаты печати, скорее всего, не смогут быть использованы – фрагменты выведенного текста могут перемешаться таким образом, что в них невозможно будет разобраться. Аналогично обстоит дело и с накопителем на магнитной ленте. Если один процесс начнет что-то читать или писать, а второй при этом запросит перемотку ленты на её начало, то оба вычислительных процесса не смогут выполнить свои вычисления.

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

Как известно, 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: либо повреждается часть кода, либо – исходные данные, от которых зависит ход вы-

[2] Процесс обращения к данным

числений. Очевидно, что однократно используемые, программные модули являются неделимым ресурсом. Более того, их обычно вообще не распределяют как ресурс системы. Системные однократно используемые программные модули, как правило, используются 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 и 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, то есть при отключенной системе прерываний (часто говорят, что прерывания закрыты), 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). С позиций стороннего наблюдателя по отношению к вычислительным процессам, которые попеременно (причем, возможно, неоднократно) в течение срока своей «жизни» вызывают некоторый привилегированный программный модуль, такой модуль будет выступать как попеременно разделяемый ресурс. Структура привилегированных программных модулей изображена на рис. 1.7. Здесь в первой секции программного модуля выключается система прерываний. В последней секции, напротив, включается система прерываний.

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 выполнения такого модуля, исполняемого в рамках одного вычислительного процесса, запустить его ещё раз по требованию другого вычислительного процесса, то промежуточные результаты для прерванных вычислений могут быть потеряны. В противоположность этому, *реентерабельные программные модули* (reenterable[1])

[1] *Reenterable* – (дословно) допускающий повторные обращения (к модулю).

допускают повторное многократное прерывание своего исполнения и повторный их запуск по обращению из других задач (вычислительных процессов). 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. Это может быть реализовано двумя способами: с помощью статических и динамических методов выделения памяти под сохраняемые значения. Основной, наиболее часто используемый динамический – способ выделения памяти для сохранения всех промежуточных результатов вычисления, относящихся к реентерабельному программному модулю, может быть проиллюстрирован с помощью рис. 1.8.

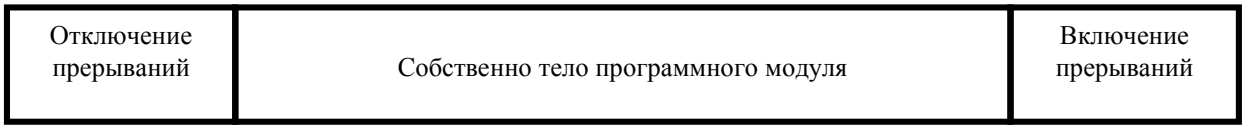

Рис. 1.7. Структура привилегированного программного модуля

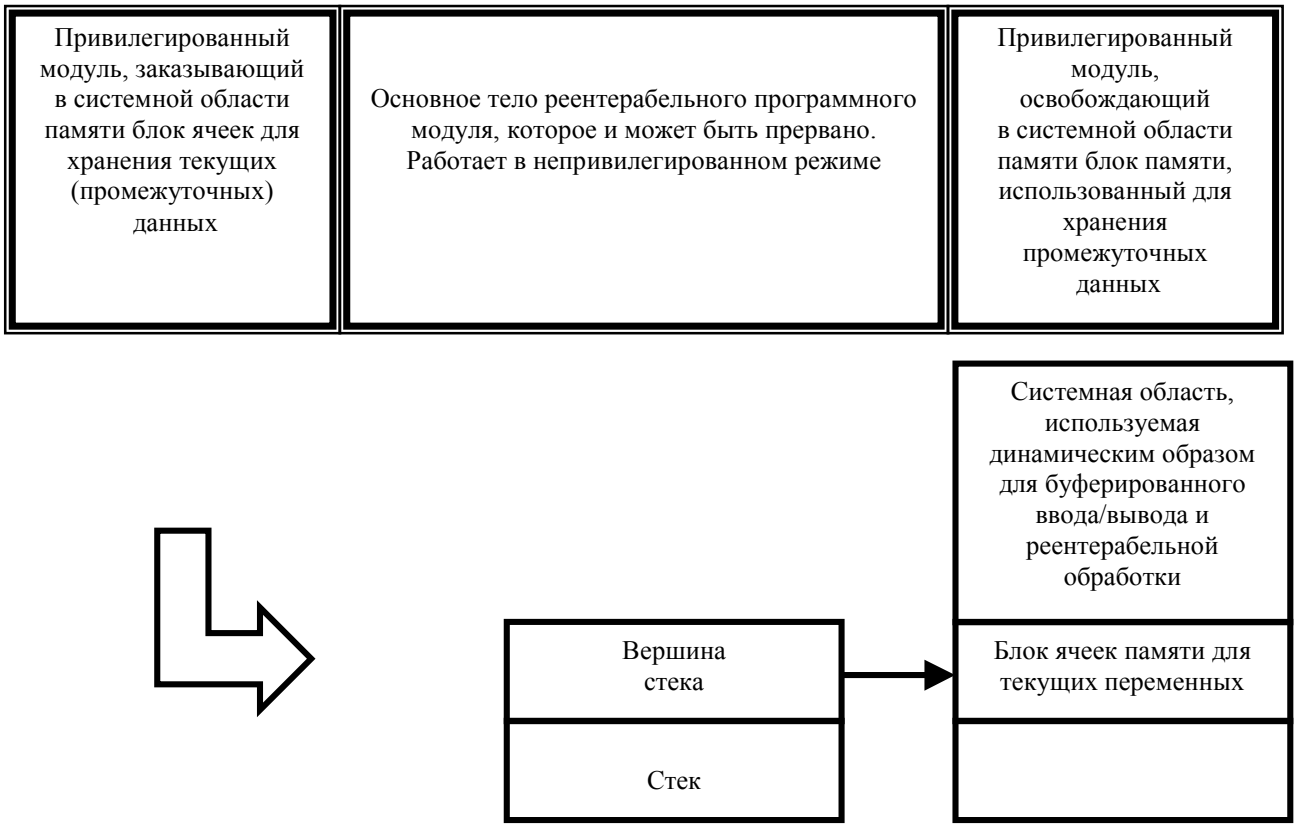

Рис. 1.8. Реентерабельный программный модуль

Основная идея построения и работы реентерабельного программного модуля, структура которого представлена на рис. 1.8, 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 её объём. Все текущие переменные реентерабельного 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 её 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.

Что касается статического способа выделения памяти, то здесь речь может идти, например, о том, что заранее для фиксированного числа вычислительных процессов резервируются области памяти, в которых будут располагаться переменные реентерабельных программных модулей: для каждого процесса – своя область памяти. Чаще всего в качестве таких процессов выступают процессы ввода/вывода и речь идёт о реентерабельных драйверах (реентерабельный драйвер может управлять параллельно несколькими однотипными устройствами. См. более подробно в главе 4 «Управление вводом/выводом и файловые системы»).

Кроме реентерабельных программных модулей существуют ещё *повторно входимые* (от re-entrance). Этим термином называют программные модули, которые

тоже допускают свое многократное параллельное использование, но в отличие от реентерабельных их нельзя прерывать. Повторно входимые программные модули состоят из привилегированных секций и повторное обращение к ним возможно только после завершения какой-нибудь из таких секций. После выполнения очередной привилегированной секции управление может быть передано супервизору, и если он предоставит возможность выполняться другому процессу, то возможно повторное вхождение в рассматриваемый программный модуль. Другими словами, в повторно входимых программных модулях четко предопределены все допустимые (возможные) точки входа. Следует отметить, что повторно входимые программные модули встречаются гораздо чаще реентерабельных (повторно прерываемых).

Наконец, имеются и информационные ресурсы, то есть в качестве ресурсов могут выступать данные.

Информационные ресурсы 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. Если же процессы могут изменять информационные ресурсы, то необходимо специальным образом организовывать работу с такими данными. Это одна из наиболее сложных проблем, достаточно подробно она обсуждается в главе 6.

Классификация операционных систем

Широко известно высказывание, согласно которому любая наука начинается с классификации. Само собой, что вариантов классификации может быть очень много, все будет зависеть от выбранного признака, по которому один объект мы будем отличать от другого. Однако, что касается ОС, здесь уже давно сформировалось относительно небольшое количество классификаций: по назначению, по режиму обработки задач, по способу взаимодействия с системой и, наконец, по способам построения (архитектурным особенностям систем).

Во введении мы уже дали «определение» операционной системы (ОС). Поэтому просто повторим, что основным предназначением ОС является организация эф-

фективных и надёжных вычислений, создание различных интерфейсов для взаимодействия с этими вычислениями и с самой вычислительной системой.

Прежде всего, различают *ОС общего и специального назначения*. ОС специального назначения, в свою очередь, подразделяются на следующие: для переносимых микрокомпьютеров и различных встроенных систем, организации и ведения баз данных, решения задач реального времени и т. п.

По режиму обработки задач различают ОС, обеспечивающие однопрограммный и мультипрограммный режимы. Под мультипрограммированием понимается способ организации вычислений, когда на однопроцессорной вычислительной системе создается видимость одновременного выполнения нескольких программ. Любая задержка в решении программы (например, для осуществления операций ввода/вывода данных) используется для выполнения других (таких же, либо менее важных) программ. Иногда при этом говорят о мультизадачном режиме. При этом, вообще говоря, мультипрограммный и мультизадачный режимы – это не синонимы, хотя и близкие понятия. Основное принципиальное отличие в этих терминах заключается в том, что мультипрограммный режим обеспечивает параллельное выполнение нескольких приложений и при этом программисты, создающие эти программы, не должны заботиться о механизмах организации их параллельной работы. Эти функции берет на себя сама ОС; именно она распределяет между выполняющимися приложениями ресурсы вычислительной системы, осуществляет необходимую синхронизацию вычислений и взаимодействие. Мультизадачный режим, наоборот, предполагает, что забота о параллельном выполнении и взаимодействии приложений ложится как раз на прикладных программистов. В современной технической и, тем более, научно-популярной литературе об этом различии часто забывают, тем самым внося некоторую путаницу. Можно, однако, заметить, что современные ОС для ПК реализуют и мультипрограммный, и мультизадачный режимы.

При организации работы с вычислительной системой в диалоговом режиме можно говорить об однопользовательских (однотерминальных) и мультитерминальных ОС. В мультитерминальных ОС с одной вычислительной системой одно-

временно могут работать несколько пользователей, каждый со своего терминала. При этом у пользователей возникает иллюзия, что у каждого из них имеется своя собственная вычислительная система. Очевидно, что для организации мультитерминального доступа к вычислительной системе необходимо обеспечить мультипрограммный режим работы. В качестве одного из примеров мультитерминальных ОС для ПК можно назвать Linux.

Основной особенностью *операционных систем реального времени* (ОСРВ) является обеспечение обработки поступающих заданий в течение заданных интервалов времени, которые нельзя превышать. Поток заданий в общем случае не является планомерным и не может регулироваться оператором (характер следования событий можно предсказать лишь в редких случаях), то есть задания поступают в непредсказуемые моменты времени и без всякой очерёдности. В ОС, не предназначенных для решения задач реального времени, имеются некоторые накладные расходы процессорного времени на этапе инициирования (при выполнении которого ОС распознает все пожелания пользователей относительно решения своей задачи, загружает в оперативную память нужную программу и выделяет другие необходимые для её выполнения ресурсы). В ОСРВ подобные затраты могут отсутствовать, так как набор задач обычно фиксирован и вся информация о задачах известна ещё до поступления запросов. Для подлинной реализации режима реального времени необходима (хотя этого и недостаточно) организация мультипрограммирования. Мультипрограммирование является основным средством повышения производительности вычислительной системы, а для решения задач реального времени производительность становится важнейшим фактором. Лучшие характеристики по производительности для систем реального времени обеспечиваются однотерминальными ОСРВ. Средства организации мультитерминального режима всегда замедляют работу системы в целом, но расширяют функциональные возможности системы. Одной из наиболее известных ОСРВ для ПК является ОС QNX.

По основному архитектурному принципу ОС разделяются на *микроядерные* и *монолитные.* В некоторой степени это разделение тоже условно, однако можно в качестве яркого примера микроядерной ОС привести ОСРВ QNX, тогда как в каче-

стве монолитной можно назвать Windows 95/98 или ОС Linux. Ядро ОС Windows мы не можем изменить, нам не доступны его исходные коды и у нас нет программы для сборки (компиляции) этого ядра. А вот в случае с Linux мы можем сами собрать ядро, которое нам необходимо, включив в него те необходимые программные модули и драйверы, которые мы считаем целесообразным включить именно в ядро (а не обращаться к ним из ядра).

Контрольные вопросы и задачи

Вопросы для проверки

1 Объясните, в чём заключается различие между такими понятиями, как процесс и задача.

2 Изобразите диаграмму состояний процесса, поясните все возможные переходы из одного состояния в другое.

3 Объясните значения следующих терминов: task (задача), process (процесс), thread (поток, нить). Как они между собой соотносятся?

4 Для чего каждая задача получает соответствующий дескриптор? Какие поля, как правило, содержатся в дескрипторе процесса (задачи)? Что такое «контекст задачи»?

5 Объясните понятие ресурса. Почему понятие ресурса является одним из фундаментальных при рассмотрении ОС? Какие виды и типы ресурсов вы знаете?

6 Как вы считаете: сколько и каких списков дескрипторов задач может быть в системе? От чего должно зависеть это число?

7 Перечислите дисциплины обслуживания прерываний; объясните, как можно реализовать каждую из этих дисциплин.

8 С какой целью в ОС вводится специальный системный модуль, иногда называемый супервизором прерываний?

9 В чём заключается различие между повторно-входимыми (re-entrance) и повторно-прерываемыми (re-enterable) программными модулями? Как они реализуются?

10 Что такое привилегированный программный модуль? Почему нельзя создать мультипрограммную ОС, в которой бы не было привилегированных программных модулей?

ГЛАВА 2 Управление задачами и памятью в операционных системах

Итак, время центрального процессора и оперативная память являются основными ресурсами в случае реализации мультипрограммных вычислений.

Оперативная память – это важнейший ресурс любой вычислительной системы, поскольку без неё (как, впрочем, и без центрального процессора) невозможно выполнение ни одной программы. Мы уже отмечали, что память является разделяемым ресурсом. От выбранных механизмов распределения памяти между выполняющимися процессорами очень сильно зависит и эффективность использования ресурсов системы, и её производительность, и возможности, которыми могут пользоваться программисты при создании своих программ. Способы распределения времени центрального процессора тоже сильно влияют и на скорость выполнения отдельных вычислений, и на общую эффективность вычислительной системы.

Понятие процесса (задачи) нам уже известно. В данной главе мы не будем стараться разделять понятия процесс (process) и поток (thread), вместо этого используя как бы обобщающий термин *task* (задача). В других разделах, если специально это не оговаривается, под задачей или процессом следует понимать практически одно и то же. Сейчас же мы будем говорить о разделении ресурса центрального процессора, поэтому термин задача может включать в себя и понятие треда (потока).

Итак, операционная система выполняет следующие основные функции, связанные с управлением задачами:

- ♦ создание и удаление задач;
- ♦ планирование процессов и диспетчеризация задач;
- ♦ 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», «сыном» или «дочерней задачей». «Предок» 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 очерёдей процессов и ресурсов.

Очевидно, что на распределение ресурсов влияют конкретные потребности тех задач, которые должны выполняться параллельно. Другими словами, 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, как мы уже знаем, оно не может распределяться между параллельно выполняющимися процессами, то процессы вынуждены будут очень долго ждать своей очерёди. Таким образом, недоступность одного ресурса может привести к тому, что длительное время не будут использоваться и многие другие ресурсы.

Если же мы возьмем набор таких процессов, которые не будут конкурировать между собой за неразделяемые ресурсы при параллельном выполнении, то, скорее всего, процессы смогут выполниться быстрее (из-за отсутствия дополнительных ожиданий), да и имеющиеся в системе ресурсы будут использоваться более эффективно. Итак, возникает задача подбора такого множества процессов, что при вы-

полнении они будут как можно реже конфликтовать из-за имеющихся в системе ресурсов. Такая задача называется *планированием вычислительных процессов.*

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, то есть текущего наиболее эффективного распределения ресурсов, возникающего практически при каждом событии. Задачи динамического планирования стали называть *диспетчеризацией*[1] .

Очевидно, что планирование осуществляется гораздо реже, чем задача текущего распределения ресурсов между уже выполняющимися процессами и потоками. Основное отличие между долгосрочным и краткосрочным планировщиками заключается в частоте запуска: краткосрочный планировщик, например, может запускаться каждые 30 или 100 мс, долгосрочный – один раз за несколько минут (или чаще; тут многое зависит от общей длительности решения заданий пользователей).

Долгосрочный планировщик решает, какой из процессов, находящихся во входной очерёди, должен быть переведен в очередь готовых процессов в случае освобождения ресурсов памяти. Он выбирает процессы из входной очерёди с целью создания неоднородной мультипрограммной смеси. Это означает, что в очерёди готовых к выполнению процессов должны находиться – в разной пропорции – как процессы, ориентированные на ввод/вывод, так и процессы, ориентированные на преимущественную работу с центральным процессором.

Краткосрочный планировщик решает, какая из задач, находящихся в очерёди готовых к выполнению, должна быть передана на исполнение. В большинстве современных операционных систем, с которыми мы сталкиваемся, долгосрочный планировщик отсутствует.

[1] К сожалению, здесь наблюдается терминологическая несогласованность. Собственно модули супервизора, отвечающие за диспетчеризацию задач, часто называют планировщиками (sheduler). Но фактически, говоря о тех же планировщиках памяти или о каких-нибудь других модулях, отвечающих за динамическое распределение ресурсов, имеют в виду, что эти планировщики осуществляют диспетчеризацию. Наконец, иногда диспетчеризацию называют краткосрочным планированием.

Планирование и диспетчеризация процессов и задач

Стратегии планирования

Прежде всего следует отметить, что при рассмотрении стратегий планирования, как правило, идёт речь о краткосрочном планировании, то есть о диспетчеризации. Долгосрочное планирование, как мы уже отметили, заключается в подборе таких вычислительных процессов, которые бы меньше всего конкурировали между собой за ресурсы вычислительной системы.

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

♦ по возможности заканчивать вычисления (вычислительные процессы) в том же самом порядке, в котором они были начаты;

♦ отдавать предпочтение более коротким процессам;

♦ предоставлять всем пользователям (процессам пользователей) одинаковые услуги, в том числе и одинаковое время ожидания.

Когда говорят о стратегии обслуживания, всегда имеют в виду понятие процесса, а не понятие задачи, поскольку процесс, как мы уже знаем, может состоять из нескольких потоков (задач).

Дисциплины диспетчеризации

Когда говорят о диспетчеризации, то всегда в явном или неявном виде имеют в виду понятие задачи (потока). Если ОС не поддерживает механизм тредов, то можно заменять понятие задачи на понятие процесса. Так как эти термины часто используются именно в таком смысле, мы вынуждены будем использовать термин «процесс» как синоним термина «задача».

Известно большое количество правил (дисциплин диспетчеризации), в соответствии с которыми формируется список (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. 2.1.

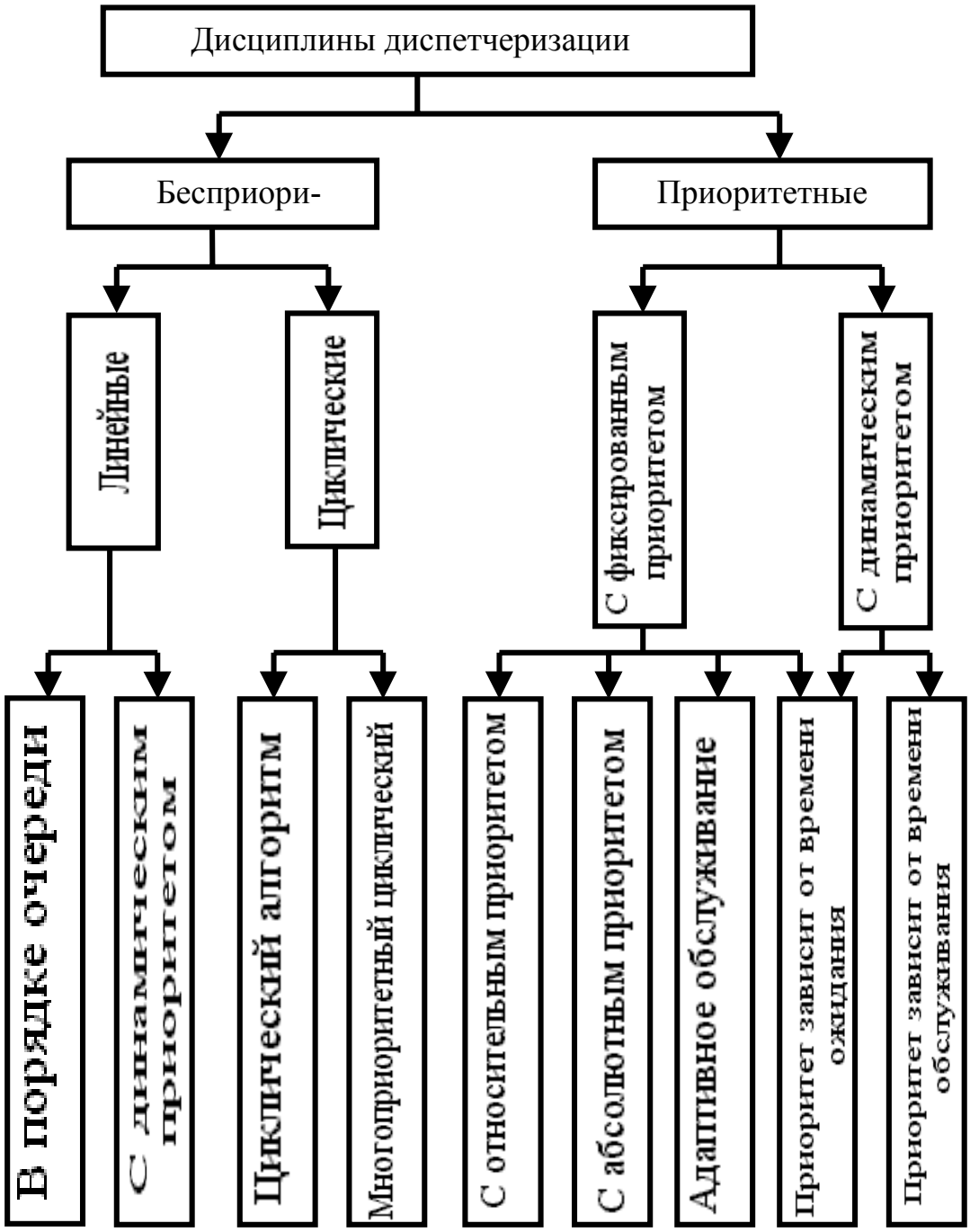

Рис. 2.1. Дисциплины диспетчеризации

Запомним о приоритетах следующее:

- ♦ приоритет, присвоенный задаче, может являться величиной постоянной;
- ♦ приоритет задачи может изменяться в процессе её решения.

Диспетчеризация с динамическими приоритетами требует дополнительных расходов на вычисление значений приоритетов исполняющихся задач, поэтому во многих ОС реального времени используются методы диспетчеризации на основе статических (постоянных) приоритетов. Хотя надо заметить, что динамические приоритеты позволяют реализовать гарантии обслуживания задач.

Рассмотрим кратко некоторые основные (наиболее часто используемые) дисциплины диспетчеризации.

Самой простой в реализации является *дисциплина FCFS* (first come – first served), согласно которой задачи обслуживаются «в порядке очерёди», то есть 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 ещё не выполнялись. Другими словами, образуются две очерёди (рис. 2.2): одна очередь образуется из новых задач, а вторая очередь – из ранее выполнявшихся, но попавших в состояние ожидание. 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 перераспределение процессорного времени. Существующие дисциплины диспетчеризации процессов могут быть разбиты на два класса – вытесняющие (preemptive) и не вытесняющие (non-preemptive). В первых пакетных ОС часто реализовывали параллельное выполнение заданий без принудительного перераспределения процессора между задачами. В большинстве современных ОС для мощных вычислительных систем, а также и в ОС для ПК, ориентированных на высокопроизводительное выполнение приложений (Windows NT, OS/2, Unix), реализована вытесняющая многозадачность. Можно сказать, что рассмотренная дисциплина относится к не вытесняющим.

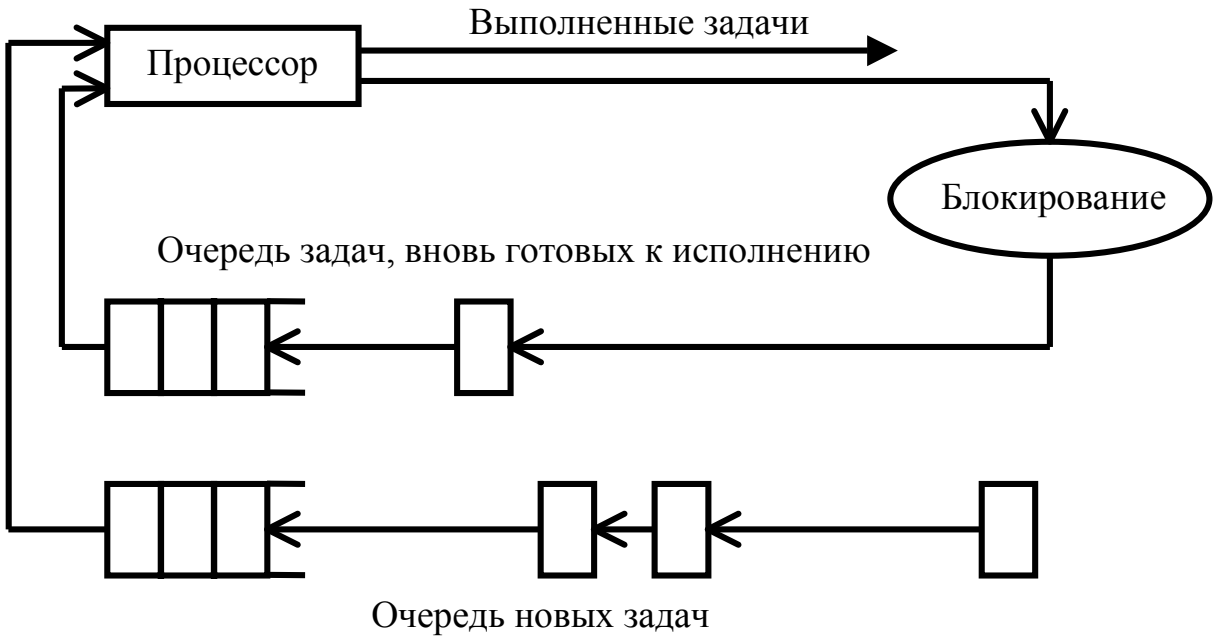

Рис. 2.2. Дисциплина диспетчеризации FCFS

К достоинствам этой дисциплины, прежде всего, можно отнести простоту реализации и малые расходы системных ресурсов на формирование очерёди 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 трудоёмкие задания. Избежать этого недостатка позволяют дисциплины SJN и SRT.

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. В частности, 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 очерёди. ещё 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. И задания, которые в процессе своего исполнения были временно заблокированы (например, ожидали завершения операций ввода/вывода), вновь попадают в конец очерёди готовых к выполнению наравне с вновь поступающими. Это приводит 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 эти три 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 свои результаты вычислений. Для интерактивных же вычислений желательно прежде всего обеспечить приемлемое время реакции системы и равенство в обслуживании, если система является мультитерминальной. Если же это однопользовательская система, но с возможностью мультипрограммной обработки, то желательно, чтобы те программы, с которыми мы сейчас непосредственно работаем, имели лучшее время реакции, нежели наши фоновые задания. При этом мы можем пожелать, чтобы некоторые приложения, выполняясь без нашего непосредственного участия (например, программа получения электронной почты, использующая модем и коммутируемые линии для передачи данных), тем не менее, гарантированно получали необходимую им долю 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*[1] *, q)*. После окончания кванта времени *q* задача снимается с процессора и он передаётся следующей задаче. Снятая задача ставится в конец очерёди задач, готовых к выполнению. Эта дисциплина обслуживания иллюстрируется рис. 2.3. Для оптимальной работы системы необходимо правильно выбрать закон, по которому кванты времени выделяются задачам.

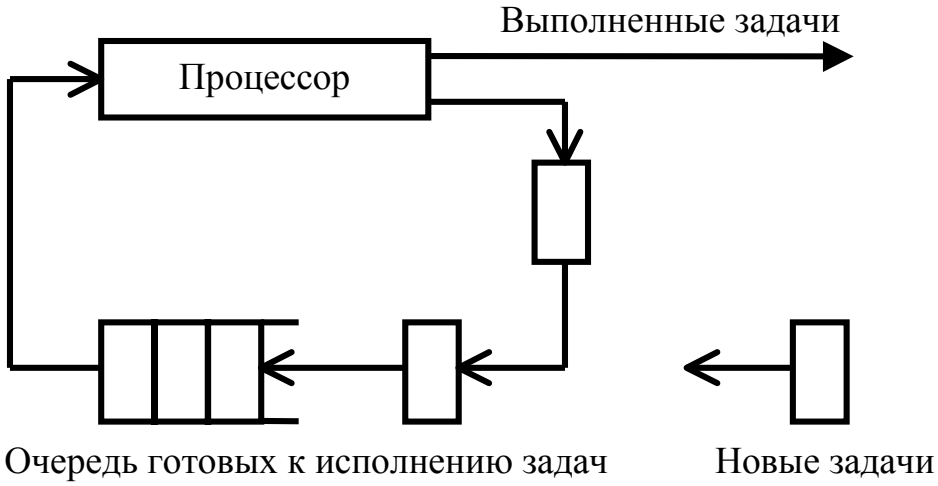

Рис. 2.3. Карусельная дисциплина диспетчеризации (round robin)

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 объём информации о текущем (прерываемом) процессе, поставить дескриптор снятой задачи в очередь, загрузить контекст задачи, которая теперь будет выполняться (её дескриптор был первым в очерёди готовых к исполнению). Если величина *q* велика, то при увеличении очерёди 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 её возможных значений, поскольку система будет стараться выбирать оптимальное значение сама.

[1] *Time slice* – квант времени.

Например, в OS/2 в файле CONFIG.SYS есть возможность с помощью оператора TIMESLICE указать минимальное и максимальное значение для кванта q. Так, например, строка TIMESLICE=32,256 указывает, что минимальное значение q равно 32 миллисекундам, а максимальное – 256. Если некоторая задача (тред) была прервана, поскольку выделенный ей квант времени q израсходован, то следующий выделенный ему интервал будет увеличен на время, равное одному периоду таймера (около 32 мс), и так до тех пор, пока квант времени не станет равным максимальному значению, указанному в операторе TIMESLICE. Этот метод позволяет OS/2 уменьшить накладные расходы на переключение задач в том случае, если несколько задач параллельно выполняют длительные вычисления.

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 очерёдности) [21]. 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 эту 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 очерёди, и поскольку этой высокоприоритетной задаче на самом деле ничего делать не надо, то она тут же освободит процессор и из очерёди готовности будет взята следующая задача.

В своей простейшей реализации дисциплина карусельной диспетчеризации предполагает, что все задачи имеют одинаковый приоритет. Если же необходимо ввести механизм приоритетного обслуживания, то это, как правило, делается за

счёт организации нескольких очерёдей. Процессорное время будет предоставляться в первую очередь тем задачам, которые стоят в самой привилегированной очерёди. Если она пустая, то диспетчер задач начнет просматривать остальные очерёди. Именно по такому алгоритму действует диспетчер задач в операционных системах OS/2 и Windows NT.

Вытесняющие и не вытесняющие алгоритмы диспетчеризации

Диспетчеризация без перераспределения процессорного времени, то есть *не вытесняющая 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 очерёди другого, готового к выполнению процесса или треда. Дисциплины обслуживания FCFS, SJN, SRT относятся к не вытесняющим.

Диспетчеризация с перераспределением процессорного времени между задачами, то есть *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 снятия с выполнения текущей задачи, сохраняет её контекст в дескрипторе задачи, выбирает из очерёди готовых задач следующую и запускает её на выполнение, предварительно загрузив её контекст. Дисциплина RR и многие другие, построенные на её основе, относятся к вытесняющим.

При не вытесняющей многозадачности механизм распределения процессорного времени распределен между системой и прикладными программами. Прикладная программа, получив управление от операционной системы, сама определяет момент завершения своей очередной итерации и передаёт управление супервизору

ОС с помощью соответствующего системного вызова. При этом естественно, что диспетчер задач, так же как и в случае вытесняющей мультизадачности, формирует очерёди задач и выбирает в соответствии с некоторым алгоритмом (например, с учётом порядка поступления задач или их приоритетов) следующую задачу на выполнение. Такой механизм создает некоторые проблемы, как для пользователей, так и для разработчиков.

Для пользователей это означает, что управление системой может теряться на некоторый произвольный период времени, который определяется процессом выполнения приложения (а не системой, старающейся всегда обеспечить приемлемое время реакции на запросы пользователей) [54]. Если приложение тратит слишком много времени на выполнение какой-либо работы (например, на форматирование диска), пользователь не может переключиться с этой задачи на другую задачу (например, на текстовый или графический редактор, в то время как форматирование продолжалось бы в фоновом режиме). Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу.

Поэтому разработчики приложений для не вытесняющей операционной среды, возлагая на себя функции диспетчера задач, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями. Так, упомянутая выше программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку. Подобный метод разделения времени между задачами работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста.

Например, в ныне уже забытой операционной среде Windows 3.x нативные приложения этой системы разделяли между собой процессорное время именно таким образом. И программисты сами должны были обеспечивать «дружественное» отношение своей программы к другим выполняемым одновременно с ней программам, достаточно часто отдавая управление ядру системы. Крайним проявлением «не дружественности» приложения является его зависание, которое при-

водит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный механизм диспетчеризации, во–первых, обеспечивает все задачи процессорным временем, а во–вторых, дает возможность иметь надёжные механизмы для мониторинга вычислений и позволяет снять зависшую задачу с выполнения.

Однако распределение функций диспетчеризации между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм распределения процессорного времени, наиболее подходящий для данного фиксированного набора задач [54]. Так как разработчик сам определяет в программе момент времени отдачи управления, то при этом исключаются нерациональные прерывания программ в «неудобные» для них моменты времени. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные. Примером эффективного использования не вытесняющей многозадачности является сетевая операционная система Novell NetWare, в которой в значительной степени благодаря этому достигнута высокая скорость выполнения файловых операций. Менее удачным оказалось использование не вытесняющей многозадачности в операционной среде Windows 3.x.

Качество диспетчеризации и гарантии обслуживания

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 могут быть не выполнены. Известны случаи, когда вследствие высокой загрузки вычислительной системы отдельные процессы так и не были вы-

полнены, несмотря на то, что прошло несколько лет (!) с момента их планирования. Поэтому вопрос гарантии обслуживания является очень актуальным.

Более жестким требованием к системе, чем просто гарантированное завершение процесса, является его гарантированное завершение к указанному моменту времени или за указанный интервал времени. 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 учётом 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 счёт других процессов) может быть обслужен быстрее, чем это ему реально необходимо. Для того чтобы избежать этого, проще всего процессорное время выделять все-таки квантами. Гарантировать обслуживание можно следующими тремя способами:

♦ 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 – пакетным (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:

♦ Использование (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). Пропускная способность процессора может измеряться количеством процессов, которые выполняются в единицу времени.

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

♦ Время ожидания (waiting time). Под временем ожидания понимается суммарное время нахождения процесса в очерёди 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:

♦ Накладные расходы на переключение процессора. Они определяются не только переключениями контекстов задач, но и (при переключении на процессы другого приложения) перемещениями страниц виртуальной памяти, а также необходимостью обновления данных в кэше (коды и данные одной задачи, находящиеся в кэше, не нужны другой задаче и будут заменены, что приводит к дополнительным задержкам).

♦ Переключение на другой процесс в тот момент, когда текущий процесс выполняет критическую секцию, а другие процессы активно ожидают входа в свою

критическую секцию (см. главу 6). В этом случае потери будут особо велики (хотя вероятность прерывания выполнения коротких критических секций мала). В случае использования мультипроцессорных систем применяются следующие методы повышения производительности системы:

♦ совместное планирование, при котором все потоки одного приложения (неблокированные) одновременно выбираются для выполнения процессорами и одновременно снимаются с них (для сокращения переключений контекста);

♦ 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 учётом так называемых «советов» программы (во время её выполнения). Например, в известной своими новациями ОС Mach имелись два класса таких советов (hints) – указания (разной степени категоричности) о снятии текущего процесса с процессора, а также указания о том процессе, который должен быть выбран взамен текущего.

Диспетчеризация задач с использованием динамических приоритетов

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 реализованы (решены) в течение длительного промежутка времени из-за возросшей нагрузки в вычислительной системе. Потери, связанные с невыполнением таких задач, могут оказаться 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 ОС реального времени имеются средства для изменения приоритета программ[1]. Есть такие средства и во многих ОС, которые не относятся к классу ОСРВ. Введение механизмов динамического изменения приоритетов позволяет

[1] Dynamic priority variation.

реализовать более быструю реакцию системы на короткие запросы пользователей, что очень важно при интерактивной работе, но при этом гарантировать выполнение любых запросов.

Рассмотрим, например, как реализован механизм динамических приоритетов в ОС UNIX, которая, как известно, не относится к ОСРВ. Приоритет процесса вычисляется следующим образом [70]. Во-первых, в вычислении участвуют значения двух полей дескриптора процесса – p_nice и p_cpu. Первое из них назначается пользователем явно или формируется по умолчанию с помощью системы программирования. Второе поле формируется диспетчером задач (планировщиком разделения времени) и называется системной составляющей или текущим приоритетом. Другими словами, каждый процесс имеет два атрибута приоритета, с учётом которого и распределяется между исполняющимися задачами процессорное время: *текущий приоритет,* на основании которого происходит планирование, и заказанный *относительный приоритет,* называемый nice number (или просто nice).

Схема нумерации (числовых значений) текущих приоритетов различна для различных версий UNIX. Например, более высокому значению текущего приоритета может соответствовать более низкий фактический приоритет планирования. Разделение между приоритетами режима ядра и задачи также зависит от версии. Рассмотрим частный случай, когда текущий приоритет процесса варьируется в диапазоне от 0 (низкий приоритет) до 127. (наивысший приоритет). Процессы, выполняющиеся в режиме задачи, имеют более низкий приоритет, чем в режиме ядра. Для режима задачи приоритет меняется в диапазоне 0-65, для режима ядра – 66-95 (системный диапазон). Процессы, приоритеты которых лежат в диапазоне 96-127, являются процессами с фиксированным приоритетом, не изменяемым операционной системой, и предназначены для поддержки приложений реального времени.

Процессу, ожидающему недоступного в данный момент ресурса, система определяет значение *приоритета сна,* выбираемое ядром из диапазона системных приоритетов и связанное с событием, вызвавшее это состояние. Когда процесс пробуждается, ядро устанавливает значение текущего приоритета процесса равным приоритету сна. Поскольку приоритет такого процесса находится в системном

диапазоне и выше, чем приоритет режима задачи, вероятность предоставления процессу вычислительных ресурсов весьма велика. Такой подход позволяет, в частности, быстро завершить системный вызов, выполнение которого, в свою очередь, может блокировать некоторые системные ресурсы.

После завершения системного вызова перед возвращением в режим задачи ядро восстанавливает приоритет режима задачи, сохраненный перед выполнением системного вызова. Это может привести к понижению приоритета, что, в свою очередь, вызовет переключение контекста.

Текущий приоритет процесса в режиме задачи p_priuser, как мы только что отметили, зависит от значения nice number и степени использования вычислительных ресурсов p_cpu:

p_priuser - a*p_nice - b*p_cpu

Задача планировщика разделения времени – справедливо распределить вычислительный ресурс между конкурирующими процессами. Для принятия решения о выборе следующего запускаемого процесса планировщику необходима информация об использовании процессора. Эта составляющая приоритета уменьшается обработчиком прерываний таймера по каждому «тику» таймера. Таким образом, пока процесс выполняется в режиме задачи, его текущий приоритет линейно уменьшается.

Каждую секунду ядро пересчитывает текущие приоритеты процессов, готовых к запуску (приоритеты которых меньше некоторого порогового значения, в нашем примере эта величина равна 65), последовательно увеличивая их. Это осуществляется за счёт того, что ядро последовательно уменьшает отрицательную компоненту времени использования процессора. Как результат, эти действия приводят к перемещению процессов в более приоритетные очерёди и повышают вероятность их последующего запуска.

Возможно использование следующей формулы:

p_cpu = p__cpu/2

Это правило проявляет недостаток нивелирования приоритетов при повышении загрузки системы. Происходит это потому, что в таком случае каждый процесс

получает незначительный объём вычислительных ресурсов и, следовательно, имеет малую составляющую p_cpu, которая ещё более уменьшается благодаря формуле пересчета величины p_cpu. В результате степень использования процессора перестает оказывать заметное влияние на приоритет, и низкоприоритетные процессы (то есть процессы с высоким значением nice number) практически «отлучаются» от вычислительных ресурсов системы.

В некоторых версиях ОС UNIX для пересчета значения p_cpu используется другая формула:

p_cpu = p_cpu*(2*load)/(2*load+1)

Здесь параметр load равен среднему числу процессов, находившихся в очерёди на выполнение за последнюю секунду, и характеризует среднюю загрузку системы за этот период времени. Такой алгоритм позволяет частично избавиться от недостатка планирования по формуле p_cpu = p_cpu/2, поскольку при значительной загрузке системы уменьшение p_cpu при пересчете будет происходить медленнее.

Описанные алгоритмы планирования позволяют учесть интересы низкоприоритетных процессов, так как в результате длительного ожидания очерёди на запуск приоритет таких процессов увеличивается, соответственно увеличивается и вероятность запуска. Эти алгоритмы также обеспечивают более вероятный выбор планировщиком интерактивных процессов по отношению к вычислительным (фоновым). Такие задачи, как командный интерпретатор или редактор, большую часть времени проводят в ожидании ввода, имея, таким образом, высокий приоритет (приоритет сна). При наступлении ожидаемого события (например, пользователь осуществил ввод данных) им сразу же предоставляются вычислительные ресурсы. Фоновые процессы, потребляющие значительные ресурсы процессора, имеют высокую составляющую p_cpu и, как следствие, менее высокий приоритет.

Аналогичные механизмы имеют место и в таких ОС, как OS/2 или Windows NT. Правда, алгоритмы изменения приоритета задач в этих системах иные. Например, в Windows NT каждый поток (тред) имеет базовый уровень приоритета, который лежит в диапазоне от двух уровней ниже базового приоритета процесса, его

породившего, до двух уровней выше этого приоритета, как показано на рис. 2.4. Базовый приоритет процесса определяет, сколь сильно могут различаться приоритеты потоков процесса и как они соотносятся с приоритетами потоков других процессов. Поток наследует этот базовый приоритет и может изменять его так, чтобы он стал немного больше или немного меньше. В результате получается приоритет планирования, с которым поток и начинает исполняться. В процессе исполнения потока его приоритет может отклоняться от базового.

На рис. 2.4 показан динамический приоритет потока, нижней границей которого является базовый приоритет потока, а верхняя – зависит от вида работ, исполняемых потоком. Например, если поток обрабатывает пользовательский ввод, то диспетчер задач Windows NT поднимает его динамический приоритет; если же он выполняет вычисления, то диспетчер постепенно снижает его приоритет до базового. Снижая приоритет одного процесса и поднимая приоритет другого, подсистемы могут управлять относительной приоритетностью потоков внутри процесса.

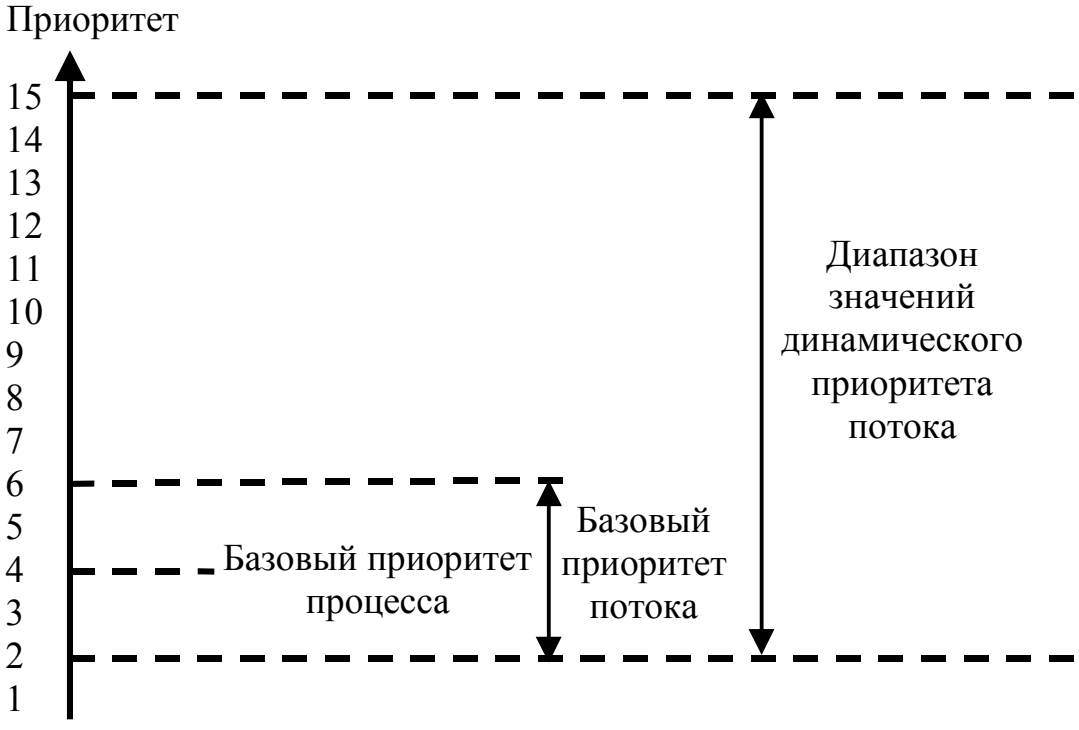

Рис. 2.4. Схема динамического изменения приоритетов в Windows NT

Для определения порядка выполнения потоков диспетчер использует систему приоритетов, направляя на выполнение потоки с высоким приоритетом раньше по-

токов с низкими приоритетами. Система прекращает исполнение или *вытесняет* (preempts) текущий поток, если становится готовой к выполнению другая задача (поток) с более высоким приоритетом.

Существует группа очерёдей – по одной для каждого приоритета. Windows NT поддерживает 32 уровня приоритетов; потоки делятся на два класса: реального времени и переменного приоритета. Потоки реального времени, имеющие приоритеты от 16 до 31 – это высокоприоритетные потоки, используемые программами с критическим временем выполнения, то есть требующие немедленного внимания системы (по терминологии Microsoft).

Диспетчер задач просматривает очерёди, начиная с самой приоритетной. При этом если очередь пустая, то есть нет готовых к выполнению задач с таким приоритетом, осуществляется переход к следующей очерёди. Следовательно, если есть задачи, требующие процессор немедленно, они будут обслужены в первую очередь. Для собственно системных модулей, функционирующих в статусе задач, зарезервирована очередь с номером 0.

Большинство потоков в системе относятся к классу переменного приоритета с уровнями приоритета (номером очерёди) от 1 до 15. Эти очерёди используются потоками с переменным приоритетом (variable priority), так как диспетчер задач корректирует их приоритеты по мере выполнения задач для оптимизации отклика системы. Диспетчер приостанавливает исполнение текущего потока после того, как тот израсходует свой квант времени. При этом если прерванный тред – это поток переменного приоритета, то диспетчер задач понижает его приоритет на единицу и перемещает в другую очередь. Таким образом, приоритет потока, выполняющего много вычислений, постепенно понижается (до значения его базового приоритета). С другой стороны, диспетчер повышает приоритет потока после освобождения задачи (потока) из состояния ожидания. Обычно добавка к приоритету потока определяется кодом исполнительной системы, находящимся вне ядра ОС, однако величина этой добавки зависит от типа события, которого ожидал заблокированный тред. Так, например, поток, ожидавший ввода очерёдного байта с клавиатуры, получает большую добавку к значению своего приоритета, чем процесс вво-

да/вывода, работавший с дисковым накопителем. Однако в любом случае значение приоритета не может достигнуть 16.

В операционной системе OS/2 схема динамической приоритетной диспетчеризации несколько иная, хотя и похожа на рассмотренную[1]. В OS/2 имеются четыре класса задач. И для каждого класса задач имеется своя группа приоритетов с интервалом значений от 0 до 31. Итого, 128 различных уровней и, соответственно, 128 возможных очерёдей готовых к выполнению задач (тредов, потоков).

Класс задач, имеющих самые высокие значения приоритета, называется *критическим* (time critical). Этот класс предназначается для задач, которые мы в обиходе называем задачами реального времени, то есть 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 (о модели клиент–сервер, по которой строятся современные ОС с микроядерной архитектурой, см. в разделе «Микроядерные операционные системы», глава 5), то его ещё иногда называют *серверным*. Приоритет таких задач должен быть выше, это будет гарантировать, что запрос на некоторую функцию со стороны обычных задач выполнится сразу, а не будет дожидаться, пока до него дойдет очередь на фоне других пользовательских приложений.

Большинство задач относят к обычному классу, его ещё называют *регулярным* или стандартным[2]. По умолчанию система программирования порождает задачу, относящуюся именно к этому классу. Наконец, существует ещё класс фоновых за-

[1] Как известно, одно время компания Microsoft принимала активное участие в разработке OS/2 совместно с IBM. Поэтому после прекращения совместных работ над этой операционной системой и начале нового проекта многие решения из OS/2 были унаследованы в варианте OS/2 ver. 3.0, названной впоследствии Windows NT.

[2] Regular.

дач, 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 вышеописанных классов задачи, имеющие одинаковый уровень приоритета, выполняются в соответствии с дисциплиной round-robin. Переход от одного треда 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).

OS/2 самостоятельно изменяет приоритет выполняющихся программ независимо от уровня, установленного самим приложением. Этот механизм называется *повышением приоритета*[1]. Операционная система изменяет приоритет задачи в следующих трех случаях [96]:

♦ Увеличение приоритета активной задачи (foreground boost). Приоритет задачи автоматически увеличивается, когда она становится активной. Это снижает время реакции активного приложения на действия пользователя по сравнению с фоновыми программами.

♦ Увеличение приоритета ввода/вывода (input/output boost). По завершении операции ввода/вывода задача получает самый высокий уровень приоритета её класса. Таким образом обеспечивается завершение всех незаконченных операций ввода/вывода.

♦ Увеличение приоритета «забытых» задач (starvation boost). Если задача не получает управление в течение достаточно долгого времени (этот промежуток времени задает оператор MAXWAIT в файле CONFIG.SYS[2]), диспетчер задач OS/2 временно присваивает ей уровень приоритета, не превышающий критический. В результате переключение на такую «забытую» программу происходит быстрее.

[1] *Priority boost*

[2] Строка MAXWAIT=l означает, что приоритет задачи при переключении на неё будет поднят до максимального не позже чем через одну секунду.

После выполнения приложения в течение одного кванта времени его приоритет вновь снижается до остаточного. В сильно загруженных системах этот механизм позволяет программам с остаточным приоритетом работать хотя бы в краткие интервалы времени. В противном случае они вообще никогда бы не получили управление.

Если нам нет необходимости использовать метод динамического изменения приоритета, то с помощью оператора PRIORITY = ABSOLUTE в файле CONFIG.SYS можно ввести дисциплину абсолютных приоритетов; по умолчанию оператор PRIORITY имеет значение DYNAMIC.

Память и отображения, виртуальное адресное пространство

Если не принимать во внимание программирование на машинном языке (эта технология практически не используется уже очень давно), то можно сказать, что программист обращается к памяти с помощью некоторого набора логических имен, которые чаще всего являются символьными, а не числовыми и для которого отсутствует отношение порядка. Другими словами, в общем случае множество переменных неупорядочено, хотя отдельные переменные и могут иметь частичную упорядоченность (например, элементы массива). *Имена переменных и входных точек программных модулей составляют пространство имен*.

С другой стороны, существует понятие физической оперативной памяти, собственно с которой и работает процессор, извлекая из неё команды и данные и помещая в неё результаты вычислений. *Физическая память* представляет собой упорядоченное множество ячеек, и все они пронумерованы, то есть к каждой из них можно обратиться, указав её порядковый номер (адрес). Количество ячеек физической памяти ограничено и фиксировано.

Системное программное обеспечение должно связать каждое указанное пользователем имя с физической ячейкой памяти, то есть осуществить отображение пространства имён на физическую память компьютера. В общем случае это отображение осуществляется в два этапа (рис.2.5): сначала системой программирования, а затем операционной системой (с помощью специальных программных

модулей управления памятью и использования соответствующих аппаратных средств вычислительной системы). Между этими этапами обращения к памяти имеют форму *виртуального* или логического адреса. При этом можно сказать, что множество всех допустимых значений виртуального адреса для некоторой программы определяет её *виртуальное адресное пространство* или виртуальную память. Виртуальное адресное пространство программы прежде всего зависит от архитектуры процессора и от системы программирования и практически не зависит от объёма реальной физической памяти, установленной в компьютер. Можно ещё сказать, что адреса команд и переменных в готовой машинной программе, подготовленной к выполнению системой программирования, как раз и являются виртуальными адресами.

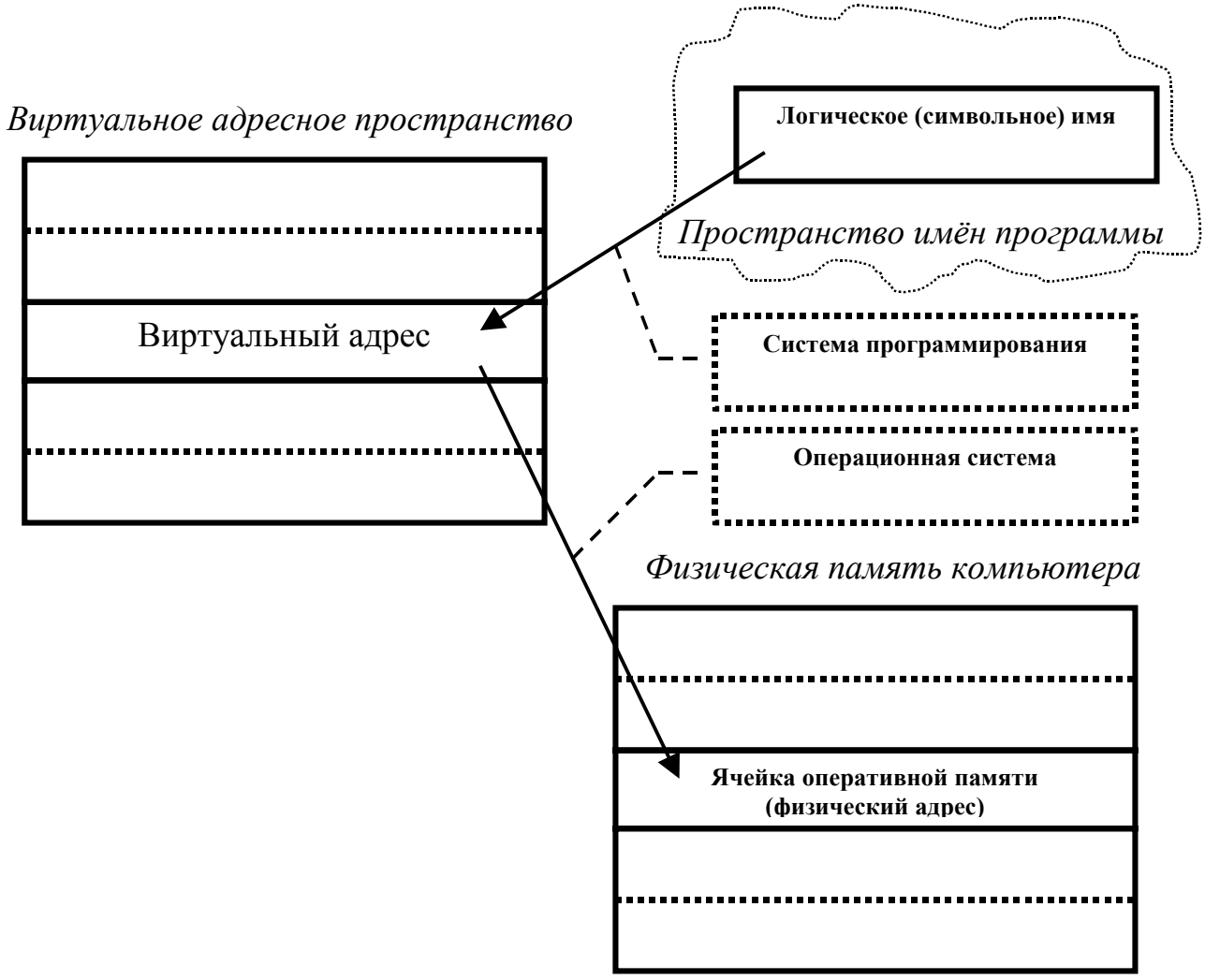

Рис. 2.5. Память и отображения

Как мы знаем, система программирования осуществляет трансляцию и компоновку программы, используя библиотечные программные модули (подробно об этом см. часть 2 настоящего учебника). В результате работы системы программирования полученные виртуальные адреса могут иметь как двоичную форму, так и символьно-двоичную, то есть некоторые программные модули (их, как правило, большинство) и их переменные получают какие-то числовые значения, а те модули, адреса для которых не могут быть сейчас определены, имеют по-прежнему символьную форму и окончательная привязка их к физическим ячейкам будет осуществлена на этапе загрузки программы в память перед её непосредственным выполнением.

Одним из частных случаев отображения пространства имен на физическую память 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 этой программе все двоичные адреса таковы, что программа может исполняться только в том случае, если её виртуальные адреса будут точно соответствовать физическим. Часть программных модулей любой операционной системы обязательно должна быть абсолютными двоичными программами. Эти программы размещаются по фиксированным адресам и с их помощью уже можно впоследствии реализовывать размещение остальных программ, подготовленных системой программирования таким образом, что они могут работать на различных физических адресах (то есть на тех адресах, на которые их разместит операционная система).

Другим частным случаем этой общей схемы трансляции адресного пространства является тождественность виртуального адресного пространства исходному пространству имен. Здесь уже отображение выполняется самой ОС, которая во время исполнения использует таблицу символьных имен. Такая схема отображения используется чрезвычайно редко, так как отображение имен на адреса необходимо выполнять для каждого вхождения имени (каждого нового имени) и особенно много времени тратится на квалификацию имен. Данную схему можно было встретить

в интерпретаторах, в которых стадии трансляции и исполнения практически неразличимы. Это характерно для простейших компьютерных систем, в которых вместо операционной системы использовался встроенный интерпретатор (например, Basic).

Возможны и промежуточные варианты. В простейшем случае транслятор-компилятор генерирует относительные адреса, которые, по сути, являются виртуальными адресами с последующей настройкой программы на один из непрерывных разделов. Второе отображение осуществляется перемещающим загрузчиком. После загрузки программы виртуальный адрес теряется, и доступ выполняется непосредственно к физическим ячейкам. Более эффективное решение достигается в том случае, когда транслятор вырабатывает в качестве виртуального адреса относительный адрес и информацию о начальном адресе, а процессор, используя подготавливаемую операционной системой адресную информацию, выполняет второе отображение не один раз (при загрузке программы), а при каждом обращении к памяти.

Термин *виртуальная память* фактически относится к системам, которые сохраняют виртуальные адреса во время исполнения. Так как второе отображение осуществляется в процессе исполнения задачи, то адреса физических ячеек могут изменяться. При правильном применении такие изменения могут улучшить использование памяти, избавив программиста от деталей управления ею, и даже увеличить надёжность вычислений.

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

- объём виртуального адресного пространства программы V_v меньше объёма физической памяти V_p;
- $V_v = V_p$;
- $V_v > V_p$.

Первая ситуация, при которой $V_v < V_p$, ныне 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}=64K адресов (обычно в качестве адресуемой единицы выступала ячейка памяти размером с байт). А физически старшие модели этих мини-ЭВМ могли иметь объём оперативной памяти в несколько мегабайт. Обращение к памяти столь большого объёма 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 объёмом по 64 Кбайт, и на каждый такой раздел осуществлялось отображение своего виртуального адресного пространства.

Ситуация, когда $V_v = V_p$ встречается очень часто, особенно характерна она была для недорогих вычислительных комплексов. Для этого случая имеется большое количество методов распределения оперативной памяти.

Наконец, в наше время мы уже достигли того, что ситуация $V_v > V_p$ встречается даже в ПК, то есть в самых распространенных и недорогих компьютерах. Теперь это самая распространенная ситуация и для неё имеется несколько методов распределения памяти, отличающихся как сложностью, так и эффективностью.

Простое непрерывное распределение и распределение с перекрытием (оверлейные структуры)

Простое непрерывное распределение – это самая простая схема, согласно которой вся память условно может быть разделена на три части:

- ♦ область, занимаемая операционной системой;
- ♦ область, в которой размещается исполняемая задача;
- ♦ незанятая ничем (свободная) область памяти.

Изначально являясь самой первой схемой, она продолжает и сегодня быть достаточно распространенной. Эта схема предполагает, что ОС не поддерживает

мультипрограммирование, поэтому не возникает проблемы 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 при этом получается непрерывной, что облегчает работу системы программирования. Поскольку в различных однотипных вычислительных комплексах может быть разный состав внешних устройств (и, соответственно, они содержат различное количество драйверов), для системных нужд могут быть отведены отличающиеся объёмы оперативной памяти, и получается, что можно не привязывать жестко виртуальные адреса программы к физическому адресному пространству. Эта привязка осуществляется на этапе загрузки задачи в память.

Чтобы для задач отвести как можно больший объём памяти, операционная система строится таким образом, что постоянно в оперативной памяти располагается только самая нужная её часть. Эту часть ОС стали называть ядром. Остальные модули ОС могут быть обычными диск-резидентными (или транзитными), то есть загружаться в оперативную память только по необходимости, и после своего выполнения вновь освобождать память.

Такая схема распределения влечет за собой два вида потерь вычислительных ресурсов – потеря процессорного времени, потому что процессор простаивает, пока задача ожидает завершения операций ввода/вывода, и потеря самой оперативной памяти, потому что далеко не каждая программа использует всю память, а режим работы в этом случае однопрограммный. Однако это очень недорогая реализация и можно отказаться от многих функций операционной системы. В частности, такая сложная проблема, как защита памяти, здесь вообще не стоит.

Мы не будем рассматривать все различные варианты простого непрерывного распределения памяти, которых было очень и очень много, а ограничимся схемой распределения памяти для MS-DOS (см. раздел «Распределение оперативной памяти в MS-DOS», глава 2).

Если есть необходимость создать программу, логическое (и виртуальное) адресное пространство которой должно быть больше, чем свободная область памяти,

или даже больше, чем весь возможный объём оперативной памяти, то используется распределение с перекрытием (так называемые *оверлейные структуры*[1]). Этот метод 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 её главная часть и один или несколько не перекрывающихся 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), и уже 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 по несколько сегментов. В некоторых вычислительных комплексах могли существовать отдельно сегменты кода и сегменты данных. 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, в отличие от сегментов данных, которые сохранять необходимо.

Первоначально программисты сами должны были включать в тексты своих программ соответствующие обращения к ОС (их называют вызовами) и тщательно планировать, какие сегменты могут находиться в оперативной памяти одновременно, чтобы их адресные пространства не пересекались. Однако с некоторых пор

[1] От *overlay* – перекрытие, расположение поверх чего-то.

эти вызовы система программирования стала подставлять в код программы сама, автоматически, если в том возникает необходимость. Так, в известной и популярной системе программирования Turbo Pascal, начиная с третьей версии, программист просто указывал, что данный модуль является оверлейным. И при обращении к нему из основной программы модуль загружался в память и ему передавалось управление. Все адреса определялись системой программирования автоматически, обращения к DOS для загрузки оверлеев тоже генерировались системой Turbo Pascal.

Распределение статическими и динамическими 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[1]). 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 (то есть процесс выделения нового раздела памяти происходит непосредственно при появлении новой задачи).

Вначале мы кратко рассмотрим статическое распределение памяти на несколько разделов.

Разделы с фиксированными границами

Разбиение всего объёма оперативной памяти на несколько разделов может осуществляться единовременно (то есть в процессе генерации варианта ОС, который потом и эксплуатируется) или по мере необходимости оператором системы. Однако и во втором случае при выполнении разбиения памяти на разделы вычислительная система более ни для каких целей в этот момент не используется.

[1] *Partition, region* – раздел.

Пример разбиения памяти на несколько разделов приведен на рис. 2.6.

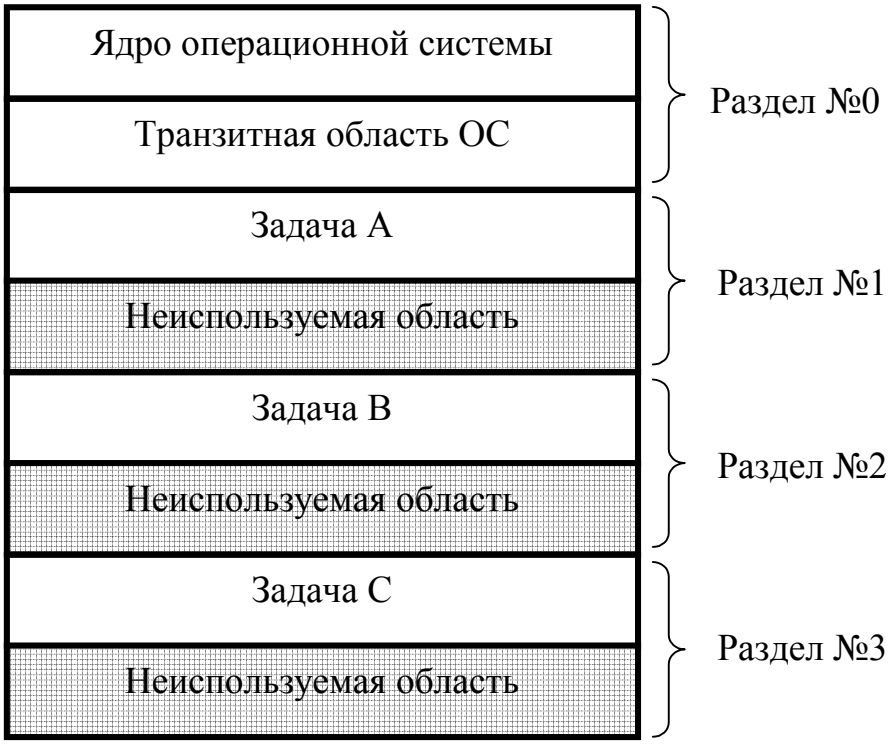

Рис. 2.6. Распределение памяти разделами с фиксированными границами

В каждом разделе в каждый момент времени может располагаться по одной программе (задаче). В этом случае по отношению к каждому разделу можно применить все те методы создания программ, которые используются для однопрограммных систем. Возможно использование оверлейных структур, что позволяет создавать большие сложные программы и в то же время поддерживать коэффициент мультипрограммирования[1] (под коэффициентом мультипрограммирования (μ) понимают количество параллельно выполняемых программ) на должном уровне. Первые мультипрограммные ОС строились по этой схеме. Использовалась эта схема и много лет спустя при создании недорогих вычислительных систем, ибо она является несложной и обеспечивает возможность параллельного выполнения программ. Иногда в некотором разделе размещалось по несколько небольших программ, которые постоянно в нем и находились. Такие программы назывались ОЗУ–резидентными (или просто – *резидентными)*. Они же используются и в современ-

[1] Обычно на практике для загрузки центрального процессора до уровня 90 % необходимо, чтобы коэффициент мультипрограммирования был не менее 4-5. А для того, чтобы наиболее полно использовать и остальные ресурсы системы, желательно иметь μ на уровне 10-15.

ных встроенных системах; правда, для них характерно, что все программы являются резидентными и внешняя память во время работы вычислительного оборудования не используется.

При небольшом объёме памяти и, следовательно, небольшом количестве разделов увеличить количество параллельно выполняемых приложений (особенно когда эти приложения интерактивны и во время своей работы они фактически не используют процессорное время, а в основном ожидают операций ввода/вывода) можно за счёт *своппинга* (swapping). При своппинге задача может быть целиком выгружена на магнитный диск (перемещена во внешнюю память), а на её место загружается либо более привилегированная, либо просто готовая к выполнению другая задача, находившаяся на диске в приостановленном состоянии. При своппинге из основной памяти во внешнюю (и обратно) перемещается вся программа, а не её отдельная часть.

Серьезная проблема, которая возникает при организации мультипрограммного режима работы вычислительной системы, – это защита как самой ОС от ошибок и преднамеренного вмешательства задач в её работу, так и самих задач друг от друга.

В самом деле, программа может обращаться к любым ячейкам в пределах своего виртуального адресного пространства. Если система отображения памяти не содержит ошибок и в самой программе их тоже нет, то возникать ошибок при выполнении программы не должно. Однако в случае ошибок адресации, что не так уж и редко случается, исполняющаяся программа может начать «обработку» чужих данных или кодов с непредсказуемыми последствиями. Одной из простейших, но достаточно сильных мер является введение регистров защиты памяти. В эти регистры ОС заносит граничные значения области памяти раздела текущей исполняющейся задачи. При нарушении адресации возникает прерывание и управление передаётся супервизору ОС. Обращения к ОС за необходимыми сервисами осуществляются не напрямую, а через команды программных прерываний, что обеспечивает передачу управления только в предопределенные входные точки кода ОС, и в системном режиме работы процессора, при котором регистры защиты памяти иг-

норируются. Таким образом, выполнение функции защиты требует введения специальных аппаратных механизмов, используемых операционной системой.

Основным недостатком такого способа распределения памяти является наличие порой достаточно большого объёма неиспользуемой памяти (см. рис. 2.6). Неиспользуемая память 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 могут быть очень значительными, однако использовать фрагменты свободной памяти при таком способе распределения не представляется возможным. Желание разработчиков сократить столь значительные потери привело их к следующим двум решениям:

♦ выделять раздел ровно такого объёма, который нужен под текущую задачу;

♦ размещать задачу не в одной непрерывной области памяти, а в нескольких областях.

Второе решение реализовалось в нескольких способах организации виртуальной памяти. Мы их обсудим в следующем разделе, а сейчас кратко рассмотрим первое решение.

Разделы с подвижными границами

Чтобы избавиться от фрагментации, можно попробовать размещать в оперативной памяти задачи плотно, одну за другой, выделяя ровно столько памяти, сколько задача требует. Одной из первых ОС, реализовавшей такой способ распределения памяти, была OS MVT[1]. Специальный планировщик (диспетчер памяти) ведет список адресов свободной оперативной памяти. При появлении новой задачи диспетчер памяти просматривает этот список и выделяет для задачи раздел, объём 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 объёму. Выделение памяти под новый раздел может осуществляться одним из трех способов:

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

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

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

Как ни странно, самым эффективным правилом является последнее, по которому для нового раздела выделяется «самый неподходящий» фрагмент свободной памяти. Для этой дисциплины список свободных областей упорядочивается по убыванию объёма свободного фрагмента. Очевидно, что если есть такой фрагмент памяти, то он сразу же и будет найден, и поскольку этот фрагмент является самым

[1] MVT (multiprogramming with a variable number of tasks) – мультипрограммирование с переменным числом задач. Эта ОС была одной из самых распространённых при эксплуатировании больших ЭВМ класса IBM 360 (370).

большим, то, скорее всего, после выделения из него раздела памяти для задачи оставшаяся область памяти ещё сможет быть использована в дальнейшем.

Однако очевидно, что при любой дисциплине обслуживания, по которой работает диспетчер памяти, из-за того, что задачи появляются и завершаются в произвольные моменты времени и при этом они имеют разные объёмы, то 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 объём свободных областей будет больше, чем необходимо для задачи. В этой ситуации возможно организовать так называемое *«уплотнение памяти».* Для уплотнения памяти все вычисления приостанавливаются, и диспетчер памяти корректирует свои списки, перемещая разделы в начало памяти (или, наоборот, в область старших адресов). При определении физических адресов задачи будут участвовать новые значения базовых регистров, с помощью которых и осуществляется преобразование виртуальных адресов в физические. Недостатком этого решения является потеря времени на уплотнение и, что самое главное, невозможность при этом выполнять сами вычислительные процессы.

Данный способ распределения памяти, тем не менее, применялся достаточно длительное время в нескольких операционных системах, поскольку в нем для задач выделяется непрерывное адресное пространство, а это упрощает создание систем программирования и их работу.

Сегментная, страничная и сегментно-страничная организация памяти

Методы распределения памяти, при которых задаче уже может не предоставляться сплошная (непрерывная) область памяти, называют разрывными. Идея выделять память задаче не одной сплошной областью, а фрагментами требует для своей реализации соответствующей аппаратной поддержки – нужно иметь относительную адресацию. Если указывать адрес начала текущего фрагмента программы и величину смещения относительно этого начального адреса, то можно указать 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 её на систему программирования.

Сегментный способ организации виртуальной памяти

Первым среди разрывных методов распределения памяти был сегментный. Для этого метода программу необходимо разбивать на части и уже каждой такой части выделять физическую память. Естественным способом разбиения программы на части является разбиение её на логические элементы – так называемые сегменты. В принципе каждый программный модуль (или их совокупность, если мы того пожелаем) может быть воспринят как отдельный сегмент, и вся программа тогда 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 этого 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 осуществит система программирования, а операционная система будет размещать сегменты в память и для каждого сегмента получит информацию о его начале. Таким образом, виртуальный адрес для этого способа будет состоять из двух полей – номер сегмента и смещение относительно начала сегмента. Соответствующая иллюстрация приведена на рис.2.7. На этом рисунке изображен случай обращения к ячейке, виртуальный адрес которой равен сегменту с номером 11 и смещением от начала этого сегмента, равным 612. Как мы видим, операционная система разместила данный сегмент в памяти, начиная с ячейки с номером 19 700.

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.

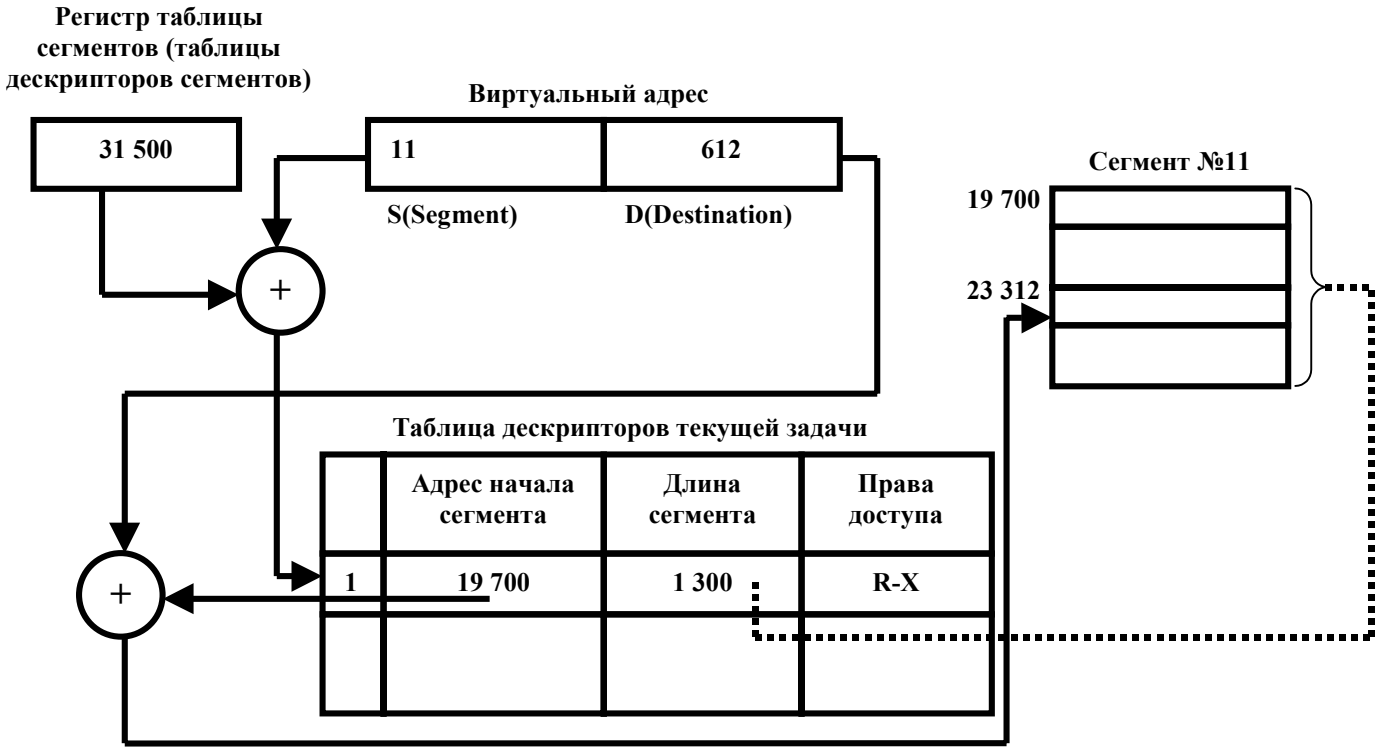

Рис. 2.7. Сегментный способ организации виртуальной памяти

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). 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 фиксировать (обнаруживать) такого рода ошибки.

Если бит present в дескрипторе указывает, что сейчас этот сегмент находится не в оперативной, а во внешней памяти (например, на винчестере), 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 или как давно/недавно этот сегмент используется или не используется, на основании которой можно принять решение о том, чтобы предоставить место, занимаемое текущим сегментом, другому сегменту).

При передаче управления следующей задаче ОС должна занести в соответствующий регистр адрес таблицы дескрипторов сегментов 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. С одной стороны, становится возможным, чтобы общий объём виртуального адресного пространства задачи превосходил объём физической памяти компьютера, на котором эта задача будет выполняться. С другой стороны, даже если потребности в памяти не превосходят имеющуюся физическую память, появляется возможность размещать в памяти как можно больше задач. А увеличение *коэффициента мультипрограммирования* μ, как мы знаем, позволяет увеличить загрузку системы и более эффективно использовать ресурсы вычислительной системы. Очевидно, однако, что увеличивать количество задач можно только до определенного предела, ибо если в памяти не будет хватать места для часто используемых сегментов, то производительность системы резко упадет. Ведь сегмент, который сейчас находится вне оперативной памяти, для участия в вычислениях должен быть перемещен в оперативную память. При этом если в памяти есть свободное пространство, то необходимо всего лишь найти его во внешней памяти и загрузить в оперативную память. А если свободного места сейчас нет, то необходимо будет принять

решение – на место какого из ныне присутствующих сегментов будет загружаться требуемый.

Итак, если требуемого сегмента в оперативной памяти нет, то возникает прерывание и управление передаётся 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, и тогда принимается решение о выгрузке какого-нибудь сегмента и его перемещение во внешнюю память. Если при этом ещё остается время, то процессор передаётся другой готовой к выполнению задаче. После загрузки необходимого сегмента процессор вновь передаётся 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 используется одна из вышеперечисленных дисциплин работы диспетчера памяти (применяются правила «первого подходящего» и «самого неподходящего» фрагментов). Если свободного фрагмента памяти достаточного объёма сейчас нет, но тем не менее сумма этих свободных фрагментов превышает требования по памяти для нового сегмента, то в принципе может быть применено «уплотнение памяти», о котором мы уже говорили в подразделе «Разделы с фиксированными границами» при рассмотрении динамического способа разбиения памяти на разделы.

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

Для решения проблемы замещения (определения того сегмента, который должен быть либо перемещен во внешнюю память, либо просто замещен новым) используются следующие дисциплины[1]:

[1] Их называют «дисциплинами замещения».

♦ правило *FIFO* (first in – first out, что означает: «первый пришедший первым и выбывает»);

♦ правило *LRU* (least recently used, что означает «последний из недавно использованных» или, иначе говоря, «дольше всего неиспользуемый»);

♦ правило *LFU* (least frequently used, что означает: «используемый реже всех остальных»);

♦ случайный (random) 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 будет обращение. Безусловно, достоверной информации о том, какой из сегментов потребуется в ближайшем будущем, в общем случае иметь нельзя, но вероятность ошибки для этих дисциплин многократно выше, чем у второй и третьей дисциплины, которые учитывают информацию об использовании сегментов.

Алгоритм FIFO ассоциирует с каждым сегментом время, когда он был помещён 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 очерёди. Алгоритм учитывает только время нахождения сегмента в памяти, но не учитывает фактическое использование 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. Минимальные требования – 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 ста-

тистическую информацию об обращениях к сегментам. В результате можно составить список, упорядоченный либо по длительности не использования (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.

При использовании сегментного способа организации виртуальной памяти появляется несколько интересных возможностей. Во-первых, появляется возможность при загрузке программы на исполнение размещать её в памяти не целиком, а «по мере необходимости». Действительно, поскольку в подавляющем большинстве случаев алгоритм, по которому работает код программы, является разветвлённым, а не линейным, то в зависимости от исходных данных некоторые части программы, расположенные в самостоятельных сегментах, могут быть и не задействованы; значит, их можно и не загружать в оперативную память. Во-вторых, некоторые программные модули могут быть разделяемыми. Эти программные модули являются сегментами, и в этом случае относительно легко организовать доступ к таким сегментам. Сегмент с разделяемым кодом располагается в памяти в единственном экземпляре, а в нескольких таблицах дескрипторов сегментов исполняющихся задач будут находиться указатели на такие разделяемые сегменты.

Однако у сегментного способа распределения памяти есть и недостатки. Прежде всего, из рис. 2.7 видно, что для получения доступа к искомой ячейке памяти необходимо потратить намного больше времени. Мы должны сначала найти и прочитать дескриптор сегмента, а уже потом, используя данные из него о местонахож-

дении нужного нам сегмента, можем вычислить и конечный физический адрес. Для того чтобы уменьшить эти потери, используется кэширование – то есть те дескрипторы, с которыми мы имеем дело в данный момент, могут быть размещены в сверхоперативной памяти (специальных регистрах, размещаемых в процессоре).

Несмотря на то, что этот способ распределения памяти приводит к существенно меньшей фрагментации памяти, нежели способы с неразрывным распределением, фрагментация остается. Кроме этого, мы имеем большие потери памяти и процессорного времени на размещение и обработку дескрипторных таблиц. Ведь на каждую задачу необходимо иметь свою таблицу дескрипторов сегментов. А при определении физических адресов необходимо выполнять операции сложения.

Поэтому следующим способом разрывного размещения задач в памяти стал способ, при котором все фрагменты задачи одинакового размера и длины, кратной степени двойки, чтобы операции сложения можно было заменить операциями конкатенации (слияния). Это – страничный способ организации виртуальной памяти.

Примером использования сегментного способа организации виртуальной памяти является операционная система для ПК OS/2 первого поколения[1], которая была создана 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.l поддерживала распределение памяти, при котором выделялись сегменты программы и сегменты данных. Система позволяла работать как с именованными, так и 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.l допускала разделение программных сегментов приложений и подсистем, а также глобальных сегментов данных подсистем. Вообще, вся концепция системы OS/2 была построена на понятии разделения памяти: процессы почти всегда разделяют сегменты с другими процессами. В этом состояло суще-

[1] OS/2 v.l начала создаваться в 1984 г. и вышла в продажу в 1987 г.

ственное отличие от систем типа UNIX, которые обычно разделяют только реентерабельные программные модули между процессами.

Сегменты, которые активно не использовались, могли выгружаться на жесткий диск. Система восстанавливала их, когда в этом возникала необходимость. Так как все области памяти, используемые сегментом, должны были быть непрерывными, OS/2 перемещала в основной памяти сегменты таким образом, чтобы максимизировать объём свободной физической памяти. Такое размещение называется *компрессией* или перемещением сегментов (уплотнением памяти). Программные сегменты не выгружались, поскольку они могли просто перезагружаться с исходных дисков. Области в младших адресах физической памяти, которые использовались для запуска DOS-программ и кода самой OS/2, не участвовали в перемещении или подкачке. Кроме этого, система или прикладная программа могли временно фиксировать сегмент в памяти с тем, чтобы гарантировать наличие буфера ввода/вывода в физической памяти до тех пор, пока операция ввода/вывода не завершится.

Если в результате компрессии памяти не удавалось создать необходимое свободное пространство, то супервизор выполнял операции фонового плана для перекачки достаточного количества сегментов из физической памяти, чтобы дать возможность завершиться исходному запросу.

Механизм перекачки сегментов использовал файловую систему для перекачки данных из физической памяти и в неё. Ввиду того, что перекачка и сжатие влияют на производительность системы в целом, пользователь может сконфигурировать систему так, чтобы эти функции не выполнялись.

Было организовано в OS/2 и динамическое присоединение обслуживающих программ. Программы OS/2 используют команды удаленного вызова. Ссылки, генерируемые этими вызовами, определяются в момент загрузки самой программы или её сегментов. Такое отсроченное определение ссылок называется динамическим присоединением. Загрузочный формат модуля OS/2 представляет собой расширение формата загрузочного модуля DOS. Он был расширен, чтобы поддерживать необходимое окружение для свопинга сегментов с динамическим присое-

динением. Динамическое присоединение уменьшает объём памяти для программ в OS/2, одновременно делая возможным перемещения подсистем и обслуживающих программ без необходимости повторного редактирования адресных ссылок к прикладным программам.

Страничный способ организации виртуальной памяти

Как мы уже сказали, при таком способе все фрагменты программы, на которые она разбивается (за исключением последней её части), получаются одинаковыми. Одинаковыми полагаются и единицы памяти, которые мы предоставляем для размещения фрагментов программы. Эти одинаковые части называют страницами и говорят, что память разбивается на физические страницы, а программа – на виртуальные страницы. Часть виртуальных страниц задачи размещается в оперативной памяти, а часть – во внешней. Обычно место во внешней памяти, в качестве которой в абсолютном большинстве случаев выступают накопители на магнитных дисках (поскольку они относятся к быстродействующим устройствам с прямым доступом), называют файлом подкачки или *страничным файлом* (paging file). Иногда этот файл называют *swap-файлом,* тем самым подчеркивая, что записи этого файла – страницы – 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 объёма 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 – это номер виртуальной страницы, P_p – это номер физической страницы и i – это индекс ячейки внутри страницы. Количество

битов, отводимое под индекс, определяет размер страницы, а количество битов, отводимое под номер виртуальной страницы, – объём возможной виртуальной памяти, которой может пользоваться программа. Отображение, осуществляемое системой во время исполнения, сводится к отображению P_v в P_p и приписывании к полученному значению битов адреса, задаваемых величиной i. При этом нет необходимости ограничивать число виртуальных страниц числом физических, то есть 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, для каждой задачи 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 – ведь 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 мы в дескрипторе имеем номер физической страницы, отведенной под данную виртуальную. Если же бит присутствия равен нулю, то в дескрипторе мы будем иметь адрес виртуальной страницы, расположенной сейчас во внешней памяти. Таким образом и осуществляется трансляция виртуального адресного пространства на физическую память. Этот механизм трансляции проиллюстрирован на рис. 2.8.

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.

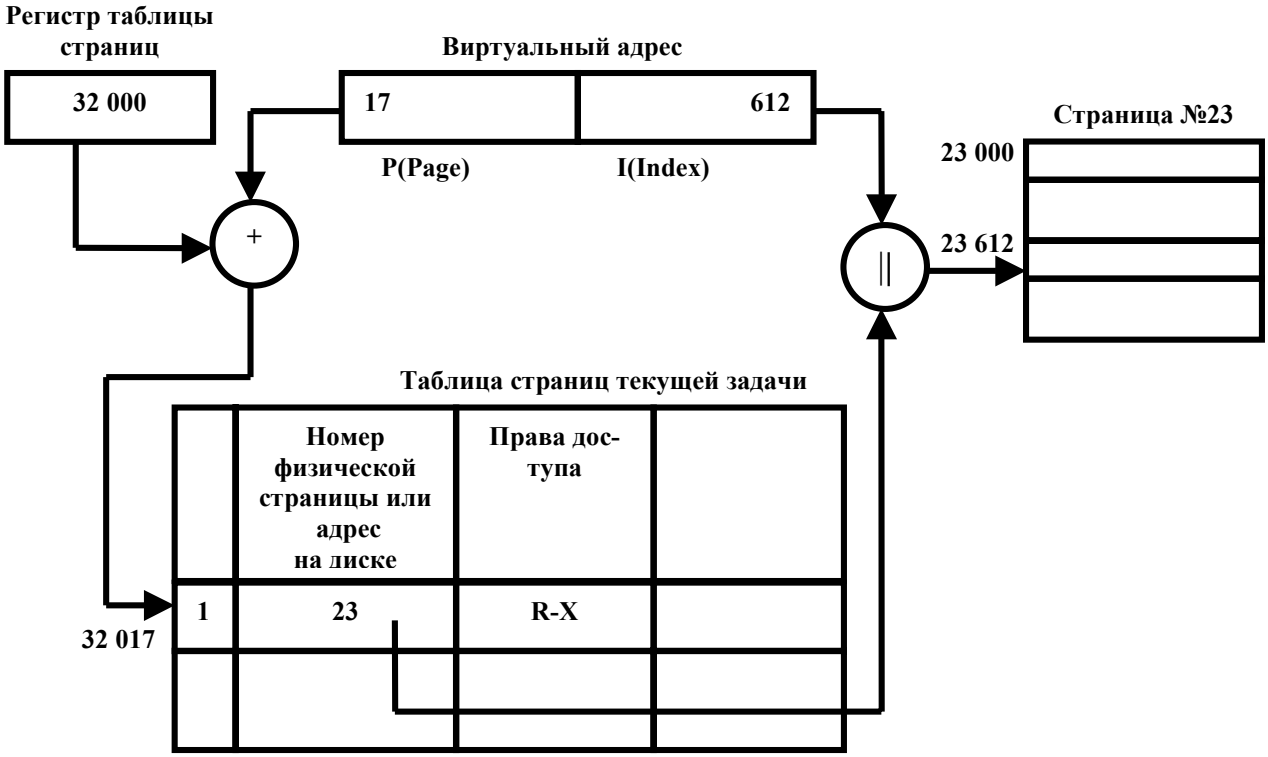

Рис. 2.8. Страничный способ организации виртуальной памяти

При обращении к виртуальной странице, не оказавшейся в данный момент в оперативной памяти, возникает прерывание и управление передаётся диспетчеру памяти, который должен найти свободное место. Обычно предоставляется первая же свободная страница. Если свободной физической страницы нет, то диспетчер памяти по одной из вышеупомянутых дисциплин замещения (LRU, LFU, FIFO, random) определит страницу, подлежащую расформированию или сохранению во внешней памяти. На её место он разместит ту новую виртуальную страницу, к которой было обращение из задачи, но её не оказалось в оперативной памяти.

Напомним, что алгоритм выбирает для замещения ту страницу, на которую не было ссылки на протяжении наиболее длинного периода времени. Дисциплина LRU (least recently used) ассоциирует с каждой страницей время последнего её использования. Для замещения выбирается та страница, которая дольше всех не использовалась.

Для использования дисциплин LRU и LFU в процессоре должны быть соответствующие аппаратные средства. В дескрипторе страницы размещается бит об-

ращения (подразумевается, что на рис. 2.8 этот бит расположен в последнем поле), и этот бит становится единичным при обращении к дескриптору.

Если объём физической памяти небольшой и даже часто требуемые страницы не удается разместить в оперативной памяти, то 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. Очевидно, что это очень плохое явление. Чтобы его не допускать, желательно увеличить объём оперативной памяти (сейчас это стало самым простым решением), 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. Однако в такой ОС, как Windows NT, разработчики, желая сделать систему максимально независимой от аппаратных возможностей процессора, пошли на отказ от этой дисциплины и применили правило FIFO. А для того, чтобы хоть как-нибудь сгладить её неэффективность, была введена «буферизация» тех страниц, которые должны быть записаны в файл подкачки на диск[1] или просто расформированы. Принцип буферирования 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 её 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 вышеназванных ОС, и явление пробуксовки начинается даже при существенно большем объёме оперативной памяти.

[1] В системе Windows NT файл с выгруженными виртуальными страницами носит название PageFile.sys. Таких файлов может быть несколько. Их совокупная длина должна быть не меньше, чем объём физической памяти компьютера плюс 11 Мб, необходимые для самой Windows NT. В системах Windows 2000 размер файла PageFile.sys намного превышает объём установленной физической памяти и часто достигает многих сотен мегабайт.

В ряде ОС с пакетным режимом работы для борьбы с пробуксовкой используется метод «рабочего множества». *Рабочее множество* – это множество «активных» страниц задачи за некоторой интервал То есть тех страниц, к которым было обращение за этот интервал времени. Реально количество активных страниц задачи (за интервал Т) все время изменяется, и это естественно, но, тем не менее, для каждой задачи можно определить среднее количество её активных страниц. Это среднее число активных страниц и есть рабочее множество задачи. Наблюдения за исполнением множества различных программ показали [28, 37, 49], что даже если Т равно времени выполнения всей работы, то размер рабочего множества часто существенно меньше, чем общее число страниц программы. Таким образом, если ОС может определить рабочие множества исполняющихся задач, то для предотвращения пробуксовки достаточно планировать на выполнение только такое количество задач, чтобы сумма их рабочих множеств не превышала возможности системы.

Как и в случае с сегментным способом организации виртуальной памяти, страничный механизм 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 поддерживает страничный способ распределения памяти, в этих микропроцессорах имеется кэш на 32 страничных дескриптора. Поскольку размер страницы в этих микропроцессорах равен 4 Кбайт, возможно быстрое обращение к 128 Кбайт памяти.

Итак, основным достоинством страничного способа распределения памяти является минимально возможная фрагментация. Поскольку на каждую задачу может приходиться по одной незаполненной странице, то становится очевидно, что память можно использовать достаточно эффективно; этот метод организации виртуальной памяти был бы одним из самых лучших, если бы не два следующих обстоятельства.

Первое – эта то, что страничная трансляция виртуальной памяти требует существенных накладных расходов. В самом деле, таблицы страниц нужно тоже размещать в памяти. Кроме этого, 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 тому, что межстраничные переходы, как правило, осуществляются чаще, нежели межсегментные, и к тому, что становится трудно организовать разделение программных модулей между выполняющимися процессами.

Для того чтобы избежать второго недостатка, постаравшись сохранить достоинства страничного способа распределения памяти, был предложен ещё один способ – сегментно-страничный. Правда, за счёт дальнейшего увеличения накладных расходов на его реализацию.

Сегментно-страничный способ организации виртуальной памяти

Как и в сегментном способе распределения памяти, программа разбивается на логически законченные части – сегменты – и виртуальный адрес содержит указание на номер соответствующего сегмента. Вторая составляющая виртуального адреса – смещение относительно начала сегмента – в свою очередь, может состоять из двух полей: виртуальной страницы и индекса. Другими словами, получается, что виртуальный адрес теперь состоит из трех компонентов: сегмент, страница, индекс. Получение физического адреса и извлечение из памяти необходимого элемента для этого способа представлено на рис. 2.9.

Из рисунка сразу видно, что этот способ организации виртуальной памяти вносит ещё большую 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 к номеру физической страницы приписать 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 этой неприятности, вводится кэширование, причем кэш, как правило, строится по ассоциативному принципу. Другими словами, просмотры двух таблиц в памяти могут быть заменены одним обращением к ассоциативной памяти.

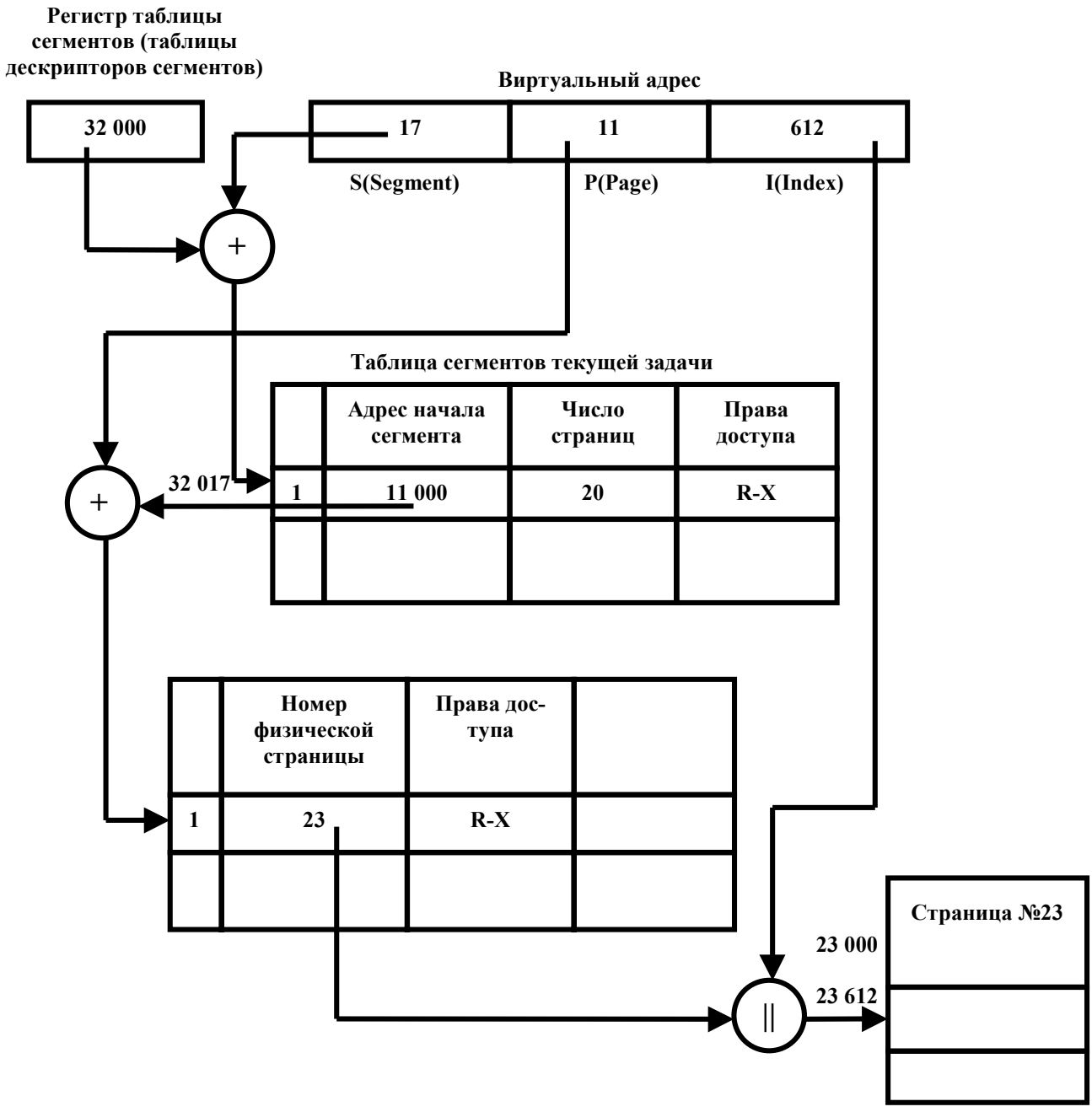

Рис. 2.9. Сегментно-страничный способ организации виртуальной памяти

Напомним, что принцип действия ассоциативного запоминающего устройства предполагает, что каждой ячейке памяти такого устройства ставится в соответствие ячейка, в которой записывается некий ключ (признак, адрес), позволяющий однозначно идентифицировать содержимое ячейки памяти. Сопутствующую ячейку с

информацией, позволяющей идентифицировать основные данные, обычно называют *полем тега.* Просмотр полей тега всех ячеек ассоциативного устройства памяти осуществляется одновременно, то есть в каждой ячейке тега есть необходимая логика, позволяющая посредством побитовой конъюнкции найти данные по их признаку за одно обращение к памяти (если они там, конечно, присутствуют). Часто поле тегов называют аргументом, а поле с данными – функцией. В качестве аргумента при доступе к ассоциативной памяти выступают номер сегмента и номер виртуальной страницы, а в качестве функции от этих аргументов получаем номер физической страницы. Остается приписать номер ячейки в странице к полученному номеру, и мы получаем искомую команду или операнд.

Оценим достоинства сегментно-страничного способа. Разбиение программы на сегменты позволяет размещать сегменты в памяти целиком. Сегменты разбиты на страницы, все страницы сегмента загружаются в память. Это позволяет уменьшить обращения к отсутствующим страницам, поскольку вероятность выхода за пределы сегмента меньше вероятности выхода за пределы страницы. Страницы исполняемого сегмента находятся в памяти, но при этом они могут находиться не рядом друг с другом, а «россыпью», поскольку диспетчер памяти манипулирует страницами. Наличие сегментов облегчает реализацию разделения программных модулей между параллельными процессами. Возможна и динамическая компоновка задачи. А выделение памяти страницами позволяет минимизировать фрагментацию.

Однако, поскольку этот способ распределения памяти требует очень значительных затрат вычислительных ресурсов и его не так просто реализовать, используется он редко, причем в дорогих, мощных вычислительных системах. Возможность реализовать сегментно-страничное распределение памяти заложена и в семейство микропроцессоров i80x86, однако вследствие слабой аппаратной поддержки, трудностей при создании систем программирования и операционной системы, практически он не используется в ПК.

Распределение оперативной памяти в современных ОС для ПК

Первый вопрос, который хочется задать, – это какие ОС следует относить к современным, а какие – нет? Стоит ли в наше время изучать такую «несовременную» ОС, как MS-DOS?[1] С нашей точки зрения, прежде всего к современным ОС следует отнести те, что используют аппаратные возможности микропроцессоров, специально заложенные для организации высокопроизводительных и надёжных вычислений. Однако эти ОС, как правило, очень сложны и громоздки. Они занимают большое дисковое пространство, требуют и большого объёма оперативной памяти. Поэтому для решения некоторого класса задач вполне подходят и системы, использующие микропроцессоры в так называемом реальном режиме работы (см. об этом в следующей главе).

В последние годы можно встретить студентов, обучающихся специальностям, непосредственно связанным с вычислительной техникой, которые совсем не знают DOS-систем. Скорее всего, это является доказательством того, что такие ОС уже не являются современными. Однако достаточно часто для обслуживания компьютера необходимо выполнить простейшие программы – утилиты. Эти программы были созданы для DOS, они не требуют больших ресурсов, для их функционирования достаточно запустить MS-DOS или аналогичную простую ОС. Однако без выполнения этих программ невозможно порой установить или загрузить иные ОС (хоть и современные, но очень сложные и громоздкие). Поэтому мы считаем правильным хотя бы первичное, пусть не очень глубокое ознакомление с MS-DOS.

[1] Широко известно, что было много версий ОС, которые мы, упрощая ситуацию, относим к MS-DOS. MS-DOS – это вариант фирмы Microsoft реализации дисковой операционной системы [3, 28]. Фирма Microsoft прекратила разработку подобных систем, и последней их реализацией была MS-DOS 6.22. Были (и есть ныне) реализации такого рода систем и от других разработчиков. Поскольку у MS-DOS 6.22 существуют известные проблемы с 2000 годом, большой популярностью пользуется PC-DOS 7.0 от IBM.

Распределение оперативной памяти в MS-DOS

Как известно, MS-DOS – это однопрограммная ОС. В ней, конечно, можно организовать запуск резидентных или TSR-задач[1], но в целом она предназначена для выполнения только одного вычислительного процесса. Поэтому распределение памяти в ней построено по самой простой схеме, которую мы уже рассматривали в разделе «Простое непрерывное распределение и распределение с перекрытием (оверлейные структуры)». Здесь мы лишь уточним некоторые характерные детали.

В IBM PC использовался 16-разрядный микропроцессор i8088, который за счёт введения сегментного способа адресации позволял адресоваться к памяти объёмом до 1 Мбайт. В последующих ПК (IBM PC AT, AT386 и др.) было принято решение поддерживать совместимость с первыми, поэтому при работе с DOS прежде всего рассматривают первый мегабайт. Вся эта память разделялась на несколько областей, что проиллюстрировано на рис. 2.10. На этом рисунке изображено, что памяти может быть и больше, чем 1 Мбайт, но более подробное рассмотрение этого вопроса мы здесь опустим, отослав желающих изучить данную тему глубже к монографии [9].

Если не вдаваться в детали, можно сказать, что в состав MS-DOS входят следующие основные компоненты:

♦ Базовая подсистема ввода/вывода – BIOS (base input-output system), включающая в себя помимо программы тестирования ПК (POST[2]) обработчики прерываний (драйверы), расположенные в постоянном запоминающем устройстве. В конечном итоге, почти все остальные модули MS-DOS обращаются к BIOS. Если и не напрямую, то через модули более высокого уровня иерархии.

♦ Модуль расширения BIOS – файл IO.SYS (в других DOS-системах он может называться иначе, например, IBMBIO.COM).

[1] *TSR* (terminate and stay resident) – резидентная в памяти программа, которая благодаря изменениям в таблице векторов прерываний позволяет перехватывать прерывания и в случае обращения к ней выполнять необходимые нам действия. Подробно об этом можно прочесть, например, в книгах [3, 23, 24, 35].

[2] *POST* (power on self test) – программа самотестирования при включении компьютера. После выполнения этой программы, входящей в состав ROM BIOS, опрашиваются устройства, которые могут содержать программы для загрузки ОС.

♦ Основной, базовый модуль обработки прерываний DOS – файл MSDOS.SYS. Именно этот модуль в основном реализует работу с файловой системой. (В PC-DOS аналогичный по значению файл называется IBMDOS.COM).

♦ Командный процессор (интерпретатор команд) – файл COMMAND.COM.

♦ Утилиты и драйверы, расширяющие возможности системы.

♦ Программа загрузки MS-DOS – загрузочная запись (boot record), расположенная на дискете (подробнее о ней и о других загрузчиках см. главу 4).

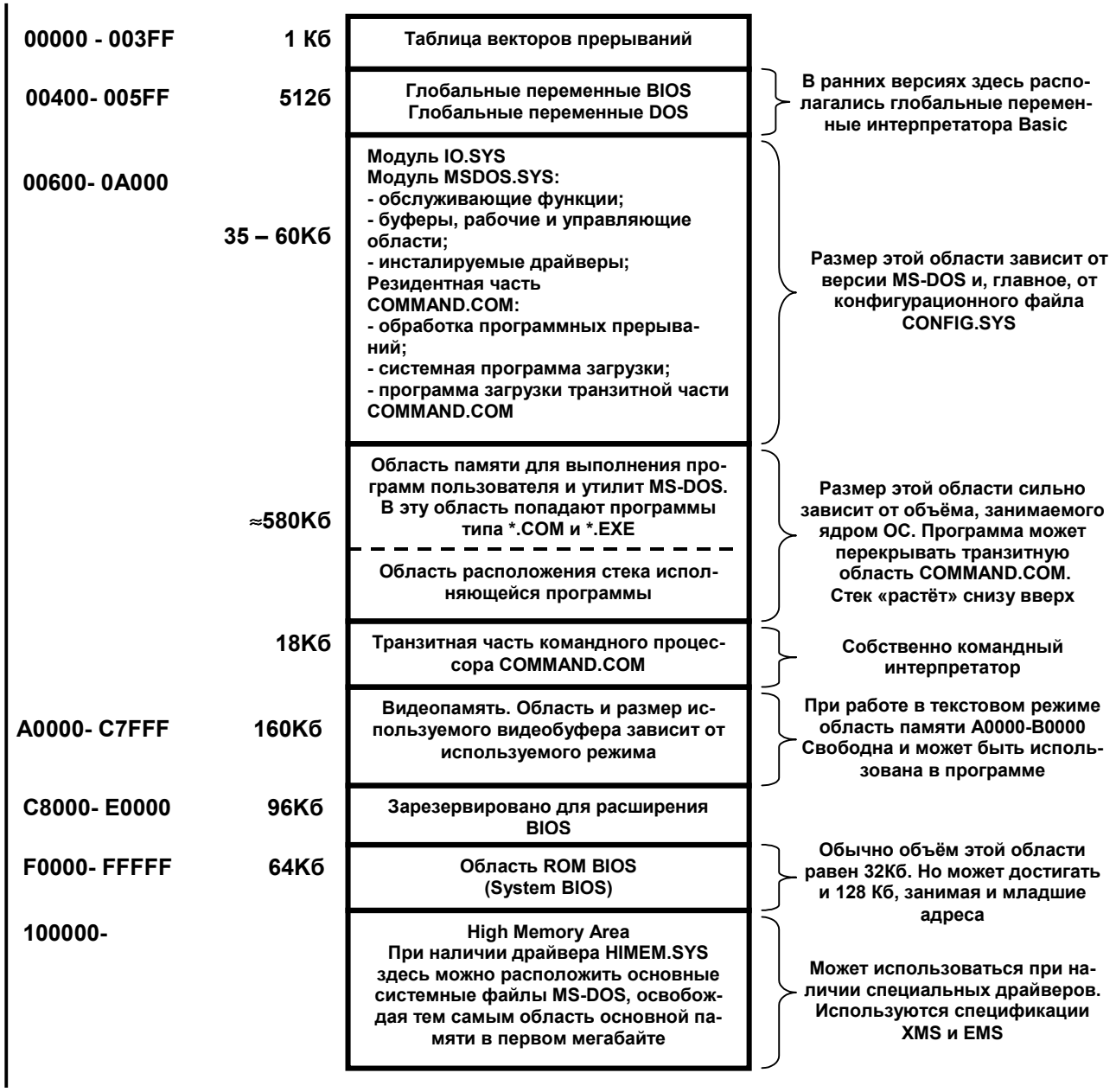

Рис. 2.10. Распределение оперативной памяти в MS-DOS

Вся память в соответствии с архитектурой IBM PC условно может быть разбита на три части.

В самых младших адресах памяти (первые 1024 ячейки) размещается таблица векторов прерываний (см. раздел «Система прерываний 32-разрядных микропроцессоров i80x86», глава 3). Это связано с аппаратной реализацией процессора i8088, на котором была реализована 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. При желании, однако, в случае использования современных микропроцессоров i80x86 можно разместить векторы прерываний и в другой области.

Вторая часть памяти отводится для размещения программных модулей самой MS-DOS и для программ пользователя. Рассмотрим их размещение чуть ниже. Здесь, однако, заметим, что эта область памяти называется *Conventional Memory* (основная, стандартная память).

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).

В младших адресах основной памяти размещается то, что можно назвать ядром этой ОС – системные переменные, основные программные модули, блоки данных для буферирования операций ввода/вывода. Для управления устройствами, драйверы которых не входят в базовую подсистему ввода/вывода, загружаются так называемые *загружаемые* (или *инсталлируемые)* драйверы. Перечень инсталлируемых драйверов определяется специальным конфигурационным файлом CONFIG.SYS. После загрузки расширения BIOS – файла IO.SYS – последний (загрузив модуль MSDOS.SYS) считывает файл CONFIG.SYS и уже в соответствии с ним подгружает в память необходимые драйверы. Кстати, в конфигурационном файле CONFIG.SYS могут иметься и операторы, указывающие на количество буферов, отводимых для ускорения операций ввода/вывода, и на количество файлов, кото-

рые могут обрабатываться (для работы с файлами необходимо зарезервировать место в памяти для хранения управляющих структур, с помощью которых выполняются операции с записями файла). В случае использования микропроцессоров i80x86 и наличия в памяти драйвера HIMEM.SYS модули IO.SYS и MSDOS.SYS могут быть размещены за пределами первого мегабайта в области, которая получила название *HMA* (high memory area).

Память с адресами, большими, чем 10FFFFh, может быть использована в DOS-программах при выполнении их на микропроцессорах, имеющих такую возможность. Так, например, микропроцессор i80286 имел 24-разрядную шину адреса, а i80386 – уже 32-разрядную шину адреса. Но для этого с помощью специальных драйверов необходимо переключать процессор в другой режим работы, при котором он сможет использовать адреса выше 10FFFFh. Широкое распространение получили две основные спецификации: *XMS* (eXtended Memory Specification) и *EMS* (Expanded Memory Specification). Поскольку основные утилиты, необходимые для обслуживания ПК, как правило, не используют эти спецификации, мы не будем здесь их рассматривать. Остальные программные модули MS-DOS (в принципе, большинство из них является утилитами) оформлены как обычные исполняемые файлы. В основном они являются транзитными модулями, то есть загружаются в память только на время своей работы, хотя среди них имеются и TSR-программы.

Для того чтобы предоставить больше памяти программам пользователя, в MS-DOS применено то же решение, что и во многих других простейших ОС – командный процессор COMMAND.COM сделан состоящим из двух частей. Первая часть является резидентной, она размещается в области ядра. Вторая часть – транзитная; она размещается в области старших адресов раздела памяти, выделяемой для программ пользователя. И если программа пользователя перекрывает собой область, в которой была расположена транзитная часть командного процессора, то последний при необходимости восстанавливает в памяти свою транзитную часть, поскольку после выполнения программы она возвращает управление резидентной части COMMAND.COM.

Поскольку размер основной памяти (conventional memory) относительно небольшой, то очень часто системы программирования реализуют оверлейные структуры. Для этого в MS-DOS есть специальные вызовы.

Распределение оперативной памяти в Microsoft Windows 95/98

С точки зрения базовой архитектуры ОС Windows 95/98 они обе являются 32-разрядными, многопотоковыми ОС с вытесняющей многозадачностью. Основной пользовательский интерфейс этих ОС – графический.

Для своей загрузки они используют операционную систему MS-DOS 7.0 (MS-DOS 98), и в случае если в файле MSDOS.SYS в секции [Options] прописано BootGUI = 0, то процессор работает в обычном реальном режиме (см. следующий раздел). Распределение памяти в MS-DOS 7.0. такое же, как и в предыдущих версиях DOS. Однако при загрузке GUI-интерфейса перед загрузкой ядра Windows 95/98 процессор переключается в защищённый режим работы и начинает распределять память уже с помощью страничного механизма.

Использование так называемой *плоской модели памяти,* при которой все возможные сегменты, которые может использовать программист, совпадают друг с другом и имеют максимально возможный размер, определяемый системными соглашениями данной ОС, приводит к тому, что с точки зрения программиста память получается неструктурированной. За счёт представления адреса как пары (P, i) память можно трактовать и как двумерную, то есть «плоскую», но при этом её можно трактовать и как линейную, и это существенно облегчает создание системного программного обеспечения и прикладных программ с помощью соответствующих систем программирования.

Таким образом, в системе фактически действует только страничный механизм преобразования виртуальных адресов в физические. Программы используют классическую «small» (малую) модель памяти [73]. Каждая прикладная программа определяется 32-битными адресами, в которых сегмент кода имеет то же значение, что и сегменты данных. Единственный сегмент программы отображается непосредственно в область виртуального линейного адресного пространства, кото-

рый, в свою очередь, состоит из 4 килобайтных страниц. Каждая страница может располагаться где угодно в оперативной памяти (естественно, в том месте, куда её разместит диспетчер памяти, который сам находится в невыгружаемой области) или может быть перемещена на диск, если не запрещено использовать страничный файл.

Младшие адреса виртуального адресного пространства совместно используются всеми процессами. Это сделано для обеспечения совместимости с драйверами устройств реального режима, резидентными программами и некоторыми 16-разрядными программами Windows. Безусловно, это плохое решение с точки зрения надёжности, поскольку оно приводит к тому, что любой процесс может непреднамеренно (или же, наоборот, специально) испортить компоненты, находящиеся в этих адресах.

В Windows 95/98 каждая 32-разрядная прикладная программа выполняется в своем собственном адресном пространстве, но все они используют совместно один и тот же 32-разрядный системный код. Доступ к чужим адресным пространствам в принципе возможен. Другими словами, виртуальные адресные пространства не используют всех аппаратных средств защиты, заложенных в микропроцессор. В результате неправильно написанная 32-разрядная прикладная программа может привести к аварийному сбою всей системы. Все 16-битовые прикладные программы Windows разделяют общее адресное пространство, поэтому они так же уязвимы друг перед другом, как и в среде Windows 3.x.

Системный код Windows 95 размещается выше границы 2 Гбайт. В пространстве с отметками 2 и 3 Гбайт находятся системные библиотеки DLL[1], используемые несколькими программами. Заметим, что в 32-битовых микропроцессорах семейства i80x86 имеются четыре уровня защиты, именуемые кольцами с номерами от 0 до 3. Кольцо с номером 0 является наиболее привилегированным, то есть максимально защищённым. Компоненты системы Windows 95, относящиеся к кольцу 0, отображаются на виртуальное адресное пространство между 3 и 4 Гбайт. К этим

[1] *DLL* (dynamic link library) – динамически загружаемый библиотечный модуль.

компонентам относятся собственно ядро Windows, подсистема управления виртуальными машинами, модули файловой системы и виртуальные драйверы (VxD).

Область памяти между 2 и 4 Гбайт адресного пространства каждой 32–разрядной прикладной программы совместно используется всеми 32-разрядными прикладными программами. Такая организация позволяет обслуживать вызовы API непосредственно в адресном пространстве прикладной программы и ограничивает размер рабочего множества. Однако за это приходится расплачиваться снижением надёжности. Ничто не может помешать программе, содержащей ошибку, произвести запись в адреса, принадлежащие системным DLL, и вызвать крах всей системы.

В области между 2 и 3 Гбайт также находятся все запускаемые 16-разрядные прикладные программы Windows. С целью обеспечения совместимости эти программы выполняются в совместно используемом адресном пространстве, где они могут испортить друг друга так же, как и в Windows 3.x.

Адреса памяти ниже 4 Мбайт также отображаются в адресное пространство каждой прикладной программы и совместно используются всеми процессами. Благодаря этому становится возможной совместимость с существующими драйверами реального режима, которым необходим доступ к этим адресам. Это делает ещё одну область памяти незащищённой от случайной записи. К самым нижним 64 Кбайт этого адресного пространства 32-разрядные прикладные программы обращаться не могут, что дает возможность перехватывать неверные указатели, но 16-разрядные программы, которые, возможно, содержат ошибки, могут записывать туда данные.

Вышеизложенную модель распределения памяти можно проиллюстрировать с помощью рис. 2.11.

Минимально допустимый объём оперативной памяти, начиная с которого ОС Windows 95 может функционировать, равен 4 Мбайт, однако при таком объёме пробуксовка столь велика, что практически работать нельзя. Страничный файл, с помощью которого реализуется механизм виртуальной памяти, по умолчанию располагается в каталоге самой Windows и имеет переменный размер. Система отслеживает его длину, увеличивая или сокращая этот файл при необходимости. Вместе

с фрагментацией файла подкачки это приводит к тому, что быстродействие системы становится меньше, чем если бы файл был фиксированного размера и располагался в смежных кластерах (был бы дефрагментирован).

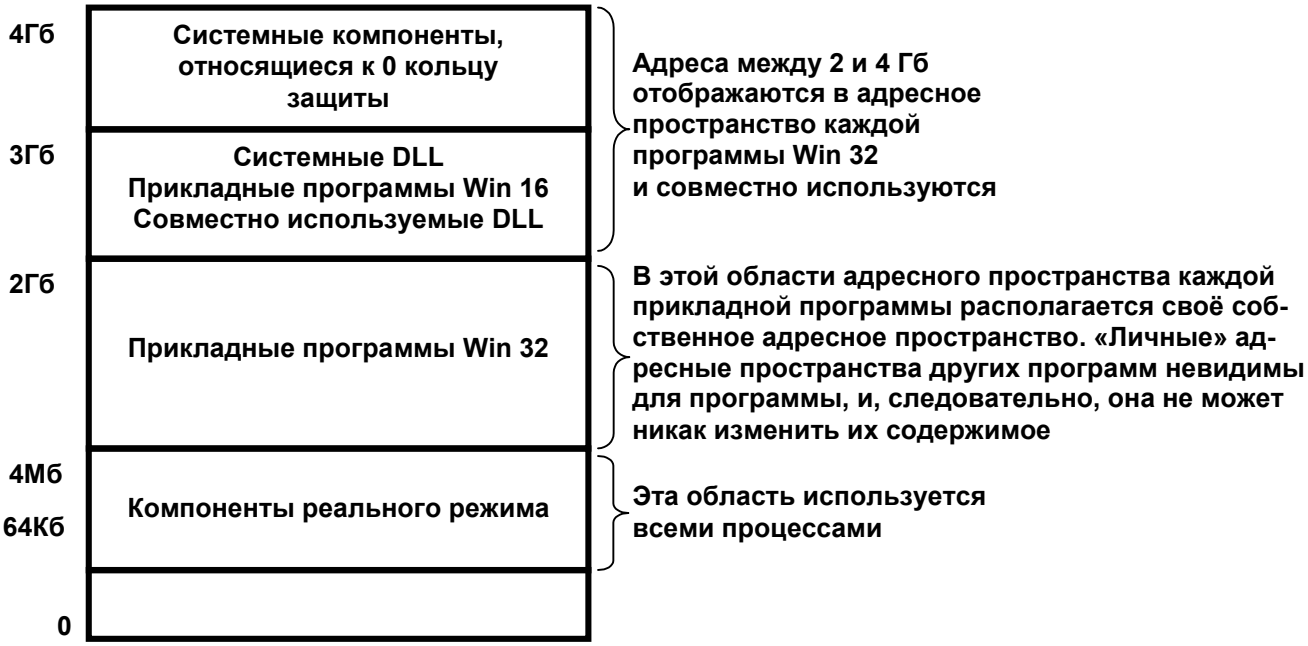

Рис. 2.11. Модель памяти ОС Windows 95/98

Сделать файл подкачки заданного размера можно либо через специально разработанный для этого апплет (Панель управления ⇒Система ⇒ Быстродействие ⇒ Файловая система), либо просто прописав в файле SYSTEM.INI в секции [386Enh] строчки с указанием диска и имени этого файла, например:

PagingDrive=C:

PagingFile=C:\PageFile.sys

MinPagingFileSize=65536

MaxPagingFileSize=262144

Первая и вторая строчки указывают имя страничного файла и его размещение, а две последних – начальный и предельный размер страничного файла (значения указываются в килобайтах). Для определения необходимого минимального размера этого файла можно рекомендовать запустить программу SysMon[1] (системный мо-

[1] Программа SysMon.exe входит в состав штатного программного обеспечения Windows 95/98, но при инсталляции этих ОС на ПК в режиме «обычная» (установка), а не «по выбору», не устанавливается.

нитор) и, выбрав в качестве наблюдаемых параметров размер файла подкачки и объём свободной памяти, оценить потребности в памяти, запуская те приложения, с которыми чаще всего приходится работать.

Распределение оперативной памяти в Microsoft Windows NT

В операционных системах Windows NT тоже используется плоская модель памяти. Заметим, что Windows NT 4.0 server практически не отличается от Windows NT 4.0 workstation; разница лишь в наличии у сервера некоторых дополнительных служб, дополнительных утилит для управления доменом и несколько иных значений в настройках системного реестра. Однако схема распределения возможного виртуального адресного пространства в системах Windows NT разительно отличается от модели памяти Windows 95/98. Прежде всего, в отличие от Windows 95/98 в гораздо большей степени используется ряд серьезных аппаратных средств защиты, имеющихся в микропроцессорах, а также применено принципиально другое логическое распределение адресного пространства.

Во-первых, все системные программные модули находятся в своих собственных виртуальных адресных пространствах, и доступ к ним со стороны прикладных программ невозможен. Ядро системы и несколько драйверов работают в нулевом кольце защиты в отдельном адресном пространстве.

Во-вторых, остальные программные модули самой операционной системы, которые выступают как серверные процессы по отношению к прикладным программам (клиентам), функционируют также в своем собственном системном виртуальном адресном пространстве, невидимом для прикладных процессов. Логическое распределение адресных пространств приведено на рис. 2.12.

Прикладным программам выделяется 2 Гбайт локального (собственного) линейного (неструктурированного) адресного пространства от границы 64 Кбайт до 2 Гбайт (первые 64 Кбайт полностью недоступны). Прикладные программы изолированы друг от друга, хотя могут общаться через буфер обмена (clipboard), механизмы DDE[2] и OLE[3]. Между отметками 2 и 4 Гбайт расположены низкоуровневые

[2] *DDE* (Dynamic Data Exchange) – механизм динамического обмена данными.
[3] OLE (Object Linking and Embedding) – механизм связи и внедрения объектов.

системные компоненты Windows NT кольца 0, в том числе ядро, планировщик потоков и диспетчер виртуальной памяти. Системные страницы в этой области наделены привилегиями супервизора, которые задаются физическими схемами кольцевой защиты процессора.

В верхней части каждой 2-гигабайтной области прикладной программы размещён код системных DLL кольца 3, который выполняет перенаправление вызовов в совершенно изолированное адресное пространство, где содержится уже собственно системный код. Этот системный код, выступающий как сервер-процесс (server process), проверяет значения параметров, исполняет запрошенную функцию и пересылает результаты назад в адресное пространство прикладной программы. Хотя сервер-процесс сам по себе остается процессом прикладного уровня, он полностью защищён от вызывающей его прикладной программы и изолирован от неё.

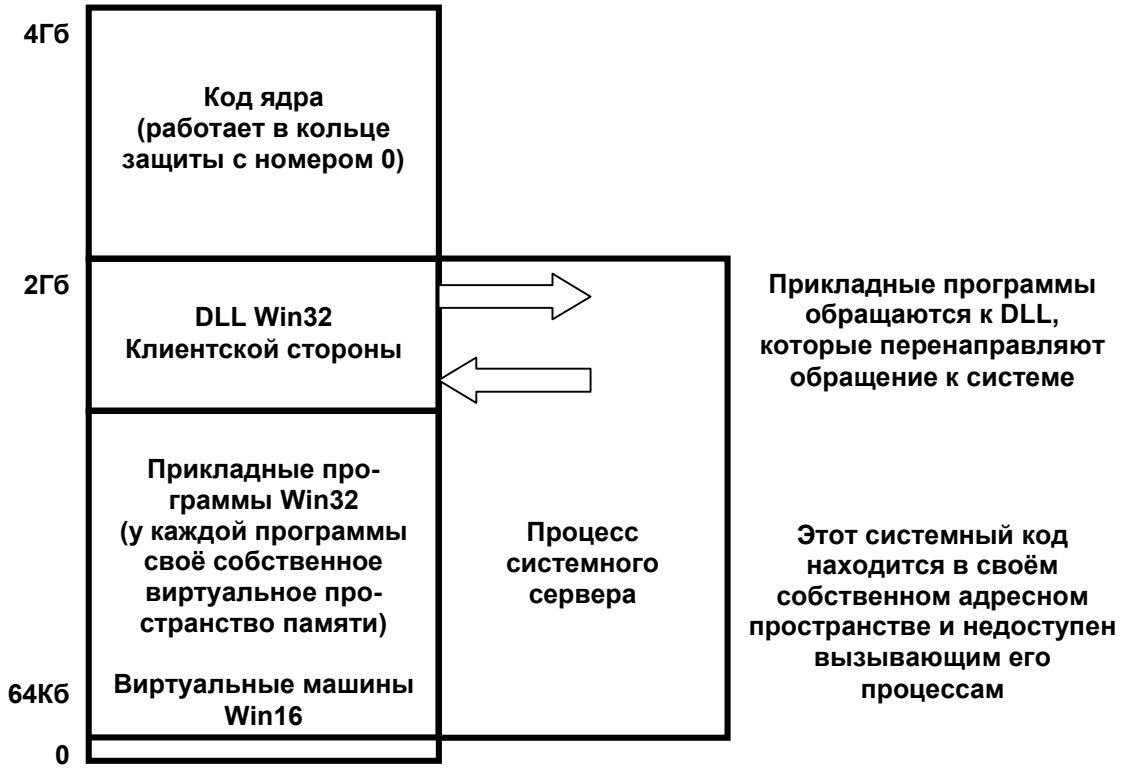

Рис. 2.12. Модель распределения виртуальной памяти в Windows NT

Это делает низкоуровневый системный код невидимым и недоступным для записи для программ прикладного уровня, но приводит к падению производительности во время переходов между кольцами.

Для 16-разрядных прикладных Windows-программ ОС Windows NT реализует сеансы Windows on Windows (WOW). В отличие от Windows 95/98 ОС Windows NT дает возможность выполнять 16-разрядные программы Windows индивидуально в собственных пространствах памяти или совместно в разделяемом адресном пространстве. Почти во всех случаях 16- и 32-разрядные прикладные программы Windows могут свободно взаимодействовать, используя OLE, независимо от того, выполняются они в отдельной или общей памяти. Собственные прикладные программы и сеансы WOW выполняются в режиме вытесняющей многозадачности, основанной на управлении отдельными потоками. Множественные 16-разрядные прикладные программы Windows в одном сеансе WOW выполняются в соответствии с кооперативной моделью многозадачности. Windows NT может также выполнять в многозадачном режиме несколько сеансов DOS. Поскольку Windows NT имеет полностью 32-разрядную архитектуру, не существует теоретических ограничений на ресурсы GDI[1] и USER.

При запуске приложения создается процесс со своей информационной структурой. В рамках процесса запускается задача. При необходимости этот тред (задача) может запустить множество других тредов (задач), которые будут выполняться параллельно в рамках одного процесса. Очевидно, что множество запущенных процессов также выполняются параллельно, и каждый из процессов может представлять из себя мультизадачное приложение. Задачи (треды) в рамках одного процесса выполняются в едином виртуальном адресном пространстве, а процессы выполняются в различных виртуальных адресных пространствах. Отображение различных виртуальных адресных пространств исполняющихся процессов на физическую память реализует сама ОС; именно корректное выполнение этой задачи гарантирует изоляцию приложений от невмешательства процессов. Для обеспечения взаимодействия между выполняющимися приложениями и между приложениями и кодом самой операционной системы используются соответствующие механизмы защиты памяти, поддерживаемые аппаратурой микропроцессора (см. следующую главу).

[1] *GDI* (Graphics Device Interface) – интерфейс графических устройств.

Процессами выделения памяти, её резервирования, освобождения и подкачки управляет диспетчер виртуальной памяти Windows NT (Windows NT virtual memory manager, VMM). В своей работе этот компонент реализует сложную стратегию учёта требований к коду и данным процесса для минимизации доступа к диску, поскольку реализация виртуальной памяти часто приводит к большому количеству дисковых операций.

Каждая виртуальная страница памяти, отображаемая на физическую страницу, переносится в так называемый *страничный фрейм* (page frame). Прежде чем код или данные можно будет переместить с диска в память, диспетчер виртуальной памяти (модуль VMM) должен найти или создать свободный страничный фрейм или фрейм, заполненный нулями. Заметим, что заполнение страниц нулями представляет собой одно из требований стандарта на системы безопасности уровня C2, принятого правительством США. Страничные фреймы должны заполняться нулями для того, чтобы исключить возможность использования их предыдущего содержимого другими процессами. Чтобы фрейм можно было освободить, необходимо скопировать на диск изменения в его странице данных, и только после этого фрейм можно будет повторно использовать. Программы, как правило, не меняют страницы кода. Страницы кода, в которые программы не внесли изменений, можно удалить.

Диспетчер виртуальной памяти может быстро и относительно легко удовлетворить программные прерывания типа «ошибка страницы» (page fault). Что касается аппаратных прерываний типа «ошибка страницы», то они приводят к подкачке (paging), которая снижает производительность системы. Мы уже говорили о том, что в Windows NT, к большому сожалению, выбрана дисциплина FIFO для замещения страниц, а не более эффективные дисциплины LRU и LFU.

Когда процесс использует код или данные, находящиеся в физической памяти, система резервирует место для этой страницы в файле подкачки Pagefile.sys на диске. Это делается с расчетом на тот случай, что данные потребуется выгрузить на диск. Файл Pagefile.sys представляет собой *зарезервированный* блок дискового пространства, который используется для выгрузки страниц, помеченных как «гряз-

ные», при необходимости освобождения физической памяти. Заметим, что этот файл может быть как непрерывным, так и фрагментированным; он может быть расположен на *системном* диске либо на любом другом и даже на нескольких дисках. Размер этого страничного файла ограничивает объём данных, которые могут храниться во внешней памяти при использовании механизмов виртуальной памяти. По умолчанию размер файла подкачки устанавливается равным объёму физической памяти плюс 12 Мбайт, однако, пользователь имеет возможность изменить его размер по своему усмотрению. Проблема нехватки виртуальной памяти часто может быть решена за счёт увеличения размера файла подкачки.

В системах Windows NT 4.0 объекты, создаваемые и используемые приложениями и операционной системой, хранятся в так называемых *пулах памяти* (memory pools). Доступ к этим пулам может быть получен только в привилегированном режиме работы процессора, в котором работают компоненты операционной системы. Поэтому для того, чтобы объекты, хранящиеся в пулах, стали видимы тредам приложений, эти треды должны переключиться в привилегированный режим.

Перемещаемый или нерезидентный пул (paged pool) содержит объекты, которые могут быть при необходимости выгружены на диск. *Неперемещаемый или резидентный пул* (nonpaged pool) содержит объекты, которые должны постоянно находиться в памяти. В частности, к такого рода объектам относятся структуры данных, используемые процедурами обработки прерываний, а также структуры, используемые для предотвращения конфликтов в мультипроцессорных системах.

Исходный размер пулов определяется объёмом физической памяти, доступной Windows NT. Впоследствии размер пула устанавливается динамически и в зависимости от работающих в системе приложений и сервисов будет изменяться в широком диапазоне.

Вся виртуальная память в Windows NT подразделяется на классы: зарезервированную (reserved), выделенную (committed) и доступную (available).

♦ *Зарезервированная память* представляет собой набор непрерывных адресов, которые диспетчер виртуальной памяти (VMM) выделяет для процесса, но не учитывает в общей квоте памяти процесса до тех пор, пока она не будет фактически

использована. Когда процессу требуется выполнить запись в память, ему выделяется нужный объём из зарезервированной памяти. Если процессу потребуется больший объём памяти, то дополнительная память может быть одновременно зарезервирована и использована, если в системе имеется доступная память

♦ *Память выделена,* если диспетчер VMM резервирует для неё место в файле Pagefile.sys на тот случай, когда потребуется выгрузить содержимое памяти на диск. Объём выделенной памяти процесса характеризует фактически потребляемый им объём памяти. Выделенная память ограничивается размером файла подкачки. Предельный объём выделенной памяти в системе (commit limit) определяется тем, какой объём памяти можно выделить процессам без увеличения размеров файла подкачки. Если в системе имеется достаточный объём дискового пространства, то файл подкачки может быть увеличен и тем самым будет расширен предельный объём выделенной памяти.

Вся память, которая не является ни выделенной, ни зарезервированной, является *доступной.* К доступной относится свободная память, обнуленная память (освобожденная и заполненная нулями), а также память, находящаяся в *списке ожидания* (standby list), которая была удалена из рабочего набора процесса, но может быть затребована вновь.

Контрольные вопросы и задачи

Вопросы для проверки

Перечислите и поясните основные функции ОС, которые связаны с управлением задачами.

1 Какие дисциплины диспетчеризации задач вы знаете? Опишите их.

2 Что такое «гарантия обслуживания»? Как её можно реализовать?

3 Сравните механизмы диспетчеризации задач в ОС Windows NT и OS/2. В чём заключаются основные различия?

4 Опишите механизм динамической диспетчеризации, реализованный в UNIX-системах.

5 Что такое «виртуальный адрес», «виртуальное адресное пространство»? Чем (в общем случае) определяется максимально возможный объём виртуального адресного пространства программы?

6 Что такое «фрагментация памяти»? Какой метод распределения памяти позволяет добиться минимальной фрагментации?

7 Что такое «уплотнение памяти»? Когда оно применяется?

8 Объясните сегментный способ организации виртуальной памяти. Что представляет собой (в общем случае) дескриптор сегмента?

9 Сравните сегментный и страничный способы организации виртуальной памяти. Перечислите достоинства и недостатки каждого.

10 Какие дисциплины применяются для решения задачи замещения страниц? Какие из них являются наиболее эффективными и как они реализуются?

11 Что такое «рабочее множество»? Что позволяет разрешить реализация этого понятия?

12 Изложите принципы распределения памяти в MS-DOS.

13 Что означает термин «плоская модель памяти»? В чём заключаются достоинства (и недостатки, если они есть) использования этой модели?

ГЛАВА 3
Особенности архитектуры микропроцессоров i80x86

В рамках данного учебного пособия мы, естественно, не будем рассматривать все многообразие современных 32-разрядных микропроцессоров, используемых в ПК и иных вычислительных системах. Здесь мы ограничимся рассмотрением только архитектурных, а не технических характеристик микропроцессоров. Под обозначением i80x86 будем понимать любые 32-битовые микропроцессоры, имеющие такой же основной набор команд, как и в первом 32-битовом микропроцессоре Intel 80386, и те же архитектурные решения, что и в микропроцессорах фирмы Intel.

Реальный и защищённый режимы работы процессора

Широко известно, что первым микропроцессором, на базе которого была создана IBM PC, был Intel 8088. Этот микропроцессор отличался от первого 16-разрядного микропроцессора фирмы Intel – 8086 – прежде всего тем, что у него была 8-битовая шина данных, а не 16-битовая (как у 8086). Оба эти микропроцессора предназначались для создания вычислительных устройств, которые бы работали в однозадачном режиме, то есть специальных аппаратных средств для поддержки надёжных и эффективных мультипрограммных ОС в них не было.

Однако к тому времени, когда разработчики осознали необходимость включения в микропроцессор специальной аппаратной поддержки для мультипрограммных вычислений, уже было создано очень много программных продуктов. Поэтому для совместимости с первыми компьютерами в последующих версиях микропроцессоров была реализована возможность использовать их в двух режимах – *реальном* (real mode – так назвали режим работы первых 16-битовых микропроцессоров) и защищён*ном* (protected mode – означает, что параллельные вычисления могут быть защищёны аппаратно-программными механизмами).

Подробно рассматривать архитектуру первых 16-битовых микропроцессоров i8086/i8088 мы не будем, поскольку этот материал должен изучаться в предыдущих дисциплинах учебного плана. Для тех же, кто с ним не знаком, можно рекомендовать такие книги, как [52, 73], и многие другие. Однако напомним, что в этих микропроцессорах (а значит, и в остальных микропроцессорах семейства i80x86 при работе их в реальном режиме) обращение к памяти с возможным адресным пространством в 1 Мбайт осуществляется посредством механизма сегментной адресации (рис. 3.1). Этот механизм был использован для увеличения количества разрядов, участвующих в указании адреса ячейки памяти, с которой в данный момент осуществляется работа, с 16 до 20 и тем самым увеличения объёма памяти.

Конкретизируем задачу и ограничимся рассмотрением определения адреса команды. Для адресации операндов используется аналогичный механизм, только участвуют в этом случае другие сегментные регистры. Напомним, что для опре-

деления физического адреса команды содержимое сегментного регистра CS (code segment) умножается на 16 за счёт добавления справа (к младшим битам) четырех нулей, после чего к полученному значению прибавляется содержимое указателя команд (регистр IP, instruction pointer). Получается двадцатибитовое значение, которое и позволяет указать любой байт из 2^{20}.[1]

Рис. 3.1. Схема определения физического адреса для процессора 8086

В защищённом режиме работы определение физического адреса осуществляется совершенно иначе. Прежде всего используется сегментный механизм для организации виртуальной памяти. При этом адреса задаются 32-битовыми значениями. Кроме этого, возможна страничная трансляция адресов, также с 32-битовыми значениями. Наконец, при работе в защищённом режиме, который по умолчанию предполагает 32-битовый код, возможно исполнение двоичных программ, созданных для работы микропроцессора в 16-битовом режиме. Для этого введён режим виртуальной 16-битовой машины и 20-битовые адреса реального режима транслируются с помощью страничного механизма в 32-битовые значения защищённого режима. Наконец, есть ещё один режим – 16-битовый защищённый, позволяющий 32-битовым микропроцессорам выполнять защищённый 16-битовый код, который был характерен для микропроцессора 80286. Правда, следует отметить, что этот

[1] На самом деле, поскольку происходит именно сложение и каждое из слагаемых может иметь значение в интервале от нуля до 2^{16}-1 = 65535= 64К, мы можем указать адрес начала сегмента, равный FFFFFFFF00H, и к нему прибавить смещение FFFFFFFFH. В этом случае мы получим переполнение разрядной сетки, но для современных 32-битовых процессоров (и для уже забытого i80286) имеется возможность указать первые 64 Кбайт выше первого мегабайта.

последний режим практически не используется, поскольку программ, созданных для него, не так уж и много.

Для изучения этих возможностей рассмотрим сначала новые архитектурные возможности микропроцессоров i80x86.

Новые системные регистры микропроцессоров i80x86

Основные регистры микропроцессора i80x86, знание которых необходимо для понимания защищённого режима работы, приведены на рис. 3.2. Следует обратить внимание на следующее:

♦ указатель команды EIP – 32 битовый регистр, младшие 16 разрядов этого регистра есть регистр IP;

♦ регистр флагов EFLAGS – 32 бита, младшие 16 разрядов представляют регистр FLAGS;

♦ регистры общего назначения EAX, EBX. ECX, EDX, а также ESP, EBP, ESI, EDI – 32-битовые, однако их младшие 16 разрядов представляют собой известные регистры AX, BX, CX, DX, SP, BP, SI, DI;

♦ сегментные регистры CS, SS, DS, ES, FS, GS – 16-битовые. При каждом из регистров CS, SS, DS, ES, FS, GS изображены пунктиром скрытые от программистов (недоступны никому, кроме собственно микропроцессора) 64-битовые регистры, в которые загружаются дескрипторы соответствующих сегментов;

♦ регистр-указатель на локальную таблицу сегментов текущей задачи – LDTR (16 битов). При этом регистре также имеется «теневой» (скрытый от программиста) 64-битовый регистр, в который микропроцессор заносит дескриптор, указывающий на таблицу дескрипторов сегментов задачи, описывающих её локальное виртуальное адресное пространство;

♦ регистр-указатель задачи TR[1] (16 битов). Указывает на дескриптор в глобальной таблице дескрипторов, позволяющий получить доступ к дескриптору за-

[1] *TR* – task register.

дачи TSS[2] – информационной структуре, которую поддерживает микропроцессор для управления задачами;

♦ регистр GDTR[1] (48 битов) глобальной таблицы GDT, содержащей как дескрипторы общих сегментов, так и специальные системные дескрипторы. В частности, в GDTR находятся дескрипторы, с помощь которых можно получить доступ к сегментам TSS;

♦ регистр IDTR (48 битов) таблицы дескрипторов прерываний. Содержит информацию, необходимую для доступа к «таблице прерываний» IDT;

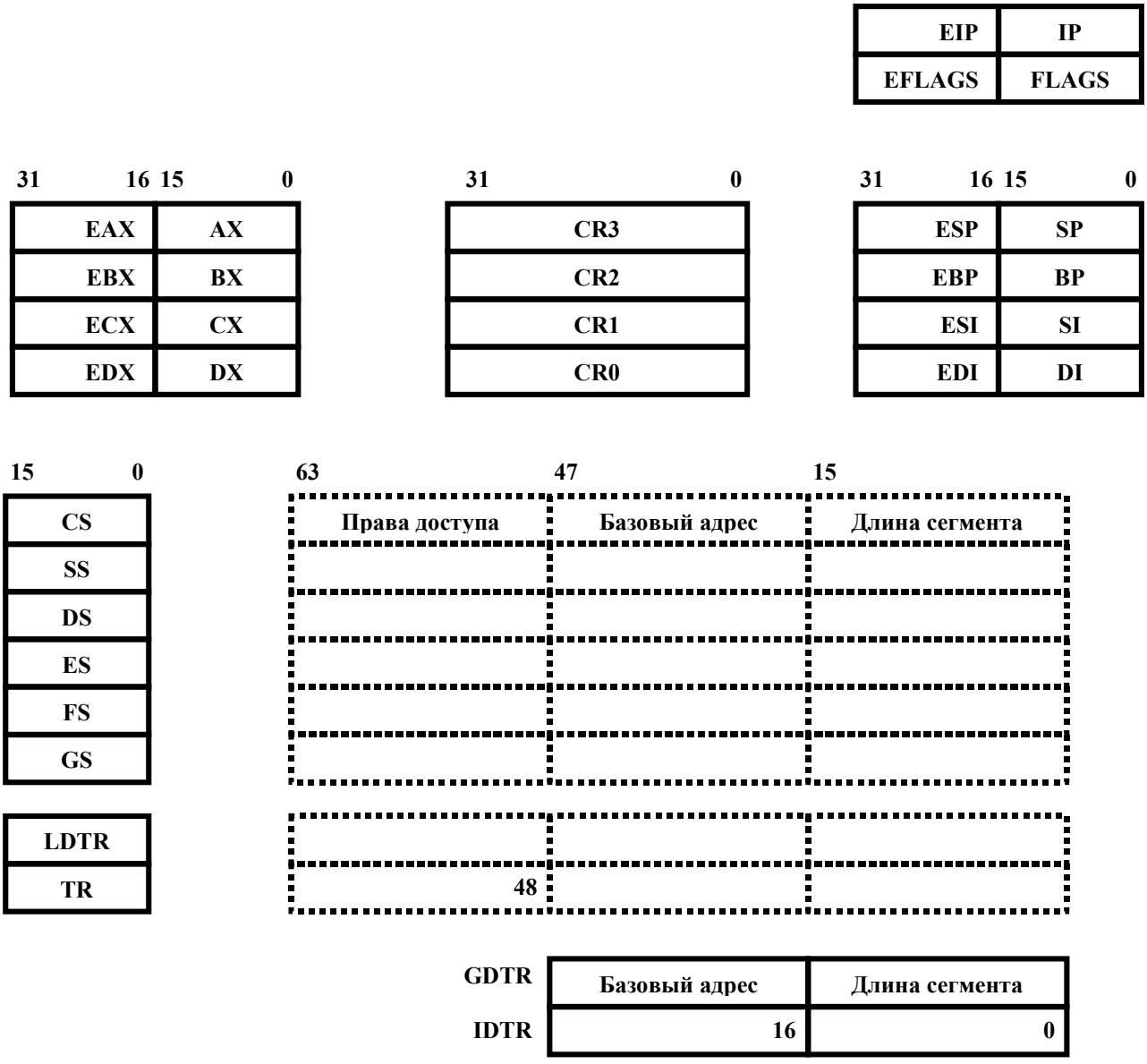

Рис. 3.2. Основные системные регистры микропроцессоров i80x86

[2] *TSS* – task state segment.

♦ управляющие регистры CR0 – CR3 (32-битовые) и некоторые другие регистры.

Управляющий регистр CR0 содержит целый ряд флагов, которые определяют режимы работы микропроцессора. Подробно об этих флагах можно прочитать в книгах [2, 22, 48]. Мы же просто ограничимся тем фактом, что самый младший бит (PE, protect enable) этого регистра определяет режим работы процессора. При PE=0 процессор функционирует в реальном режиме работы, а при единичном значении микропроцессор переключается в защищённый режим. Самый старший бит регистра CR0 (бит PG, paging) определяет, включен (PG=l) или нет (PG=0) режим страничного преобразования адресов.

Регистр CR2 предназначен для размещения в нем адреса подпрограммы обработки страничного исключения, то есть в случае использования страничного механизма отображения памяти обращение к отсутствующей странице будет вызывать переход на соответствующую подпрограмму диспетчера памяти, и для определения этой подпрограммы будет задействован регистр CR2.

Регистр CR3 содержит номер физической страницы, в которой располагается таблица каталогов таблиц страниц текущей задачи. Очевидно, что, приписав к этому номеру нули, мы попадем на начало этой страницы.

Адресация в 32-разрядных микропроцессорах i80x86 при работе в защищённом режиме

Поддержка сегментного способа организации виртуальной памяти

Как мы уже знаем, для организации эффективной и надёжной работы вычислительной системы в мультипрограммном режиме необходимо иметь соответствующие аппаратные механизмы, поддерживающие независимость адресных пространств каждой задачи и в то же время позволяющие организовать обмен данными и разделение кода. Для этого желательно выполнение следующих двух требований:

[1] *GDTR* – global descriptor table register.

♦ чтобы у каждого вычислительного процесса могло быть свое собственное (личное, локальное) адресное пространство, которое никак не может пересекаться с адресными пространствами других задач;

♦ чтобы существовало общее (разделяемое) адресное пространство.

Поэтому в микропроцессорах i80x86 реализован сегментный способ организации распределения памяти. Помимо этого, в этих микропроцессорах может быть задействована и страничная трансляция. Поскольку для каждого сегмента нужен дескриптор, устройство управления памятью поддерживает соответствующую информационную структуру. Формат дескриптора сегмента приведен на рис. 3.3.

Поля дескриптора (базовый адрес, поле предела) размещены в дескрипторе не непрерывно, а в разбивку, во-первых, из-за того, что разработчики постарались минимизировать количество перекрестных соединений в полупроводниковой структуре микропроцессора, а во-вторых – вследствие необходимости обеспечить полную совместимость[1] микропроцессоров (предыдущий микропроцессор i80286 работал с 16-битовым кодом и тоже поддерживал сегментный механизм реализации виртуальной памяти). Необходимо заметить, что формат дескриптора сегмента, изображенный на рис.3.3, справедлив только для случая нахождения соответствующего сегмента в оперативной памяти. Если же бит присутствия в поле прав доступа равен нулю (сегмент отсутствует в памяти), то все биты, за исключением поля прав доступа, считаются неопределенными и могут использоваться системными программистами (для указания адреса сегмента во внешней памяти) произвольным образом.

Локальное адресное пространство задачи определяется через таблицу LDT (local descriptor table). У каждой задачи может быть свое локальное адресное пространство. Общее или *глобальное адресное пространство* определяется через таблицу GDT (global descriptor table). Само собой, что работу с этими таблицами (их заполнение и последующую модификацию) должна осуществлять операционная

[1] Естественно, совместимость обеспечена только «снизу вверх», то есть программы, разработанные для предыдущих версий микропроцессора, должны выполняться на последующих без какой-либо переделки.

система. Доступ к таблицам LDT и GDT со стороны прикладных задач должен быть исключен.

Рис.3.3. Дескриптор сегмента

При переключении микропроцессора в защищённый режим он начинает совершенно другим образом, чем в реальном режиме, вычислять физические адреса команд и операндов. Прежде всего, содержимое сегментных регистров интерпретируется иначе: считается, что там содержится не адрес начала сегмента, а номер соответствующего сегмента. Для того чтобы подчеркнуть этот факт, сегментные регистры CS, SS, DS, ES, FS, GS в таком случае даже называются иначе – *селекторами сегментов.* При этом каждый селекторный регистр разбивается на следующие три поля (рис. 3.4):

♦ поле индекса (index) – старшие 13 битов (3-15). Определяет собственно номер сегмента (его индекс в соответствующей таблице дескрипторов);

♦ поле индикатора таблицы сегментов (table index, TI) – бит с номером 2. Определяет часть виртуального адресного пространства (общее или принадлежащее только данной задаче). Если TI=0, то Index указывает на элемент в глобальной таблице дескрипторов GDT, то есть идёт обращение к общей памяти. Если TI=1, то идёт обращение к локальной области памяти текущей задачи; это пространство описывается локальной таблицей дескрипторов LDT;

♦ поле уровня привилегий – биты 0 и 1. Указывает запрашиваемый уровень привилегий (RPL, requested privilege level).

Операционная система в процессе своего запуска инициализирует многие регистры и, прежде всего, GDTR. Этот регистр содержит начальный адрес глобальной таблицы дескрипторов (GDT) и её размер. Как мы уже знаем, в GDT находятся дескрипторы глобальных сегментов и системные дескрипторы.

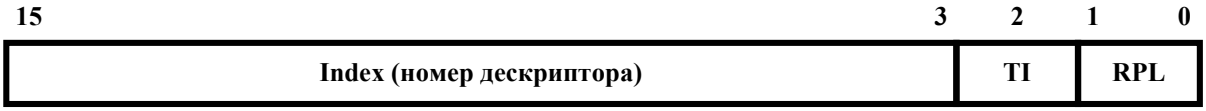

Рис.3.4. Селектор сегмента

Для манипулирования задачами ОС имеет информационную структуру, которую мы уже определили как *дескриптор задачи* (см. раздел «Понятия вычислительного процесса и ресурса», глава 1). Микропроцессор поддерживает работу с наиболее важной частью дескриптора задачи, которая меньше всего зависит от операционной системы. Эта инвариантная часть дескриптора, с которой и работает микропроцессор, названа *сегментом состояния задачи* (task state segment, TSS). Перечень полей TSS изображен на рис. 3.5. Видно, что в основном этот сегмент содержит контекст задачи. Процессор получает доступ к этой структуре с помощью регистра задачи (task register, TR).

Регистр TR содержит индекс (селектор) элемента в GDT. Этот элемент представляет собой дескриптор сегмента TSS. Дескриптор заносится в теневую часть регистра (см. рис. 3.2). К рассмотрению TSS мы ещё вернемся, а сейчас заметим, что в одном из полей TSS содержится указатель (селектор) на локальную таблицу дескрипторов данной задачи. При переходе процессора с одной задачи на другую содержимое поля LDTR заносится микропроцессором в одноименный регистр. Инициализировать регистр TR можно и явным образом.

Итак, регистр LDTR содержит селектор, указывающий на один из дескрипторов глобальной таблицы GDT. Этот дескриптор заносится микропроцессором в теневую часть регистра LDTR и описывает таблицу LDT для текущей задачи. Теперь, когда у нас определены как глобальная, так и локальная таблица дескрипторов,

можно рассмотреть процесс определения линейного[1] адреса. Для примера рассмотрим процесс получения адреса команды. Адреса операндов определяются по аналогии, но задействованы будут другие регистры.

Микропроцессор анализирует бит TI селектора кода и в зависимости от его значения, извлекает из таблицы GDT или LDT дескриптор сегмента кода с номером (индексом), который равен полю *index* (биты 3-15 селектора, см. рис. 3.4). Этот дескриптор заносится в теневую (скрытую) часть регистра CS. Далее микропроцессор сравнивает значение регистра EIP[2] с полем размера сегмента, содержащегося в извлеченном дескрипторе, и если смещение относительно начала сегмента не превышает размера предела, то значение EIP прибавляется к значению поля начала сегмента и мы получаем искомый линейный адрес команды.

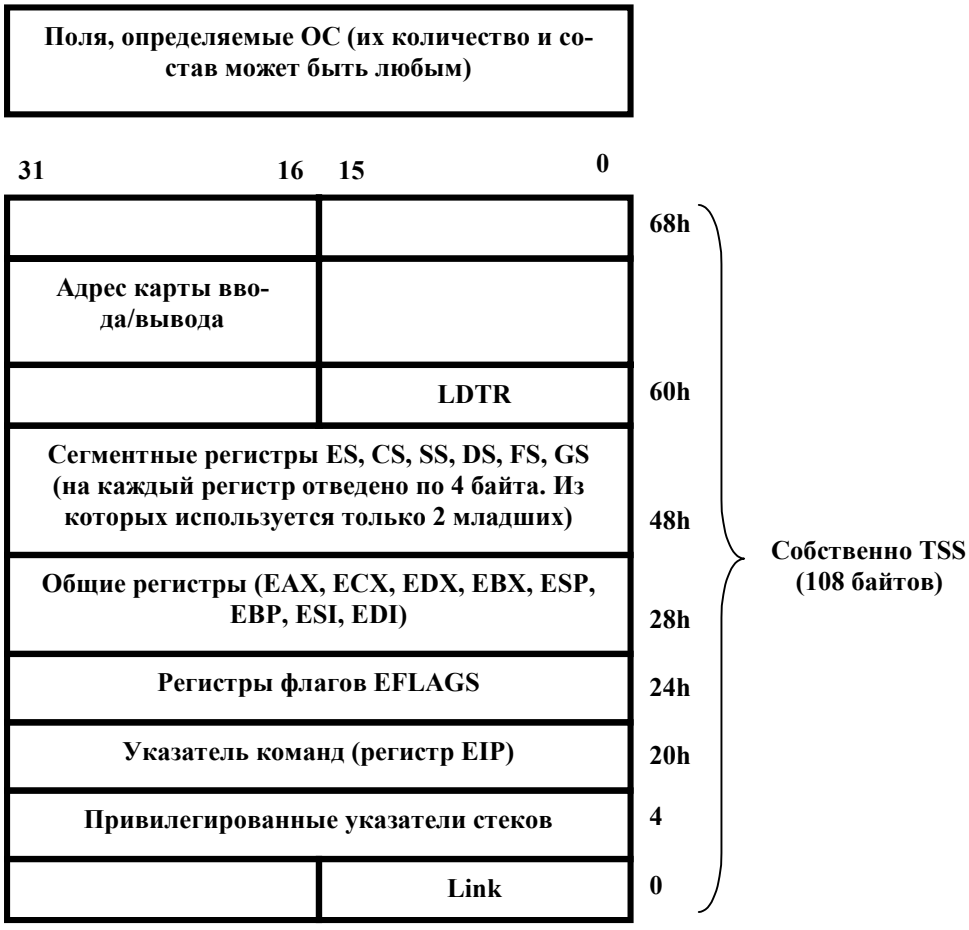

Рис. 3.5. Сегмент состояния задачи (TSS)

[1] В микропроцессорах i80x86 линейным называется адрес, полученный в результате преобразования виртуального адреса формата (S, d) в 32-битовый адрес.
[2] *EIP* (extended instruction pointer) – указатель инструкции (команды).

Линейный адрес – это одна из форм виртуального адреса. Исходный двоичный виртуальный адрес, вычисляемый в соответствии с используемой адресацией, преобразуется в линейный. В свою очередь, линейный адрес будет либо равен физическому (если страничное преобразование отключено), либо с помощью страничной трансляции преобразуется в физический адрес. Если же смещение из регистра EIP превышает размер сегмента кода, то эта аварийная ситуация вызывает прерывание и управление должно передаваться супервизору ОС.

Рассмотренный нами процесс получения линейного адреса проиллюстрирован на рис. 3.6. Стоит отметить, что поскольку межсегментные переходы происходят нечасто, то, как правило, определение линейного адреса заключается только в сравнении значения EIP с полем предела сегмента и в прибавлении смещения к началу сегмента. Все необходимые данные уже находятся в микропроцессоре, и операция получения линейного адреса происходит очень быстро.

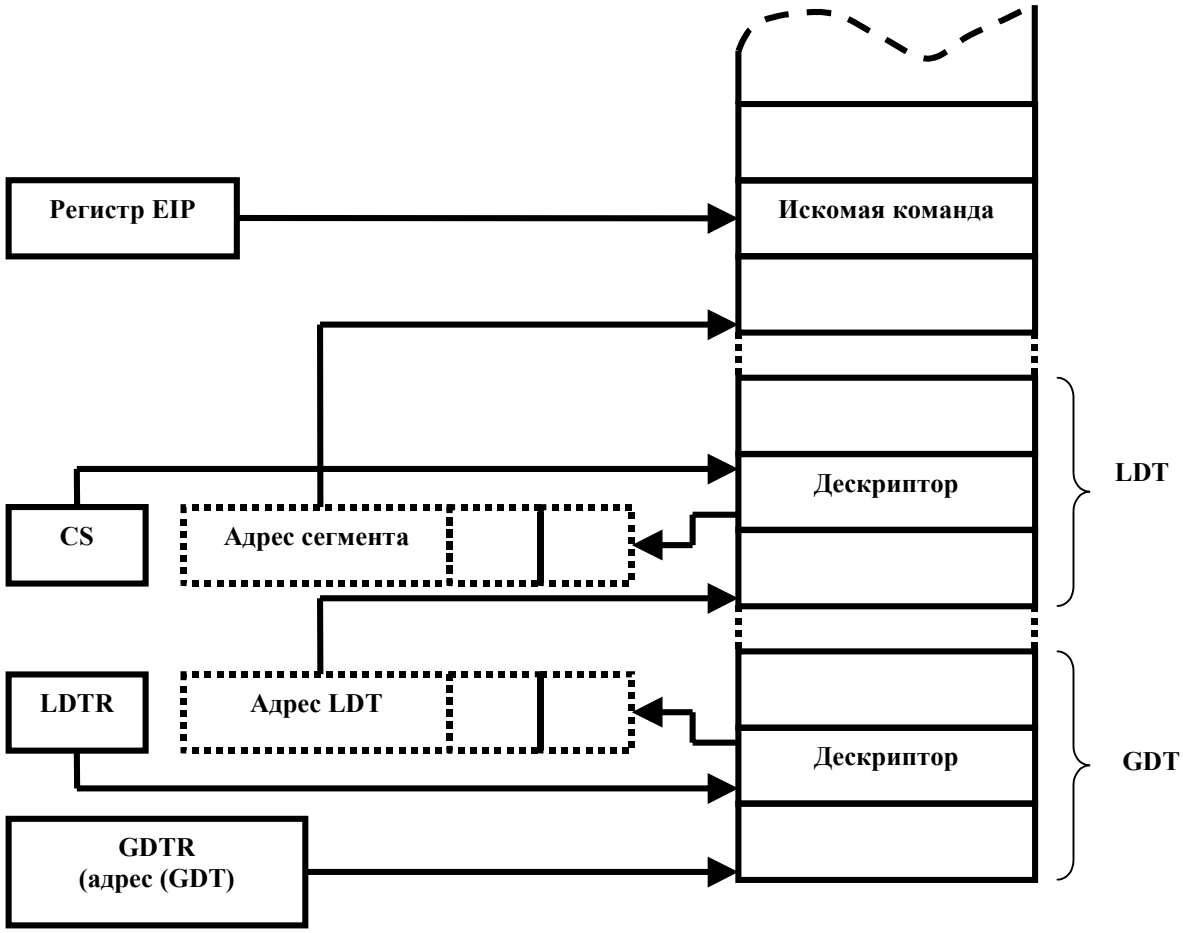

Рис. 3.6. Процесс получения линейного адреса команды

Итак, линейный адрес может считаться физическим адресом, если не включен режим страничной трансляции адресов. Аппаратные средства микропроцессора для поддержки рассмотренного способа двойной трансляции виртуальных адресов в физические явно недостаточны, и при наличии большого количества небольших сегментов приводят к медленной работе. В самом деле, теневой регистр при каждом селекторе имеется в единственном экземпляре, и при переходе на другой сегмент потребуется вновь находить и извлекать соответствующий дескриптор сегмента, а это требует времени. Страничный же способ трансляции виртуальных адресов, как мы знаем, имеет немало своих достоинств. Поэтому в защищённом режиме работы, при котором всегда действует описанный выше механизм определения линейных адресов, может быть включен ещё и страничный механизм.

Поддержка страничного способа организации виртуальной памяти

При создании микропроцессора i80386 разработчики столкнулись с очень серьезной проблемой в реализации страничного механизма. Дело в том, что микропроцессор имеет широкую шину адреса – 32 бита – и возникает вопрос о разбиении всего адреса на поле страницы и поле индекса. Если большое количество битов адреса отвести под индекс, то страницы станут очень большими, что повлечет большие потери и на фрагментацию, и на операции ввода/вывода, связанные с замещением страниц. Хотя количество страниц стало бы при этом меньше, и накладные расходы на их поддержание тоже уменьшились бы. Если же размер страницы уменьшить, то большое поле номера страницы привело бы к появлению громадного количества возможных страниц и необходимо было либо вводить какие-то механизмы контроля за номером страницы (с тем, чтобы он не выходил за размеры таблицы страниц), либо создавать эти таблицы максимально возможного размера. Разработчики пошли по пути, при котором размер страницы все же небольшой (он выбран равным $2^{12} = 4096 = 4K$), а поле номера страницы величиной в 20 битов, в свою очередь, разбивается на два поля и осуществляется двухэтапная (двухшаговая) страничная трансляция.

Для описания каждой страницы создается соответствующий дескриптор. Длина дескриптора выбрана равной 32 битам: 20 битов линейного адреса определяют номер страницы (по существу – её адрес, поскольку добавление к нему (приписывание в качестве младших разрядов) 12 нулей приводит к определению начального адреса страницы), а остальные биты разбиты на следующие поля, которые изображены на рис. 3.7. Как видно, три бита дескриптора зарезервированы для использования системными программистами при разработке подсистемы организации виртуальной памяти. С этими битами микропроцессор сам не работает.

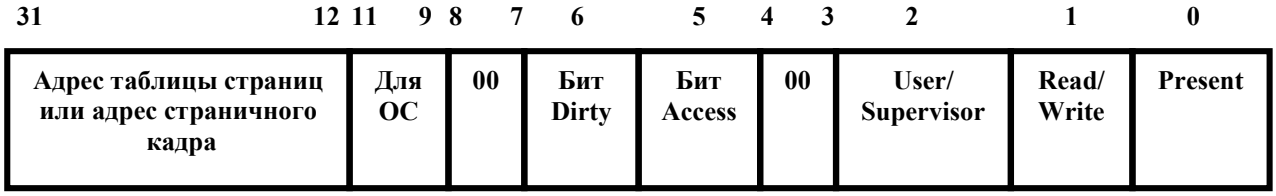

Рис.3.7. Дескриптор страницы

Прежде всего, микропроцессор анализирует самый младший бит дескриптора – бит присутствия, ибо если поле *present* равно нулю, то это означает отсутствие данной страницы в оперативной памяти, и такая ситуация влечет прерывание в работе процессора с передачей управления соответствующей программе, которая должна будет загрузить затребованную страницу. Бит *dirty* – «грязный» – предназначен для отметки, что данную страницу модифицировали и при замещении этого страничного кадра следующим её необходимо сохранить во внешней памяти. Бит обращения (*access*) свидетельствует о том, что к данной таблице или странице осуществлялся доступ. Он используется для определения страницы, которая будет участвовать в замещении при использовании дисциплин LRU или LFU. Наконец, первый и второй биты используются для защиты памяти.

Старшие 10 битов линейного адреса определяют номер таблицы страниц (page table entry, PTE), из которой посредством вторых 10 битов линейного адреса выбирается соответствующий дескриптор виртуальной страницы. И уже из этого дескриптора выбирается номер физической страницы, если данная виртуальная страница отображена сейчас на оперативную память. Эта схема определения физического адреса по линейному изображена на рис. 3.8.

Первая таблица, которую мы индексируем первыми (старшими) 10 битами линейного адреса, названа таблицей каталогов таблиц страниц (page directory entry, PDE). Её адрес в оперативной памяти определяется старшими 20 битами управляющего регистра CR3.

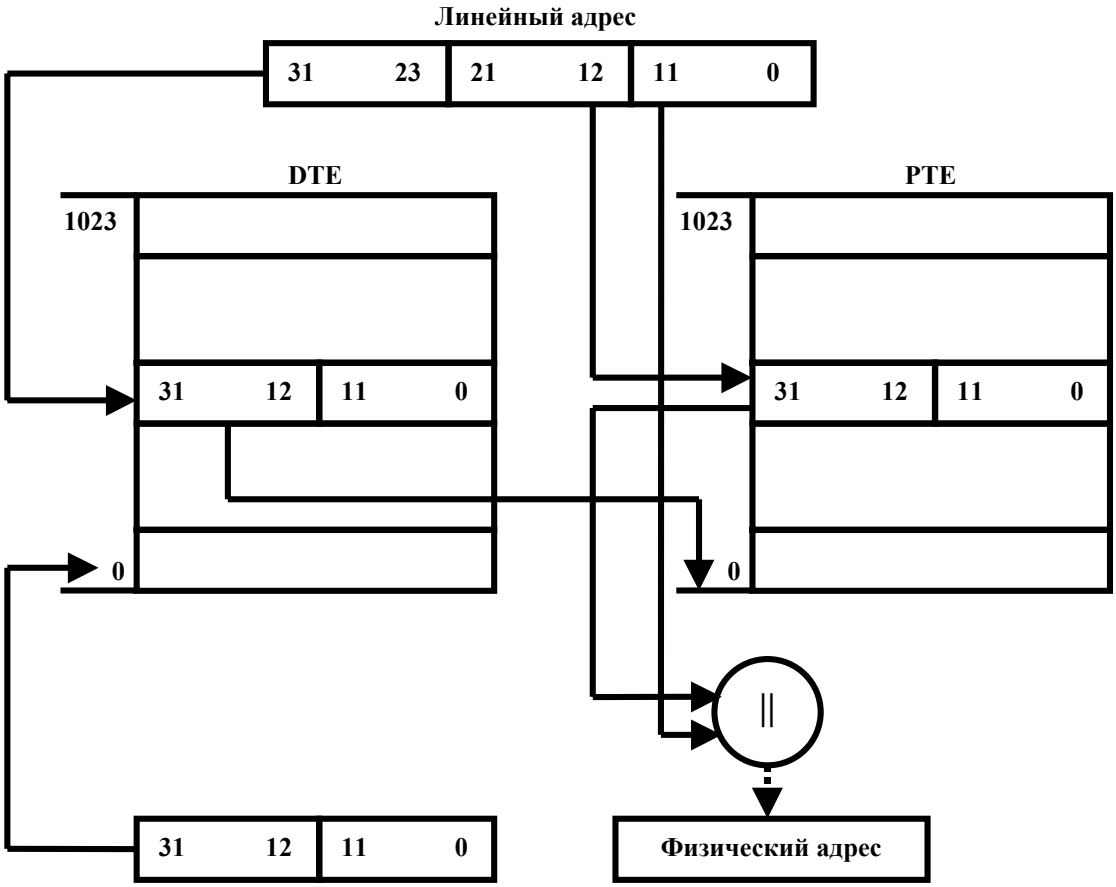

Рис. 3.8. Трансляция линейного адреса в микропроцессорах i80x86

Каждая из таблиц PDE и PTE состоит из 1024 элементов (2^{10} = l024). В свою очередь, каждый элемент (дескриптор страницы) имеет длину 4 байта (32 бита), поэтому размер этих таблиц как раз соответствует размеру страницы.

Оценим теперь эту двухшаговую схему трансляции с позиций расхода памяти. Каждый дескриптор описывает страницу размером 4 Кбайт. Следовательно, одна таблица страниц, содержащая 1024 дескриптора, описывает пространство памяти в 4 Мбайт. Если наша задача пользуется виртуальным адресным пространством, например, в 50 Мбайт (предположим, что речь идёт о некотором графическом редакторе, который обрабатывает изображение, состоящее из большого количества пик-

селов[1]), то для описания этой памяти необходимо иметь 14 страниц, содержащих таблицы PTE. Кроме этого, нам потребуется для этой задачи ещё одна таблица PDE (тоже размером в одну страницу), в которой 14 дескрипторов будут указывать на местонахождение упомянутых таблиц PTE. Остальные дескрипторы PDE могут быть не задействованы. Итого, для описания 50 Мбайт адресного пространства задачи потребуется всего 15 страниц, то есть 60 Кбайт памяти, что можно считать приемлемым.

Если бы не был использован такой двухшаговый механизм трансляции, то потери памяти на описание адресного пространства могли бы составить $4(\text{Кбайт})\cdot 2^{10} = 4$ (Мбайт)! Очевидно, что это уже неприемлемое решение.

Итак, микропроцессор для каждой задачи, для которой у него есть TSS, позволяет иметь таблицу PDE и некоторое количество PTE. Поскольку это дает возможность адресоваться к любому байту из 2^{32}, а шина адреса как раз и позволяет использовать физическую память с таким объёмом, то можно как бы отказаться от сегментного способа адресации. Другими словами, если считать, что задача состоит из одного единственного сегмента, который, в свою очередь, разбит на страницы, то фактически мы получаем только один страничный механизм работы с виртуальной памятью. Этот подход получил название «плоской памяти». При использовании плоской модели памяти упрощается создание и операционных систем, и систем программирования. Кроме этого, уменьшаются расходы памяти для поддержки системных информационных структур. Поэтому в абсолютном большинстве современных 32-разрядных ОС, создаваемых для микропроцессоров i80x86, используется плоская модель памяти.

Режим виртуальных машин для исполнения приложений реального режима

Разработчики рассматриваемого семейства микропроцессоров в своем стремлении обеспечить максимально возможную совместимость архитектуры пошли не только на то, чтобы обеспечить возможность программам, созданным для первых

[1] Напомним, что термин *пиксел* происходит от английского picture element (графический элемент). Множество пикселов образуют изображение.

16-разрядных ПК, без проблем выполняться на компьютерах с более поздними моделями микропроцессоров за счёт введения реального режима работы. Они также обеспечили возможность выполнения 16-разрядных приложений реального режима при условии, что сам процессор при этом функционирует в защищённом режиме работы и операционная система, используя соответствующие аппаратные средства микропроцессора, организует мультипрограммный (мультизадачный) режим. Другими словами, микропроцессоры i80x86 поддерживают возможность создания операционных сред реального режима при работе микропроцессора в защищённом режиме. Если условно назвать 16-разрядные приложения DOS-приложениями (поскольку в абсолютном большинстве случаев это именно так), то можно сказать, что введена поддержка для организации виртуальных DOS-машин, работающих вместе с обычными 32-битовыми приложениями защищённого режима. Это даже нашло отражение в названии режима работы микропроцессоров i80x86 – режим виртуального процессора i8086, иногда (для краткости) его называют режимом V86 или просто *виртуальным режимом,* – при котором в защищённом режиме работы может исполняться код DOS-приложения. Мультизадачность при выполнении нескольких программ реального режима будет поддержана аппаратными средствами защищённого режима.

Переход в виртуальный режим осуществляется посредством изменения бита VM (virtual mode) в регистре EFLAGS. Когда процессор находится в виртуальном режиме, для адресации памяти используется схема реального режима работы – (сегмент: смещение) с размером сегментов до 64 Кбайт, которые могут располагаться в адресном пространстве размером в 1 Мбайт, однако полученные адреса считаются не физическими, а линейными. В результате применения страничной трансляции осуществляется отображение виртуального адресного пространства 16-битового приложения на физическое адресное пространство. Это позволяет организовать параллельное выполнение нескольких задач, разработанных для реального режима, да ещё и совместно с обычными 32-битовыми приложениями, требующих защищённого режима работы. Естественно, что для обработки прерываний, возникающих при выполнении 16-битовых приложений в виртуальном режиме,

процессор возвращается из этого режима в обычный защищённый режим. В противном случае невозможно было бы организовать полноценную виртуальную машину. Очевидно, что обработчики прерываний для виртуальной машины должны эмулировать работу подсистемы прерываний процессора i8086. Другими словами, прерывания отображаются в операционную систему, работающую в защищённом режиме, и уже основная ОС моделирует работу операционной среды выполняемого приложения.

Вопрос, связанный с операциями ввода/вывода, которые недоступны для обычных приложений (см. следующую главу), решается аналогично. При попытке выполнить недопустимые команды ввода/вывода возникают прерывания, и необходимые операции выполняются операционной системой, хотя задача об этом и «не подозревает». При выполнении команд IN, OUT, INS, OUTS, CLI, STI процессор, находящийся в виртуальном режиме и исполняющий код на уровне привилегий третьего (самого нижнего) кольца защиты, за счёт возникающих вследствие этого прерываний переводится на выполнение высоко привилегированного кода операционной системы.

Таким образом, ОС может полностью виртуализировать ресурсы компьютера: и аппаратные[1], и программные, создавая иную полноценную операционную среду; при существовании так называемых нативных приложений, создаваемых по собственным спецификациям данной ОС. Очень важным моментом для организации полноценной виртуальной машины является реализация виртуализации не только программных, но и аппаратных ресурсов. Так, например, в ОС Windows NT эта задача выполнена явно неудачно, тогда как в OS/2 имеется полноценная виртуальная машина как для DOS-приложений, так и для приложений, работающих в среде спецификаций Win 16. Правда, в последнее время это уже перестало быть актуальным, поскольку появилось большое количество приложений, работающих по спецификациям Win32 API.

[1] Речь идёт о памяти, портах ввода/вывода, системе обработки прерываний и других архитектурных особенностях.

Защита адресного пространства задач

Для возможности создания надёжных мультипрограммных ОС в процессорах семейства i80x86 имеется несколько механизмов защиты. Это и разделение адресных пространств задач, и введение уровней привилегий для сегментов кода и сегментов данных. Все это позволяет обеспечить как защиту задач друг от друга, так и защиту самой операционной системы от прикладных задач, защиту одной части ОС от других её компонентов, защиту самих задач от некоторых своих собственных ошибок.

Защита адресного пространства задач осуществляется относительно легко за счёт того, что каждая задача может иметь свое собственное локальное адресное пространство. Операционная система должна корректно манипулировать таблицами трансляции сегментов (дескрипторными таблицами) и таблицами трансляции страничных кадров. Сами таблицы дескрипторов как сегменты данных (а соответственно, в свою очередь, и как страничные кадры) относятся к адресному пространству операционной системы и имеют соответствующие привилегии доступа; исправлять их задачи не могут. Этими информационными структурами процессор пользуется сам, на аппаратном уровне, без возможности их читать и редактировать из пользовательских приложений. Если используется модель плоской памяти, то возможность микропроцессора контролировать обращения к памяти только внутри текущего сегмента фактически не используется, и остается в основном только механизм отображения страничных кадров. Выход за пределы страничного кадра невозможен, поэтому фиксируется только выход за пределы своего сегмента. В этом случае приходится полагаться только на систему программирования, которая должна корректно распределять программные модули в пределах единого неструктурированного адресного пространства задачи. Поэтому при создании многопоточных приложений, когда каждая задача (в данном случае – поток) может испортить адресное пространство другой задачи, эта проблема становится очень сложной, особенно если не использовать системы программирования на языках высокого уровня.

Однако для организации взаимодействия задач, имеющих разные виртуальные адресные пространства, необходимо, как мы уже говорили, иметь общее адресное пространство. И здесь, для обеспечения защиты самой ОС, а значит, и повышения надёжности всех вычислений, используется механизм защиты сегментов с помощью уровней привилегий.

Уровни привилегий для защиты адресного пространства задач

Для того чтобы запретить пользовательским задачам модифицировать области памяти, принадлежащие самой ОС, необходимо иметь специальные средства. Одного разграничения адресных пространств через механизм сегментов мало, ибо можно указывать различные значения адреса начала сегмента и тем самым получать доступ к чужим сегментам. Другими словами, необходимо в явном виде разграничивать системные сегменты данных и кода от сегментов, принадлежащих пользовательским программам. Поэтому были введены два основных режима работы процессора: пользователя и супервизора. Большинство современных процессоров имеют по крайней мере два этих режима. Так, в *режиме супервизора* программа может выполнять все действия и иметь доступ по любым адресам, тогда как в *пользовательском режиме* должны быть ограничения, с тем, чтобы обнаруживать и пресекать запрещенные действия, перехватывая их и передавая управление супервизору ОС. Часто в пользовательском режиме запрещается выполнение команд ввода/вывода и некоторых других, чтобы гарантировать, что только ОС выполняет эти операции. Можно сказать, что эти два режима имеют разные уровни привилегий.

В микропроцессорах i80x86 имеются не два, а четыре *уровня привилегий.* Часто уровни привилегий называют кольцами защиты, поскольку это иногда помогает объяснить принцип действия самого механизма; поэтому говорят, что некоторый программный модуль «исполняется в кольце защиты с таким-то номером». Для указания уровня привилегий используются два бита, поэтому код 00 обозначает самый высший уровень, а код $11_{(2)}$ (=3) – самый низший. Самый высокий уровень привилегий предназначен для операционной системы (прежде всего, для ядра ОС),

самый низкий – для прикладных задач пользователя. Промежуточные уровни привилегий введены для большей свободы системных программистов в организации надёжных вычислений при создании ОС и иного системного ПО. Предполагалось, что уровень с номером (кодом) 1 может быть использован, например, для системного сервиса – программ обслуживания аппаратуры, драйверов, работающих с портами ввода/вывода. Уровень привилегий с кодом 2 может быть использован для создания пользовательских интерфейсов, систем управления базами данных и т. п., то есть для реализации специальных системных функций, которые по отношению к супервизору ОС ведут себя как обычные приложения. Так, например, система OS/2 использует три уровня привилегий: с нулевым уровнем привилегий исполняется код самой ОС, на втором уровне исполняются системные процедуры подсистемы ввода/вывода, на третьем уровне исполняются прикладные задачи пользователей. Однако чаще всего на практике используются только два уровня – нулевой и третий. Таким образом, упомянутый режим супервизора для микропроцессоров i80x86 соответствует выполнению кода с уровнем привилегий 0 (его обозначают так: PL0[1]). Подводя итог, можно констатировать, что именно уровень привилегий задач определяет, какие команды в них можно использовать и какое подмножество сегментов и/или страниц в их адресном пространстве они могут обрабатывать.

Основными системными объектами, которыми манипулирует процессор при работе в защищённом режиме, являются дескрипторы. Дескрипторы сегментов содержат информацию об уровне привилегий соответствующего сегмента кода или данных. Уровень привилегии исполняющейся задачи определяется значением поля привилегий, находящегося в дескрипторе её текущего кодового сегмента. Напомним, что в каждом дескрипторе сегмента (см. рис.3.3) имеется поле DPL в байте прав доступа, которое и определяет уровень привилегии связанного с ним сегмента. Таким образом, поле DPL текущего сегмента кода становится полем CPL. При обращении к какому-нибудь сегменту в соответствующем селекторе указывается запрашиваемый уровень привилегий RPL[2] (см. рис. 3.4).

[1] *PL* (privilege level) – уровень привилегий.
[2] *RPL* (requested privilege level) – запрашиваемый уровень привилегий. Поле RPL определяется программистом (системой программирования). В отличие от поля DPL поле RPL легко может быть изменено.

В пределах одной задачи используются сегменты с различным уровнем привилегии и в определенные моменты времени выполняются или обрабатываются сегменты с соответствующими им уровнями привилегии. Механизм проверки привилегий работает в ситуациях, которые можно назвать межсегментными переходами (обращениями). Это доступ к сегменту данных или стековому сегменту, межсегментные передачи управления в случае прерываний (и особых ситуаций), при использовании команд CALL, JMP, INT, IRET, RET. В таких межсегментных обращениях участвуют два сегмента: целевой сегмент (к которому мы обращаемся) и текущий сегмент кода, из которого идёт обращение.

Процессор сравнивает упомянутые значения CPL, RPL, DPL и на основе понятия *эффективного уровня привилегий*[1] (EPL=max(RPL, DPL)) ограничивает возможности доступа к сегментам по следующим правилам, в зависимости от того, идёт ли речь об обращении к коду или к данным.

При доступе к сегментам данным проверяется условие CPL ≤ EPL. Нарушение этого условия вызывает так называемую особую ситуацию ошибки защиты и возникает прерывание. Уровень привилегии сегмента данных, к которому осуществляется обращение, должен быть таким же, как и текущий уровень, или меньше его. Обращение к сегменту с более высоким уровнем привилегии воспринимается как ошибка, так как существует опасность изменения данных с высоким уровнем привилегий в программе с низким уровнем привилегии. Доступ к данным с меньшим уровнем привилегии разрешается.

Если целевой сегмент является сегментом стека, то правило проверки имеет вид CPL=DPL=RPL.

В случае его нарушения также возникает исключение. Поскольку стек может использоваться в каждом сегменте кода и всего имеются четыре уровня привилегий кода, то используются и четыре стека. Сегмент стека, адресуемый регистром SS, должен иметь тот же уровень привилегий, что и текущий сегмент кода.

[1] *EPL* (effective privilege level) – эффективный уровень привилегий. Значение эффективного уровня привилегий определяется минимальной привилегией, то есть как максимальное значение из двух: RPL и DPL.

Правила для передачи управления, то есть когда осуществляется межсегментный переход с одного сегмента кода на другой сегмент кода, несколько сложнее. Если для перехода с одного сегмента данных на другой сегмент данных считается допустимым обрабатывать менее привилегированные сегменты, то передача управления из высоко привилегированного кода на менее привилегированный код должна контролироваться дополнительно. Другими словами, код операционной системы не должен доверять коду прикладных задач. И обратно, нельзя просто так давать задачам возможность исполнять высоко привилегированный код, хотя потребность в этом всегда имеется (ведь многие функции, в том числе и функции ввода/вывода, считаются привилегированными и должны выполняться только самой ОС). Для реализации возможностей передачи управления в сегменты кода с иными уровнями привилегий введен механизм шлюзования, который мы вкратце рассмотрим ниже. Итак, если DPL = CPL, то переход в другой сегмент кода возможен. Более подробное рассмотрение затронутых вопросов по замыслу авторов выходит за рамки настоящего учебника (для получения более детальных сведений по этому и некоторым другим вопросам особенностей архитектуры микропроцессоров i80x86 рекомендуется обратиться к материалам [1, 8]). Здесь мы рассмотрим только основные идеи.

Механизм шлюзов для передачи управления на сегменты кода с другими уровнями привилегий

Поскольку межсегментные переходы контролируются с использованием уровней привилегий и существует потребность в передаче управления с одного уровня привилегий на другой уровень, в микропроцессорах i80x86 реализован *механизм шлюзов,* который мы поясним с помощью рис.3.9. Шлюзование позволяет организовать обращение к так называемым подчинённым сегментам кода, которые выполняют часто встречающиеся функции и должны быть доступны многим задачам, располагающимся на том же или более низком уровне привилегии.

Помимо дескрипторов сегментов системными объектами, с которыми работает микропроцессор, являются специальные системные дескрипторы, названные шлюзами или вентилями. Главное различие между дескриптором сегмента и шлюзом вызова заключается в том, что содержимое дескриптора указывает на сегмент в

памяти, а шлюз обращается к дескриптору. Другими словами, если дескриптор служит механизмом отображения памяти, то шлюз служит механизмом перенаправления.

Для получения доступа к более привилегированному коду задача должна обратиться к нему не непосредственно (путем указания дескриптора этого кода), а обращением к шлюзу этого сегмента (рис. 3.10).

В этом дескрипторе вместо адреса сегмента указываются селектор, позволяющий найти дескриптор искомого сегмента кода, и адрес (смещение назначения), с которого будет выполняться подчиненный сегмент, то есть полный 32-битный адрес. Формат *дескриптора шлюза* приведен на рис. 3.11. Адресовать шлюз вызова можно с помощью команды CALL[1]. По существу, дескрипторы щлюзов вызова не являются дескрипторами (сегментов), но они могут быть расположены среди обычных дескрипторов в дескрипторных таблицах процесса. Смещение, указываемое в команде перехода на другой сегмент (FAR CALL), игнорируется, и фактически осуществляется переход на команду, адрес которой определяется через смещение из шлюза вызова. Этим гарантируется попадание только на разрешенные точки входа в подчиненные сегменты.

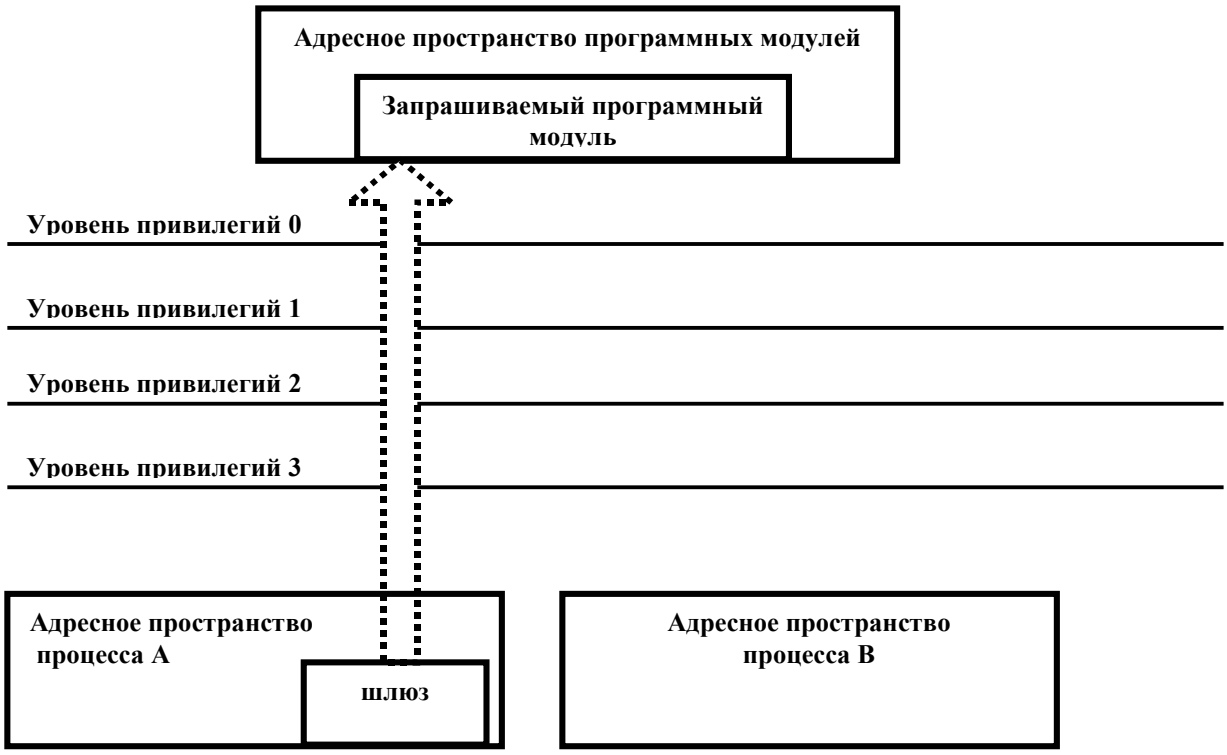

Рис. 3.9. Механизм шлюзов для перехода на другой уровень привилегий

[1] FAR CALL – межсегментный вызов процедуры.

Введены следующие правила использования шлюзов:

♦ значение DPL шлюза вызова должно быть больше или равно значению текущего уровня привилегий CPL;

♦ значение DPL шлюза вызова должно быть больше или равно значению поля RPL селектора шлюза;

♦ значение DPL шлюза вызова должно быть больше или равно значению DPL целевого сегмента кода;

♦ значение DPL целевого сегмента кода должно быть меньше или равно значению текущего уровня привилегий CPL.

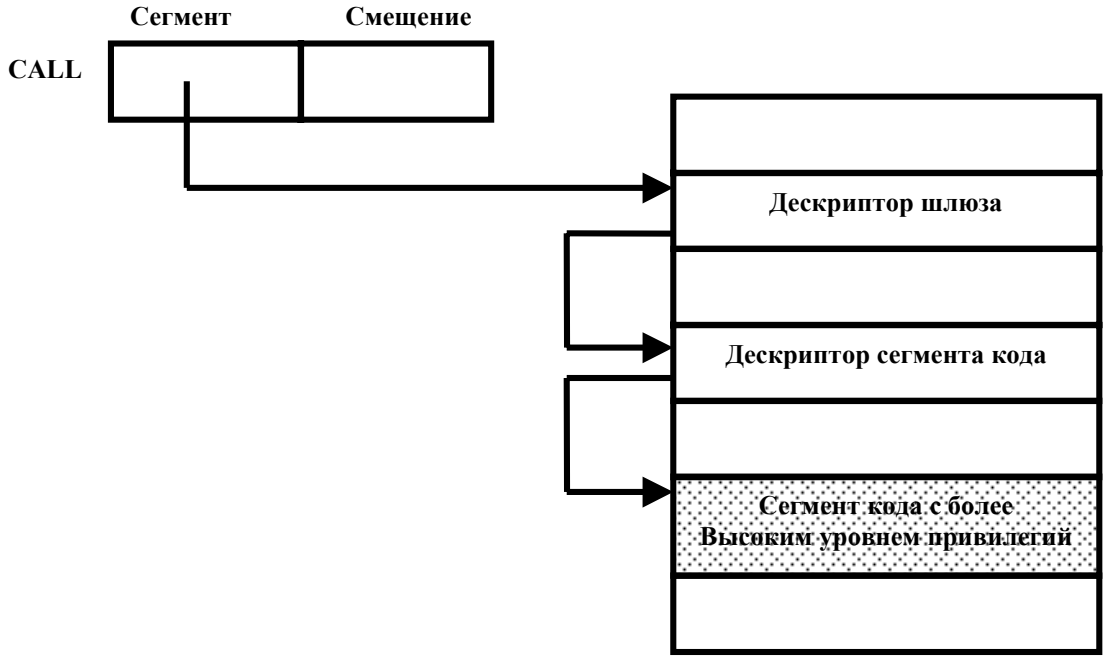

Рис.3.10. Переход на сегмент более привилегированного кода

Рис.3.11.Формат дескриптора шлюза

Требование наличия и доступности шлюза вызова для перехода на более привилегированный код ограничивает менее привилегированный код заданным набором точек входа в код с большей привилегией. Так как шлюзы вызова являются элементами в дескрипторных таблицах (а мы говорили, что их не только можно, но и желательно там располагать), то менее привилегированная программа не может создать дополнительных (а значит, и неконтролируемых) шлюзов. Таким образом, рассмотренный механизм шлюзов дает следующие преимущества в организации среды для выполнения надёжных вычислений:

♦ привилегированный код надёжно защищён и вызывающие его программы не могут его разрушить. Естественно, что такой системный код должен быть особенно тщательно отлаженным, не содержать ошибок, быть максимально эффективным;

♦ шлюзы межсегментных переходов для вызова системных функций делают эти самые системные функции невидимыми для программных модулей, расположенных на внешних (более низких) уровнях привилегий;

♦ шлюзы межсегментных переходов для вызова системных функций делают эти самые системные функции невидимыми для программных модулей, расположенных на внешних (более низких) уровнях привилегий;

♦ поскольку вызывающая программа непосредственно адресует только шлюз вызова, реализуемые вызываемым модулем (сегментным кодом) функции можно изменить или переместить в адресном пространстве, не затрагивая интерфейс со шлюзом;

♦ программные модули вызываются с более привилегированного уровня.

Изложенный коротко аппаратный механизм защиты по привилегиям оказывается довольно сложным и жёстким. Однако поскольку все практические ситуации учесть в схемах микропроцессора невозможно, то при разработке процедур операционных систем и иного высоко привилегированного кода следует придерживаться приведенных ниже рекомендаций, заимствованных из книги [22].

Основной риск связан с передачей управления через шлюз вызова более привилегированной процедуре. Нельзя предоставлять вызывающей программе ника-

ких преимуществ, вытекающих из-за временного повышения привилегий. Это замечание особенно важно для процедур нулевого уровня привилегий (PL0–процедур).

Вызывающая программа может нарушить работу процедуры, передавая ей «плохие» параметры. Поэтому целесообразно как можно раньше проконтролировать передаваемые процедуре параметры. Шлюз вызова сам по себе не проверяет значений параметров, которые копируются в новый стек, поэтому достоверность каждого передаваемого параметра должна контролировать вызванная процедура. Вот некоторые способы контроля передаваемых параметров.

1 Следует проверять счетчики циклов и повторений на минимальные и максимальные значения.

2 Необходимо проверить 8- и 16-битные параметры, передаваемые в 32–битных регистрах. Когда процедуре передаётся короткий параметр, его следует расширить со знаком или нулём для заполнения всего 32-битного регистра.

3 Следует стремиться свести к минимуму время работы процессора с запрещёнными прерываниями. Если процедуре требуется запрещать прерывания, необходимо, чтобы вызывающая программа не могла влиять на время нахождения процессора с запрещенными прерываниями (флажок IF=0).

4 Процедура никогда не должна воспринимать как параметр код или указатель кода.

5 В операциях процессора следует явно задавать состояние флажка направления DF для цепочечных команд.

6 Заключительная команда RET или RETn в процедуре должна точно соответствовать полю счетчика WC шлюза вызова; при этом n = 4x(WC[1]), так как счётчик задает число двойных слов, а n соответствует байтам.

7 Не следует применять шлюзы вызовов для функций, которым передаётся переменное число параметров (см. рекомендацию 6). При необходимости нужно воспользоваться счётчиком и указателем параметров.

[1] WC (Word Counter) – счётчик слов, см. рис. 3.11.

8 Функции не могут возвращать значения в стеке (см. рекомендацию 6), так как после возврата стеки процедуры и вызывающей программы находятся точно в таком состоянии, в каком они были до вызова.

9 В процедуре следует сохранять и восстанавливать все сегментные регистры. Иначе, если какой-либо сегментный регистр привлекался для адресации данных, недоступных вызывающей программе, процессор автоматически загрузит в него пустой селектор.

Рекомендуется контролировать все обращения к памяти. Нетрудно представить себе, что PL3-программа передаст PL0-процедуре указатель селектор: смещение и запросит считывание или запись нескольких байтов по этому адресу. Типичным примером может служить процедура дискового ввода/вывода, которая воспринимает как параметр системный номер файла, счётчик байт и адрес, по которому записываются данные с диска. Хотя PL0-процедура имеет привилегии для производства такой операции, но у PL3-программы разрешения на это может не быть.

Система прерываний 32-разрядных микропроцессоров i80x86

В микропроцессорах семейства i80x86 система прерываний построена таким образом, чтобы, с одной стороны, обеспечить возможность создавать эффективные и надёжные мультипрограммные операционные системы, которые должны функционировать в защищённом режиме, а с другой стороны – обеспечить возможность выполнять программы, разработанные для реального режима. Рассмотрим коротко оба режима.

Работа системы прерываний в реальном режиме работы процессора

В реальном режиме работы система прерываний использует понятие *вектора прерывания.* Термин «вектор прерываний» используется потому, что для указания адреса используется не одно значение, а два, то есть мы имеем дело не со скалярной величиной, а с «векторной».

Итак, каждый вектор прерываний состоит из 4 байтов или 2 слов; первые два содержат новое значение для регистра IP, а следующие два – новое значение регистра CS. *Таблица векторов прерываний* занимает 1024 байта. Таким образом, в ней

может быть задано 256 векторов прерываний. В процессоре i8086 эта таблица располагается на адресах 00000-003FFH. Расположение этой таблицы в процессорах i80286 и старше определяется значением регистра IDTR – Interrupt Descriptor Table Register. При включении или сбросе процессора i80x86 этот регистр обнуляется. Однако при необходимости можно в регистре IDTR указать смещение и, таким образом, перейти на новую таблицу векторов прерываний.

Таблица векторов прерываний заполняется (инициализируется) при запуске системы, но в принципе может быть изменена или перемещена.

Каждый вектор прерывания имеет свой номер, называемый номером прерывания, который указывает его место в таблице. Этот номер, помноженный на четыре (сдвиг на два разряда влево и заполнение освободившихся битов нулями), и сложенный с содержимым регистра IDTR, дает абсолютный адрес первого байта вектора в оперативной памяти.

Подобно вызову процедуры, прерывание заставляет микропроцессор сохранить в стеке информацию для последующего возврата, а затем перейти к группе команд, адрес которых определяется вектором прерывания. Таким образом, прерывание вызывает косвенный переход к своей подпрограмме обработки за счёт получения её адреса из вектора прерывания.

В IBM PC, как и в других вычислительных системах, прерывания бывают двух видов: внутренние и внешние.

Внутренние прерывания, как мы уже знаем, возникают в результате работы процессора. Они возникают в ситуациях, которые нуждаются в специальном обслуживании, или при выполнении специальных инструкций – INT или INT0. Это следующие прерывания:

♦ прерывание при делении на ноль; номер прерывания – 0;

♦ прерывание по флагу TF (trap flag[1]). В этом случае прерывание обычно используется специальными программами отладки типа DEBUG. Номер прерывания – 1;

[1] *Trap Flag* – специальный бит в регистре PSW (program status word, слово состояния программы), который в случае значения TF=1 вызывает «останов» после каждой команды и генерируется прерывание для организации режима отладки с пошаговым выполнением программы. Чаще всего этот регистр в микропроцессорах Intel 80x86 называют *регистром флагов.*

♦ инструкции INT (interrupt – выполнить прерывание с соответствующим номером) и INT0 (interrupt if overflow – прерывание по переполнению). Эти прерывания называются программными.

В качестве операнда команды INT указывается номер прерывания, которое нужно выполнить, например INT 10H. Программные прерывания как средство перехода на соответствующую процедуру были введены для того, чтобы выполнение этой процедуры осуществлялось в привилегированном режиме, а не в обычном пользовательском.

Внешние прерывания возникают по сигналу какого-нибудь внешнего устройства. Существуют два специальных внешних сигнала среди входных сигналов процессора, при помощи которых можно прервать выполнение текущей программы и тем самым переключить работу центрального процессора. Это сигналы NMI (no mask interrupt, немаскируемое прерывание) и INTR (interrupt request, запрос на прерывание). Соответственно, внешние прерывания подразделяются на немаскируемые и маскируемые.

Маскируемые прерывания генерируются контроллером прерываний по заявке определенных периферийных устройств[2]. Контроллер прерываний (его обозначение – i8259A) поддерживает восемь уровней (линий) приоритета; к каждому уровню «привязано» одно периферийное устройство[1]. Маскируемые прерывания часто называют ещё аппаратными прерываниями. В ПК, начиная с IBM PC AT, построенных на базе микропроцессора i80286, используются два контроллера прерываний i8259A; они соединяются каскадным образом. Схема последовательного соединения этих контроллеров изображена на рис. 3.12.

Таким образом, на IBM PC AT предусмотрено 15 линий IRQ[2], часть которых используется внутренними контроллерами системной платы, а остальные заняты стандартными адаптерами либо не используются. Ниже перечислены линии запроса на прерывание, которые мы приводим потому, что каждый специалист по вычислительной технике должен знать основные стандарты ПК. Итак, линии IRQ:

0 – системный таймер;

---

[2] Сигнал запроса на прерывание чаще всего является сигналом готовности внешнего устройства (соответствующего контроллера внешнего устройства) на выполнение следующей команды, связанной с управлением операциями ввода/вывода.

[1] В качестве внешнего периферийного устройства, занимающего одну линию запроса на прерывание, может быть использовано специальное управляющее устройство, которое позволяет разделять эту самую линию запроса между несколькими внешними устройствами.

[2] *IRQ* (Interrupt Request) – запрос на прерывание.

1 – контроллер клавиатуры;

2 – сигнал возврата по кадру (EGA/VGA), на AT соединен с IRQ 9;

3 – обычно COM2/COM4;

4 – обычно COM1/COM3;

5 – контроллер HDD (на первых компьютерах IBM PC XT), обычно свободен на IBM PC AT и используется звуковой картой;

6 – контроллер FDD;

7 – LPT1, многими LPT-контроллерами не используется;

8 – часы реального времени с автономным питанием (RTC – real time clock);

9 – параллельна IRQ 2;

10 – не используется, то есть свободно;

11 – свободно;

12 – обычно контроллер мыши типа PS/2;

13 – математический сопроцессор;

14 – обычно контроллер IDE0 (первый канал);

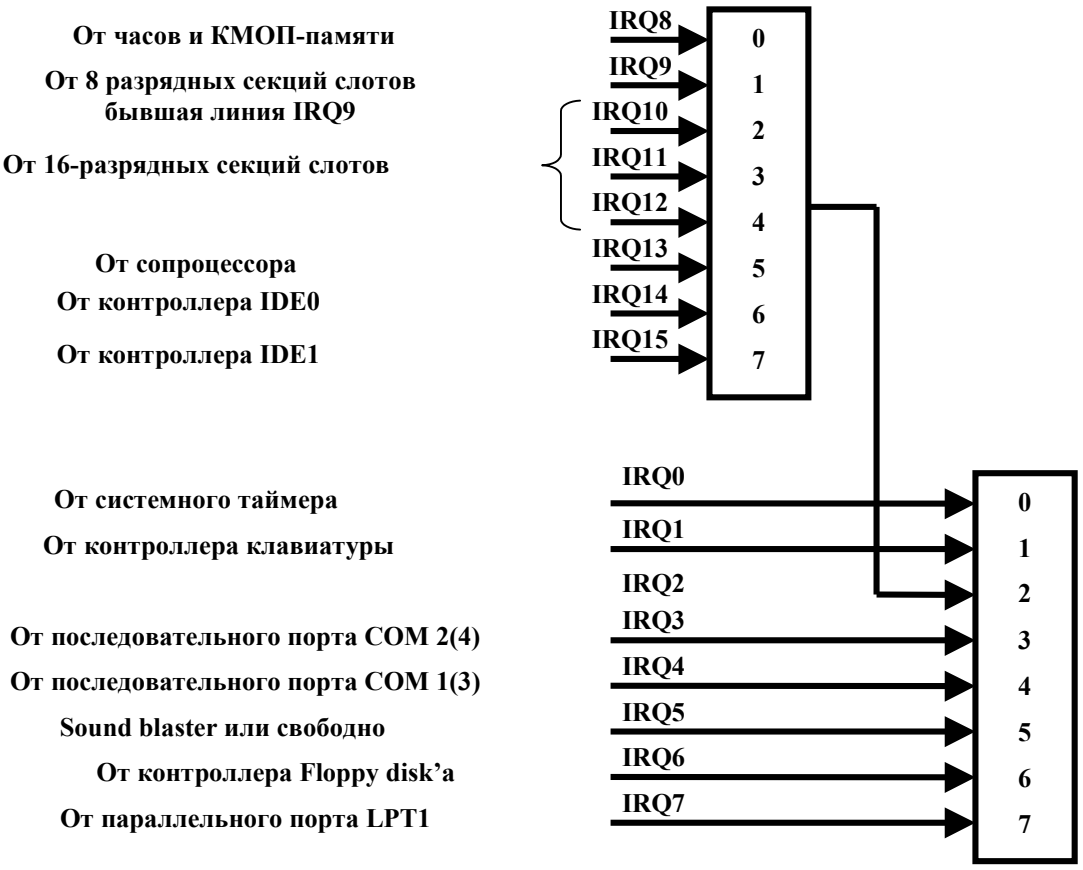

Рис. 3.12. Каскадирование контроллеров прерывания

Как известно, прерывания могут быть инициированы внешним устройством ПК или специальной командой прерывания из программы. В любом случае, если прерывания разрешены, то выполняется следующая процедура:

1 В стек помещается регистр флагов PSW.

2 Флаг включения/выключения прерываний IF и флаг трассировки TF, находящиеся в регистре PSW, обнуляются для блокировки других маскируемых прерываний и исключения пошагового режима исполнения команд.

3 Значения регистров CS и IP сохраняются в стеке вслед за PSW.

4 Вычисляется адрес вектора прерывания, и из вектора, соответствующего номеру прерывания, загружаются новые значения IP и CS.

Когда системная подпрограмма принимает управление, она может снова разрешить маскируемые прерывания командой STI (set interrupt flag, установить флаг прерываний), которая переводит флаг IF в состояние 1, что разрешает микропроцессору вновь реагировать на прерывания, инициируемые внешними устройствами, поскольку стековая организация позволяет вложение прерываний друг в друга.

Закончив работу, подпрограмма обработки прерывания должна выполнить инструкцию IRET (interrupt return), которая извлекает из стека три 16-битовых значения и загружает их в указатель команд IP, регистр сегмента команд CS и регистр PSW соответственно. Таким образом, процессор сможет продолжить работу с того места, где он был прерван.

В случае внешних прерываний процедура перехода на подпрограмму обработки прерывания дополняется следующими шагами:

1 Контроллер прерываний получает заявку от определенного периферийного устройства и, соблюдая схему приоритетов, генерирует сигнал INTR (interrupt request), который является входным для микропроцессора.

2 Микропроцессор проверяет флаг IF в регистре PSW. Если он установлен в 1, то переходим к шагу 3. В противном случае работа процессора не прерывается. Часто говорят, что прерывания замаскированы, хотя правильнее говорить, что они

отключены. Маскируются (запрещаются) отдельные линии запроса на прерывания посредством программирования контроллера прерываний.

3 Микропроцессор генерирует сигнал INTA (подтверждение прерывания). В ответ на этот сигнал контроллер прерывания посылает по шине данных номер прерывания. После этого выполняется описанная нами ранее процедура передачи управления соответствующей программе обработки прерывания.

Номер прерывания и его приоритет устанавливаются на этапе инициализации системы. После запуска ОС пользователь, как мы уже отмечали, может изменить таблицу векторов прерывания, поскольку она ему доступна.

Работа системы прерываний в защищённом режиме работы процессора

В защищённом режиме работы система прерываний действует совершенно иначе. Прежде всего, система прерываний микропроцессора i80x86 при работе в защищённом режиме вместо таблицы векторов, о которой мы говорили выше, имеет дело с таблицей *дескрипторов прерываний* (IDT, interrupt descriptor table). Дело здесь не столько в названии таблицы, сколько в том, что таблица IDT представляет собой не таблицу с адресами обработчиков прерываний, а таблицу со специальными системными структурами данных (дескрипторами), доступ к которой со стороны пользовательских (прикладных) программ невозможен. Только сам микропроцессор (его система прерываний) и код операционной системы могут получить доступ к этой таблице, которая представляет собой специальный сегмент, адрес и длина которого содержатся в регистре IDTR (см. рис. 3.2). Этот регистр аналогичен регистру GDTR в том отношении, что он инициализируется один раз при загрузке системы. Интересно заметить, что в реальном режиме работы регистр IDTR так же указывает адрес таблицы прерываний, но при этом, как и в процессоре i8086, каждый элемент таблицы прерываний (вектор) занимает всего 4 байта и содержит 32-битный адрес в формате селектор: смещение (CS:IP). Начальное значение этого регистра равно нулю, но в него можно занести и другое значение. В этом случае таблица векторов прерываний будет находиться в другом месте оперативной памяти. Естественно, что перед тем, как это сделать (занести в регистр IDTR новое значение), необходимо подготовить саму таблицу векторов. В защищённом режиме работы загрузку регистра IDTR может произвести только код с максимальным уровнем привилегий.

Каждый элемент в таблице дескрипторов прерываний, о которой мы говорим уже в защищённом режиме, представляет собой 8-байтовую структуру, более похожую на дескриптор шлюза (gate), нежели на дескриптор сегмента.

Как мы уже знаем, в зависимости от причины прерывания процессор автоматически индексирует таблицу прерываний и выбирает соответствующий элемент, с помощью которого и осуществляется перенаправление в исполнении кода, то есть передача управления на обработчик прерывания. Однако таблица IDT содержит только шлюзы, а не дескрипторы сегментов кода, поэтому фактически получается косвенная адресация, но с использованием рассмотренного ранее механизма защиты с помощью уровней привилегии. Благодаря этому пользователи уже не могут сами изменить обработку прерываний, которая предопределяется системным программным обеспечением.

Дескриптор прерываний может принадлежать к одному из трех типов:

- ♦ коммутатор прерывания (interrupt gate);
- ♦ коммутатор перехвата (trap gate);
- ♦ коммутатор задачи (task gate).

При обнаружении запроса на прерывание и при условии, что прерывания сейчас разрешены, процессор действует в зависимости от типа дескриптора (коммутатора), соответствующего номеру прерывания. Первые два типа дескриптора прерываний вызывают переход на соответствующие сегменты кода, принадлежащие виртуальному адресному пространству текущего вычислительного процесса. Поэтому про них говорят, что обработка прерываний по этим дескрипторам осуществляется *под контролем текущей задачи.* Последний тип дескриптора – коммутатор задачи – вызывает полное переключение процессора на новую задачу со сменой всего контекста в соответствии с сегментом состояния задачи (TSS). Рассмотрим эти варианты.

Обработка прерываний в контексте текущей задачи

Рассмотрим рис. 3.13, поясняющий обработку прерывания в контексте текущей задачи. При возникновении прерывания процессор по номеру прерывания индексирует таблицу IDT, то есть адрес соответствующего коммутатора определяется путем сложения содержимого поля адреса в регистре IDTR и номера прерывания, умноженного на 8 (справа к номеру прерывания добавляются три нуля). Полученный дескриптор анализируется, и если его тип соответствует коммутатору *trap gate* или коммутатору *interrupt gate*, то выполняются следующие действия.

1 В стек на уровне привилегий текущего сегмента кода помещаются:

♦ значения SS и SP, если уровень привилегий в коммутаторе выше уровня привилегий ранее исполнявшегося кода;

♦ регистр флагов EFLAGS;

♦ регистры CS и IP.

2 Если рассматриваемому прерыванию соответствовал коммутатор *interrupt gate*, то запрещаются прерывания (флаг IF=0 в регистре EFLAGS). В случае коммутатора *trap gate* флаг прерываний не сбрасывается и обработка новых прерываний на период обработки текущего прерывания тем самым не запрещается.

3 Поле селектора из дескриптора прерываний используется для индексирования таблицы дескрипторов задачи. Дескриптор сегмента заносится в теневой регистр, а смещение относительно начала нового сегмента кода определяется полем смещения из дескриптора прерывания.

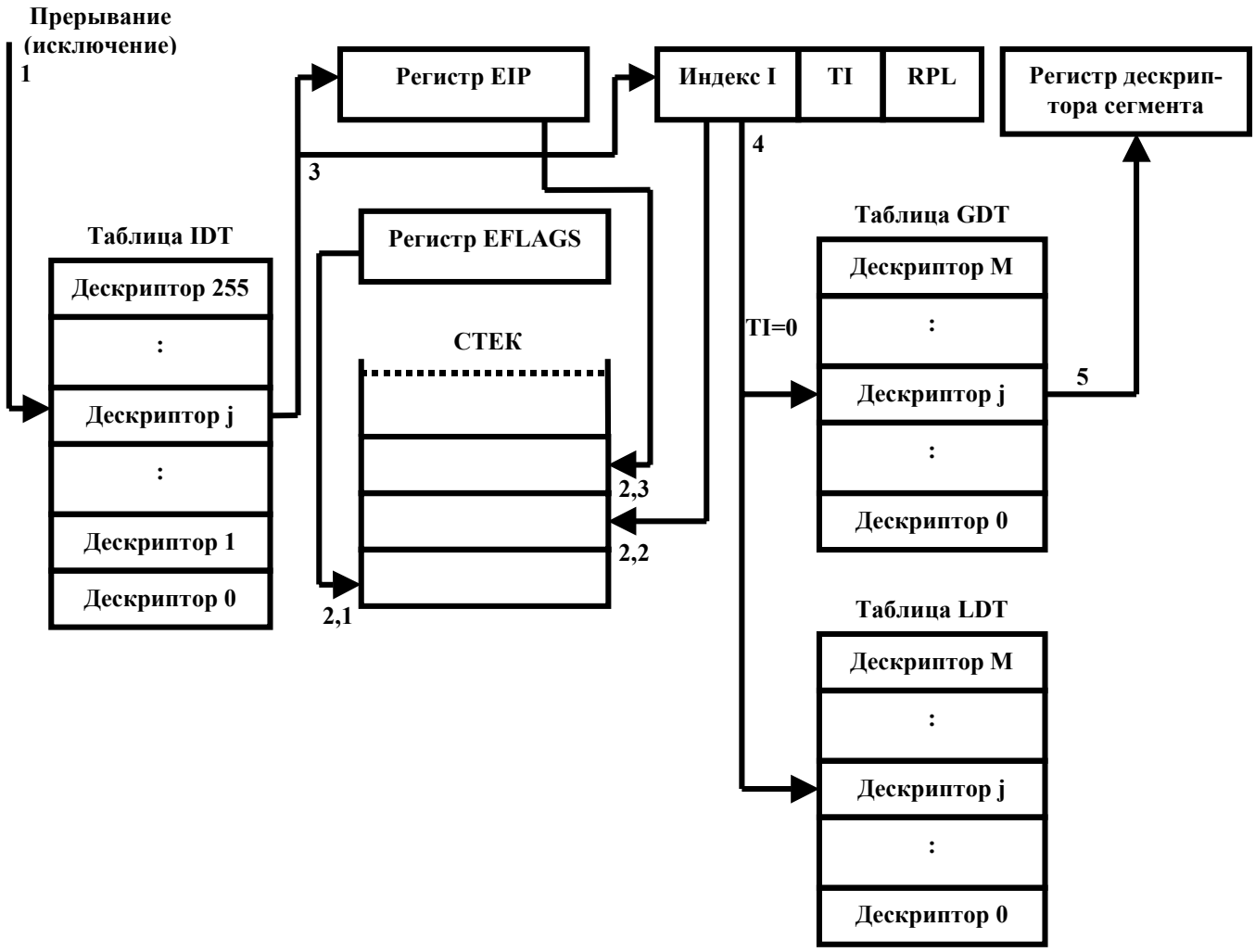

Рис.3.13. Схема передачи управления при прерывании в контексте текущей задачи

Таким образом, в случае обработки прерываний, когда дескриптором прерываний является коммутатор *interrupt gate* или *trap gate*, мы остаемся в том же виртуальном адресном пространстве, и полной смены контекста текущей задачи не происходит. Просто мы переключаемся на исполнение другого (как правило, более привилегированного) кода, но также принадлежащего (или, правильнее сказать, доступного) исполняемой задаче. Этот код создается системными программистами, и прикладные программисты его просто используют. В то же время механизмы защиты микропроцессора позволяют обеспечить недоступность этого кода для его исправления (со стороны приложений, его вызывающих) и недоступность самой таблицы дескрипторов прерываний. Удобнее всего код обработчиков прерываний располагать в общем адресном пространстве, то есть селекторы, указывающие на такой код, должны располагаться в глобальной таблице дескрипторов.

Обработка прерываний с переключением на новую задачу

Совершенно иначе осуществляется обработка прерываний в случае, если дескриптором прерываний является коммутатор задачи. Формат коммутатора задачи (*task gate*) отличается от коммутаторов *interrupt gate* и *trap gate*, прежде всего, тем, что в нем вместо селектора сегмента кода, на который передаётся управление, указывается селектор сегмента состояния задачи (TSS). Это иллюстрируется с помощью рис.3.14. В результате осуществляется процедура перехода на новую задачу с полной сменой контекста, ибо сегмент состояния задачи полностью определяет новое виртуальное пространство и адрес начала программы, а текущее состояние прерываемой задачи аппаратно (по микропрограмме микропроцессора) сохраняется в её собственном TSS.

При этом происходит полное переключение на новую задачу с вложением, то есть выполняются следующие действия:

1 Сохраняются все рабочие регистры процессора в текущем сегменте TSS, базовый адрес этого сегмента берется в регистре TR (см. раздел «Адресация в 32-разрядных микропроцессорах i80x86 при работе в защищённом режиме»).

2 Текущая задача отмечается как занятая.

3 По селектору из *Task Gate* выбирается новый TSS (поле селектора помещается в регистр TR) и загружается состояние новой задачи. Напомним, что загружаются значения регистра LDTR, EFLAGS, восемь регистров общего назначения, указатель команды регистр EIP и шесть сегментных регистров.

4 Устанавливается бит NT (next task).

5 В поле обратной связи TSS помещается селектор прерванной задачи.

6 Значения CS:IP, взятые из нового TSS, позволяют найти и выполнить первую команду обработчика прерывания.

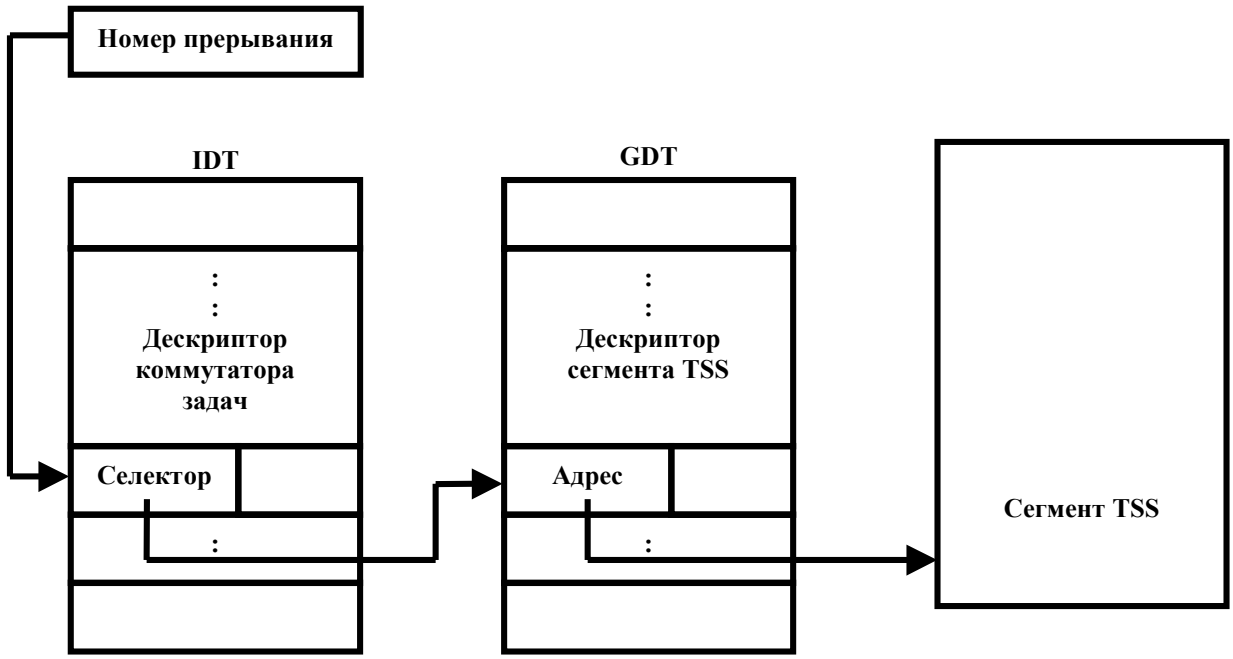

Рис.3.14. Схема передачи управления при прерывании с переключением на новую задачу

Таким образом, коммутатор *task gate* даёт указание процессору произвести переключение задачи, и обработка прерывания осуществляется под контролем отдельной внешней задачи. В каждом сегменте TSS имеется селектор локальной дескрипторной таблицы LDT, поэтому при переключении задачи процессор загружает в регистр LDTR новое значение. Это позволяет обратиться к сегментам кода, которые абсолютно не пересекаются с сегментами кода любых других задач, поскольку именно локальные дескрипторные таблицы обеспечивают эффективное изолирование виртуальных адресных пространств. Новая задача начинает своё выполнение на уровне привилегий, определяемом полем RPL нового содержимого ре-

гистра CS, которое загружается из сегмента TSS. Достоинством этого коммутатора является то, что он позволяет сохранить все регистры процессора с помощью механизма переключения задач, тогда как коммутаторы *trap gate* и *interrupt gate* сохраняют только содержимое регистров IFLAGS, CS и IP и сохранение других регистров возлагается на программиста, разрабатывающего соответствующую программу обработки прерывания.

Справедливости ради следует признать, что, несмотря на возможности коммутатора *task gate*, разработчики современных операционных систем достаточно редко его используют, поскольку переключение на другую задачу требует существенно больших затрат времени, а полное сохранение всех рабочих регистров часто и не требуется. В основном обработку прерываний осуществляют в контексте текущей задачи, так как это приводит к меньшим накладным расходам и повышает быстродействие системы.

Контрольные вопросы и задачи

Вопросы для проверки

1 Как в реальном режиме работы микропроцессоров i80x86 осуществляется преобразование виртуального адреса в физический?

2 Какие механизмы виртуальной памяти используются в защищённом режиме работы микропроцессоров i80x86?

3 Для чего в микропроцессоры i80x86 введен регистр-указатель задачи TR? Какой он разрядности?

4 Как в микропроцессорах i80x86 реализована поддержка сегментного способа организации виртуальной памяти?

5 Что понимается под термином «линейный адрес»? Как осуществляется преобразование линейного адреса в физический? А может ли линейный адрес быть равным физическому?

6 Что дало введение двухшаговой страничной трансляции в механизме страничного способа реализации виртуальной памяти? Как разработчики микропроцессора i80386 решили проблему замедления доступа к памяти, которое при двухшаговом преобразовании адресов очень существенно?

7 Что дало введение виртуального режима? Как в этом режиме осуществляется вычисление физического адреса?

8 Что имеется в микропроцессорах i80x86 для обеспечения защиты адресного пространства задач?

9 Что такое «уровень привилегий»? Сколько уровней привилегий имеется в микропроцессорах i80x86? Для каких целей введено такое количество уровней привилегий?

10 Что такое текущий уровень привилегий? Что такое эффективный уровень привилегий?

11 Объясните правила работы с уровнями привилегий для различных типов сегментов.

12 Поясните механизм шлюзования: для чего он предназначен, как осуществляется передача управления на сегменты кода с другими уровнями привилегий.

13 Расскажите о работе системы прерываний микропроцессоров i80x86 в реальном режиме.

14 В чём заключаются основные принципиальные отличия работы системы прерываний микропроцессоров i80x86 в защищённом режиме по сравнению с реальным режимом?

15 Как осуществляется переход на программу обработки прерываний, если дескриптор прерываний является коммутатором прерываний?

16 Как осуществляется переход на программу обработки прерываний, если дескриптор прерываний является коммутатором перехвата?

17 Как осуществляется переход на программу обработки прерываний, если дескриптор прерываний является коммутатором задачи?

ГЛАВА 4 Управление вводом/выводом и файловые системы

Необходимость обеспечить программам возможность осуществлять обмен данными с внешними устройствами и при этом не включать в каждую двоичную программу соответствующий двоичный код, осуществляющий собственно управление устройствами ввода/вывода, привела разработчиков к созданию системного программного обеспечения и, в частности, самих операционных систем. Программирование задач управления вводом/выводом является наиболее сложным и трудоёмким, требующим очень высокой квалификации. Поэтому код, позволяющий осуществлять операции ввода/вывода, стали оформлять в виде системных библиотечных процедур; потом его стали включать не в системы программирования, а в операционную систему с тем, чтобы в каждую отдельно взятую программу его не вставлять, а только позволить обращаться к такому коду. Системы программирования стали генерировать обращения к этому системному коду ввода/вывода и осуществлять только подготовку к собственно операциям ввода/вывода, то есть автоматизировать преобразование данных к соответствующему формату, понятному устройствам, избавляя прикладных программистов от этой сложной и трудоёмкой работы. Другими словами, системы программирования вставляют в машинный код необходимые библиотечные подпрограммы ввода/вывода и обращения к тем системным программным модулям, которые, собственно, и управляют операциями обмена между оперативной памятью и внешними устройствами. Таким образом, управление вводом/выводом – это одна из основных функций любой ОС.

С одной стороны, в организации ввода/вывода в различных ОС много общего. С другой стороны, реализация ввода/вывода в ОС так сильно отличается от системы к системе, что очень нелегко выделить и описать именно основные принципы реализации этих функций. Проблема усугубляется ещё тем, что в большинстве ныне используемых систем эти моменты вообще, как правило, подробно не описаны, и исключение по этому вопросу касается только системы Linux, для которой *имеются комментированные исходные тексты*. Детально описываются функции API,

реализующие ввод/вывод. Поэтому в этом разделе, самом небольшом по объёму, мы рассмотрим только основные идеи и концепции.

Поскольку внешняя память, как правило, реализуется на таких устройствах ввода/вывода, как накопители на магнитных дисках, мы также рассмотрим логическую структуру диска.

Закончим настоящую главу рассмотрением наиболее распространенных файловых систем.

Основные понятия и концепции организации ввода/вывода в ОС

Как известно, ввод/вывод считается одной из самых сложных областей проектирования операционных систем, в которой сложно применить общий подход из-за изобилия частных методов. Сложность возникает из-за огромного числа устройств ввода/вывода разнообразной природы, которые должна поддерживать ОС. При этом перед создателями ОС встает очень непростая задача – не только обеспечить эффективное управление устройствами ввода/вывода, но и создать удобный и эффективный виртуальный интерфейс устройств ввода/вывода, позволяющий прикладным программистам просто считывать или сохранять данные, не обращая внимание на специфику устройств и проблемы распределения устройств между выполняющимися задачами. Система ввода/вывода, способная объединить в одной модели широкий набор устройств, должна быть универсальной. Она должна учитывать потребности существующих устройств, от простой мыши до клавиатур, принтеров, графических дисплеев, дисковых накопителей, компакт-дисков и даже сетей. С другой стороны, необходимо обеспечить доступ к устройствам ввода/вывода для множества параллельно выполняющихся задач, причем так, чтобы они как можно меньше мешали друг другу.

Поэтому самым главным является следующий принцип: любые операции по управлению вводом/выводом объявляются привилегированными и могут выполняться только кодом самой ОС. Для обеспечения этого принципа в большинстве процессоров даже вводятся режимы пользователя и супервизора. Как правило, в режиме супервизора выполнение команд ввода/вывода разрешено, а в пользова-

тельском режиме – запрещено. Использование команд ввода/вывода в пользовательском режиме вызывает исключение[1] и управление через механизм прерываний передаётся коду ОС. Хотя возможны и более сложные системы, в которых в ряде случаев пользовательским программам разрешено непосредственное выполнение команд ввода/вывода.

Еще раз подчеркнем, что, прежде всего, мы говорим о мультипрограммных ОС, для которых существует проблема разделения ресурсов. Одним из основных видов ресурсов являются устройства ввода/вывода и соответствующее программное обеспечение, с помощью которого осуществляется управление обменом данными между внешними устройствами и оперативной памятью. Помимо разделяемых устройств ввода/вывода (эти устройства допускают разделение посредством механизма доступа) существуют неразделяемые устройства. Примерами разделяемого устройства могут служить накопитель на магнитных дисках, устройство для чтения компакт-дисков. Это устройства с прямым доступом. Примеры неразделяемых устройств – принтер, накопитель на магнитных лентах. Это устройства с последовательным доступом. Операционные системы должны управлять и теми и другими устройствами, предоставляя возможность параллельно выполняющимся задачам использовать различные устройства ввода/вывода.

Можно назвать три основные причины, по которым нельзя разрешать каждой отдельной пользовательской программе обращаться к внешним устройствам непосредственно:

♦ Необходимость разрешать возможные конфликты доступа к устройствам ввода/вывода. Например, две параллельно выполняющиеся программы пытаются вывести на печать результаты своей работы. Если не предусмотреть внешнее управление устройством печати, то в результате мы можем получить абсолютно нечитаемый текст, так как каждая программа будет время от времени выводить свои данные, которые будут перемежаться данными другой программы. Другой пример: ситуация, когда одной программе необходимо прочитать данные с некото-

[1] *Исключение* – это определенный вид внутреннего прерывания. Этим термином, во-первых, обозначают некоторое множество синхронных прерываний, а во-вторых, подчеркивают, что ситуация, вызвавшая запрос на прерывание, является исключительной, то есть, отличается от обычной.

рого сектора магнитного диска, а другой – записать результаты в другой сектор того же накопителя. Если операции ввода/вывода не будут отслеживаться каким-то третьим (внешним) процессом-арбитром, то после позиционирования магнитной головки для первого запроса может тут же появиться команда позиционирования головки для второй задачи, и обе операции ввода/вывода не смогут быть выполнены корректно.

♦ Желание увеличить эффективность использования этих ресурсов. Например, у накопителя на магнитных дисках время подвода головки чтения/записи к необходимой дорожке и обращение к определенному сектору может значительно (до тысячи раз) превышать время пересылки данных. В результате, если задачи по очерёди обращаются к цилиндрам, далеко отстоящим друг от друга, то полезная работа, выполняемая накопителем, может быть существенно снижена.

♦ Ошибки в программах ввода/вывода могут привести к краху всех вычислительных процессов, ибо часть операций ввода/вывода осуществляется для самой операционной системы. В ряде ОС системный ввод/вывод имеет существенно более высокие привилегии, чем ввод/вывод задач пользователя. Поэтому системный код, управляющий операциями ввода/вывода, очень тщательно отлаживается и оптимизируется для повышения надёжности вычислений и эффективности использования оборудования.

Итак, управление вводом/выводом осуществляется операционной системой, компонентом, который чаще всего называют *супервизором ввода/вывода,* В перечень основных задач, возлагаемых на супервизор, входят следующие;

♦ супервизор ввода/вывода получает запросы на ввод/вывод от прикладных задач и от программных модулей самой операционной системы. Эти запросы проверяются на корректность, и если запрос выполнен по спецификациям и не содержит ошибок, он обрабатывается дальше, в противном случае пользователю (задаче) выдается соответствующее диагностическое сообщение о недействительности (некорректности) запроса;

♦ супервизор ввода/вывода вызывает соответствующие распределители каналов и контроллеров, планирует ввод/вывод (определяет очерёдность предостав-

ления устройств ввода/вывода задачам, затребовавшим их). Запрос на ввод/вывод либо тут же выполняется, либо ставится в очередь на выполнение;

♦супервизор ввода/вывода инициирует операции ввода/вывода (передаёт управление соответствующим драйверам) и в случае управления вводом/выводом с использованием прерываний предоставляет процессор диспетчеру задач с тем, чтобы передать его первой задаче, стоящей в очерёди на выполнение;

♦ при получении сигналов прерываний от устройств ввода/вывода супервизор идентифицирует их (рис.4.1) и передаёт управление соответствующей программе обработки прерывания (как правило, на секцию продолжения драйвера – см. раздел «Режимы управления вводом/выводом»);

♦ супервизор ввода/вывода осуществляет передачу сообщений об ошибках, если таковые происходят в процессе управления операциями ввода/вывода;

♦ супервизор ввода/вывода посылает сообщения о завершении операции ввода/вывода запросившему эту операцию процессу и снимает его с состояния ожидания ввода/вывода, если процесс ожидал завершения операции.

В случае, если устройство ввода/вывода является *инициативным*[1], управление со стороны супервизора ввода/вывода будет заключаться в активизации соответствующего вычислительного процесса (перевод его в состояние готовности к выполнению).

Таким образом, прикладные программы (а в общем случае – все обрабатывающие программы) не могут непосредственно связываться с устройствами ввода/вывода независимо от использования устройств (монопольно или совместно). Установив соответствующие значения параметров в *запросе на ввод/вывод,* определяющих требуемую операцию и количество потребляемых ресурсов, они могут

[1] *Инициативным* называют такое устройство (обычно это датчики, внешнее устройство, а не устройство ввода/вывода), по сигналу прерывания от которого запускается соответствующая ему программа. Такая программа, с одной стороны, не является драйвером, и управлять операциями обмена данными нет необходимости. Но, с другой стороны, запуск такой программы осуществляется именно по событиям, связанным с генерацией устройством ввода/вывода соответствующего сигнала. Разница между драйверами, работающими по прерываниям, и инициативными программами заключается в статусе этих программных модулей. Драйвер является компонентом операционной системы и часто выполняется не как вычислительный процесс, а как системный объект, а инициативная программа является обычным вычислительным процессом, только его запуск осуществляется по инициативе внешнего устройства.

передать управление супервизору ввода/вывода, который и запускает необходимые логические и физические операции.

Упомянутый выше запрос на ввод/вывод должен удовлетворять требованиям API той операционной системы, в среде которой выполняется приложение. Параметры, указываемые в запросах на ввод/вывод, передаются не только в вызывающих последовательностях, создаваемых по спецификациям API, но и как данные, хранящиеся в соответствующих системных таблицах. Все параметры, которые будут стоять в вызывающей последовательности, поставляются компилятором и отражают требования программиста и постоянные сведения об операционной системе и архитектуре компьютера в целом. Переменные сведения о вычислительной системе (её конфигурация, состав оборудования, состав и особенности системного программного обеспечения) содержатся в специальных системных таблицах. Процессору, каналам прямого доступа в память, контроллерам необходимо передавать конкретную двоичную информацию, с помощью которой и осуществляется управление оборудованием. Эта конкретная двоичная информация в виде кодов и данных часто готовится с помощью препроцессоров, но часть её хранится в системных таблицах.

Режимы управления вводом/выводом

Как известно, имеются два основных режима ввода/вывода: *режим обмена с опросом готовности* устройства ввода/вывода и *режим обмена с прерываниями.* Рассмотрим рис. 4.1.

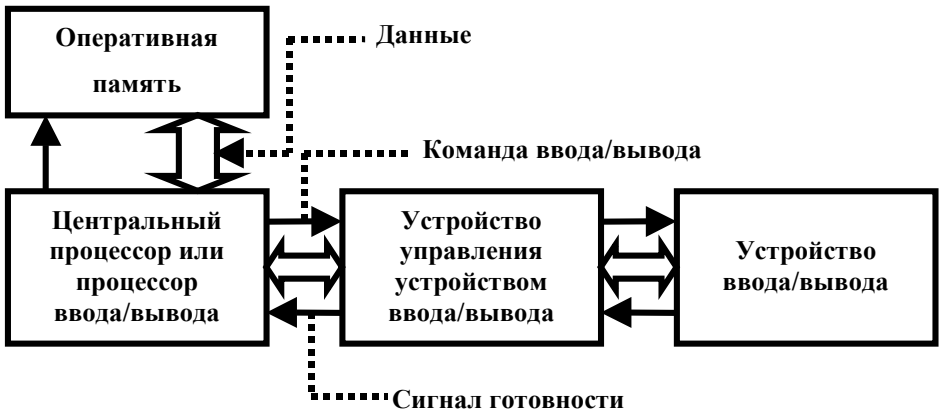

Рис.4.1. Управление вводом/выводом

Пусть для простоты управление вводом/выводом осуществляет центральный процессор (в этом случае часто говорят о наличии программного канала обмена данными между внешними устройством и оперативной памятью, в отличие от канала прямого доступа к памяти, при котором управление вводом/выводом осуществляет специальное дополнительное оборудование; эти вопросы мы обсудим несколько позже). Центральный процессор посылает устройству управления команду выполнить некоторое действие устройству ввода/вывода. Последнее исполняет команду, транслируя сигналы, понятные центральному устройству и устройству управления в сигналы, понятные устройству ввода/вывода. Но быстродействие устройства ввода/вывода намного меньше быстродействия центрального процессора (порой на несколько порядков). Поэтому сигнал готовности (транслируемый или генерируемый устройством управления и сигнализирующий процессору о том, что команда ввода/вывода выполнена и можно выдать новую команду для продолжения обмена данными) приходится очень долго ожидать, постоянно опрашивая соответствующую линию интерфейса на наличие или отсутствие нужного сигнала. Посылать новую команду, не дождавшись сигнала готовности, сообщающего об исполнении предыдущей команды, бессмысленно. В режиме опроса готовности драйвер, управляющий процессом обмена данными с внешним устройством, как раз и выполняет в цикле команду «проверить наличие сигнала готовности». До тех пор пока сигнал готовности не появится, драйвер ничего другого не делает. При этом, естественно, нерационально используется время центрального процессора. Гораздо выгоднее, выдав команду ввода/ вывода, на время забыть об устройстве ввода/вывода и перейти на выполнение другой программы. А появление сигнала готовности трактовать как запрос на прерывание от устройства ввода/вывода. Именно эти сигналы готовности и являются сигналами запроса на прерывание (см. раздел «Прерывания», глава 1).

Режим обмена с прерываниями по своей сути является режимом асинхронного управления. Для того чтобы не потерять связь с устройством (после того как процессор выдал очередную команду по управлению обменом данными и переключился на выполнение других программ), может быть запущен отсчёт времени, в те-

чение которого устройство обязательно должно выполнить команду и выдать таки сигнал запроса на прерывание. Максимальный интервал времени, в течение которого устройство ввода/вывода или его контроллер должны выдать сигнал запроса на прерывание, часто называют *уставкой тайм-аута.* Если это время истекло после выдачи устройству очередной команды, а устройство так и не ответило, то делается вывод о том, что связь с устройством потеряна и управлять им больше нет возможности. Пользователь и/или задача получают соответствующее диагностическое сообщение.

Драйверы, работающие в режиме прерываний, представляют собой сложный комплекс программных модулей и могут иметь несколько секций: секцию запуска, одну или несколько секций продолжения и секцию завершения.

Секция запуска инициирует операцию ввода/вывода. Эта секция запускается для включения устройства ввода/вывода либо просто для инициации очередной операции ввода/вывода.

Секция продолжения (их может быть несколько, если алгоритм управления обменом данными сложный и требуется несколько прерываний для выполнения одной логической операции) осуществляет основную работу по передаче данных. Секция продолжения, собственно говоря, и является основным обработчиком прерывания. Используемый интерфейс может потребовать для управления вводом/выводом несколько последовательностей управляющих команд, а сигнал прерывания у устройства, как правило, только один. Поэтому после выполнения очередной секции прерывания супервизор прерываний при следующем сигнале готовности должен передать управление другой секции. Это делается за счёт изменения адреса обработки прерывания после выполнения очередной секции, если же имеется только одна секция прерываний, то она сама передаёт управление тому или иному модулю обработки.

Секция завершения обычно выключает устройство ввода/вывода либо просто завершает операцию.

Управление операциями ввода/вывода в режиме прерываний, требует больших усилий со стороны системных программистов – такие программы создавать слож-

нее, чем те, что работают в режиме опроса готовности. Примером тому может служить ситуация с драйверами, обеспечивающими печать. Так, в ОС Windows (и Windows 9x, и Windows NT) драйвер печати через параллельный порт работает не в режиме с прерываниями, как это сделано в других ОС, а в режиме опроса готовности, что приводит к 100%-й загрузке центрального процессора на всё время печати. При этом, естественно, выполняются и другие задачи, запущенные на исполнение, но исключительно за счёт того, что ОС Windows реализует вытесняющую мультизадачность и время от времени прерывает процесс управления печатью и передаёт центральный процессор остальным задачам.

Закрепление устройств, общие устройства ввода/вывода

Как известно, многие устройства не допускают совместного использования. Прежде всего, это устройства с последовательным доступом. Такие устройства могут стать закреплёнными, то есть быть предоставленными некоторому вычислительному процессу на всё время жизни этого процесса. Однако это приводит к тому, что вычислительные процессы часто не могут выполняться параллельно – они ожидают освобождения устройств ввода/вывода. Для организации использования многими параллельно выполняющимися задачами устройств ввода/вывода, которые не могут быть разделяемыми, вводится понятие *виртуальных* устройств. Использование принципа виртуализации позволяет повысить эффективность вычислительной системы.

Вообще говоря, понятие виртуального устройства шире, нежели использование этого термина для обозначения *спулинга* (SPOOLing – simultaneous peripheral operation on-line, то есть имитация работы с устройством в режиме «он-лайн»). Главная задача спулинга – создать видимость параллельного разделения устройства ввода/вывода с последовательным доступом, которое фактически должно использоваться только монопольно и быть закрепленным. Например, мы уже говорили, что в случае, когда несколько приложений должны выводить на печать результаты своей работы, если разрешить каждому такому приложению печатать строку по первому же требованию, то это приведет к потоку строк, не представ-

ляющих никакой ценности. Однако можно каждому вычислительному процессу предоставлять не реальный, а виртуальный принтер и поток выводимых символов (или управляющих кодов для их печати) сначала направлять в специальный файл[1] на магнитном диске. Затем, по окончании виртуальной печати, в соответствии с принятой дисциплиной обслуживания и приоритетами приложений выводить содержимое спул-файла на принтер. Системный процесс, который управляет спул-файлом, называется *спулером* (spool-reader или spool-writer).

Основные системные таблицы ввода/вывода

Каждая ОС имеет свои таблицы ввода/вывода, их состав (количество и назначение каждой таблицы) может сильно отличаться. В некоторых ОС вместо таблиц создаются списки, хотя использование статических структур данных для организации ввода/вывода, как правило, приводит к большему быстродействию. Здесь очень трудно вычленить общие составляющие, тем более что подробной документации на эту тему крайне мало, только если воспользоваться материалами ныне устаревших ОС. Тем не менее, попытаемся это сделать, опираясь на идеи семейства простых, но эффективных ОС реального времени, разработанных фирмой Hewlett–Packard для своих мини-компьютеров.

Исходя из принципа управления вводом/выводом через супервизор ОС и учитывая, что драйверы устройств ввода/вывода используют механизм прерываний для установления обратной связи центральной части с внешними устройствами, можно сделать вывод о необходимости создания по крайней мере трёх системных таблиц.

Первая таблица (или список) содержит информацию обо всех устройствах ввода/вывода, подключенных к вычислительной системе. Назовем её условно *таблицей оборудования* (equipment table), а каждый элемент этой таблицы пусть называется UCB (unit control block, блок управления устройством ввода/вывода). Каждый элемент UCB таблицы оборудования, как правило, содержит следующую информацию об устройстве:

[1] Такой файл называют *спул-файлом* (spool-file).

♦ тип устройства, его конкретная модель, символическое имя и характеристики устройства;

♦ как это устройство подключено (через какой интерфейс, к какому разъёму, какие порты и линия запроса прерывания используются и т. д.);

♦ номер и адрес канала (и подканала), если такие используются для управления устройством;

♦ указание на драйвер, который должен управлять этим устройством, адрес секции запуска и секции продолжения драйвера;

♦ информация о том, используется или нет буферирование при обмене данными с этим устройством, «имя» (или просто адрес) буфера, если такой выделяется из системной области памяти;

♦ уставка тайм-аута и ячейки для счетчика тайм-аута;

♦ состояние устройства;

♦ поле указателя для связи задач, ожидающих устройство, и, возможно, много ещё каких сведений. Поясним перечисленное. Поскольку во многих ОС драйверы могут обладать свойством реентерабельности (напомним, это означает, что один и тот же экземпляр программного модуля может обеспечить параллельное обслуживание сразу нескольких однотипных устройств), то в элементе UCB должна храниться либо непосредственно сама информация о текущем состоянии устройства и сами переменные для реентерабельной обработки, либо указание на место, где такая информация может быть найдена. Наконец, важнейшим компонентом элемента таблицы оборудования является указатель на дескриптор той задачи, которая сейчас использует данное устройство. Если устройство свободно, то поле указателя будет иметь нулевое значение. Если же устройство уже занято и рассматриваемый указатель не нулевой, то новые запросы к устройству фиксируются посредством образования списка из дескрипторов тех задач, которые сейчас ожидают данное устройство.

Вторая таблица предназначена для реализации ещё одного принципа виртуализации устройств ввода/вывода – независимости от устройства. Желательно, чтобы программист не был озабочен учётом конкретных параметров (и/или возможностей) того или иного устройства ввода/вывода, которое установлено (или не установлено) в компьютер. Для него должны быть важны только самые общие возможности, характерные для данного класса устройств ввода/вывода, которыми он желает воспользоваться. Например, принтер должен уметь выводить (печатать) символы или графическое изображение. А накопитель на магнитных дисках – считывать или записывать по указанному адресу (в координатах C-H-S[1]) порцию данных. Хотя чаще всего программист и не использует прямую адресацию при работе с магнитными дисками, а работает на уровне файловой системы (см. главу 4). Однако в таком случае уже разработчики файловой системы не должны зависеть от того, накопитель какого конкретного типа и модели, а также какого производителя используется в данном конкретном компьютере (например, HDD IBM DTLA 307030, WDAC 450AA или какой-нибудь ещё). Важным должен быть только сам факт существования накопителя, имеющего некоторое количество цилиндров, головок чтения/записи и секторов на дорожке магнитного диска. Упомянутые значения количества цилиндров, головок и секторов должны быть взяты из элемента таблицы оборудования. При этом для программиста также не должно иметь значения, каким образом то или иное устройство подключено к вычислительной системе, а не только какая конкретная модель устройства используется. Поэтому в запросе на ввод/вывод программист указывает именно *логическое имя устройства.* Действительное устройство, которое сопоставляется виртуальному (логическому), выбирается супервизором с помощью таблицы, о которой мы сейчас говорим. Итак, способ подключения устройства, его конкретная модель и соответствующий ей драйвер содержатся в уже рассмотренной таблице оборудования. Но для того, чтобы связать некоторое виртуальное устройство, использованное программистом при создании приложения с системной таблицей, отображающей информацию о том, какое конкретно устройство и каким образом подключено к компьютеру, ис-

[1] *C-H-S* (cylinder-head-sector) – номер цилиндра, номер головки и номер сектора.

пользуется вторая системная таблица. Назовем её условно *таблицей описания виртуальных логических устройств* (DRT, device reference table). Назначение этой второй таблицы – установление связи между виртуальными (логическими) устройствами и реальными устройствами, описанными посредством первой таблицы оборудования. Другими словами, вторая таблица позволяет супервизору перенаправить запрос на ввод/вывод из приложения на те программные модули и структуры данных, которые (или адреса которых) хранятся в соответствующем элементе первой таблицы. Во многих многопользовательских системах такая таблица не одна, а несколько: одна общая и по одной – на каждого пользователя, что позволяет строить необходимые связи между логическими (символьными) именами устройств и реальными физическими устройствами, которые имеются в системе.

Наконец, третья таблица необходима для организации обратной связи между центральной частью и устройствами ввода/вывода. Это *таблица прерываний,* которая указывает для каждого сигнала запроса на прерывание тот элемент UCB, который сопоставлен данному устройству, подключенному так, что оно использует настоящую линию (сигнал) прерывания. Как системная таблица ввода/вывода, таблица прерываний может в явном виде и не присутствовать. В принципе можно сразу из основной таблицы прерываний попадать на программу обработки (драйвер), имеющей связи с элементом UCB. Важно наличие связи между сигналами прерываний и таблицей оборудования.

В современных сложных ОС имеется гораздо больше системных таблиц или списков, используемых для организации процессов управления операциями ввода/вывода. Например, одной из возможных и часто реализуемых информационных структур, сопровождающих практически каждый запрос на ввод/вывод, является блок управления данными (data control block, DCB). Назначение этого DCB – подключение препроцессоров к процессу подготовки данных на ввод/вывод, то есть учет конкретных технических характеристик и используемых преобразований. Это необходимо для того, чтобы имеющееся устройство получало не какие-то непонятные ему коды либо форматы данных, которые не соответствуют режиму его рабо-

ты, а коды, созданные специально под данное устройство и используемый в настоящий момент формат представления данных.

Взаимосвязи между описанными таблицами изображены на рис.4.2.

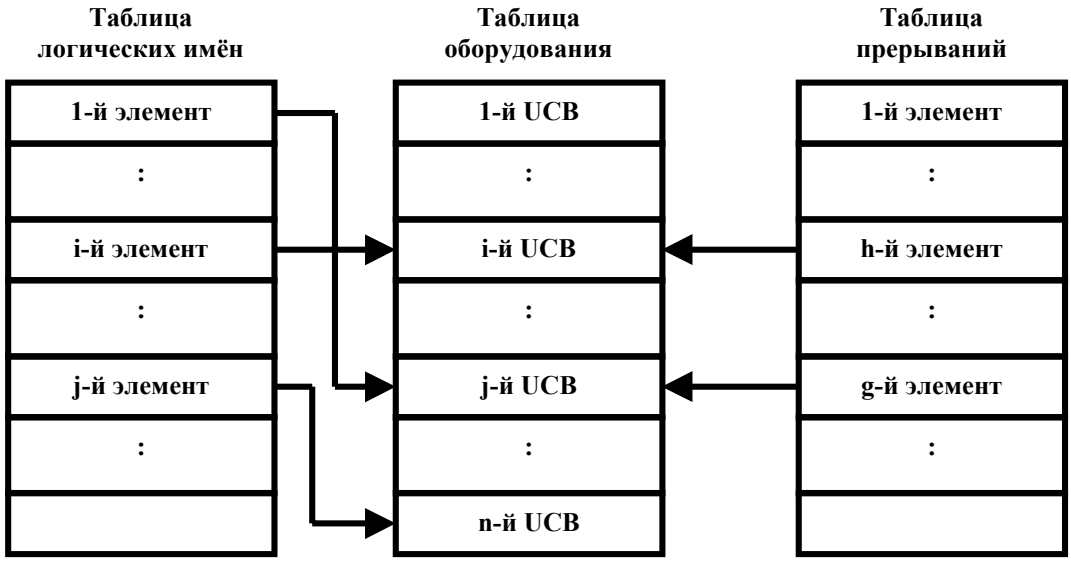

Рис.4.2. Взаимосвязь системных таблиц ввода/вывода

Теперь нам осталось лишь ещё раз, с учётом изложенных принципов и таблиц, рассмотреть процесс управления вводом/вывода с помощью рис. 4.3.

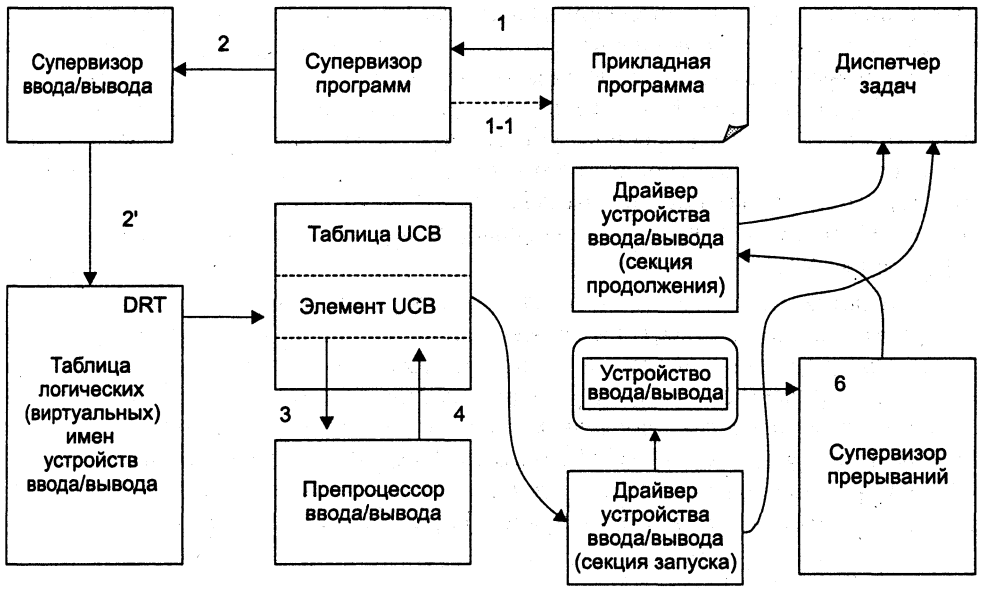

Рис.4.3. Процесс управления вводом/выводом

Запрос на операцию ввода/вывода от выполняющейся программы поступает на супервизор (действие 1). Тот проверяет системный вызов на соответствие принятым спецификациям и в случае ошибки возвращает задаче соответствующее со-

общение (действие 1–1). Если же запрос корректен, то он перенаправляется в супервизор ввода/вывода (действие 2). Последний по логическому (виртуальному) имени с помощью таблицы DRT находит соответствующий элемент UCB в таблице оборудования. Если устройство уже занято, то описатель задачи, запрос которой сейчас обрабатывается супервизором ввода/вывода, помещается в список задач, ожидающих настоящее устройство. Если же устройство свободно, то супервизор ввода/вывода определяет из UCB тип устройства и при необходимости запускает препроцессор, позволяющий получить последовательность управляющих кодов и данных, которую сможет правильно понять и отработать устройство (действие 3). Когда «программа» управления операцией ввода/вывода будет готова, супервизор ввода/вывода передаст управление соответствующему драйверу на секцию запуска (действие 4). Драйвер инициализирует операцию управления, обнуляет счётчик тайм-аута и возвращает управление супервизору (диспетчеру задач) с тем, чтобы он поставил на процессор готовую к исполнению задачу (действие 5). Система работает своим чередом, но когда устройство ввода/вывода отработает посланную ему команду, оно выставляет сигнал запроса на прерывания, по которому через таблицу прерываний управление передаётся на секцию продолжения (действие 6). Получив новую команду, устройство вновь начинает её обрабатывать, а управление процессором опять передаётся диспетчеру задач, и процессор продолжает полезную работу. Таким образом, получается параллельная обработка задач, на фоне которой процессор осуществляет управление операциями ввода/вывода.

Очевидно, что если имеются специальные аппаратные средства для управления вводом/выводом, снимающие эту работу с центрального процессора (речь идет о каналах прямого доступа к памяти), то в функции центрального процессора будут по-прежнему входить все только что рассмотренные шаги, за исключением последнего – непосредственного управления операциями ввода/вывода. В случае использования каналов прямого доступа к памяти последние исполняют соответствующие канальные программы и разгружают центральный процессор, избавляя его от непосредственного управления обменом данными между памятью и внешними устройствами.

При описании этой схемы мы не стали затрагивать вопросы распределения каналов, контроллеров и собственно самих устройств. Также были опущены детали получения канальных программ. Будем считать, что они выходят за круг вопросов, рассматриваемых в учебнике по дисциплине «Системное программное обеспечение», тем более что в последнем государственном образовательном стандарте для направления 46 – «Информатика и вычислительная техника» имеется отдельная общепрофессиональная дисциплина «Операционные системы» и специальная дисциплина «Интерфейсы периферийных устройств».

Синхронный и асинхронный ввод/вывод

Задача, выдавшая запрос на операцию ввода/вывода, переводится супервизором в состояние ожидания завершения заказанной операции. Когда супервизор получает от секции завершения сообщение о том, что операция завершилась, он переводит задачу в состояние готовности к выполнению, и она продолжает свою работу. Эта ситуация соответствует синхронному вводу/выводу. Синхронный ввод/вывод является стандартным для большинства ОС. Чтобы увеличить скорость выполнения приложений, было предложено при необходимости использовать асинхронный ввод/вывод.

Простейшим вариантом асинхронного вывода является так называемый буферированный вывод данных на внешнее устройство, при котором данные из приложения передаются не непосредственно на устройство ввода/вывода, а в специальный системный буфер. В этом случае логически операция вывода для приложения считается выполненной сразу же, и задача может не ожидать окончания действительного процесса передачи данных на устройство. Процессом реального вывода данных из системного буфера занимается супервизор ввода/ вывода. Естественно, что выделением буфера из системной области памяти занимается специальный системный процесс по указанию супервизора ввода/вывода. Итак, для рассмотренного случая вывод будет асинхронным, если, во-первых, в запросе на ввод/вывод было указано на необходимость буферирования данных, а во-вторых, если устройство ввода/вывода допускает такие асинхронные операции и это отмечено в UCB. Можно организовать и асинхронный ввод данных. Однако для этого необходимо

не только выделить область памяти для временного хранения считываемых с устройства данных и связать выделенный буфер с задачей, заказавшей операцию, но и сам запрос на операцию ввода/вывода разбить на две части (на два запроса). В первом запросе указывается операция на считывание данных, подобно тому, как это делается при синхронном вводе/выводе. Однако тип (код) запроса используется другой, и в запросе указывается ещё по крайней мере один дополнительный параметр – имя (код) того системного объекта, которое получает задача в ответ на запрос и которое идентифицирует выделенный буфер. Получив имя буфера (будем этот системный объект условно называть таким образом, хотя в различных ОС для его обозначения используются и другие термины, например – класс), задача продолжает свою работу. Здесь очень важно подчеркнуть, что в результате запроса на асинхронный ввод данных задача не переводится супервизором ввода/вывода в состояние ожидания завершения операции ввода/ вывода, а остается в состоянии выполнения или в состоянии готовности к выполнению. Через некоторое время, выполнив необходимый код, который был определен программистом, задача выдаёт второй запрос на завершение операции ввода/вывода. В этом втором запросе к тому же устройству, который, естественно, имеет другой код (или имя запроса), задача указывает имя системного объекта (буфера для асинхронного ввода данных) и в случае успешного завершения операции считывания данных тут же получает их из системного буфера. Если же данные ещё не успели до конца переписаться с внешнего устройства в системный буфер, супервизор ввода/вывода переводит задачу в состояние ожидания завершения операции ввода/вывода, и далее всё напоминает обычный синхронный ввод данных.

Обычно асинхронный ввод/вывод предоставляется в большинстве мультипрограммных ОС, особенно если ОС поддерживает мультизадачность с помощью механизма тредов. Однако если асинхронный ввод/вывод в явном виде отсутствует, его идеи можно реализовать самому, организовав для вывода данных самостоятельный поток.

Аппаратуру ввода/вывода можно рассматривать как совокупность *аппаратурных процессоров*, которые способны работать параллельно относительно друг дру-

га, а также относительно центрального процессора (процессоров). На таких «процессорах» выполняются так называемые *внешние процессы*. Например, для внешнего устройства (устройства ввода/вывода) внешний процесс может представлять собой совокупность операций, обеспечивающих перевод печатающей головки, продвижение бумаги на одну позицию, смену цвета чернил или печать каких-то символов. Внешние процессы, используя аппаратуру ввода/вывода, взаимодействуют как между собой, так и с обычными «программными» процессами, выполняющимися на центральном процессоре. Важным при этом является то обстоятельство, что скорости выполнения внешних процессов будут существенно (порой, на порядок или больше) отличаться от скорости выполнения обычных («*внутренних*») процессов. Для своей нормальной работы внешние и внутренние процессы обязательно должны синхронизироваться. Для сглаживания эффекта сильного несоответствия скоростей между внутренними и внешними процессами используют упомянутое выше буферирование. Таким образом, можно говорить о системе параллельных взаимодействующих процессов (см. главу 6).

Буферы являются критическим ресурсом в отношении внутренних (программных) и внешних процессов, которые при параллельном своем развитии информационно взаимодействуют. Через буфер (буферы) данные либо посылаются от некоторого процесса к адресуемому внешнему (операция вывода данных на внешнее устройство), либо от внешнего процесса передаются некоторому программному процессу (операция считывания данных). Введение буферирования как средства информационного взаимодействия выдвигает проблему управления этими системными буферами, которая решается средствами супервизорной части ОС. При этом на супервизор возлагаются задачи не только по выделению и освобождению буферов в системной области памяти, но и синхронизации процессов в соответствии с состоянием операций по заполнению или освобождению буферов, а также их ожидания, если свободных буферов в наличии нет, а запрос на ввод/вывод требует буферирования. Обычно супервизор ввода/вывода для решения перечисленных задач использует стандартные средства синхронизации, принятые в данной ОС. Поэтому если ОС имеет развитые средства для решения проблем параллельного выполнения

взаимодействующих приложений и задач, то, как правило, она реализует и асинхронный ввод/вывод.

Кэширование операций ввода/вывода при работе с накопителями на магнитных дисках

Как известно, накопители на магнитных дисках обладают крайне низкой скоростью по сравнению с быстродействием центральной части компьютера. Разница в быстродействии отличается на несколько порядков. Например, современные процессоры за один такт работы, а они работают уже с частотами в 1 ГГц и более, могут выполнять по две операции. Таким образом, время выполнения операции (с позиции внешнего наблюдателя, не видящего конвейеризации при выполнении машинных команд, благодаря которой производительность возрастает в несколько раз) может составлять 0,5 нс (!). В то же время переход магнитной головки с дорожки на дорожку составляет несколько миллисекунд. Такие же временные интервалы имеют место и при ожидании, пока под головкой чтения/ записи не окажется нужный сектор данных. Как известно, в современных приводах средняя длительность на чтение случайным образом выбранного сектора данных составляет около 20 мс, что существенно медленнее, чем выборка команды или операнда из оперативной памяти и уж, тем более, из кэша. Правда, после этого данные читаются большим пакетом (сектор, как мы уже говорили, имеет размер в 512 байтов, а при операциях с диском часто читается или записывается сразу несколько секторов). Таким образом, средняя скорость работы процессора с оперативной памятью на 2-3 порядка выше, чем средняя скорость передачи данных из внешней памяти на магнитных дисках в оперативную память. Для того чтобы сгладить такое сильное несоответствие в производительности основных подсистем, используется буферирование и/или кэширование[1] данных. Простейшим вариантом ускорения дисковых операций чтения данных можно считать использование двойного буферирования. Его суть заключается в том, что пока в один буфер заносятся данные с магнитного диска, из второго буфера ранее считанные данные могут быть прочитаны и переда-

[1] *Disk cache* – кэш диска.

ны запросившей их задаче. Аналогичный процесс происходит и при записи данных. Буферирование используется во всех операционных системах, но помимо буферирования применяется и кэширование. Кэширование исключительно полезно в том случае, когда программа неоднократно читает с диска одни и те же данные. После того как они один раз будут помещены в кэш, обращений к диску больше не потребуется и скорость работы программы значительно возрастет.

Если не вдаваться в подробности, то под кэшем можно понимать некий пул буферов, которыми мы управляем с помощью соответствующего системного процесса. Если мы считываем какое-то множество секторов, содержащих записи того или иного файла, то эти данные, пройдя через кэш, там остаются (до тех пор, пока другие секторы не заменят эти буферы). Если впоследствии потребуется повторное чтение, то данные могут быть извлечены непосредственно из оперативной памяти без фактического обращения к диску. Ускорить можно и операции записи: данные помещаются в кэш, и для запросившей эту операцию задачи можно считать, что они уже фактически и записаны. Задача может продолжить своё выполнение, а системные внешние процессы через некоторое время запишут данные на диск. Это называется *операцией отложенной записи* (lazy write, «ленивая запись»). Если отложенная запись отключена, только одна задача может записывать на диск свои данные. Остальные приложения должны ждать своей очереди. Это ожидание подвергает информацию риску не меньшему (если не большему), чем отложенная запись, которая к тому же и более эффективна по скорости работы с диском.

Интервал времени, после которого данные будут фактически записываться, с одной стороны, желательно выбрать больше, поскольку если потребуется ещё раз прочитать эти данные, то они уже и так фактически находятся в кэше. И после модификации эти данные опять же помещаются в быстродействующий кэш. С другой стороны, для большей надёжности данные желательно поскорее отправить во внешнюю память, поскольку она энергонезависима и в случае какой-нибудь аварии (например, нарушения питания) данные в оперативной памяти пропадут, в то время как на магнитном диске они с большой вероятностью останутся в безопасности. Количество буферов, составляющих кэш, ограничено, поэтому возникает ситуация,

когда вновь прочитанные или записываемые новые секторы данных должны будут заменить данные в этих буферах. Возможно использование различных дисциплин, в соответствии с которыми будет назначен какой-либо буфер под вновь затребованную операцию кэширования.

Кэширование дисковых операций может быть существенно улучшено за счёт введения техники *упреждающего чтения* (read ahead). Она основана на чтении с диска гораздо большего количества данных, чем на самом деле запросила операционная система или приложение. Когда некоторой программе требуется считать с диска только один сектор, программа кэширования читает ещё и несколько дополнительных блоков данных, А операции последовательного чтения нескольких секторов фактически несущественно замедляют операцию чтения затребованного сектора с данными. Поэтому, если программа вновь обратится к диску, вероятность того, что нужные ей данные уже находятся в кэше, достаточно высока. Поскольку передача данных из одной области памяти в другую происходит во много раз быстрее, чем чтение их с диска, кэширование существенно сокращает время выполнения операций с файлами.

Итак, путь информации от диска к прикладной программе пролегает как через буфер, так и через файловый кэш. Когда приложение запрашивает с диска данные, программа кэширования перехватывает этот запрос и читает вместе с необходимыми секторами ещё и несколько дополнительных. Затем она помещает в буфер требующуюся задаче информацию и ставит об этом в известность операционную систему. Операционная система сообщает задаче, что её запрос выполнен и данные с диска находятся в буфере. При следующем обращении приложения к диску программа кэширования прежде всего проверяет, не находятся ли уже в памяти затребованные данные. Если это так, то она копирует их в буфер; если же их в кэше нет, то запрос на чтение диска передаётся операционной системе. Когда задача изменяет данные в буфере, они копируются в кэш.

В ряде ОС имеется возможность указать в явном виде параметры кэширования, в то время как в других за эти параметры отвечает сама ОС. Так, например, в системе Windows NT нет возможности в явном виде управлять ни объёмом файло-

вого кэша, ни параметрами кэширования. В системах Windows 95/98 такая возможность уже имеется, но она представляет не слишком богатый выбор. Фактически мы можем указать только объём памяти, отводимый для кэширования, и объём порции данных (буфер или chunk[1]), из которых набирается кэш. В файле System.ini есть возможность в секции [VCACHE] прописать, например, следующие значения:

```
[vcache]
MinFileCache=4096
MaxFileCache=32768
ChunkSize=512
```

Здесь указано, что минимально под кэширование данных зарезервировано 4 Мбайт оперативной памяти, максимальный объём кэша может достигать 32 Мбайт, а размер данных, которыми манипулирует менеджер кэша, равен одному сектору.

В других ОС можно указывать больше параметров, определяющих работу подсистемы кэширования. Пример, демонстрирующий эти возможности, можно посмотреть в разделе «Файловая система HPFS».

Помимо описанных действий ОС может выполнять и работу по оптимизации перемещения головок чтения/записи данных, связанного с выполнением запросов от параллельно выполняющихся задач. Время, необходимое на получение данных с магнитного диска, складывается из времени перемещения магнитной головки на требуемый цилиндр и времени ожидания заданного сектора; временем считывания найденного сектора и затратами на передачу этих данных в оперативную память мы можем пренебречь. Таким образом, основные затраты времени уходят на поиск данных. В мультипрограммных ОС при выполнении многих задач запросы на чтение и запись данных могут идти таким потоком, что при их обслуживании образуется очередь. Если выполнять эти запросы в порядке поступления их в очередь, то вследствие случайного характера обращений к тому или иному сектору магнитного диска мы можем иметь значительные потери времени на поиск данных. Напрашивается очевидное решение: поскольку выполнение переупорядочивания запросов с целью минимизации затрат времени на поиск данных можно выполнить очень бы-

[1] *Chunk* – кусочек.

стро (практически этим временем можно пренебречь, учитывая разницу в быстродействии центральной части и устройств ввода/вывода), то необходимо найти метод, позволяющий перестраивать очередь запросов оптимальным образом. Изучение этой проблемы позволило найти наиболее эффективные дисциплины планирования.

Перечислим известные дисциплины, в соответствии с которыми можно перестраивать очередь запросов на операции чтения/записи данных [28]:

♦ *SSTF* (shortest seek time – first) – с наименьшим временем поиска – первым. В соответствии с этой дисциплиной при позиционировании магнитных головок следующий выбирается запрос, для которого необходимо минимальное перемещение с цилиндра на цилиндр, даже если этот запрос не был первым в очереди на ввод/вывод. Однако для этой дисциплины характерна резкая дискриминация определенных запросов, а ведь они могут идти от высокоприоритетных задач. Обращения к диску проявляют тенденцию концентрироваться, в результате чего запросы на обращение к самым внешним и самым внутренним дорожкам могут обслуживаться существенно дольше и нет никакой гарантии обслуживания. Достоинством такой дисциплины является максимально возможная пропускная способность дисковой подсистемы.

♦ *Scan* (сканирование). По этой дисциплине головки перемещаются то в одном, то в другом «привилегированном» направлении, обслуживая «по пути» подходящие запросы. Если при перемещении головок чтения/записи более нет попутных запросов, то движение начинается в обратном направлении.

♦ *Next-Step Scan* – отличается от предыдущей дисциплины тем, что на каждом проходе обслуживаются только запросы, которые уже существовали на момент начала прохода. Новые запросы, появляющиеся в процессе перемещения головок чтения/записи, формируют новую очередь запросов, причем таким образом, чтобы их можно было оптимально обслужить на обратном ходу.

♦ *C-Scan* (циклическое сканирование). По этой дисциплине головки перемещаются циклически с самой наружной дорожки к внутренним, по пути обслуживая

имеющиеся запросы, после чего вновь переносятся к наружным цилиндрам. Эту дисциплину иногда реализуют таким образом, чтобы запросы, поступающие во время текущего прямого хода головок, обслуживались не попутно, а при следующем ходе, что позволяет исключить дискриминацию запросов к самым крайним цилиндрам; она характеризуется очень малой дисперсией времени ожидания обслуживания [28]. Эту дисциплину обслуживания часто называют «элеваторной».

Функции файловой системы ОС и иерархия данных

Напомним, что под *файлом* обычно понимают набор данных, организованных в виде совокупности записей одинаковой структуры. Для управления этими данными создаются соответствующие системы управления файлами. Возможность иметь дело с логическим уровнем структуры данных и операций, выполняемых над ними в процессе их обработки, предоставляет файловая система. Таким образом, *файловая система* – это 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[1].

Как правило, все современные ОС имеют соответствующие *системы управления файлами.* В дальнейшем постараемся различать файловую систему и систему управления файлами.

Система управления файлами является основной подсистемой в абсолютном большинстве современных операционных систем, хотя в принципе можно обходиться и без неё. Во-первых, через систему управления файлами связываются по данным все системные обрабатывающие программы. Во-вторых, с помощью этой

[1] В MS-DOS, OS/2, Windows 95, Windows NT, Linux имеется поддержка работы с файлами, организованными по принципам FAT. Однако программные модули соответствующих файловых систем не взаимозаменяемы. Кроме этого, все эти файловые системы имеют свои индивидуальные особенности и ограничения. Таким образом, иногда только из контекста ясно, о чём идет речь – о принципах работы файловой системы или о её конкретной реализации.

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 не дисковыми периферийными устройствами как с файлами;

♦ обмен данными между файлами, между устройствами, между файлом и устройством (и наоборот);

♦ работа с файлами с помощью обращений к программным модулям системы управления файлами (часть API ориентирована именно на работу с файлами);

♦ защита файлов от несанкционированного доступа.

В некоторых ОС может быть несколько систем управления файлами, что обеспечивает им возможность работать с несколькими файловыми системами. Очевидно, что системы управления файлами, будучи компонентом ОС, не являются независимыми от этой ОС, поскольку они активно используют соответствующие вызовы API (application program interface, прикладной программный интерфейс). 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, то есть вместо низкоуровневого доступа к данным с указанием конкретных физических адресов нужной нам записи используется логический доступ с указанием имени файла и записи в нём.

Другими словами, термин «файловая система» определяет, 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. А термин «система управления файлами» следует употреблять по отношению к конкретной реализации файловой системы, то есть это – комплекс

программных модулей, обеспечивающих работу с файлами в конкретной операционной системе.

Следует ещё раз 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 реализации как система управления файлами. Так, система, получившая это название и разработанная для первых персональных компьютеров, называлась просто FAT (сейчас её называют FAT-12). Её разрабатывали для работы с дискетами, и некоторое время она использовалась при работе с жесткими дисками. Потом её усовершенствовали для работы с жесткими дисками большего объёма, 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 отличие – 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.

Структура магнитного диска (разбиение дисков на разделы)

Для того чтобы можно было загрузить с магнитного диска собственно саму ОС, а уже с её помощью и организовать работу той или иной системы управления файлами, были приняты специальные системные соглашения о структуре диска. Расположение структуры данных, несущее информацию о логической организации диска и простейшую программу, с помощью которой можно находить и загружать

программы загрузки той или иной ОС, очевидно – это самый первый сектор магнитного диска.

Как известно, информация на магнитных дисках размещается и передаётся блоками. Каждый такой блок называется *сектором* (sector), сектора расположены на концентрических дорожках поверхности диска. Каждая дорожка (track) образуется при вращении магнитного диска под зафиксированной в некотором предопределенном положении головкой чтения/записи. Накопитель на жестких магнитных дисках (НЖМД)[1] содержит один или более дисков (в современных распространенных НЖМД часто – два или три). Однако обычно под термином «жесткий диск» понимают весь пакет магнитных дисков [24].

Группы дорожек (треков) 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 (число цилиндров) ныне, как правило, составляет всего восемьдесят[2].

Каждый сектор состоит из *поля данных* и *поля служебной информации,* ограничивающей и идентифицирующей его. Размер сектора (точнее – ёмкость поля данных) устанавливается контроллером или драйвером. Пользовательский интерфейс DOS поддерживает единственный размер сектора – 512 байт. BIOS же непосредственно предоставляет возможности работы с секторами размером 128, 256, 512 или 1024 байт. Если управлять контроллером непосредственно, а не через программный интерфейс более высокого уровня (например, уровень DOS), то можно обрабатывать секторы и с другими размерами. Однако в большинстве современных ОС размер сектора выбирается равным 512 байт.

Физический адрес сектора на диске определяется с помощью трех «координат», то есть представляется триадой [c-h-s], где c – номер цилиндра (дорожки на

[1] *НЖМД* обычно называется «винчестером» – так в свое время стали называть одну из первых моделей НЖМД, которая имела обозначение «30/30» и этим напоминала маркировку известного оружия. Возможно, также, что название происходит от места первоначальной разработки – филиала фирмы IBM в г. Винчестере (Великобритания), где впервые в 1973 г. была применена технология создания винчестеров.

[2] Это справедливо, конечно, не для всех дискет. Однако для современных 3.5" дискет мы действительно имеем 80 цилиндров, две рабочие поверхности и по 18 секторов на каждой дорожке, что в итоге дает нам 2x80x18x512=1474560 байтов (или 1,44 Мбайт).

поверхности диска, cylinder), h – номер рабочей поверхности диска (магнитной головки, head), а 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...H-1, где H – 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. В дальнейшем мы тоже будем пользоваться именно этими обозначениями.

Напомним, что обмен информацией между ОЗУ и дисками физически осуществляется только секторами. Вся совокупность физических секторов на винчестере представляет его неформатированную ёмкость.

Жесткий диск может быть разбит на несколько *разделов* (partition) которые в принципе затем могут использоваться 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, передаётся управление при включении компьютера и загрузке операционной системы. Остальные primary-разделы в этом случае считаются[1] «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; можно это

[1] Невидимыми они являются только для тех систем, которые используют спецификации DOS.

слово перевести и как «простейший, примитивный». В этом случае становится понятным и логичным термин extended.

Один из primary-разделов должен быть *активным,* 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 её 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, она размещается в памяти, начиная с адреса 0:7C00h, после чего управление передаётся коду, содержащемуся в этом самом 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 её хватает только на то, чтобы загрузить следующую, чуть более сложную программу, обычно – стартовый сектор операционной системы – и передать ему управление;

♦ таблица описания разделов диска (partition table). 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. Сигнатура эта выбрана не случайно. Её успешная проверка позволя-

ет 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 эта таблица 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 жёсткий диск был разбит на несколько разделов.

Смещение (Offset)	Размер (Size)	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)

Рис.4.4. Структура MBR

Упрощенно структура MBR представлена на рис. 4.4. Из неё видно, 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 – не активен, 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 System ID (длиной в один байт), указывающий на принадлежность данного раздела к той или иной операционной системе и установке на нём соответствующей файловой системы. В табл. 4.1 приведены некоторые (наиболее известные) идентификаторы.

Таблица 4.1. Сигнатуры (типы) разделов

System ID	Тип раздела	System ID	Тип раздела
00	Empty («пустой» раздел)	41	PPC PreP Boot
01	FAT12	42	SFS
02	XENIX root	4D	QNX 4.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		

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. Ниже представлен формат элемента таблицы разделов.

Таблица 4.2. Формат элемента таблицы разделов

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

Номера сектора и номер цилиндра секторов в разделах занимают по 6 и 10 бит соответственно. Ниже представлен формат записи, содержащей номера сектора и цилиндра.

Биты номера цилиндра										Биты номера сектора					
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Как мы уже сказали, 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 сами разделы (рис. 4.5). В процессе начальной загрузки сектора MBR, содержащего таблицу partition table, работают программные модули BIOS. Начальная загрузка считается выполненной корректно 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.

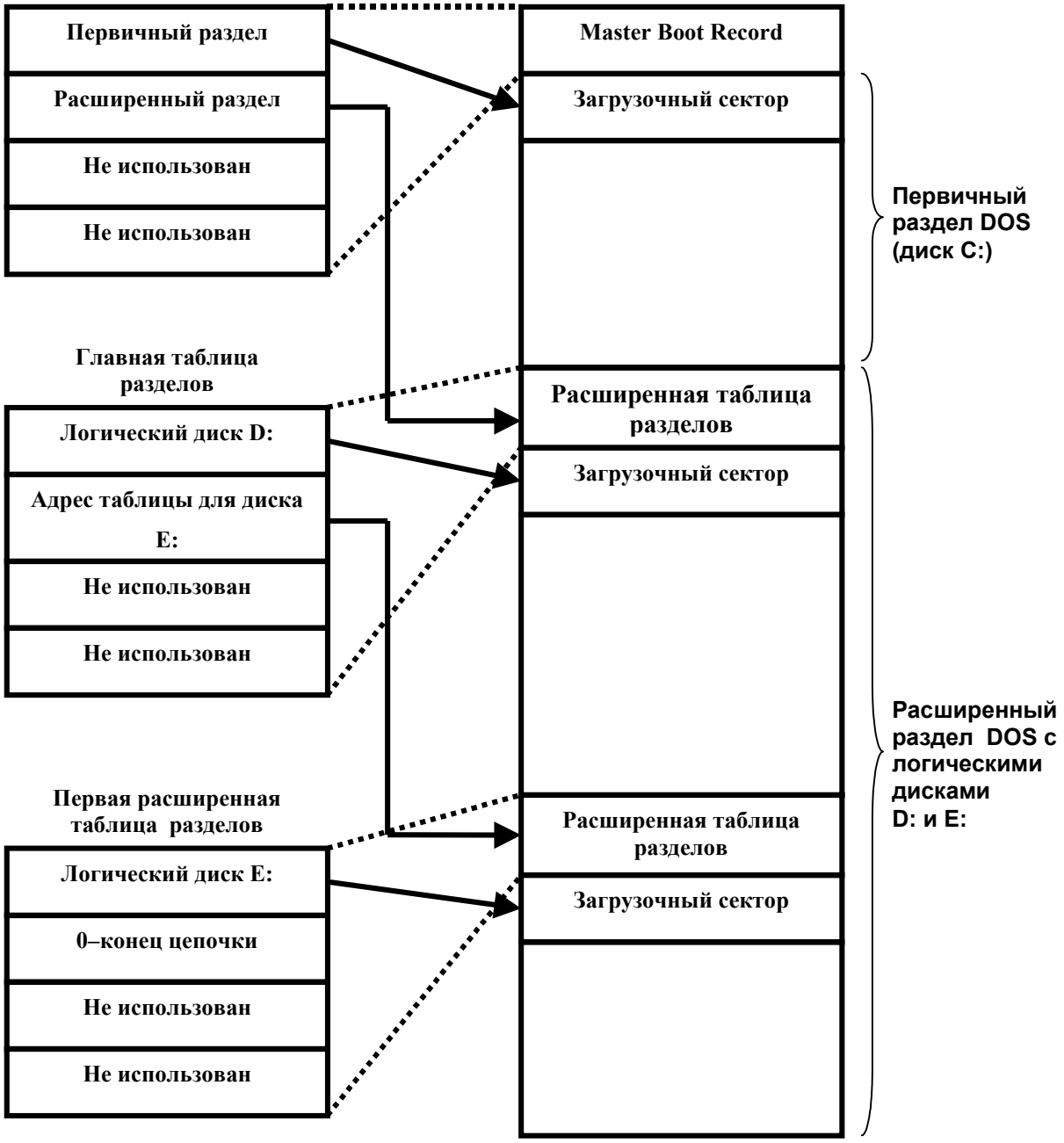

Рис.4.5. Разбиение диска на разделы

Расширенный раздел DOS содержит вторичную запись MBR (secondary MBR, SMBR), в состав которой вместо partition table входит таблица логического диска (LDT, logical disk table), ей аналогичная. 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.

Утилиты, позволяющие разбить диск на разделы и тем самым сформировать как partition table, так и, возможно, списковую структуру из LDT (подобно тому, как это изображено на рис. 4.6), обычно называются FDISK (от form disk), хотя есть утилиты и с другим названием, умеющие выполнить разбиение диска. Напомним, что FDISK от MS-DOS позволяет создать только один primary-раздел и один extended, который, в свою очередь, предлагается разделить на несколько логических дисков. Аналогичные утилиты других ОС имеют существенно больше возможностей. Так, например, FDISK системы OS/2 позволяет создавать несколько primary–разделов, причем их можно выделять не только последовательно, начиная с первых цилиндров, но и с конца свободного дискового пространства. Это удобно, если нужно исключить из работы некоторый диапазон дискового пространства (например, из-за дефектов на поверхности магнитного диска). С помощью этой утилиты можно установить и менеджер загрузки (boot-менеджер).

Прежде чем перейти к boot-менеджерам, рассмотрим ещё раз процесс загрузки ОС. Процедура начальной загрузки (bootstrap loader) вызывается как программное прерывание (BIOS INT 19h). Эта процедура определяет первое готовое устройство из списка разрешенных и доступных (гибкий или жесткий диск, в современных компьютерах это могут быть также CD-ROM, привод ZIP-drive компании Iomega, сетевой адаптер или другое устройство) и пытается загрузить с него в ОЗУ короткую главную программу-загрузчик. Для винчестеров – это загрузчик non-system bootstrap из MBR, и ему передаётся управление. Главный загрузчик определяет на диске активный раздел, загружает его собственный загрузчик (system bootstrap) и передаёт управление ему. И, наконец, этот загрузчик загружает необходимые файлы операционной системы и передаёт ей управление. Далее операционная система выполняет инициализацию подведомственных ей программных и аппаратных средств. Она добавляет новые сервисы, вызываемые, как правило, тоже через механизм программных прерываний, и расширяет (или заменяет) некоторые сервисы BIOS. Необходимо отметить, что в современных мультипрограммных операцион-

ных системах большинство сервисов BIOS, изначально расположенных в ПЗУ, как правило, заменяются собственными драйверами, поскольку они должны работать в режиме прерываний, а не в режиме сканирования готовности.

Соответственно рассмотренному процессу загрузки, мы каждый раз при запуске компьютера будем попадать в одну и ту же ОС. Но иногда это нас не устраивает. Так называемые boot-менеджеры (менеджеры загрузки) предназначены для того, чтобы пользователь мог выбрать среди нескольких установленных на компьютере ОС нужную и передать управление загрузчику этой выбранной ОС. Существует большое количество таких менеджеров, хороший обзор таких программ приведен в работе [78]. В ней рекомендуется в качестве менеджера загрузки использовать System commander. Однако этот менеджер должен располагаться в активном разделе с файловой системой FAT, что в наше время нельзя признать хорошим решением. На наш взгляд, одним из лучших долгое время был Boot manager компании IBM, входящий в состав утилит OS/2. Для его размещения создается отдельный primary-раздел, который, естественно, и является активным (он помечается как *startable).* Раздел для Boot-менеджера OS/2 занимает всего один цилиндр и может размещаться не только на нулевом (начальном) цилиндре, но и на последнем цилиндре. В этом разделе не организуется никакой файловой системы, поскольку обращение к менеджеру идет с использованием абсолютной адресации, а сам Boot manager представляет собой простейшую абсолютную двоичную программу. Установка Boot manager осуществляется из программы FDISK. При этом появляется возможность указать, какие разделы могут быть загружаемыми (они помечаются как bootable), то есть в каких разделах на первом логическом секторе будут размещены программы загрузки ОС. Загружаемыми могут быть как primary-, так и extended-разделы. При этом, естественно, имеется возможность указать как время на принятие решения, так и загрузку некоторой ОС «по умолчанию». Удобным является и то, что при разбиении диска на разделы можно вообще не создавать primary-разделов. Это особенно важно, если в компьютере установлено более одного дискового накопителя либо если мы подготавливаем винчестер, который должен достаточно часто переноситься с одного компьютера на другой. Дело в том,

что в большинстве ОС принято следующее правило именования логических дисков: первый логический диск обозначается литерой C:, второй – D: и т. д. При этом литеру C: получает активный primary-раздел. Однако если к одному винчестеру в персональном компьютере подключить второй винчестер и он тоже имеет активный primary-раздел, то этот primary-раздел второго винчестера получит литеру D:, отодвигая логический диск D: первого винчестера на место диска E: (и т. д. по цепочке). Если же второй (и последующие) винчестер не имеет primary-раздела с установленной на нем файловой системой, которую данная ОС знает, то «именование» логических дисков первого винчестера не нарушается. Естественно, что логические диски второго винчестера получают литеры логических дисков вслед за дисками первого винчестера. Boot manager OS/2, создавая только логические диски на винчестере, на самом деле создает и «пустой» primary-раздел, однако этот раздел не становится активным и не получает статус логического диска. К сожалению, все более популярная в наши дни ОС Windows 2000 теперь не только снимает флаг активности с раздела, в котором размещен Boot manager (как это происходило при инсталляции любых ОС от компании Microsoft), но и физически уничтожает его двоичный код. Замена драйвера FASTFAT.SYS системы Windows 2000 на более раннюю версию (в бета-версии ОС Windows 2000 система не уничтожала код Boot manager) помогает лишь до установки Service pack. Поэтому рекомендовать этот менеджер загрузки уже нельзя. Из последних менеджеров загрузки, пожалуй, наиболее мощным является Boot star, но его нельзя рекомендовать неподготовленным пользователям.

Одной из самых известных и до сих пор достаточно часто используемых утилит, с помощью которой можно посмотреть и отредактировать таблицу разделов, а также выполнить и другие действия, связанные с изучением и исправлением данных как на дискете, так и на винчестере, является программа Disk editor из комплекта нортоновских утилит. Работа с ней достаточно подробно изложена в книге [9]. Следует, однако, заметить, что Disk editor, с одной стороны, очень мощное средство, и поэтому его следует использовать с большой осторожностью, а с другой стороны – эта утилита работает только в среде DOS. На сегодняшний день

главным недостатком этой утилиты является ограничение на максимальный размер диска в 8 Гбайт.

Надо признать, что в последнее время появилось большое количество утилит, которые предоставляют возможность более наглядно представить разбиение диска на разделы, поскольку в них используется графический интерфейс. Эти программы успешно и корректно работают с наиболее распространенными типами разделов (разделы под FAT, FAT32, NTFS). Однако созданы они в основном для работы в среде Win32API, что часто ограничивает возможность их использования. Одной из самых известных и мощных программ для работы с разделами жесткого диска является Partition Magic фирмы Power Quest.

Файловая система FAT

Как мы уже отмечали, аббревиатура FAT (file allocation table) расшифровывается как «таблица размещения файлов». Этот термин относится к линейной табличной структуре со сведениями о файлах – именами файлов, их атрибутами и другими данными, определяющими местонахождение файлов (или их фрагментов) в среде FAT. Элемент FAT определяет фактическую область диска, в которой хранится начало физического файла.

В файловой системе FAT логическое дисковое пространство любого логического диска делится на две области (рис. 4.6): *системную область* и *область данных.*

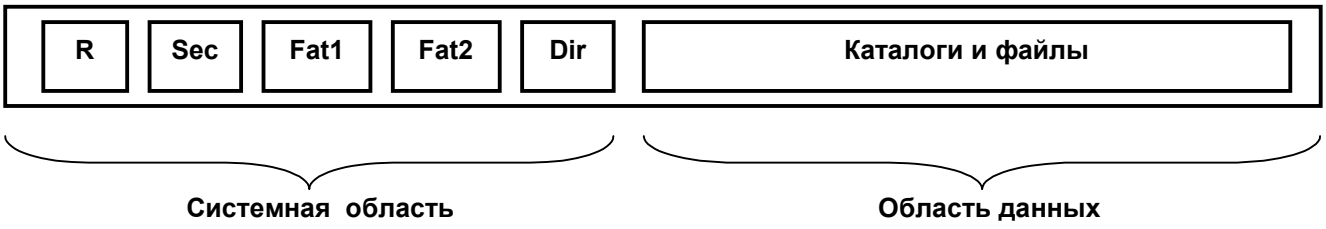

Рис.4.6. Структура логического диска

Системная область логического диска создается и инициализируется при форматировании, а впоследствии обновляется при манипулировании файловой структурой. Область данных логического диска содержит файлы и каталоги, подчинённые корневому. Она, в отличие от системной области, доступна через пользова-

тельский интерфейс DOS. Системная область состоит из следующих компонентов, расположенных в логическом адресном пространстве подряд:

- ♦ загрузочной записи (boot record, BR);
- ♦ зарезервированных секторов (reserved sector, ResSecs);
- ♦ таблицы размещения файлов (file allocation table, FAT);
- ♦ корневого каталога (root directory, RDir).

Таблица размещения файлов

Таблица размещения файлов является очень важной информационной структурой. Можно сказать, что она представляет собой карту (образ) области данных, в которой описывается состояние каждого участка области данных. Область данных разбивают на так называемые кластеры. *Кластер* представляет собой один или несколько смежных секторов в логическом дисковом адресном пространстве (точнее – только в области данных). В таблице FAT кластеры, принадлежащие одному файлу (некорневому каталогу), связываются в цепочки. Для указания номера кластера и системе управления файлами FAT-16 используется 16-битовое слово, следовательно, можно иметь до $2^{16} = 65536$ кластеров (с номерами от 0 до 65535).

Кластер – это минимальная адресуемая единица дисковой памяти, выделяемая файлу (или некорневому каталогу). Файл или каталог занимает целое число кластеров. Последний кластер при этом может быть задействован не полностью, что приведет к заметной потере дискового пространства при большом размере кластера. На дискетах кластер занимает один или два сектора, а на жёстких дисках – в зависимости от объёма раздела (см. табл. 4.3).

Таблица 4.3. Соотношения между размером раздела и размером кластеров в FAT16

Ёмкость раздела, Мбайт	Количество секторов в кластере	Размер кластеров, Кбайт
16-127	4	2
128-255	8	4
256-511	16	8
512-1023	32	16
1024-2047	64	32

Номер кластера всегда относится к области данных диска (пространству, зарезервированному для файлов и подкаталогов). Первый допустимый номер кластера всегда начинается с 2. Номера кластеров соответствуют элементам таблицы размещения файлов.

Логическое разбиение области данных на кластеры как совокупности секторов взамен использования одиночных секторов имеет следующий смысл: прежде всего, уменьшается размер самой таблицы FAT; уменьшается возможная фрагментация файлов; ускоряется доступ к файлу, так как в несколько раз сокращается длина цепочек фрагментов дискового пространства, выделенных для него.

Однако слишком большой размер кластера ведет к неэффективному использованию области данных, особенно в случае большого количества маленьких файлов. Как мы только что заметили, в среднем на каждый файл теряется около половины кластера. Из табл. 4.3 следует, что при размере кластера в 32 сектора (объем раздела – от 512 Мбайт до 1023 Мбайт), то есть 16 Кбайт, средняя величина потерь на файл составит 8 Кбайт, и при числе файлов в несколько тысяч[1] потери могут составлять более 100 Мбайт. Поэтому в современных файловых системах (к ним, прежде всего, следует отнести HPFS, NTFS, FAT32 и некоторые другие, поддерживаемые ОС семейства UNIX) размеры кластеров ограничиваются (обычно – от 512 байт до 4 Кбайт). В FAT32 проблема решается за счёт того, что собственно сама FAT в этой файловой системе может содержать до 2^{28} кластеров[2]. Наконец, заметим, что системы управления файлами, созданные для Windows 9x и Windows NT, могут работать с разделами размером до 4 Гбайт, на которых установлена система FAT, тогда как DOS, естественно, с такими разделами работать не сможет.

Достаточно наглядно идея файловой системы с использованием таблицы размещения файлов FAT проиллюстрирована рис.4.7. Из этого рисунка видно, что

[1] Например, число 10 000-15 000 файлов (или даже более, особенно когда файлы небольшого размера) на логическом диске с объёмом в 1000 Мбайт встречается достаточно часто.

[2] В файловой системе FAT32 в 32-битовом слове, используемом для представления номера кластера, фактически учитываются только 28 разрядов, что приводит к тому, что длина FAT в этой системе не может превышать 2^{28} элементов.

файл с именем MYFILE.TXT размещается, начиная с восьмого кластера. Всего файл MYFILE.TXT занимает 12 кластеров. Цепочка кластеров (chain) для нашего примера может быть записана следующим образом: 8, 9, 0A, 0B, 15, 16, 17, 19, 1A, 1B, 1C, 1D. Кластер с номером 18 помечен специальным кодом F7 как плохой (bad), он не может быть использован для размещения данных. При форматировании обычно проверяется поверхность магнитного диска, и те секторы, при контрольном чтении с которых происходили ошибки, помечаются в FAT как плохие. Кластер 1D помечен кодом FF как конечный (последний в цепочке) кластер, принадлежащий данному файлу. Свободные (незанятые) кластеры помечаются кодом 00; при выделении нового кластера для записи файла берется первый свободный кластер. Поскольку файлы на диске изменяются – удаляются, перемещаются, увеличиваются или уменьшаются, – то упомянутое правило выделения первого свободного кластера для новой порции данных приводит к *фрагментации* файлов, то есть данные одного файла могут располагаться не в смежных кластерах, а, порой, в очень удаленных друг от друга, образуя сложные цепочки. Естественно, что это приводит к существенному замедлению работы с файлами.

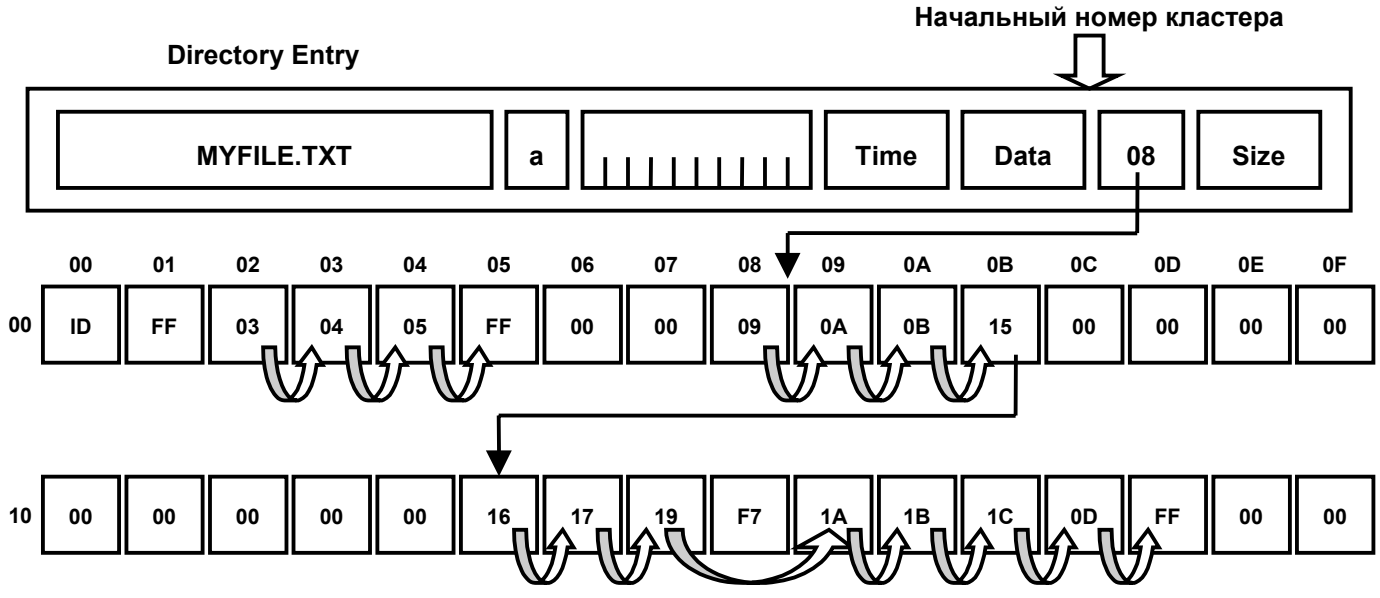

Рис.4.7. Основная концепция FAT

Так как FAT используется при доступе к диску очень интенсивно, она обычно загружается в ОЗУ (в буфера ввода/вывода или кэш) и остается там настолько долго, насколько это возможно.

В связи с чрезвычайной важностью FAT она обычно хранится в двух идентичных экземплярах, второй из которых непосредственно следует за первым. Обновляются копии FAT одновременно. Используется же только первый экземпляр. Если он по каким-либо причинам окажется разрушенным, то произойдет обращение ко второму экземпляру. Так, например, утилита проверки и восстановления файловой структуры ScanDisk из ОС Windows 9x при обнаружении несоответствия первичной и резервной копии FAT предлагает восстановить главную таблицу, используя данные из копии. Упомянутый корневой каталог отличается от обычного каталога тем, что он, помимо размещения в фиксированном месте логического диска, ещё имеет и фиксированное число элементов. Для каждого файла и каталога в файловой системе хранится информация в соответствии со структурой, изображенной в табл.4.4.

Таблица 4.4. Элемент каталога

Размер поля данных, байт	**Содержание поля**
11	**Имя файла или каталога**
1	**Атрибуты файла**
1	**Резервное поле**
3	**Время создания**
2	**Дата создания**
2	**Дата последнего доступа**
2	**Зарезервировано**
2	**Время последней модификации**
2	**Дата последней модификации**
2	**Номер начального кластера в FAT**
4	**Размер файла**

Структура системы файлов является иерархической. Это иллюстрируется рис.4.8, из которого видно, что элементом каталога может быть такой файл, который сам, в свою очередь, является каталогом.

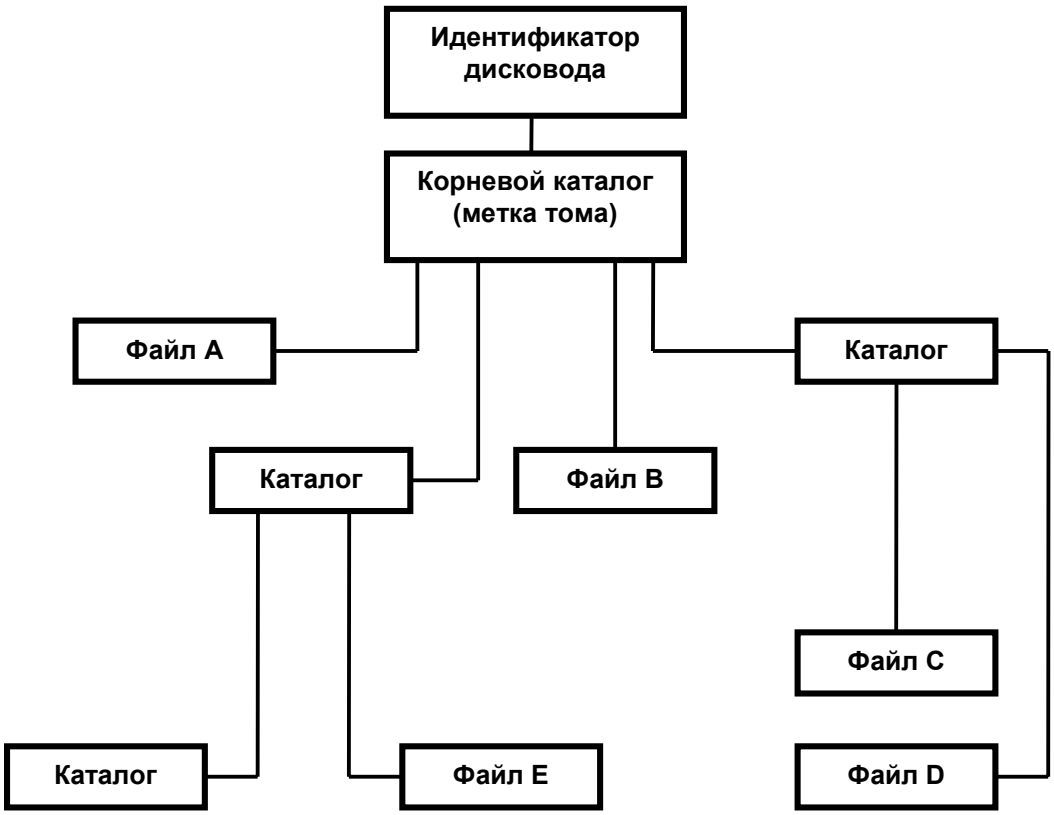

Рис.4.8. Структура системы файлов

Структура загрузочной записи DOS

Сектор, содержащий загрузочную запись, является самым первым на логическом диске (на дискете – имеет физический адрес [0-0-1]). Boot Record состоит, как мы уже знаем, из двух частей – disk parameter block (DPB) и system bootstrap (SB). Структура блока параметров диска (DPB) служит для идентификации физического и логического форматов логического диска, а загрузчик system bootstrap играет существенную роль в процессе загрузки DOS. Эта информационная структура приведена в табл. 4.5.

Первые два байта boot record занимает JMP – команда безусловного перехода в программу SB. Третий байт содержит код 90H (NOP – нет операции). Далее располагается восьмибайтовый системный идентификатор, включающий информацию о фирме-разработчике и версии операционной системы. Затем следует DPB, а после него – SB.

Таблица 4.5. Структура загрузочной записи Boot Record для FAT16

Смещение поля, байт	Длина поля, байт	Обозначение поля	Содержимое поля
00H(0)	3	JUMP 3EH	Безусловный переход на начало SB
03H(3)	8		Системный идентификатор
0BH(11)	2	SectSize	Размер сектора, байт
0DH (13)	1	ClastSize	Число секторов в кластере
0EH (14)	2	ResSecs	Число зарезервированных секторов
10H (16)	1	FATcnt	Число копий FAT
11H (17)	2	RootSize	Максимальное число элементов Rdir
13H (19)	2	TotSecs	Число секторов на логическом диске, если его размер не превышает 32 Мбайт, иначе 0000H
15H (21)	1	Media	Дескриптор носителя
16H (22)	2	FATsize	Размер FAT, секторов
18H (24)	2	TrkSecs	Число секторов на дорожке
1AH (26)	2	HeadCnt	Число рабочих поверхностей
1CH (28)	4	HidnSecs	Число скрытых секторов
20H (32)	4		Число секторов на логическом диске, если его размер превышает 32 Мбайт
24H (36)	1		Тип логического диска (00H – гибкий, 80H – жесткий)
25H (37)	1		Пусто (резерв)
26H (38)	1		Маркер с кодом 29H
27H (39)	4		Серийный номер тома
2BH (43)	11		Метка тома
36H(54)	8		Имя файловой системы
3EH (62)			System bootstrap
1FEH(510)	2		Сигнатура (слово AA55H)

Для работы с загрузочной записью удобно использовать широко известную утилиту Disk Editor из комплекта утилит Питера Нортона. Эта утилита снабжена встроенной системой подсказок и необходимой справочной информацией. Используя её, можно сохранять, модифицировать и восстанавливать загрузочную запись, а также выполнять много других операций. Достаточно подробно работа с этой утилитой описана в книге [9].

Таблица 4.6. Структура загрузочной записи boot record для FAT32

Смещение поля, байт	Длина поля, байт	Обозначение поля	Содержимое поля
00H (0)	3	JUMP 3EH	Безусловный переход на начало SB
03H (3)	8		Системный идентификатор
0BH (11)	2	SectSize	Размер сектора, байт
0DH (13)	1	ClastSize	Число секторов в кластере
0EH (14)	2	ResSecs	Число зарезервированных секторов, для FAT32 равно 32
10H (16)	1	FATcnt	Число копий FAT
11H(17)	2	RootSize	0000H
13H (19)	2	TotSecs	0000H
15H(21)	1	Media	Дескриптор носителя
16H (22)	2	FATsize	0000H
18H (24)	2	TrkSecs	Число секторов на дорожке
1AH (26)	2	HeadCnt	Число рабочих поверхностей
1CH(28)	4	HidnSecs	Число скрытых секторов (располагаются перед загрузочным сектором). Используется при загрузке для вычисления абсолютного смещения корневого каталога и данных
20H(32)	4		Число секторов на логическом диске
24H(36)	4		Число секторов в таблице FAT
28H(37)	2		Расширенные флаги
2AH (38)	2		Версия файловой системы
2CH (39)	4		Номер кластера для первого кластера корневого Каталога
34H(43)	2		Номер сектора с резервной копией загрузочного Сектора
36H (54)	12		Зарезервировано

Загрузочные записи других операционных систем отличаются от рассмотренной. Так, например, в загрузочном секторе для тома с FAT32 в блоке DPB содержатся дополнительные поля, а те поля, что находятся в привычном для системы FAT16 месте, перенесены. Поэтому ОС, в которой имеется возможность работать с файловой системой FAT16, но нет системы управления файлами, понимающей спецификации FAT32, не может читать данные с томов, отформатированных под файловую систему FAT32. В загрузочном секторе для файловой системы FAT32 по-прежнему байты 00H по 0AH содержат команду перехода и OEM ID, а в байтах

0BH по 59H содержатся данные блока DPB. Отличие заключается именно в несколько другой структуре блока DPB; его содержимое приведено[1] в табл. 4.6.

Файловые системы VFAT и FAT32

Одной из важнейших характеристик исходной FAT было использование имён файлов формата «8.3», в котором 8 символов отводится на указание имени файла и 3 символа – для расширения имени. К стандартной FAT (имеется в виду прежде всего реализация FAT16) добавились ещё две разновидности, используемые в широко распространенных операционных системах Microsoft (конкретно – в Windows 95 и Windows NT): VFAT (виртуальная FAT) и FAT32, используемая в одной из редакций ОС Windows 95 и Windows 98. Ныне эта файловая система (FAT32) поддерживается и такими ОС, как Windows Millennium Edition, и всеми ОС семейства Windows 2000. Имеются реализации систем управления файлами для FAT32, Windows NT и ОС Linux.

Файловая система VFAT впервые появилась в Windows for Workgroups 3.11 и была предназначена для выполнения файлового ввода/вывода в защищённом режиме (см. раздел «Реальный и защищенный режимы работы процессора», глава 3). С выходом Windows 95 в VFAT добавилась поддержка длинных имён файлов (long file name, LFN). Тем не менее, VFAT сохраняет совместимость с исходным вариантом FAT; это означает, что наряду с длинными именами в ней поддерживаются имена формата «8.3», а также существует специальный механизм для преобразования имен «8.3» в длинные имена, и наоборот. Именно файловая система VFAT поддерживается исходными версиями Windows 95, Windows NT 4. При работе с VFAT крайне важно использовать файловые утилиты, поддерживающие VFAT вообще и длинные имена в частности. Дело в том, что более ранние файловые утилиты DOS запросто модифицируют то, что кажется им исходной структурой FAT. Это может привести к потере или порче длинных имен из таблицы FAT, поддерживаемой VFAT (или FAT32). Следовательно, для томов VFAT необходимо пользо-

[1] Поскольку файловая система FAT32 нынче имеет очень широкое распространение, мы считаем, что информация, приведенная в табл. 4.6, может быть полезна.

ваться файловыми утилитами, которые понимают и сохраняют файловую структуру VFAT.

В исходной версии Windows 95 основной файловой системой была 32–разрядная VFAT. VFAT может использовать 32-разрядные драйверы защищённого режима или 16-разрядные драйверы реального режима. При этом элементы FAT остаются 12- или 16-разрядными, поэтому на диске используется та же структура данных, что и в предыдущих реализациях FAT. VFAT обрабатывает все обращения к жёсткому диску и использует 32-разрядный код для всех файловых операций с дисковыми томами.

Основными недостатками файловых систем FAT и VFAT являются большие потери на кластеризацию при больших размерах логического диска и ограничения на сам размер логического диска. Это привело к разработке новой реализации файловой системы с использованием той же идеи использования таблицы FAT. Поэтому в Microsoft Windows 95 OEM Service Release 2 (эта версия Windows 95 часто называется Windows 95 OSR2) на смену системе VFAT пришла файловая система FAT32. FAT32 является полностью самостоятельной 32-разрядной файловой системой и содержит многочисленные усовершенствования и дополнения по сравнению с предыдущими реализациями FAT.

Принципиальное отличие заключается в том, что FAT32 намного эффективнее расходует дисковое пространство. Прежде всего, система FAT32 использует кластеры меньшего размера по сравнению с предыдущими версиями, которые ограничивались 65 535 кластерами на том (соответственно, с увеличением размера диска приходилось увеличивать и размер кластеров). Следовательно, даже для дисков размером до 8 Гбайт FAT32 может использовать 4-килобайтные кластеры. В результате по сравнению с дисками FAT 16 экономится значительное дисковое пространство (в среднем 10-15 %) [53].

FAT32 также может перемещать корневой каталог и использовать резервную копию FAT вместо стандартной. Расширенная загрузочная запись FAT32 позволяет создавать копии критических структур данных; это повышает устойчивость дисков FAT32 к нарушениям структуры FAT по сравнению с предыдущими версиями.

Корневой каталог в FAT32 представлен в виде обычной цепочки кластеров. Следовательно, корневой каталог может находиться в произвольном месте диска, что снимает действовавшее ранее ограничение на размер корневого каталога (512 элементов). Windows 95 OSR2 и Windows 98 могут работать и с разделами VFAT, созданными Windows NT. То, что говорилось ранее об использовании файловых утилит VFAT с томами VFAT, относится и к FAT32. Поскольку прежние утилиты FAT (для FAT32 в эту категорию входят обе файловые системы, FAT и VFAT) могут повредить или уничтожить важную служебную информацию, для томов FAT32 нельзя пользоваться никакими файловыми утилитами, кроме утилит FAT32.

Кроме повышения ёмкости FAT до величины в 4 Тбайт, файловая система FAT32 вносит ряд необходимых усовершенствований в структуру корневого каталога. Предыдущие реализации требовали, чтобы вся информация корневого каталога FAT находилась в одном дисковом кластере. При этом корневой каталог мог содержать не более 512 файлов. Необходимость представлять длинные имена и обеспечить совместимость с прежними версиями FAT привела разработчиков компании Microsoft к компромиссному решению: для представления длинного имени они стали использовать элементы каталога, в том числе и корневого. Рассмотрим способ представления в VFAT длинного имени файла (рис. 4.9).

Первые одиннадцать байтов элемента каталога DOS используются для хранения имени файла. Каждое такое имя разделяется на две части: в первых восьми байтах хранятся символы собственно имени, а в последних трех – символы так называемого расширения, с помощью которого реализуются механизмы предопределенных типов. Были введены соответствующие системные соглашения, и файлы определенного типа необходимо (желательно) именовать с оговоренным расширением. Например, исполняемые файлы с расширением COM определяют исполняемую двоичную программу с простейшей односегментной структурой[1]. Более сложные программы имеют расширение EXE. Определенны расширения для большого количества типов файлов, и эти расширения используются для ассоциированного запуска обрабатывающих файлы программ.

[1] Для программных модулей, имеющих такую структуру, может использоваться и расширение BIN.

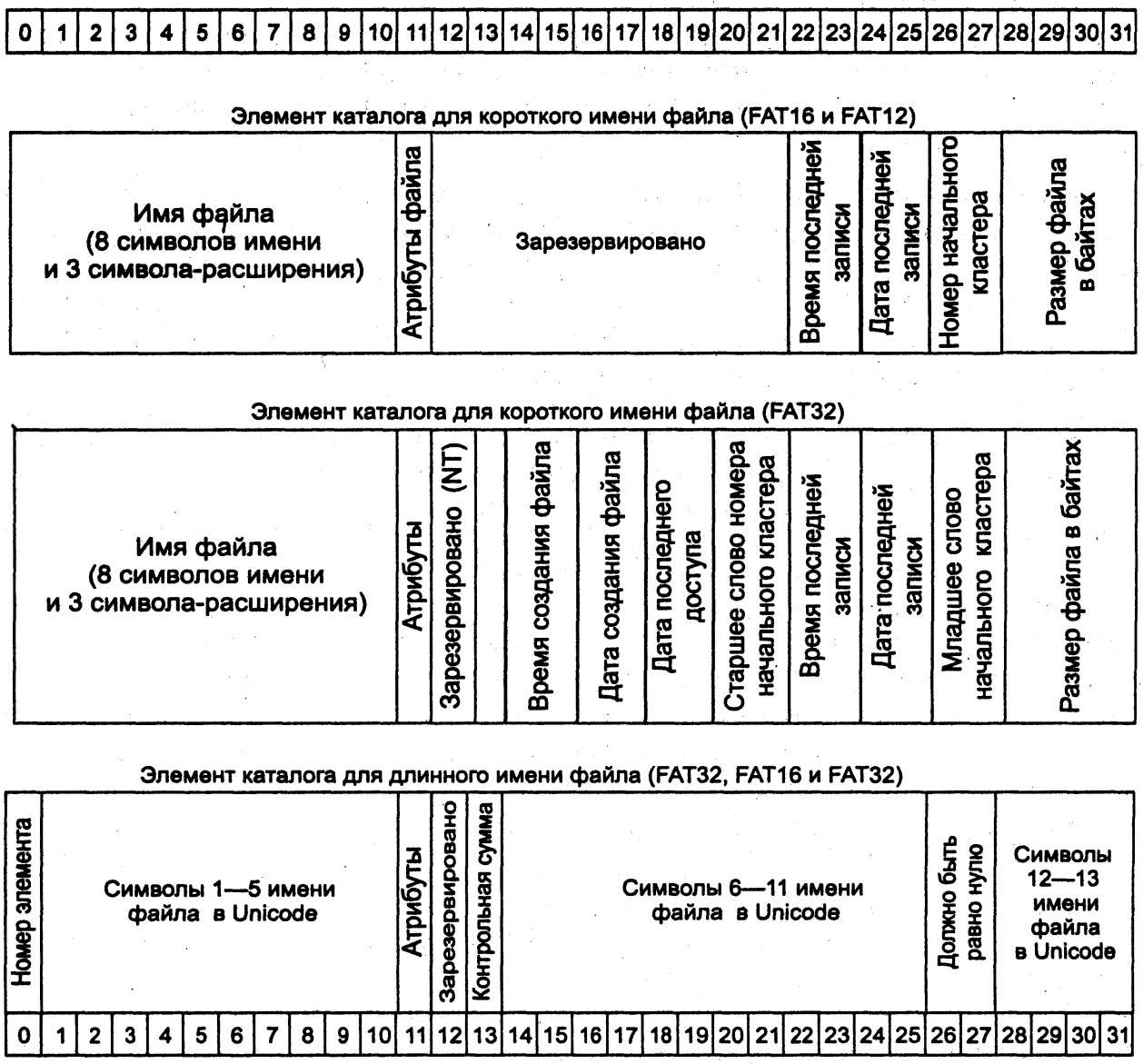

Рис.4.9. Элементы каталогов для FAT, VFAT и FAT32

Если имя файла состоит менее чем из восьми символов, то в элементе каталога оно дополняется символами пробела, чтобы полностью заполнить все восемь байтов соответствующего поля. Аналогично и расширение может содержать от нуля до трех символов. Остальные (незаполненные) позиции в элементе каталога, определяющем расширение имени файла, заполняются символами пробела. Поскольку при работе с именем файла учитываются все одиннадцать свободных мест, то необходимость в отображении точки, которая обычно вводится между именем файла и его расширением, отпадает. В элементе каталога она просто подразумевается.

В двенадцатом байте элемента каталога хранятся *атрибуты файла*. Шесть из восьми указанных разрядов используются DOS[1]. К атрибутам DOS относятся следующие:

♦ атрибут «архивный» (A – archive). Показывает, что файл был открыт программой таким образом, чтобы у неё была возможность изменить содержимое этого файла. DOS устанавливает этот разряд атрибута в состояние ON (включено) при открытии файла. Программы резервного копирования нередко устанавливают его в OFF (выключено) при выполнении резервного копирования файла. Если применяется подобная методика, то в следующую создаваемую по порядку резервную копию будут добавлены только файлы с данным разрядом, установленным в состояние ON;

♦ атрибут каталога (D – directory). Показывает, что данный элемент каталога указывает на подкаталог, а не на файл;

♦ атрибут тома (V – volume). Применяется только к одному элементу каталога в корневом каталоге. В нём, собственно, и хранится имя дискового тома. Этот атрибут также применяется в случае длинных имён файлов, о чём можно будет узнать из следующего раздела;

♦ атрибут «системный» (S – system). Показывает, что файл является частью операционной системы или специально отмечен подобным образом прикладной программой, что иногда делается в качестве составной части метода защиты от копирования;

♦ атрибут «скрытый» (H – hidden). Сюда относятся, в частности, файлы с установленным в состояние ON атрибутом «системный» (S), которые не отображаются в обычном списке, выводимом по команде DIR;

♦ атрибут «только для чтения» (R – read only). Показывает, что данный файл не подлежит изменению. Разумеется, поскольку это лишь разряд байта, хранящегося на диске, то любая программа может изменить этот разряд, после чего DOS свободно разрешила бы изменение данного файла. Этот атрибут в основном

[1] В некоторых операционных системам, в частности в Novell Netware, используется один или два дополнительных разряда атрибутов.

используется для примитивной защиты от пользовательских ошибок, то есть он позволяет избежать неумышленного удаления или изменения ключевых файлов.

Следует отметить, что наличие файла, помеченного одним или более из указанных выше атрибутов, может иметь вполне определенный смысл. Например, большинство файлов, отмечаемых в качестве системных, отмечаются также признаками скрытых файлов и «только для чтения».

На дисках FAT12 или FAT16 следующие за именем десять байтов не используются. Обыкновенно они заполняются нулями и считаются резервными значениями. А на диске с файловой системой FAT32 эти 10 байт содержат самую разную информацию о файле. При этом байт, отмеченный как «зарезервировано для NT», представляет собой, как подразумевает его название, поле, не используемое в DOS или Windows 9x, но применяемое в Windows NT.

Из соображений совместимости поля, которые встречаются в элементах каталога для коротких имен формата FAT12 и FAT16, находятся на тех же местах и в элементах каталога для коротких имен формата FAT32. Остальные поля, которые встречаются только в элементах каталога для коротких имен формата FAT32, соответствуют зарезервированной области длиной в десять байт в элементах каталога для коротких имен форматов FAT12 и FAT16.

Как видно из рис.4.9, для длинного имени файла используется несколько элементов каталога. Таким образом, появление длинных имён фактически привело к дальнейшему уменьшению количества файлов, которые могут находиться в корневом каталоге. Поскольку длинное имя может содержать до 256 символов, всего один файл с полным длинным именем занимает до 25 элементов FAT ,(1 для имени 8.3 и ещё 24 для самого длинного имени). Количество элементов корневого каталога VFAT уменьшается до 21. Очевидно, что это не самое изящное решение, поэтому компания Microsoft советует избегать длинных имен в корневых каталогах FAT при отсутствии FAT32, у которой количество элементов каталога соответственно просто увеличено. Помните и о том, что длина полной файловой спецификации, включающей путь и имя файла (длинное или в формате 8.3), тоже ограничивается 260 символами. FAT32 успешно справляется с проблемой длинных имён в корне-

вом каталоге, но проблема с ограничением длины полной файловой спецификации остаётся. По этой причине Microsoft рекомендует ограничивать длинные имена 75–80 символами, чтобы оставить достаточно места для пути (180–185 символов).

Файловая система HPFS

Сокращение HPFS расшифровывается как «High Performance File System» – высокопроизводительная файловая система. HPFS впервые появилась в OS/2 1.2 и LAN Manager. HPFS была разработана совместными усилиями лучших специалистов компании IBM и Microsoft на основе опыта IBM по созданию файловых систем MVS, VM/CMS и виртуального метода доступа[1]. Архитектура HPFS начала создаваться как файловая система, которая сможет использовать преимущества многозадачного режима и обеспечит в будущем более эффективную и надёжную работу с файлами на дисках большого объёма.

HPFS была первой файловой системой для ПК, в которой была реализована поддержка длинных имен [96]. HPFS, как FAT и многие другие файловые системы, обладает структурой каталогов, но в ней также предусмотрены автоматическая сортировка каталогов и специальные расширенные атрибуты[2], упрощающие реализацию безопасности файлового уровня и создание множественных имен. HPFS поддерживает те же самые атрибуты, что и файловая система FAT, по историческим причинам, но также поддерживает и новую форму file-associated, то есть информацию, называемую *расширенными атрибутами* (EAs[3]). Каждый EA концептуально подобен переменной окружения. Но самым главным отличием всё же являются базовые принципы хранения информации о местоположении файлов.

Принципы размещения файлов на диске, положенные в основу HPFS, увеличивают как производительность файловой системы, так и её надёжность и отказоустойчивость. Для достижения этих целей предложено несколько способов: размещение каталогов в середине дискового пространства, использование методов бинарных сбалансированных деревьев для ускорения поиска информации о файле,

[1] Так, со стороны компании Microsoft проектом руководил известный системный программист Gordon Letwin.
[2] *Расширенные атрибуты* (extended attributes, EAs) позволяют хранить дополнительную информацию о файле. Например, каждому файлу может быть сопоставлено его уникальное графическое изображение (значок или «фотография»), описание файла, комментарий, сведения о владельце файла и т. д.
[3] *Extended Atributes* (EAs) – расширенные атрибуты.

рассредоточение информации о местоположении записей файлов по всему диску, при том, что записи каждого конкретного файла размещаются (по возможности) в смежных секторах и поблизости от данных об их местоположении. Действительно, система HPFS стремится, прежде всего, к тому, чтобы расположить файл в смежных кластерах, или, если такой возможности нет, разместить его на диске таким образом, чтобы *экстенты*[4] (фрагменты) файла физически были как можно ближе друг к другу. Такой подход существенно уменьшает *время позиционирования* головок записи/чтения жёсткого диска и *время ожидания* (rotational latency) – задержку между установкой головки чтения/записи на нужную дорожку диска и началом чтения данных с диска[5]. Можно сказать, что файловая система HPFS имеет, по сравнению с FAT, следующие основные преимущества:

- ♦ высокая производительность;
- ♦ надёжность;
- ♦ работа с расширенными атрибутами, что позволяет управлять доступом к файлам и каталогам;
- ♦ эффективное использование дискового пространства.

Все эти преимущества обусловлены структурой диска HPFS. Рассмотрим её более подробно (рис. 4.10).

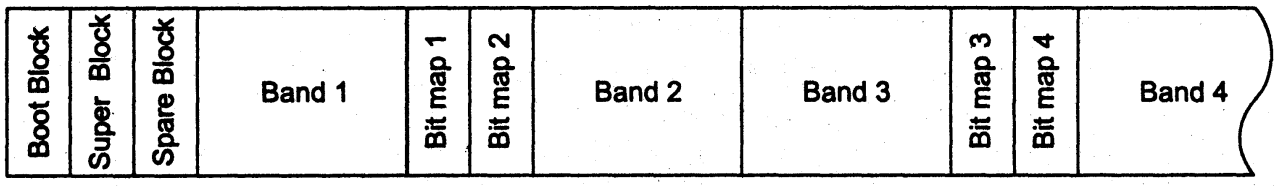

Рис.4.10. Структура раздела HPFS

В начале диска расположено несколько управляющих блоков. Всё остальное дисковое пространство в HPFS разбито на части («полосы», «ленты» из смежных секторов, в оригинале – band). Каждая такая группа данных занимает на диске пространство в 8 Мбайт и имеет свою собственную битовую карту распределения секторов. Эти битовые карты показывают, какие секторы данной полосы заняты, а ка-

[4] *Экстенты* (extent) – фрагменты файла, располагающиеся в смежных секторах диска. Файл имеет по крайней мере один экстент, если он не фрагментирован, а в противном случае – несколько экстентов.

кие – свободны. Каждому сектору ленты данных соответствует один бит в её битовой карте. Если бит имеет значение 1, то соответствующий сектор занят, а если 0 – свободен.

Битовые карты двух полос располагаются на диске рядом, так же располагаются и сами полосы. То есть последовательность полос и карт выглядит следующим образом: битовая карта, битовая карта, лента с данными, лента с данными, битовая карта, битовая карта и т. д. Такое расположение «лент» позволяет непрерывно разместить на жёстком диске файл размером до 16 Мбайт и в то же время не удалять от самих файлов информацию об их местонахождении. Это иллюстрируется рис. 4.10.

Очевидно, что если бы на весь диск была только одна битовая[1] карта, как это сделано в FAT, то для работы с ней приходилось бы перемещать головки чтения/записи в среднем через половину диска. Именно для того, чтобы избежать этих потерь, в HPFS и разбит диск на «полосы». Получается своего рода распределенная структура данных об используемых и свободных блоках.

Дисковое пространство в HPFS выделяется не кластерами, как в FAT, а *блоками.* В современной реализации размер блока взят разным одному сектору, но в принципе он мог бы быть и иного размера. По сути дела, блок – это и есть кластер. Размещение файлов в таких небольших блоках позволяет более эффективно использовать пространство диска, так как непроизводительные потери свободного места составляют в среднем всего 256 байт на каждый файл. Вспомните, что чем больше размер кластера, тем больше места на диске расходуется напрасно. Например, кластер на отформатированном под FAT диске объёмом от 512 до 1024 Мбайт имеет размер 16 Кбайт. Следовательно, непродуктивные потери свободного пространства на таком разделе в среднем составляют 8 Кбайт (8192 байт) на один файл, в то время как на разделе HPFS эти потери всегда будут составлять всего 256 байт на файл. Таким образом, на каждый файл экономится почти 8Кбайт.

---

[5] Эта задержка обусловлена тем, что система вынуждена ждать, пока диск не повернется таким образом, что нужный сектор окажется под головкой чтения/записи.

[1] *Bit map* – битовая карта.

На рис. 4.10 показано, что помимо «лент» с записями файлов и битовых карт в *томе*[1] с HPFS имеются ещё три информационные структуры. Это так называемый загрузочный блок (boot block), дополнительный блок (super block) и запасной (резервный) блок (spare block). Загрузочный блок (boot block) располагается в секторах с 0 по 15; он содержит имя тома, его серийный номер, блок параметров BIOS[2] и программу начальной загрузки. Программа начальной загрузки находит файл OS2LDR, считывает его в память и передаёт управление этой программе загрузки ОС, которая, в свою очередь, загружает с диска в память ядро OS/2 - OS2KRNL. И уже OS2KRNL с помощью сведений из файла CONFIG.SYS загружает в память все остальные необходимые программные модули и блоки данных.

В блоке (super block) содержится указатель на список битовых карт (bitmap block list). В этом списке перечислены все блоки на диске, в которых расположены битовые карты, используемые для обнаружения свободных секторов. Также в дополнительном блоке хранится указатель на список дефектных блоков (bad block list), указатель на группу каталогов (directory b and), указатель на файловый узел (F-node) корневого каталога, а также дата последней проверки раздела программой CHKDSK. В списке дефектных блоков перечислены все поврежденные секторы (блоки) диска. Когда система обнаруживает повреждённый блок, он вносится в этот список и для хранения информации больше не используется. Кроме этого, в структуре super block содержится информация о размере «полосы». Напомним, что в текущей реализации HPFS размер «полосы» взят равным 8 Мбайт. Блок super block размещается в секторе с номером 16 логического диска, на котором установлена файловая система HPFS.

Резервный блок (spare block) содержит указатель на карту аварийного замещения (hotfix map или hotfix-areas), указатель на список свободных запасных блоков (directory emergency free block list), используемых для операций на почти переполненном диске, и ряд системных флагов и дескрипторов. Этот блок размещается в

[1] По сути дела, *том* (volume) – это не что иное, как раздел или логический диск.

[2] Блок параметров BIOS содержит информацию о жестком диске – количество цилиндров и головок диска, число секторов на дорожке. Эта информация используется программными модулями HPFS для поиска конкретного сектора (блока), поскольку все блоки пронумерованы 32-разрядными числами.

17 секторе диска. Резервный блок обеспечивает высокую отказоустойчивость файловой системы HPFS и позволяет восстанавливать повреждённые данные на диске.

Файлы и каталоги в HPFS базируются на фундаментальном объекте, называемом F-Node[1]. Эта структура характерна для HPFS и аналога в файловой системе FAT не имеет. Каждый файл и каталог диска имеет свой файловый узел F-Node. Каждый объект F-Node занимает один сектор и всегда располагается поблизости от своего файла или каталога (обычно – непосредственно перед файлом или каталогом). Объект F-Node содержит длину и первые 15 символов имени файла, специальную служебную информацию, статистику по доступу к файлу, расширенные атрибуты файла и список прав доступа[2] (или только часть этого списка, если он очень большой), ассоциативную информацию о расположении и подчинении файла и т. д. Структура распределения в F-node может принимать несколько форм в зависимости от размера каталога или файлов. HPFS просматривает файл как совокупность одного или более секторов. Из прикладной программы это не видно; файл появляется как непрерывный поток байтов. Если расширенные атрибуты слишком велики для файлового узла, то в него записывается указатель на них.

Сокращенное имя файла (в формате 8.3) используется, когда файл с длинным именем копируется или перемещается на диск с системой FAT, не допускающей подобных имён. Сокращенное имя образуется из первых 8 символов оригинального имени файла, точки и первых трех символов расширения имени, если расширение имеется. Если в имени файла присутствует несколько точек, что не противоречит правилам именования файлов в HPFS, то для расширения сокращенного имени используются три символа после самой последней из этих точек.

Так как HPFS при размещении файла на диске стремится избежать его фрагментации, то структура информации, содержащаяся в файловом узле, достаточно проста. Если файл непрерывен, то его размещение на диске описывается двумя 32–битными числами. Первое число представляет собой указатель на первый блок

[1] *Файловый узел* (F-Node) – это структура, в которой содержится информация о расположении файла и о его расширенных атрибутах.

[2] *ACL* (access control list) – список прав доступа к данному файлу или объекту.

файла, а второе – длину экстента, то есть число следующих друг за другом блоков, принадлежащих файлу[1]. Если файл фрагментирован, то размещение его экстентов описывается в файловом узле дополнительными парами 32-битных чисел. Фрагментация происходит, когда на диске нет непрерывного свободного участка, достаточно большого, чтобы разместить файл целиком. В этом случае файл приходится разбивать на несколько экстентов и располагать их на диске раздельно. Файловая система HPFS старается разместить экстенты фрагментированного файла как можно ближе друг к другу, чтобы сократить время позиционирования головок чтения/записи жесткого диска. Для этого HPFS использует статистику, а также старается условно резервировать хотя бы 4 килобайта места в конце файлов, которые растут. Ещё один способ уменьшения фрагментирования файлов – это расположение файлов, растущих навстречу друг другу, или файлов, открытых разными тредами или процессами, в разных полосах диска.

В файловом узле можно разместить информацию максимум о восьми экстентах файла. Если файл имеет больше экстентов, то в его файловый узел записывается указатель на блок размещения (allocation block), который может содержать до 40 указателей на экстенты или, по аналогии с блоком дерева каталогов, на другие блоки размещения. Таким образом, двухуровневая структура блоков размещения может хранить информацию о 480 секторах, что позволяет работать с файлами размером до 7,68 Гбайт. На практике размер файла не может превышать 2 Гбайт, но это обусловлено текущей реализацией интерфейса прикладного программирования [96].

«Полоса», находящаяся в центре диска, используется для хранения каталогов. Эта полоса называется directory band. Как и все остальные «полосы», она имеет размер 8 Мбайт. Однако, если она будет полностью заполнена, HPFS начинает располагать каталоги файлов в других полосах. Расположение этой информационной структуры в середине диска значительно сокращает среднее время позиционирования головок чтения/записи. Действительно, для перемещения головок чтения/записи из произвольного места диска в его центр требуется в два раза

[1] Из этого следует, что максимальный объём диска может составлять (2^{32}–l)x512=2 Тбайта.

меньше времени, чем для перемещения к краю диска, где находится корневой каталог в случае файловой системы FAT. Уже только одно это обеспечивает более высокую производительность файловой системы HPFS по сравнению с FAT. Аналогичное замечание справедливо и для NTFS, которая тоже располагает свой master file table в начале дискового пространства, а не в его середине.

Однако существенно больший (по сравнению с размещением Directory Band в середине логического диска) вклад в производительность HPFS дает использование метода *сбалансированных двоичных деревьев* для хранения и поиска информации о местонахождении файлов. Как известно, в файловой системе FAT каталог имеет линейную структуру, специальным образом не упорядоченную, поэтому при поиске файла требуется последовательно просматривать его с самого начала. В HPFS структура каталога представляет собой сбалансированное дерево с записями, расположенными в алфавитном порядке (рис. 4.11). Каждая запись, входящая в состав B-Tree дерева, содержит атрибуты файла, указатель на соответствующий файловый узел, информацию о времени и дате создания файла, времени и дате последнего обновления и обращения, длине данных, содержащих расширенные атрибуты, счётчик обращений к файлу, длине имени файла и само имя, и другую информацию.

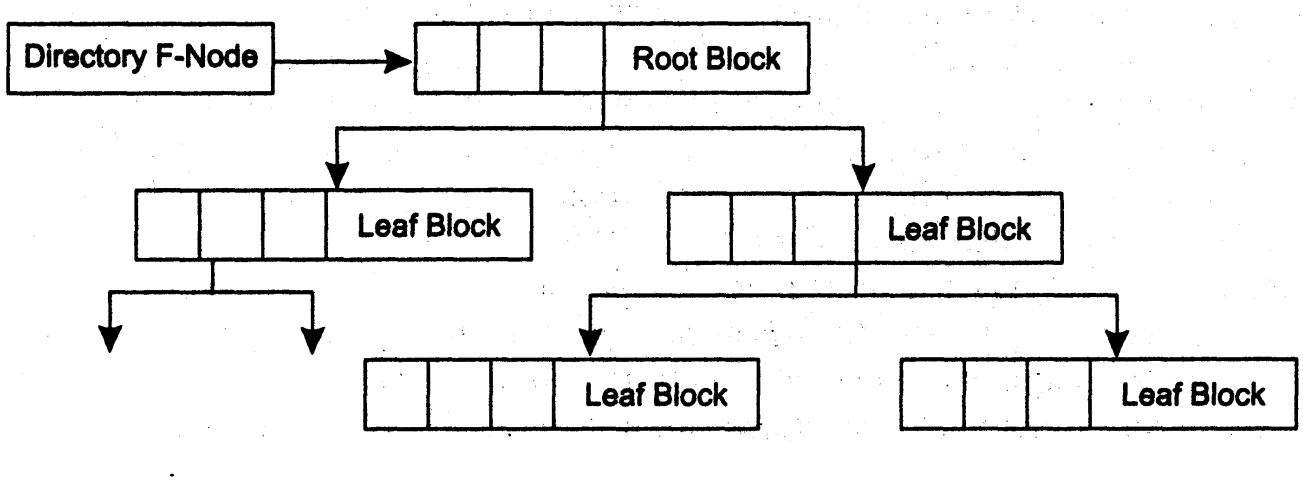

Рис.4.11. Сбалансированное двоичное дерево

Файловая система HPFS при поиске файла в каталоге просматривает только необходимые ветви двоичного дерева (B-Tree). Такой метод во много раз эффективнее, чем последовательное чтение всех записей в каталоге, что имеет место в

системе FAT. Для того чтобы найти искомый файл в каталоге (точнее, указатель на его информационную структуру F-node), организованном на принципах сбалансированных двоичных деревьев, большинство записей вообще читать не нужно. В результате для поиска информации о файле необходимо выполнить существенно меньшее количество операций чтения диска.

Действительно, если, например, каталог содержит 4096 файлов, то файловая система FAT потребует чтения в среднем 64 секторов для поиска нужного файла внутри такого каталога, в то время как HPFS осуществит чтение всего только 2-4 секторов (в среднем) и найдёт искомый файл. Несложные расчёты позволяют увидеть явные преимущества HPFS над FAT. Так, например, при использовании 40 входов на блок блоки каталога дерева с двумя уровнями могут содержать 1640 входов, а каталога дерева с тремя уровнями – уже 65 640 входов. Другими словами, некоторый файл может быть найден в типичном каталоге из 65 640 файлов максимум за три обращения. Это намного лучше файловой системы FAT, где для нахождения файла нужно прочитать в худшем случае более 4000 секторов.

Размер каждого из блоков, в терминах которых выделяются каталоги в текущей реализации HPFS, равен 2 Кбайт. Размер записи, описывающей файл, зависит от размера имени файла. Если имя занимает 13 байтов (для формата 8.3), то блок из 2 Кбайт вмещает до 40 описателей файлов. Блоки связаны друг с другом посредством списковой структуры (как и описатели экстентов) для облегчения последовательного обхода.

При переименовании файлов может возникнуть так называемая перебалансировка дерева. Создание файла, переименование или стирание может приводить к каскадированию блоков каталогов. Фактически, переименование может потерпеть неудачу из-за недостатка дискового пространства, даже если файл непосредственно в размерах не увеличился. Во избежание этого «бедствия» HPFS поддерживает небольшой пул свободных блоков, которые могут использоваться при «аварии». Эта операция может потребовать выделения дополнительных блоков на заполненном диске. Указатель на этот пул свободных блоков сохраняется в SpareBlock.

Важное значение для повышения скорости работы с файлами имеет уменьшение их фрагментации. В HPFS считается, что файл является фрагментированным, если он содержит больше одного экстента. Снижение фрагментации файлов сокращает время позиционирования и время ожидания за счёт уменьшения количества перемещений головок, необходимого для доступа к данным файла. Алгоритмы работы файловой системы HPFS работают таким образом, чтобы по возможности размещать файлы в последовательных смежных секторах диска, что обеспечивает максимально быстрый доступ к данным впоследствии. В системе FAT, наоборот, запись следующей порции данных в первый же свободный кластер неизбежно приводит к фрагментации файлов. HPFS тоже, если это предоставляется возможным, записывает данные в смежные секторы диска (но не в первый попавшийся). Это позволяет несколько снизить число перемещений головок чтения/записи от дорожки к дорожке. При этом, когда данные дописываются в существующий файл, HPFS сразу же резервирует как минимум 4 Кбайт непрерывного пространства на диске. Если же часть этого пространства не потребовалась, то после закрытия файла она высвобождается для дальнейшего использования. Файловая система HPFS равномерно размещает непрерывные файлы по всему диску для того, чтобы впоследствии без фрагментации обеспечить их возможное увеличение. Если же файл не может быть увеличен без нарушения его непрерывности, HPFS опять-таки резервирует 4Кбайт смежных блоков как можно ближе к основной части файла с целью сократить время позиционирования головок чтения/записи и время ожидания соответствующего сектора.

Очевидно, что степень фрагментации файлов на диске зависит как от числа файлов, расположенных на нём, их размеров и размеров самого диска, так и от характера и интенсивности самих дисковых операций. Незначительная фрагментация файлов практически не сказывается на быстродействии операций с файлами. Файлы, состоящие из двух-трех экстентов, практически не снижают производительность HPFS, так как эта файловая система следит за тем, чтобы области данных, принадлежащие одному и тому же файлу, располагались как можно ближе друг к другу. Файл из трех экстентов имеет только два нарушения непрерывности, и, сле-

довательно, для его чтения потребуется всего лишь два небольших перемещения головки диска. Программы (утилиты) дефрагментации, имеющиеся для этой файловой системы, по умолчанию считают наличие двух-трех экстентов у файла нормой. Например, программа HPFSOPT из набора утилит Gamma-Tech по умолчанию не дефрагментирует файлы, состоящие из трех и менее экстентов, а файлы, которые имеют большее количестве экстентов, приводятся к 2 или 3 экстентам, если это возможно (файлы объёмом в несколько десятков мегабайт всегда будут фрагментированы, ибо максимально возможный размер экстента, как вы помните, равен 8 Мбайт). Надо сказать, что практика показывает, что в среднем на диске имеется не более 2 процентов файлов, имеющих три и более экстентов [96]. Даже общее количество фрагментированных файлов, как правило, не превышает 3 процентов. Такая ничтожная фрагментация оказывает пренебрежимо малое влияние на общую производительность системы.

Теперь кратко рассмотрим вопрос надёжности хранения данных в HPFS. Любая файловая система должна обладать средствами исправления ошибок, возникающих при записи информации на диск. Система HPFS для этого использует *механизм аварийного замещения* (hotfix).

Если файловая система HPFS сталкивается с проблемой в процессе записи данных на диск, она выводит на экран соответствующее сообщение об ошибке. Затем HPFS сохраняет информацию, которая должна была быть записана в дефектный сектор, в одном из запасных секторов, заранее зарезервированных на этот случай. Список свободных запасных блоков хранится в резервном блоке HPFS. При обнаружении ошибки во время записи данных в нормальный блок HPFS выбирает один из свободных запасных блоков и сохраняет эти данные в нём. Затем файловая система обновляет карту аварийного замещения в резервном блоке. Эта карта представляет собой просто пары двойных слов, каждое из которых является 32–битным номером сектора. Первый номер указывает на дефектный сектор, а второй – на тот сектор среди имеющихся запасных секторов, который был выбран для его замены. После замены дефектного сектора запасным карта аварийного замещения записывается на диск, и на экране появляется всплывающее окно, информирующее

пользователя о произошедшей ошибке записи на диск. Каждый раз, когда система выполняет запись или чтение сектора диска, она просматривает карту аварийного замещения и подменяет все номера дефектных секторов номерами запасных секторов с соответствующими данными. Следует заметить, что это преобразование номеров существенно не влияет на производительность системы, так как оно выполняется только при физическом обращении к диску, но не при чтении данных из дискового кэша. Очистка карты аварийного замещения автоматически выполняется программой CHKDSK при проверке диска HPFS. Для каждого замещённого блока (сектора) программа CHKDSK выделяет новый сектор в наиболее подходящем для файла (которому принадлежат данные) месте жёсткого диска. Затем программа перемещает данные из запасного блока в этот сектор и обновляет информацию о положении файла, что может потребовать новой балансировки дерева блоков размещения. После этого CHKDSK вносит повреждённый сектор в список дефектных блоков, который хранится в дополнительном блоке HPFS, и возвращает освобожденный сектор в список свободных запасных секторов резервного блока. Затем удаляет запись из карты аварийного замещения и записывает отредактированную карту на диск.

Все основные файловые объекты в HPFS, в том числе файловые узлы, блоки размещения и блоки каталогов, имеют уникальные 32-битные идентификаторы и указатели на свои родительские и дочерние блоки. Файловые узлы, кроме того, содержат сокращённое имя своего файла или каталога. Избыточность и взаимосвязь файловых структур HPFS позволяют программе CHKDSK полностью восстанавливать файловую структуру диска, последовательно анализируя все файловые узлы, блоки размещения и блоки каталогов. Руководствуясь собранной информацией, CHKDSK реконструирует файлы и каталоги, а затем заново создает битовые карты свободных секторов диска. Запуск программы CHKDSK следует осуществлять с соответствующими ключами. Так, например, один из вариантов работы этой программы позволяет найти и восстановить удаленные файлы.

HPFS относится к так называемым монтируемым файловым системам. Это означает, что она не встроена в операционную систему, а добавляется к ней при не-

обходимости. Файловая система HPFS устанавливается оператором IFS[1] в файле CONFIG.SYS. Этот оператор всегда помещается в первой строке данного конфигурационного файла. В приводимом далее примере оператор IFS устанавливает файловую систему HPFS с кэшем в 2 Мбайт, длиной записи кэша в 8 Кбайт и автоматической процедурой проверки дисков C и D:

```
IFS=E:\OS2\HPFS.IFS /CACHE:2048 /CRECL:4 /AUTOCHECK:CD
```

Для запуска программы управления процессом кэширования следует прописать в файле CONFIG.SYS ещё одну строку:

```
RUN=E:\OS2\CACHE.EXE /Lazy:On /BufferIdle:2000 /DiskIdle:4000
/MaxAge:8000 /D1rtyMax:256 /ReadAhead:On
```

В этой строке включается режим отложенной («ленивой») записи, устанавливаются параметры работы этого режима, а также включается режим упреждающего чтения данных, что в целом позволяет существенно сократить количество обращений к диску и ощутимо повысить быстродействие файловой системы. Так, ключ Lazy с параметром On включает «ленивую запись», а с параметром Off – выключает. Ключ BufferIdle определяет время в миллисекундах, в течение которого буфер кэша должен оставаться в неактивном состоянии, чтобы стало возможным осуществить запись данных из кэша на диск. По умолчанию (то есть если не прописывать данный ключ явным образом) это время равно 500 мс. Ключ DiskIdle задает время (в миллисекундах), в течение которого диск должен оставаться в неактивном состоянии, чтобы стало возможным осуществить запись данных из кэша на диск. По умолчанию это время равно 1 с. Этот параметр позволяет избежать записи из кэша на диск во время выполнения других операций с диском.

Ключ MaxAge задаёт время (тоже в миллисекундах), по истечении которого часто сохраняемые в кэше данные наконец помечаются как «устаревшие» и при переполнении кэша могут быть замещены новыми. По умолчанию это время равно 5 с.

[1] *IFS* (installable file system) – устанавливаемая, монтируемая система управления файлами.

Остальные подробности установки параметров и возможные значения ключей имеются в HELP-файлах, устанавливаемых вместе с операционной системой OS/2 Warp.

Наконец, следует сказать и ещё об одной системе управления файлами – речь идет о реализации HPFS для работы на серверах, функционирующих под управлением OS/2. Это система управления файлами, получившая название HPFS386.IPS. Ёё принципиальное отличие от системы HPFS.IFS заключается в том, что HPFS386.IFS позволяет (посредством более полного использования технологии расширенных атрибутов) организовать ограничения на доступ к файлам и каталогами с помощью соответствующих списков доступа – ACL (access control list). Эта технология, как известно, используется в файловой системе NTFS. Кроме этого, в системе HPPS386.IFS в отличие от HPFS.IFS нет ограничений на объём памяти, выделяемой для кэширования файловых записей. Иными словами, при наличии достаточного объёма оперативной памяти объём файлового кэша может быть в несколько десятков мегабайт, в то время как для обычной HPFS.IFS этот объём не может превышать 2 Мбайт, что по сегодняшним меркам безусловно мало. Наконец, при установке режимов работы файлового кэша HPFS386.IFS есть возможность явным образом указать алгоритм кэширования. Наиболее эффективным алгоритмом можно считать так называемый «элеваторный», когда при записи данных из кэша на диск они предварительно упорядочиваются таким образом, чтобы минимизировать время, отводимое на позиционирование головок чтения/ записи. Головки чтения/записи при этом перемещаются от внешних цилиндров к внутренним и по ходу своего движения осуществляют запись и чтение данных в соответствии со специальным образом упорядочиваемым списком запросов на дисковые операции.

Приведем пример записи строк в конфигурационном файле CONFIG.SYS, которые устанавливают систему HPFS386.IFS и определяют параметры работы её подсистемы кэширования:

```
IFS=E:\IBM386FS\HPFS386.IFS /AUTOCHECK:EGH
RUN=E:\IBM386FS\CACHE386.EXE /Lazy:On /BufferIdle:4000 /MaxAge:20000
```

Эти записи следует понимать следующим образом. При запуске операционной системы в случае обнаружения флага, означающего, что не все файлы были закрыты в процессе предыдущей работы, система управления файлами HPFS386.IFS сначала запустит программу проверки целостности файловой системы для томов E:, G: и H:. Для кэширования файлов при работе этой системы управления файлами устанавливается режим отложенной записи со временем жизни буферов до 20 с. Остальные параметры, в частности алгоритм обслуживания запросов, устанавливаются в файле HPFS386.INI, который в данном случае располагается в директории E:\IBM386FS.

Опишем кратко некоторые наиболее интересные параметры, управляющие работой кэша в этой системе управления файлами. Прежде всего, отметим, что файл HPFS386.INI разбит на несколько секций. В настоящий момент рассмотрим секцию [ULTIMEDIA]:

```
[ULTIMEDIA]
QUEUESORT={FIFO|ELEVATOR|DEFAULT|CURRENT}
QUEUEMETHOD={PRIORITY|NOPRIORITY|DEFAULT|CURRENT}
QUEUEDEPTH={1...255|DEFAULT|CURRENT}
```

Параметр QUEUESORT задаёт способ ведения очереди запросов к диску. Он может принимать значения FIFO, ELEVATOR, DEFAULT и CURRENT. Если задано значение FIFO, то каждый новый запрос просто добавляется в конец очереди, то есть запросы выполняются в том порядке, в котором они поступают в систему. Однако можно упорядочить некоторое количество запросов по возрастанию номеров дорожек. Если задано значение ELEVATOR, то включается режим поддержки упорядоченной очереди запросов. При этом запросы начинают обрабатываться по алгоритму ELEVATOR (он же C-SCAN или «режим плавающей головки» [24, 28]). Напомним, этот алгоритм подразумевает, что головка чтения/записи сканирует диск в выбранном направлении (например, в направлений возрастания номеров дорожек), останавливаясь для выполнения запросов, находящихся на пути следования. Когда она доходит до последнего запроса, головка чтения/записи переносится на начальную дорожку и процесс обслуживания запросов продолжается.

Если для параметра QUEUESORT задано значение DEFAULT, то выбирается алгоритм по умолчанию. Сейчас это ELEVATOR. Если задано значение CURRENT, то остается в силе тот алгоритм, который был выбран DASD Manager при инициализации.

Параметр QUEUEMETHOD определяет, должны ли учитываться приоритеты запросов при построении очереди. Он может принимать значения PRIORITY, NOPRIORITY, DEFAULT и CURRENT. Если задано значение NOPRIORITY, то все запросы включаются в общую очередь, а их приоритеты игнорируются. Если задано значение PRIORITY, то модуль DASD Manager будет поддерживать несколько очередей запросов, по одной на каждый приоритет. Когда DASD Manager передаёт запросы на исполнение драйверу диска, он сначала выбирает запросы из самой приоритетной очереди, потом из менее приоритетной и т. д. Приоритеты назначает HPFS386, а распределены они следующим образом.

High:

1 Shutdown или экстренная запись из-за сбоя питания.

2 Страничный обмен.

3 Обычные запросы от foreground[1] сессий.

4 Обычные запросы от background[2] сессии. (Приоритеты 3 и 4 равны, если в файле CONFIG.SYS задан параметр RIORITY_DISK_IO=NO.)

5 Read-ahead и низкоприоритетные запросы страничного обмена (страничная предвыборка).

6 Lazy-Write и прочие запросы, не требующие немедленной реакции.

Low:

7 Предвыборка.

Если для параметра QUEUEMETHOD задано значение DEFAULT, то выбирается метод по умолчанию. Сейчас это PRIORITY. Если задано значение CURRENT,

[1] *Foreground session* – сессия «переднего плана», то есть та задача, с которой сейчас работает пользователь, окно этой задачи является активным.

[2] *Background session* – «фоновая сессия», то есть задача, запущенная пользователем, но в настоящий момент не находящаяся непосредственно в работе. Говорят, что эта задача выполняется на фоне текущих активных вычислений.

то остается в силе тот метод, который был выбран DASD Manager при инициализации.

Параметр QUEUEDEPTH задает глубину просмотра очереди при выборке запросов. Он может принимать значения из диапазона (1...255), а также DEFAULT и CURRENT. Если в качестве значения параметра QUEUEDEPTH задано число, то оно определяет количество запросов, которые должны находиться в очереди дискового адаптера одновременно. Например, для SCSI-адаптеров имеет смысл поддерживать такую длину очереди, при которой они смогут загрузить все запросы в свои аппаратные структуры (tagged queue или mailbox). Если очередь запросов к адаптеру будет слишком короткой, то аппаратура будет работать с неполной загрузкой, а если она будет слишком длинной – драйвер SCSI-адаптера будет перегружен «лишними» запросами. Поэтому разумным значением для QUEUEDEPTH будет число, немного превышающее длину аппаратной очереди команд адаптера. Если для параметра QUEUEDEPTH задано значение DEFAULT, то глубина просмотра очереди определяется автоматически на основании значения, которое рекомендовано драйвером дискового адаптера. Если задано значение CURRENT, то глубина просмотра очереди не изменяется. В текущей реализации CURRENT эквивалентно DEFAULT.

Итак, текущие умолчания для HPFS386 имеют вид:

QUEUESORT=FIFO

QUEUEMETHOD=DEFAULT ^

QUEUEDEPTH=2

А текущие умолчания для DASD Manager таковы:

QUEUESORT=ELEVATOR

QUEUEMETHOD=PRIORITY

QUEUEDEPTH=<Зависит от адаптера диска>

Умолчания DASD Manager можно менять с помощью параметра /QF:

BASEDEV=OS2DASD.DMD /QF:{1|2|3)

где 1 - QUEUESORT = FIFO; 2 - QUEUEMETHOD = NOPRIORITY; 3 - QUEUESORT = FIFO и QUEUEMETHOD = NOPRIORITY.

Наконец, добавим ещё несколько слов об устанавливаемых файловых системах (installable file systems – IFS), представляющих собой специальные «драйверы» для доступа к разделам, отформатированным под другую файловую систему. Это очень удобный и мощный механизм добавления в ОС новых файловых систем и замены одной системы управления файлами на другую. Сегодня, например, для OS/2 уже реально существуют IFS-модули для файловой системы VFAT (FAT с поддержкой длинных имен), FAT32, Ext2FS (файловая система Linux), NTFS (правда, пока только для чтения). Для работы с данными на CD-ROM имеется CDFS.IFS. Есть и FTP.IFS, позволяющая монтировать ftp-архивы как локальные диски. Механизм устанавливаемых файловых систем был перенесён и в систему Windows NT.

Файловая система NTFS (New Technology File System)

В название файловой системы NTFS входят слова «New Technology», то есть «новая технология». Действительно, NTFS содержит ряд значительных усовершенствований и изменений, существенно отличающих её от других файловых систем. С точки зрения пользователей, файлы по-прежнему хранятся в каталогах (часто называемых «папками» или *фолдерами*[1] в среде Windows). Однако в NTFS в отличие от FAT работа на дисках большого объёма происходит намного эффективнее; имеются средства для ограничения в доступе к файлам и каталогам, введены механизмы, существенно повышающие надёжность файловой системы, сняты многие ограничения на максимальное количество дисковых секторов и/или кластеров.

Основные возможности файловой системы NTFS

При проектировании системы NTFS особое внимание было уделено следующим характеристикам [53]:

♦ надёжн*ость*. Высокопроизводительные компьютеры и системы совместного пользования (серверы) должны обладать повышенной надёжностью, которая является ключевым элементом структуры и поведения NTFS. Одним из способов уве-

[1] *Folder* – папка. Кстати, термин folder был позаимствован из других операционных систем и не имеет к системе Windows отношения.

личения надёжности является введение механизма транзакций, при котором осуществляется *журналирование*[1] файловых операций;

♦ *расширенная функциональность.* NTFS проектировалась с учётом возможного расширения. В ней были воплощены многие дополнительные возможности – усовершенствованная отказоустойчивость, эмуляция других файловых систем, мощная модель безопасности, параллельная обработка потоков данных и создание файловых атрибутов, определяемых пользователем;

♦ *поддержка POSIX*[2]. Поскольку правительство США требовало, чтобы все закупаемые им системы хотя бы в минимальной степени соответствовали стандарту POSIX, такая возможность была предусмотрена и в NTFS. К числу базовых средств файловой системы POSIX относится необязательное использование имён файлов с учётом регистра, хранение времени последнего обращения к файлу и механизм так называемых «жёстких ссылок» – альтернативных имен, позволяющих ссылаться на один и тот же файл по двум и более именам;

♦ *гибкость.* Модель распределения дискового пространства в NTFS отличается чрезвычайной гибкостью. Размер кластера может изменяться от 512 байт до 64 Кбайт; он представляет собой число, кратное внутреннему кванту распределения дискового пространства. NTFS также поддерживает длинные имена файлов, набор символов Unicode и альтернативные имена формата 8.3 для совместимости с FAT.

NTFS превосходно справляется с обработкой больших массивов данных и достаточно хорошо проявляет себя при работе с томами объёмом от 300-400 Мбайт и выше. Максимально возможные размеры тома (и размеры файла) составляют 16

[1] При *журналировании* файловых операций система управления файлами фиксирует в специальном служебном файле происходящие изменения. В начале операции, связанной с изменением файловой структуры, делается соответствующая пометка. Если во время операций над файлами происходит какой-нибудь сбой, то упомянутая отметка о начале операции остается указанной как незавершенная. При выполнении процедуры проверки целостности файловой системы после перезагрузки машины эти незавершенные операции будут отменены и файлы будут приведены к исходному состоянию. Если же операция изменения данных в файлах завершается нормальным образом, то в этом самом служебном файле поддержки журналирования операция отмечается как завершенная.

[2] *POSIX* (Portable operating system for computing environments). Синоним IEEE Std 1003.1. С 1990 года является международным стандартом на *машинно-независимый (переносимый) интерфейс компьютерной среды.* Этот стандарт разработан американским институтом IEEE в 1988 году. Он представляет собой набор функций, взятых из операционных систем AT&T UNIX System V и Berkeley Standard Distribution UNIX. Основное внимание в требованиях этого стандарта уделяется взаимодействию прикладных программ с операционной системой. Написание прикладных программ в данном стандарте позволяет создавать программы, легко переносимые из одной операционной среды в другую.

Эбайт[1]. Количество файлов в корневом и некорневом каталогах не ограничено. Поскольку в основу структуры каталогов NTFS заложена эффективная структура данных, называемая «бинарным деревом» (см. раздел «Файловая система HPFS»), время поиска файлов в NTFS (в отличие от систем на базе FAT) не связано линейной зависимостью с их количеством.

Система NTFS также обладает определенными средствами самовосстановления. NTFS поддерживает различные механизмы проверки целостности системы, включая ведение журналов транзакций, позволяющих воспроизвести файловые операции записи по специальному системному журналу.

Файловая система NTFS поддерживает объектную модель безопасности NT и рассматривает все тома, каталоги и файлы как самостоятельные объекты. NTFS обеспечивает безопасность на уровне файлов; это означает, что права доступа к томам, каталогам и файлам могут зависеть от учётной записи пользователя и тех групп, к которым он принадлежит. Каждый раз, когда пользователь обращается к объекту файловой системы, его права доступа проверяются по списку разрешений данного объекта. Если пользователь обладает достаточным уровнем прав, его запрос удовлетворяется; в противном случае запрос отклоняется. Эта модель безопасности применяется как при локальной регистрации пользователей на компьютерах с NT, так и при удалённых сетевых запросах.

Наконец, помимо огромных размеров томов и файлов, система NTFS также обладает встроенными средствами сжатия, которые можно применять к отдельным файлам, целым каталогам и даже томам (и впоследствии отменять или назначать их по своему усмотрению).

Структура тома с файловой системой NTFS

Рассмотрим теперь структуру файловой системы NTFS. Наиболее полно она описана в книге [23]. Мы же здесь коснемся только основных моментов.

[1] *Экзабайт* (один экзабайт равен 2^{64}, или приблизительно 16 000 млрд гигабайт).

Одним из основных понятий, используемых при работе с NTFS, является понятие тома (volume)[1]. Возможно также создание отказоустойчивого тома, занимающего несколько разделов, то есть использование RAID-технологии. Как и многие другие системы, NTFS делит всё полезное дисковое пространство тома на кластеры – блоки данных, адресуемые как единицы данных. NTFS поддерживает размеры кластеров от 512 байт до 64 Кбайт; стандартом же считается кластер размером 2 или 4 Кбайт.

Всё дисковое пространство в NTFS делится на две неравные части (рис.4.12). Первые 12 % диска отводятся под так называемую MFT-зону – пространство, которое может занимать, увеличиваясь в размере, главный служебный *метафайл* MFT[2]. Запись каких-либо данных в эту область невозможна. MFT-зона всегда держится пустой – это делается для того, чтобы самый главный, служебный файл (MFT) по возможности не фрагментировался при своем росте. Остальные 88 % тома представляют собой обычное пространство для хранения файлов.

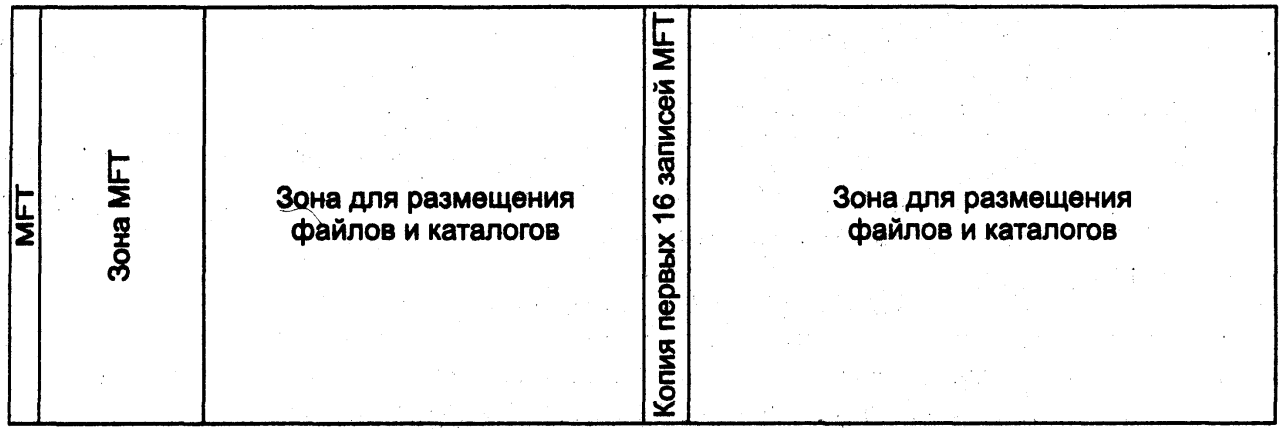

Рис.4.12. Структура тома NTFS

MFT (master file table, общая таблица файлов) представляет собой централизованный каталог всех остальных файлов диска, в том числе и себя самого. MFT по-

[1] Следует, однако, заметить, что понятие тома известно уже около 30 лет. Оно активно используется и в системе OS/2, использующей файловую систему HPFS. См. об этом в разделе «Файловая система HPFS».
[2] MFT (master file table) – это специальный файл, главная системная структура данных, которая и позволяет определять местонахождение всех остальных файлов.

делен на записи фиксированного размера в 1 Кбайт[1], и каждая запись соответствует какому-либо файлу (в общем смысле этого слова). Первые 16 файлов носят служебный характер и недоступны операционной системе – они называются *метафайлами,* причем самый первый метафайл – сам MFT. Эти первые 16 элементов MFT – единственная часть диска, имеющая строго фиксированное положение. Копия этих же 16 записей хранится в середине тома для надёжности, поскольку они очень важны. Остальные части MFT-файла могут располагаться, как и любой другой файл, в произвольных местах диска – восстановить его положение можно с помощью его самого, «зацепившись» за самую основу – за первый элемент MFT.

Таблица 4.7. Метафайлы NTFS

Имя метафайла	Назначение метафайла
$MFT	**Сам Master File Table**
$MFTmirr	**Копия первых 16 записей MFT, размещенная посередине тома**
$LogFile	**Файл поддержки операций журналирования**
$Volume	**Служебная информация – метка тома, версия файловой системы и т. д.**
$AttrDef	**Список стандартных атрибутов файлов на томе**
$.	**Корневой каталог**
$ Bitmap	**Карта свободного места тома**
$Boot	**Загрузочный сектор (если раздел загрузочный)**
$Quota	**Файл, в котором записаны права пользователей на использование дискового пространства (этот файл начал работать лишь в Windows 2000 с системой NTFS 5.0)**
$Upcase	**Файл – таблица соответствия заглавных и прописных букв в именах файлов. В NTFS имена файлов записываются в Unicode (что составляет 65 тысяч различных символов) и искать большие и малые эквиваленты в данном случае – нетривиальная задача**

Упомянутые первые 16 файлов NTFS (метафайлы) носят служебный характер; каждый из них отвечает за какой-либо аспект работы системы. Метафайлы находятся в корневом каталоге NTFS-тома. Все они начинаются с символа имени «$», хотя получить какую-либо информацию о них стандартными средствами сложно.

[1] Размер файловых записей MFT для тома – минимум 1 Кбайт и максимум 4 Кбайт – определяется во время форматирования тома.

В табл. 4.7 приведены основные известные метафайлы и их назначение. Таким образом, можно узнать, например, сколько операционная система тратит на каталогизацию тома, посмотрев размер файла $MFT.

Итак, все файлы тома упоминаются в MFT. В этой структуре хранится вся информация о файлах, за исключением собственно данных. Имя файла, размер, положение на диске отдельных фрагментов и т. д. – всё это хранится в соответствующей записи. Если для информации не хватает одной записи MFT, то используется несколько записей, причем не обязательно идущих подряд. Файлы могут иметь не очень большой размер. Тогда применяется довольно удачное решение: данные файла хранятся прямо в MFT, в оставшемся от основных данных месте в пределах одной записи MFT. Файлы, занимающие сотни байт, обычно не имеют своего «физического» воплощения в основной файловой области – все данные такого файла хранятся в одном месте, в MFT.

Файл в томе с NTFS идентифицируется так называемой файловой ссылкой (File Reference), которая представляется как 64-разрядное число. Файловая ссылка состоит из номера файла, который соответствует позиции его файловой записи в MFT, и номера последовательности. Последний увеличивается всякий раз, когда данная позиция в MFT используется повторно, что позволяет файловой системе NTFS выполнять внутренние проверки целостности.

Каждый файл в NTFS представлен с помощью *потоков* (streams), то есть у него нет как таковых «просто данных», а есть «потоки». Для правильного понимания потока достаточно указать, что один из потоков и носит привычный нам смысл – данные файла. Но большинство атрибутов файла – это тоже потоки. Таким образом, получается, что базовая сущность у файла только одна – номер в MFT, а всё остальное, включая и его потоки, – опционально. Данный подход может эффективно использоваться – например, файлу можно «прилепить» ещё один поток, записав в него любые данные. В Windows 2000 таким образом записана информация об авторе и содержании файла (одна из закладок в свойствах файла, просматриваемых, например, из проводника). Интересно, что эти дополнительные потоки не видны стандартными средствами работы с файлами: наблюдаемый размер файла – это

лишь размер основного потока, который содержит традиционные данные. Можно, к примеру, иметь файл нулевой длины, при стирании которого освободится 1 Гбайт свободного места – просто потому, что какая-нибудь хитрая программа или технология «прилепила» к нему дополнительный поток (альтернативные данные) такого большого размера. Но на самом деле в настоящее время потоки практически не используются, так что опасаться подобных ситуаций не следует, хотя гипотетически они возможны[1]. Просто необходимо иметь в виду, что файл в NTFS – это более глубокое понятие, чем можно себе представить, просматривая каталоги диска.

Стандартные же атрибуты для файлов и каталогов в томе NTFS имеют фиксированные имена и коды типа, они перечислены в табл. 4.8.

Таблица 4.8. Атрибуты файлов в системе NTFS

Системный атрибут	Описание атрибута
Стандартная информация о файле	Традиционные атрибуты Read Only, Hidden, Archive, System, отметки времени, включая время создания или последней модификации, число каталогов, ссылающихся на файл
Список атрибутов	Список атрибутов, из которых состоит файл, и файловая ссылка на файловую запись и MFT, в которой расположен каждый из атрибутов. Последний используется, если файлу необходимо более одной записи в MFT
Имя файла	Имя файла в символах Unicode. Файл может иметь несколько атрибутов – имён файла, подобно тому как это имеет место в UNIX-Системах. Это случается, когда имеется связь POSIX с данным файлом или если у файла есть автоматически сгенерированное имя в формате 8.3
Дескриптор защиты	Структура данных защиты (ACL), предохраняющая файл от несанкционированного доступа. Атрибут «дескриптор защиты» определяет, кто владелец файла и кто имеет доступ к нему
Данные	Собственно данные файла, его содержимое. В NTFS у файла по умолчанию есть один безымянный атрибут данных, и он может иметь дополнительные именованные атрибуты данных. У каталога нет атрибута данных по умолчанию, но он может иметь необязательные именованные атрибуты данных
Корень индекса, размещение индекса, битовая карта (только для каталогов)	Атрибуты, используемые для индексов имён файлов в больших каталогах
Расширенные атрибуты HPFS	Атрибуты, используемые для реализации расширенных атрибутов HPFS для подсистемы OS/2 и OS/2–клиентов файл-серверов Windows NT

Атрибуты файла в записях MFT расположены в порядке возрастания числовых значений кодов типа, причем некоторые типы атрибутов могут встречаться в записи более одного раза: например, если у файла есть несколько атрибутов дан-

[1] Так, на идее подмены потока основан один из новейших вирусов, который был не так давно создан для распро-

ных или несколько имен. Обязательными для каждого файла в томе NTFS являются атрибут стандартной информации, атрибут имени файла, атрибут дескриптора защиты и атрибут данных. Остальные атрибуты могут встречаться при необходимости.

Имя файла в NTFS, в отличие от файловых систем FAT и HPFS, может содержать любые символы, включая полный набор национальных алфавитов, так как данное представлены в Unicode – 16-битном представлении, которое дает 65 535 разных символов. Максимальная длина имени файла в NTFS – 255 символов.

Большой вклад в эффективность файловой системы вносит организация каталога. Каталог в NTFS представляет собой специальный файл, хранящий ссылки на другие файлы и каталоги, создавая иерархическое строение данных на диске. Файл каталога поделён на блоки, каждый из которых содержит имя файла, базовые атрибуты и ссылку на элемент MFT, который уже предоставляет полную информацию об элементе каталога. Главный каталог диска – корневой – ничем не отличается от обычных каталогов, кроме специальной ссылки на него из начала метафайла MFT.

Внутренняя структура каталога представляет собой бинарное дерево, подобно тому, как это организовано в HPFS. Кстати, при создании файловой системы NTFS разработчики решили использовать максимально возможное количество эффективных решений из HPFS. К сожалению, не было взято на вооружение разбиение всего дискового пространства на зоны, в каждой из которых хранилась бы информация об имеющихся свободных кластерах. В результате отказа от этого подхода и введения механизма транзакций скорость работы файловой системы NTFS существенно ниже скорости работы системы HPFS.

Итак, как нам теперь известно, бинарное дерево каталога располагает имена файлов таким образом, чтобы поиск файла осуществлялся с помощью получения двухзначных ответов на вопросы о положении файла. Бинарное дерево способно дать ответ на вопрос: в какой группе, относительно данного элемента, находится искомое имя – выше или ниже? Мы начинаем с такого вопроса к среднему элементу, и каждый ответ сужает зону поиска в среднем в два раза. Если представить,

---

странения в среде Windows 2000.

что файлы отсортированы по алфавиту, то ответ на вопрос осуществляется очевидным способом – сравнением начальных букв. Область поиска, суженная в два раза, начинает исследоваться аналогичным образом, начиная опять же со среднего элемента.

Заметим, что добавлять файл в каталог в виде дерева не намного труднее, чем в линейный каталог системы FAT. Это сопоставимые по времени операции. Для того чтобы добавить новый файл в каталог, нужно сначала убедиться, что файла с таким именем там ещё нет. Поэтому в системе FAT с линейной организацией записей каталога у нас появляются трудности не только с поиском файла. И это с лихвой компенсирует саму простоту добавления файла в каталог.

Возможности файловой системы NTFS по ограничению доступа к файлам и каталогам

Рассмотрим основные возможности, связанные как с организацией различных прав доступа к файлам и каталогам при использовании сетевого доступа, так и локальные ограничения на файлы и каталоги. NTFS рассматривает каталоги (папки) и файлы как разнотипные объекты и ведёт отдельные (хотя и перекрывающиеся) списки прав доступа для каждого типа [36]. Ниже перечислены права NTFS, назначаемые папкам (соответствующие права для файлов приведены ниже):

- ♦ нет доступа (no access) (None)(нет);
- ♦ полный доступ (full control) (All)(All) (все)(все);
- ♦ право чтения (read) (RX)(RX) (чтение)(чтение);
- ♦ право добавления (add) (WX)(not specified) (запись/выполнение не указано);
- ♦ право добавления и чтения (add&read) (RWX)(RX) (чтение/запись/выполнение) (чтение/выполнение);
- ♦ право просмотра (list) (RX)(not specified) (чтение/выполнение)(не указано);
- ♦ право изменения (change) (RWXD)(RWXD) (чтение/запись/ выполнение/ удаление) (чтение/запись/выполнение/удаление).

Обратите внимание на два выражения в скобках, указанные после имени права доступа. Первое выражение относится к самой папке, а второе – ко всем файлам,

которые могут быть созданы внутри неё. Например, при полном доступе для папки разрешаются любые действия, однако пользователь с полным доступом к папке также будет обладать полным правом доступа ко всем созданным в ней файлам (если только права доступа к файлу не были изменены его владельцем или администратором). Другими словами, в NTFS файлы и папки по умолчанию наследуют права доступа, установленные для их родительской папки, однако эти права могут быть изменены любым пользователем, которому разрешено изменять права доступа для соответствующих объектов NTFS.

Файлы в NTFS могут обладать следующими правами:

- ♦ полный доступ (full control) (All) (все);
- ♦ нет доступа (no access) (None) (нет);
- ♦ право изменения (change) (RWXD) (чтение/запись/выполнение/удаление);
- ♦ право чтения (read) (RX) (чтение/выполнение).

Для прав доступа NTFS, как и для прав общих каталогов, действует принцип поглощения. Исключение составляет право «нет доступа», отменяющее действие всех остальных прав.

При сетевом подключении пользователей права NTFS могут вступить в конфликт с правами общих каталогов. В такой ситуации применяется право доступа с наиболее жесткими ограничениями. У многих возникают проблемы с пониманием получаемых при сетевом доступе ограничений. Однако здесь можно легко разобраться, если помнить, что при доступе по сети к каталогам и файлам, располагающихся на томах с NTFS, у нас получаются задействованными два последовательных механизма. Сначала мы получаем доступ к файлам, который был определён сетевыми механизмами. Это право «нет доступа» – «по access», право на «чтение» – «read», право «изменение» – «change» и «полный доступ» – «full control». После этого вступают в силу ограничения на файлы и каталоги, определённые свойствами NTFS. То есть нам нужно преодолеть последовательно два препятствия. Другими словами, итоговые права на папки и файлы будут определяться максимальными ограничениями, которые были заданы в каждом из механизмов.

Помимо перечисленных прав имеется ещё так называемый *специальный доступ* (Special Access). Если выбрать это право доступа, то на самом деле появляется возможность выбирать несколько прав одновременно из следующего перечня:

- ♦ полный доступ (full control) (All);
- ♦ чтение (read) (R);
- ♦ запись (write) (W);
- ♦ выполнение (execute) (X);
- ♦ удаление (delete) (D);
- ♦ изменение разрешений (change permissions) (P);
- ♦ изменение владельца (take ownership) (O).

В принципе можно было бы выбирать любые совокупности перечисленных разрешений, однако на практике это, увы, не работает. Например, нельзя указать право X (исполнение) без права R (чтение), хотя в других системах управления файлами такое право обеспечивается. Оно позволяет выполнять программу, файл которой помечен таким атрибутом, но не дает возможности её скопировать. Многие другие комбинации специальных разрешений тоже не работают должным образом и это надо обязательно иметь в виду. Лучше пользоваться штатными правами на файлы и каталоги, которые были перечислены выше.

Рассмотрим теперь, что происходит с правами на защищённые файлы в NTFS при их перемещении. Папки более высокого уровня в NTFS обычно обладают теми же правами, что и находящиеся в них файлы и папки. Например, если вы создаете папку внутри другой папки, для которой администраторы обладают правом полного доступа, а операторы архива – правом чтения, то новая папка унаследует эти права. То же относится и к файлам, копируемым из другой папки или перемещаемым из другого раздела NTFS.

Если папка или файл перемещается в другую папку того же раздела NTFS, то атрибуты безопасности не наследуются от нового объекта-контейнера. Например, если из папки с правами чтения для группы everyone файл перемещается в папку того же раздела с полным доступом для той же группы, то для перемещенного файла будет сохранено исходное право чтения. Дело в том, что при перемещении

файлов в границах одного раздела NTFS изменяется только указатель местонахождения объекта, а все остальные атрибуты (включая атрибуты безопасности) остаются без изменений.

Три следующих важных правила помогут определить состояние прав доступа при перемещении или копировании объектов NTFS:

♦ При перемещении файлов в границах раздела NTFS сохраняются исходные права доступа.

♦ При выполнении других операций (создании или копировании файлов, а также их перемещении между разделами NTFS) наследуются права доступа родительской папки.

♦ При перемещении файлов из раздела NTFS в раздел FAT все права NTFS теряются.

Основные отличия FAT и NTFS

Если говорить о накладных расходах на хранение служебной информации, FAT отличается от NTFS большей компактностью и меньшей сложностью. В большинстве томов FAT на хранение таблицы размещения, содержащей информацию обо всех файлах тома, расходуется менее 1 Мбайт. Столь низкие накладные расходы позволяют форматировать в FAT жесткие диски малого объёма и флоппи–диски. В NTFS служебные данные занимают больше места, чем в FAT. Так, каждый элемент каталога занимает 2 Кбайт. Однако это имеет и свои преимущества, так как содержимое файлов объёмом 1500 байт и менее может полностью храниться в элементе каталога.

Система NTFS не может использоваться для форматирования флоппи-дисков. Не стоит пользоваться ею для форматирования разделов объёмом менее 50-100 Мбайт. Относительно высокие накладные расходы приводят к тому, что для малых разделов служебные данные могут занимать до 25 % объёма носителя. Корпорация Microsoft рекомендует использовать FAT для разделов объёмом 256 Мбайт и менее, а NTFS – для разделов объёмом 400 Мбайт и более[1].

[1] Следут заметить, что этим рекомендациям уже более 6 лет. Сейчас, как известно, объёмы дисковых накопителей уже давно перешли рубеж в десяток гигабайт, поэтому упоминание логического диска объёмом в 400 Мбайт представляется неактуальным.

Следующий критерий сравнения – размер файлов. Разделы FAT имеют объём до 2 Гбайт, VFAT – до 4 Гбайт и FAT32 – до 4 Гбайт.

Тем не менее, из-за особенностей своего внутреннего строения разделы FAT лучше всего работают для разделов объёмом 200 Мбайт и менее. Разделы NTFS могут достигать 16 Эбайт, однако в настоящее время из-за аппаратных и других системных причин размер файлов ограничивается 2 Тбайт.

Разделы FAT могут использоваться практически во всех операционных системах. За редкими исключениями, с разделами NTFS можно работать напрямую только из Windows NT, хотя и имеются для ряда ОС соответствующие реализации систем управления файлами для чтения файлов из томов NTFS. Так, например, утилита (драйвер) NTFSDOS позволяет читать данные NTFS на компьютере, загруженном в режиме MS-DOS. Однако полноценных реализаций для работы с NTFS вне системы Windows NT пока нет.

Разделы FAT не обеспечивают локальной безопасности. С другой стороны, разделы NTFS обеспечивают локальную безопасность как файлов, так и каталогов. Для разделов FAT могут устанавливаться общие права, связанные с общим доступом к каталогам в сети. Однако такая защита не помещает пользователю с локальным входом получить доступ к файлам своего компьютера. В отношении безопасности NTFS оказывается предпочтительным вариантом. Разделы NTFS могут запрещать или ограничивать доступ как удаленных, так и локальных пользователей. Следовательно, к защищенным файлам смогут обратиться лишь те пользователи, которым были предоставлены соответствующие права.

Напомним, что Windows NT содержит специальную утилиту CONVERT.EXE, которая преобразует тома FAT в эквивалентные тома NTFS, однако для обратного преобразования (из NTFS в FAT) подобных утилит не существует. Чтобы выполнить такое обратное преобразование, вам придется создать раздел FAT, скопировать в него файлы из раздела NTFS и затем удалить оригиналы. Важно при этом не забывать и о том, что при копировании файлов из NTFS в FAT теряются все атрибуты безопасности NTFS (напомним, что в FAT не предусмотрены средства для определения и последующего хранения этих атрибутов).

В последнее время появилось ещё одно очень важное обстоятельство, связанное с тем, что объёмы дисковых механизмов намного превысили максимально допустимый размер, приемлемый для FAT, – 8,4 Гбайт. Этот предел объясняется максимально возможными значениями в адресе сектора, для которого, как мы уже знаем, отводится всего 3 байта. Поэтому в подавляющем большинстве случаев при работе в среде Windows-систем используют либо FAT32, либо NTFS. Последняя, безусловно, лучше, но она не поддерживается в широко распространённых ОС Windows 98 и ныне всё более часто встречающейся Windows Millennium Edition.

Контрольные вопросы и задачи

Вопросы для проверки

1 Почему создание подсистемы ввода/вывода считается одной из самых сложных областей проектирования операционных систем?

2 Почему операции ввода/вывода в ОС объявляются привилегированными?

3 Перечислите основные задачи, возлагаемые на супервизор ввода/вывода.

4 В каких случаях устройство ввода/вывода называется инициативным?

5 Какие режимы управления вводом/выводом вы знаете? Опишите каждый из них.

6 Что означает термин «spooling» и что означает термин «swapping»?

7 Чем обеспечивается независимость пользовательских программ от устройств ввода/вывода, подключенных к компьютеру?

8 Что такое синхронный и асинхронный ввод/вывод?

9 Расскажите о кэшировании операций ввода/вывода при работе с накопителями на магнитных дисках.

10 Что такое «файловая система»? Что обеспечивает использование той или иной файловой системы? Какие файловые системы, используемые в ОС и ПК, вы знаете?

11 Опишите структуру магнитного диска (разбиение дисков на разделы). Сколько (и каких) разделов может быть на магнитном диске?

12 Как в общем случае осуществляется загрузка ОС после включения компьютера? Что такое системный и внесистемный загрузчики? Где они располагаются?

13 Объясните общие принципы файловой системы FAT. Что такое кластер, от чего зависит его размер?

14 Сравните файловые системы FAT16 и FAT32. В чём заключаются их достоинства и недостатки?

15 За счёт чего в файловой системе HPFS обеспечена высокая производительность?

16 Что означает «журналирование» файловых операций? Что это даёт?

17 Расскажите о правилах, которые определяют состояние прав доступа при перемещении или копировании объектов, если используется NTFS.

18 Проведите сравнительный анализ файловых систем HPFS и NTFS; перечислите достоинства и недостатки каждой.

Задания

1 Используя ПК с установленной на нем ОС Windows NT 4.0, исследуйте ограничения доступа к файлам и каталогам, которыми обладает файловая система NTFS. Расскажите о результатах.

2 Используя ПК с установленной на нем ОС Windows NT 4.0, проверьте правила, которые определяют состояние прав доступа при перемещении или копировании объектов при использовании NTFS. Расскажите о результатах.

3 Используя специально выделенный для этих целей ПК, изучите структуру диска и освойте работу с программой Disk Editor. Выполните следующее задание:

♦ Включите компьютер. Во время выполнения программы самотестирования войдите в BIOS и установите возможность загрузки с дискеты.

♦ Загрузите операционную систему, расположенную на магнитном диске. Покажите преподавателю, что она работает.

♦ Загрузитесь с системной дискеты (на ней должна быть MS-DOS 6.2 или какая-нибудь другая версия DOS).

♦ Запустите программу Disk Editor из комплекта утилит от Питера Нортона. С помощью встроенных подсказок изучите основные возможности этой утилиты.

♦ Посмотрите структуру диска. Сохраните MBR и загрузочный сектор на свою дискету.

♦ Найдите таблицы размещения файлов для двух разделов магнитного диска. Сохраните их на дискете. Выйдите из программы Disk Editor.

♦ Запустите программу FDISK. Посмотрите структуру диска – сколько и каких разделов на нём расположено?

♦ Удалите все логические диски с помощью программы FDISK.

♦ Перезапустите компьютер и убедитесь, что операционная система, расположенная раньше на магнитном диске, больше не функционирует. Покажите преподавателю.

♦ Восстановите операционную систему и файлы, расположенные на магнитном диске, используя программу Disk Editor и файлы, которые вы ранее создали с её помощью. Покажите преподавателю.

ГЛАВА 5 Архитектура операционных систем и интерфейсы прикладного программирования

Несмотря на тот факт, что в наши дни уже практически никто не разрабатывает операционные системы (естественно, за исключением нескольких известных компаний, специализирующихся на этом направлении, кстати, одном из сложнейших) и все являются пользователями наиболее распространенных систем, мы всё-таки рассмотрим кратко вопросы архитектуры ОС. Сделать это необходимо потому, что многие возможности и характеристики ОС определяются в значительной мере её архитектурой.

Основные принципы построения операционных систем

Среди множества принципов, которые используются при построении ОС, перечислим несколько наиболее важных (на наш взгляд, так как в соответствующих публикациях на эту тему перечисляется существенно большее их количество).

Принцип модульности

Под *модулем* в общем случае понимают функционально законченный элемент системы, выполненный в соответствии с принятыми межмодульными интерфейсами. По своему определению модуль предполагает возможность без труда заменить его на другой при наличии заданных интерфейсов. Способы обособления составных частей ОС в отдельные модули могут существенно различаться, но чаще всего разделение происходит именно по функциональному признаку. В значительной степени разделение системы на модули определяется используемые методом проектирования ОС (снизу вверх или наоборот).

Особо важное значение при построении ОС имеют *привилегированные, повторно входимые* и *реентерабельные* модули, так как они позволяют более эффективно использовать ресурсы вычислительной системы. Как мы уже знаем (см. раздел «Основные виды ресурсов», глава 1), достижение реентерабельности реализуется различными способами. В некоторых системах реентерабельность программа получают автоматически, благодаря неизменяемости кодовых частей программ при исполнении (из-за особенностей системы команд машины), а также автоматическому распределению регистров, автоматическому отделению кодовых частей программ от данных и помещению последних в системную область памяти Естественно, что для этого необходима соответствующая аппаратная поддержка В других случаях это достигается программистами за счёт использования специальных системных модулей.

Принцип модульности отражает технологические и эксплуатационные свойства системы. Наибольший эффект от его использования достижим в случае, когда

принцип распространён одновременно на операционную систему, прикладные программы и аппаратуру.

Принцип функциональной избирательности

В ОС выделяется некоторая часть важных модулей, которые должны постоянно находиться в оперативной памяти для более эффективной организации вычислительного процесса. Эту часть в ОС называют ядром, так как это действительно основа системы. При формировании состава ядра требуется учитывать два противоречивых требования. В состав ядра должны войти наиболее часто используемые системные модули. Количество модулей должно быть таковым, чтобы объём памяти, занимаемый ядром, был бы не слишком большим. В состав ядра, как правило, входят модули по управлению системой прерываний, средства по переводу программ из состояния счёта в состояние ожидания, готовности и обратно, средства по распределению таких основных ресурсов, как оперативная память и процессор.

Помимо программных модулей, входящих в состав ядра и постоянно располагающихся в оперативной памяти, может быть много других системных программных модулей, которые получают название *транзитных.* Транзитные программные модули загружаются в оперативную память только при необходимости и в случае отсутствия свободного пространства могут быть замещены другими транзитными модулями. В качестве синонима к термину «транзитный» можно использовать термин «диск-резидентный».

Принцип генерируемости ОС

Основное положение этого принципа определяет такой способ исходного представления центральной системной управляющей программы ОС (её ядра и основных компонентов, которые должны постоянно находиться в оперативной памяти), который позволял бы настраивать эту системную супервизорную часть, исходя из конкретней конфигурации конкретного вычислительного комплекса и круга решаемых задач. Эта процедура проводится редко, перед достаточно протяженным периодом эксплуатации ОС. Процесс генерации осуществляется с помощью специ-

альной программы-генератора и соответствующего входного языка для этой программы, позволяющего описывать программные возможности системы и конфигурацию машины. В результате генерации получается полная версия ОС. Сгенерированная версия ОС представляет собой совокупность системных наборов модулей и данных.

Упомянутый раньше принцип модульности положительно проявляется при генерации ОС. Он существенно упрощает настройку ОС на требуемую конфигурацию вычислительной системы. В наши дни при использовании персональных компьютеров с принципом генерируемости ОС можно столкнуться разве что только при работе с Linux. В этой UNIX-системе имеется возможность не только использовать какое-либо готовое ядро ОС, но и самому сгенерировать (скомпилировать) такое ядро, которое будет оптимальным для данного конкретного персонального компьютера и решаемых на нем задач. Кроме генерации ядра в Linux имеется возможность указать и набор подгружаемых драйверов и служб, то есть часть функций может реализовываться модулями, непосредственно входящими в ядро системы, а часть – модулями, имеющими статус подгружаемых, транзитных.

В остальных современных распространенных ОС для персональных компьютеров конфигурирование ОС под соответствующий состав оборудования осуществляется на этапе инсталляции, а потом состав драйверов и изменение некоторых параметров ОС может быть осуществлено посредством редактирования конфигурационного файла.

Принцип функциональной избыточности

Этот принцип учитывает возможность проведения одной и той же работы различными средствами. В состав ОС может входить несколько типов мониторов (модулей супервизора, управляющих тем или другим видом ресурса), различные средства организации коммуникаций между вычислительными процессами. Наличие нескольких типов мониторов, нескольких систем управления файлами позволяет пользователям быстро и наиболее адекватно адаптировать ОС к определенной конфигурации вычислительной системы, обеспечить максимально эффективную

загрузку технических средств при решении конкретного класса задач, получить максимальную производительность при решении заданного класса задач.

Принцип виртуализации

Построение виртуальных ресурсов, их распределение и использование теперь используется практически в любой ОС. Этот принцип позволяет представить структуру системы в виде определенного набора планировщиков процессов и распределителей ресурсов (мониторов) и использовать единую централизованную схему распределения ресурсов. Наиболее естественным и законченным проявлением концепции виртуальности является понятие *виртуальной* машины. По сути, любая операционная система, являясь средством распределения ресурсов и организуя по определенным правилам управление процессами, скрывает от пользователя и его приложений реальные аппаратные и иные ресурсы, заменяя их некоторой абстракцией. В результате пользователи видят и используют виртуальную машину как некое устройство, способное воспринимать их программы, написанные на определённом языке программирования, выполнять их и выдавать результаты. При таком языковом представлении пользователя совершенно не интересует реальная конфигурация вычислительной системы, способы эффективного использования её компонентов и подсистем. Он мыслит и работает с машиной в терминах используемого им языка и тех ресурсов, которые ему предоставляются в рамках виртуальной машины.

Чаще виртуальная машина, предоставляемая пользователю, воспроизводит архитектуру реальной машины, но архитектурные элементы в таком представлении выступают с новыми или улучшенными характеристиками, часто упрощающими работу с системой. Характеристики могут быть произвольными, но чаще всего пользователи желают иметь собственную «идеальную» по архитектурным характеристикам машину в следующем составе:

♦ единообразная по логике работы память (виртуальная) практически неограниченного объёма. Среднее время доступа соизмеримо со значением этого параметра оперативной памяти. Организация работы с информацией в такой памяти

производится в терминах обработки данных – в терминах работы с сегментами данных на уровне выбранного пользователем языка программирования;

♦ произвольное количество процессоров (виртуальных), способных работать параллельно и взаимодействовать во время работы. Способы управления процессорами, в том числе синхронизация и информационные взаимодействия, реализованы и доступны пользователям на уровне используемого языка в терминах управления процессами;

♦ произвольное количество внешних устройств (виртуальных), способных работать с памятью виртуальной машины параллельно или последовательно, асинхронно или синхронно по отношению к работе того или иного виртуального процессора, которые инициируют работу этих устройств. Информация, передаваемая или хранимая на виртуальных устройствах, не ограничена допустимыми размерами. Доступ к такой информации осуществляется на основе либо последовательного, либо прямого способа доступа в терминах соответствующей системы управления файлами. Предусмотрено расширение информационных структур данных, хранимых на виртуальных устройствах.

Степень приближения к «идеальной» виртуальной машине может быть большей или меньшей в каждом конкретном случае. Чем больше виртуальная машина, реализуемая средствами ОС на базе конкретной аппаратуры, приближена к «идеальной» по характеристикам машине и, следовательно, чем больше ее архитектурно-логические характеристики отличны от реально существующих, тем больше степень виртуальности у полученной пользователем машины. Одним из аспектов виртуализации является организация возможности выполнения в данной ОС приложений, которые разрабатывались для других ОС. Другими словами, речь идет об организации нескольким операционных сред, о чем мы уже говорили (см. главу 1). Реализация этого принципа позволяет такой ОС иметь очень сильное преимущество перед аналогичными ОС, не имеющими такой возможности. Примером реализации принципа виртуализации может служить *VDM-машина* (virtual DOS machine) – защищённая подсистема, предоставляющая полную среду MS-DOS и консоль для выполнения MS-DOS приложений. Одновременно может выполняться практически

произвольное число VDM-сессий. Такие VDM-машины имеются и в системах Microsoft Windows, и в OS/2.

Принцип независимости программ от внешних устройств

Этот принцип реализуется сейчас в подавляющем большинстве ОС общего применения. Мы уже говорили о нём, рассматривая принципы организации ввода/вывода. Пожалуй, впервые наиболее последовательно данный принцип был реализован в ОС UNIX. Реализован он и в большинстве современных ОС для ПК. Напомним, этот принцип заключается в том, что связь программ с конкретными устройствами производится не на уровне трансляции программы, а в период планирования её исполнения. В результате перекомпиляция при работе программы с новым устройством, на котором располагаются данные, не требуется.

Принцип позволяет одинаково осуществлять операции управления внешними устройствами независимо от их конкретных физических характеристик. Например, программе, содержащей операции обработки последовательного набора данных, безразлично, на каком носителе эти данные будут располагаться. Смена носителя и данных, размещаемых на них (при неизменности структурных характеристик данных), не принесёт каких-либо изменений в программу, если в системе реализован принцип независимости.

Принцип совместимости

Одним из аспектов совместимости является способность ОС выполнять программы, написанные для других ОС или для более ранних версий данной операционной системы, а также для другой аппаратной платформы.

Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить её на выполнение на другой ОС. Для этого необходимы: совместимость на уровне команд процессора, совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость на уровне исходных текстов требует наличия соответствующего транслятора в составе системного программного обеспечения, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль. Гораздо сложнее достичь двоичной совместимости между процессорами, основанными на разных архитектурах. Для того чтобы один компьютер выполнял программы другого (например, программу для ПК типа IBM PC желательно выполнить на ПК типа Macintosh фирмы Apple), этот компьютер должен работать с машинными командами, которые ему изначально непонятны. В таком случае процессор типа 680x0 (или PowerPC) на Mac должен исполнять двоичный код, предназначенный для процессора i80x86. Процессор 80x86 имеет свои собственные дешифратор команд, регистры и внутреннюю архитектуру. Процессор 680x0 не понимает двоичный код 80x86, поэтому он должен выбрать каждую команду, декодировать её, чтобы определить, для чего она предназначена, а затем выполнить эквивалентную подпрограмму, написанную для 680x0. Так как к тому же у 680x0 нет в точности таких же регистров, флагов и внутреннего арифметико-логического устройства, как в 80x86, он должен имитировать все эти элементы с использованием своих регистров или памяти. И он должен тщательно воспроизводить результаты каждой команды, что требует специально написанных подпрограмм для 680x0, гарантирующих, что состояние эмулируемых регистров и флагов после выполнения каждой команды будет в точности таким же, как и на реальном 80x86. Выходом в таких случаях является использование так называемых прикладных сред или эмуляторов. Учитывая, что основную часть программы, как правило, составляют вызовы библиотечных функций, прикладная среда имитирует библиотечные функции целиком, используя заранее написанную библиотеку функций аналогичного назначения, а остальные команды эмулирует каждую по отдельности.

Одним из средств обеспечения совместимости программных и пользовательских интерфейсов является соответствие стандартам POSIX. Использование стандарта POSIX позволяет создавать программы в стиле UNIX, которые впоследствии могут легко переноситься из одной системы в другую.

Принцип открытой и наращиваемой ОС

Открытая ОС доступна для анализа как пользователям, так и системным специалистам, обслуживающим вычислительную систему. Наращиваемая (модифицируемая, развиваемая) ОС позволяет не только использовать возможности генерации, но и вводить в её состав новые модули, совершенствовать существующие и т.д. Другими словами, необходимо, чтобы можно было легко внести дополнения и изменения, если это потребуется, и не нарушить целостность системы. Прекрасные возможности для расширения предоставляет подход к структурированию ОС по типу клиент–сервер с использованием микроядерной технологии. В соответствии с этим подходом ОС строится как совокупность привилегированной управляющей программы и набора непривилегированных услуг – «серверов». Основная часть ОС остается неизменной и в то же время могут быть добавлены новые серверы или улучшены старые.

Этот принцип иногда трактуют как расширяемость системы.

К открытым ОС, прежде всего, следует отнести UNIX-системы и, естественно, ОС Linux.

Принцип мобильности (переносимости)

Операционная система относительно легко должна переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которая включает наряду с типом процессора и способ организации всей аппаратуры компьютера, иначе говоря, архитектуру вычислительной системы) одного типа на аппаратную платформу другого типа. Заметим, что принцип переносимости очень близок принципу совместимости, хотя это и не одно и то же.

Написание переносимой ОС аналогично написанию любого переносимого кода – нужно следовать некоторым правилам. Во-первых, большая часть ОС должна быть написана на языке, который имеется на всех системах, на которые планируется в дальнейшем её переносить. Это, прежде всего, означает, что ОС должна быть написана на языке высокого уровня, предпочтительно стандартизованном, например, на языке С. Программа, написанная на ассемблере, не является в общем случае переносимой. Во-вторых, важно минимизировать или, если возможно, исклю-

чить те части кода, которые непосредственно взаимодействуют с аппаратными средствами. Зависимость от аппаратуры может иметь много форм. Некоторые очевидные формы зависимости включают прямое манипулирование регистрами и другими аппаратными средствами. Наконец, если аппаратно-зависимый код не может быть полностью исключен, то он должен быть изолирован в нескольких хорошо локализуемых модулях. Аппаратно-зависимый код не должен быть распределен по всей системе. Например, можно спрятать аппаратно-зависимую структуру в программно задаваемые данные абстрактного типа. Другие модули системы будут работать с этими данными, а не с аппаратурой, используя набор некоторых функций. Когда ОС переносится, то изменяются только эти данные и функции, которые ими манипулируют.

Введение стандартов POSIX преследовало цель обеспечить переносимость создаваемого программного обеспечения.

Принцип обеспечения безопасности вычислений

Обеспечение безопасности при выполнении вычислений является желательным свойством для любой многопользовательской системы. Правила безопасности определяют такие свойства, как защита ресурсов одного пользователя от других и установление квот по ресурсам для предотвращения захвата одним пользователем всех системных ресурсов (таких, как память).

Обеспечение защиты информации от несанкционированного доступа является обязательной функцией сетевых операционных систем. Во многих современных ОС гарантируется степень безопасности данных, соответствующая уровню С2 в системе стандартов США. Основы стандартов в области безопасности были заложены в документе «Критерии оценки надёжных компьютерных систем». Этот документ, изданный Национальным центром компьютерной безопасности (NCSC – National Computer Security Center) в США в 1983 году, часто называют Оранжевой книгой.

В соответствии с требованиями Оранжевой книги безопасной считается система, которая «посредством специальных механизмов защиты контролирует доступ

к информации таким образом, что только имеющие соответствующие полномочия лица или процессы, выполняющиеся от их имени, могут получить доступ на чтение, запись, создание или удаление информации».

Иерархия уровней безопасности, приведенная в Оранжевой книге, помечает низший уровень безопасности как D, а высший – как A.

В класс D попадают системы, оценка которых выявила их несоответствие требованиям всех других классов.

Основными свойствами, характерными для систем класса C, являются наличие подсистемы учёта событий, связанных с безопасностью, и избирательный контроль доступа. Класс (уровень) C делится на 2 подуровня: уровень C1, обеспечивающий защиту данных от ошибок пользователей, но не от действий злоумышленников; и более строгий уровень C2. На уровне C2 должны присутствовать:

♦ средства секретного входа, обеспечивающие идентификацию пользователей путём ввода уникального имени и пароля перед тем, как им будет разрешен доступ к системе;

♦ избирательный контроль доступа, позволяющий владельцу ресурса определить, кто имеет доступ к ресурсу и что он может с ним делать. Владелец делает это путём предоставляемых прав доступа пользователю или группе пользователей;

♦ средства учёта и наблюдения (auditing), обеспечивающие возможность обнаружить и зафиксировать важные события, связанные с безопасностью, или любые попытки создать, получить доступ или удалить системные ресурсы;

♦ защита памяти, заключающаяся в том, что память инициализируется перед тем, как повторно используется.

На этом уровне система не защищена от ошибок пользователя, но поведение его может быть проконтролировано по записям в журнале, оставленным средствами наблюдения и аудита.

Системы уровня B основаны на помеченных данных и распределении пользователей по категориям, то есть реализуют мандатный контроль доступа. Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к дан-

ным только в соответствии с этим рейтингом. Этот уровень в отличие от уровня C защищает систему от ошибочного поведения пользователя.

Уровень A является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня B выполнения формального, математически обоснованного доказательства соответствия системы требованиям безопасности.

Различные коммерческие структуры (например, банки) особо выделяют необходимость учётной службы, аналогичной той, что предлагают государственные рекомендации C2. Любая деятельность, связанная с безопасностью, может быть отслежена и тем самым учтена. Это как раз то, чего требует стандарт для систем класса C2, и что обычно нужно банкам. Однако коммерческие пользователи, как правило, не хотят расплачиваться производительностью за повышенный уровень безопасности. A-уровень безопасности занимает своими управляющими механизмами до 90 % процессорного времени, что, безусловно, в большинстве случаев уже неприемлемо. Более безопасные системы не только снижают эффективность, но и существенно ограничивают число доступных прикладных пакетов, которые соответствующим образом могут выполняться в подобной системе. Например, для ОС Solaris (версия UNIX) есть несколько тысяч приложений, а для ее аналога B-уровня – только около ста.

Микроядерные операционные системы

Микроядро – это минимальная стержневая часть операционной системы, служащая основой модульных и переносимых расширений. Существует мнение, что большинство операционных систем следующих поколений будут обладать микроядрами. Однако имеется масса разных мнений по поводу того, как следует организовывать службы операционной системы по отношению к микроядру: как проектировать драйверы устройств, чтобы добиться наибольшей эффективности, но сохранить функции драйверов максимально независимыми от аппаратуры; следует ли выполнять операции, не относящиеся к ядру, в пространстве ядра или в пространстве пользователя; стоит ли сохранять программы имеющихся подсистем (например, UNIX) или лучше отбросить всё и начать с нуля.

Основная идея, заложенная в технологию микроядра, будь то собственно ОС или её графический интерфейс, заключается в том, чтобы конструировать необходимую среду верхнего уровня, из которой можно легко получить доступ ко всем функциональным возможностям уровня аппаратного обеспечения. При такой структуре ядро служит стартовой точкой для создания системы. Искусство разработки микроядра заключается в выборе базовых примитивов, которые должны в нём находиться для обеспечения необходимого и достаточного сервиса. В микроядре содержится и исполняется минимальное количество кода, необходимое для реализации основных системных вызовов. В число этих вызовов входят передача сообщений и организация другого общения между внешними по отношению к микроядру процессами, поддержка управления прерываниями, а также ряд некоторых других функций. Остальные функции, характерные для «обычных» (не микроядерных) ОС, обеспечиваются как модульные дополнения-процессы, взаимодействующие главным образом между собой и осуществляющие взаимодействие посредством передачи сообщений.

Микроядро является маленьким, передающим сообщения модулем системного программного обеспечения, работающим в наиболее приоритетном состоянии компьютера и поддерживающим остальную часть операционной системы, рассматриваемую как набор серверных приложений. Интерес к микроядрам возрастал по мере того, как системные разработчики реагировали на сложность современных реализации операционных систем, и поддержан тем, что исследовательское сообщество успешно продемонстрировало реализуемость концепции микроядра.

Созданная в университете Карнеги Меллон технология микроядра Mach служит основой для многих ОС.

Исполняемые микроядром функции ограничены в целях сокращения его размеров и максимизации количества кода, работающего как прикладная программа. Микроядро включает только те функции, которые требуются для определения набора абстрактных сред обработки для прикладных программ и для организации совместной работы приложений в обеспечении сервисов и в действии клиентами и

серверами. В результате микроядро обеспечивает только пять различных типов сервисов:

- ♦ управление виртуальной памятью;
- ♦ задания и потоки;
- ♦ межпроцессные коммуникации (IPC[1]);
- ♦ управление поддержкой ввода/вывода и прерываниями;
- ♦ сервисы набора хоста[1] и процессора.

Другие подсистемы и функции операционной системы, такие как системы управления файлами, поддержка внешних устройств и традиционные программные интерфейсы, размещаются в одном или более системных сервисах либо в задаче операционной системы. Эти программы работают как приложения микроядра.

Следуя концепции множественных потоков на одно задание, микроядро создаёт прикладную среду, обеспечивающую использование мультипроцессоров без требования, чтобы какая-то конкретная машина была мультипроцессорной: на однопроцессорной различные потоки просто выполняются в разные времена. Вся поддержка, требуемая для мультипроцессорных машин, сконцентрирована в сравнительно малом и простом микроядре.

Благодаря своим размерам и способности поддерживать стандартные сервисы программирования и характеристики в виде прикладных программ сами микроядра проще, чем ядра монолитных или модульных операционных систем. С микроядром функция операционной системы разбивается на модульные части, которые могут быть сконфигурированы целым рядом способов, позволяя строить большие системы добавлением частей к меньшим. Например, каждый аппаратно-независимый нейтральный сервис логически отделён и может быть сконфигурирован в широком диапазоне способов. Микроядра также облегчают поддержку мультипроцессоров созданием стандартной программной среды, которая может использовать множественные процессоры в случае их наличия, однако не требует их, если их нет. Специализированный код для мультипроцессоров ограничен самим микроядром. Более того, сети из общающихся между собой микроядер могут быть использованы для

[1] *IPC* (inter-process communication) – межпроцессные коммуникации.

обеспечения операционной системной поддержки возникающего класса массивно параллельных машин. В некоторых случаях определенной сложностью использования микроядерного подхода на практике является замедление скорости выполнения системных вызовов при передаче сообщений через микроядро по сравнению с классическим подходом. С другой стороны, можно констатировать и обратное. Поскольку микроядра малы и имеют сравнительно мало требуемого к исполнению кода уровня ядра, они обеспечивают удобный способ поддержки характеристик реального времени, требующихся для мультимедиа, управления устройствами и высокоскоростных коммуникаций. Наконец, хорошо структурированные микроядра обеспечивают изолирующий слой для аппаратных различий, которые не маскируются применением языков программирования высокого уровня. Таким образом, они упрощают перенесение кода и увеличивают уровень его повторного использования.

Наиболее ярким представителем микроядерных ОС является ОС реального времени QNX. Микроядро QNX поддерживает только планирование и диспетчеризацию процессов, взаимодействие процессов, обработку прерываний и сетевые службы нижнего уровня (более подробно об ОС QNX см. в главе 8). Микроядро обеспечивает всего лишь пару десятков системных вызовов, но благодаря этому оно может быть целиком размещено во внутреннем кэше даже таких процессоров, как Intel 486. Как известно, разные версии этой ОС имели и различные объемы ядер – от 8 до 46 Кбайт.

Чтобы построить минимальную систему QNX, требуется добавить к микроядру менеджер процессов, который создаёт процессы, управляет процессами и памятью процессов. Чтобы ОС QNX была применима не только во встроенных и бездисковых системах, нужно добавить файловую систему и менеджер устройств. Эти менеджеры исполняются вне пространства ядра, так что ядро остаётся небольшим.

[1] *Host* – главный компьютер. Сейчас этим термином обозначают любой компьютер, имеющий IP-адрес.

Монолитные операционные системы

Монолитные ОС являются прямой противоположностью микроядерным ОС. При этом можно согласиться с тем, как трактуется архитектура монолитных ОС в работе [54]. В монолитной ОС, несмотря на её возможную сильную структуризацию, очень трудно удалить один из уровней многоуровневой модульной структуры. Добавление новых функций и изменение существующих для монолитных ОС требует очень хорошего знания всей архитектуры ОС и чрезвычайно больших усилий. Поэтому более современный подход к проектированию ОС, который может быть условно назван как «клиент-серверная» технология, позволяет в большей мере и с меньшими трудозатратами реализовать перечисленные выше принципы проектирования ОС.

Модель клиент–сервер предполагает наличие программного компонента, являющегося потребителем какого-либо сервиса – *клиента,* и программного компонента, служащего поставщиком этого сервиса – *сервера.* Взаимодействие между клиентом и сервером стандартизируется, так что сервер может обслуживать клиентов, реализованных различными способами и, может быть, разными разработчиками. При этом главным требованием является использование единообразного интерфейса. Инициатором обмена обычно является клиент, который посылает запрос на обслуживание серверу, находящемуся в состоянии ожидания запроса. Один и тот же программный компонент может быть клиентом по отношению к одному виду услуг и сервером для другого вида услуг. Модель клиент–сервер является скорее удобным концептуальным средством ясного представления функций того или иного программного элемента в какой-либо ситуации, нежели технологией. Эта модель успешно применяется не только при построении ОС, но и на всех уровнях программного обеспечения и имеет в некоторых случаях более узкий, специфический смысл, сохраняя, естественно, при этом все свои общие черты.

При поддержке монолитных ОС возникает ряд проблем, связанных с тем, что все функции макроядра работают в едином адресном пространстве. Во-первых, это опасность возникновения конфликта между различными частями ядра; во-вторых – сложность подключения к ядру новых драйверов. Преимущество микроядерной

архитектуры перед монолитной заключается в том, что каждый компонент системы представляет собой самостоятельный процесс, запуск или остановка которого не отражается на работоспособности остальных процессов.

Микроядерные ОС в настоящее время разрабатываются чаще монолитных. Однако следует заметить, что использование технологии клиент–сервер – это ещё не гарантия того, что ОС станет микроядерной. В качестве подтверждения можно привести пример с ОС Windows NT, которая построена на идеологии клиент–сервер, но которую тем не менее трудно назвать микроядерной. Для того чтобы согласиться с таким высказыванием, достаточно сравнить ОС QNX и ОС Windows NT.

Требования, предъявляемые к ОС реального времени

Как известно, система реального времени (СРВ) должна давать отклик на любые непредсказуемые внешние воздействия в течение предсказуемого интервала времени. Для этого должны быть обеспечены следующие свойства:

♦ *Ограничение времени отклика,* то есть после наступления события реакция на него гарантированно последует до предустановленного крайнего срока. Отсутствие такого ограничения рассматривается как серьезный недостаток программного обеспечения.

♦ *Одновременность обработки:* даже если наступает более одного события одновременно, все временные ограничения для всех событий должны быть выдержаны. Это означает, что системе реального времени должен быть присущ параллелизм. Параллелизм достигается использованием нескольких процессоров в системе и/или многозадачного подхода.

Примерами систем реального времени являются системы управления атомными электростанциями или какими-нибудь технологическими процессами, медицинского мониторинга, управления вооружением, космической навигации, разведки, управления лабораторными экспериментами, управления автомобильными двигателями, робототехника, телеметрические системы управления, системы антиблокировки тормозов, системы сигнализации – список в принципе бесконечен.

Иногда можно услышать из разговоров специалистов, что различают системы «мягкого» и «жёсткого» реального времени. Различие между жёсткой и мягкой СРВ зависит от требований к системе – система считается жёсткой, если «нарушение временных ограничений не допустимо», и мягкой, если «нарушение временных ограничений нежелательно». Ведётся множество дискуссий о точном смысле терминов «жёсткая» и «мягкая» СРВ. Можно даже аргументировать, что мягкая СРВ не является СРВ вовсе, ибо основное требование соблюдения временных ограничений не выполнено. В действительности термин СРВ часто неправомерно применяют по отношению к быстрым системам.

Часто путают понятия СРВ и ОСРВ[1], а также неправильно используют атрибуты «мягкая» и «жёсткая». Иногда говорят, что та или иная ОСРВ мягкая или жёсткая. Нет мягких или жёстких ОСРВ. ОСРВ может только служить основой для построения мягкой или жёсткой СРВ. Сама по себе ОСРВ не препятствует тому, что ваша СРВ будет мягкой. Например, вы решили создать СРВ, которая должна работать через Ethernet по протоколу TCP/IP. Такая система не может быть жёсткой СРВ, поскольку сама сеть Ethernet в принципе непредсказуема вследствие использования случайного метода доступа к среде передачи данных, в отличие, например, от IBM Token Ring или ARC-Net, в которых используются детерминированные методы доступа.

Итак, перечислим *основные* требования к ОСРВ.

Мультипрограммность и многозадачность

Требование 1. ОС должна быть мультипрограммной и многозадачной (*многопоточной* – multi-threaded) и активно использовать прерывания для диспетчеризации.

Как указывалось выше, ОСРВ должна быть предсказуемой. Это означает не то, что ОСРВ должна быть быстрой, а то, что максимальное время выполнения того или иного действия должно быть известно заранее и должно соответствовать требованиям приложения. Так, например, ОС Windows 3.11 даже на Pentium III

[1] *ОСРВ* – операционная система реального времени

1000 MHz бесполезна для ОСРВ, ибо одно приложение может захватить управление и заблокировать систему для остальных.

Первое требование состоит в том, что ОС должна быть многопоточной по принципу абсолютного приоритета (прерываемой). То есть планировщик должен иметь возможность прервать любой поток и предоставить ресурс тому потоку, которому он более необходим. ОС (и аппаратура) должны также обеспечивать прерывания на уровне обработки прерываний.

Приоритеты задач (потоков)

Требование 2. В ОС должно существовать понятие приоритета потока. Проблема в том, чтобы определить, какой задаче ресурс требуется более всего. В идеальной ситуации ОСРВ отдаёт ресурс потоку или драйверу с ближайшим крайним сроком (это называется управлением временным ограничением, deadline driven OS). Чтобы реализовать это временное ограничение, ОС должна знать, сколько времени требуется каждому из выполняющихся потоков для завершения.

ОС, построенных по этому принципу, практически нет, так как он слишком сложен для реализации. Поэтому разработчики ОС принимают иную точку зрения: вводится понятие уровня приоритета для задачи, и временные ограничения сводят к приоритетам. Так как умозрительные решения чреваты ошибками, показатели СРВ при этом снижаются. Чтобы более эффективно осуществить указанное преобразование ограничений, проектировщик может воспользоваться теорией расписаний или имитационным моделированием, хотя и это может оказаться бесполезным. Тем не менее, так как на сегодняшний день не имеется иного решения, понятие приоритета потока неизбежно.

Наследование приоритетов

Требование 3. В ОС должна существовать система наследования приоритетов.

На самом деле именно этот механизм синхронизации и тот факт, что различные треды используют одно и то же пространство памяти, отличают их от процессов, Как мы уже знаем, процессы почти не разделяют одно и то же пространство памяти, а в основном работают в своих локальных адресных пространствах. Так, например, старые версии UNIX не являются мультитредовыми (multi-threaded).

«Старый» UNIX – многозадачная ОС, где задачами являются процессы (а не треды), которые сообщаются через потоки (pipes) и разделяемую память. Оба эти механизма используют файловую систему, а её поведение – непредсказуемо.

Комбинация приоритетов тредов и разделения ресурсов между ними приводит к другому явлению – классической проблеме инверсии приоритетов. Это можно проиллюстрировать на примере, когда есть как минимум три треда. Когда тред низшего приоритета захватил ресурс, разделяемый с тредом высшего приоритета, и начал выполняться поток среднего приоритета, выполнение треда высшего приоритета будет приостановлено, пока не освободится ресурс и не отработает тред среднего приоритета. В этой ситуации время, необходимое для завершения треда высшего приоритета, зависит от нижних приоритетных уровней, – это и есть инверсия приоритетов. Ясно, что в такой ситуации трудно выдержать ограничение на время исполнения.

Чтобы устранить такие инверсии, ОСРВ должна допускать наследование приоритета, то есть повышение уровня приоритета треда до уровня треда, который его вызывает. Наследование означает, что блокирующий ресурс тред наследует приоритет треда, который он блокирует (разумеется, это справедливо лишь в том случае, если блокируемый тред имеет более высокий приоритет).

Иногда можно услышать утверждение, что в грамотно спроектированной системе такая проблема не возникает. В случае сложных систем с этим нельзя согласиться. Единственный способ решения этой проблемы состоит в увеличении приоритета треда «вручную» прежде, чем ресурс окажется заблокированным. Разумеется, это возможно в случае, когда два треда разных приоритетов претендуют на один ресурс. В общем случае решения не существует.

Синхронизация процессов и задач

Требование 4. ОС должна обеспечивать мощные, надежные и удобные механизмы синхронизации задач. Так как задачи разделяют данные (ресурсы) и должны сообщаться друг с другом, представляется логичным, что должны существовать механизмы блокирования и коммуникации. Необходимы механизмы, гарантированно предоставляющие возможность параллельно выполняющимся задачам и

процессам оперативно обмениваться сообщениями и синхросигналами. Эти системные механизмы должны быть всегда доступны процессам, требующим реального времени. Следовательно, системные ресурсы для их функционирования должны быть распределены заранее.

Предсказуемость

Требование 5. Поведение ОС должно быть известно и достаточно точно прогнозируемо.

Времена выполнения системных вызовов и временные характеристики поведения системы в различных обстоятельствах должны быть известны разработчику. Поэтому создатель ОСРВ должен приводить следующие характеристики:

♦ латентную задержку прерывания (то есть время от момента прерывания до момента запуска задачи): она должна быть предсказуема и согласована с требованиями приложения. Эта величина зависит от числа одновременно «висящих» прерываний;

♦ максимальное время выполнения каждого системного вызова. Оно должно быть предсказуемо и не зависимо от числа объектов в системе;

♦ максимальное время маскирования прерываний драйверами и ОС.

Принципы построения интерфейсов операционных систем

Напомним, что ОС всегда выступает как интерфейс между аппаратурой компьютера и пользователем с его задачами. Под интерфейсами операционных систем здесь и далее следует понимать специальные интерфейсы системного прикладного программирования, предназначенные для выполнения следующих задач:

♦ Управление процессами, которое включает в себя следующий набор основных функций:

• запуск, приостанов и снятие задачи с выполнения;

• задание или изменение приоритета задачи;

• взаимодействие задач между собой (механизмы сигналов, семафорные примитивы, очереди, конвейеры, почтовые ящики);

• RPC (remote procedure call) – удаленный вызов подпрограмм.

♦ Управление памятью:

• запрос на выделение блока памяти;

• освобождение памяти;

• изменение параметров блока памяти (например, память может быть заблокирована процессом либо предоставлена в общий доступ);

• отображение файлов на память (имеется не во всех системах).

♦ Управление вводом/выводом:

• запрос на управление виртуальными устройствами (напомним, что управление вводом/выводом является привилегированной функцией самой ОС, и никакая из пользовательских задач не должна иметь возможности непосредственно управлять устройствами);

• файловые операции (запросы к системе управления файлами на создание, изменение и удаление данных, организованных в файлы).

Здесь мы перечислили основные наборы функций, которые выполняются ОС по соответствующим запросам от задач. Что касается пользовательского интерфейса операционной системы, то он реализуется с помощью специальных программных модулей, которые принимают его команды на соответствующем языке (возможно, с использованием графического интерфейса) и транслируют их в обычные вызовы в соответствии с основным интерфейсом системы. Обычно эти модули называют интерпретатором команд. Так, например, функции такого интерпретатора в MS-DOS выполняет модуль COMMAND.COM. Получив от пользователя команду, такой модуль после лексического и синтаксического анализа либо сам выполняет действие, либо, что случается чаще, обращается к другим модулям ОС, используя механизм API. Надо заметить, что в последние годы большую популярность получили графические интерфейсы (GUI), в которых задействованы соответствующие манипуляторы типа «мышь» или «трекбол»[1]. Указание курсором на объекты и щелчок (клик) или двойной щелчок по соответствующим клавишам приводит к каким-либо действиям – запуску программы, ассоциированной с указываемым объ-

[1] *Trackball* – в переносных компьютерах очень часто используется для управления перемещением курсора специальный шарик, который размещается рядом с клавиатурой и прокручивается пальцами.

ектом, выбору и/или активизации пунктов меню и т. д. Можно сказать, что такая интерфейсная подсистема транслирует «команды» пользователя в обращения к ОС.

Поясним также, что управление GUI – частный случай задачи управления вводом/выводом, не являющийся частью ядра операционной системы, хотя в ряде случаев разработчики ОС относят функции GUI к основному системному API.

Следует отметить, что имеются два основных подхода к управлению задачами. Так, в одних системах порождаемая задача наследует все ресурсы задачи–родителя, тогда как в других системах существуют равноправные отношения, и при порождении нового процесса ресурсы для него запрашиваются у операционной системы.

Обращения к операционной системе, в соответствии с имеющимся API, может осуществляться как посредством вызова подпрограммы с передачей ей необходимых параметров, так и через механизм программных прерываний. Выбор метода реализации вызовов функций API должен определяться архитектурой платформы.

Так, например, в операционной системе MS-DOS, которая разрабатывалась для однозадачного режима (поскольку процессор i8086 не поддерживал мультипрограммирование), использовался механизм программных прерываний. При этом основной набор функций API был доступен через точку входа обработчика int 21h.

В более сложных системах имеется не одна точка входа, а множество – по количеству функций API. Так, в большинстве операционных систем используется метод вызова подпрограмм. В этом случае вызов сначала передается в модуль API (например, это *библиотека времени выполнения*[1]), который и перенаправляет вызов соответствующим обработчикам программных прерываний, входящим в состав операционной системы. Использование механизма прерываний вызвано, главным образом, тем, что при этом процессор переводится в режим супервизора.

[1] *RTL* (run time library) – библиотека времени выполнения; она включает в себя те стандартные подпрограммы, которые система программирования подставляет на этапе компиляции. В общем случае RTL включает в себя не только модули из системы программирования, но и модули самой ОС.

Интерфейс прикладного программирования

Прежде всего необходимо однозначно разделить общий термин *API* (application program interface, интерфейс прикладного программирования) на следующие направления:

- ♦ API как интерфейс высокого уровня, принадлежащий к библиотекам RTL;
- ♦ API прикладных и системных программ, входящих в поставку операционной системы;
- ♦ прочие API.

Интерфейс прикладного программирования, как это и следует из названия, предназначен для использования прикладными программами системных ресурсов ОС и реализуемых ею функций. API описывает совокупность функций и процедур, принадлежащих ядру или надстройкам ОС.

Итак, API представляет собой набор функций, предоставляемых системой программирования разработчику прикладной программы и ориентированных на организацию взаимодействия результирующей прикладной программы с *целевой вычислительной системой.* Целевая вычислительная система представляет собой совокупность программных и аппаратных средств, в окружении которых выполняется результирующая программа. Сама результирующая программа порождается системой программирования на основании кода исходной программы, созданного разработчиком, а также объектных модулей и библиотек, входящих в состав системы программирования.

В принципе API используется не только прикладными, но и многими системными программами, как в составе ОС, так и в составе системы программирования.

Но дальше речь пойдет только о функциях API с точки зрения разработчика прикладной программы. Для системной программы существуют некоторые дополнительные ограничения на возможные реализации API.

Функции API позволяют разработчику строить результирующую прикладную программу так, чтобы использовать средства целевой вычислительной системы для выполнения типовых операций. При этом разработчик программы избавлен от необходимости создавать исходный код для выполнения этих операций.

Программный интерфейс API включает в себя не только сами функции, но и соглашения об их использовании, которые регламентируются операционной системой (ОС), архитектурой целевой вычислительной системы и системой программирования.

Существует несколько вариантов реализации API:

♦ реализация на уровне ОС;

♦ реализация на уровне системы программирования;

♦ реализация на уровне внешней библиотеки процедур и функций.

Система программирования в каждом из этих вариантов предоставляет разработчику средства для подключения функций API к исходному коду программы и организации их вызовов. Объектный код функций API подключается к результирующей программе компоновщиком при необходимости.

Возможности API можно оценивать со следующих позиций:

♦ эффективность выполнения функций API – включает в себя скорость выполнения функций и объём вычислительных ресурсов, потребных для их выполнения;

♦ широта предоставляемых возможностей;

♦ зависимость прикладной программы от архитектуры целевой вычислительной системы.

В идеале хотелось бы иметь набор функций API, выполняющихся с наивысшей эффективностью, предоставляющих пользователю все возможности современных ОС и имеющих минимальную зависимость от архитектуры вычислительной системы (ещё лучше – лишённых такой зависимости).

Добиться наивысшей эффективности выполнения функций API практически трудно по тем же причинам, по которым невозможно добиться наивысшей эффективности выполнения для любой результирующей программы. Поэтому об эффективности API можно говорить только в сравнении его характеристик с другим API.

Что касается двух других показателей, то в принципе нет никаких технических ограничений на их реализацию. Однако существуют организационные проблемы и узкие корпоративные интересы, тормозящие создание такого рода библиотек.

Реализация функций API на уровне ОС

При реализации функций API на уровне ОС за их выполнение ответственность несет ОС. Объектный код, выполняющий функции, либо непосредственно входит в состав ОС (или даже ядра ОС), либо поставляется в составе динамически загружаемых библиотек, разработанных для данной ОС. Система программирования ответственна только за то, чтобы организовать интерфейс для вызова этого кода.

В таком варианте результирующая программа обращается непосредственно к ОС. Поэтому достигается наибольшая эффективность выполнения функций API по сравнению со всеми другими вариантами реализации API.

Недостатком организации API по такой схеме является практически полное отсутствие переносимости не только кода результирующей программы, но и кода исходной программы. Программа, созданная для одной архитектуры вычислительной системы, не сможет исполняться на вычислительной системе другой архитектуры даже после того, как её объектный код будет полностью перестроен. Чаще всего система программирования не сможет выполнить перестроение исходного кода для новой архитектуры вычислительной системы, поскольку многие функции API, ориентированные на определённую ОС, будут в новой архитектуре просто отсутствовать.

Таким образом, в данной схеме для переноса прикладной программы с одной целевой вычислительной системы на другую будет требоваться изменение исходного кода программы.

Переносимости можно было бы добиться, если унифицировать функции API в различных ОС. С учётом корпоративных интересов производителей ОС такое направление их развития представляется практически невозможным. В лучшем случае при приложении определённых организационных усилий удаётся добиться стандартизации смыслового (семантического) наполнения основных функций API, но не их программного интерфейса.

Хорошо известным примером API такого рода может служить набор функций, предоставляемых пользователю со стороны ОС типа Microsoft Windows – WinAPI (Windows API). Надо сказать, что даже внутри этого корпоративного API существует определенная несогласованность, которая несколько ограничивает переноси-

мость программ между различными ОС типа Windows. Еще одним примером такого API можно считать набор сервисных функций ОС типа MS-DOS, реализованный в виде набора подпрограмм обслуживания программных прерываний.

Реализация функций API на уровне системы программирования

Если функции API реализуются на уровне системы программирования, они предоставляются пользователю в виде библиотеки функций соответствующего языка программирования. Обычно речь идет о библиотеке времени исполнения – RTL (run time library). Система программирования предоставляет пользователю библиотеку соответствующего языка программирования и обеспечивает подключение к результирующей программе объектного кода, ответственного за выполнение этих функций. Очевидно, что эффективность функций API в таком варианте будет несколько ниже, чем при непосредственном обращении к функциям ОС. Так происходит, поскольку для выполнения многих функций API библиотека RTL языка программирования должна всё равно выполнять обращения к функциям ОС. Наличие всех необходимых вызовов и обращений к функциям ОС в объектном коде RTL обеспечивает система программирования.

Однако переносимость исходного кода программы в таком варианте будет самой высокой, поскольку синтаксис и семантика всех функций будут строго регламентированы в стандарте соответствующего языка программирования. Они зависят от языка и не зависят от архитектуры целевой вычислительной системы. Поэтому для выполнения прикладной программы на новой архитектуре вычислительной системы достаточно заново построить код результирующей программы с помощью соответствующей системы программирования.

Единообразное выполнение функций языка обеспечивается системой программирования. При ориентации на различные архитектуры целевой вычислительной системы в системе программирования могут потребоваться различные комбинации вызовов функций ОС для выполнения одних и тех же функций исходного языка. Это должно быть учтено в коде RTL. В общем случае для каждой архитектуры целевой вычислительной системы будет требоваться свой код RTL языка программирования. Выбор того или иного объектного кода RTL для подключения

к результирующей программе система программирования обеспечивает автоматически.

Например, рассмотрим функции динамического выделения памяти в языках C и Pascal. В C это функции malloc, realloc и free (функции new и delete в C++), в Pascal – функции new и dispose. Если использовать эти функции в исходном тексте программы, то с точки зрения исходной программы они будут действовать одинаковым образом в зависимости только от семантики исходного кода. При этом для разработчика исходной программы не имеет значения, на какую архитектуру ориентирована его программа. Это имеет значение для системы программирования, которая для каждой из этих функций должна подключить к результирующей программе объектный код библиотеки. Этот код будет выполнять обращение к соответствующим функциям ОС. Не исключено даже, что для однотипных по смыслу функций в разных языках (например, malloc в C и new в языке Pascal выполняют схожие по смыслу действия) этот код будет выполнять схожие обращения к ОС. Однако для различных вариантов ОС этот код будет различен даже при использовании одного и того же исходного языка.

Проблема, главным образом, заключается в том, что большинство языков программирования предоставляют пользователю не очень широкий набор стандартизованных функций. Поэтому разработчик исходного кода существенно ограничен в выборе доступных функций API. Как правило, набора стандартных функций оказывается недостаточно для создания полноценной прикладной программы. Тогда разработчик может обратиться к функциям других библиотек, имеющихся в составе системы программирования. В этом случае нет гарантии, что функции, включённые в состав данной системы программирования, но не входящие в стандарт языка программирования, будут доступны в другой системе программирования. Особенно если она ориентирована на другую архитектуру целевой вычислительной системы. Такая ситуация уже ближе к третьему варианту реализации API.

Например, те же функции malloc, realloc и free в языке C фактически не входят в стандарт языка. Они входят в состав стандартной библиотеки, которая «де–факто» входит во все системы программирования, построенные на основе языка C.

Общепринятые стандарты существует для многих часто используемых функций языка. Если же взять более специфические функции, такие как функции порождения новых процессов, то для них ни в C, ни в языке Pascal не окажется общепринятого стандарта.

Реализация функций API с помощью внешних библиотек

При реализации функций API с помощью внешних библиотек они предоставляются пользователю в виде библиотеки процедур и функций, созданной сторонним разработчиком. Причем разработчиком такой библиотеки может выступать тот же самый производитель.

Система программирования ответственна только за то, чтобы подключить объектный код библиотеки к результирующей программе. Причем внешняя библиотека может быть и динамически загружаемой (загружаемой во время выполнения программы).

С точки зрения эффективности выполнения этот метод реализации API имеет самые низкие результаты, поскольку внешняя библиотека обращается как к функциям ОС, так и к функциям RTL языка программирования. Только при очень высоком качестве внешней библиотеки её эффективность становится сравнимой с библиотекой RTL.

Если говорить о переносимости исходного кода, то здесь требование только одно – используемая внешняя библиотека должна быть доступна в любой из архитектур вычислительных систем, на которые ориентирована прикладная программа. Тогда удаётся достигнуть переносимости. Это возможно, если используемая библиотека удовлетворяет какому-то принятому стандарту, а система программирования поддерживает этот стандарт.

Например, библиотеки, удовлетворяющие стандарту POSIX (см. следующий подраздел), доступны в большинстве систем программирования для языка C. И если прикладная программа использует только библиотеки этого стандарта, то ее исходный код будет переносимым. Еще одним примером является широко известная библиотека графического интерфейса XLib, поддерживающая стандарт графической среды X Window.

Для большинства специфических библиотек отдельных разработчиков это не так. Если пользователь использует какую-то библиотеку, то она ориентирована на ограниченный набор доступных архитектур целевой вычислительной системы. Примерами могут служить библиотеки MFC (Microsoft foundation classes) фирмы Microsoft и VCL (visual controls library) фирмы Borland, жёстко ориентированные на архитектуру ОС типа Windows. Тем не менее, многие фирмы-разработчики сейчас стремятся создать библиотеки, которые бы обеспечивали переносимость исходного кода приложений между различными архитектурами целевой вычислительной системы. Пока ещё такие библиотеки не получили широкого распространения, но есть несколько попыток их реализации – например, библиотека CLX производства фирмы Borland ориентирована на архитектуру ОС типа Linux и ОС типа Windows.

В целом развитие функций прикладного API идет в направлении попытки создать библиотеки API, обеспечивающие широкую переносимость исходного кода. Однако, учитывая корпоративные интересы различных производителей и сложившуюся ситуацию на рынке системного программного обеспечения, в ближайшее время вряд ли удастся достичь значительных успехов в этом направлении. Разработка широко применимого стандарта API пока ещё остается делом будущего.

Поэтому разработчики системных программ вынуждены оставаться в довольно узких рамках ограничений стандартных библиотек языков программирования.

Что касается прикладных программ, то гораздо большую перспективу для них предоставляют технологии, связанные с разработками в рамках архитектуры «клиент–сервер» или трехуровневой архитектуры создания приложений. В этом направлении ведущие производители ОС, СУБД и систем программирования скорее достигнут соглашений, чем в направлении стандартизации API.

Итак, нами были рассмотрены основные принципы, цели и подходы к реализации системных API. Отметим ещё один очень важный момент: желательно, что–бы интерфейс прикладного программирования не зависел от системы программирования. Конечно, были одно время персональные компьютеры, у которых базовой системой программирования выступал интерпретатор с языка Basic, но это скорее

исключение. Обычно базовые функции API не зависят от системы программирования и могут использоваться из любой системы программирования, хотя и с применением соответствующих правил построения вызывающих последовательностей. В то же время в ряде случаев система программирования может сама генерировать обращения к функциям API. Например, мы можем написать в программе вызов функции по запросу 256 байт памяти

```
unsigned char * ptr = malloc (256);
```

Система программирования языка С сгенерирует целую последовательность обращений. Из кода пользовательской программы будет осуществлен вызов библиотечной функции malloc, код которой расположен в RTL языка С. Библиотека времени выполнения в данном случае реализует вызов malloc уже как вызов системной функции API HeapAlloc

```
LPVOID HeapAlloc{
HANDLE hHeap, // handle to the private heap block – указатель на блок
DWORD dwFlags, //heap allocation control flags – свойства блока
DWORD dwBytes // number of bytes to allocate – размер блока
};
```

Параметры выделяемого блока памяти в таком случае задаются системой программирования, и пользователь лишён возможности задавать их напрямую. С другой стороны, если это необходимо, возможно использование функций API прямо в тексте программы.

```
unsigned char * ptr = (LPVOID) HeapAlloc( GetProcessHeap(), 0, 256);
```

В этом случае программирование вызова немного усложняется, но получаемый конечный результат будет, как правило, короче и, что самое важное, будет работать эффективнее. Следует отметить, что далеко не все возможности API доступны через обращения к функциям системы программирования. Непосредственное обращение к функциям API позволяет пользователю обращаться к системным ресурсам более эффективным способом. Однако это требует знания функций API, количество которых нередко достигает нескольких сотен.

Как правило, API не стандартизированы. В каждом конкретном случае набор вызовов API определяется, прежде всего, архитектурой ОС и её назначением. В то же время принимаются попытки стандартизировать некоторый базовый набор функций, поскольку это существенно облегчает перенос приложений с одной ОС в другую. Таким примером может служить очень известный и, пожалуй, один из самых распространенных стандартов – стандарт POSIX. В этом стандарте перечислен большой набор функций, их параметров и возвращаемых значений. Стандартизированными, согласно POSIX, являются не только обращения к API, но и файловая система, организация доступа к внешним устройствам, набор системных команд[1]. Использование в приложениях этого стандарта позволяет в дальнейшем легко переносить такие программы с одной ОС в другую путем простейшей перекомпиляции исходного текста.

Частным случаем попытки стандартизации API является внутренний корпоративный стандарт компании Microsoft, известный как WinAPI. Он включает в себя следующие реализации: Win 16, Win32s, Win32, WinCE. С точки зрения WinAPI (в силу ряда идеологических причин – обязательный графический «оконный» интерфейс пользователя), базовой задачей является окно. Таким образом, WinAPI изначально ориентирован на работу в графической среде. Однако базовые понятия дополнены традиционными функциями, в том числе частично поддерживается стандарт POSIX.

Платформенно-независимый интерфейс POSIX

POSIX (Portable Operating System Interface for Computer Environments) – платформенно независимый системный интерфейс для компьютерного окружения. Это стандарт IEEE, описывающий системные интерфейсы для открытых операционных систем, в том числе оболочки, утилиты и инструментарии. Помимо этого, согласно POSIX, стандартизированными являются задачи обеспечения безопасности, задачи реального времени, процессы администрирования, сетевые функции и обработка транзакций. Стандарт базируется на UNIX-системах, но допускает реализацию и в других ОС.

[1] В данном контексте под системными командами следует понимать некий набор программ, позволяющих управлять вычислительными процессами. Например, pstat, kill, dir и др.

POSIX возник как попытка всемирно известной организации IEEE[2] пропагандировать переносимость приложений в UNIX-средах путём разработки абстрактного, платформенно-независимого стандарта. Однако POSIX не ограничивается только UNIX-системами; существуют различные реализации этого стандарта в системах, которые соответствуют требованиям, предъявляемым стандартом IEEE Standard 1003.1-1990 (POSIX.1). Например, известная ОС реального времени QNX соответствует спецификациям этого стандарта, что облегчает перенос приложений в эту систему, но UNIX-системой не является ни в каком виде, ибо её архитектура использует абсолютно иные принципы.

Этот стандарт подробно описывает VMS (virtual memory system, систему виртуальной памяти), многозадачность (MPE, multiprocess executing) и технологию переноса операционных систем (CTOS). Таким образом, на самом деле POSIX представляет собой множество стандартов, именуемых POSIX.1 – POSIX.12.

Таблица 5.1. Семейство стандартов **POSIX**

Стандарт	Стандарт ISO	Краткое описание
POSIX.0	**Нет**	**Введение в стандарт открытых систем. Данный документ не является стандартом в чистом виде, а представляет собой рекомендации и краткий обзор технологий**
POSIX.1	**Да**	**Системный API (язык C)**
POSIX.2	**Нет**	**Оболочки и утилиты (одобренные IEEE)**
POSIX.3	**Нет**	**Тестирование и верификация**
POSIX.4	**Нет**	**Задачи реального времени и нити**
POSIX.5	**Да**	**Использование языка ADA применительно к стандарту POSIX.1**
POSIX.6	**Нет**	**Системная безопасность**
POSIX.7	**Нет**	**Администрирование системы**
POSIX.8	**Нет**	**Сети «Прозрачный» доступ к файлам Абстрактные сетевые интерфейсы, не зависящие от физических протоколов RPC (remote procedure calls, вызовы удаленных процедур)**
POSIX.9	**Да**	**Использование языка FORTRAN применительно к стандарту POSIX. 1**
POSIX.10	**Нет**	**Super-computing Application Environment Profile (AEP)**
POSIX.11	**Нет**	**Обработка транзакций AEP**
POSIX. 12	**Нет**	**Графический интерфейс пользователя (GUI)**

[2] *IEEE* (Institute of Electrical and Electronical Engineers) – американский Институт инженеров по электротехнике и радиоэлектронике.

В табл. 5.1 приведены основные направления, описываемые данными стандартами. Следует также особо отметить, что POSIX.1 предполагает язык С как основной язык описания системных функций API.

Таким образом, программы, написанные с соблюдением данных стандартов, будут одинаково выполняться на всех POSIX-совместимых системах. Однако стандарт в некоторых случаях носит лишь рекомендательный характер. Часть стандартов описана очень строго, тогда как другая часть только поверхностно раскрывает основные требования. Нередко программные системы заявляются как POSIX–совместимые, хотя таковыми их назвать нельзя. Причины кроются в формальности подхода к реализации POSIX-интерфейса в различных ОС. На рис. 5.1 изображена типовая схема реализации строго соответствующего POSIX приложения.

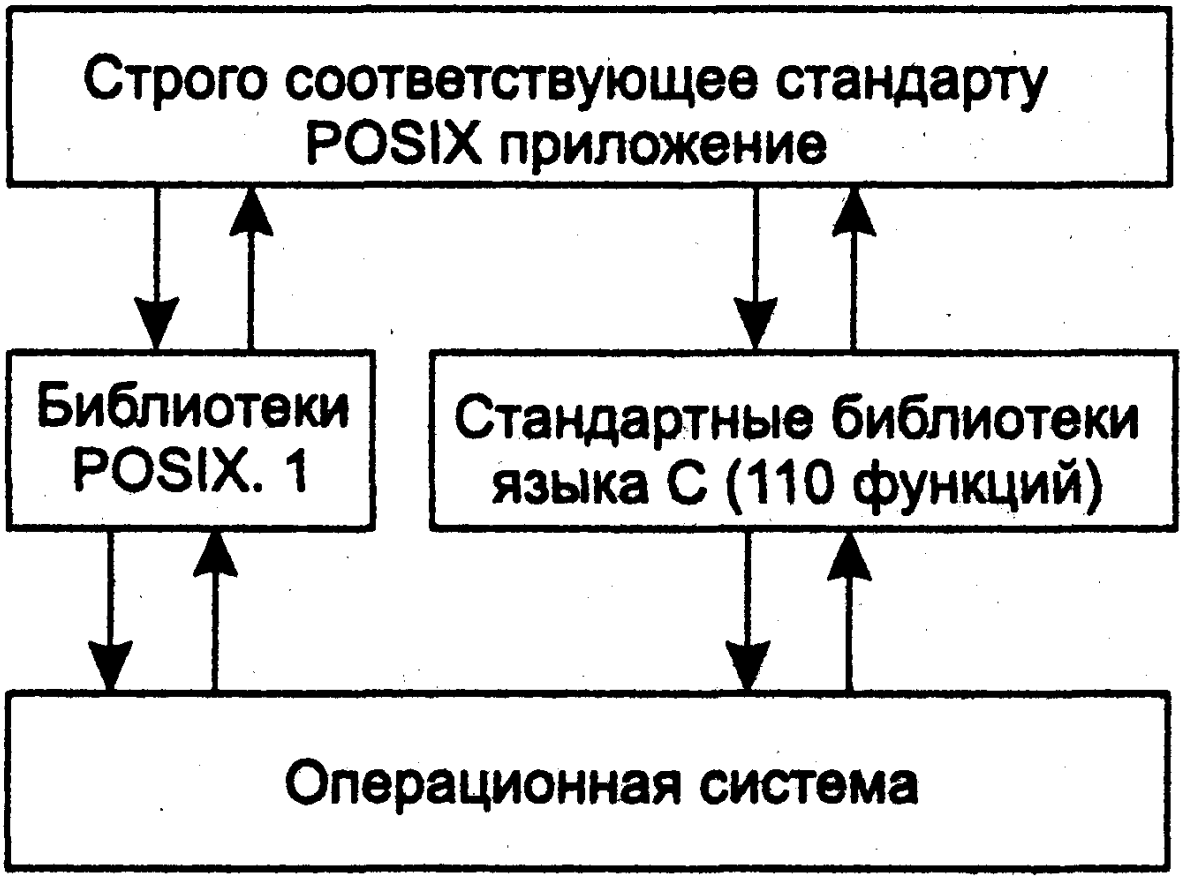

Рис.5.1. Приложения, строго соответствующие стандарту POSIX

Из рис.5.1 видно, что для взаимодействия с операционной системой программа использует только библиотеки POSIX.1 и стандартную библиотеку RTL языка С, в которой возможно использование лишь 110 различных функций, также описанных стандартом POSIX.1.

К сожалению, достаточно часто с целью увеличить производительность той или иной подсистемы либо из соображений введения фирменных технологий, которые ограничивают использование приложения соответствующей операционной, средой, при программировании используются другие функции, не отвечающие стандарту POSIX.

Реализации POSIX API на уровне операционной системы различны. Если UNIX-системы в своём абсолютном большинстве изначально соответствуют спецификациям IEEE Standard 1003.1-1990, то WinAPI не является POSIX–совместимым. Однако для поддержки данного стандарта в операционной системе MS Windows NT введен специальный модуль поддержки POSIX API, работающий на уровне привилегий пользовательских процессов. Данный модуль обеспечивает конвертацию и передачу вызовов из пользовательской программы к ядру системы и обратно, работая с ядром через WinAPI. Прочие, приложения, написанные с использованием WinAPI, могут передавать информацию POSIX-приложениям через стандартные механизмы потоков ввода/вывода (stdin, stdout) [97].

Пример программирования в различных API ОС

Для наглядной демонстрации принципиальных различий API наиболее популярных современных операционных систем для ПК рассмотрим простейший пример, в котором реализуется следующая задача.

Постановка задачи: необходимо подсчитать количество пробелов в текстовых файлах, имена которых должны указываться в командной строке. Рассмотрим два варианта программы, решающей эту задачу, – для Windows (с использованием WinAPI) и для Linux (POSIX API).

Поскольку нас интересует работа с параллельными задачами, пусть при выполнении программы для каждого перечисленного в командной строке файла создаётся свой процесс или задача (тред), который параллельно с другими процессами (тредами) производит работу по подсчёту пробелов в «своём» файле. Результатом работы программы будет являться список файлов с подсчитанным количеством пробелов для каждого.

Следует обратить особое внимание на то, что приведенная ниже конкретная реализация данной задачи не является единственно возможной. В обеих рассматриваемых операционных системах существуют различные методы как работы с файловой системой, так и управления процессами. В данном случае рассматривается только один, но наиболее характерный для соответствующего API вариант.

Текст программы для Windows (WinAPI)

Для того чтобы было удобнее сравнивать эту и следующую программы, а также, из-за того, что настоящая задача не требует для своего решения оконного интерфейса, в нижеприведённом тексте использованы только те вызовы API, которые не затрагивают графический интерфейс. Конечно, нынче редко какое приложение не использует возможностей GUI, но в нашем случае зато сразу можно увидеть разницу в организации параллельной работы запускаемых вычислений.

```
#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
// Название: processFile
// Описание: исполняемый код треда
// Входные параметры: lpFileName – имя файла для обработки
// Выходные параметры: нет
//
DWORD processFile(LPVOID lpFileName)
{
 HANDLE handle;  // описатель файла
 DWORD numRead, total = 0;
 char buf;
 // запрос к ОС на открытие файла (только для чтения)
 handle = CreateFile( (LPCTSTR)lpFileName, GENERIC_READ,
 FILE_SHARE_READ, NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);
 // цикл чтения до конца файла
 do {
      // чтение одного символа из файла
     ReadFile( handle, (LPVOID) &buf, 1, &numRead, NULL);
     if (buf == 0x20)  total++;
    } while ( numRead > 0);
 fprintf( stderr, "(ThreadID: %Lu), File %s, spaces = %d\n",
 GetCurrentThreadId(), lpFileName, total);
 // закрытие файла
 CloseHandle( handle);
 return(0);
}
// Название: main
// Описание: главная программа
// Входные параметры: список имён файлов для обработки
```

```
// Выходные параметры: нет
//
int main(int argc, char *argv[ ])
{
 int i;
 DWORD pid;
 HANDLE hThrd[255];  // массив ссылок на треды
 // для всех файлов, перечисленных в командной строке
 for (i = 0; i< (argc-1); i++)
      {
      // запуск треда – обработка одного файла
      hThrd[i] = CreateThread( NULL, 0x4000,
              (LPTHREAD_START_ROUTINE) processFile, (LPVOID) argv[i+1], 0, &pid);
      fprintf( stdout, "processFile started (HND=%d)\n", hThrd[i]);
     }
 // ожидание окончания выполнения всех запущенных тредов
 WaitForMultipleObjects( argc-1, hThrd, true, INFINITE);
 return(0);
}
```

Обратите внимание, что основная программа запускает треды и ждёт окончания их выполнения.

Текст программы для Linux (POSIX API)

```
#include <sys/types.h>
#include <sys/stat.h>
#include <wait.h>
#include <fcntl.h>
#include <stdio.h>
// Название: processFile
//Описание: обработка файла, подсчет кол-ва пробелов
// Входные параметры: fileName – имя файла для обработки
// Выходные параметры: кол-во пробелов в файле
//
int processFile( char *fileName)
{
 nt handle, numRead, total = 0;
 char buf;
 // запрос к ОС на открытие файла (только для чтения)
 handle = open( fileName, O_RDONLY);
 // цикл чтения до конца файла
 do
    {
     // чтение одного символа из файла
     numRead = read( handle, &buf, 1);
     if (buf = 0x20)  total++;
    } while (numRead > 0);
 // закрытие файла
 close( handle);
 return( total);
}
// Название: main
```

```
// Описание: главная программа
// Входные параметры: список имён файлов для обработки
// Выходные параметры: нет
//
int main(int argc, char *argv[ ])
{
 int i, pid, status;
 // для всех файлов, перечисленных в командной строке
 for (i = 1; i< argc; i++)
      {
       // запускаем дочерний процесс
       pid = fork();
       if (pid == 0)
          {
           // если выполняется дочерний процесс
           // вызов функции счёта количества пробелов в файле
           printf( "(PID: %d),  File %s, spaces = %d\n", getpid(), argv[i], processFile( argv[i]));
          // выход из процесса
          exit();
         }
        // если выполняется родительский процесс
        else
          printf( "processFile started (pid = %d)\n", pid);
      }
 // ожидание окончания выполнения всех запущенных процессов
 if (pid != 0)  while (wait(&status )>0);
 return;
}
```

Из этого текста видно, что в этом случае все вычисления принимают статус процессов, а не тредов.

В заключение можно заметить, что очень трудно сравнивать API. При их разработке создатели, как правило, стараются реализовать полный набор основных функций, используя которые можно решать различные задачи, хотя, порой, и различными способами. Один набор будет хорош для одного набора задач, другой – для иного набора задач. Тем более что фактически у нас сейчас существенно ограниченное множество API. Причина в том, что доминируют наиболее распространённые ОС, на распространение которых в большей степени оказали влияние не достоинства или недостатки этих ОС и их API, а правильная маркетинговая политика фирм, их создавших.

Контрольные вопросы и задачи

Вопросы для проверки

1 Перечислите и поясните основные принципы построения операционных систем.

2 Расскажите об основных моментах, характерных для микроядерных ОС. Какие основные функции должно выполнять микроядро ОС?

3 Перечислите основные требования, предъявляемые к операционным системам реального времени.

4 Какие задачи возлагаются на интерфейс прикладного программирования (API)?

5 Какими могут быть варианты реализации API? В чем заключаются достоинства и недостатки каждого варианта?

6 Что такое библиотека времени выполнения (RTL)?

7 Что такое POSIX? Какими преимуществами обладают программы, созданные с использованием только стандартных функций, предусмотренных POSIX?

ГЛАВА 6 Проектирование параллельных взаимодействующих вычислительных процессов

При создании современных приложений, позволяющих использовать все возможности операционных систем в плане организации параллельных и распределённых вычислений, одной из важнейших проблем является проблема синхронизации взаимодействия параллельных вычислительных процессов, обмена между ними данными. Существующие методы синхронизации и обмена сообщениями различаются по таким параметрам, как удобство использования при программировании параллельных процессов, стоимость реализации, эффективность функционирования созданных приложений и всей вычислительной системы в целом.

Операционные системы имеют в своем составе различные средства синхронизации. Знание этих средств и их правильное использование позволяет создавать программы, которые при своей работе осуществляют корректный обмен информацией, а также исключают возможность возникновения тупиковых ситуаций.

В настоящем разделе рассматриваются основные понятия и проблемы, характерные для параллельных процессов. Описываются основные механизмы синхронизации, даётся их сравнительный анализ, приводятся примеры характерных программ, использующих данные механизмы.

Независимые и взаимодействующие вычислительные процессы

Основной особенностью мультипрограммных операционных систем является то, что в их среде параллельно развивается несколько (последовательных) вычислительных процессов. С точки зрения внешнего наблюдателя эти последовательные вычислительные процессы, выполняются одновременно, мы будем использовать термин «параллельно». При этом под *параллельными* понимаются не только процессы, одновременно развивающиеся на различных процессорах, каналах и устройствах ввода/вывода, но и те последовательные процессы, которые разделяют центральный процессор и хотя бы частично перекрываются во времени. Любая мультипрограммная операционная система вместе с параллельно выполняющимися в ней задачами пользователей может быть логически описана как совокупность последовательных процессов, которые, с одной стороны, состязаются за использование ресурсов, переходя из одного состояния в другое, а с другой – действуют почти независимо друг от друга, но образуют систему вследствие установления всевозможного рода связей между ними (путем пересылки сообщений и синхронизирующих сигналов).

Итак, *параллельными* мы будем называть такие последовательные вычислительные процессы, которые одновременно находятся в каком-либо активном состоянии. Два параллельных процесса могут быть *независимыми* (independing processes) либо *взаимодействующими* (cooperating processes).

Независимыми являются процессы, множества переменных которых не пересекаются. Под переменными в этом случае понимают файлы данных, а также области оперативной памяти, сопоставленные определённым в программе и промежуточным переменным. Независимые процессы не влияют на результаты работы друг друга, так как не могут изменить значения переменных другого независимого процесса. Они могут только явиться причиной задержек исполнения других процессов, так как вынуждены разделять ресурсы системы.

Взаимодействующие процессы совместно используют некоторые (общие) переменные, и выполнение одного процесса может повлиять на выполнение другого.

Как мы уже говорили, при выполнении вычислительные процессы разделяют ресурсы системы. Подчеркнём, что при рассмотрении вопросов синхронизации вычислительных процессов из числа разделяемых ими ресурсов исключаются: центральный процессор и программы, реализующие эти процессы; то есть с логической точки зрения каждому процессу соответствуют свои процессор и программа, хотя в реальных системах обычно несколько процессов разделяют один процессор и одну или несколько программ. Многие ресурсы вычислительной системы могут совместно использоваться несколькими процессами, но в каждый момент времени к разделяемому ресурсу может иметь доступ только один процесс. Ресурсы, которые не допускают одновременного использования несколькими процессами, называются *критическими.*

Если нескольким вычислительным процессам необходимо пользоваться критическим ресурсом в режиме разделения, им следует синхронизировать свои действия таким образом, чтобы ресурс всегда находился в распоряжении не более чем одного из процессов. Если один процесс пользуется в данный момент критическим ресурсом, то все остальные процессы, которым нужен этот ресурс, должны получить отказ и ждать, пока он не освободится. Если в операционной системе не предусмотрена защита от одновременного доступа процессов к критическим ресурсам, в ней могут возникать ошибки, которые трудно обнаружить и исправить. Основной причиной возникновения этих ошибок является то, что процессы в мультипрограммных операционных системах развиваются с различными скоростями, а отно-

сительные скорости развития каждого из взаимодействующих процессов неизвестны и не подвластны ни одному из них. Более того, на их скорости могут влиять решения планировщиков, касающиеся других процессов, с которыми ни одна из этих программ не взаимодействует. Кроме того, содержание одного процесса и скорость его исполнения обычно неизвестны другому процессу. Поэтому влияние, которое оказывают друг на друга взаимодействующие процессы, не всегда предсказуемо и воспроизводимо.

Взаимодействовать могут либо *конкурирующие процессы,* либо процессы, совместно выполняющие общую работу. Конкурирующие процессы, на первый взгляд, действуют относительно независимо, но они имеют доступ к общим переменным.

Процессы, выполняющие общую совместную работу таким образом, что результаты вычислений одного процесса в явном виде передаются другому, то есть их работа построена именно на обмене данными, называются *сотрудничающими.* Взаимодействие сотрудничающих процессов удобно всего рассматривать в схеме «производитель – потребитель» (producer – consumer) или, как часто говорят – «поставщик – потребитель».

Процесс P1		Процесс P2	
№ оператора		№ оператора	
1	R1:=X	4	R2:=X
2	R1:=R1+X	5	R2:=R2+1
3	X:=R1	6	X:=R2

Рис.6.1. Пример конкурирующих процессов

В качестве первого примера рассмотрим работу двух процессов P1 и P2 с общей переменной X. Пусть оба процесса асинхронно, независимо один от другого, изменяют (например, увеличивают) значение переменной **X,** считывая её значение

в локальную область памяти R_i[1], при этом каждый процесс выполняет некоторые последовательности операций во времени (рис. 6.1).

Здесь мы рассмотрим не все операторы каждого из процессов, а только те, в которых осуществляется работа с общей переменной X. Каждому из операторов мы присвоили некоторый условный номер.

Поскольку при мультипрограммировании процессы могут иметь различные скорости исполнения, то может иметь место любая последовательность выполнения операций во времени. Если сначала будут выполнены все операции процесса P1, а уже потом – все операции процесса P2 (или, наоборот, сначала операции 4-6, а затем – операции 1-3), то в итоге переменная X получит значение, равное X+2 (рис. 6.2).

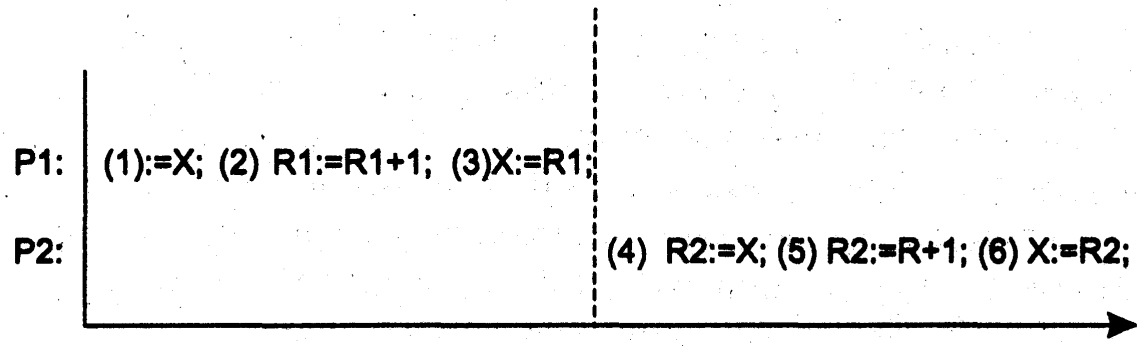

Рис.6.2. Первый вариант развития событий при выполнении процессов

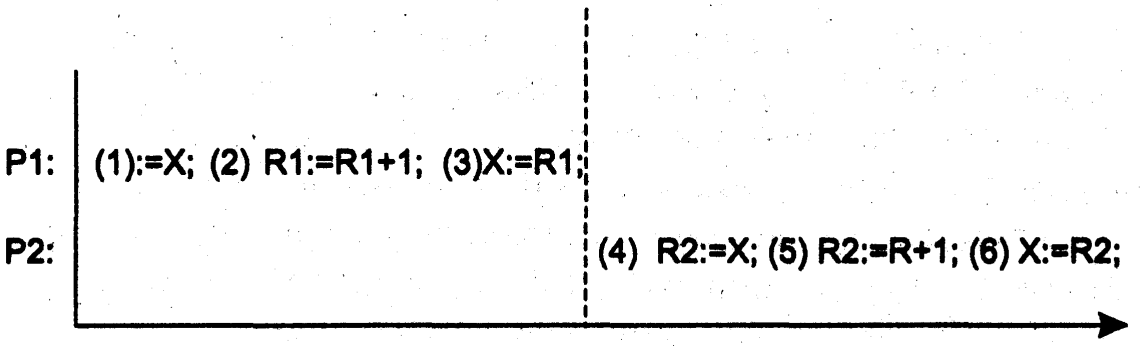

Рис.6.3. Второй вариант развития событий при выполнении процессов

Однако, если в промежуток времени между выполнением операций 1 и 3 будет выполнена хотя бы одна из операций 4-6 (рис. 6.3), то значение переменной X после выполнения всех операций будет не (X+2), а (X+1).

Понятно, что это очень серьезная (и, к сожалению, неисправимая, так как её нельзя проконтролировать) ошибка. Например, если бы процессы P1 и P2 осущест-

[1] Ri – имя переменной для процесса с номером i.

вляли продажу билетов и переменная X фиксировала количество уже проданных, то в результате некорректного взаимодействия было бы продано несколько билетов на одно и то же место.

В качестве второго примера рассмотрим ситуацию, которая ещё совсем недавно была достаточно актуальной для первых персональных компьютеров. Пусть на ПК с простейшей однопрограммной операционной системой (типа MS-DOS) установлена некоторая резидентная программа с условным названием TIME, которая по нажатию на комбинацию клавиш (например, Ctrl+T) воспроизводит на экране дисплея время. Допустим, что значения переменных, указывающих час, минуты и секунды, равны 18:20:59, причём вывод на дисплей осуществляется справа налево (то есть в порядке: секунды, минуты, часы). Пусть сразу же после передачи программой TIME на дисплей информации «59 секунд» генерируется прерывание от таймера и значение времени обновляется: 18:21:00.

После этого программа TIME, прерванная таймером, продолжит своё выполнение, и на дисплей будут выданы значения: минуты = 21, часы = 18. В итоге на экране мы увидим: 18:21:59.

Рассмотрим теперь несколько иной случай развития событий обновления значений времени по сигналу таймера. Если программа ведения системных часов после вычислений количества секунд 59+1 = 60 и замены его на 00 прерывается от нажатия клавиш Ctrl+T, то есть программа не успевает осуществить пересчёт количества минут, то время, индицируемое на дисплее, станет равным 18:20:00. И в этом случае мы получим неверное значение времени.

Наконец, в качестве третьего примера приведем пару процессов, которые изменяют различные поля записей служащих какого-либо предприятия [37]. Пусть процесс АДРЕС изменяет домашний адрес служащего, а процесс СТАТУС – его должность и зарплату. Пусть каждый процесс копирует всю запись СЛУЖАЩИЙ в свою рабочую область. Предположим, что каждый процесс должен обработать некоторую запись ИВАНОВ. Предположим также, что после того, как процесс АДРЕС скопировал запись ИВАНОВ в свою рабочую область, но до того, как он записал скорректированную запись обратно, процесс СТАТУС скопировал первона-

чальную запись ИВАНОВ в свою рабочую область. Изменения, выполненные тем из процессов, который первым запишет скорректированную запись назад в файл СЛУЖАЩИЕ, будут утеряны и, возможно, никто не будет знать об этом.

Чтобы предотвратить некорректное исполнение конкурирующих процессов вследствие нерегламентированного доступа к разделяемым переменным, необходимо ввести механизм *взаимного исключения,* который не позволит двум процессам одновременно обращаться к разделяемым переменным.

Кроме реализации в операционной системе средств, организующих взаимное исключение и тем самым регулирующих доступ процессов к критическим ресурсам, в ней должны быть предусмотрены средства, синхронизирующие работу взаимодействующих процессов. Другими словами, процессы должны обращаться к неким средствам не только ради синхронизации с целью взаимного исключения, но и чтобы обмениваться данными.

Допустим, что «поставщика – это процесс, который отправляет порции информации (сообщения) другому процессу, имя которого «потребитель». Например, процесс пользователя, порождающий строки для вывода, может выступать как «поставщик», а процесс, который выводит эти строки на печать, – как «потребитель». Один из методов, применяемых при реализации передачи сообщений, состоит в том, что заводится *пул*[1] свободных буферов, каждый из которых может содержать одно сообщение (длина сообщения может быть произвольной, но ограниченной).

В этом случае между процессами «поставщик» и «потребитель» будем иметь очередь заполненных буферов, содержащих сообщения. Когда «поставщик» хочет послать очередное сообщение, он добавляет в конец этой очереди ещё один буфер. «Потребитель», чтобы получить сообщение, забирает из очереди буфер, который стоит в её начале. Такое решение, хотя и кажется тривиальным, требует, чтобы «поставщик» и «потребитель» синхронизировали свои действия. Например, они должны следить за количеством свободных и заполненных буферов. «Поставщик» может передавать сообщения только до тех пор, пока имеются свободные буферы.

[1] *Пул* – pool – это совокупность однородных, динамически распределяемых объектов, например, блоков памяти одинаковой длины.

Аналогично, «потребитель» может получать сообщения только если очередь не пуста. Ясно, что для учёта заполненных и свободных буферов нужны разделяемые переменные, поэтому для сотрудничающих процессов, как и для конкурирующих, тоже возникает необходимость во взаимном исключении.

Таким образом, до окончания обращения одной задачи к общим переменным следует исключить возможность обращения к ним другой задачи. Эта ситуация и называется взаимным исключением. Другими словами, при организации различного рода взаимодействующих процессов приходится организовывать взаимное исключение и решать проблему корректного доступа к общим переменным (критическим ресурсам). Те места в программах, в которых происходит обращение к критическим ресурсам, называются *критическими секциями* или критическими интервалами (Critical Section – CS). Решение этой проблемы заключается в организации такого доступа к критическому ресурсу» когда только одному процессу разрешается входить в критическую секцию. Данная задача только на первый взгляд кажется простой, ибо критическая секция, вообще говоря, не является последовательностью операторов программы, а является процессом, то есть последовательностью действий, которые выполняются этими операторами. Другими словами, несколько процессов, которые выполняются по одной и той же программе, могут выполнять критические интервалы, базирующиеся на одной и той же последовательности операторов программы.

Когда какой-либо процесс находится в своём критическом интервале, другие процессы могут, конечно, продолжать своё исполнение, но без входа в их критические секции. Взаимное исключение необходимо только в том случае, когда процессы обращаются к разделяемым, общим данным. Если же они выполняют операции, которые не приводят к конфликтным ситуациям, они должны иметь возможность работать параллельно. Когда процесс выходит из своего критического интервала, то одному из остальных процессов, ожидающих входа в свои критические секции, должно быть разрешено продолжить работу (если в этот момент действительно есть процесс в состоянии ожидания входа в свой критический интервал).

Обеспечение взаимоисключения является одной из ключевых проблем параллельного программирования. При этом можно перечислить следующие требования к критическим секциям [37, 92]:

♦ в любой момент времени только один процесс должен находиться в своей критической секции;

♦ ни один процесс не должен находиться в своей критической секции бесконечно долго;

♦ ни один процесс не должен ждать бесконечно долго входа в свой критический интервал. В частности:

• никакой процесс, бесконечно долго находящийся вне своей критической секции (что допустимо), не должен задерживать выполнение других процессов, ожидающих входа в свои критические секции. Другими словами, процесс, работающий вне своей критической секции, не должен блокировать критическую секцию другого процесса;

• если два процесса хотят войти в свои критические интервалы, то принятие решения о том, кто первым войдет в критическую секцию, не должно откладываться бесконечно долго;

♦ если процесс, находящийся в своём критическом интервале, завершается либо естественным, либо аварийным путем, то режим взаимоисключения должен быть отменён, с тем чтобы другие процессы получили возможность входить в свои критические секции.

Было предложено несколько способов решения этой проблемы – программные и аппаратные; частные, низкоуровневые и глобальные, высокоуровневые; предусматривающие свободное взаимодействие между процессами и требующие строгого соблюдения жёстких протоколов.

Средства синхронизации и связи при проектировании взаимодействующих вычислительных процессов

Все известные средства для решения проблемы взаимного исключения основаны на использовании специально введённых аппаратных возможностей, к кото-

рым относятся блокировка памяти, специальные команды типа «проверка и установка» и управление системой прерываний, позволяющее организовать такие механизмы, как семафорные операции, мониторы, почтовые ящики и др. С помощью перечисленных средств можно разрабатывать взаимодействующие процессы, при исполнении которых будут корректно решаться все задачи, связанные с проблемой критических интервалов. Рассмотрим эти средства в порядке их появления, а значит, по мере их усложнения, перехода к функциям операционной системы и увеличения предоставляемых ими удобств для пользователя. При этом будем опираться на далеко не новую, но все же ещё достаточно актуальную работу Дейкстры[1] [28].

Использование блокировки памяти при синхронизации параллельных процессов

Все вычислительные машины и системы (в том числе и с многопортовыми блоками оперативной памяти) имеют такое средство для организации взаимного исключения, как *блокировка памяти.* Это средство запрещает одновременное исполнение двух (и более) команд, которые обращаются к одной и той же ячейке памяти. Поскольку в некоторой ячейке памяти хранится значение разделяемой переменной, то получить доступ к ней может только один процесс, несмотря на возможное совмещение выполнения команд во времени на различных процессорах (или на одном процессоре, но с конвейерной организацией параллельного выполнения команд).

Механизм блокировки памяти предотвращает одновременный доступ к разделяемой переменной, но не предотвращает чередование доступа. Таким образом, если критические интервалы исчерпываются одной командой обращения к памяти, данного средства может быть достаточно для непосредственной реализации взаимного исключения. Если же критические секции требуют более одного обращения к памяти, то задача становится сложной, но алгоритмически разрешимой. Рассмотрим различные попытки использования механизма блокировки памяти для организации взаимного исключения при выполнении критических интервалов и покажем

[1] На наш взгляд, эта работа Дейкстры очень полезна с методической точки зрения. Она позволяет понять наиболее тонкие моменты в этой проблематике.

некоторые важные моменты, пренебрежение которыми приводит к неприемлемым или даже ошибочным решениям.

Возможные проблемы при организации взаимного исключения посредством использования только блокировки памяти

Пусть имеются два (или более) циклических процесса, в которых есть абстрактные критические секции, то есть каждый из процессов состоит из двух частей: некоторого критического интервала и оставшейся части кода, в которой нет работы с общими (критическими) переменными. Пусть эти два процесса асинхронно разделяют во времени единственный процессор либо выполняются на отдельных процессорах, каждый из которых имеет доступ к некоторой общей области памяти, с которой и работают критические секции. Проиллюстрируем эту ситуацию с помощью рис. 6.4.

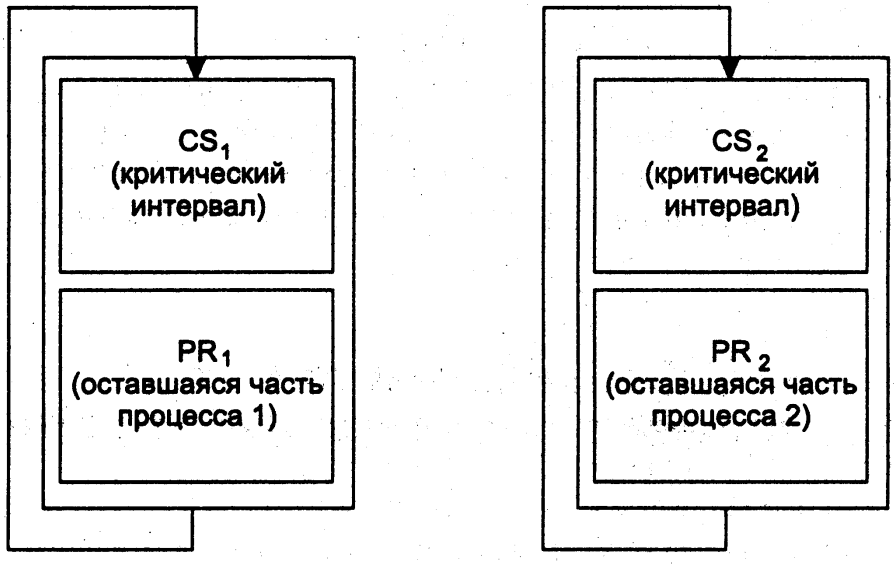

Рис.6.4. Модель взаимодействующих процессов

Проблема кажется легко решаемой, если потребовать, чтобы процессы ПР1 и ПР2 входили в свои критические интервалы попеременно. Для этого одна общая переменная может хранить указатель того, чья очередь войти в критическую секцию. Текст такого решения на языке, близком к Pascal, приведен в листинге 6.1.

Листинг 6.1. Первый вариант попытки реализации взаимного исключения

```
Var перекл : integer;
```

```
Begin перекл := 1:  {при перекл=1 в CS находится процесс ПР1 }
Parbegin
  While true do
        Begin
              while перекл = 2 do begin end;
              CS1:   { Критическая секция процесса ПР1 }
              перекл := 2;
              PR1;   { оставшаяся часть процесса ПР1 }
        End
And
        While true do
        Begin
              while перекл = 1 do begin end;
              CS2;   { Критическая секция процесса ПР2 }
              перекл := 1;
              PR2;   { оставшаяся, часть процесса ПР2}
        End
Parend
End.
```

Здесь и далее конструкция следующего типа

parbegin...S11; S12; ... ; $S1N_1$

and... S21; S22; ... ; $S2N_2$

and... SK1; SK2; ... ; SKN_{1k}

parend.

означает параллельность К описываемых последовательных процессов. Конструкция из операторов S11; S12; ... ; S1N1 выполняется оператор за оператором, о чём свидетельствует наличие точки с запятой между операторами.

Конструкция

while true do

begin S1; S2; SN end

означает, что каждый процесс может выполняться неопределённое время. Конструкция типа begin end означает «пустой» оператор.

Итак, решение, представленное в листинге 6.1, обеспечивает взаимное исключение в работе критических интервалов. Однако если бы часть программы PR1 была намного длиннее, чем программа PR2, или если бы процесс ПР1 был заблокирован в секции PR1, или если бы процессор для ПР2 был с большим быстродействием, то процесс ПР2 вскоре вынужден был бы ждать (и, может быть, чрезвычайно долго) входа в свою критическую секцию CS2, хотя процесс ПР1 и был бы вне CS1. То есть при таком решении один процесс вне своей критической секции может помещать вхождению другого в свою критическую секцию.

Попробуем устранить это блокирование с помощью использования двух общих переключателей, которые используются как флаги для указания, находится ли соответствующий процесс вне своей критической секции.

Пусть с каждым из процессов ПР1 и ПР2 будет связана переменная, по смыслу являющаяся переключателем, которая принимает значение true, когда процесс находится в своем критическом интервале, и false – в противном случае. Прежде чем войти в свой критический интервал, процесс проверяет значение переключателя другого процесса. Если это значение равно true, процессу не разрешается входить в свой критический интервал. В противном случае процесс «включает» свой собственный переключатель и входит в критический интервал. Этот алгоритм взаимного исключения представлен в листинге 6.2.

Данный алгоритм не гарантирует полного выполнения условия нахождения только одного процесса внутри критического интервала. Отсутствие гарантий связано с различными, в общем случае, скоростями развития процессов. Поэтому, например, между проверкой значения переменной перекл2 процессом ПР1 и последующей установкой им значения переменной перекл1 параллельно выполняющийся процесс ПР2 может установить перекл2 в значение true, так как перекл1 ещё не успел установиться в значение true. Отсюда следует, что оба процесса могут войти одновременно в свои критические интервалы.

Листинг 6.2. Второй вариант попытки реализации взаимного исключения

```
Var перекл1,перекл2: boolean;
begin перекл1=false;
        перекл2:=false;
parbegin
  while true do
        begin
              while перекл2 do
                    begin
                    end;
              перекл1:=true;
              CS1 (* Критический Интервал процесса ПР1 *)
              перекл1:=false;
              PR1 (* процесс ПР1 после критического интервала *)
        end
and
        while true do
              begin
                    while перекл1 do
                          begin
                          end;
                    перекл2:=true;
              (* Критический интервал процесса ПР2 *)
                    перекл2:=false;
              (* процесс ПР2 после критического интервала *)
        end
parend
end.
```

Следующий (третий) вариант решения этой задачи (листинг 6.3) усиливает взаимное исключение, так как в процессе ПР1 проверка значения переменной пе-

рекл2 выполняется после установки переменной перекл1 в значение true (аналогично для ПР2).

Листинг 6.3. Третий вариант попытки реализации взаимного исключения

```
var перекл1, перекл2 : boolean;
begin перекл1:=false; перекл2:=false;
parbegin
        ПР1: while true do
                begin
                        перекл1:=true;
                while перекл2 do
                        begin end;
        CS1    {Критический интервал процесса ПР1 }
        перекл1:=false;
        PR1   { ПР1 после критического интервала }
end
and
        ПР2: while true do
                begin
                        перекл2:=true;
                        while перекл1 do
                                begin end;
                        CS2 { Критический интервал процесса ПР2 }
                        перекл2:=false;
                        PR2 { ПР2 после критического интервала }
                end
parend
end.
```

Алгоритм, приведенный в листинге 6.3, также имеет свои недостатки. Действительно, возможна ситуация, когда оба процесса одновременно установят свои переключатели в значение true и войдут в бесконечный цикл. В этом случае будет

нарушено требование отсутствия бесконечного ожидания входа в свой критический интервал. Предположив, что скорости исполнения процессов произвольны, мы получили такую последовательность событий, при которой процессы вообще перестанут нормально выполняться.

Рассмотренные попытки решить проблему критических интервалов иллюстрируют некоторые тонкости, лежащие в основе этой проблемы.

Последний вариант решения задачи взаимного исключения, использующий только блокировку памяти, который мы рассмотрим, – это известный алгоритм, предложенный математиком Деккером.

Алгоритм Деккера

Алгоритм Деккера основан на использовании трёх переменных (листинг 6.4): перекл1, перекл2 и ОЧЕРЕДЬ. Пусть по-прежнему переменная перекл1=true тогда, когда процесс ПР1 хочет войти в свой критический интервал (для ПР2 аналогично), а значение переменной ОЧЕРЕДЬ указывает, чьё сейчас право сделать попытку входа, при условии, что оба процесса хотят выполнить свои критические интервалы.

Листинг 6.4. Алгоритм Деккера

```
label 1, 2;
var    перекл1, перекл2: boolean;
ОЧЕРЕДЬ : Integer;
Begin перекл1=false; перекл2:=false;
ОЧЕРЕДЬ:=1;
parbegin
        while true do
                begin перекл1:=true;
1:                      if перекл2=true then
                                if ОЧЕРЕДЬ=1 then go to 1
                                    else begin перекл1:=false;
                                         while ОЧЕРЕДЬ=2 do
```

```
                                        begin    end
                                end
                        else begin
                                CS1 { Критический интервал ПР1 }
                                ОЧЕРЕДЬ:=2; перекл1:=false
                            end
                        end
    and
            while true do
                    begin перекл2:=1;
    2:              if перекл1=true then
                            if ОЧЕРЕДЬ=2 then go to 2
                                else begin перекл2:=false;
                                    while ОЧЕРЕДЬ=1 do
                                        begin end
                                end
                        else begin
                            CS2 { Критический интервал ПР2 }
                            ОЧЕРЕДЬ:=1; перекл2=false
                        end
                    end
            parend
    end.
```

Если перекл2=true и перекл1=fa1se, то выполняется критический интервал процесса ПР2 независимо от значения переменной ОЧЕРЕДЬ. Аналогично для случая перекл2=fa1se и перекл1=true.

Если же оба процесса хотят выполнить свои критические интервалы, то есть перекл2=true и перекл1=true, то выполняется критический интервал того процесса, на который указывало значение переменной ОЧЕРЕДЬ, независимо от скоростей развития обоих процессов. Использование переменной ОЧЕРЕДЬ совместно с пе-

рекл1 и перекл2 в алгоритме Деккера позволяет гарантированно решать проблему критических интервалов. Переменные перекл1 и перекл2 обеспечивают то, что взаимное выполнение не может иметь места; переменная ОЧЕРЕДЬ гарантирует от взаимного блокирования. Взаимное блокирование невозможно, так как переменная не изменяет своего значения во время выполнения программы принятия решения о том, кому же сейчас проходить свой критический интервал.

Тем не менее, реализация критических интервалов на основе описанного алгоритма практически не используется из-за чрезмерной сложности, особенно в случаях, когда алгоритм Деккера обобщается с двух до *N* процессов.

Синхронизация процессов посредством операции «ПРОВЕРКА И УСТАНОВКА»

Операция «ПРОВЕРКА И УСТАНОВКА» является, как и блокировка памяти, одним из аппаратных средств решения задачи критического интервала. Данная операция реализована на многих компьютерах. Так, в знаменитой IBM 360 (370) эта команда называлась TS (test and set). Команда TS является двухадресной (двухоперандной). Её действие заключается в том, что процессор присваивает значение второго операнда первому, после чего второму операнду присваивается значение, равное единице. Команда TS является неделимой операцией, то есть между ее началом и концом не могут выполняться никакие другие команды.

Чтобы использовать команду TS для решения проблемы критического интервала, свяжем с ней переменную common, которая будет общей для всех процессов, использующих некоторый критический ресурс. Данная переменная будет принимать единичное значение, если какой-либо из взаимодействующих процессов находится в своем критическом интервале. С каждым процессом связана своя локальная переменная, которая принимает значение, равное единице, если данный процесс хочет войти в свой критический интервал. Операция TS будет присваивать значение common локальной переменной и устанавливать common в единицу. Программа решения проблемы критического интервала на примере двух параллельных процессов приведена в листинге 6.5.

Листинг 6.5. Взаимное исключение с помощью операции «ПРОВЕРКА И УСТАНОВКА»

```
var common, local1, local2 : integer;
begin
 common:=0;
 parbegin
ПР1: while true do
 begin
        local1:=1;
        white local1=1 do TS(local1, common);
        CS1; { Критический интервал процесса ПР1 }
        common:=0;
        PR1; { ПР1 после критического интервала }
 end
and
ПР2: while true do
        begin
                local2:1;
                while local2=1 do TS(local2, common);
                CS2; { Критический интервал процесса ПР2 }
                common:=0;
                PR2; { ПР2 после критического интервала }
        end
 parend
end.
```

Предположим, что первым захочет войти в свой критический интервал процесс ПР1. В этом случае значение local1 сначала установится в единицу, а после цикла проверки с помощью команды TS(local1, common) – в ноль. При этом значение common станет равным единице. Процесс ПР1 войдет в свой критический интервал. После выполнения этого критического интервала common примет значение,

равное нулю, что даст возможность второму процессу ПР2 войти в свой критический интервал.

Безусловно, мы предполагаем, что в компьютере предусмотрена блокировка памяти, то есть операция common:=0 неделима. Команда «ПРОВЕРКА И УСТАНОВКА» значительно упрощает решение проблемы критических интервалов. Главное свойство этой команды – её неделимость.

Основной недостаток использования операций типа «ПРОВЕРКА И УСТАНОВКА» состоит в следующем: находясь в цикле проверки переменной common, процессы впустую потребляют время центрального процессора и другие ресурсы. Действительно, если предположить, что произошло прерывание процесса ПР1 во время выполнения своего критического интервала в соответствии с некоторой дисциплиной обслуживания и начал выполняться процесс ПР2, то он войдет в цикл проверки, впустую тратя процессорное время. В этом случае до тех пор, пока диспетчер супервизора не поставит на выполнение процесс ПР1 и не даст ему закончиться, процесс ПР2 не сможет войти в свой критический интервал.

В микропроцессорах i80386 и старше, с которыми мы теперь сталкиваемся постоянно, есть специальные команды: BTC, BTS, BTR, которые как раз и являются вариантами реализации команды типа «ПРОВЕРКА И УСТАНОВКА». Рассмотрим одну из них – BTS.

Команда BTS (bit test and reset – проверка бита и установка) является двухадресной [48]. По этой команде процессор сохраняет значение бита из первого операнда со смещением, указанным вторым операндом, во флаге CF[1] регистра флагов, а затем устанавливает указанный бит в 1. Значение индекса выбираемого бита может быть представлено постоянным непосредственным значением в команде BTS или значением в общем регистре. В этой команде используется только 8-битное непосредственное значение. Значение этого операнда берется по модулю 32, таким образом, смещение битов находится в диапазоне от 0 до 31. Это позволяет выбирать любой бит внутри регистра. Для битовых строк в памяти это поле непосредственного значения даёт только смещение внутри слова или двойного слова.

[1] *CF* (carry flag) – флаг переноса. Располагается в слове состояния программы. Для процессоров i80x86 этот регистр – управляющий регистр CR0.

С учётом изложенного можно привести фрагмент текста, в котором используется данная команда для решения проблемы взаимного исключения (листинг 6.6).

Однако здесь следует заметить, что некоторые ассемблеры поддерживают значения битовых смещений больше 31, используя поле непосредственного значения в комбинации с полем смещения операнда в памяти. В этом случае ассемблером младшие 3 или 5 битов (3 – для 16-битных операндов, 5 – для 32-битных операндов) смещения бита (второй операнд команды) сохраняются в поле непосредственного операнда, а старшие биты сдвигаются и комбинируются с полем смещения. Процессор же игнорирует ненулевые значения старших битов поля второго операнда [48]. При доступе к памяти процессор обращается к четырем байтам, начинающимся по полученному следующим образом адресу

Effective Address + (4 * (BitOffset DIV 32))

для размера операнда 32 бита, или к двум байтам, начинающимся по адресу

Effective Address + (2 * (BitOffset DIV 16))

для 16-битного размера операнда. Такое обращение происходит, даже если необходим доступ только к одиночному байту. Поэтому избегайте ссылок к областям памяти, близким к «пустым» адресным пространствам. В частности, избегайте ссылок на распределённые в памяти регистры ввода/вывода. Вместо этого используйте команду MOV для загрузки и сохранения значений по таким адресам и регистровую форму команды BTS для работы с данными.

Листинг 6.6. Фрагмент программы на ассемблере с критическим интервалом

```
L: BTC  M,1
   JC  L
      .          }
      .          } критическая секция
      .          }
   AND M, 0B
      .
      .
```

Несмотря на то, что и алгоритм Деккера, основанный только на блокировке памяти, и операция «ПРОВЕРКА И УСТАНОВКА» пригодны для реализации взаимного исключения, оба эти приёма очень неэффективны. Всякий раз, когда один

из процессов выполняет свой критический интервал, любой другой процесс, который пытается войти в свою критическую секцию, попадает в цикл проверки соответствующих переменных-флагов, регламентирующих доступ к критическим переменным. При таком неопределенном пребывании в цикле, которое называется *активным ожиданием,* напрасно расходуется процессорное время, поскольку процесс имеет доступ к тем общим переменным, которые и определяют возможность или невозможность входа в критическую секцию. При этом процесс занимает ценное время центрального процессора, на самом деле ничего реально не выполняя. Как результат, мы имеем общее замедление вычислительной системы процессами, которые реально не выполняют никакой полезной работы.

До тех пор, пока процесс, занимающий в данный момент критический ресурс, не решит его уступить, все другие процессы, ожидающие этого ресурса, могли бы вообще не конкурировать за процессорное время. Для этого их нужно перевести в состояние ожидания (заблокировать их выполнение). Когда вход в критическую секцию снова будет свободен, можно будет опять перевести заблокированный процесс в состояние готовности к выполнению и дать ему возможность получить процессорное время. Самый простой способ предоставить процессорное время только одному вычислительному процессу – это отключить систему прерываний, поскольку тогда никакое внешнее событие не сможет прервать выполняющийся процесс. Однако это, как мы уже знаем, приведет к тому, что система не сможет реагировать на внешние события.

Вместо того чтобы связывать с каждым процессом свою собственную переменную, как это было в рассмотренных нами решениях, можно со всем множеством конкурирующих критических секций связать одну переменную, которую и рассматривать как некоторый ключ. Вначале доступ к критической секции открыт. Однако перед входом в свой критический интервал процесс забирает «ключ» и тем самым блокирует другие процессы. Покидая критическую секцию, процесс открывает доступ, возвращая «ключ» на место. Если процесс, который хочет войти в свою критическую секцию, обнаруживает, что ключ «отсутствует», то он должен быть переведён в состояние блокирования до тех пор, пока процесс, имеющий

ключ, не вернёт его. Таким образом, каждый процесс, входящий в критический интервал, должен вначале проверить, доступен ли ключ, и если это так, то сделать его недоступным для других процессов. Причем самым главным является то, что эти два действия должны быть неделимыми, чтобы два или более процессов не могли одновременно получить доступ к ключу. Более того, проверку того, можно ли войти в критический интервал, лучше всего выполнять не самим конкурирующим процессам, так как это приводит к активному ожиданию, а возложить эту функцию на операционную систему. Таким образом, мы подошли к одному из самых главных механизмов решения проблемы взаимного исключения – семафорам Дейкстры.

Семафорные примитивы Дейкстры

Понятие семафорных механизмов было введено Дейкстрой [27]. *Семафор* – переменная специального типа, которая доступна параллельным процессам для проведения над ней только двух операций: «закрытия» и «открытия», названных соответственно P- и V-операциями. Эти операции являются примитивами относительно семафора, который указывается в качестве параметра операций. Здесь семафор выполняет роль вспомогательного критического ресурса, так как операции P и V неделимы при своём выполнении и взаимно исключают друг друга.

Семафорный механизм работает по схеме, в которой сначала исследуется состояние критического ресурса, идентифицируемое значением семафора, а затем уже осуществляется допуск к критическому ресурсу или отказ от него на некоторое время. При отказе доступа к критическому ресурсу используется режим «пассивного ожидания». Поэтому в состав механизма включаются средства формирования и обслуживания очереди ожидающих процессов. Эти средства реализуются супервизором операционной системы. Необходимо отметить, что в силу взаимного исключения примитивов попытка в различных параллельных процессах одновременно выполнить примитив над одним и тем же семафором приведет к тому, что она будет успешной только для одного процесса. Все остальные процессы будут взаимно исключены на время выполнения примитива. Основным достоинством использования семафорных операций является отсутствие состояния «активного ожидания»,

что может существенно повысить эффективность работы мультипрограммной вычислительной системы.

В настоящее время на практике используется много различных видов семафорных механизмов [75]. Варьируемыми параметрами, которые отличают различные виды примитивов, являются начальное значение и диапазон изменения значений семафора, логика действий семафорных операций, количество семафоров, доступных для обработки при исполнении отдельного примитива.

Обобщенный смысл примитива P(S)[1] состоит в проверке текущего значения семафора S, и если оно не меньше нуля, то осуществляется переход к следующей за примитивом операции. В противном случае процесс снимается на некоторое время с выполнения и переводится в состояние «пассивного ожидания». Находясь в списке заблокированных, ожидающий процесс не проверяет семафор непрерывно, как в случае активного ожидания. Вместо него на процессоре может исполняться другой процесс, который реально совершает полезную работу.

Операция V(S)[2] связана с увеличением значения семафора на единицу и переводом одного или нескольких процессов в состояние готовности к центральному процессору.

Отметим еще раз, что операции P и V выполняются операционной системой в ответ на запрос, выданный некоторым процессом и содержащий имя семафора в качестве параметра.

Рассмотрим первый вариант алгоритма работы семафорных операций, который представлен в листинге 6.7.

Допустимыми значениями семафоров являются только целые числа. Двоичным семафором будем называть семафор, максимально возможное значение которого будет равно единице. В противном случае семафоры называют N-ичными. Есть реализации, в которых семафорные переменные не могут быть отрицательными, а есть и такие, где отрицательное значение указывает на длину очереди процессов, стоящих в состоянии ожидания открытия семафора.

[1] P – от голландского Proberen – проверить.
[2] V – от голландского Verhogen – увеличить.

Листинг 6.7. Вариант реализации семафорных примитивов

```
P(S): S:=S-1;
        if S<0 then
  { остановить процесс и поместить его в очередь ожидания к семафору S }
V(s): if S<0 then
 {поместить один из ожидающих процессов очереди семафора S в очередь
  готовности};
        S:=S+1
```

Заметим, что для работы с семафорными переменными необходимо еще иметь операцию инициализации самого семафора, то есть операцию задания ему начального значения. Обычно эту операцию называют InitSem; как правило, она имеет два параметра – имя семафорной переменной и её начальное значение; следовательно, обращение к ней имеет вид:

```
lmtSem (Имя_семафора, Начальное_значение_семафора);
```

Попытаемся на нашем примере двух конкурирующих процессов ПР1 и ПР2 проанализировать использование данных семафорных примитивов для решения проблемы критического интервала. Программа, иллюстрирующая это решение, представлена в листинге 6.8.

Листинг 6.8. Взаимное исключение с помощью семафорных операций

```
var S:semafor;
begin InitSem(S,1);
parbegin
  ПР1: while true do
                begin
                      P(S);
                      CS1 { Критический интервал процесса ПР1 }
                      V(S)
                end
and
  ПР2: while true do
```

```
        begin
            P(S);
            CS2 { Критический интервал процесса ПР2 }
            V(S)
        end
parend
end.
```

Семафор S имеет начальное значение, равное 1. Если процессы ПР1 и ПР2 попытаются одновременно выполнить примитив P(S), то это удастся успешно сделать только одному из них. Предположим, это сделал процесс ПР2, тогда он закрывает семафор S, после чего выполняется его критический интервал. Процесс ПР1 в рассматриваемой ситуации будет заблокирован на семафоре S. Тем самым будет гарантировано взаимное исключение.

После выполнения примитива V(S) процессом ПР2 семафор S открывается, указывая на возможность захвата каким-либо процессом освободившегося критического ресурса. При этом производится перевод процесса ПР1 из заблокированного состояния в состояние готовности.

На уровне реализации возможно одно из двух решений в отношении процессов, которые переводятся из очереди ожидания в очередь готовности при выполнении примитива V:

♦ процесс при его активизации (выборка из очереди готовности) вновь пытается выполнить примитив P, считая предыдущую попытку неуспешной;

♦ процесс при помещении его и очередь готовности отмечается как успешно выполнивший примитив P. Тогда при его активизации управление будет передано не на повторное выполнение примитива P, а на команду, следующую за ним. Рассмотрим первый способ реализации. Пусть процесс ПР2 в некоторый момент времени выполняет операцию P(S). Тогда семафор S становится равным нулю. Пусть далее процесс ПР1 пытается выполнить операцию P(S). Процесс ПР1 в этом случае «засыпает» на семафоре S, так как значение семафора S равнялось нулю, а теперь станет равным (-1). После выполнения критического интервала процесс ПР2

выполняет операцию V(S), при этом значение семафора S становится равным нулю, а процесс ПР1 переводится в очередь готовности. Пусть через некоторое время процесс ПР1 будет активизирован, то есть выведен из состояния ожидания, и сможет продолжить своё исполнение. Он повторно пытается выполнить операцию P(S), однако это ему не удается, так как S=0. Процесс ПР1 «засыпает» на семафоре, а его значение становится равным (-1). Если через некоторое время процесс ПР2 попытается выполнить P(S), то он тоже «уснет». Таким образом, возникнет так называемая тупиковая ситуация, так как «будить» процессы ПР1 и ПР2 некому.

При втором способе реализации тупиковой ситуации не будет. Действительно, пусть всё происходит также до момента окончания исполнения процессом ПР2 примитива V(S). Пусть примитив V(S) выполнен и S=0. Через некоторое время процесс ПР1 активизировался. Согласно данному способу реализации, он сразу же попадёт в свой критический интервал. При этом никакой другой процесс не попадёт в свой критический интервал, так как семафор остался закрытым. После исполнения своего критического интервала процесс ПР1 выполнит V(S). Если за время выполнения критического интервала процесса ПР1 процесс ПР2 не делал попыток выполнить операцию P(S), семафор S установится в единицу. В противном случае значение семафора будет не больше нуля. Но в любом варианте после завершения операции V(S) процессом ПР1 доступ к критическому ресурсу со стороны процесса ПР2 будет разрешен.

Заметим, что возникновение тупиков возможно в случае несогласованного выбора механизма реактивации процессов из очереди, с одной стороны, и выбора алгоритмов семафорных операций – с другой.

Возможен другой алгоритм работы семафорных операций:

```
P(S): if S>=1 then S:=S-1
        else WAIT(S)
{остановить процесс и поместить в очередь ожидания к семафору S}
V(S): if S<0 then RELEASE(S)
{поместить один из ожидающих процессов очереди семафора S
в очередь готовности};
```

S:-S+1.

Здесь вызов WAIT(S) означает, что супервизор ОС должен перевести задачу в состояние ожидания, причём очередь процессов связана с семафором S. Вызов RELEASE(S) означает обращение к диспетчеру задач с просьбой перевести первый из процессов, стоящих в очереди S, в состояние готовности к исполнению.

Использование семафорных операций, выполненных подобным образом, позволяет решать проблему критических интервалов на основе первого способа реализации без вероятности возникновения тупиков. Действительно, пусть ПР2 в некоторый момент времени выполнит операцию P(S). Тогда семафор S становится равным нулю. Пусть далее процесс ПР1 пытается выполнить операцию P(S). Процесс ПР1 в этом случае «заснет» на семафоре S, так как S=0, причём значение S не изменится. После выполнения своего критического интервала процесс ПР2 выполнит операцию V(S), при этом значение семафора S станет равный единице, а процесс ПР1 переведётся в очередь готовности. Если через некоторое время процесс ПР1 будет активизирован, он успешно выполнит P(S) и войдёт в свой критический интервал.

В однопроцессорной вычислительной системе неделимость P- и V-операций можно обеспечить с помощью простого запрета прерываний. Сам же семафор S можно реализовать в виде записи с двумя полями (листинг 6.9). В одном поле хранится целое значение S, во втором – указатель на список процессов, заблокированных на семафоре S.

Листинг 6.9. Реализация P- и V-операций для однопроцессорной системы

```
type Semaphore = record
        счетчик    :integer;
        указатель :pointer;
        end;
var S :Semaphore;

procedure P ( var S : Semaphore);
begin ЗАПРЕТИТЬ_ПРЕРЫВАНИЯ;
```

```
        S.счетчик:= S-счетчик-1;
        if S.счетчик < 0 then
              WAIT(S); {ВСТАВИТЬ ОБРАТИВШИЙСЯ ПРОЦЕСС В
                        СПИСОК ПО S.указатель и УСТАНОВИТЬ НА
                        ПРОЦЕССОР ГОТОВЫЙ К ВЫПОЛНЕНИЮ
                        ПРОЦЕСС }
        РАЗРЕШИТЬ__ПРЕРЫВАНИЯ
end;

procedure V ( var S : Semaphore);
begin ЗАПРЕТИТЬ ПРЕРЫВАНИЯ;
        S.счетчик:=S.счетчик+1;
        if S.счетчик <= 0 then
              RELEASE (S);  { ДЕБЛОКИРОВАТЬ ПЕРВЫЙ ПРОЦЕСС ИЗ
                              СПИСКА ПО S.указатель }
        РАЗРЕШИТЬ_ПРЕРЫВАНИЯ
end;

procedure InitSem (var S : Semaphore);
begin
        S.счетчик:=1;
        S.указатель:=nil;
end;
```

Реализация семафоров в мультипроцессорных системах более сложна, чем в однопроцессорных. Одновременный доступ к семафору S двух процессов, выполняющихся на однопроцессорной вычислительной системе, предотвращается запретом прерываний. Однако этот механизм не подходит для мультипроцессорных систем, так как он не препятствует двум или более процессам одновременно обращаться к одному семафору. В силу того, что такой доступ должен реализовываться как критический интервал, необходимо дополнительное аппаратное взаимное ис-

ключение доступа для различных процессоров. Одним из решений является использование уже знакомых нам неделимых команд типа «ПРОВЕРКА И УСТАНОВКА» (TS). Двухкомпонентный семафор в этом случае расширяется включением третьего компонента – логического признака ВзаимоИскл (листинг 6.10).

Листинг 6.10. Реализация P- и V-операций для мультипроцессорной системы

```
type Semaphore = record
                    счетчик : integer;
                    указатель : pointer;
                    взаимоискл : boolean;
          end;
var S : Semaphore;

procedure InitSem (var S : Semaphore);
begin
With S do
          begin
                    счетчик:=1;
                    указатель :=nil;
                    взаимоискл:=true;
          end;
end;

procedure P ( var S : Semaphore);
var разрешено : boolean;
begin
          ЗАПРЕТИТЬ_ПРЕРЫВАНИЯ;
          repeat TS(разрешено, S.взаимоискл) until разрешено;
          S. счетчик:= счетчик-1;
          if S.счетчик < 0 then WAIT(S); { ВСТАВИТЬ ОБРАТИВШИЙСЯ
                                        ПРОЦЕСС В СПИСОК
```

```
                                        ПО S.указатель И УСТАНОВИТЬ
                                        НА CPU ГОТОВЫЙ ПРОЦЕСС }
        S.взаимоискл:=true;
        РАЗРЕШИТЬ_ПРЕРЫВАНИЯ
end;

procedure V ( var S : Semaphore );
var разрешено : boolean;
begin
        ЗАПРЕТИТЬ_ПРЕРЫВАНИЯ;
        repeat TS(разрешено, S. взаимоискл) until разрешено;
        S.счетчик:=S.счетчик+1;
        if S.счетчик < 0 then RELEASE(S); { ДЕБЛОКИРОВАТЬ ПЕРВЫЙ
                                            ПРОЦЕСС ИЗ СПИСКА ПО
                                            S.указатель }
        S.взаимоискл:=true;
        РАЗРЕшИТЬ_ПРЕРЫВАНИЯ;
end;
```

Обратите внимание, что в данном тексте команда типа «проверка и установка» – TS(разрешено, S.взаимоискл) – работает не с целочисленными, а с булевыми значениями. Практически это ничего не меняет, ибо текст программы и её машинная (двоичная) реализация – это всё-таки разные вещи.

Мьютексы

Одним из вариантов семафорных механизмов для организации взаимного исключения являются так называемые *мьютексы* (mutex). Термин mutex произошёл от английского словосочетания mutual exclusion semaphore, что дословно и переводится как семафор взаимного исключения. Мьютексы реализованы во многих **ОС,** их основное назначение – организация взаимного исключения для задач (потоков) из одного и того же или из разных процессов. Мьютексы – это простейшие двоич-

ные семафоры, которые могут находиться в одном из двух состояний – отмеченном или неотмеченном (открыт и закрыт соответственно). Когда какая-либо задача, принадлежащая любому процессу, становится владельцем объекта mutex, последний переводится в неотмеченное состояние. Если задача освобождает мьютекс, его состояние становится отмеченным.

Организация последовательного (а не параллельного) доступа к ресурсам с использованием мьютексов становится несложной, поскольку в каждый конкретный момент только одна задача может владеть этим объектом. Для того чтобы объект mutex стал доступен задачам (потокам), принадлежащим разным процессам, при создании ему необходимо присвоить имя. Потом это имя нужно передать «по наследству» задачам, которые должны его использовать для взаимодействия. Для этого вводятся специальные системные вызовы (CreateMutex), в которых указываются начальное значение мыотекса, его имя и, возможно, атрибуты защиты. Если начальное значение мьютекса равно true, то считается, что задача, создающая этот объект, будет им сразу владеть. Можно указать в качестве начального значение false – в этом случае мьютекс не принадлежит ни одной из задач и только специальным обращением к нему можно изменить его состояние.

Для работы с мьютексом имеется несколько функций. Помимо уже упомянутой функции создания такого объекта (CreateMutex), есть функции открытия (OpenMutex), ожидания событий (WaitForSingleObject и WaitForMultipleObjects) и, наконец, освобождение этого объекта (ReleaseMutex).

Конкретные обращения к этим функциям и перечни передаваемых и получаемых параметров нужно смотреть в документации на соответствующую ОС.

Использование семафоров при проектировании взаимодействующих вычислительных процессов

Семафорные примитивы чрезвычайно широко используются при проектировании разнообразных вычислительных процессов. При этом некоторые задачи являются настолько «типичными», что их детальное рассмотрение уже стало классическим в соответствующих учебных пособиях. Не будем делать исключений и мы.

Задача «поставщик – потребитель»

Решение задачи «поставщик – потребитель» является характерным примером использования семафорных операций. Содержательная постановка этой задачи уже была нами описана в первом разделе данной главы. Разделяемыми переменными здесь являются счётчики свободных и занятых буферов, которые должны быть защищены со стороны обоих процессов, то есть действия по посылке и получению сообщений должны быть синхронизированы.

Использование семафоров для решения данной задачи приведено в листинге 6.11.

Листинг 6.11. Решение задачи «поставщик – потребитель»

```
var S_свободно, S_заполнено, S_взаимоискл : semaphore;
begin
        initSem(S_свободно,N);
        initSem(S_заполнено,0);
        initSem(S_взаимоискл,1);
parbegin
        ПОСТАВЩИК:   while true do
                                begin
                                        …{ приготовить сообщение }
                                        P(S_свободно);
                                        P(S_взаимоискл);
                                        …{послать сообщение }
                                        V(S_заполнено);
                                        V(S_взаимоискл);
                                end
and
        ПОТРЕБИТЕЛЬ: while true do
                                begin
                                        P(S_заполнено);
                                        P(S_взаимоискл);
```

```
                    …{ получить сообщение }
                    V(S_свободно);
                    V(S_взаимоискл);
                    …{ обработать сообщение }
                end
    parend
    end.
```

Здесь переменные S_свободно, S_заполнено являются числовыми семафорами, S_взаимоискл – двоичный семафор. S_свободно имеет начальное значение, равное N, где N – количество буферов, с помощью которых процессы связываются. Предполагается, что в начальный момент количество свободных буферов равно N; соответственно, количество занятых равно нулю. Двоичный семафор S_взаимоискл гарантирует, что в каждый момент только один процесс сможет работать с критическим ресурсом, выполняя свой критический интервал. Семафоры S_свободно и S_заполнено используются как счётчики свободных и заполненных буферов. Действительно, перед посылкой сообщения «поставщик» уменьшает значение S_свободно на единицу в результате выполнения P(S_свободно), а после посылки сообщения увеличивает значение S_заполнено на единицу в результате выполнения V(S_заполнено). Аналогично, перед получением сообщения «потребитель» уменьшает значение S_заполнено в результате выполнения P(S_заполнено), а после получения сообщения увеличивает значение S_свободно в результате выполнения V(S_свободно). Семафоры S_заполнено, S_свободно могут также использоваться для блокировки соответствующих процессов. Если пул буферов оказался пустым и к нему первым обратится процесс «потребитель», он заблокируется на семафоре S_заполнено в результате выполнения P(S_заполнено). Если пул буферов заполнен и к нему в этот момент обратится процесс «поставщик», то он будет заблокирован на семафоре S_свободно в результате выполнения P(S_свободно).

В решении задачи о «поставщике» и «потребителе» общие семафоры применены для учёта свободных и заполненных буферов. Их можно также применить и для распределения иных ресурсов.

Пример простейшей синхронизации взаимодействующих процессов

Можно использовать семафорные операции для решения таких задач, в которых успешное завершение одного процесса связано с ожиданием завершения другого. Предположим что существуют два процесса ПР1 и ПР2. Необходимо, чтобы процесс ПР1 запускал процесс ПР2 с ожиданием его выполнения, то есть ПР1 не продолжал бы выполняться до тех пор, пока процесс ПР2 до конца не выполнит свою работу. Программа, реализующая такое взаимодействие, представлена в листинге 6.12.

Листинг 6.12. Пример синхронизации процессов

```
var S : Semaphore;
begin
        initSem(S,0):
        ПР1:   begin
        ПР11;     { первая часть ПР1 }
        ON ( ПР2 );   { поставить на выполнение ПР2 }
        P(S);
        ПР12;      { оставшаяся часть ПР1 }
        STOP
end;
ПР2:    begin
                ПР2;       { вся работа программы ПР2 }
                V(S);
                STOP
        end
end.
```

Начальное значение семафора S равно нулю. Если процесс ПР1 начал выполняться первым, то через некоторое время он поставит на выполнение процесс ПР2, после чего выполнит операцию P(S) и «заснёт» на семафоре, перейдя в состояние пассивного ожидания. Процесс ПР2, осуществив все необходимые действия, выполнит примитив V(S) и откроет семафор, после чего процесс ПР1 будет готов к дальнейшему выполнению.

Решение задачи «читатели – писатели»

Другой важной и часто встречающейся задачей, решение которой также требует синхронизации, является задача «читатели – писатели». Эта задача имеет много вариантов. Наиболее характерная область использования этой задачи – при построении систем управления файлами. Два класса процессов имеют доступ к некоторому ресурсу (области памяти, файлам). «Читатели» – это процессы, которые могут параллельно считывать информацию из некоторой общей области памяти, являющейся критическим ресурсом. «Писатели» – это процессы, записывающие информацию в эту область памяти, исключая при этом и друг друга, и процессы «читатели». Имеются различные варианты взаимодействия между «писателями» и «читателями». Наиболее широко распространены следующие условия.

Устанавливается приоритет в использование критического ресурса процессам «читатели». Это означает, что если хотя бы один «читатель» пользуется ресурсом, то он закрыт для использования всем «писателям» и доступен для использования всем «читателям». Во втором варианте, наоборот, больший приоритет у процессов «писатели». При появлении запроса от «писателя» необходимо закрыть дальнейший доступ всем тем процессам «читателям», которые выдадут запрос на критический ресурс после него.

Другим типичным примером для задачи «читатели – писатели» помимо систем управления файлами может служить система автоматизированной продажи билетов. Процессы «читатели» обеспечивают нас справочной информацией о наличии свободных билетов на тот или иной рейс. Процессы «писатели» запускаются с

пульта кассира, когда он оформляет для нас тот или иной билет. Имеется большое количество как «читателей», так и «писателей».

Пример программы, реализующей решение данной задачи в первой постановке, представлен в листинге 6.13. Процессы «читатели» и «писатели» описаны в виде соответствующих процедур.

Листинг 6.13. Решение задачи «читатели–писатели» с приоритетом в доступе к критическому ресурсу «читателей»

```
Var R, W : semaphore;
        NR : integer;
procedure ЧИТАТЕЛЬ;
begin
        P(R);
        Inc(NR);      { NR:=NR +1 }
        if NR = 1 then P(W);
        V(R);
        Read_Data:      { критический интервал }
        P(R);
        Dec(NR);
        if NR = 0 then V(W);
        V(R)
end;
procedure ПИСАТЕЛЬ;
begin
        P(W);
        Write_Data;      { критический интервал }
        V(W)
end
begin
        NR:=0;
        initSem(S,1); InitSem(W,1);
        parbegin
                while true do ЧИТАТЕЛЬ
        and
                while true do ЧИТАТЕЛЬ
        and
                .
                .
                .
```

```
        while true do ЧИТАТЕЛЬ
    and
        while true do ПИСАТЕЛЬ
    and
        while true do ПИСАТЕЛЬ
    and
        .
        .
        .
        while true do ПИСАТЕЛЬ
        parend
end.
```

При решении данной задачи используются два семафора R и W и переменная NR, предназначенная для подсчёта текущего числа процессов типа «читатели», находящихся в критическом интервале. Доступ к разделяемой области памяти осуществляется через семафор W. Семафор R используется для взаимоисключения процессов типа «читатели».

Если критический ресурс не используется, то первый появившийся процесс при входе в критический интервал выполнит операцию P(W) и закроет семафор. Если процесс является «читателем», то переменная NR будет увеличена на единицу и последующие «читатели» будут обращаться к ресурсу, не проверяя значение семафора W, что обеспечивает параллельность их доступа к памяти. Последний «читатель», покидающий критический интервал, является единственным, кто выполнит операцию V(W) и откроет семафор W. Семафор R предохраняет от некорректного изменения значения NR, а также от выполнения «читателями» операций P(W) и V(W). Если в критическом интервале находится «писатель», то на семафоре W может быть заблокирован только один «читатель», все остальные будут блокироваться на семафоре R. Другие «писатели» блокируются на семафоре W.

Когда «писатель» выполняет операцию V(W), неясно, какого типа процесс войдёт в критический интервал. Чтобы гарантировать получение процессами «читателями» наиболее свежей информации, необходимо при постановке в очередь готовности использовать дисциплину обслуживания, учитывающую более высокий приоритет «писателей». Однако этого оказывается недостаточно, ибо если в крити-

ческом интервале продолжает находиться, по крайней мере, один «читатель», то он не даст обновить данные, но и не воспрепятствует вновь приходящим процессам «читателям» войти в свою критическую секцию. Необходим дополнительный семафор. Пример правильного решения этой задачи приведен в листинге 6.14.

Листинг 6.14. Решение задачи «читатели – писатели» с приоритетом в доступе к критическому ресурсу первых с дополнительным семафором

```
var S, W, R : semaphore;
        NR : =integer:
procedure ЧИТАТЕЛЬ;
begin
        P(S);  P(R);
        Inc(NR);
        if NR =1 then P(W);
        V(S); V(R);
        Read_Data;    { критический интервал }
        P(R);
        Dec(NR);
        if NR = 0 then V(W);
        V(R)
end;

procedure ПИСАТЕЛЬ;
begin
        P(S);  P(W);
        Write_Data;      { критический интервал }
        V(S); V(W)
end
begin
        NR:=0;
        InitSem(S, 1);  InitSem(W, 1); InitSem(R, 1);
        parbegin
                while true do ЧИТАТЕЛЬ
 and
                while true do ЧИТАТЕЛЬ
 and
                .
                .
```

```
            .
            while true do ЧИТАТЕЛЬ
    and
            while true do ПИСАТЕЛЬ
    and
            while true do ПИСАТЕЛЬ
    and
            .
            .
            .
            while true do ПИСАТЕЛЬ
        parend
end.
```

Как можно заметить, семафор S блокирует приход новых «читателей», если появился хотя бы один процесс «писатель». Обратите внимание, что в процедуре «читатель» использование семафора S имеет место только при входе в критический интервал. После выполнения чтения уже категорически нельзя использовать этот семафор, ибо он тут же заблокирует первого же «читателя», если хотя бы один «писатель» захотел войти в свою критическую секцию. И получится так называемая тупиковая ситуация, ибо «писатель» не сможет войти в критическую секцию, поскольку в ней уже находится процесс «читатель». А «читатель» не сможет покинуть критическую секцию, потому что процесс «писатель» желает войти в свой критический интервал.

Обычно программы, решающие проблему «читатели – писатели», используют как семафоры, так и мониторные схемы с взаимным исключением, то есть такие, которые блокируют доступ к критическим ресурсам для всех остальных процессов, если один из них модифицирует значения общих переменных. Взаимное исключение требует, чтобы «писатель» ждал завершения всех текущих операций чтения. При условии, что «писатель» имеет более высокий приоритет, чем «читатель», такое ожидание в ряде случаев весьма нежелательно. Кроме того, реализация принципа исключения в многопроцессорных системах может вызвать определённую избыточность. Поэтому ниже приводится схема, применяемая иногда для решения задачи «читатели – писатели», которая в случае одного «писателя» допускает одновременное выполнение операций чтения и записи (листинг 6.15). После чтения

данных процесс «читатель» проверяет, мог ли он получить неправильное значение, некорректные данные (вследствие того, что параллельно с ним процесс «писатель» мог их изменить), и если обнаруживает, что это именно так, то операция, чтения повторяется.

Листинг 6,15. Синхронизация процессов «читатели» и «писатель» без взаимного исключения

```
Var VI, V2 : integer;

Procedure ПИСАТЕЛЬ;
begin
        Inc(V1);
        Write_Data;
        V2:=V1
end;

Procedure ЧИТАТЕЛЬ;
Var V: integer
begin
        repeat V:= V2;
                Read_Data
        until V1 = V
end;
begin
        V1:= 0;
        V2:= 0;
        parbegin
                while true do ЧИТАТЕЛЬ
        and
                while true do ЧИТАТЕЛЬ
        and
                .
                .
                .
                while true do ЧИТАТЕЛЬ
        and
                while true do ПИСАТЕЛЬ
        parend
```

end.

Этот алгоритм использует для данных два номера версий, которым соответствуют переменные V1 и V2. Перед записью порции новых данных процесс «писатель» увеличивает на 1 значение переменной V1, а после записи – переменную V2. «Читатель» обращается к V2 перед чтением данных, а к V1 – после. Если при этом V1 и V2 равны, то, очевидно, что получена правильная версия данных. Если же данные обновлялись за время чтения, то операция повторяется. Данный алгоритм может быть использован в случае, если нежелательно заставлять процесс «писатель» ждать, пока «читатели» закончат операцию чтения, или если вероятность повторения операции чтения достаточно мала и обусловленное повторными операциями снижение эффективности системы меньше потерь, связанных с избыточностью решения при использовании взаимного исключения. Однако необходимо иметь в виду ненулевую вероятность зацикливания чтения при высокой интенсивности операций записи. Наконец, если само чтение представляет собой достаточно длительную операцию, то оператор V1:=V2 для процесса «читатель» может быть заменён на оператор

repeat V:= V2 until V1 = V

Это предотвратит выполнение «читателем» операции чтение, если «писатель» уже начал запись.

Мониторы Хоара

Анализ рассмотренных задач показывает, что, несмотря на очевидные достоинства (простота, независимость от количества процессов, отсутствие «активного ожидания»), семафорные механизмы имеют и ряд недостатков. Семафорные механизмы являются слишком примитивными, так как семафор не указывает непосредственно на синхронизирующее условие, с которым он связан, или на критический ресурс. Поэтому при построении сложных схем синхронизации алгоритмы решения задач порой получаются весьма непростыми, ненаглядными и затруднительными для доказательства их правильности.

Необходимо иметь понятные, очевидные решения, которые позволят прикладным программистам без лишних усилий, связанных с доказательством правильности алгоритмов и отслеживанием большого числа взаимосвязанных объектов,

создавать параллельные взаимодействующие программы. К таким решениям можно отнести так называемые *мониторы,* предложенные Хоаром [31].

В параллельном программировании *монитор* – это пассивный набор разделяемых переменных и повторно входимых процедур доступа к ним, которым процессы пользуются в режиме разделения, причём в каждый момент им может пользоваться только один процесс.

Рассмотрим, например, некоторый ресурс, который разделяется между процессами каким-либо планировщиком [37]. Каждый раз, когда процесс желает получить в своё распоряжение какие-то ресурсы, он должен обратиться к программе–планировщику. Этот планировщик должен иметь переменные, с помощью которых он отслеживает, занят ресурс или свободен. Процедуру планировщика разделяют все процессы, и каждый процесс может в любой момент захотеть обратиться к планировщику. Но планировщик не в состоянии обслуживать более одного процесса одновременно. Такая процедура-планировщик и представляет собой пример монитора.

Таким образом, монитор – это механизм организации параллелизма, который содержит как данные, так и процедуры, необходимые для реализации динамического распределения конкретного общего ресурса или группы общих ресурсов. Процесс, желающий получить доступ к разделяемым переменным, должен обратиться к монитору, который либо предоставит доступ, либо откажет в нём. Необходимость входа в монитор с обращением к какой-либо его процедуре (например, с запросом на выделение требуемого ресурса) может возникать у многих процессов. Однако вход в монитор находится под жёстким контролем – здесь осуществляется взаимоисключение процессов, так что в каждый момент времени только одному процессу разрешается войти в монитор. Процессам, которые хотят войти в монитор, когда он уже занят, приходится ждать, причем режимом ожидания автоматически управляет сам монитор. При отказе в доступе монитор блокирует обратившийся к нему процесс и определяет условие, по которому процесс ждёт. Проверка условия выполняется самим монитором, который и деблокирует ожидающий процесс. Поскольку механизм монитора гарантирует взаимоисключение процессов,

отсутствуют серьёзные проблемы, связанные с организацией параллельных взаимодействующих процессов. Внутренние данные монитора могут быть либо глобальными (относящимися ко всем процедурам монитора), либо локальными (относящимися только к одной конкретной процедуре). Ко всем этим данным можно обращаться только изнутри монитора; процессы, находящиеся вне монитора и, по существу, только вызывающие его процедуры, просто не могут получить доступ к данным монитора. При первом обращении монитор присваивает своим переменным начальные значения. При каждом последующем обращении используются те значения переменных, которые сохранились от предыдущего обращения.

Если процесс обращается к некоторой процедуре монитора и обнаруживается, что соответствующий ресурс уже занят, эта процедура монитора выдает команду ожидания WAIT с указанием условия ожидания. Процесс мог бы оставаться внутри монитора, однако это противоречит принципу взаимоисключения, если в монитор затем вошёл бы другой процесс. Поэтому процесс, переводящийся в режим ожидания, должен вне монитора ждать того момента, когда необходимый ему ресурс освободится.

Со временем процесс, который занимал данный ресурс, обратится к монитору, чтобы возвратить ресурс системе. Соответствующая процедура монитора при этом может просто принять уведомление о возвращении ресурса, а затем ждать, пока не поступит запрос от другого процесса, которому потребуется этот ресурс. Однако может оказаться, что уже имеются процессы, ожидающие освобождения данного ресурса. В этом случае монитор выполняет команду извещения (сигнализации) SIGNAL, чтобы один из ожидающих процессов мог получить данный ресурс и покинуть монитор. Если процесс сигнализирует о возвращении (иногда называемом освобождением) ресурса и в это время нет процессов, ожидающих данного ресурса, то подобное оповещение не вызывает никаких других последствий кроме того, что монитор, естественно, вновь вносит ресурс в список свободных. Очевидно, что процесс, ожидающий освобождения некоторого ресурса, должен находиться вне монитора, чтобы другой процесс имел возможность войти в монитор и возвратить ему этот ресурс.

Чтобы гарантировать, что процесс, находящийся в ожидании некоторого ресурса, со временем получит этот ресурс, делается так, что ожидающий процесс имеет более высокий приоритет, чем новый процесс, пытающийся войти в монитор. В противном случае новый процесс мог бы перехватить ожидаемый ресурс до того, как ожидающий процесс вновь войдет в монитор. Если допустить многократное повторение подобной нежелательной ситуации, то ожидающий процесс мог бы откладываться бесконечно. Однако для систем реального времени можно допустить использование дисциплины обслуживания на основе абсолютных или динамически изменяемых приоритетов.

В листинге 6.16 в качестве примера приведён простейший монитор для выделения одного ресурса.

Листинг 6.16. Пример монитора Хоара

```
monitor Resourse;
condition free;      { условие - свободный }
var busy : boolean;  { busy - занят }

procedure REQUEST;       {запрос}
begin
        if busy then WAIT ( free);
        busy:=rtrue;
        TakeOff;    { ВЫДАТЬ РЕСУРС }
end;

procedure RELEASE;
begin
        TakeOn;    { ВЗЯТЬ РЕСУРС }
        busy:=false;
        SIGNAL ( free )
end;

begin
        busy:=false;
end.
```

Единственный ресурс динамически запрашивается и освобождается процессами, которые обращаются к процедурам REQUEST (запрос) и RELEASE (осво-

бодить). Если процесс обращается к процедуре REQUEST в тот момент, когда ресурс используется, значение переменной busy (занято) будет равно true, и процедура REQUEST выполнит операцию монитора WAIT(free). Эта операция блокирует не процедуру REQUEST, а обратившийся к ней процесс. Процесс помещается в конец очереди процессов, ожидающих, пока не будет выполнено условие free (свободный).

Когда процесс, использующий ресурс, обращается к процедуре RELEASE, операция монитора SIGNAL деблокирует процесс, находящийся в начале очереди, не позволяя исполняться никакой другой процедуре внутри того же монитора. Этот деблокированный процесс готов возобновить исполнение процедуры REQUEST сразу же после операции WAIT(free), которая его и блокировала. Если операция SIGNAL(free) выполняется в то время, когда нет процесса, ожидающего условия free, то никакие действия не выполняются.

Использование монитора в качестве основного средства синхронизации и связи освобождает процессы от необходимости явно разделять между собой информацию. Напротив, доступ к разделяемым переменным всегда ограничен телом монитора, и поскольку мониторы входят в состав ядра операционной системы, то разделяемые переменные становятся системными переменными. Это автоматически исключает критические интервалы (так как в каждый момент монитором может пользоваться только один процесс, то два процесса никогда не могут получить доступ к разделяемым переменным одновременно).

Монитор является пассивным объектом в том смысле, что это не процесс; его процедуры выполняются только по требованию процесса.

Хотя по сравнению с семафорами мониторы не представляют собой существенно более мощного инструмента для организации параллельных взаимодействующих вычислительных процессов, у них есть некоторые преимущества перед более примитивными синхронизирующими средствами. Во-первых, мониторы очень гибки. В форме мониторов можно реализовать не только семафоры, но и многие другие синхронизирующие операции. Например, разобранный во втором разделе данной главы механизм решения задачи «поставщик – потребитель» легко

запрограммировать в виде монитора. Во-вторых, локализация всех разделяемых переменных внутри тела монитора позволяет избавиться от малопонятных конструкций в синхронизируемых процессах. Сложные взаимодействия процессов можно синхронизировать наглядным образом. В-третьих, мониторы дают процессам возможность совместно использовать программные модули, представляющие собой критические секции. Если несколько процессов совместно используют ресурс и работают с ним совершенно одинаково, то в мониторе нужна только одна процедура, тогда как решение с семафорами требует, чтобы в каждом процессе имелся собственный экземпляр критической секции. Таким образом, мониторы обеспечивают по сравнению с семафорами значительное упрощение организации взаимодействующих вычислительных процессов и большую наглядность при лишь незначительной потере в эффективности.

Почтовые ящики

Тесное взаимодействие между процессами предполагает не только синхронизацию – обмен временными сигналами, но и передачу, и получение произвольных данных – обмен сообщениями. В системе с одним процессором посылающий и получающий процессы не могут работать одновременно. В мультипроцессорных системах также нет никакой гарантии их одновременного исполнения. Следовательно, для хранения посланного, но ещё не полученного сообщения необходимо место. Оно называется *буфером сообщений или почтовым ящиком*[1].

Если процесс P1 хочет общаться с процессом P2, то P1 просит систему образовать или предоставить ему почтовый ящик, который свяжет эти два процесса так, чтобы они могли передавать друг другу сообщения. Для того чтобы послать процессу P2 какое-то сообщение, процесс P1 просто помещает это сообщение в почтовый ящик, откуда процесс P2 может его в любое время взять. При применении почтового ящика процесс P2 в конце концов обязательно получит сообщение, когда обратится за ним, если вообще обратится. Естественно, что процесс P2 должен знать о существовании почтового ящика. Поскольку в системе может быть много почтовых ящиков, необходимо обеспечить доступ процессу к конкретному почто-

[1] Название «почтовый ящик» происходит от обычного приспособления для отправки почты.

вому ящику. Почтовые ящики являются системными объектами, и для пользования таким объектом необходимо получить его у операционной системы, что осуществляется с помощью соответствующих запросов.

Если объём передаваемых данных велик, то эффективнее не передавать их непосредственно, а отправлять в почтовый ящик сообщение, информирующее процесс-получатель о том, где можно их найти.

Почтовый ящик может быть связан с парой процессов, только с отправителем, только с получателем, или его можно получить из множества почтовых ящиков, которые используют все или несколько процессов. Подовый ящик, связанный с процессом-получателем, облегчает посылку сообщений от нескольких процессов в фиксированный пункт назначения. Если почтовый ящик не связан жестко с процессами, то сообщение должно содержать идентификаторы и процесса–отправителя, и процесса-получателя.

Итак, почтовый ящик – это информационная структура, поддерживаемая операционной системой. Она состоит из головного элемента, в котором находится информация о данном почтовом ящике, и из нескольких буферов (гнёзд), в которые помещают сообщения. Размер каждого буфера и их количество обычно задаются при образовании почтового ящика.

Правила работы почтового ящика могут быть различными в зависимости от его сложности [37]. В простейшем случае сообщения передаются только в одном направлении. Процесс P1 может посылать сообщения до тех пор, пока имеются свободные гнёзда. Если все гнёзда заполнены, то P1 может либо ждать, либо заняться другими делами и попытаться послать сообщение позже. Аналогично процесс P2 может получать сообщения до тех пор, пока имеются заполненные гнёзда. Если сообщений нет, то он может либо ждать сообщений, либо продолжать свою работу. Эту простую схему работы почтового ящика можно усложнять в нескольких направлениях и получать более хитроумные системы общения – двунаправленные и многовходовые почтовые ящики.

Двунаправленный почтовый ящик, связанный с парой процессов, позволяет подтверждать приём сообщений. Если используется множество гнёзд, то каждое из

них хранит либо сообщение, либо подтверждение. Чтобы гарантировать передачу подтверждений, когда все гнёзда заняты, подтверждение на сообщение помещается в то же гнездо, которое было использовано для сообщения, и оно уже не используется для другого сообщения до тех пор, пока подтверждение не будет получено. Из-за того, что некоторые процессы не забрали свои сообщения, связь может быть приостановлена. Если каждое сообщение снабдить пометкой времени, то управляющая программа может периодически уничтожать старые сообщения.

Процессы могут быть также остановлены в связи с тем, что другие процессы не смогли послать им сообщения. Если время поступления каждого остановленного процесса в очередь заблокированных процессов регистрируется, то управляющая программа может периодически посылать им пустые сообщения, чтобы они не ждали чересчур долго.

Реализация почтовых ящиков требует использования примитивных операторов низкого уровня, таких как P- и V-операции, или каких-либо других средств, но пользователям может дать средства более высокого уровня (наподобие мониторов Хоара), например, ввести следующие операции:

1 SEND_MESSAGE (Получатель, Сообщение, Буфер)

переписывает сообщение в некоторый буфер, помещает его адрес в переменную Буфер и добавляет буфер к очереди Получатель. Процесс, выдавший операцию SEND_MESSAGE, продолжит своё исполнение.

2 WAIT_MESSAGE (Отправитель, Сообщение, Буфер)

блокирует процесс, выдавший операцию, до тех пор, пока в его очереди не появится какое-либо сообщение. Когда процесс устанавливается на процессор, он получает имя отправителя в переменной Отправитель, текст сообщения – в Сообщение и адрес буфера – в Буфер. Затем буфер удаляется из очереди, и процесс может записать в него ответ отправителю.

3 SEND_ANSWER (Результат, Ответ, Буфер)

записывает Ответ в тот Буфер, из которого было получено сообщение, и добавляет буфер к очереди отправителя. Если отправитель ждёт ответ, он деблокируется.

4 WAIT_ANSWER (Результат, Ответ, Буфер)

блокирует процесс, выдавший операцию, до тех пор, пока в Буфер не поступит ответ. После того как ответ поступил, и процесс установлен на процессор, Ответ переписывается в память процессу, а буфер освобождается. Результат указывает, является ли ответ пустым, то есть выданным операционной системой, так как сообщение было адресовано несуществующему (или так и не ставшим активным) процессу.

Основные достоинства почтовых ящиков:

♦ процессу не нужно знать о существовании других процессов до тех пор, пока он не получит сообщения от них;

♦ два процесса могут обмениваться более чем одним сообщением за один раз;

♦ операционная система может гарантировать, что никакой процесс не вмешается в «беседу» других процессов;

♦ очереди буферов позволяют процессу-отправителю продолжать работу, не обращая внимания на получателя.

Основным недостатком буферизации сообщений является появление еще одного ресурса, которым нужно управлять, самих почтовых ящиков.

Другим недостатком можно считать статический характер этого ресурса: количество буферов для передачи сообщений через почтовый ящик фиксировано. Поэтому естественным стало появление механизмов, подобных почтовым ящикам, но реализованных на принципах динамического выделения памяти под передаваемые сообщения.

Конвейеры и очереди сообщений

Конвейеры (программные каналы)

Конвейер (pipe – программный канал (связи), или, как его иногда называют, *транспортер)* является средством, с помощью которого можно производить обмен данными между процессами. Принцип работы конвейера основан на механизме ввода/вывода, который используется для работы с файлами в UNIX, то есть задача, передающая информацию, действует так, как будто она записывает данные в файл, в то время как задача, для которой предназначается эта информация, читает её из

этого файла. Операции записи и чтения осуществляются не записями, как это делается в обычных файлах, а потоком байтов, как это было принято в UNIX-системах. Таким образом, функции, с помощью которых выполняется запись в канал и чтение из него, являются теми же самыми, что и при работе с файлами. По сути, канал представляет собой поток данных между двумя (или более) процессами. Это упрощает программирование и избавляет программистов от использования каких-то новых механизмов. На самом деле конвейеры не являются файлами на диске, а представляют собой буферную память, работающую по принципу FIFO, то есть по принципу обычной очереди. Однако не следует путать конвейеры с очередями сообщений; последние реализуются иначе и имеют другие возможности.

Конвейер имеет определенный размер[1], который не может превышать 64 Кбайт, и работает циклически. Вспомните реализацию очереди на массивах, когда имеются указатели начала и конца очереди, которые перемещаются циклически по массиву. Имеется некий массив и два указателя: один показывает на первый элемент (назовем его условно, head), а второй – на последний (назовем его tail).

В начальный момент оба указателя равны нулю. Добавление самого первого элемента в пустую очередь приводит к тому, что указатели head и tail принимают значение, равное 1 (в массиве появляется первый элемент). В последующем добавление нового элемента вызывает изменение значения второго указателя, поскольку он отмечает расположение именно последнего элемента очереди. Чтение (и удаление) элемента (читается и удаляется всегда первый элемент из созданной очереди) приводит к необходимости модифицировать значение указателя head. В результате операций записи (добавления) и чтения (удаления) элементов в массиве, моделирующем очередь элементов, указатели будут перемещаться от начала массива к его концу. При достижении указателем значения индекса последнего элемента массива значение указателя вновь становится единичным (если при этом не произошло переполнение массива, то есть количество элементов в очереди не стало больше числа элементов в массиве). Можно сказать, что мы как бы замыкаем массив в кольцо, организуя круговое перемещение указателей head и tail, которые отслеживают пер-

[1] *Pipe* (канал или конвейер) был введен в UNIX-системах и имеет максимальный размер в 64 Кбайт, поскольку в 16-разрядных мини-ЭВМ, для которых создавалась эта система, нельзя было создать массив данных большего размера.

вый и последний элементы в очереди. Сказанное проиллюстрировано на рис. 6.5. Именно так и функционирует конвейер.

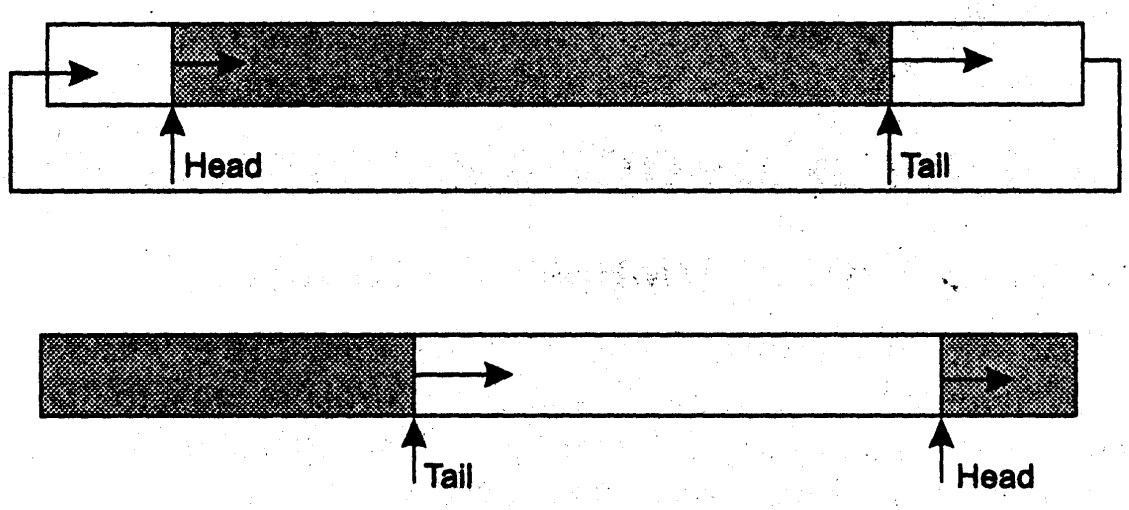

Рис. 6.5. Организация очереди на массиве

Как информационная структура канал описывается идентификатором, размером и двумя указателями. Конвейеры представляют собой системный ресурс. Чтобы начать работу с конвейером, процесс сначала должен заказать его у операционной системы и получить в своё распоряжение. Процессы, знающие идентификатор конвейера, могут через него обмениваться данными.

Теперь рассмотрим основные системные запросы для работы с ними. В качестве примера возьмем вызовы из API OS/2 (в следующем разделе мы ими воспользуемся). Традиционные вызовы для работы с каналами (конвейерами) приведены в разделе, где описывается архитектура системы UNIX (см. главу 8). Итак:

♦ Функция создания конвейера:

```
DosCreatePipe (&ReadHand1e, &WriteHandle, PipeSize);
```

где ReadHandle – описатель для чтения из конвейера, WriteHandle – описатель для записи в конвейер, PipeSize – размер конвейера.

♦ Функция чтения из конвейера:

```
DosRead (&ReadHandle, (PVOID)&Inform, sizeof(Inform), &BytesRead);
```

где ReadHandle – описатель для чтения из конвейера, Inform – переменная любого типа, sizeof(Inform) – размер переменной Inform, BytesRead – количество

прочитанных байтов. Данная функция при обращении к пустому конвейеру будет ожидать, пока в конвейере не появится информация для чтения.

♦ Функция записи в конвейер:

DosWrite (&WriteHand1e, (PVOID)&Inform, s1zeof(Inform), &BytesWrite);

где WriteHandle – описатель для записи в конвейер, BytesWrite – количество записанных байтов.

Читать из конвейера может только тот процесс, который знает идентификатор соответствующего конвейера. При работе с конвейером данные непосредственно помещаются в него. Ещё раз отметим, что из-за ограничения на размер конвейера программисты сталкиваются и с ограничениями на размеры передаваемых через него сообщений.

Очереди сообщений

Очереди сообщений (Queue) являются более сложным методом связи между взаимодействующими процессами по сравнению с каналами. С помощью очередей также можно из одной или нескольких задач независимым образом посылать сообщения некоторой задаче-приёмнику. При этом только процесс-приёмник может читать и удалять сообщения из очереди, а процессы-клиенты имеют право лишь помещать в очередь свои сообщения. Таким образом, очередь работает только в одном направлении. Если же необходима двухсторонняя связь, то можно создать две очереди.

Работа с очередями сообщений имеет много отличий от работы с конвейерами. Во-первых, очереди сообщений предоставляют возможность использовать несколько дисциплин обработки сообщений:

♦ FIFO – сообщение, записанное первым, будет первым и прочитано;

♦ LIFO – сообщение, записанное последним, будет прочитано первым;

♦ приоритетный – сообщения читаются с учётом их приоритетов;

♦ произвольный доступ, то есть можно читать любое сообщение, тогда как канал обеспечивает только дисциплину FIFO.

Во-вторых, если при чтении сообщения из канала (конвейера) оно удаляется из него, то при чтении сообщения из очереди этого не происходит, и сообщение при желании может быть прочитано несколько раз,

В третьих, в очередях присутствуют не непосредственно сами сообщения, а только их адреса в памяти и размер. Эта информация размещается системой в сегменте памяти, доступном для всех задач, общающихся с помощью данной очереди.

Каждый процесс, использующий очередь, должен предварительно получить разрешение на использование общего сегмента памяти с помощью системных запросов API, ибо очередь – это системный механизм и для работы с ним требуются системные ресурсы и, соответственно, обращение к самой ОС. Во время чтения из очереди задача-приёмник пользуется следующей информацией:

- ♦ идентификатор процесса (PID – process ID), который передал сообщение;
- ♦ адрес и длина переданного сообщения;
- ♦ ждать или нет, если очередь пуста;
- ♦ приоритет переданного сообщения;
- ♦ номер освобождаемого семафора, когда сообщение передаётся в очередь.

Наконец, приведём перечень основных функций, управляющих работой очереди (без подробного описания передаваемых параметров), поскольку в различных ОС обращения к этим функциям могут существенно различаться:

- ♦ CreateQueue – создание новой очереди;
- ♦ OpenQueue – открытие существующей очереди;
- ♦ ReadQueue – чтение и удаление сообщения из очереди;
- ♦ PeekQueue – чтение сообщения без его последующего удаления из очереди;
- ♦ WriteQueue – добавление сообщения в очередь;
- ♦ CloseQueue – завершение использования очереди;
- ♦ PurgeQueue – удаление из очереди всех сообщений;
- ♦ QueryQueue – определение числа элементов в очереди.

Примеры создания параллельных взаимодействующих вычислительных процессов

В завершение данного раздела рассмотрим учебный пример, в котором ставится задача создания комплекса параллельно выполняющихся взаимодействующих программ. Пример не будет иметь содержательного смысла в том плане, что никакой полезной работы программные модули, взаимодействующие между собой, не выполняют. Они только синхронизируют друг друга по предопределенной схеме, демонстрируя главным образом способы организации взаимодействующих вычислений – взаимное исключение при выполнении критических интервалов, обмены синхросигналами и данными.

Пусть в нашем примере каждая программа (при своём исполнении либо как самостоятельный вычислительный процесс, либо как задача) должна сообщить время начала своей работы, время окончания работы и, возможно, имена тех программ, которые она (при определённых условиях) должна будет запланировать на выполнение.

Для иллюстрации различий в организаций взаимодействия полноценных вычислительных процессов и многозадачных (многопотоковых) приложений приведём два примера реализации, что позволит увидеть разные механизмы.

Начнем с более простого случая, когда создается обычное мультитредовое приложение, причём воспользуемся не средствами API, а методами, специально созданными для системы программирования. Второй пример будет иллюстрировать применение более мощных средств для организации взаимного исключения и обмена сообщениями; здесь будут использованы средства самой ОС.

Пример создания многозадачного приложения с помощью системы программирования Borland Delphi

Рассмотрим пример использования механизмов, специально созданных для организации взаимного исключения, которые имеются в системе программирования Borland Delphi 3.0. Эта система программирования, будучи ориентированной на создание приложений в многозадачной операционной системе Microsoft Win-

dows 9x, содержит в себе стандартные классы, позволяющие без особых усилий использовать многопоточные возможности этих ОС. Воспользуемся стандартным классом TThread. Объект, создаваемый на основе этого класса, можно охарактеризовать следующими теперь уже очевидными для нас свойствами:

♦ каждый тред имеет свою, при необходимости уникальную, исполняемую часть;

♦ каждый тред для своего исполнения требует отдельного процессорного времени, то есть диспетчер задач принимает во внимание только приоритет треда;

♦ диспетчеризация выполнения тредов осуществляется операционной системой и не требует вмешательства программиста;

♦ несколько тредов, принадлежащих одному процессу, могут использовать один и тот же ресурс (например, глобальную переменную). Как нам известно, треды могут обращаться к полям другого треда из того же вычислительного процесса. При этом программисту необходимо самостоятельно ограничивать доступ к этому ресурсу во избежание известных проблем.

Итак, пусть необходимо создать многопоточное приложение, схема взаимодействия отдельных потоков в котором (в рамках единого вычислительного процесса) приведена на рис. 6.6.

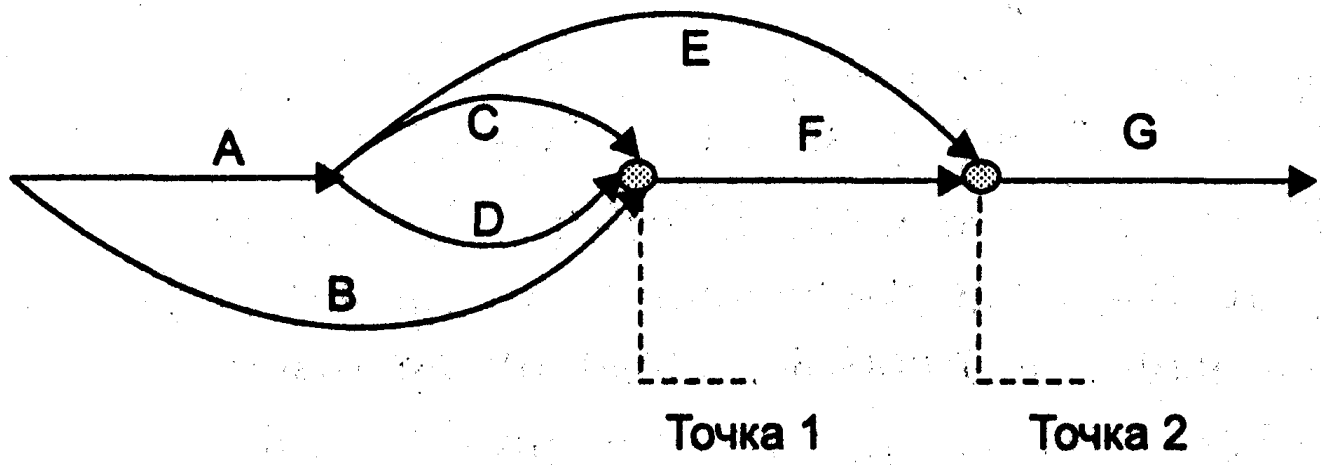

Рис. 6.6. Схема №1 взаимодействия параллельно выполняющихся задач

Так, согласно этому рисунку, процесс A после своего завершения запускает задачи D, C и E. Считаем, что задачи B, D и C завершаются примерно в одинаковое

время. По крайней мере, нам не известно, какой из потоков должен быть первым, а какой – последним. Однако по условиям задачи пусть поток F будет запускаться тем из перечисленных тредов, который завершается первым, но только после того, как завершатся два оставшихся треда, приходящие в «точку синхронизации». Наконец, пусть задача G запускается последним закончившим работу потоком E или F.

Все указанные задачи создадим как потомки объекта TThread. Тексты всех программных модулей приведены в приложении А. Поскольку, согласно условию, действия, выполняемые задачами, для нас не имеют значения, представим исполняемую часть простейшим циклом с соответствующей задержкой. Для наглядности внутри цикла можно организовать вывод текущего состояния выполнения задачи в процентах на «строке состояния», для чего используем компонент TGauge. Благодаря тому, что все семь тредов похожи (используют одни и те же методы) и отличаются только в части принятия решения о синхронизации, опишем организацию базового объекта-треда.

Базовый объект (TTreadProgress) является потомком объекта TTread. При этом он имеет следующие поля:

- ♦ имя треда;
- ♦ строка состояния треда;
- ♦ «длина» треда (время его работы в отсутствие конкурентов);
- ♦ текущее состояние треда;
- ♦ признак завершения треда;
- ♦ имя запустившего треда;
- ♦ строка для вывода сообщений в компонент TMemo.

В базовом объекте объявлены следующие процедуры:

- ♦ исполняемая часть;
- ♦ завершающая часть;
- ♦ процедура прорисовки строки состояния;
- ♦ процедура вывода сообщения;
- ♦ конструктор объекта.

Все треды (от A до G) являются потомками этого объекта и перекрывают единственный метод – процедуру завершения процесса. В исполняемой части задачи после завершения цикла задержки, имитирующего выполнение полезной работы, устанавливается признак завершения и вызывается процедура завершения задачи, которая и выполняет соответствующие действия.

Общую схему работы программы, реализующей задание, можно описать следующим образом. Все задачи инициализируются соответствующей процедурой одновременно, но в режиме ожидания запуска. В качестве параметров инициализации в создаваемый поток передаются его имя, длительность и имя запускающего объекта (если оно известно заранее). Сразу после инициализации запускаются задачи A и B. Обе задачи сигнализируют об этом соответствующим сообщением. После своего завершения поток A запускает задачи (потоки) C, D и E. Далее всё идет в соответствии с заданной блок-схемой. Задача, запускающая другую задачу, передаёт ей свое имя, обращаясь непосредственно к полю этого объекта. Информацию о том, завершился тот или иной поток, можно получить, обратившись к соответствующему полю – признаку завершения задачи.

Естественно, что при подобной организации доступа к полям тредов вероятно возникновение разного рода критических ситуаций. Напомним, основная причина их возникновения заключена в том, что несколько задач (в нашем случае – потоков) реально имеют возможность обращения к общим ресурсам практически одновременно, то есть с таким интервалом времени, за который этот ресурс не успеет изменить своё состояние. В результате задачи могут получать некорректные значения, о чем мы уже немало говорили.

Каждый процесс имеет связь с так называемыми VCL-объектами – видимыми компонентами. В данном случае такими являются строка состояния TGauge и поле сообщений TMemo. Для того чтобы в процессе работы нескольких параллельно выполняющихся задач не возникало критических ситуаций с выводом информации на эти видимые на экране объекты, к ним необходимо обеспечить синхронизированный доступ. Это довольно легко достигается с помощью стандартного для объекта TThread метода Synchronize. Метод имеет в качестве параметра имя процеду-

ры, в которой производится вывод на VCL-объекты. При этом сама эта процедура нигде в программе не должна вызываться напрямую без использования метода Synchronize. В нашей программе такими процедурами являются прорисовка строки состояния (Do Visual Progress) и вывод текстового сообщения (WriteToMemo). Подобное использование метода Synchronize обеспечивает корректную работу нескольких параллельных процессов с VCL-объектами.

Однако метод Synchronize не помогает в случае совместного доступа к другим общим ресурсам. Поэтому необходимо применять другие средства для организации взаимного исключения. Главная цель этих средств заключается в обеспечении монопольного доступа для каждой задачи к общим ресурсам, то есть пока один поток не закончил обращение к подобному ресурсу, другой не имеет право этот ресурс использовать.

В системе программирования Delphi для этой цели имеется довольно-таки простой в использовании и достаточно эффективный метод критической секции с помощью объекта TCriticalSection. Этот метод заключается в следующем:

♦ участок кода каждого потока, в котором производится обращение к общему ресурсу, заключается в «скобки» критической секции – используются методы Enter и Leave;

♦ если какой-либо тред уже находится внутри критической секции, то другой поток, который дошел до «открывающей скобки» Enter, не имеет права входить в критическую секцию до тех пор, пока первый поток находится в ней. Когда первый тред выйдет из критической секции, второй сможет войти в неё и, в свою очередь, обратиться к критическому ресурсу.

Очевидно, что такой метод надежно обеспечивает задачам монопольный доступ к общим (критическим) ресурсам.

Метод критической секции имеет ряд преимуществ перед его аналогами. Так, например, использование семафоров (Semaphore) сложнее в реализации. Другой метод взаимных исключений – mutex – в целом похож на метод критической секции, но он требует больше системных ресурсов и имеет своё время тайм-аута, по

истечении которого ожидающий процесс может всё-таки войти в защищённый блок, в то время как в критической секции подобного механизма нет.

Текст всей программы с необходимыми комментариями приведен в приложении А.

Пример создания комплекса параллельных взаимодействующих программ, выступающих как самостоятельные вычислительные процессы

Теперь рассмотрим более сложную задачу: пусть параллельно выполняющиеся задачи имеют статус полноценного вычислительного процесса, а не потока и, кроме этого, организуем обращение к соответствующим системным механизмам непосредственно на уровне API. Схема взаимодействия программ для этого примера изображена на рис. 6.7.

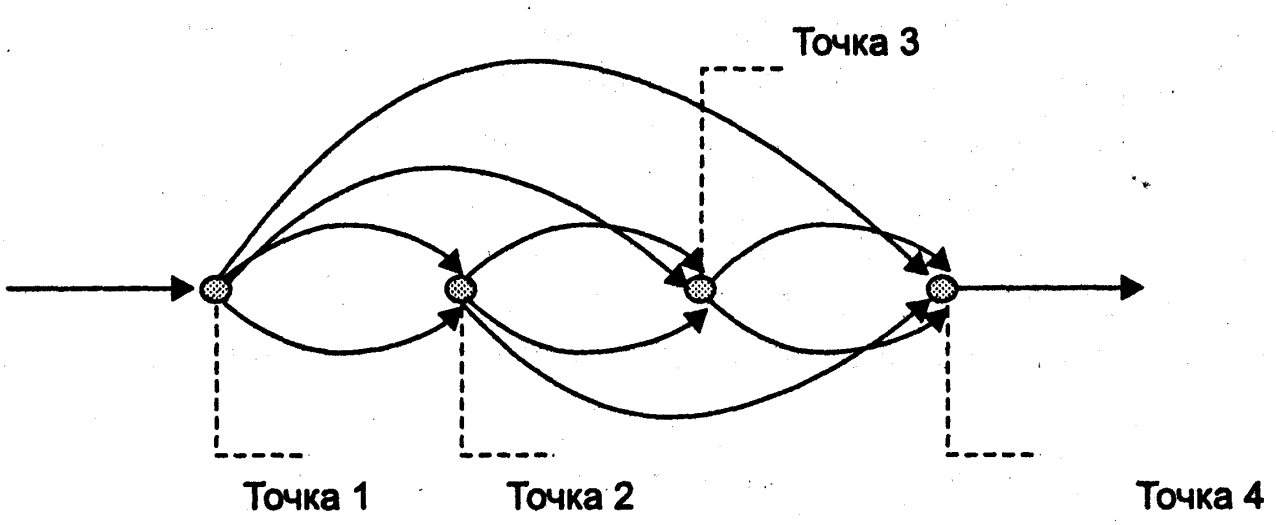

Рис. 6.7. Схема № 2 взаимодействия параллельно выполняющихся вычислительных процессов

На этом рисунке программа с именем A в точке 1 порождает сразу четыре параллельно выполняющихся задачи (вычислительных процесса): B, C, I и J. Процессы B и C завершаются и должны запустить, в свою очередь, два параллельных процесса D и E. В точке 2, в которой происходит синхронизация задач B и C, процессы D и E должны быть запущены той из задач B или C, которая закончит свое выполнение раньше, но только после момента окончания второй. Далее, в точке 3 необходимо запустить программы G и H. Запускает их первая завершившая свою работу программа (D, E или I), но дождавшись завершения остальных программ. Точка

4 синхронизирует выполнение процессов F, G, H и J, при этом программу K запускает последний из завершившихся процессов.

Для решения поставленной задачи будем использовать соответствующие механизмы, описанные нами выше (применительно к OS/2), а именно конвейер (pipe) и семафоры. В нашей задаче через конвейер передается переменная типа ParentInfo:

```
struct ParentInfo
{
  char ParentName [15];
  char LaunchTime [12];
  int Number;
}
```

ParentInfo – хранит имя программы-предка; LaunchTIme – содержит время запуска текущей программы программой-предком (время, когда программа-предок выполнила функцию DosExecPgm); переменная Number указывает, для какого числа программ предназначена переменная ParentInfo.

Читать из конвейера мы можем при помощи функции DorRead, но при этом уничтожается запись в конвейере, и мы обязаны перезаписывать данные в конвейер для других программ. Каждая программа читает данные из конвейера и затем уменьшает значение Number. Программа, установившая значение 1, является последней программой, прочитавшей данные из конвейера; она посылает сигнал предку о завершении инициализации необходимых ресурсов. Программа пишет в конвейер данные только в том случае, если именно она будет запускать следующие программы.

Для чтения и записи в конвейер необходимы переменные WriteHandle и ReadHandle, указывающие на описатели записи и чтения в конвейер. Необходимо, чтобы значения этих переменных, полученных в процессе A, были известны остальным процессам для совместного использования. Для этого значения этих переменных мы можем передать процессам-потомкам в качестве строк аргументов.

При попытке считывания информации из пустого конвейера процесс переводится в состояние ожидания. После записи информации в конвейер другим процессом ожидающий процесс считывает поступившую информацию и продолжает свою работу.

Поскольку программы, которые мы сейчас рассматриваем, созданы для операционной системы OS/2, приведём краткое описание использованных механизмов, которые имеют отличия от вышеописанных общих принципов и схем.

♦ Функция создания семафора:

```
DosCreateEventSem (“\\SEM\\PIPESEM”, &PipeSem, 1, 0);
```

где "\\SEM\\PIPESEM" – имя семафора, PipeSem – идентификатор семафора, 1 – параметр совместного использования семафора (DC_SEM_SHARED), 0 – зарезервированный системный параметр.

♦ Функция открытия семафора:

```
DosOpenEventSem ("\\SEM\\PIPESEM", &PipeSem);
```

где "\\SEM\\PIPESEM" – имя семафора, PipeSem – идентификатор семафора.

♦ Функция установки семафора:

```
DosPostEventSem (PipeSem);
```

где PipeSem – идентификатор семафора.

♦ Функция сброса семафора:

```
DosResetEventSem (PipeSem, &NPost);
```

где PipeSem – идентификатор семафора, NPost – количество обращений к установке семафора с момента последнего сброса.

♦ Функция ожидания установки семафора:

```
DosPostEventSem (PipeSem, -1):
```

где PipeSem – идентификатор семафора, -1 – ожидание семафора до его установки (положительное значение – это задержка в миллисекундах).

Для синхронизации процессов и обработки критических участков программ необходимы семафоры, к которым имеют доступ все работающие программы. Программа-потомок может унаследовать семафор от программы-предка, которая создала или открыла необходимый семафор. Но это произойдет только тогда, когда

программа-предок дождётся момента открытия семафора в программе-потомке. Это обеспечивается за счёт дополнительных семафоров ExitSem1, ExitSem2, ExitSem3. Когда последняя программа-потомок прочитывает данные из конвейера (работа с конвейером) и обнаруживает, что она последней прошла участок открытия уже созданных семафоров, она устанавливает необходимый семафор, принадлежащий программе-предку. Программа-предок, ожидающая установки семафора, завершает свою работу.

Для управления запуском и завершением программ также используются соответствующие функции.

DosExecPgm (FailFile, sizeof(FailFile), 1, Argument, 0, ResCode, "progr_b.exe") – функция запуска программы-потомка, где

♦ FailFile – буфер для помещения имени объекта, из-за которого возникла ошибка запуска программы, sizeof(FailFile) – размер буфера;

♦ 1 означает, что процесс-потомок следует выполнять асинхронно с процессом-предком, Argument – строки аргументов программы, 0 – строки среды, ResCode – результирующие коды запуска программы, "progr_b.exe" – имя запускаемой (планируемой на выполнение) программы;

♦ DosExit – функция завершения процесса (и всех его подпроцессов);

♦ DosSetPriority – установка приоритета программы (и всех его подпроцессов).

Каждую программу в соответствии с заданием создаём как независимую, то есть имеющую своё локальное адресное пространство. Благодаря этому переменные в текстах программ, имеющие одинаковые имена (см. листинги в приложении Б), фактически являются разными переменными. Программа с именем A является начальной, стартовой. Именно она создаёт те системные объекты, которыми потом пользуются её потомки для своей синхронизации. Имена системных объектов они получают в наследство при своём порождении.

Для совместного использования вычислительными процессами одного файла необходимо правильно его открыть в программе A. После корректного открытия или создания файла к нему могут обращаться все программы, имеющие идентифи-

катор открытого файла. Значение этого идентификатора передается запускающимся процессам в строке аргументов, сопровождающих вызов.

В соответствии с заданием каждая программа записывает в файл время своего запуска, имя процесса-предка и время завершения своей работы. Поскольку файл используется всеми программами и запись производится в строгой последовательности, часть программы, которая обеспечивает запись в файл, должна быть признана нами как критический участок программы (критический интервал). Алгоритм обработки этого участка описан ниже. Если не пытаться регулировать доступ процессов к файлу, может возникнуть беспорядочная запись в файл. При регулировании записи каждый процесс производит сразу все три записи в файл.

Опишем теперь алгоритм обработки критических участков программ – записи в файл и работы с конвейером.

♦ Критический участок – работа с конвейером.

Приведем фрагмент программы, обеспечивающий чтение из конвейера.

```
1: do { DosWaitEventSem (PipeSem, -1);
2:     rc=DosResetEventSem (PipeSem, &NPost);
3: } while (rc!=0);
4: DosRead(ReadHandle,(PVOID)&OldInform,sizeof(OldInform),
BytesReaden);
5: DosPostEventSem(PipeSem);
```

Программа А создает семафор PipeSem, который затем используют все программы для прохождения критической части программы, связанной с чтением и записью в конвейер.

В строке 1 программа ожидает установки семафора PipeSem; после того как этот семафор будет установлен, программа переходит к строке 2. Переменная rc возвращает код ошибки при попытке сбросить семафор в строке 2. Это сделано со следующей целью: если после завершения строки 1 будет выполняться другая программа и она успеет сбросить семафор, то когда программа вновь продолжит своё выполнение, она обнаружит в строке 2, что семафор уже сброшен и rc не будет равно нулю; затем цикл повторится. Так будет продолжаться до тех пор, пока rc не

станет равно нулю, то есть текущая программа первой сбросила семафор. Тогда в строке 4 программа считывает из конвейера данные и сигнализирует об освобождении ресурса в строке 5.

Для записи в конвейер используется аналогичный алгоритм.

♦ Критический участок –запись в файл.

Здесь использован такой же алгоритм, как и при работе с конвейером, но задействован специально созданный для этого семафор FileSem. Теперь рассмотрим алгоритмы, решающие задачи синхронизации в каждой из точек нашей схемы (см. рис. 6.7). Начнем с прохождения точки 2. Приведём фрагмент исходного текста программы (см. приложение Б).

```
1: rc=DosPostEventSem(Point1Sem) :
2: if (rc==0)
3: { DosWrite(WriteHandle, (PVOID)&NewInform,  sizeof(NewInform),
&BytesWr1tten);
4: DosCreateEventSem("\\SEM32\\EXITSEM2", &ExitSem2,
DC_SEM_SHARED, 0);
5: DosExecPgm(FailFileb, sizeof(FailFileb), 1, Argument, 0,  &ResCodeb,
"progr_d.exe");
6: DosExecPgm(FailFileb, sizeof(FailFileb), 1, Argument, 0,  &ResCodeb,
"progr_e.exe");
7: DosExecPgm(FailFileb, sizeof(FailFileb), 1, Argument, 0,  &ResCodeb,
"progr_f.exe");
8: DosWaitEventSem(ExitSem2, -1);
9: DosCloseEventSem(ExitSem2); }
```

В точке 2 программы D, E, и F должны быть запущены процессом, который первым завершит свою работу. Для определения последовательности завершения работы программ (B или C) нами используется семафор Point1Sem. В строке 1 производится установка данного семафора. Если значение rc не равно нулю, значит, семафор уже установлен, то есть текущая программа не первой завершила свою работу. Если rc равно нулю, то текущая программа закончила работу первой и

осуществляется переход к строке 3. В строке 3 осуществляется запись в конвейер. Строка 4 – создание семафора ожидания. Этот семафор используется для ожидания открытия семафоров в программах-потомках. Ожидание открытия семафоров происходит в строке 8, затем семафор закрывается. В строках 5–7 осуществляется запуск программ D, E, F.

Теперь приведём алгоритм прохождения точки 3. Фрагмент исходного текста программы изображен ниже.

```
1:  rc=DosPostEventSem(Point2Sem);
2: if (rc==0)
3:  { do{ DosQueryEventSem(Point2Sem, &BytesWritten);
4: }while (BytesWritten<=2);
5: DosWrite(WriteHandle, (PVOID)&NewInform, sizeof(NewInform),
  &BytesWritten);
6: DosCreateEventSem("\\SEM32\\EXITSEM3",  &ExitSem3,
  DC_SEM_SHARED, 0);
7: DosExecPgm(FailFileb, sizeof(FailFileb), Argument, 0,  &ResCodeb,
   "progr_g.exe");
8: DosExecPgm(FailFileb, sizeof(FailFileb), Argument, 0, &ResCodeb,
   "progr_h.exe");
9: DosWaitEventSem(ExitSem3, -1);
```

В точке 3 программы G и H запускаются той программой, которая первой завершает свою работу, но только после того, как работу завершат остальные программы. Для определения последовательности завершения работы программ (I, D или E) используется семафор Point2Sem. В строке 1 производится установка данного семафора. Если значение rc не равно нулю, значит, семафор уже установлен, то есть текущая программа не первой завершила свою работу. Если rc равно нулю, то текущая программа закончила работу первой и осуществляется переход к строке 3. В этой строке подсчитывается количество обращений к данному семафору. Цикл повторяется до тех пор, пока количество обращений не станет равно 3, то есть когда все остальные программы завершили свою работу. После этого в строке 5 осу-

ществляется запись в конвейер. Строка 6 – создание семафора ожидания. Этот семафор используется для ожидания открытия семафоров в программах-потомках. Ожидание открытия семафоров происходит в строке 9. В строках 7 и 8 осуществляется запуск программ G и H соответственно.

Наконец, приведём ещё алгоритм прохождения точки 4. Фрагмент исходного текста программы, реализующей эту задачу, приведён ниже.

```
1: do { DosWaitEventSem(Point1Sem, -1);
2: rc=DosResetEventSem(Point1Sem, &BytesReaden);
3: } while (rc!=0);
4: DosPostEventSem(Point3Sem);
5: DosQueryEventSem(Point3Sem, &BytesWritten);
6: DosPostEventSem(Point1Sem);
7: if (BytesWritten==4)
8: { DosWrite(WriteHandle, (PVOID)&NewInform,  sizeof(NewInform),
  &BytesWritten);
9: DosCreateEventSem("\\SEM32\\EXITSEM4", &ExTtSem4,
  DC_SEM_SHARED, 0);
10: DosExecPgm(FailFileb, sizeof(FailFileb), 1, Argument, 0, &ResCodeb,
   "progr_k.exe"):
11: DosWaitEventSem(ExitSem4, -1);
```

Итак, в точке 4 программа K запускается задачей, которая последней завершила свою работу. Для определения последовательности завершения работы программ (J, G, H или F) используется семафор Point3Sem. Каждая программа должна обратиться к данному семафору, установить его и проверить количество обращений к нему. Но при выполнении этих двух операций другие программы могут также установить семафор, и значение обращений к семафору в текущей программе окажется неверным. Для доступа к семафору Point3Sem используется семафор Point1Sem. В строке 1 программа ожидает установки семафора Point1Sem, который сигнализирует о доступности неразделяемого ресурса Point3Sem. В строке 2 семафор сбрасывается, но если другие программы успели сбросить семафор, то об

этом сообщит значение rc. Если текущей программе удалось завладеть ресурсом, то есть значение rc равно нулю, то дальше в строках 4, 5 производится установка семафора Point3Sem и подсчёт количества обращений. Строка 6 сигнализирует о завершении работы критической части программы установкой семафора Point1Sem. Затем, как и в предыдущих алгоритмах, программа записывает информацию в транспортёр (строка 8), создает семафор ожидания открытия семафоров в программах-потомках (строка 9), запускает программу К (строка 10) и ожидает открытия семафоров в программе К (строка 11).

Тексты четырех программ (A, B, D и G), действующих по рассмотренным алгоритмам, приведены в приложении Б. Остальные программы аналогичны им и отличаются только использованием других имён.

Контрольные вопросы и задачи

Вопросы для проверки

1 Какие последовательные вычислительные процессы мы называем параллельными и почему? Какие параллельные процессы называются независимыми, а какие – взаимодействующими?

2 Изложите алгоритм Деккера, позволяющий разрешить проблему взаимного исключения путём использования одной только блокировки памяти.

3 Объясните команду «проверка и установка».

4 Расскажите о семафорах Дейкстры. Чем обеспечивается взаимное исключение при выполнении P- и V-примитивов?

5 Изложите, как могут быть реализованы семафорные примитивы для мультипроцессорной системы.

6 Что такое мьютекс (mutex)?

7 Изложите алгоритм решения задачи «поставщик – потребитель» при использовании семафоров Дейкстры.

8 Изложите алгоритм решения задачи «читатели – писатели» при использовании семафоров Дейкстры.

9 Что такое «монитор Хоара»? Приведите пример такого монитора.

10 Что представляют собой «почтовые ящики»?

11 Что представляют собой «конвейеры» (программные каналы)?

12 Что представляют собой «очереди сообщений»? Чем отличаются очереди сообщений от почтовых ящиков?

ГЛАВА 7 Проблема тупиков и методы борьбы с ними

Рассмотрим одну из самых серьезных и трудно разрешимых проблем, возникающих при организации мультипрограммирования – проблему тупиков и основные подходы при борьбе с ними. В настоящей главе приводятся некоторые модели параллельных вычислительных процессов, позволяющие проводить анализ последних в аспекте корректного решения указанных проблем.

Понятие тупиковой ситуации при выполнении параллельных вычислительных процессов

При организации параллельного выполнения нескольких процессов одной из главных функций операционной системы является решение сложной задачи корректного распределения ресурсов между выполняющимися процессами и обеспечение последних средствами взаимной синхронизации и обмена данными.

При параллельном исполнении процессов могут возникать ситуации, при которых два или более процесса всё время находятся в заблокированном состоянии. Самым простым является случай, когда каждый из двух процессов ожидает ресурс, занятый другим процессом. Из-за такого ожидания ни один из процессов не может продолжить исполнение и освободить в конечном итоге ресурс, необходимый другому процессу. Эта тупиковая ситуация называется *дедлоком*[1], тупиком или клинчем. Говорят, что в мультипрограммной системе процесс находится в состоянии тупика, если он ждёт события, которое никогда не произойдёт. Тупики чаще всего возникают из-за конкуренции несвязанных параллельных процессов за ресурсы вычислительной системы, но иногда к тупикам приводят и ошибки программирования.

При рассмотрении проблемы тупиков целесообразно понятие ресурсов системы обобщить и разделить их все на два класса – повторно используемые (или системные) ресурсы (типа RR или SR – reusable resource или system resource) и потребляемые (или расходуемые) ресурсы (типа CR – consumable resource).

Повторно используемый ресурс (SR) есть конечное множество идентичных единиц со следующими свойствами [92]:

♦ число единиц ресурса постоянно;

♦ каждая единица ресурса или доступна, или распределена одному и только одному процессу (разделение либо отсутствует, либо не принимается во внимание, так как не оказывает влияния на распределение ресурсов, а значит, и на возникновение тупиковой ситуации);

♦ процесс может освободить единицу ресурса (сделать её доступной), только если он ранее получил эту единицу, то есть никакой процесс не может оказывать какое-либо влияние ни на один ресурс, если он ему не принадлежит.

Данное определение выделяет существенные для изучения проблемы тупика свойства обычных системных ресурсов, к которым мы относим такие компоненты аппаратуры, как основная память, вспомогательная (внешняя) память, периферийные устройства и, возможно, процессоры, а также программное и информационное обеспечение, такое как файлы данных, таблицы и «разрешение войти в критическую секцию».

Расходуемый ресурс (CR) отличается от ресурса типа SR в нескольких важных отношениях [37]:

♦ число доступных единиц некоторого ресурса типа CR изменяется по мере того, как приобретаются (расходуются) и освобождаются (производятся) отдельные их элементы выполняющимися процессами, и такое число единиц ресурса является потенциально неограниченным; процесс «производитель» увеличивает число единиц ресурса, освобождая одну или более единиц, которые он «создал»;

♦ процесс «потребитель» уменьшает число единиц ресурса, сначала запрашивая и затем приобретая (потребляя) одну или более единиц. Единицы ресурса, ко-

[1]Dead lock – смертельное объятие

торые приобретены, в общем случае не возвращаются ресурсу, а потребляются (расходуются). Эти свойства потребляемых ресурсов присущи многим синхронизирующим сигналам, сообщениям и данным, порождаемым как аппаратурой, так и программным обеспечением, и могут рассматриваться как ресурсы типа CR при изучении тупиков. В их число входят: прерывания от таймера и устройств ввода/вывода; сигналы синхронизации процессов; сообщения, содержащие запросы на различные виды обслуживания или данные, а также соответствующие ответы.

Для исследования параллельных процессов и, в частности, проблемы тупиков было разработано несколько моделей. Одной из них является модель повторно используемых ресурсов Холта [92]. Согласно этой модели система представляется как набор (множество) процессов и набор ресурсов, причём каждый из ресурсов состоит из фиксированного числа единиц. Любой процесс может изменять состояние системы с помощью запроса, получения или освобождения единицы ресурса.

В графической форме процессы и ресурсы представляются квадратами и кружками соответственно. Каждый кружок содержит некоторое количество маркеров (фишек) в соответствии с существующим количеством единиц этого ресурса. Дуга, указывающая из «процесса» на «ресурс», означает запрос одной единицы этого ресурса. Дуга, указывающая из «ресурса» на «процесс», представляет выделение ресурса процессу. Поскольку каждая единица любого ресурса типа SR может быть выделена одновременно не более чем одному процессу, то число дуг, исходящих из ресурса к различным процессам, не может превышать общего числа единиц этого ресурса. Такая модель называется *графом повторно используемых ресурсов.*

Одно из состояний примера системы из двух процессов с ресурсами типа SR представлено на рис. 7.1.

Пусть процесс P1 запрашивает две единицы ресурса R1 и одну единицу ресурса R2. Процессу P2 принадлежат две единицы ресурса R1 и ему нужна одна единица R2. Предположим, что процесс P1 получил бы теперь запрошенную им единицу R2. Если принято правило, по которому процесс должен получить все запрошенные им ресурсы, прежде чем освободить хотя бы один из них, то удовлетворение

запроса P1 приведет к тупиковой ситуации: P1 не сможет продолжиться до тех пор, пока P2 не освободит единицу ресурса R1, а процесс P2 не сможет продолжиться до тех пор, пока P1 не освободит единицу R2. Причиной этого дедлока являются неупорядоченные попытки процессов войти в критический интервал, связанный с выделением соответствующей единицы ресурса.

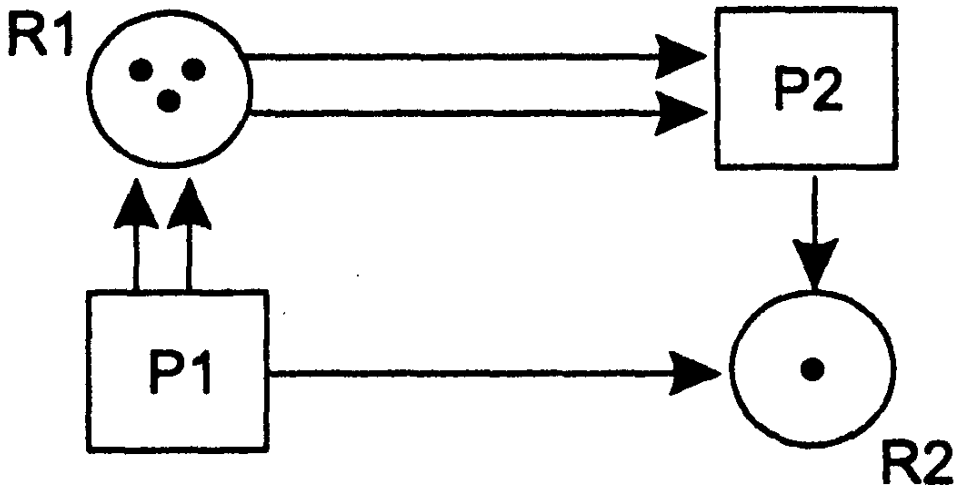

Рис. 7.1. Пример модели Холта для системы из двух процессов

Примеры тупиковых ситуаций и причины их возникновения

Для понимания основных причин возникновения тупиков рассмотрим несколько простых характерных примеров.

Пример тупика на ресурсах типа CR

Пусть имеются три процесса ПР1, ПР2 и ПР3, которые вырабатывают соответственно сообщения Ml, М2 и М3. Эти сообщения представляют собой ресурсы типа CR. Пусть процесс ПР1 является «потребителем» сообщения М3, процесс ПР2 получает сообщение Ml, а ПР3 – сообщение М2 от процесса ПР2, то есть каждый из процессов является и «поставщиком» и «потребителем» одновременно, и вместе они образуют «кольцевую» систему (рис. 7.2) передачи сообщений через почтовые ящики (ПЯ). Если связь с помощью этих сообщений со стороны каждого процесса устанавливается в порядке, изображенном в листинге 7.1, то никаких трудностей

не возникает. Однако перестановка этих двух процедур в каждом из процессов (листинг 7.2) вызывает тупик:

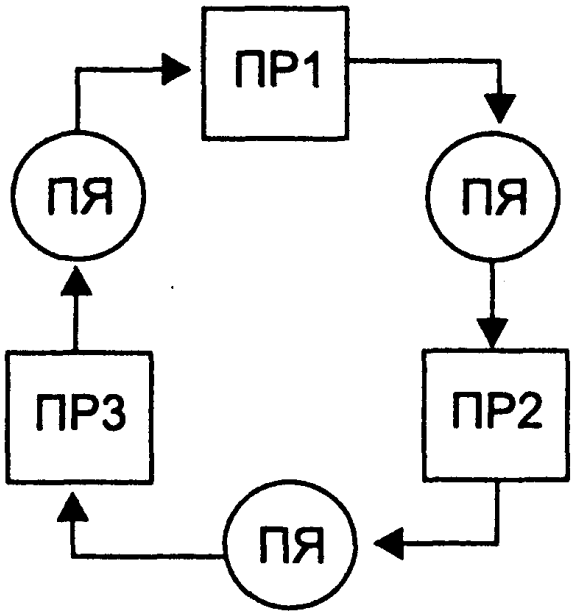

Рис. 7.2. Кольцевая схема взаимодействия процессов

Листинг 7.1. Вариант без тупиковой ситуации

```
ПР1:    . . .
        ПОСЛАТЬ СООБЩЕНИЕ (ПР2, М1, ПЯ2);
        ЖДАТЬ СООБЩЕНИЕ (ПР3, М3, ПЯ1);
              . . .

ПР2:    . . .
        ПОСЛАТЬ СООБЩЕНИЕ (ПР3, М2, ПЯ3);
        ЖДАТЬ СООБЩЕНИЕ (ПР1, М1, ПЯ2);
              . . .

ПР3:    . . .
        ПОСЛАТЬ СООБЩЕНИЕ (ПР1, М3, ПЯ1);
        ЖДАТЬ СООБЩЕНИЕ (ПР2, М2, ПЯ3);
              . . .
```

Листинг 7.2. Вариант с тупиковой ситуацией

```
ПР1:    . . .
        ЖДАТЬ СООБЩЕНИЕ (ПР3, М3, ПЯ1);
        ПОСЛАТЬ СООБЩЕНИЕ (ПР2, М1, ПЯ2);
              . . .
```

```
ПР2:    . . .
        ЖДАТЬ СООБЩЕНИЕ (ПР1, М1, ПЯ2);
        ПОСЛАТЬ СООБЩЕНИЕ (ПР3, М2, ПЯ3);
            . . .
ПР3:    . . .
        ЖДАТЬ СООБЩЕНИЕ (ПР2, М2, ПЯ3);
        ПОСЛАТЬ СООБЩЕНИЕ (ПР1, М3, ПЯ1);
            . . .
```

В самом деле, во втором варианте ни один из процессов не сможет послать сообщения до тех пор, пока сам его не получит, а этого события никогда не произойдет, поскольку ни один процесс не может этого сделать.

Пример тупика на ресурсах типа CR и SR

Пусть некоторый процесс ПР1 должен обменяться сообщениями с процессом ПР2 и каждый из них запрашивает некоторый ресурс R, причем ПР1 требует три единицы этого ресурса для своей работы, а ПР2 – две единицы и только на время обработки сообщения. Всего же имеются только четыре единицы ресурса R. Запрос ресурса можно реализовать через соответствующий монитор с процедурами REQUEST(R, N) – запрос N единиц ресурса R и RELEASE(R, N) – освобождение, возврат N единиц ресурса R. Обмен сообщениями будем осуществлять через почтовый ящик MB. Фрагменты программ ПР1 и ПР2 приведены в листинге. 7.3.

Листинг 7.3. Пример тупика на CR- и SR-ресурсах

```
        . . .
        . . .
ПР1:  REQUEST (R, 3);
      SEND_MESSAGE (ПР2, сообщение, MB);
      WAIT ANSWER (ответ, MB);
        . . .
        . . .
      RELEASE (R, 3);
        . . .
        . . .
```

```
        . . .
        . . .
```

```
ПР2:  WAIT_MESSAGE (ПР1, сообщение, МВ);
      REQUEST (R, 2);
      ОБРАБОТКА СООБЩЕНИЯ;
      RELEASE (R, 2);
      SEND_ANSWER (ответ, МВ);
```

Эти два процесса всегда будут попадать в тупик. Процесс ПР2, если будет выполняться первым, сначала ожидает сообщения от процесса ПР1, после чего будет заблокирован при запросе ресурса R, часть которого будет уже отдана ПР1. Процесс ПР1, завладев частью ресурса R, будет заблокирован на ожидании ответа от ПР2, которого никогда не получит, так как для этого ПР2 нужно получить ресурс R, находящийся в распоряжении ПР1. Тупика можно избежать лишь при условии, что на время ожидания ответа от ПР2 процесс ПР1 будет отдавать хотя бы одну единицу ресурса R, которыми он сейчас владеет. В данном примере, как и в предыдущем, причиной тупика являются ошибки программирования.

Пример тупика на ресурсах типа SR

Предположим, что существуют два процесса ПР1 и ПР2, разделяющих два ресурса типа SR: R1 и R2. Пусть взаимное исключение доступов к этим ресурсам реализуется с помощью семафоров S1 и S2 соответственно. Процессы ПР1 и ПР2 обращаются к ресурсам следующим образом [37] (рис. 7.3):

Процесс ПР1	Процесс ПР2
:	:
1: P(S2);	5: P(S1);
:	:
2: P(S1);	6: P(S2);
:	:
3: V(S1);	7: V(S1);
:	:
4: V(S2);	8: V(S2);
:	:

Рис. 7.3. Пример последовательности операторов для двух процессов, которые могут привести к тупиковой ситуации

Здесь несущественные (с точки зрения обращения к ресурсам) детали опущены. Считаем, что оба семафора первоначально установлены в единицу. Пространство возможных вычислений приведено на рис. 7.4.

Горизонтальная ось задаёт выполнение процесса ПР1, вертикальная – ПР2. Вертикальные линии, пронумерованные от 1 до 4, соответствуют операторам 1– 4 процесса ПР1. Аналогично горизонтальные линии, пронумерованные от 5 до 8, соответствуют операторам 5 – 8 программы ПР2. Точка на плоскости определяет состояние вычислений в некоторый момент времени. Так, точка А соответствует ситуации, при которой ПР1 начал исполнение, но не достиг оператора 1, а ПР2 выполнил оператор 6, но не дошел до оператора 7. По мере выполнения точка будет двигаться горизонтально вправо, если исполняется ПР1, и вертикально вверх, если исполняется ПР2.

Интервалы исполнения, во время которых ресурсы R1 и R2 используются каждым процессом, показаны с помощью фигурных скобок. Линии 1 – 8 делят пространство вычислений на 25 прямоугольников, каждый из которых задаёт состояние вычислений. Закрашенные серым цветом состояния являются недостижимыми из-за взаимного исключения ПР1 и ПР2 при доступе к ресурсам R1 и R2.

Рассмотрим последовательность исполнения 1–2–5–3–6–4–7–8, представленную траекторией Т1. Когда процесс ПР2 запрашивает ресурс R1 (оператор 5), ресурс недоступен (оператор выполнен, семафор закрыт). Поэтому процесс ПР2 заблокирован в точке В. Как только процесс ПР1 достигнет оператора 3, процесс ПР2 деблокируется по ресурсу R1. Аналогично в точке С процесс ПР2 будет заблокирован при попытке доступа к ресурсу R2 (оператор 6). Как только процесс ПР1 достигнет оператора 4, процесс ПР2 деблокируется по ресурсу R2.

Если же, например, выполняется последовательность 1–5–2–6, то процесс ПР1 заблокируется в точке X при выполнении оператора 2, а процесс ПР2 заблокируется в точке Y при выполнении оператора 6. При этом процесс ПР1 ждёт, когда процесс ПР2 выполнит оператор 7, а ПР2 ждёт, когда ПР1 выполнит оператор 4. Оба процесса будут находиться в тупике, ни ПР1, ни ПР2 не могут закончить выполнение. При этом все ресурсы, которые получили ПР1 и ПР2, становятся недоступны-

ми для других процессов, что резко снижает возможности вычислительной системы по обслуживанию их. Отметим одно очень важное обстоятельство: тупик будет неизбежным, если вычисления зашли в прямоугольник D, являющийся критическим состоянием.

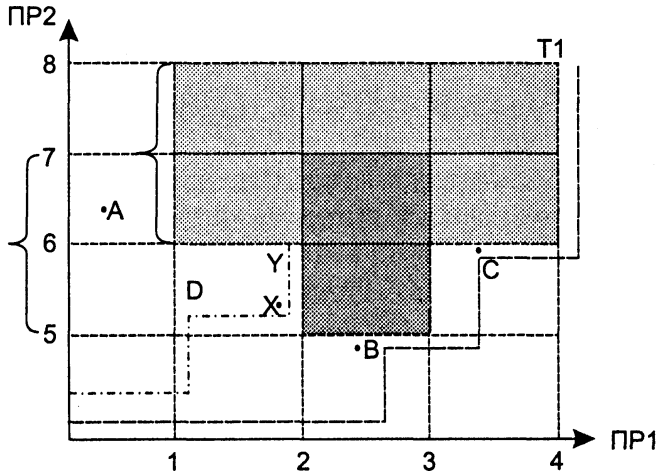

Рис. 7.4. Пространство состояний системы двух параллельных конкурирующих процессов

Для того чтобы возник тупик, необходимо, чтобы одновременно выполнялись четыре условия [37, 92]:

♦ взаимного исключения, при котором процессы осуществляют монопольный доступ к ресурсам;

♦ ожидания, при котором процесс, запросивший ресурс, ждёт до тех пор, пока запрос не будет удовлетворен, при этом удерживая ранее полученные ресурсы;

♦ отсутствия перераспределения, при котором ресурсы нельзя отобрать у процесса, если они ему уже выделены;

♦ кругового ожидания, при котором существует замкнутая цепь процессов, каждый из которых ждёт ресурс, удерживаемый его предшественником в этой цепи.

Проанализировав содержательный смысл этих четырех условий, легко убедиться, что все они выполняются в точке Y (см. рис. 7.4).

Формальные модели для изучения проблемы тупиковых ситуаций

Проблема борьбы с тупиками становится всё более актуальной и сложной по мере развития и внедрения параллельных вычислительных систем. При проектировании таких систем разработчики стараются проанализировать возможные неприятные ситуации, используя специальные модели и методы.

Таких моделей много; к настоящему времени разработано несколько десятков различных моделей, предназначенных для анализа и моделирования систем с параллельными асинхронными процессами, для которых возможность возникновения тупиковых ситуаций является очень серьёзной проблемой. Изложение и сравнительный анализ этих моделей может составить большую монографию, поэтому здесь мы кратко рассмотрим только четыре из них – сети Петри, вычислительные схемы, модель пространства состояний и уже упомянутую нами модель Холта.

Сети Петри

Среди многих существующих методов описания и анализа параллельных систем уже более 35 лет значительное место занимают сетевые модели, восходящие к сетям специального вида, предложенных в 1962 году Карлом Петри для моделирования асинхронных информационных потоков в системах преобразования данных [64].

Взаимодействие событий в параллельных асинхронных дискретных системах имеет, как правило, сложную динамическую структуру. Эти взаимодействия описываются более просто, если указывать не непосредственные связи между событиями, а те ситуации, при которых данное событие может реализоваться. При этом глобальные ситуации в системе формируются с помощью локальных операций, называемых условиями реализации событий. Определённые сочетания условий разрешают реализоваться некоторому событию (предусловия события), а реализация события изменяет некоторые условия (постусловия события), то есть события взаимодействуют с условиями, а условия – с событиями. Таким образом, предполагается, что для решения задач достаточно представить системы как структуры, образованные из элементов двух типов – событий и условий. Удобный формальный

механизм для этого, предложенный Петри, был развит А. Холтом, который назвал его сетью Петри.

В сетях Петри события и условия представлены абстрактными символами из двух непересекающихся алфавитов, называемых соответственно множеством переходов и множеством позиций. Имеется несколько формальных представлений сетей Петри:

♦ теоретико-множественное;

♦ графовое – бихроматический (двудольный ориентированный) граф и, соответственно, графическое;

♦ матричное.

При использовании теоретико-множественного подхода к описанию сети Петри (поскольку эта модель представляет и структуру, и функционирование системы) она формально может быть определена как двойка вида: $N = \langle S, M_0 \rangle$. Здесь S – это структура сети, которая представляется двудольным ориентированным мультиграфом $S=(V, U)$, где V – вершины этого графа, U – его дуги. M_0 – начальное состояние сети Петри, которое также называется начальной маркировкой. В силу того, что граф S является двудольным, можно перейти к формальному описанию структуры сети Петри в виде тройки:

$S\langle P, T, Y \rangle$,

где P – конечное множество позиций, $P = \{p_i\}, i = \overline{1,n}$; T – конечное множество переходов, $T = \{t_j\}, j = \overline{1,m}$; $T \cup P = V$, $T \cap P = \varnothing$, то есть T и P – это два типа вершин биграфа S; Y – конечное множество дуг, заданное отношениями между вершинами графа S : $Y \in (P * T) \cup (T * P)$.

Поскольку двудольный мультиграф S является ориентированным, то любой переход t_j, $j = \overline{1,m}$ соединяется с позициями $p_i \in P$ через входные и выходные дуги, которые задаются через функцию предшествования $B:(P * T) \rightarrow \{0,1, 2,...\}$ и через функцию следования $E:(T * P) \rightarrow \{0,1,2..\}$, являющиеся отображениями из множества переходов в комплекты позиций [64] (синонимом термина комплект является понятие мультимножества). Эти функции определяют комплекты позиций $\{p_i\} \in \overline{P}$,

связанных с переходом $t_j \in T$ через множество дуг $\{(p_i,t_j)_l\}$, где $l \le |\{(p_i,t_j)_l : i,j = \text{const}\}| \le W$, и комплекты позиций $\{p_k\} \in \overline{P}$, связанных с переходом $t_j \in T$ через множество_дуг $\{(t_j,p_k)_l\}$, где $l \le |\{(t_i,p_k)_l : j,k = \text{const}\}| \le W$. Здесь W – мультичисло графа S; $\overline{P}$ – пространство комплектов, заданное на множестве функциями E и B; $\{(p_i,t_j)_v\}$ – v-я дуга, выходящая из позиции p_i и входящая в переход t_j, $\{(t_i,p_k)_v$ – v-я дуга, выходящая из перехода t_j и входящая в позицию p_k.

Таким образом, теперь структура S сети Петри N может быть представлена как четверка: $S(P,T,B,E)$. Представим множество позиций P как объединение двух пересекающихся множеств: $P=I \cup O$; $I \cap O \neq \varnothing$. Здесь мы через I и O обозначим следующие множества:

$$I = \bigcup_{j=1}^{m} I(t_j); \qquad O = \bigcup_{j=1}^{m} O(t_j),$$

где $I(t_j) = \{p_i : B(p_i,t_j) \ge 1,\ i = \overline{1,n}\},\ j = \overline{1,m}$; $O(t_j) = \{p_k : E(t_j,p_k) \ge 1\ \text{k} = \overline{1,n}\},\ j = \overline{1,m}$;

(p_i,t_j) – дуга с весом $w \le W$, выходящая из вершины p_i и входящая в вершину t_j

(t_j,p_k) – дуга с весом $w \le W$, выходящая из вершины t_j и входящая в вершину p_k

то есть $I(t_j)$ и $O(t_j)$ – комплекты соответственно входных и выходных позиций перехода t_j.

Элементы множества T обычно представляют собой те возможности (возможные ситуации, условия), при которых могут быть реализованы интересующие нас процессы (действия).

Начальная маркировка M_0 (как и текущая маркировка M, которая соответствует некоторому состоянию сети в текущий момент модельного времени) определяется одномерной матрицей (вектором), число компонентов которого равно числу позиций сети n, $n = |P|$, а значение i-го компонента, $1 \le i \le n$, есть натуральное число $m(p_i)$, которое определяет количество маркеров (меток) в позиции p_i то есть

$M_0 = (m_0(p_1), m_0(p_2), \ldots, m_0(p_n))$;

$M = (m(p_1), m(p_2), \ldots, m(p_n))$,

где $m(p_1)$ $m(p_i) \in Z$; Z – множество неотрицательных целых чисел. Маркировку M можно представлять и как множество или комплект с той лишь только разницей,

что отсутствие некоторого элемента в множестве будем обозначать специальным элементом – нулём. В этом случае запись вида $M_i = M_{i-1} – I(t)$ означает разность множеств и такое изменение маркировки, при котором на соответствующих местах вектора M_i будут уменьшенные значения.

Передвижение маркеров по сети осуществляется посредством срабатывания её переходов. Срабатывание возбужденного перехода, являющееся локальным актом, в целом ведёт к изменению маркировки сети, то есть к изменению её состояния. Таким образом, если в сети задано начальное маркирование M_0, при котором хотя бы один переход возбуждён, то в ней начинается движение маркеров, отображающее смену состояний сети. Переход t_j может сработать, если

$p_i \in I(t_j)$: $m(p_i) \geq (p_i , I(t_j)) – w$.

Переход, для которого выполняется это условие, называется *возбуждённым.* Здесь запись вида $\#(p_i, I(t_j))$ означает число появлений позиций p_i во входном комплекте перехода t_j оно, естественно, равно весу w, если вместо мультиграфа рассматривать взвешенный граф. При срабатывании перехода t_j маркировка M_0 изменяется на маркировку M_1 следующим образом: $M_1 = M_0 – I(t_j) + O(t_j)$. Иначе говоря,

$\forall p_i \in P$: $m_i(p_i) – m_0(p_i) - \#(p_i, I(t_j)) + \#(p_i, O(t_j))$.

Из последнего выражения видно, что количество маркеров, которое переход t_j изымает из своих входных позиций, может не равняться количеству маркеров, которое этот переход помещает в свои выходные позиции, так как совсем не обязательно, чтобы число входных дуг перехода равнялось числу его выходных дуг.

В графическом представлении сетей (оно наиболее наглядно и легко интерпретируемо) переходы изображаются вертикальными (или горизонтальными) планками (чёрточками), а позиции – кружками (рис. 7.5). Условия–позиции и события–переходы связаны отношением непосредственной зависимости (непосредственной причинно-следственной связи), которое изображается с помощью направленных дуг, ведущих из позиций в переходы и из переходов в позиции. Позиции, из которых ведут дуги на данный переход, называются его входными позициями, а позиции, на которые ведут дуги из данного перехода, – выходными позициями.

Выполнение условия изображается разметкой соответствующей позиции, а именно помещением числа n или изображением n маркеров (фишек) в то место, где $n > 0$ – ёмкость условия.

Сети Петри могут быть использованы с точки зрения анализа системы на возможность возникновения в ней тупиковых ситуаций. Этот анализ проводится посредством исследования пространства возможных состояний сети Петри. При этом под последним понимается множество возможных маркировок сети. Анализ сетей посредством матричных методов имеет множество проблем, поэтому в основном используется подход, основанный на построении редуцированного до дерева[1] графа возможных маркировок [49]. В таком дереве вершины графа – это состояния (маркировки) сети, а ветви дерева, помеченные соответствующими переходами сети, – это возможные изменения состояний сети, то есть срабатывания её переходов. Если взять любую вершину в таком дереве (за исключением корневой), то путь к этой вершине от корня дерева (путь из начальной маркировки к заданной) будет представлять собой последовательность срабатывания переходов.

Говорят, что переход t_j для разметки M является *живым,* если для всех разметок $M' \in \sigma(M)$ существует последовательность срабатывания переходов, которая приводит к маркировке M' при которой переход t_j может сработать. Сеть Петри называется *живой,* если все её переходы живы; живучая разметка – это разметка, при которой каждый из её переходов сможет запускаться бесконечное число раз. Когда достигнута такая разметка, при которой ни один из переходов не может быть запущен, говорят, что сеть Петри завершилась (достигнута желаемая конечная маркировка) или же зависла (то есть имеет место тупиковая ситуация).

Сети Петри очень удобны для описания процессов синхронизации и альтернатив. Например, семафор может быть представлен входной позицией, связанной с несколькими взаимоисключающими переходами (критическими секциями). Сети Петри позволяют моделировать асинхронность и недетерминизм параллельных независимых событий, параллелизм конвейерного типа, конфликтные взаимодействия между процессами. Сети Петри очень удобны для описания процессов синхро-

[1] Напомним, что деревом в теории графов называют граф, не имеющий циклов.

низации и альтернатив. Например, семафор может быть представлен входной позицией, связанной с несколькими взаимоисключающими переходами (критическими секциями). Говорят, что два перехода конфликтуют, если они взаимно исключают друг друга, то есть они не могут быть оба запущены одновременно. Два перехода, готовые к срабатыванию, находятся в конфликте, если они связаны с общей входной позицией.

В качестве примера рассмотрим рис. 7.5.

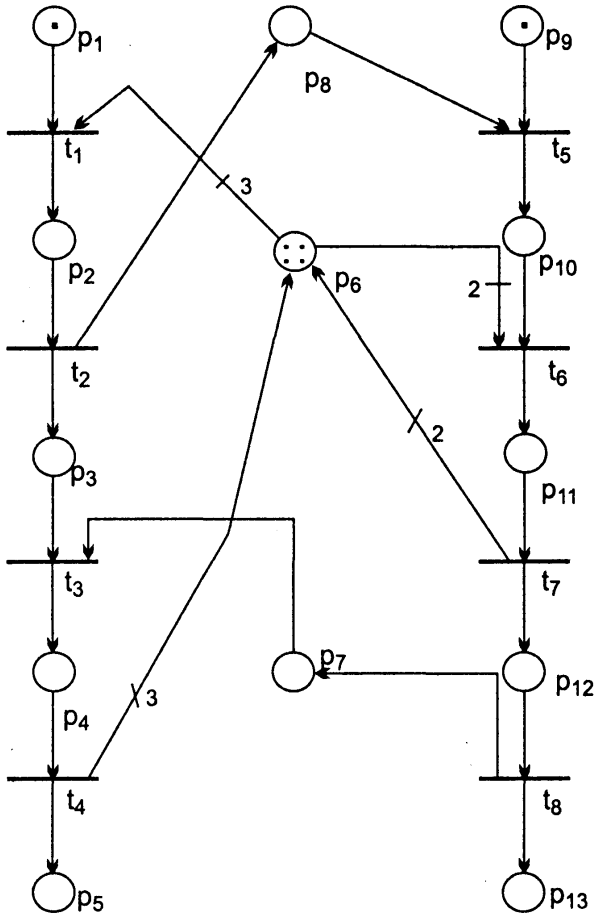

Рис. 7.5. Сеть Петри для системы двух взаимодействующих процессов

Эта сеть соответствует рассмотренному нами ранее примеру тупиковой ситуации (см. рис. 7.2), которая возникает при взаимодействии процессов ПР1 и ПР2 во время передачи сообщений и потреблении ресурса R каждым из процессов. Начальная маркировка для сети, показанной на рис. 7.5, будет равна (1,0,0,0,0,4, 0,0,1,0,0,0,0). Здесь позиция p_2 означает, что процесс ПР1 получил три единицы ресурса R. Дуга, соединяющая позицию p_6 (число маркеров в ней соответствует ко-

личеству доступных единиц ресурса R), имеет вес 3 и при срабатывании перехода t_1 процесс ПР1 получает затребованные 3 единицы ресурса. Переход t_2 представляет посылку процессом ПР1 сообщения для ПР2; переход t_j – приём этого сообщения. Появление маркера в позиции p_7 означает, что процесс ПР2 обработал сообщение и послал ответ процессу ПР1. Срабатывание перехода t_j представляет возврат в систему трёх единиц ресурса, которыми владел процесс ПР1. Рассмотренная сеть не является живой, так как в ней всегда будут мертвы переходы t_3 ,t_j , t_6 , t_7 и t_8.

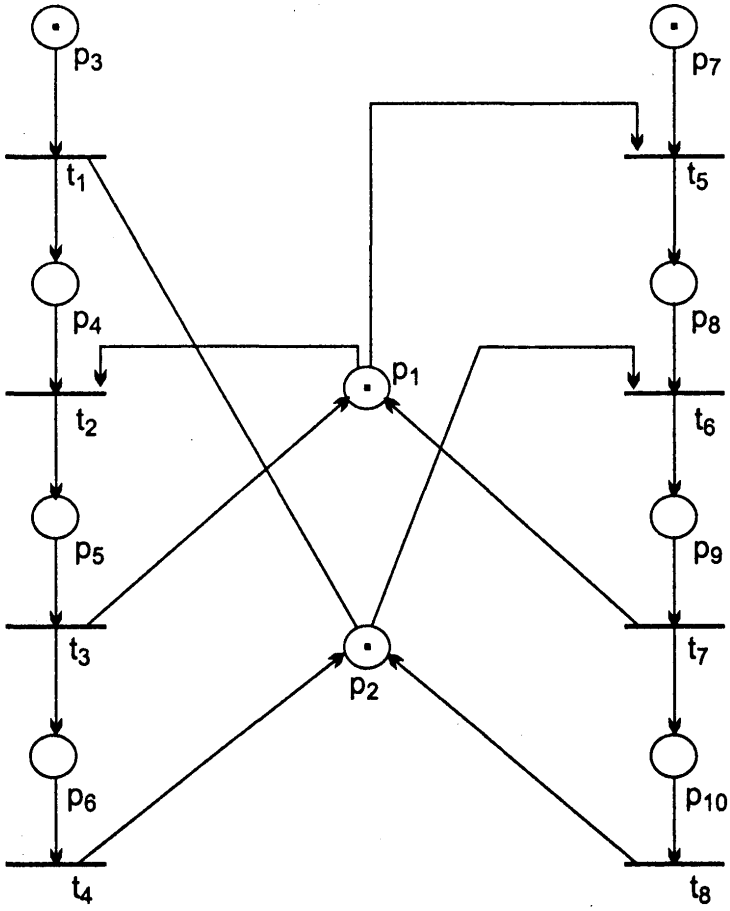

Рис. 7.6. Сеть Петри для тупиковой ситуации на ресурсах типа SR

Примеру тупиковой ситуации, возникающему при работе с ресурсами типа SR, который мы также уже рассматривали ранее (см. рис. 7.3), соответствует сеть Петри, показанная на рис. 7.6.

В этой сети номера переходов соответствуют отмеченным номерам операторов, которые выполняют процессы ПР1 и ПР2, а позиции p_1 и p_2 – семафорам S_1 и S_2, над которыми выполняются P- и V-операции. Сеть на рис. 7.6 также не является

живой, хотя для неё и существуют такие последовательности срабатывания переходов, что тупиковой ситуации не наступит.

Алгоритм построения дерева достижимости изложен, например, в работе [64].

Вычислительные схемы

Вычислительная схема – это представление в графической форме асинхронной системы, состоящей из набора операторов (процессов), которые воздействуют на множество «регистов» (данных). Каждая вычислительная схема определяется с помощью двух графов: графа потока данных и графа управления [89].

Граф потока данных (информационный граф) определяет входные и выходные данные для каждого оператора [18]. Дуга (R_i S_k) от регистра R_i к оператору S_k означает, что данные R_i являются элементом входных данных этого оператора; дуга (S_k R_j) определяет данные R_j как выходные. Очевидно, что некоторые данные R могут являться выходными для оператора S_i и входными для оператора S_j. Пример графа потока данных для некоторой вычислительной схемы представлен на рис. 7.7, *а;* операторы и регистры данных представлены соответственно кружками и прямоугольниками.

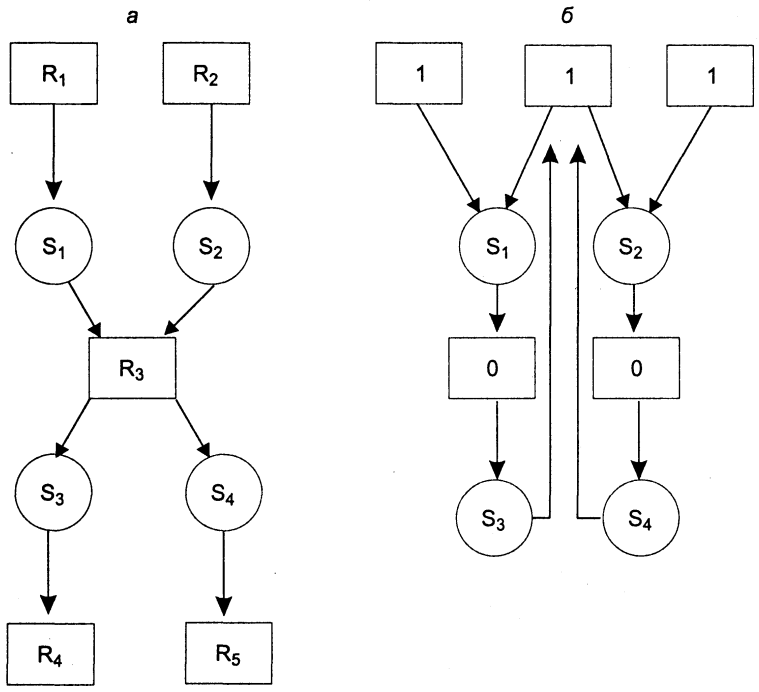

Рис. 7.7. Пример вычислительной схемы: *а* – граф потока данных;
б – граф управления

Граф управления определяет последовательность выполнения операторов. Каждый оператор (представлен кружком) связан с некоторым количеством *управляющих счётчиков* (представлены прямоугольниками). Каждый из управляющих счётчиков содержит неотрицательное целое число. Текущие значения счётчиков совместно со значениями данных на графе потока данных определяют состояние вычислительной схемы. Пример графа управления представлен на рис. 7.7, *б.* Если все счётчики, указывающие на оператор S (то есть входные счётчики), имеют ненулевые значения, то говорят, что оператор S определен. В этом случае он может выполниться, изменив свои выходные регистры в соответствии с графом потока данных и изменив счётчики графа управления по следующему правилу:

значения всех выходных счётчиков оператора S уменьшаются на единицу, а значение выходных – увеличивается, причём для каждого выходного счётчика оператора S приращение может быть своё, персональное, и определяется оно с помощью специальной функции от значений регистров.

Обратите внимание на сходство между графом управления и сетью Петри. Если под операторами и счётчиками понимать переходы и позиции, то единственным существенным различием между этими моделями будет зависимость приращения счётчика от входных данных оператора S.

Такая последовательность операторов S_1, S_2, ... , S_n, ..., что каждый оператор S, определён (то есть его входные счётчики не равны нулю) при тех значениях счётчиков, которые получаются в результате выполнения предшествующих операторов, называется последовательностью исполнения схемы. Поскольку с операторами не связано никакого особого отсчёта времени (подобно сетям Петри[1]), то порядок, в котором операторы будут выполняться, не всегда может быть предсказан. Любая допустимая последовательность исполнения является возможной последовательностью событий. Как мы уже знаем, для системы взаимодействующих параллельных процессов результаты вычислений зависят от последовательно-

[1] Следует заметить, что к настоящему времени появилось большое количество различных модификаций исходной модели, называемой сетью Петри. Многие расширения и модификации базовой модели сетей Петри позволяют учитывать модельное время, дополнительные условия возбуждения и срабатывания переходов, отслеживать атрибуты при маркерах, их изменения и т. д., что позволяет получать более мощные средства моделирования параллельных процессов.

сти исполнения, если не обеспечить взаимное исключение для критических интервалов. В случае, когда вычислительная схема вырабатывает одинаковые результаты для всех допустимых последовательностей исполнения, говорят, что она детерминирована. Схема на рис. 7.7 является детерминированной.

Рассмотрим вычислительную схему на рис. 7.8. Операторы S_1 и S_2 как это видно из графа управления, выполняются параллельно и асинхронно. Очевидно, что значение регистра R_3 будет различным в зависимости от того, выполняется ли оператор S_1 раньше или позже оператора S_2. Поскольку граф управления здесь допускает последовательности исполнения, которые приводят к различным результатам, то эта схема не детерминирована.

Говорят, что два оператора соперничают в регистре R, если один из них изменяет R, а другой либо изменяет R, либо обращается к нему. Если два оператора, которые соперничают в некотором регистре, могут быть выполнены в одно и то же время, то говорят, что в схеме существует условие соперничества и такая схема является недетерминированной. Одна из возможных форм недетерминированного исполнения заключается в том, что схема может «зависнуть» (попасть в тупиковую ситуацию).

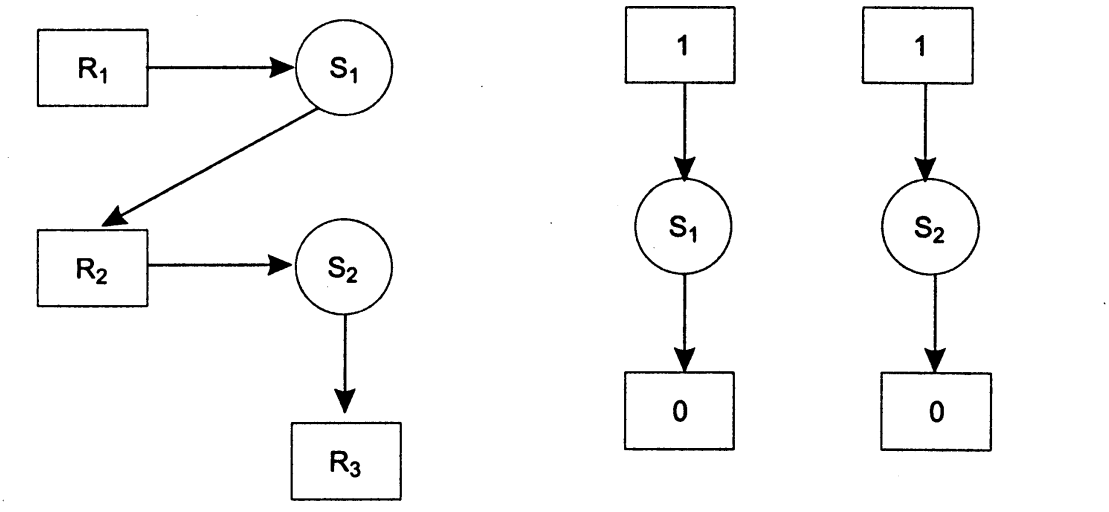

Рис. 7.8. Пример недетерминированной вычислительной схемы

К сожалению, вычислительные схемы, как и сети Петри, не являются конструктивной моделью (с точки зрения борьбы с тупиковыми ситуациями, возникающими в операционных системах), несмотря на свою интуитивную привлекатель-

ность и возможность сделать вывод о возможности существования тупиков в той или иной системе [89]. Мы знаем, что возможность существования тупиковой ситуации в большинстве ОС существует. Но ведь это же не означает, что эти ОС нельзя использовать. Важнее уметь обнаружить существование тупиковой ситуации в конкретный момент времени и поправить ситуацию (насколько это возможно). Поэтому гораздо более продуктивной с этой точки зрения является модель Холта.

Модель пространства состояний системы

Приведём еще одну формальную модель (она подробно рассмотрена в работе [92]). Эта модель очень проста, однако она позволяет сформулировать, что нам нужно делать, чтобы не попасть в тупиковое состояние.

Пусть состояние операционной системы будет сводиться к состоянию различных ресурсов в системе (свободны они или кому-то распределены). Состояние системы изменяется процессами, когда они запрашивают, приобретают или освобождают ресурсы; это будут единственно возможные действия (точнее, мы принимаем во внимание только такие действия). Если процесс не блокирован в данном состоянии, он может изменить это состояние на новое. Однако, так как в общем случае невозможно знать заранее, какой путь может избрать произвольный процесс в своей программе (это неразрешимая проблема!), то новое состояние может быть любым из конечного числа возможных. Следовательно, процессы нами будут трактоваться как недетерминированные объекты. Введённые ограничения на известные понятия приводят нас к следующим формальным определениям:

♦ Система есть пара $<\sigma, \pi>$, где

σ – множество состояний системы $\{S_1, S_2, S_3, \ldots, S_n\}$;

π – множество процессов $\{P_1, P_2, P_3, \ldots, P_k\}$.

♦ Процесс P_i есть частичная функция, отображающая состояние системы в непустые подмножества её же состояний. Это обозначается так:

$P_i: \sigma \rightarrow \{\sigma\}$

Если P_i определён на состоянии S, то область значений P_i , обозначается как $P_i(S)$. Если $S_k \in P_i(S_e)$, то мы говорим, что P_i может изменить состояние S_e в состояние S_k посредством операции, и используем обозначение $S_e \xrightarrow{P_i} S_k$.

Наконец, запись S_e S_w обозначает, что

$S_e = Sw$ или

$S_e \xrightarrow{P_i} S_w$ для некоторого i или

$S_e \xrightarrow{P_i} S_k$ для некоторого *i* и S_k, и $S_k \xrightarrow{\bullet} S_w$.

Другими словами, система может быть переведена посредством $n \geq 0$ операций с помощью $m \geq 0$ различных процессов из состояния S_e в состояние S_w.

Мы говорим, что процесс заблокирован в данном состоянии, если он не может изменить состояние, то есть в этом состоянии процесс не может ни затребовать, ни получать, ни освобождать ресурсы. Поэтому справедливо будет записать следующее:

♦ Процесс P_i заблокирован в состоянии S_e, если не существует ни одного состояния S_k, такого что $S_e \xrightarrow{P_i} S_k$ ($P_i(S_e) = \varnothing$).

Далее, мы говорим, что процесс находится в тупике в данном состоянии S_e, если он заблокирован в состоянии S_e и если вне зависимости от того, какие операции (изменения состояний) произойдут в будущем, процесс остается заблокированным. Поэтому дадим еще одно определение:

♦ Процесс P_i находится в тупике в состоянии S_e, если для всех состояний S_k, таких что $S_e \xrightarrow{\bullet} S_k$, процесс P_i блокирован в S_k.

Приведём пример. Определим систему $<\sigma, \pi>$:

$\sigma = \{S_1, S_2, S_3, S_4\}$; $\pi = \{P_1, P_2\}$;

$P_1(S_1) = \{S_2, S_3\}$; $P_2(S_l) = \{S_3\}$;

$P_l(S_2) = \varnothing$; $P_2(S_2) = \{S_1, S_4\}$;

$P_l(S_3) = \{S_4\}$; $P_2(S_3) = \varnothing$;

$P_1(S_4) = \{S_3\}$; $P_2(S_4) = \varnothing$.

Некоторые возможные последовательности изменений для этой системы таковы:

$S_1 \xrightarrow{P_1} S_3$; $S_2 \xrightarrow{P_2} S_4$; $S_1 \xrightarrow{\bullet} S_4$.

Последовательность $S_1 \xrightarrow{\bullet} S_4$ может быть реализована, например, следующим образом: $S_1 \xrightarrow{P_1} S_2$; $S_2 \xrightarrow{P_2} S_4$ или же $S_1 \xrightarrow{P_1} S_3$; $S_3 \xrightarrow{P_1} S_4$.

Заметим, что процесс P_2 находится в тупике как в состоянии S_3, так и в состоянии S_4; для последнего случая $S_2 \xrightarrow{P_2} S_4$ или $S_2 \xrightarrow{P_2} S_1$ и процесс P_1 не становится заблокированным ни в S_4, ни в S_1.

Диаграмма переходов этой системы изображена на рис. 7.9.

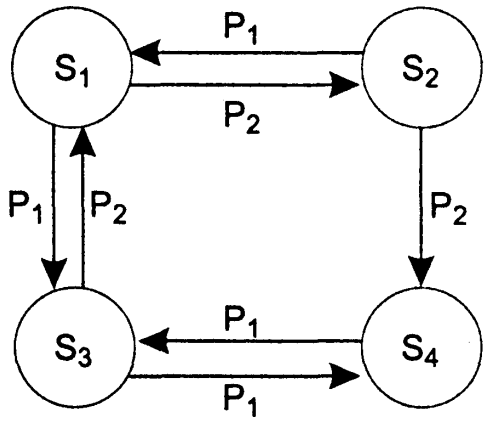

Рис. 7.9. Пример системы <σ, π> на 4 состояния

♦ Состояние S называется тупиковым, если существует процесс P_i, находящийся в тупике в состоянии S.

Теперь мы можем сказать, что тупик предотвращается, по определению, при введении такого ограничения на систему, чтобы каждое её возможное состояние не было тупиковым состоянием.

Введем еще одно определение.

♦ Состояние S_i есть *безопасное состояние,* если для всех S_k, таких что

$S_i \xrightarrow{\bullet} S_k$, S_k не является тупиковым состоянием.

Рассмотрим ещё один пример системы <σ, π>. Граф её состояний приведен на рис. 7.10. Здесь состояния S_2 и S_3 являются безопасными; из них система никогда не сможет попасть в тупиковое состояние. Состояния S_1 и S_4 могут привести как к безопасным состояниям, так и к опасному состоянию S_5. Состояние S_6 и S_7 является тупиковым.

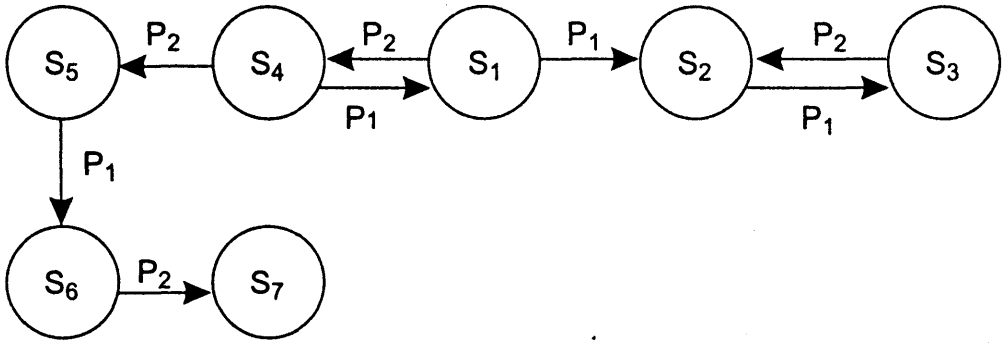

Рис. 7.10. Пример системы <σ, π> с безопасными, опасными и тупиковым состояниями

Наконец, в качестве ещё одной простейшей системы вида <σ, π> приведём пример тупика с SR-ресурсами, уже рассмотренный нами в этой главе ранее. Определим следующим образом состояния процессов P_1 и P_2, которые используют ресурсы R_1 и R_2.

Таблица 7.1. Состояния процессов P_1 и P_2

Состояния для процесса P_1		Состояния для процесса P_2	
0	Не содержит никаких ресурсов	0	Не содержит никаких ресурсов
1	Запросил ресурс R_2, не держит никаких ресурсов	1	Запросил ресурс R_1, не держит никаких ресурсов
2	Держит ресурс R_2	2	Держит ресурс R_1
3	Запросил ресурс R_1, держит ресурс R_2	3	Запросил ресурс R_2, держит ресурс R_1
4	Держит ресурсы R_1 и R_2	4	Держит ресурсы R_1 и R_2
5	Держит ресурс R_2 после освобождения ресурса R_1	5	Держит ресурс R_2 после освобождения ресурса R_1

Пусть состояние системы S_{ij} означает, что процесс P_1 находится в состоянии S_i, а процесс P_2 – в состоянии Sj. Возможные изменения в пространстве состояний этой системы изображены на рис.7.11. «Вертикальными» стрелками показаны возможные переходы из одного состояния в другое для процесса P_1, а «горизонтальными» – для процесса P_2. В данной системе имеются три опасных состояния. Попав в любое из них, мы неминуемо перейдем в тупиковое состояние.

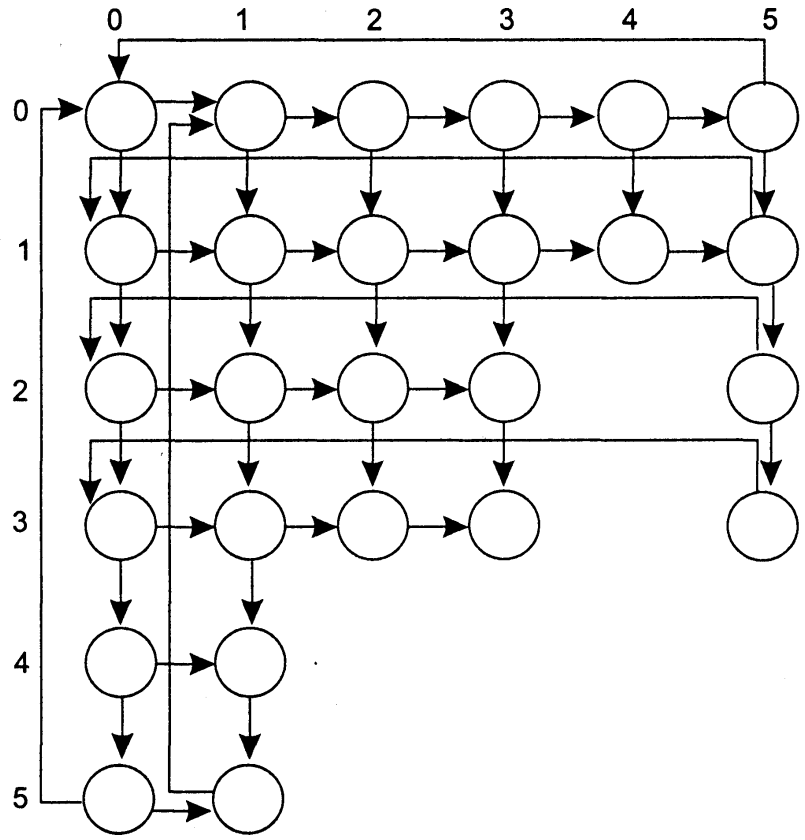

Рис. 7.11. Пример системы

Теперь, когда мы знаем понятия надежного, опасного и безопасного состояний, можно рассмотреть и методы борьбы с тупиками.

Методы борьбы с тупиками

Проблема тупиков является чрезвычайно серьезной и сложной. В настоящее время разработано несколько подходов и методов разрешения этой проблемы, однако, ни один из них нельзя считать панацеей. В некоторых случаях цена, которую приходится платить за то, чтобы сделать систему свободной от тупиков, слишком высока. В других случаях, например в системах управления процессами реального времени, просто нет иного выбора, поскольку возникновение тупика может привести к катастрофическим последствиям.

Проблема борьбы с тупиками становится всё более актуальной и сложной по мере развития и внедрения параллельных вычислительных систем. При проектировании таких систем разработчики стараются проанализировать возможные неприятные ситуации, используя специальные модели и методы. Борьба с тупиковыми ситуациями основывается на одной из трех стратегий:

- предотвращение тупика;
- обход тупика;
- распознавание тупика с последующим восстановлением.

Предотвращение тупиков

Предотвращение тупика основывается на предположении о чрезвычайно высокой его стоимости, поэтому лучше потратить дополнительные ресурсы системы, чтобы исключить вероятность возникновения тупика при любых обстоятельствах. Этот подход используется в наиболее ответственных системах, часто это системы реального времени.

Предотвращение можно рассматривать как запрет существования опасных состояний. Поэтому дисциплина, предотвращающая тупик, должна гарантировать, что какое-либо из четырех условий, необходимых для его наступления, не может возникнуть.

Условие взаимного исключения можно подавить путем разрешения неограниченного разделения ресурсов. Это удобно для повторно входимых программ и ряда драйверов, но совершенно неприемлемо к совместно используемым переменным в критических интервалах.

Условие ожидания можно подавить, предварительно выделяя ресурсы. При этом процесс может начать исполнение, только получив все необходимые ресурсы заранее. Следовательно, общее число затребованных параллельными процессами ресурсов должно быть не больше возможностей системы. Поэтому предварительное выделение может привести к снижению эффективности работы вычислительной системы в целом. Необходимо также отметить, что предварительное выделение зачастую невозможно, так как необходимые ресурсы становятся известны процессу только после начала исполнения.

Условие отсутствия перераспределения можно исключить, позволяя операционной системе отнимать у процесса ресурсы. Для этого в операционной системе должен быть предусмотрен механизм запоминания состояния процесса с целью последующего восстановления. Перераспределение процессора реализуется доста-

точно легко, в то время как перераспределение устройств ввода/вывода крайне нежелательно.

Условие кругового ожидания можно исключить, предотвращая образование цепи запросов. Это можно обеспечить с помощью принципа *иерархического выделения ресурсов.* Все ресурсы образуют некоторую иерархию. Процесс, затребовавший ресурс на одном уровне, может затем потребовать ресурсы только на более высоком уровне. Он может освободить ресурсы на данном уровне только после освобождения всех ресурсов на всех более высоких уровнях. После того как процесс получил, а потом освободил ресурсы данного уровня, он может запросить ресурсы на том же самом уровне. Пусть имеются процессы ПР1 и ПР2, которые могут иметь доступ к ресурсам R1 и R2, причем R2 находится на более высоком уровне иерархии. Если ПР1 захватил R1, то ПР2 не может захватить R2, так как доступ к нему проходит через доступ к R1, который уже захвачен ПР1. Таким образом, создание замкнутой цепи исключается. Иерархическое выделение ресурсов часто не дает никакого выигрыша, если порядок использования ресурсов, определенный в описании процессов, отличается от порядка уровней иерархии. В этом случае ресурсы будут использоваться крайне неэффективно.

В целом стратегия предотвращения тупиков – это очень дорогое решение проблемы, и она используется нечасто.

Обход тупиков

Обход тупика можно интерпретировать как запрет входа в опасное состояние. Если ни одно из упомянутых четырех условий не исключено, то вход в опасное состояние можно предотвратить при наличии у системы информации о последовательности запросов, связанных с каждым параллельным процессом. Доказано [92], что если вычисления находятся в любом неопасном состоянии, то существует по крайней мере одна последовательность состояний, которая обходит опасное. Следовательно, достаточно проверить, не приведет ли выделение затребованного ресурса сразу же к опасному состоянию. Если да, то запрос отклоняется. Если нет, его можно выполнить. Определение того, является ли состояние опасным или нет, требует анализа последующих запросов процессов.

Рассмотрим следующий пример. Пусть имеется система из трех вычислительных процессов, которые потребляют некоторый ресурс R типа SR, который выделяется дискретными взаимозаменяемыми единицами, причем существует всего десять единиц этого ресурса. В табл. 7.2 приведены сведения о текущем распределении процессами этого ресурса R, об их текущих запросах на этот ресурс и о максимальных потребностях процессов в ресурсе R.

Последний столбец в табл. 7.2 показывает, сколько ещё единиц ресурса может затребовать каждый из процессов, если получит ресурс на свой текущий запрос.

Если запрос процесса A будет удовлетворен первым, то он в принципе может запросить еще одну единицу ресурса R, и уже в этом случае мы тогда получим тупиковую ситуацию, поскольку ни один из процессов не сможет продолжить свои вычисления. Следовательно, при выполнении запроса процесса A мы попадаем в *ненадежное*[1] состояние.

Таблица 7.2. Пример распределения ресурсов

Имя процесса	Выделено	Запрос	Максимальная потребность	«Остаток» потребностей
A	2	3	6	1
B	3	2	7	2
C	2	3	5	0

Если первым будет выполнен запрос процесса B, то у нас останется свободной еще одна единица ресурса R. Однако если процесс B запросит еще две, а не одну единицу ресурса R, а он может это сделать, то мы опять получим тупиковую ситуацию.

Если же мы сначала выполним запрос процесса C и выделим ему не две (как у процесса B), а все три единицы ресурса R и у нас при этом даже не останется никакого резерва, то, поскольку на этом его потребности в ресурсах заканчиваются, процесс C сможет благополучно завершиться и вернуть системе все свои ресурсы. Это приведет к тому, что свободное количество ресурса R станет равно пяти. Те-

[1] Термин «ненадежное состояние» не предполагает, что в данный момент существует или в какое-то время обязательно возникнет тупиковая ситуация. Он просто говорит о том, что в случае некоторой неблагоприятной последовательности событий система может зайти в тупик.

перь уже можно будет выполнить запрос либо процесса B, либо процесса A, но не обоих сразу.

Часто бывает так, что последовательность запросов, связанных с каждым процессом, неизвестна заранее. Но если заранее известен общий запрос на ресурсы каждого типа, то выделение ресурсов можно контролировать. В этом случае необходимо для каждого требования, предполагая, что оно удовлетворено, определить, существует ли среди общих запросов от всех процессов некоторая последовательность требований, которая может привести к опасному состоянию. Данный подход является примером контролируемого выделения ресурса.

Классическое решение этой задачи известно как *алгоритм банкира* Дейкстры [89]. Алгоритм банкира напоминает процедуру принятия решения, может ли банк безопасно для себя дать взаймы денег. Принятие решения основывается на информации о потребностях клиента (текущих и максимально возможных) и учёте текущего баланса банка. Несмотря на то, что этот алгоритм нигде практически не используется, рассмотрим его, так как он интересен с методической и академической точек зрения. Текст программы алгоритма банкира приведен в листинге 7.4.

Пусть существует N процессов, для каждого из которых известно максимальное количество потребностей в некотором ресурсе R (обозначим эти потребности через Max(i)). Ресурсы выделяются не сразу все, а в соответствии с текущим запросом. Считается, что все ресурсы i-го процесса будут освобождены по его завершении. Количество полученных ресурсов для i-го процесса обозначим Получ(i). Остаток в потребностях i-го процесса на ресурс R обозначим через Остаток(i). Признак того, что процесс может не завершиться – это значение false для переменной Заверш(i). Наконец, переменная Своб_рес будет означать количество свободных единиц ресурса R, а максимальное количество ресурсов в системе определено значением Всего_рес.

Листинг 7.4. Алгоритм банкира Дейкстры

```
Begin
   Своб_рес := Всего_рес;
   For i := 1 to N do
      Begin
```

```
            Своб_рес := Своб_рес – Получ(i);
            Остаток(i) := Max(i) – Получ(i);
            Заверш(i) := false:     { процесс может не завершиться }
         end
      flag := true;     ( признак продолжения анализа }
      while flag do
         begin
            flag := false;
            for i := 1 to N do
               begin
                  if ( not ( За верш(i) )) and ( Остаток(i) <= Своб_рес )
                     then begin
                              Заверш(i) := true;
                              Своб_рес := Своб_рес + Получ(i);
                              Flag := true
                          end
               end
         end;
      If Своб_рес = Всего_рес
         then     Состояние системы безопасное
                  и можно выдать ресурс
         else     Состояние не безопасное
                  и выдавать ресурс нельзя
   end.
```

Каждый раз, когда какой-то остаток может быть выделен из числа остающихся незанятыми ресурсов, предполагается, что соответствующий процесс работает, пока не окончится, а затем его ресурсы освобождаются. Если, в конце концов, все ресурсы освобождаются, значит, все процессы могут завершиться и система находится в безопасном состоянии. Другими словами, согласно алгоритму банкира система удовлетворяет только те запросы, при которых её состояние остается надёжным. Новое состояние безопасно тогда и только тогда, когда каждый процесс все же может окончиться. Именно это условие и проверяется в алгоритме банкира. Запросы процессов, приводящие к переходу системы в ненадёжное состояние, не выполняются и откладываются до момента, когда его всё же можно будет выполнить.

Алгоритм банкира позволяет продолжать выполнение таких процессов, которым в случае системы с предотвращением тупиков пришлось бы ждать. Хотя алгоритм банкира относительно прост, его реализация может обойтись довольно дорого. Основным накладным расходом стратегии обхода тупика с помощью контролируемого выделения ресурса является время выполнения алгоритма, так как он выполняется при каждом запросе. Причем алгоритм работает медленнее всего, когда система близка к тупику. Необходимо отметить, что обход тупика неприменим при отсутствии информации о требованиях процессов на ресурсы.

Рассмотренный алгоритм примитивен, в нём учитывается только один вид ресурса, тогда как в реальных системах количество различных типов ресурсов бывает очень большим. Были опубликованы варианты этого алгоритма для большого числа различных типов системных ресурсов. Однако все равно алгоритм не получил распространения. Причин тому несколько:

♦ Алгоритм исходит из того, что количество распределяемых ресурсов в системе фиксировано, постоянно. Иногда это не так, например, вследствие неисправности отдельных устройств.

♦ Алгоритм требует, чтобы пользователи заранее указывали свои максимальные потребности в ресурсах. Это чрезвычайно трудно реализовать. Часть таких сведений, конечно, могла бы подготавливать система программирования, но всё равно часть информации о потребностях в ресурсах должны давать пользователи. Однако, поскольку компьютеры становятся всё более дружественными по отношению к пользователям, всё чаще встречаются пользователи, которые не имеют ни малейшего представления о том, какие ресурсы им потребуются.

♦ Алгоритм требует, чтобы число работающих процессов оставалось постоянным. Это возможно только для очень редких случаев. Очевидно, что выполнение этого требования в общем случае не реально, особенно в мультитерминальных системах либо если пользователь может запускать по несколько процессов параллельно.

Обнаружение тупика

Чтобы распознать тупиковое состояние, необходимо для каждого процесса определить, сможет ли он когда-либо снова развиваться, то есть изменять свои состояния. Так как нас интересует возможность развития процесса, а не сам процесс смены состояния, то достаточно рассмотреть только самые благоприятные изменения состояния.

Очевидно, что незаблокированный процесс (он только что получил ресурс и поэтому не заблокирован) через некоторое время освобождает все свои ресурсы и затем благополучно завершается. Освобождение ранее занятых ресурсов может «разбудить» некоторые ранее заблокированные процессы, которые, в свою очередь, развиваясь, смогут освободить другие ранее занятые ресурсы. Это может продолжаться до тех пор, пока либо не останется незаблокированных процессов, либо какое-то количество процессов всё же останется заблокированными. В последнем случае (если существуют заблокированные процессы при завершении указанной последовательности действий) начальное состояние S является состоянием тупика, а оставшиеся процессы находятся в тупике в состоянии S. В противном случае S не есть состояние тупика.

Обнаружение тупика посредством редукции графа повторно используемых ресурсов

Наиболее благоприятные условия для незаблокированного процесса P_i могут быть представлены *редукцией* (сокращением) графа повторно используемых ресурсов (см. первый раздел данной главы, описание модели Холта). Редукция графа может быть описана следующим образом:

♦ Граф повторно используемых ресурсов сокращается процессом Pp который не является ни заблокированной, ни изолированной вершиной, с помощью удаления всех ребер, входящих в вершину P_i и выходящих из P_i. Эта процедура является эквивалентной приобретению процессом P_i неких ресурсов, на которые он ранее выдавал запросы, а затем освобождению всех его ресурсов. Тогда P_i становится изолированной вершиной.

♦ Граф повторно используемых ресурсов несокращаем (не редуцируется), если он не может быть сокращен ни одним процессом.

♦ Граф ресурсов типа RS является полностью сокращаемым, если существует последовательность сокращений, которая удаляет все дуги графа.

Приведем лемму, которая позволяет предложить алгоритмы обнаружения тупика.

Лемма. Для ресурсов типа SR порядок сокращений несуществен; все последовательности ведут к одному и тому же несокращаемому графу.

Доказательство. Допустим, что лемма неверна. Тогда должно существовать некоторое состояние S, которое сокращается до некоторого несокращаемого состояния S_1 с помощью последовательности seq_1 и до несокращаемого состояния S_2 – с помощью последовательности seq_2 так, что $S_1 \neq S_2$ (то есть все процессы в состояниях S_1 и S_2 или заблокированы, или изолированы).

Если сделать такое предположение, то мы приходим к противоречию, которое устраняется только при условии, что $S_1 = S_2$. Действительно, предположим, что последовательность seq_1 состоит из упорядоченного списка процессов (P_1, P_2, ... , P_k). Тогда последовательность seq_1 должна содержать процесс P, который не содержится в последовательности seq_2. В противном случае $S_1 = S_2$, потому что редукция графа только удаляет дуги, уже существующие в состоянии S, а если последовательности seq_1 и seq_2 содержат одно и то же множество процессов (пусть и в различном порядке), то должно быть удалено одно и то же множество дуг. И доказательство по индукции покажет, что $P \neq P_i$ (i = 1,2,..., k) приводит к указанному нами противоречию.

♦ $P \neq P_1$, так как вершина S может быть редуцирована процессом P_1, а состояние S_2 должно, следовательно, также содержать процесс P_1.

♦ Пусть $P \neq P_i$, (i = 1, 2, ... , j). Однако, поскольку после редукции процессами P_i (i = 1, 2, ... , j) возможно ещё сокращение графа процессом P_{j+1}, это же самое должно быть справедливо и для последовательности seq_2 независимо от порядка следования процессов. То же самое множество рёбер графа удаляется с помощью процесса P_i. Таким образом, $P \neq P_{j+1}$

Следовательно, $P \neq P_i$ для i = 1, 2, ..., k и P не может существовать, а это противоречит нашему предположению, что $S_1 \neq S_2$. Следовательно, $S_1 = S_2$.

Теорема о тупике. Состояние S есть состояние тупика тогда и только тогда, когда граф повторно используемых ресурсов в состоянии S не является полностью сокращаемым.

Доказательство.

а) Предположим, что состояние S есть состояние тупика и процесс P_i находится в тупике в S. Тогда для всех S_j, таких что $S \xrightarrow{\bullet} S_j$ процесс P_i заблокирован в S_j (по определению). Так как сокращения графа идентичны для серии операций процессов, то конечное несокращаемое состояние в последовательности сокращений должно оставить процесс P_i блокированным. Следовательно, граф не является полностью сокращаемым.

б) Предположим, что S не является полностью сокращаемым. Тогда существует процесс P_i, который остаётся заблокированным при всех возможных последовательностях операций редукции в соответствие с леммой. Так как любая последовательность операций редукции графа повторно используемых ресурсов, оканчивающаяся несокращаемым состоянием, гарантирует, что все ресурсы типа SR, которые могут когда-либо стать доступными, в действительности освобождены, то процесс P_i навсегда заблокирован и, следовательно, находится в тупике.

Следствие 1. Процесс P_i не находится в тупике тогда и только тогда, когда серия сокращений приводит к состоянию, в котором P_i, не заблокирован.

Следствие 2. Если S есть состояние тупика (по ресурсам типа SR), то по крайней мере два процесса находятся в тупике в S.

Из теоремы о тупике непосредственно следует и алгоритм обнаружения тупиков. Нужно просто попытаться сократить граф по возможности эффективным способом; если граф полностью не сокращается, то начальное состояние было состоянием тупика для тех процессов, вершины которых остались в несокращенном графе. Рассмотренная нами лемма позволяет удобным образом упорядочивать сокращения.

Граф повторно используемых ресурсов может быть представлен или матрицами, или списками. В обоих случаях экономия памяти может быть достигнута использованием взвешенных ориентированных мультиграфов (слиянием определён-

ных дуг получения или дуг запроса между конкретным ресурсом и данным процессом в одну дугу с соответствующим весом, определяющим количество единиц ресурса).

Рассмотрим вариант с матричным представлением. Поскольку граф двудольный, он представляется двумя матрицами размером n×m. Одна матрица – *матрица распределения* D = $||d_{ij}||$, в которой элемент d_{ij} отражает количество единиц R_j ресурса, распределенного процессу P_i то есть $d_{ij} = |(Rj, P;)|$, где (R_j, P_i) – это дуга между вершинами R_j и P_i, ведущая из R_j в P_i. Вторая матрица – *матрица запросов* N = $||n_{ij}||$, где $n_{ij} = |(P_i , R_j)|$.

В случае использования связанных списков для отображения той же структуры можно построить две группы списков. Ресурсы, распределенные некоторому процессу P_i, связаны с P_i указателями:

$P_i \rightarrow (R_x, d_x) \rightarrow (R_y, d_y) \rightarrow ... \rightarrow (R_z, d_z)$, где R_j – вершина, представляющая ресурс, и d_j – вес дуги, то есть $d_j = |(R_j, P_i)|$.

Подобным образом и ресурсы, запрошенные процессом P_i , связаны вместе.

Аналогичные списки создаются и для ресурсов (списки распределенных и запрошенных ресурсов).

$R_i \rightarrow (P_u, n_u) \rightarrow (P_v, n_v) \rightarrow ... \rightarrow (P_w, n_w)$, где $n_j = |(P_j, R_j)|$.

Для обоих представлений удобно также иметь одномерный массив доступных единиц ресурсов $(r_1, r_2, ..., r_m)$, где r_i указывает число доступных (нераспределённых единиц ресурса R_i, то есть $r_i = |Ri| – \Sigma|(R_i , P_k)|$.

Простой метод прямого обнаружения тупика заключается в просмотре по порядку списка (или матрицы) запросов, причём, где возможно, производятся сокращения дуг графа до тех пор, пока нельзя будет сделать более ни одного сокращения. При этом самая плохая ситуация возникает, когда процессы упорядочены в некоторой последовательности $P_1, P_2, ... , P_n$, а единственно возможным порядком сокращений является обратная последовательность, то есть $P_n, P_{n-1}, ..., P_2, P_1$, а также в случае, когда процесс запрашивает все m ресурсов. Тогда число проверок процессов равно

$n + (n-1) + ... + 1 = n\times(n+1)/2$,

причём каждая проверка требует испытания m ресурсов. Таким образом, время выполнения такого алгоритма в наихудшем случае пропорционально $m \times n^2$.

Более эффективный алгоритм может быть получен за счёт хранения некоторой дополнительной информации о запросах. Для каждой вершины процесса P_i определяется так называемый счётчик *ожиданий* w_i , отображающий количество ресурсов (не число единиц ресурса), которые в какое-то время вызывают блокировку процесса. Кроме этого, можно сохранять для каждого ресурса запросы, упорядоченные по размеру (числу единиц ресурса). Тогда следующий алгоритм сокращений, записанный на псевдокоде, имеет максимальное время выполнения, пропорциональное $m \times n$.

```
For all P ∈ L do
Begin for all Rj э |(Rj, P)| > 0 do
   Begin rj := rj + |(Rj, P)|;
        For all Pi э 0 < |(Pi, Rj)| <= rj do
            Begin wi := wi – 1;
               If wi = 0 then L := L ∪ {Pi}
            End
   End
End
Deadlock := Not (L = {P1, P2, ... , Pn});
```

Здесь L – это текущий список процессов, которые могут выполнять редукцию графа. Можно сказать, что $L := \{P_i \mid w_i = 0\}$. Программа выбирает процесс P из списка L, процесс P сокращает граф, увеличивая число доступных единиц r_j всех ресурсов R_j, распределенных процессу P, обновляет счётчик ожидания w_i каждого процесса P_i который сможет удовлетворить свой запрос на частный ресурс R_j, и добавляет P_i к L, если счётчик ожидания становится нулевым.

Методы обнаружения тупика по наличию замкнутой цепочки запросов

Структура графа обеспечивает простое необходимое (но не достаточное) условие для тупика. Для любого графа $G = \langle X, E \rangle$ и вершины $x \in X$ пусть P(x) обозначает множество вершин, достижимых из вершины x, то есть

$$P(x) = \{ y \mid (x, y) \in E \} \cup \{ z \mid (y, z) \in E \ \& \ y \in P(x) \}.$$

Можно сказать, что в ориентированном графе *потомством вершины x,* которое мы обозначаем как P(x), называется множество всех вершин, в которые ведут пути из x.

Тогда если существует некоторая вершина $x \in X : x \in P(x)$, то в графе G имеется цикл.

Теорема 1. Цикл в графе повторно используемых ресурсов является необходимым условием тупика.

Для доказательства этой теоремы (которое мы опустим, указав, что при желании его можно найти в работе [45]) можно воспользоваться следующим свойством ориентированных графов: если ориентированный граф не содержит цикла, то существует линейное упорядочение вершин, такое, что если существует путь от вершины i к вершине j, то i появляется перед j в этом упорядочении.

Теорема 2. Если S не является состоянием тупика и $S \xrightarrow{P_i} S_T$, где S_T есть состояние тупика в том и только в том случае, когда операция процесса P_i есть запрос и P_i находится в тупике в S_T.

Это следует понимать таким образом, что тупик может быть вызван только при запросе, который не удовлетворён немедленно. Учитывая эту теорему, можно сделать вывод, что проверка на тупиковое состояние может быть выполнена более эффективно, если она проводится непрерывно, то есть по мере развития процессов. Тогда надо применять редукцию графа только после запроса от некоторого процесса P_i и на любой стадии необходимо сначала попытаться сократить с помощью процесса P_i Если процесс P_i смог провести сокращение графа, то никакие дальнейшие сокращения не являются необходимыми.

Ограничения, накладываемые на распределители, на число ресурсов, запрошенных одновременно, и количество единиц ресурсов, приводят к более простым условиям для тупика.

Пучок (или *узел)* в ориентированном графе G = <X, E> – это подмножество вершин $Z \subseteq X$, таких что $\forall x \in Z$, P(x) = Z, то есть потомством каждой вершины из Z является само множество Z. Каждая вершина в узле достижима из каждой другой

вершины этого узла, и узел есть максимальное подмножество с этим свойством. Поясним сказанное рис. 7.12.

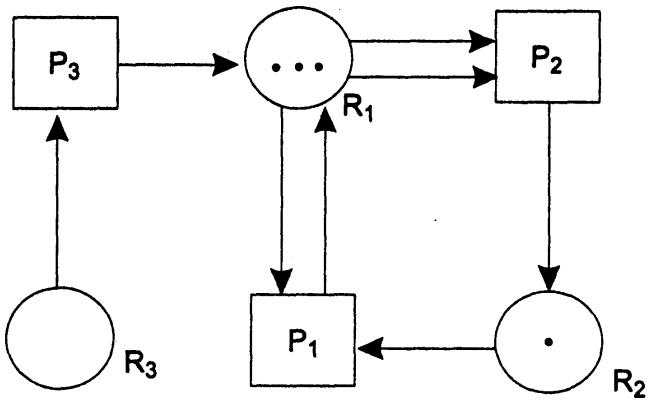

Рис. 7.12. Пример узла на модели Холта

Следует заметить, что наличие цикла – это необходимое, но не достаточное условие для узла. Так, на рис.7.13 изображены два подграфа. Подграф *а* представляет собой пучок (узел), тогда как подграф *б* представляет собой цикл, но узлом не является. В узел должны входить дуги, но они не должны из него выходить.

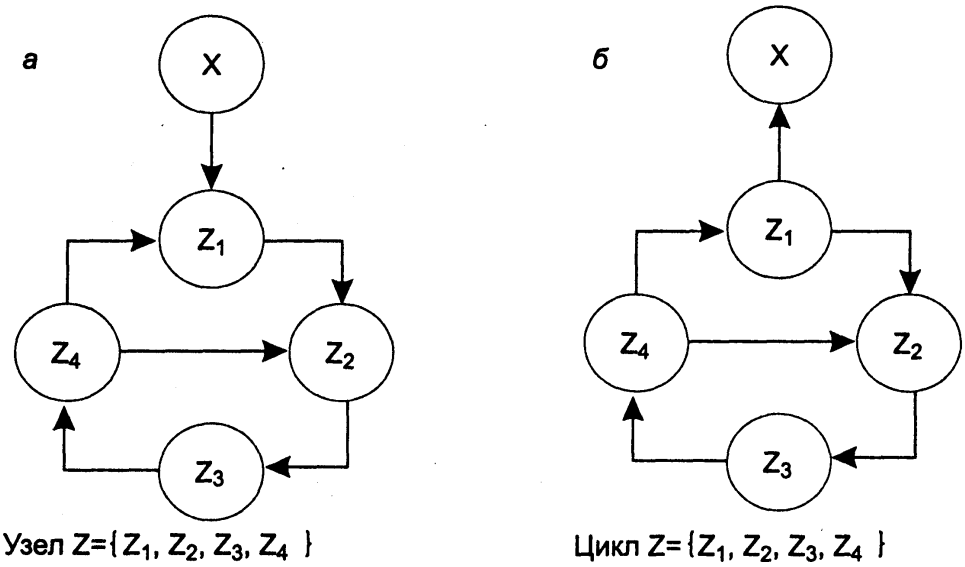

Рис. 7.13. Узел и цикл в ориентированном графе

Если состояние системы таково, что удовлетворены все запросы, которые могут быть удовлетворены, то существует простое достаточное условие существова-

ния тупика. Эта ситуация возникает, если распределители ресурсов не откладывают запросы, которые могут быть удовлетворены, а выполняют их по возможности немедленно (большинство распределителей следует этой дисциплине).

Состояние называется *фиксированным,* если каждый процесс, выдавший запрос, заблокирован.

Теорема 3. Если состояние системы фиксированное (все процессы, имеющие запросы, удовлетворены), то наличие узла в соответствующем графе повторно используемых ресурсов является достаточным условием тупика.

Доказательство. Предположим, что граф содержит узел Z. Тогда все процессы в этом узле должны быть заблокированы только по ресурсам, принадлежащим Z**,** так как никакие ребра не могут выходить из Z по определению. Аналогично, по той же самой причине все распределенные ресурсы узла Z принадлежат процессам из Z. Наконец, все процессы в Z должны быть заблокированы согласно условию фиксированности и определению узла. Следовательно, все процессы в узле Z должны быть в тупике.

Допустим, что каждый ресурс имеет единичную ёмкость (по одной единице ресурса), то есть $|R_j| = 1$, (i = 1, 2, ..., m). При этом ограничении наличие цикла также становится достаточным условием тупика.

Теорема 4. Граф повторно используемых ресурсов с единичной ёмкостью указывает на состояние тупика тогда и только тогда, когда он содержит цикл.

Доказательство. Необходимость цикла доказывает теорема 1. Для доказательства достаточности допустим, что граф содержит цикл, и рассмотрим только лишь процессы и ресурсы, принадлежащие циклу. Так как каждая вершина–процесс должна иметь входящее и исходящее ребро, она должна иметь выданный запрос на некоторый ресурс, принадлежащий циклу, и должна удерживать некоторый ресурс, принадлежащий тому же циклу. Аналогично, каждый ресурс единичной ёмкости в цикле должен быть захвачен некоторым процессом. Следовательно, каждый процесс в цикле блокируется на ресурсе, который может быть освобождён только некоторым процессом из этого цикла; поэтому процессы в цикле находятся в тупике.

Чтобы обнаружить тупик в случае ресурса единичной ёмкости, мы должны просто проверить граф повторно используемых ресурсов на наличие циклов.

Алгоритм обнаружения тупика по наличию замкнутой цепочки запросов

Итак, распознавание тупика может быть основано на анализе модели распределения ресурсов. Один из алгоритмов, основанный на методе обнаружения замкнутой цепи запросов, был разработан сотрудниками фирмы IBM; этот алгоритм использовался в одной из ОС этой компании. Он использует информацию о состоянии системы, содержащуюся в двух таблицах: таблице текущего распределения (назначения) ресурсов RATBL и «таблице» заблокированных процессов PWTBL (для каждого вида ресурса может быть свой список заблокированных процессов). При каждом запросе на получение или освобождении ресурсов содержимое этих таблиц модифицируется, а при запросе – анализируется в соответствии со следующим алгоритмом, который описан по пунктам [49].

1 Запрос от процесса J на занятый ресурс I.

2 Поместить номер ресурса I в PWTBL в строке с номером процесса J.

3 Использовать I в качестве смещения в RATBL, чтобы найти номер процесса K, который владеет ресурсом.

4 Использовать K в качестве смещения в PWTBL.

5 Проверить, ждёт ли процесс K освобождения какого-либо ресурса I’. Если нет, то перейти к шагу 6, в противном случае – к шагу 7.

6 Перевести J в состояние ожидания и выйти из алгоритма.

7 Использовать I’ в качестве смещения в RATBL, чтобы найти номер блокирующего его процесса K'.

8 Проверить K' = J. Если нет, то перейти к шагу 9, в противном случае – к шагу 11.

9 Проверить, вся ли таблица PWTBL просмотрена. Если да, то перейти к шагу 6, в противном случае – к шагу 10.

10 Присвоить K:= K' и перейти к шагу 4.

11 Вывод о наличии тупика с последующим восстановлением.

12 Конец алгоритма.

Для иллюстрации описанного алгоритма распознавания тупика рассмотрим следующую последовательность событий:

1 Процесс P1 занимает ресурс R1.

2 Процесс P2 занимает ресурс R3.

3 Процесс P3 занимает ресурс R2.

4 Процесс P2 занимает ресурс R4.

5 Процесс P1 занимает ресурс R5.

В результате таблица распределения ресурсов (RATBL) имеет следующий вид:

Таблица 7.3. Таблица распределения ресурсов RATBL

Ресурсы	Процессы
1	1
2	3
3	2
4	2
5	1

1 Пусть процесс P1 пытается занять ресурс R3, поэтому в соответствии с описанным алгоритмом J = 1, I = 3, К = 2. Процесс К не ждёт никакого ресурса I’, поэтому процесс P1 блокируется по ресурсу R3.

2 Далее, пусть процесс P2 пытается занять ресурс R2.

J = 3, I = 2, К = 3.

Процесс К не ждёт никакого ресурса, поэтому процесс P2 блокируется по ресурсу R2.

3 И наконец, пусть процесс P3 пытается обратиться к ресурсу R5.

J = 3, I = 5, К = 1, I’ = 3, К’ = 2, К'< > J, поэтому берём К = 2, I’ = 2, К' = 3.

В этом случае K' = J, то есть тупик определён. Таблица заблокированных процессов (PWTBL) теперь имеет следующий вид.

Таблица 7.4, Таблица заблокированных процессов PWTBL

Процесс	Ресурс
1	3
2	2
3	5

Равенство J = K' означает, что существует замкнутая цепь взаимоисключающих и ожидающих процессов, то есть выполняются все четыре условия существования тупика.

Для описанного нами примера можно построить модель Холта; её изображение приведено на рис. 7.14. На этом рисунке пронумерованы дуги запросов, которые процессы последовательно генерировали в соответствии с нашим примером. Из рисунка сразу видно, что в результате такой последовательности запросов образовалась замкнутая цепочка: (8, 5, 6, 2, 7, 3), что и говорит о существовании тупика.

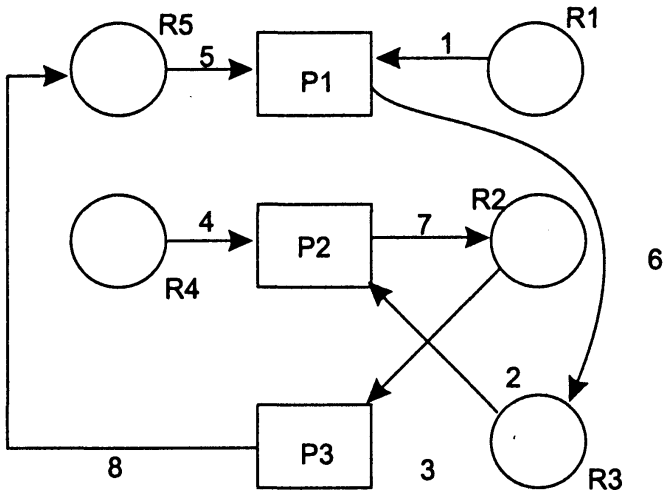

Рис. 7.14. Граф распределения ресурсов

Распознавание тупика требует дальнейшего восстановления.

Восстановление можно интерпретировать как запрет постоянного пребывания в опасном состоянии. Существуют следующие методы восстановления:

♦ принудительный перезапуск системы, характеризующийся потерей информации обо всех процессах, существовавших до перезапуска;

♦ принудительное завершение процессов, находящихся в тупике;

♦ принудительное последовательное завершение процессов, находящихся в тупике, с последующим вызовом алгоритма распознавания после каждого завершения до исчезновения тупика;

♦ перезапуск процессов, находящихся в тупике, с некоторой контрольной точки, то есть из состояния, предшествовавшего запросу на ресурс;

♦ перераспределение ресурсов с последующим последовательным перезапуском процессов, находящихся в тупике.

Основные издержки восстановления составляют потери времени на повторные вычисления, которые могут оказаться весьма существенными. К сожалению, в ряде случаев восстановление может стать невозможным: например, исходные данные, поступившие с каких-либо датчиков, могут уже измениться, а предыдущие значения будут безвозвратно потеряны.

Контрольные вопросы и задачи

Вопросы для проверки

1 Что такое тупиковое состояние? Перечислите условия, при которых возникает тупик.

2 Что является причиной возникновения тупиков на SR–ресурсах?

3 Приведите пример графа повторно используемых ресурсов. Что позволяет сделать эта модель Холта?

4 Приведите пример теоретико-множественного описания сети Петри.

5 Что такое маркировка сети Петри? Что представляет собой пространство возможных состояний сети Петри?

6 Приведите пример графического представления сети Петри.

7 Что представляет собой «предотвращение тупика»? Как его можно реализовать?

8 Что представляет собой «обход тупика»? Приведите алгоритм банкира Дейкстры.

9 Что такое «опасное состояние»? Приведите пример опасного состояния на модели состояний системы.

10 Изложите метод обнаружения тупика посредством редукции графа повторно–используемых ресурсов.

11 Изложите алгоритм обнаружения тупика по наличию замкнутой цепочки запросов.

ГЛАВА 8 Современные операционные системы

В заключение первой части учебника кратко рассмотрим основные архитектурные особенности современных ОС, которые используются на ПК типа IBM PC.

Прежде всего отметим тот общеизвестный факт, что наиболее популярными являются ОС семейства Windows компании Microsoft. Это и Windows 95/98, и Windows NT, и новое поколение Windows 2000. Однако, поскольку в настоящее время практически вся литература, связанная с программным обеспечением для ПК (в том числе и по системному программному обеспечению), в той или иной степени, прежде всего, касается ОС этой компании, то в данном случае мы сделаем исключение и не будем описывать ОС Windows. Желающие без труда найдут любую литературу по этому вопросу.

Мы же кратко рассмотрим ОС UNIX (не Linux, по которому сейчас тоже появляется немало монографий и учебников, а именно основы UNIX, которые в абсолютном своем большинстве относятся и к Linux), OS/2 (хотя эта система уже практически всеми и забыта, но она была одной из первых, предоставивших пользователям ПК полноценную, мультипрограммную надёжную среду, и имеет много очень интересных и эффективных механизмов) и QNX (как наиболее известный и удачный вариант ОС для реализации систем реального времени).

Семейство операционных систем UNIX

Общая характеристика семейства операционных систем UNIX, особенности архитектуры семейства ОС UNIX

UNIX является примером исключительно удачной реализации простой мультипрограммной и многопользовательской операционной системы. В своё время она проектировалась как инструментальная система для разработки программного обеспечения. Своей уникальностью система UNIX обязана во многом тому обстоятельству, что она была, по сути, создана всего двумя разработчиками[1], причём создававшие её люди делали систему для себя, и первое время её использовали на мини-ЭВМ с очень скромными вычислительными ресурсами. По этой причине UNIX, прежде всего, обладает простым, но очень мощным командным языком и независимой от устройств файловой системой. Поскольку при создании этой ОС использовался язык высокого уровня, на котором пишутся не только системные, но и прикладные программы (речь идет о языке С), то система и приложения, выполняющиеся в ней, получились легко переносимыми (мобильными). Компилятор с языка С для всех оттранслированных программ дает реентерабельный и разделяемый код, что позволяет эффективно использовать имеющиеся в системе ресурсы.

Первой целью при разработке этой системы было стремление сохранить простоту и обойтись минимальным количеством функций. Все реальные сложности оставлялись пользовательским программам.

Второй целью была общность. Одни и те же методы и механизмы должны были использоваться во многих случаях. Поэтому общность в UNIX-системах проявляется во многих аспектах, и в частности:

♦ обращения к файлам, устройствам ввода/вывода и буферам межпроцессных сообщений выполняются с помощью одних и тех же примитивов;

♦ одни и те же механизмы именования, присвоения альтернативных имен и защиты от несанкционированного доступа применяются к файлам с данными и директориями и устройствам;

[1] Создателями системы UNIX считаются Кен Томпсон и Деннис Ритчи.

♦ одни и те же механизмы работают в отношении программно и аппаратно инициируемых прерываний.

Наконец, третья цель заключалась в создании операционной среды, в которой большие задачи можно было бы решать, комбинируя существующие небольшие программы, а не разрабатывая программы заново.

Важным, хотя и простым с позиций его реализации, является тот факт, что система UNIX предоставляет пользователям возможность направить выход одной программы непосредственно на вход другой (речь идет о программных каналах (pipe). См. об этом ниже и в разделе «Конвейеры и очереди сообщений», глава 6). В результате большие программные системы можно создавать путём композиции имеющихся небольших программ, а не путём написания новых, что в большинстве случаев упрощает задачу. UNIX-системы существуют уже 30 лет, и к настоящему времени имеется чрезвычайно большой набор легко переносимых из системы в систему отлично отлаженных и проверенных временем приложений.

UNIX-системы поставляются с большим набором системных и прикладных программ, включающим редакторы текстов, программируемые интерпретаторы командного языка, компиляторы с нескольких популярных языков программирования, включая C, C++, ассемблер, PERL, FORTRAN и многие другие, компоновщики (редакторы межпрограммных связей), отладчики, многочисленные библиотеки системных и пользовательских программ, средства сортировки и ведения баз данных, многочисленные административные и обслуживающие программы. Для абсолютного большинства этих программ имеется документация, включающая в себя такие важные документы, как исходные (как правило, снабженные хорошими комментариями) тексты программ. Кроме этого, описание и документация в большей части доступны пользователю непосредственно за экраном в интерактивном режиме. Используется иерархическая файловая система с полной защитой, работа со съёмными томами, обеспечивается независимость от устройств.

Центральной частью системы UNIX является ядро (kernel).

Основные понятия системы UNIX

Одним из достоинств ОС UNIX является то, что система базируется на небольшом числе понятий; рассмотрим их вкратце. Необходимо заметить, что настоящий учебник не претендует на полноценное изложение основ работы в системе UNIX и детальное описание её архитектуры. На эту тему имеется достаточное количество специальной литературы, например отличная монография А. М. Робачевского [70]. Тем не менее, исходя из учебного плана и нашего опыта преподавания системного программного обеспечения, мы считаем полезным изложить здесь минимальный набор основных понятий, который часто помогает студентам погрузиться в мир UNIX, отличающийся от привычного всем окружения Windows.

Виртуальная машина

Система UNIX – многопользовательская. Каждому пользователю после регистрации (входа в систему) предоставляется виртуальный компьютер, в котором есть все необходимые ресурсы: процессор (процессорное время выделяется на основе «карусельной» диспетчеризации (RR – round robin) и с использованием динамических приоритетов с тем, чтобы обеспечить равенство в обслуживании), память, устройства, файлы. Текущее состояние такого виртуального компьютера, предоставляемого пользователю, называется образом. Можно сказать, что процесс – это выполнение образа. Образ состоит из:

- образа памяти;
- значений общих регистров процессора;
- состояния открытых файлов;
- текущего директория (каталога файлов) и другой информации.

Образ процесса во время его выполнения размещается в основной памяти. В старых версиях системы UNIX образ мог быть выгружен (откачан) на диск, если какому-либо более приоритетному процессу потребуется место в основной памяти. В современных реализациях, поддерживающих, как правило, страничный механизм виртуальной памяти, прежде всего выгружаются неиспользуемые страницы.

Образ памяти делится на три логических сегмента:

♦ сегмент реентерабельных процедур (начинается с нулевого адреса в виртуальном адресном пространстве процесса);

♦ сегмент данных (располагается следом за сегментом процедур и может расти в сторону больших адресов);

♦ сегмент стека (начинается со старшего адреса и растет в сторону младших адресов по мере занесения в него информации при вызовах подпрограмм и при прерываниях).

Пользователь

Мы уже отмечали, что с самого начала ОС UNIX замышлялась как интерактивная многопользовательская система. Другими словами, UNIX предназначен для мультитерминальной работы. Чтобы начать работать, человек должен «войти» в систему, введя со свободного терминала свое учётное имя (account name) и, возможно, пароль (password). Человек, зарегистрированный в учётных файлах системы и, следовательно, имеющий учётную запись (account), называется зарегистрированным пользователем системы. Регистрацию новых пользователей обычно выполняет администратор системы. Пользователь не может изменить своё учётное имя, но может установить и/или изменить свой пароль. Пароли хранятся в отдельном файле в закодированном виде.

Все пользователи ОС UNIX явно или неявно работают с файлами. Файловая система ОС UNIX имеет древовидную структуру [70]. Промежуточными узлами дерева являются каталоги со ссылками на другие каталоги или файлы, а листья дерева соответствуют файлам или пустым каталогам. Каждому зарегистрированному пользователю соответствует некоторый каталог файловой системы, который называется «домашним» (home) каталогом пользователя. При входе в систему пользователь получает неограниченный доступ к своему домашнему каталогу и всем каталогам и файлам, содержащимся в нём. Пользователь может создавать, удалять и модифицировать каталоги и файлы, содержащиеся в домашнем каталоге. Потенциально возможен доступ и ко всем другим файлам, однако он может быть ограничен, если пользователь не имеет достаточных привилегий.

Интерфейс пользователя

Традиционный способ взаимодействия пользователя с системой UNIX основывается на использовании командных языков (правда, поскольку в настоящее время всё большее распространение получают графические интерфейсы, то и в ОС UNIX стали всё чаще работать в X Window). После входа пользователя в систему для него запускается один из командных интерпретаторов (в зависимости от параметров, сохраняемых в файле /etc/passwd). Обычно в системе поддерживается несколько командных интерпретаторов с похожими, но различающимися своими возможностями командными языками. Общее название для любого командного интерпретатора ОС UNIX – shell (оболочка), поскольку любой интерпретатор представляет внешнее окружение ядра системы. Вызванный командный интерпретатор выдает приглашение на ввод пользователем командной строки, которая может содержать простую команду, конвейер команд или последовательность команд. После выполнения очередной командной строки и выдачи на экран терминала или в файл соответствующих результатов, shell снова выдает приглашение на ввод командной строки, и так до тех пор, пока пользователь не завершит свой сеанс работы и не выйдет из системы

Командные языки, используемые в ОС UNIX, достаточно просты, чтобы новые пользователи могли быстро начать работать, и достаточно мощны, чтобы можно было использовать их для написания сложных программ. Последняя возможность опирается на механизм командных файлов (shell scripts), которые могут содержать произвольные последовательности командных строк. При указании имени командного файла вместо очередной команды интерпретатор читает файл строка за строкой и последовательно интерпретирует команды.

Привилегированный пользователь

Ядро ОС UNIX идентифицирует каждого пользователя по его идентификатору (UID – user identifier), уникальному целому значению, присваиваемому пользователю при регистрации в системе. Кроме того, каждый пользователь относится к некоторой группе пользователей, которая также идентифицируется некоторым целым значением (GID – group identifier). Значения UID и GID для каждого зарегистриро-

ванного пользователя сохраняются в учётных файлах системы и приписываются процессу, в котором выполняется командный интерпретатор, запущенный при входе пользователя в систему. Эти значения наследуются каждым новым процессом, запущенным от имени данного пользователя, и используются ядром системы для контроля правомочности доступа к файлам, выполнения программ и т. д.

Очевидно, что администратор системы, который тоже является зарегистрированным пользователем, должен обладать большими возможностями, чем обычные пользователи. В ОС UNIX эта задача решается путём выделения единственного нулевого значения UID. Пользователь с таким UID называется суперпользователем (superuser) или root. Он имеет неограниченные права на доступ к любому файлу и на выполнение любой программы. Кроме того, такой пользователь имеет возможность полного контроля над системой. Он может остановить её и даже разрушить.

Еще одним важным отличием суперпользователя от обычного пользователя ОС UNIX является то, что на суперпользователя не распространяются ограничения на используемые ресурсы. Для обычных пользователей устанавливаются такие ограничения, как максимальный размер файла, максимальное число сегментов разделяемой памяти, максимально допустимое пространство на диске и т. д. Суперпользователь может изменять эти ограничения для других пользователей, но на него они не действуют.

Команды и командный интерпретатор

Оболочкой (shell) в системе UNIX называют механизм взаимодействия между пользователями и системой. По сути дела, это интерпретатор команд, который считывает набираемые пользователем строки и запускает выполнение запрошенных системных функций. Полный командный язык, интерпретируемый оболочкой, богат по возможностям и достаточно сложен, однако большинство команд просты в использовании и запомнить их не составляет труда.

Командная строка состоит из имени команды (то есть имени выполняемого файла), за которым следует список аргументов, разделённых пробелами. Оболочка разбивает командную строку на компоненты. Указанный в команде файл загружается, и ему обеспечивается доступ к заданным в команде аргументам.

Любой командный язык семейства shell фактически состоит из трёх частей:

♦ служебных конструкций, позволяющих манипулировать с текстовыми строками и строить сложные команды на основе простых команд;

♦ встроенных команд, выполняемых непосредственно интерпретатором командного языка;

♦ команд, представляемых отдельными выполняемыми файлами.

В свою очередь, набор команд последнего вида включает стандартные команды (системные утилиты, такие как vi, cc и т. д.) и команды, созданные пользователями системы. Для того чтобы выполняемый файл, разработанный пользователем ОС UNIX, можно было запускать как команду shell, достаточно определить в одном из исходных файлов функцию с именем main (имя main должно быть глобальным, то есть перед ним не должно указываться ключевое слово static). Если употребить в качестве имени команды имя такого выполняемого файла, командный интерпретатор создаст новый процесс и запустит в нём указанную выполняемую программу, начиная с вызова функции main.

Тело функции main, вообще говоря, может быть произвольным (для интерпретатора существенно только наличие входной точки в программу с именем main), но для того, чтобы создать команду, которой можно задавать параметры, нужно придерживаться некоторых стандартных правил. В этом случае каждая функция main должна определяться с двумя параметрами – argc и argv. После вызова команды параметру argc будет соответствовать число символьных строк, указанных в качестве аргументов вызова команды, а argv – массив указателей на переменные, содержащие эти строки. При этом имя самой команды составляет первую строку аргументов (то есть после вызова значение argc всегда больше или равно 1). Код функции main должен проанализировать допустимость заданного значения argc и соответствующим образом обработать заданные текстовые строки.

Например, следующий текст на языке C может быть использован для создания команды, которая выводит на экран текстовую строку, заданную в качестве её аргумента:

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

```
main(argc, argv)
{
 int argc;
 char* argv[ ];
 if(argc!=2)
  {
    printf(“usage: %s your-text\n”, argv[0]);
    exit;
  }
 printf(“%s\n”, argv[1]);
}
```

Процессы

Процесс в ОС UNIX понимается в классическом смысле этого термина, то есть как программа, выполняемая в собственном виртуальном адресном пространстве. Когда пользователь входит в систему, автоматически создается процесс, в котором выполняется программа командного интерпретатора. Если командному интерпретатору встречается команда, соответствующая выполняемому файлу, то он создает новый процесс и запускает в нём соответствующую программу, начиная с функции main. Эта запущенная программа, в свою очередь, может создать процесс и запустить в нём другую программу (она тоже должна содержать функцию main) и т. д.

Для образования нового процесса и запуска в нём программы используются два системных вызова API – fork( ) и exec(имя_выполняемого_файла). Системный вызов fork приводит к созданию нового адресного пространства, состояние которого абсолютно идентично состоянию адресного пространства основного процесса (то есть в нём содержатся те же программы и данные). Для дочернего процесса заводятся копии всех сегментов данных.

Другими словами, сразу после выполнения системного вызова fork основной (родительский) и порожденный процессы являются абсолютными близнецами;

управление и в том и в другом находится в точке, непосредственно следующей за вызовом fork. Чтобы программа могла разобраться, в каком процессе она теперь работает – в основном или порождённом, функция fork возвращает разные значения: 0 в порождённом процессе и целое положительное число (идентификатор порождённого процесса – так называемый PID) в основном процессе.

Теперь, если мы хотим запустить новую программу в порождённом процессе, нужно обратиться к системному вызову exec, указав в качестве аргументов вызова имя файла, содержащего новую выполняемую программу, и, возможно, одну или несколько текстовых строк, которые будут переданы в качестве аргументов функции main новой программы. Выполнение системного вызова exec приводит к тому, что в адресное пространство порожденного процесса загружается новая выполняемая программа и запускается с адреса, соответствующего входу в функцию main. Другими словами, это приводит к замене текущего программного сегмента и текущего сегмента данных, которые были унаследованы при выполнении вызова fork, на новые соответствующие сегменты, заданные в файле. Прежние сегменты теряются. Это эффективный метод смены выполняемой процессом программы, но не самого процесса. Файлы, уже открытые до выполнения примитива exec, остаются открытыми после его выполнения.

В следующем примере пользовательская программа, вызываемая как команда shell, выполняет в отдельном процессе стандартную команду shell ls, которая выдаёт на экран содержимое текущего каталога файлов.

```
main( )
{
 if (fork ( )==(0) wait(0); /* родительский процесс */
  else execl('ls", "ls", 0); /* порождённый процесс */
}
```

Таким образом, с практической точки зрения процесс в UNIX является объектом, создаваемым в результате выполнения функции fork( ). Каждый процесс, за исключением начального (нулевого), порождается в результате запуска другим процессом операции fork( ). Каждый процесс имеет одного родителя, но может породить много процессов. Начальный (нулевой) процесс является особенным процессом, который создается в результате загрузки системы. После порождения нового процесса с идентификатором 1 нулевой процесс становится процессом подкачки и реализует механизм виртуальной памяти. Процесс с идентификатором 1, известный под именем init, является предком любого другого процесса в системе и связан с каждым процессом особым образом.

Функционирование системы UNIX

Теперь, когда мы знаем основные понятия, рассмотрим наиболее характерные моменты функционирования этой системы.

Выполнение процессов

Процесс может выполняться в одном из двух состояний, а именно *пользовательском* и *системном.* В пользовательском состоянии процесс выполняет пользовательскую программу и имеет доступ к пользовательскому сегменту данных. В системном состоянии процесс выполняет программы ядра и имеет доступ к системному сегменту данных.

Когда пользовательскому процессу требуется выполнить системную функцию, он создает системный вызов. Фактически происходит вызов ядра системы как подпрограммы. С момента появления системного вызова процесс считается системным. Таким образом, пользовательский и системный процессы являются двумя фазами одного и того же процесса, но они никогда не пересекаются между собой. Каждая фаза пользуется своим собственным стеком. Стек задачи содержит аргументы, локальные переменные и другую информацию относительно функций, выполняемых в режиме задачи. Диспетчерский процесс не имеет пользовательской фазы.

В UNIX-системах используется разделение времени, то есть каждому процессу выделяется квант времени. Либо процесс завершается сам до истечения отведённого ему кванта времени, либо он откладывается по истечении кванта. Механизм диспетчеризации характеризуется достаточно справедливым распределением процессорного времени между всеми процессами. Пользовательским процессам приписываются приоритеты в зависимости от количества получаемого ими процессорного времени. Процессам, которые получили большое количество процессорного времени, назначают более низкие приоритеты, в то время как процессам, которые получили лишь небольшое количество процессорного времени – наоборот, повышают приоритет. Вспомните рассмотренные ранее механизмы динамических приоритетов (см. подраздел «Диспетчеризация задач с использованием динамических приоритетов», глава 2). Такой метод диспетчеризации обеспечивает хо-

рошее время реакции для всех пользователей системы. Все системные процессы имеют более высокие приоритеты по сравнению с пользовательскими и поэтому всегда обслуживаются в первую очередь.

Подсистема ввода/вывода

Функции ввода/вывода в UNIX задаются в основном с помощью пяти системных вызовов, а именно: open, close, read, write и seek.

Открыть файл можно командой

```
file_descriptor = open (file_name, mode)
```

где параметр mode (режим) указывает, разрешено ли чтение, запись или и то и другое; file_descriptor – дескриптор файла, служит для последующих ссылок на данный файл.

Чтение и запись осуществляется командами следующего вида:

```
after_reading_bytes = read (fne_descriptor, buffer, bytes)
after_writing_bytes = write (file_descriptor, buffer, bytes)
```

где bytes – это число байтов, которые должны быть прочитаны или записаны;

after_reading_bytes и after_writing_bytes – это реально прочитанное и записанное количество байтов соответственно.

При чтении возможны три ситуации, в каждой из которых чтение происходит последовательно:

♦ если это первое чтение из файла, то оно осуществляется последовательно с самого начала файла;

♦ если операции чтения предшествовала другая операция чтения из этого файла, то текущая операция предоставит нам данные, непосредственно следующие за предыдущими;

♦ если предшествовала операция поиска seek (см. далее), то чтение осуществляется последовательно от точки смещения, указанной в операции seek.

Это же справедливо и по отношению к операции записи в файл. Обратите внимание, что все эти вызовы относятся к последовательному доступу и эффект прямой адресации достигается с помощью команды seek, смещающей текущую позицию файла

```
Seek (file_descriptor, displacement, displacement_type).
```

Здесь параметр displacement_type (тип смещения) определяет в команде, является ли смещение абсолютным или относительным, а также задано ли оно числом байтов или числом блоков по 512 байт.

Важно заметить, что команда seek исполняется для магнитных дисков так же, как и для магнитных лент, которые нынче уже практически не используются, но во времена появления и становления UNIX-систем были часто используемым устройством.

Чтобы закрыть файл, достаточно выполнить команду

close (file_descnptor)

Еще три примитива – gtty, stty, stat позволяют получать и задавать информацию о файлах и терминалах.

Те же самые команды ввода/вывода применяются и к физическим устройствам. В системе UNIX физические устройства представлены специальными файлами в единой структуре файловой системы. Это означает, что пользователь не может написать зависящую от устройств программу, если только эта зависимость не отражена в самом потоке передаваемых данных. Стандартные файлы ввода и вывода, приписываемые пользовательскому терминалу, открывать обычным путем не требуется. Терминал открывается автоматически по команде входа в систему – login.

Система ввода/вывода UNIX, в отличие от большинства других систем, ориентирована скорее на работу с потоком, а не с записями. Здесь *поток* (stream) – это последовательность байтов, заканчивающаяся разделителем (то есть символом конца потока end-of-stream). Понятие потока позволяет проще добиться независимости от устройств и унификации файлов с физическими устройствами и транспортерами (конвейерами). Тем самым пользователь получает гибкость в работе с группами данных, но на него ложатся и дополнительные заботы, поскольку ему приходится писать программы управления данными. Пользователь может при необходимости относительно легко самостоятельно реализовать работу с записями. Чтобы работать с записями фиксированной длины, достаточно просто задавать постоянную длину во всех командах чтения и записи. Прямой доступ при фиксиро-

ванной длине записей получается путем умножения длины записи на номер записи и выполнения команды seek для нахождения позиций нужной записи. Работу с записями переменной длины можно организовать, если разместить в начале каждой записи поле фиксированного размера, содержащее длину записи.

Перенаправление ввода/вывода

Механизм перенаправления ввода/вывода является одним из наиболее элегантных, мощных и одновременно простых механизмов ОС UNIX. Цель, которая ставилась при разработке этого механизма, состоит в следующем. Поскольку UNIX – это интерактивная система, которая создавалась в конце 60-х – начале 70-х годов, то обычно программы вводили текстовые строки с терминала и выводили результирующие текстовые строки на экран терминала. Для того чтобы обеспечить более гибкое использование таких программ, желательно было уметь обеспечить им ввод из файла или из вывода других программ и направить их вывод в файл или на ввод другим программам.

Реализация этого механизма основывается на следующих свойствах ОС UNIX. Во-первых, любой ввод/вывод трактуется как ввод из некоторого файла и вывод в некоторый файл. Клавиатура и экран терминала тоже интерпретируются как файлы (первый можно только читать, а во второй можно только писать). Во-вторых, доступ к любому файлу производится через его дескриптор (положительное целое число). Фиксируются три значения дескрипторов файлов. Файл с дескриптором 1 называется файлом стандартного ввода (stdin), файл с дескриптором 2 – файлом стандартного вывода (stdout) и файл с дескриптором 3 – файлом стандартного вывода диагностических сообщений (stderr). В-третьих, программа, запущенная в некотором процессе, «наследует» от породившего процесса все дескрипторы открытых файлов.

В головном процессе интерпретатора командного языка файлом стандартного ввода является клавиатура терминала пользователя, а файлами стандартного вывода и вывода диагностических сообщений – экран терминала. Однако при запуске любой команды можно сообщить интерпретатору (средствами соответствующего командного языка), какой файл или вывод какой программы должен служить фай-

лом стандартного ввода для запускаемой программы и какой файл или ввод какой программы должен служить файлом стандартного вывода или вывода диагностических сообщений для запускаемой программы. Тогда интерпретатор перед выполнением системного вызова exec открывает указанные файлы, подменяя смысл дескрипторов 1, 2 и 3.

То же самое может проделать и любая другая программа, запускающая третью программу в специально созданном процессе. Следовательно, всё, что требуется для нормального функционирования механизма перенаправления ввода/вывода, – это придерживаться при программировании соглашения об использовании дескрипторов stdin, stdout и stderr. Это не очень трудно, поскольку в наиболее распространенных функциях библиотеки ввода/вывода printf, scanf и error вообще не требуется указывать дескриптор файла. Функция printf неявно использует stdout, функция scanf – stdin, а функция error – stderr.

Файловая система

Файл в системе UNIX представляет собой множество символов с произвольным доступом. В файле содержатся произвольные данные, помещенные туда пользователем, и файл не имеет никакой другой структуры, кроме той, какую налагает на него пользователь.

Структура файловой системы

Здесь мы рассмотрим одну из первых реализации файловой системы, поскольку основные её идеи сохраняются до сих пор.

Информация на дисках размещается поблочно, по 512 байт в каждом блоке. Обратите внимание, что блок специально взят равным размеру сектора. Диск разбивается на следующие области (рис. 8.1):

- ♦ неиспользуемый блок;
- ♦ управляющий блок или суперблок, в котором содержится размер диска и границы других областей;
- ♦ i-список, состоящий из описаний файлов, называемых i-узлами;
- ♦ область для хранения содержимого файлов.

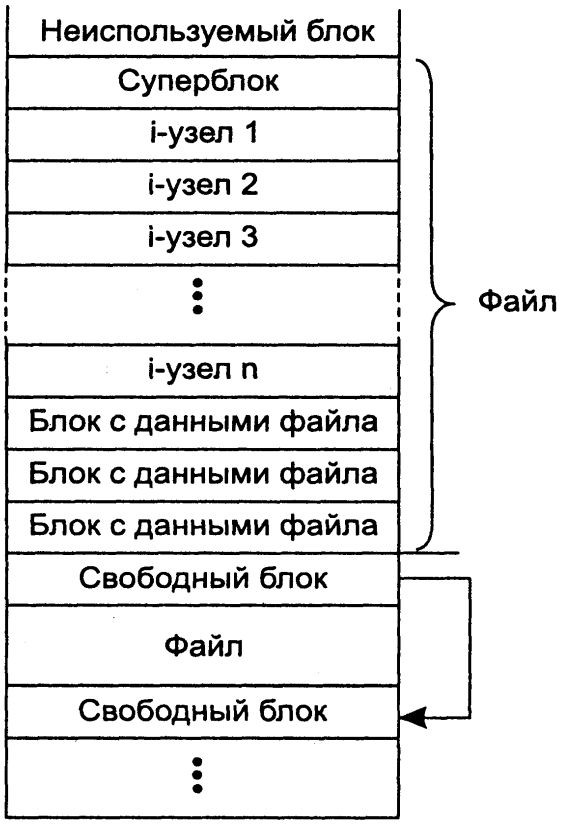

Рис. 8.1. Организация файлов в UNIX на диске

Каждый i-узел содержит:

♦ идентификацию владельца;

♦ идентификацию группы владельца;

♦ биты защиты;

♦ физические адреса на диске или ленте, где находится содержимое файла;

♦ размер файла;

♦ время создания файла;

♦ время последнего использования файла (modification time);

♦ время последнего изменения атрибутов (change time);

♦ число связей-ссылок, указывающих на файл;

♦ индикацию, является ли файл директорией, обычным файлом или специальным файлом. Следом за i-списком идут блоки, предназначенные для хранения содержимого файлов. Пространство на диске, оставшееся свободным от файлов, образует связанный список свободных блоков.

Таким образом, файловая система UNIX представляет собой структуру данных, размещённую на диске и содержащую управляющий суперблок, в котором определена файловая система в целом; массив i-узлов, где определены все файлы в файловой системе; сами файлы; и, наконец, совокупность свободных блоков. Выделение пространства под данные осуществляется блоками фиксированного размера.

Каждый файл однозначно идентифицируется *старшим номером устройства, младшим номером устройства* и *i-номером* (индексом i-узла данного файла в массиве i-узлов). Когда вызывается драйвер устройства, по старшему номеру индексируется массив входных точек в драйверы. По младшему номеру драйвер выбирает одно устройство из группы идентичных физических устройств.

Файл-директория, в котором перечислены имена файлов, позволяет установить соответствие между именами и самими файлами. Директории образуют древовидную структуру. На каждый обычный файл или файл устройства могут иметься ссылки в различных узлах этой структуры. В непривилегированных программах запись в директории не разрешена, но при наличии соответствующих разрешений они могут читать их. Дополнительных связей между директориями нет.

Большое число системных директорий система UNIX использует для своих собственных нужд. Одна из них, корневая директория, является базой для всей структуры директорий, и, «отталкиваясь» от неё, можно найти размещение всех файлов. В других системных директориях содержатся программы и команды, предоставляемые пользователям, и файлы устройств.

Имена файлов задаются последовательностью имён директорий, разделенных косой чертой («/») и приводящих к концевому узлу (листу) некоторого дерева. Если имя файла начинается с косой черты, то поиск по дереву начинается в корневой директории. Если же имя файла не имеет в начале косой черты, то поиск начинается с текущей директории. Имена файлов, начинающиеся с «../», подразумевают начало поиска в директории, родительской по отношению к текущей. Имя файла stuff (персонал) указывает на элемент stuff в текущей директории. Имя файла /work/alex/stuff приводит к поиску директории work в корневой директории. Затем

к поиску директории alex в директории work и, наконец, к поиску элемента stuff в директории alex. Сама по себе косая черта / обозначает корневую директорию. В приведенном примере нашла отражение типичная иерархическая структура файловой системы, например, work может обозначать диск (устанавливаемый при работе пользователя), alex может быть директорией пользователя, а stuff может принадлежать alex.

Файл, не являющийся директорией, может встречаться в различных директориях, возможно, под разными именами. Это называется связыванием. Элемент в директории, относящийся к одному файлу, называется связью. В системе UNIX все такие связи имеют равный статус. Файлы не принадлежат директориям. Скорее, файлы существуют независимо от элементов директорий, а связи в директориях указывают действительно на физические файлы. Файл «исчезает», когда удаляется последняя связь, указывающая на него. Биты защиты, заданные в связях, могут отличаться от битов в исходном файле. Таким образом, решается проблема избирательного ограничения доступа к файлам.

С каждым поддерживаемым системой устройством ассоциируется один или большее число специальных файлов. Операции ввода/вывода для специальных файлов осуществляются так же, как и для обычных дисковых файлов, с той лишь разницей, что эти операции активизируют соответствующие устройства. Специальные файлы обычно находятся в справочнике /dev. На специальные файлы могут указывать связи точно так же, как на обычные файлы.

От файловой системы не требуется, чтобы она вся целиком размещалась на том устройстве, где находится корень. Запрос от системы mount (на установку носителей и т. п.) позволяет встраивать в иерархию файлов файлы на сменных томах. Команда mount имеет несколько опций, но обязательных аргументов у стандартного варианта её использования два: имя файла блочного устройства и имя каталога. В результате выполнения этой команды файловая система, расположенная на указанном устройстве, подключается к системе таким образом, что ее содержимое заменяет собой содержимое заданного в команде каталога. Поэтому для *монтирования* соответствующего тома обычно используют пустой каталог. Команда umount

выполняет обратную операцию – «отсоединяет» (размонтирует) файловую систему, после чего диск с данными можно физически извлечь из системы. Например, для записи данных на дискету необходимо её смонтировать, а после работы – размонтировать.

Защита файлов

Защита файлов осуществляется при помощи номера, идентифицирующего пользователя, и установки десяти битов защиты – атрибутов доступа. Права доступа подразделяются на три типа: чтение (read), запись (write) и выполнение (execute). Эти типы прав доступа могут быть предоставлены трём классам пользователей:

владельцу файла, группе, в которую входит владелец, и всем (прочим) пользователям. Девять из этих битов управляют защитой по чтению, записи и исполнению для владельца файла, других членов группы, в которую входит владелец, и всех других пользователей. Файл всегда связан с определенным пользователем – своим владельцем – и с определенной группой, то есть у него есть уже известные нам UID (user ID, идентификатор пользователя) и GID (group ID, идентификатор группы). Изменять права доступа к файлу разрешено только его владельцу. Изменить владельца файла может суперпользователь, группу – суперпользователь или владелец файла.

Программа, выполняющаяся в системе, всегда запускается от имени какого-то пользователя и какой-то группы (обычно – основной группы этого пользователя), но связь процессов с пользователями и группами организована сложнее:

здесь различаются идентификатор для доступа к файловой системе (FSUID – file system access user ID, FSGID – file system access group ID) и эффективный идентификатор (EUID – effective user ID, EGID – effective group ID), а при доступе к файлам учитываются ещё и полномочия (capabilities), присвоенные самому процессу. При создании файл получает UID, совпадающий с FSUID процесса, который его создаёт, и, как правило, GID, совпадающий с FSGID этого процесса. Атрибуты доступа определяют, что разрешено делать с данным файлом данной категории пользователей. Имеются всего три операции: чтение, запись и выполнение.

При создании файла (или ещё одного имени для уже существующего файла) модифицируется не сам файл, а каталог, в котором появляются новые ссылки на узлы. Удаление файла заключается в удалении ссылки. Таким образом, право на создание или удаление файла – это право на запись в каталог.

Право на выполнение каталога интерпретируется как право на поиск в нём (прохождение через него). Оно позволяет обратиться к файлу по пути, содержащему данный каталог, даже тогда, когда каталог не разрешено читать и список всех его файлов поэтому недоступен.

Помимо трёх названных основных атрибутов доступа существуют дополнительные, используемые в следующих случаях. Атрибуты SUID и SGID существенны при запуске программы на выполнение: они требуют, чтобы программа выполнялась не от имени запустившего ее пользователя (группы), а от имени владельца (группы) того файла, в котором она находится. Выражаясь формально, если файл программы имеет атрибут SUID (SGID), то FSUID и EUID (FSGID и EGID) соответствующего процесса не наследуются от процесса, запустившего его, а совпадают с UID (GID) файла. Благодаря этому пользователи получают возможность запустить системную программу, которая создает свои рабочие файлы в закрытых для них каталогах.

Кроме того, если процесс создает файл в каталоге, имеющем атрибут SGID, то файл получает GID не по FSGID процесса, а по GID каталога. Это удобно для коллективной работы: все файлы и подкаталоги в каталоге автоматически оказываются принадлежащими одной и той же группе, хотя создавать их могут разные пользователи. Есть еще один атрибут – CVTX, который теперь относится к каталогам. Он показывает, что из каталога, имеющего этот атрибут, ссылку на файл может удалить только владелец файла.

Существуют две стандартные формы записи прав доступа – символьная и восьмеричная. Символьная представляет собой цепочку из десяти знаков, первый из которых не относится собственно к правам, а обозначает тип файла. Используются следующие обозначения:

♦ «-» – обычный файл;

- «d» – каталог (директория);
- «c» – символьное устройство;
- «b» – блочное устройство;
- «p» – именованный канал (named pipe);
- «s» – «гнездо» (socket[1]);
- «l» – символическая ссылка.

Далее следуют три последовательности, каждая из трёх символов, соответствующие правам пользователя, группы и всех остальных. Наличие права на чтение обозначается буквой «r», на запись – «w», на выполнение – «x», отсутствие какого–либо права – знаком «-» в соответствующей позиции.

Наличие атрибута SUID (SGID) обозначается заглавной буквой «S» в позиции права на выполнение для владельца (группы), если выполнение не разрешено, и прописной буквой «s», если разрешено.

Восьмеричная запись – это шестизначное число, первые два знака которого обозначают тип файла и довольно часто опускаются, третья цифра – атрибуты GUID (4), SGID (2) и SVTX (1), а оставшиеся три – соответственно права владельца, группы и всех остальных. Очевидно, что право на чтение можно представить числом «4», право на запись – числом «2», а право на выполнение кодируется как «1».

Например, стандартный набор прав доступа для каталога /tmp в символьной форме выглядит как drwxrwxrwt, а в восьмеричной – как 041777 (это каталог; чтение, запись и поиск разрешены всем; установлен атрибут SVTX). А набор прав -r-S-xw-, или в числовом виде – 102412, будет означать, что это обычный файл, владельцу разрешается читать его, но не выполнять и не изменять; пользователям из группы файла (за исключением владельца) – выполнять (причём во время работы программа получит права владельца файла), но не читать и не изменять; а всем остальным – изменять, но не читать и не выполнять.

[1] *Socket* – это понятие, связанное со стеком протоколов TCP/IP, который является «родным» для UNIX. Его следует понимать как некий адрес или порт, через который связываются удаленные программы.

Большинство программ создают файлы с разрешением на чтение и запись для всех пользователей, а каталоги – с разрешением на чтение, запись и поиск для всех пользователей. Этот исходный набор атрибутов логически складывается с «пользовательской маской» – *user file-creation mask,* сокращенно umask, которая обычно ограничивает доступ. Например, следующие значения для umask u=rwx, g=rwx, o=r-x следует понимать так: у владельца и группы остается полный набор прав, а всем остальным запрещается запись. В восьмеричном виде оно запишется как 002 (первая цифра – ограничения для владельца, вторая – для группы, третья – для остальных, запрещение чтения – 4, записи – 2, выполнения – 1). Владелец файла может изменить права доступа к нему командой chmod.

Межпроцессные коммуникации в UNIX

ОС UNIX в своей основе наиболее полно отвечает требованиям технологии «клиент–сервер». Эта универсальная модель служит основой построения любых сколь угодно сложных систем, в том числе и сетевых. Разработчики СУБД, коммуникационных систем, систем электронной почты, банковских систем и т. д. во всем мире широко используют технологию «клиент–сервер». Для построения программных систем, работающих по принципам модели типа «клиент–сервер» в UNIX существуют следующие механизмы:

- ♦ сигналы;
- ♦ семафоры;
- ♦ программные каналы;
- ♦ очереди сообщений;
- ♦ сегменты разделяемой памяти;
- ♦ вызовы удаленных процедур.

Многие из этих механизмов нам уже знакомы, поэтому рассмотрим их вкратце.

Сигналы

Если рассматривать выполнение процесса в виртуальном компьютере, который предоставляется каждому пользователю, то в такой системе должна существовать система прерываний, отвечающая стандартным требованиям:

♦ обработка исключительных ситуаций;

♦ средства обработки внешних и внутренних прерываний;

♦ средства управления системой прерываний (маскирование и демаскирование).

Всем этим требованиям в UNIX отвечает техника сигналов, которая может не только воспринимать и обрабатывать сигналы, но и порождать их и посылать на другие машины (процессы). Сигналы могут быть синхронными, когда инициатор сигнала – сам процесс, и асинхронными, когда инициатор возникновения сигнала – интерактивный пользователь за терминалом. Источником асинхронных сигналов может быть также ядро, когда оно контролирует определенные состояния аппаратуры, рассматриваемые как ошибочные.

Сигналы можно рассматривать как простейшую форму межпроцессного взаимодействия, которое используется для передачи от одного процесса другому или от ядра ОС какому–либо процессу уведомления о возникновении определенного события.

Семафоры

Механизм семафоров, реализованный в ОС UNIX, является обобщением классического механизма семафоров общего вида, предложенного известным голландским специалистом профессором Дейкстрой. Семафор в ОС UNIX состоит из следующих элементов:

♦ значение семафора;

♦ идентификатор процесса, который хронологически последним работал с семафором;

♦ число процессов, ожидающих увеличения значения семафора;

♦ число процессов, ожидающих нулевого значения семафора.

Для работы с семафорами имеются следующие три системных вызова:

♦ semget – для создания и получения доступа к набору семафоров;

♦ semop – для манипулирования значениями семафоров (с помощью именно этого системного вызова осуществляют синхронизацию процессов на основе использования семафоров);

♦ semctl – для выполнения разнообразных управляющих операций над набором семафоров.

Системный вызов semget имеет следующий синтаксис:

```
id = semget(key, count, flag);
```

где параметры key и flag и возвращаемое значение системного вызова (id) имеют тот же смысл, что для других системных вызовов семейства «get», а параметр count задает число семафоров в наборе семафоров, обладающих одним и тем же ключом. После этого индивидуальный семафор идентифицируется дескриптором набора семафоров и номером семафора в этом наборе. Если к моменту выполнения системного вызова semget набор семафоров с указанным ключом уже существует, то обращающийся процесс получит соответствующий дескриптор, но так и не узнает о реальном числе семафоров в группе (хотя позже это все-таки можно узнать с помощью системного вызова semctl).

Основным системным вызовом для манипулирования семафором является semop:

```
oldval = semop(id,oplist, count);
```

где id – это ранее полученный дескриптор группы семафоров, oplist – массив описателей операций над семафорами группы, а count – размер этого массива. Значение, возвращаемое системным вызовом, является значением последнего обработанного семафора. Каждый элемент массива oplist имеет следующую структуру:

♦ номер семафора в указанном наборе семафоров;

♦ операция;

♦ флаги.

Если проверка прав доступа проходит нормально и указанные в массиве oplist номера семафоров не выходят за пределы общего размера набора семафоров, то системный вызов выполняется следующим образом. Для каждого элемента массива oplist значение соответствующего семафора изменяется в соответствии со значением поля «операция»:

♦ если значение поля операции положительно, то значение семафора увеличивается на единицу, а все процессы, ожидающие увеличения значения семафора, активизируются (пробуждаются в терминологии UNIX);

♦ если значение поля операции равно нулю, то если значение семафора также равно нулю, выбирается следующий элемент массива oplist. Если же значение семафора отлично от нуля, то ядро увеличивает на единицу число процессов, ожидающих нулевого значения семафора, а обратившийся процесс переводится в состояние ожидания (усыпляется в терминологии UNIX);

♦ если значение поля операции отрицательно и его абсолютное значение меньше или равно значению семафора, то ядро прибавляет это отрицательное значение к значению семафора. Если в результате значение семафора стало нулевым, то ядро активизирует (пробуждает) все процессы, ожидающие нулевого значения этого семафора. Если же значение семафора меньше абсолютной величины поля операции, то ядро увеличивает на единицу число процессов, ожидающих увеличения значения семафора, и откладывает (усыпляет) текущий процесс до наступления этого события.

Интересно заметить, что основным поводом для введения массовых операций над семафорами было стремление дать программистам возможность избегать тупиковых ситуаций в связи с семафорной синхронизацией. Это обеспечивается тем, что системный вызов semop, каким бы длинным он ни был (по причине потенциально неограниченной длины массива oplist), выполняется как атомарная операция, то есть во время выполнения semop ни один другой процесс не может изменить значение какого-либо семафора.

Наконец, среди флагов-параметров системного вызова semop может содержаться флаг с символическим именем IPC_NOWAIT, наличие которого заставляет ядро ОС UNIX не блокировать текущий процесс, а лишь сообщать в ответных параметрах о возникновении ситуации, которая может привести к блокированию процесса при отсутствии флага IPC_NOWAIT. Мы не будем обсуждать здесь возможности корректного завершения работы с семафорами при незапланированном завершении процесса; заметим только, что такие возможности обеспечиваются.

Системный вызов semctl имеет формат

semctl (id, number, cmd, arg);

где id – это дескриптор группы семафоров, number – номер семафора в группе, cmd – код операции, а arg – указатель на структуру, содержимое которой интерпретируется по-разному, в зависимости от операции. В частности, с помощью semctl можно уничтожить индивидуальный семафор в указанной группе. Однако детали этого системного вызова настолько громоздки, что мы рекомендуем в случае необходимости обращаться к технической документации используемого варианта операционной системы.

Программные каналы

Мы с вами уже знакомились с программными каналами в главе 6. Однако рассмотрим этот механизм ещё раз, так сказать в его исходном, изначальном толковании.

Программные каналы (pipes) в ОС UNIX являются очень важным средством взаимодействия и синхронизации процессов. Теоретически программный канал позволяет взаимодействовать любому числу процессов, обеспечивая дисциплину *FIFO* (first-in-first-out). Другими словами, процесс, читающий из программного канала, прочитает самые давние записанные в программный канал данные. В традиционной реализации программных каналов для хранения данных использовались файлы. В современных версиях ОС UNIX для реализации программных каналов применяются другие средства IPC (в частности, очереди сообщений).

В UNIX различаются два вида программных каналов – именованные и неименованные. Именованный программный канал может служить для общения и синхронизации произвольных процессов, знающих имя данного программного канала и имеющих соответствующие права доступа. Неименованным программным каналом могут пользоваться только создавший его процесс и его потомки (необязательно прямые).

Для создания именованного программного канала (или получения к нему доступа) используется обычный файловый системный вызов open. Для создания же неименованного программного канала существует специальный системный вызов pipe (исторически более ранний). Однако после получения соответствующих деск-

рипторов оба вида программных каналов используются единообразно с помощью стандартных файловых системных вызовов read, write и close.

Системный вызов pipe имеет следующий синтаксис:

plpe(fdptr);

где fdptr – это указатель на массив из двух целых чисел, в который после создания неименованного программного канала будут помещены дескрипторы, предназначенные для чтения из программного канала (с помощью системного вызова read) и записи в программный канал (с помощью системного вызова write). Дескрипторы неименованного программного канала – это обычные дескрипторы файлов, то есть такому программному каналу соответствуют два элемента таблицы открытых файлов процесса. Поэтому при последующем использовании системных вызовов read и write процесс совершенно не обязан отличать случай использования программных каналов от случая использования обычных файлов (*собственно, на этом и основана идея перенаправления ввода/вывода и организации конвейеров*).

Для создания именованных программных каналов (или получения доступа к уже существующим каналам) используется обычный системный вызов open. Основным отличием от случая открытия обычного файла является то, что если именованный программный канал открывается на запись и ни один процесс не открыл тот же программный канал для чтения, то обращающийся процесс блокируется до тех пор, пока некоторый процесс не откроет данный программный канал для чтения. Аналогично обрабатывается открытие для чтения.

Запись данных в программный канал и чтение данных из программного канала (независимо от того, именованный он или неименованный) выполняются с помощью системных вызовов read и write. Отличие от случая использования обычных файлов состоит лишь в том, что при записи данные помещаются в начало канала, а при чтении выбираются (освобождая соответствующую область памяти) из конца канала.

Окончание работы процесса с программным каналом (независимо от того, именованный он или неименованный) производится с помощью системного вызова close.

Очереди сообщений

Для обеспечения возможности обмена сообщениями между процессами этот механизм поддерживается следующими системными вызовами:

♦ msgget для образования новой очереди сообщений или получения дескриптора существующей очереди;

♦ msgsnd для посылки сообщения (вернее, для его постановки в указанную очередь сообщений);

♦ msgrcv для приёма сообщения (вернее, для выборки сообщения из очереди сообщений);

♦ msgctl для выполнения ряда управляющих действий.

Ядро хранит сообщения в виде связного списка (очереди), а дескриптор очереди сообщений является индексом в массиве заголовков очередей сообщений.

Системный вызов msgget обладает стандартным для семейства «get» системных вызовов синтаксисом:

msgqid = msgget (key, flag);

При выполнении системного вызова msgget ядро ОС UNIX либо создает новую очередь сообщений, помещая её заголовок в таблицу очередей сообщений и возвращая пользователю дескриптор вновь созданной очереди, либо находит элемент таблицы очередей сообщений, содержащий указанный ключ, и возвращает соответствующий дескриптор очереди.

Для посылки сообщения используется системный вызов msgsnd:

msgsnd(msgqid, msg, count, flag);

где msg – это указатель на структуру, содержащую определяемый пользователем целочисленный тип сообщения и символьный массив – собственно сообщение;

count задает размер сообщения в байтах, а flag определяет действия ядра при выходе за пределы допустимых размеров внутренней буферной памяти.

Для приёма сообщения используется системный вызов msgrcv:

count = msgrcv(id, msg, maxcount, type, flag);

Здесь msg – это указатель на структуру данных в адресном пространстве пользователя, предназначенную для размещения принятого сообщения; maxcount задаёт размер области данных (массива байтов) в структуре msg; значение type специфи-

цирует тип сообщения, которое желательно принять; значение параметра flag указывает ядру, что следует предпринять, если в указанной очереди сообщений отсутствует сообщение с указанным типом. Возвращаемое значение системного вызова задаёт реальное число байтов, переданных пользователю.

Системный вызов

```
Msgctl(id, cmd, mstatbuf);
```

служит для опроса состояния описателя очереди сообщений, изменения его состояния (например, изменения прав доступа к очереди) и для уничтожения указанной очереди сообщений.

Разделяемая память

Для работы с разделяемой памятью используются четыре системных вызова:

♦ shmget – создаёт новый сегмент разделяемой памяти или находит существующий сегмент с тем же ключом;

♦ shmat – подключает сегмент с указанным дескриптором к виртуальной памяти обращающегося процесса;

♦ shmdt – отключает от виртуальной памяти ранее подключенный к ней сегмент с указанным виртуальным адресом начала;

♦ shmctl – служит для управления разнообразными параметрами, связанными с существующим сегментом.

После того как сегмент разделяемой памяти подключен к виртуальной памяти процесса, этот процесс может обращаться к соответствующим элементам памяти с использованием обычных машинных команд чтения и записи, не прибегая к использованию дополнительных системных вызовов.

Синтаксис системного вызова shmget выглядит следующим образом:

```
shmid = shmget (key, size, flag);
```

Параметр size определяет желаемый размер сегмента в байтах. Далее работа происходит по общим правилам. Если в таблице разделяемой памяти находится элемент, содержащий заданный ключ, и права доступа не противоречат текущим характеристикам обращающегося процесса, то значением системного вызова является дескриптор существующего сегмента (и обратившийся процесс так и не узнает

реального размера сегмента, хотя впоследствии его все-таки можно узнать с помощью системного вызова shmctl). В противном случае создаётся новый сегмент с размером не меньше установленного в системе минимального размера сегмента разделяемой памяти и не больше установленного максимального размера. Создание сегмента не означает немедленного выделения под него основной памяти. Это действие откладывается до выполнения первого системного вызова подключения сегмента к виртуальной памяти некоторого процесса. Аналогично, при выполнении последнего системного вызова отключения сегмента от виртуальной памяти соответствующая основная память освобождается.

Подключение сегмента к виртуальной памяти выполняется путем обращения к системному вызову shmat:

```
virtaddr = shmat(id, addr, flags);
```

Здесь id – это ранее полученный дескриптор сегмента, а addr – желаемый процессом виртуальный адрес, который должен соответствовать началу сегмента в виртуальной памяти. Значением системного вызова является реальный виртуальный адрес начала сегмента (его значение не обязательно совпадает со значением прямого параметра addr). Если значением addr является нуль, ядро выбирает подходящий виртуальный адрес начала сегмента.

Для отключения сегмента от виртуальной памяти используется системный вызов shmdt:

```
shmdt(addr);
```

где addr – это виртуальный адрес начала сегмента в виртуальной памяти, ранее полученный от системного вызова shmat. При этом система гарантирует (на основе использования таблицы сегментов процесса), что указанный виртуальный адрес действительно является адресом начала разделяемого сегмента в виртуальной памяти данного процесса.

Для управления памятью служит системный вызов shmctl:

```
shmctl(id, cmd, shsstatbuf);
```

Он содержит прямой параметр cmd, идентифицирующий требуемое конкретное действие, и предназначен для выполнения различных функций. Наиболее важной является функция уничтожения сегмента разделяемой памяти, которое произ-

водится следующим образом. Если к моменту выполнения системного вызова ни один процесс не подключил сегмент к своей виртуальной памяти, то основная память, занимаемая сегментом, освобождается, а соответствующий элемент таблицы разделяемых сегментов объявляется свободным. В противном случае в элементе таблицы сегментов выставляется флаг, запрещающий выполнение системного вызова shmget по отношению к этому сегменту, но процессам, успевшим получить дескриптор сегмента, по-прежнему разрешается подключать сегмент к своей виртуальной памяти. При выполнении последнего системного вызова отключения сегмента от виртуальной памяти операция уничтожения сегмента завершается.

Вызовы удаленных процедур (RPC)

Во многих случаях взаимодействие процессов носит характер «клиент–сервер». Один из процессов («клиент») запрашивает у другого процесса («сервера») некоторую услугу (сервис) и не продолжает свое выполнение до тех пор, пока эта услуга не будет выполнена (и пока процесс-клиент не получит соответствующие результаты). Видно, что семантически такой режим взаимодействия эквивалентен вызову процедуры. Отсюда и соответствующее название. Кроме этого, ОС UNIX по своей идеологии идеально подходит для того, чтобы быть сетевой операционной системой. И на её основе можно создавать распределённые системы и организовывать распределённые вычисления. Свойства переносимости позволяют создавать «операционно однородные» сети, включающие разнородные компьютеры. Однако остаётся проблема разного представления данных в компьютерах разной архитектуры. Поэтому одной из основных идей RPC является автоматическое обеспечение преобразования форматов данных при взаимодействии процессов, выполняющихся на разнородных компьютерах.

Реализация технологии вызовов удаленных процедур (remote procedure call – RPC) достаточно сложна, поскольку этот механизм должен обеспечить работу взаимодействующих процессов, которые находятся на разных компьютерах. Если в случае обращения к процедуре, расположенной на том же компьютере, процесс общается с ней через стек или общие области памяти, то в случае удаленного вызова передача параметров процедуре превращается в передачу запроса по сети. Соот-

ветственно, и получение результата так же осуществляется посредством использования сетевых механизмов.

Удаленный вызов процедур включает следующие шаги [70]:

♦ процесс-клиент осуществляет локальный вызов процедуры, которую называют «заглушкой» (stub). Задача этого модуля-заглушки – принять аргументы, преобразовать их в стандартную форму и сформировать сетевой запрос. Упаковка аргументов и создание сетевого запроса называется сборкой (marshalling);

♦ сетевой запрос пересылается на удалённую систему, где соответствующий модуль ожидает такой запрос и при его получении извлекает параметры вызова процедуры (unmarshalling), а затем передаёт их серверу удаленной процедуры. После выполнения осуществляется обратная передача.

Операционная система Linux

Linux – это современная POSIX-совместимая и UNIX-подобная операционная система для персональных компьютеров и рабочих станций.

Как известно, Linux – это свободно распространяемая версия UNIX, которая первоначально была разработана Линусом Торвальдсом (Linus Torvalds) (torvalds@kruuna.helsinki.fi) в университете Хельсинки (Финляндия). Все компоненты системы, включая исходные тексты, распространяются с лицензией на свободное копирование и установку для неограниченного числа пользователей.

Linux был создан с помощью многих UNIX-программистов и энтузиастов из Интернета. К данному проекту добровольно подключились те, кто имеет достаточно навыков и способностей развивать систему. Большинство программ Linux разработано в рамках проекта GNU из Free Software Foundation в Кэмбридже, Массачусетс. Но в него внесли лепту также программисты всего мира.

Изначально Linux создавался как «самодельная» UNIX-подобная реализация для ПК типа IBM PC с процессором i80386. Однако Linux стал настолько популярен и его нынче поддерживает такое большое число компаний, что в настоящее время имеется реализация этой ОС практически для всех типов процессоров и компьютеров на их основе. На базе ОС Linux создаются и встроенные системы, и

суперкомпьютеры. Система поддерживает кластеризацию и большинство современных интерфейсов и технологий.

Linux поддерживает большинство свойств, присущих другим реализациям UNIX, плюс ряд тех, которых больше нигде нет. Поэтому этот раздел можно считать лишь поверхностным обзором характеристик ядра Linux.

Linux – это полноценная многозадачная многопользовательская операционная система (точно так же, как и все другие версии UNIX). Это означает, что одновременно много пользователей могут работать на одной машине, одновременно выполняя много программ. Linux достаточно хорошо совместим с рядом стандартов для UNIX (насколько можно говорить о стандартизации UNIX) на уровне исходных текстов, включая IEEE POSIX.l, System V и BSD. Такая совместимость учитывалась при его создании. Большинство свободно распространяемых по сети Интернет 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), псевдотерминалы (pty), поддержку национальных и стандартных клавиатур динамически загружаемыми драйверами клавиатур.

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, например Minix-1 и Xenix. Реализована также система управления файлами на основе FAT, позволяющая непосредственно обращаться к файлам, находящимся в разделах с этой файловой системой. Поддерживается и файловая система ISO 9660 CD–ROM для работы с дисками CD-ROM. Имеются системы управления файлами и на томах с HPFS и NTFS, правда, они работают только на чтение файлов. Созданы варианты системы управления файлами и для доступа к FAT32.

Linux, как и все UNIX-системы, обеспечивает полный набор протоколов стека TCP/IP для сетевой работы. Это включает драйверы устройств для многих попу-

лярных сетевых адаптеров технологии Ethernet, протоколы SLIP (serial line Internet protocol, обеспечивающий 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. Очень часто на компьютерах, работающих под управлением Linux, реализуют DNS-сервер, WWW-серверы (Apache), файерволы для защиты локальных сетей при работе в Интернете, почтовые серверы, сервер DHCP.

Ядро Linux сразу было создано с учётом возможностей защищённого режима процессоров Intel 80386 и 80486. В частности, Linux использует парадигму описания памяти в защищённом режиме и другие новые свойства процессоров. В отличие от старых версий UNIX, в которых задачи выгружались во внешнюю память на магнитных дисках целиком, ядро Linux поддерживает загрузку только нужных страниц. То есть с диска в память загружаются те сегменты программы, которые действительно используются. Возможно использование одной страницы, физически один раз загруженной в память, несколькими выполняемыми программами, то есть реентерабельность кода, присущая всем UNIX-системам, сохранилась. В настоящее время имеются ядра для этой системы, оптимизированные для работы с процессорами Intel и AMD последнего поколения, хотя основные архитектурные особенности защищенного режима работы изменились мало.

Ядро также поддерживает универсальный пул памяти для пользовательских программ и дискового кэша. При этом для кэширования может использоваться вся свободная память, и наоборот, кэш уменьшается при работе больших программ. Этот механизм агрессивного кэширования позволяет увеличить производительность системы.

Выполняемые программы используют динамически связываемые библиотеки, то есть выполняемые программы могут совместно использовать библиотечную программу, представленную одним физическим файлом на диске. Это позволяет выполняемым файлам занимать меньше места на диске, особенно тем, которые многократно используют библиотечные функции. Есть также статические связываемые библиотеки для тех, кто желает пользоваться отладкой на уровне объект-

ных 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.

Семейство операционных систем OS/2 Warp компании IBM

История появления, расцвета и практического ухода со сцены операционных систем под общим названием OS/2 и странна, и поучительна. Будучи одной из самых лучших ОС для ПК по очень большому числу параметров и появившись раньше своих основных конкурентных систем, она тем не менее не смогла стать самой распространенной, хотя могла бы, и с легкостью. Основная причина тому – законы бизнеса (умение рекламировать свой товар, всячески поддерживать его продвижение, вкладывать деньги в завоевание рынка), а не качество самой ОС. Во–первых, компания IBM не сочла необходимым продвигать свою ОС на рынок программного обеспечения, ориентированного на конечного пользователя, а решила продолжить свою практику работы исключительно с корпоративными клиентами. А этот рынок (корпоративного ПО) оказался существенно уже для ПК, чем рынок ПО для конечного пользователя, ибо компьютеры типа IBM PC прежде всего являются персональными. Во–вторых, основные доходы компания IBM получала не от продажи системного ПО для ПК, а за счёт продаж дорогостоящих серверов и другого оборудования. Доходы от продажи своей ОС не представлялись руководству компании IBM значимыми. Для успеха на рынке ОС для ПК необходимо было обеспечить всестороннюю поддержку своей системы соответствующей учебной литературой, широкой рекламой, заинтересовать разработчиков программного обеспечения. Увы, этого не произошло, и сегодня уже практически мало кто знает о системах OS/2. В то же время следует отметить, что те, кто в свое время освоил эту систему и создал для неё соответствующее ПО, до сих пор не переходят на ныне чрезвычайно популярные ОС Windows NT, поскольку последние требуют существенно больше системных ресурсов и при этом функционируют медленнее.

Семейство 32-разрядных ОС для IBM-совместимых компьютеров начало свою историю с появления первой OS/2 v 2.0 в 1992 году. Сейчас мы, как правило, имеем дело уже с четвёртой версией ОС этого семейства. Все ОС в своём названии имеют слово Warp, что переводится с английского как «основа».

OS/2 Warp 4.0 практически представляет собой OS/2 Warp 3.0 (вышедшую ещё в 1994 г.) с несколько улучшенными параметрами для DOS-задач, обновлёнными элементами объектно-ориентированного интерфейса, и включает в себя:

♦ вытесняющую многозадачность (preemptive multitasking) и поддержку DOS- и Windows- (Win32s) приложений;

♦ по-настоящему интуитивно понятный и действительно удобный объектный пользовательский интерфейс;

♦ поддержку стандарта открытого объектного документооборота OpenDoc;

♦ поддержку стандарта OpenGL;

♦ поддержку и встроенную разработку на языке Java;

♦ поддержку шрифтов True Type (TTF);

♦ управление голосом без предварительной подготовки (технология Voice Type);

♦ полную поддержку глобальных сетей Интернет и технологии intranet, доступ в CompuServe[1];

♦ средства построения одноранговых сетей и клиентские части для IBM LAN Server, Windows, Lantastic, Novell Netware 4.1 (в том числе поддержку службы каталогов);

♦ систему удалённого доступа через модемные соединения;

♦ Mobile File System для поддержки мобильных пользователей;

♦ стандарт автораспознавания аппаратных устройств Plug-and-Play (но без столь навязчивого механизма, который реализован в Windows);

[1] *CompuServe* – американская популярная сетевая служба.

♦ набор офисных приложений[1] (базы данных, электронные таблицы, текстовый процессор, генератор отчетов, деловая графика, встроенная система приёма/передачи факсимильных сообщений, информационный менеджер);

♦ полную MultiMedia поддержку, включая работу с видеокамерой, расширенную систему помощи WarpGuide.

Однако наиболее заманчивы не перечисленные из рекламного буклета возможности системы, а удобная и надёжная среда при работе с базами данных, работа в сетях, организованная как клиентское рабочее место при работе с большими системами.

OS/2 Warp предлагает единый интерфейс для программирования прикладных программ (API), совместимый с рядом операционных систем, что позволяет снизить стоимость разработок. Все версии OS/2 и LAN Server, включая текущие версии OS/2 Warp и OS/2 Warp Server 4.5, совместимы по восходящей линии, что позволяет экономить средства, необходимые для поддержания уже существующих прикладных программ.

Чрезвычайно важным для пользователей является тот факт, что компания IBM для всех версий своей ОС регулярно выпускает пакеты обновления (FixPak). Эти пакеты исправляют обнаруженные ошибки, а также вносят новые функции. Для пользователей такая практика сопровождения фирмой своей ОС, безусловно, более выгодна, нежели практика частого выпуска новых версий ОС (ей следует компания Microsoft), в которых обещается исправление обнаруженных ранее недостатков и появление новых функций. Действительно, значительные капиталовложения требуются не только для приобретения новой системы, но и на её освоение.

Так, например, для версии одной из своих самых удачных ОС – Windows NT 4.0 – компания Microsoft выпустила всего только 6 пакетов обновления (ServicePak), тогда как для OS/2 Warp 3.0, которая вышла в свет в 1994 году, компания IBM выпустила уже несколько десятков FixPak. Для OS/2 Warp 4.0 вышло 15 FixPak. Пакеты исправлений и обновлений пользователи получают бесплатно, тогда как за

[1] Справедливости ради следует заметить, что этот набор приложений (называемый BonusPak) не совместим с современными версиями Microsoft Office, и поэтому его используют, как правило, только в «закрытых системах», когда нет обмена с документами, изготовленными посредством приложений Microsoft Office.

новую систему приходится платить большие деньги. К тому же длительная работа по исправлению имеющихся в системе ошибок приводит к уменьшению количества последних со временем, и система становится всё более надёжной и функциональной, в то время как новая версия ОС, как правило, содержит существенно больше ошибок, нежели предыдущая, поскольку объём её становится всё больше и больше, а времени на создание ОС отводится столько же.

Очень полезным, как для управления приложениями, так и для создания несложных собственных программ, является наличие системы программирования на языке высокого уровня REXX, который иногда называют языком процедур. Можно сказать, что это встроенный командный язык, служащий для тех же целей, что и язык для пакетных (batch) файлов в среде DOS, но он обладает несравнимо большими возможностями. Это язык высокого уровня с нетипизированными переменными. Язык легко расширяем, любая программа OS/2 может добавлять в него новые функции. Помимо встроенного интерпретатора с языка REXX имеется система программирования Visual REXX. Есть и объектно-ориентированная версия языка REXX с соответствующим интерпретатором.

Наиболее сильное впечатление, которое можно получить при работе в OS/2, оставляет объектно-ориентированный графический пользовательский интерфейс, а особой популярностью у программистов эта система пользовалась вследствие очень хорошей организации VDM-машин и высокого быстродействия при выполнении обычных DOS-приложений.

Особенности архитектуры и основные возможности OS/2 Warp

Строение и функционирование OS/2 можно считать практически идеальными с точки зрения теории и довольно неплохими – в реализации. В качестве подтверждения этому можно привести один пример, который представляется очень показательным: OS/2 до сегодняшних дней практически неизменна, начиная с версии 2.0, увидевшей свет в 1992 году. Этот факт говорит о глубокой продуманности архитектуры системы, ведь и по сей день OS/2 является одной из самых мощных и продуктивных ОС. Здесь наиболее показательными являются тесты серверов.

В одной из вычислительных лабораторий Санкт-Петербургского государственного университета аэрокосмического приборостроения (ГУАП) с 1995 года в течение нескольких лет функции сервера кафедры вычислительных систем и сетей выполняла OS/2 Warp Adwanced Server. При переходе на сервер Windows NT 4.0 пришлось в два раза увеличить объём оперативной памяти и поменять процессор (с Pentium 90 до Pentium II 300), и даже после этого скорость работы обычных приложений на рабочих станциях не смогла достичь той производительности, какую имели пользователи при сервере на базе OS/2. Аналогичные замечания можно прочесть и в зарубежных публикациях – однопроцессорный OS/2 Warp Server обгоняет по производительности двухпроцессорную Windows NT. Разработчики системы OS/2 решили использовать статические структуры данных (таблицы) для различных системных информационных структур, что приводит к большему быстродействию. Для реализации механизмов поддержки виртуальной памяти использованы наиболее эффективные алгоритмы. Очень удачно реализована диспетчеризация задач.

В OS/2 имеется несколько видов виртуальных машин для выполнения прикладных программ. Собственные 32- и 16-разрядные программы OS/2 выполняются на отдельных виртуальных машинах в режиме вытесняющей многозадачности и могут общаться между собой с помощью средств DDE OS/2. Прикладные программы DOS и Win 16 могут запускаться на отдельных виртуальных машинах в многозадачном режиме. При этом они поддерживают полноценные связи DDE и OLE 2.0 друг с другом и связи DDE с 32-разрядными программами OS/2. Кроме того, при желании можно запустить несколько программ Win 16 на общей виртуальной машине Win 16, где они работают в режиме невытесняющей многозадачности, как это реализовано в Windows 3.x.

Разнообразные сервисные функции API OS/2, в том числе SOM (system object model – модель системных объектов), обеспечиваются с помощью системных динамических библиотек DLL, к которым можно обращаться без требующих затрат времени переходов между кольцами защиты. Ядро OS/2 предоставляет многие базовые сервисные функции API, обеспечивает поддержку файловой системы, управ-

ление памятью и имеет диспетчер аппаратных прерываний. В ядре виртуальных DOS-машин (VDM-ядре) осуществляется эмуляция DOS и процессора 8086, а также управление VDM. Драйверы виртуальных устройств обеспечивают уровень аппаратной абстракции. Драйверы физических устройств напрямую взаимодействуют с аппаратурой.

Модуль реализации механизмов виртуальной памяти в ядре OS/2 поддерживает большие, постраничные, разбросанные адресные пространства, составленные из объектов памяти. Каждый объект памяти управляется так называемым «пейджером» – задачей вне ядра, обеспечивающей резервное хранение страниц объекта памяти. Адресные пространства управляются отображением или размещением объектов памяти внутри них. Ядро управляет защитой памяти и её распределением на основе объектов памяти абстрактным образом, вне зависимости от каких-либо конкретных аппаратных средств трансляции процессорных адресов. В частности, ядро интенсивно использует режим копирования при записи для придания программам способности делить объекты памяти без копирования большого числа страниц, когда новое адресное пространство получает доступ к объекту памяти. Новые копии страниц создаются только тогда, когда программа в одном из адресных пространств обновляет их. Когда ядро принимает страничный сбой в объекте памяти и не имеет страницы памяти в наличии или когда оно должно удалить страницы из памяти по требованию других программ, работающих в машине, оно с помощью механизма IPC[1] уведомляет пейджер об объекте памяти, в котором произошёл сбой. Теперь дело пейджера сервера приложений определить, каким образом предоставить или сохранить данные. Это позволяет системе устанавливать различную семантику для объектов памяти, основываясь на потребностях программ, которые их используют.

Ядро управляет средами исполнения для программ обеспечением множественных заданий и потоков. Каждое задание имеет своё собственное адресное пространство или отображение. Оно назначает объекты памяти, которые задание отобразило на диапазон адресов внутри адресного пространства. Задание также явля-

[1] *IPC* (interprocessing communications) – межпроцессное взаимодействие.

ется блоком размещения ресурсов и защиты, при этом заданиям придаются возможности и права доступа к средствам IPC системы. Для поддержки параллельного исполнения с другой программой в пределах одного адресного пространства ядро отделяет среду исполнения от действительно идущего потока инструкций. Потоки вычислений, включая процессорные ресурсы, потребные для их поддержки, называются потоками. Таким образом, программа может быть загружена в задание и может быть исполнена в нескольких различных местах в коде в одно и то же время на мультипроцессоре или параллельной машине. Это приводит к повышению быстродействия приложения.

Система IPC обеспечивает базовый механизм, позволяющий потокам работать в различных заданиях для связи друг с другом. Система IPC поддерживает надёжную доставку сообщений на порты. Порты представляют собой защищенные каналы между заданиями. Каждому заданию, использующему порт, приписывается набор прав на этот порт. Права могут быть различными для разных заданий. Только одно задание может получать по какому-либо порту, хотя любой поток внутри задания может выполнять операцию приёма. Одно или более заданий могут иметь права посылать в порт. Ядро позволяет заданиям применять систему IPC на передачу друг другу прав на порт. Оно также обеспечивает высокопроизводительный способ передачи больших областей данных в сообщениях. Вместо того чтобы копировать данные, сообщение содержит указатель на них, он называется указателем на данные вне линии. Когда ядро передает сообщение от передатчика к приёмнику, оно заставляет передаваемую память появиться в адресном пространстве приёмника и, как вариант, исчезнуть из адресного пространства передатчика. Ядро само структурировано как задание с потоками, и большинство системных сервисов реализованы как механизмы IPC к ядру, а не как прямые системные вызовы.

Для поддержки операций ввода/вывода (I/O) и доступа к внешним устройствам ядро ОС обеспечивает доступ к ресурсам I/O, таким как устройства с отображаемой памятью, I/O порты и каналы прямого доступа к памяти (DMA[1]), а также возможность отражать прерывания на драйверы устройств, исполняемые в пользо-

[1] *DMA* (direct nemory access) – канал прямого доступа к памяти.

вательском пространстве. Оно имеет сервисы, которые позволяют приоритетным программам заполучать устройства в своё владение: такими программами обычно являются задачно-нейтральные сервисы, типа серверов драйверов устройств, работающих как приложения. Поскольку ядро обязано разместить все прерывания (в силу того, что прерывания обычно выдаются в приоритетном состоянии компьютера, а также в целях поддержания целостности системы), оно имеет логику, которая определяет, должно ли оно обрабатывать прерывание или его следует отразить на сервер. Если прерывание следует отразить в приложение, оно должно быть зарегистрировано в ядре и содержать код, который будет ожидать, пока ядро не отразит прерывание. Как только прерывание отражено, в приложении запускается поток по обработке прерывания.

Характеристики наборов хоста (host) и процессора предоставляют два связанных набора функций, требующихся, если прикладным программам следует обеспечить максимум сервисов операционной системы. Характеристики набора хоста возвращают информацию о процессорном комплексе, работающем в системе, и предоставляют определённые функции системного менеджмента типа времени, даты, останова и рестарта системы. Характеристики набора процессора используются в мультипроцессорных машинах для группировки процессоров в классы. Эти классы позволяют параллельному приложению выполнять несколько потоков одновременно на различных процессорах в машине, в итоге происходит истинно параллельное исполнение.

В соответствии с концепцией микроядерных ОС непосредственно поверх ядра системы OS/2 располагается ряд серверов приложений, которые обеспечивают системные сервисы общего назначения, то есть задачно-нейтральные сервисы. Они зависят только от ядра, некоторых вспомогательных сервисов, экспортируемых доминирующей задачей операционной системы, и от себя самих. В числе задачно–нейтральных сервисов имеются пейджер умолчания, мастер-сервер, который загружает другие задачно-нейтральные серверы в память, сервис низкоуровневых имён, сервис защиты, сервисы инициализации, набор драйверов устройств со связанным кодом поддержки, а также библиотечные подпрограммы для стандартной про-

граммной среды. Дополнительные задачно-нейтральные сервисы типа одиночного файлового сервера могут быть просто добавлены.

С помощью ядра ОС и задачно-нейтральных сервисов приоритетная задача может обеспечить операционную системную среду типа UNIX. Поскольку приоритетная задача является прикладным сервером, возможно добавлять другие серверы для различных задач, исполняющих программы, написанные в различных операционных системах, работающих на машине в одно и то же время.

Существуют некоторые операционные системные сервисы, такие как трансляция сообщений об ошибках, не обеспечиваемые задачно-нейтральными сервисами. Поскольку лучше не дублировать подобные сервисы, приоритетная задача обеспечивает эти сервисы не только своим клиентским приложениям, но и любой другой задаче, исполняющейся в машине.

Особенности интерфейса OS/2 Warp

В OS/2 Warp в качестве стандартной графической оболочки используется среда Workplace Shell (WPS), организованная более логично и удобно, чем известный Windows-интерфейс. Оболочка Workplace Shell основана на мощной системно–объектной модели[1] IBM – технологии, специально разработанной для решения таких проблем, как жёсткая привязка объектов к их клиентам и необходимость использования одного и того же языка программирования. Объекты Workplace Shell работают в среде SOM, доступ в которую можно реализовать почти на всех языках программирования, предусматривающих внешние процедуры, в том числе и на REXX. В отличие от GUI Windows, в котором те же ярлыки[2] объектов никак не связаны между собой, в WPS объекты, имеющие аналогичные ярлыки (shadow[3] в терминологии WPS), просто имеют дополнительные свойства – быть многократно отображёнными почти как самостоятельные объекты. Можно сделать несколько shadow-значков с уже существующей shadow-значков или объекта. При этом любые shadow-значки могут быть перемещены в любое место, и их связи с основным объектом не теряются. Аналогично и в GUI Windows. Но в WPS можно перемес-

[1] System object model – SOM.
[2] Shortcut (ярлык) – значок (или иконка) на рабочем столе Windows, который имеет связь с тем или иным объектом.
[3] Shadow (тень) – значок на рабочем столе OS/2, которая является частью объекта, то есть имеется постоянная двухсторонняя связь между shadow и собственно объектом.

тить основной объект, и его shadow-значки тоже изменят свои параметры, тогда как в GUI Windows произойдет разрушение связей, поскольку связи являются односторонними.

Про SOM можно сказать, что это не связанная ни с одним конкретным языком объектно-ориентированная технология для создания, хранения и использования двоичных библиотек классов. Ключевые слова здесь «двоичные» и «не связанная ни с одним конкретным языком». Хотя теперь многие считают OS/2 технологией прошлого, модель SOM на самом деле представляет собой одну из наиболее интересных разработок в области компьютерной индустрии даже на сегодняшний день. По существу, некоторые идеи, реализованные в OS/2 в начале 90-х годов прошлого столетия, сейчас только обещают быть реализованными в новом поколении ОС Windows с кодовым названием Whistler. Объектно-ориентированное программирование (ООП) заслужило безоговорочное признание в качестве основной парадигмы, однако его применению в коммерческом программном обеспечении препятствуют отсутствие в языках ООП средств для обращения к библиотекам классов, подготовленным на других языках, и необходимость поставлять с библиотеками классов исходные тексты. Многим независимым разработчикам библиотек классов приходится продавать заказчикам исходные тексты, поскольку разные компиляторы по-разному отображают объекты. Настоящий потенциал SOM заключается в её совместимости практически с любой платформой и любым языком программирования. SOM соответствует спецификации CORBA[1], которая определяет стандарт условий взаимодействия между прикладными программами в неоднородной сети.

Интересно отметить тот факт, что существует довольно много альтернативных оболочек для OS/2, начиная с FileBar, примитивной, но зато отлично работающей на компьютерах с 4 Мбайт памяти, и кончая мощной Object Desktop, которая значительно улучшает внешний вид экрана OS/2 и делает работу с системой более удобной.

Помимо оболочек, улучшающих интерфейс OS/2, имеется также ряд программ, расширяющих ее функциональность. В первую очередь это Xfree86 для

[1] *CORBA* (common object request broker architecture) – архитектура посредника стандартного объектного запроса.

OS/2 – полноценная система X Window, которая может использоваться как X-терминал при работе в сети с UNIX-машинами, а также для запуска программ, перенесенных из UNIX в OS/2. К сожалению, таких программ немного, однако большое количество UNIX-программ поставляется вместе с исходными кодами, которые, как правило, практически не нужно изменять для перекомпиляции под Xfree86/OS2.

Серверная операционная система OS/2 Warp 4.5

Новая серверная ОС компании IBM, которая вышла в свет в 1999 году, носит название OS/2 WarpServer for e-Business, что подчеркивает ее основное назначение, но поскольку в процессе её создания она носила кодовое название «Аврора» (Aurora), фактически все её так теперь и называют.

Как известно, предыдущие версии системы OS/2 могли предоставить программисту только 512 Мбайт виртуального адресного пространства для нативных 32-битовых задач. В своё время это было очень много. Однако сегодня, хоть и крайне редко ещё встречаются задачи, требующие столь большого объёма оперативной памяти, некоторые считают серьёзным недостатком ограничение в 512 Мбайт. Поэтому в последней версии системы это ограничение снято, и теперь объем виртуальной памяти может достигать 3 Гбайт (напомним, что в Windows NT 4.0 объём виртуального адресного пространства для задач пользователя составляет 2 Гбайт). По умолчанию максимальный объём виртуальной памяти задачи OS/2 v. 4.5 составляет 2 Гбайт, но командой «VIRTUALADDRESSLIMIT=3072» в конфигурационном файле CONFIG.SYS он может быть увеличен.

В этой системе разработчики постарались все прежние остатки старого 16-битового кода, который ещё оставался в предыдущих версиях системы, заменить на полностью 32-битовые реализации, что повышает скорость работы системы. Прежде всего, сделана поддержка 32-битовых драйверов инсталлируемых файловых систем (IFS), ибо в предыдущих системах работа с ними велась через трансляцию вызовов 32bit→16bit→32bit. В то же время для обеспечения совместимости со старым программным обеспечением кроме 32-битового используется и 16-битовый API.

Для повышения надежности файловой подсистемы создана новая журналирующая файловая система JFS (journaling file system). JFS введена для удовлетворения потребности в более живучей файловой системе для OS/2 Warp. JFS имеет большую безопасность в структурах данных благодаря технике, разработанной для СУБД. Работа с файловой системой происходит в режиме транзакций с ведением журнала транзакций. В случае системных сбоев есть возможность обработки журнала транзакций с целью занесения или сброса каких-либо изменений, произведённых во время системного сбоя, эта система также повышает скорость восстановления файловой системы после сбоя. Сохраняя целостность файловой системы, эта система управления файлами не гарантирует восстановление пользовательских данных. Следует отметить, что файловая система JFS обеспечивает самую высокую скорость работы с файлами из всех известных систем, созданных для ПК, что очень важно для серверной ОС.

Для работы с дисками создан специальный менеджер дисков – LVM[1]. Все устанавливаемые файловые системы содержатся в LVM. LVM осуществляет определение имён дисков для программ, которые этого требуют. Это позволяет избирательно назначить любую букву любому разделу. И даже больше – ОС не будет сама использовать имена дисков. LVM в совокупности с JFS позволяет объединять несколько томов и даже несколько физических дисков в один большой логический том.

Сетевая ОС реального времени QNX

Операционная система QNX является мощной операционной системой, позволяющей проектировать сложные программные системы, работающие в реальном времени как на одном-единственном компьютере, так и в локальной вычислительной сети. Встроенные средства операционной системы QNX обеспечивают поддержку многозадачного режима на одном компьютере и взаимодействие параллельно выполняемых задач на разных компьютерах, работающих в среде локальной вычислительной сети. Основным языком программирования в системе является язык С. Основная операционная среда соответствует стандартам POSIX–интер-

[1] *LVM* (logical volume manager) – менеджер томов (логических дисков).

фейса. Это позволяет с небольшими доработками перенести необходимое накопленное программное обеспечение в среду операционной системы QNX для организации их работы в среде распределенной обработки.

ОС QNX является сетевой, мультизадачной, многопользовательской (многотерминальной) и масштабируемой. С точки зрения пользовательского интерфейса и API она очень похожа на UNIX. Однако QNX – это не версия UNIX, хотя почему–то многие так считают. Она была разработана, что называется «с нуля», канадской фирмой QNX Software Systems Limited в 1989 году по заказу Министерства обороны США. Причём эта система построена на совершенно других архитектурных принципах, отличных от принципов, использованных при создании ОС UNIX.

QNX была первой коммерческой ОС, построенной на принципах микроядра и обмена сообщениями. Система реализована в виде совокупности независимых (но взаимодействующих через обмен сообщениями) процессов различного уровня (менеджеры и драйверы), каждый из которых реализует определенный вид сервиса. Эти идеи позволили добиться нескольких важнейших преимуществ. Вот как они описаны на сайте, посвященном системе QNX [60].

♦ *Предсказуемость,* означающая её применимость к задачам жёсткого реального времени. QNX является операционной системой, которая дает полную гарантию в том, что процесс с наивысшим приоритетом начнет выполняться практически немедленно и что критическое событие (например, сигнал тревоги) никогда не будет потеряно. Ни одна версия UNIX не может достичь подобного качества, поскольку нереентерабельный код ядра слишком велик. Любой системный вызов из обработчика прерывания в UNIX может привести к непредсказуемой задержке (то же самое касается Windows NT, где реальное время заканчивается между ISR и DPC).

♦ *Масштабируемость* и *эффективность,* достигаемые оптимальным использованием ресурсов и означающие её применимость для встроенных (embedded) систем; вы не увидите в каталоге /dev огромной кучи файлов, соответствующих ненужным драйверам. Драйверы и менеджеры можно запускать и удалять (кроме

файловой системы, что очевидно) динамически, просто из командной строки. Вы можете иметь только тот сервис, который вам реально нужен, причем это не требует серьезных усилий и не порождает проблем.

♦ *Расширяемость* и *надёжность* одновременно, поскольку написанный вами драйвер не нужно компилировать в ядро, рискуя вызвать нестабильность системы.

Менеджеры ресурсов (сервис логического уровня) работают в кольце защиты 3, и вы можете добавлять свои, не опасаясь за систему. Драйверы работают в кольце с уровнем привилегий 1 и могут вызвать проблемы, но не фатального характера. Кроме того, их достаточно просто писать и отлаживать.

♦ *Быстрый сетевой протокол FLEET*[1]*,* прозрачный для обмена сообщениями, автоматически обеспечивающий отказоустойчивость, балансирование нагрузки и маршрутизацию между альтернативными путями доступа.

♦ *Компактная графическая подсистема Photon,* построенная на тех же принципах модульности, что и сама ОС, позволяет получить полнофункциональный GUI[2] (расширенный Motif), работающий вместе с POSIX-совместимой ОС всего в 4Мбайт памяти, начиная с i80386 процессора.

Вспомним основные принципы, обязательная реализация которых позволят создавать ОСРВ (см. раздел «Требования, предъявляемые к ОС реального времени», глава 5). Первым обязательным требованием к архитектуре ОСРВ является *многозадачность* в истинном смысле этого слова. Очевидно, что варианты с псевдомногозадачностью (а точнее – не вытесняющая многозадачность) типа MS Windows 3.x или Novell NetWare неприемлемы, поскольку они допускают возможность блокировки или даже полного развала системы одним неправильно работающим процессом. Для предотвращения блокировок ОСРВ должна использовать квантование времени (то есть вытесняющую многозадачность), что является достаточно легкой задачей. Вторая проблема (организация *надёжных вычислений)* может быть эффективно решена при полном использовании возможностей процессоров Intel 80386 и старше, что предполагает работу ОС в 32-разрядном режиме процессора. Для эффективного обслуживания прерываний ОС должна использовать алгоритм

[1] Это фирменная технология, несколько более подробно об этом изложено ниже.

диспетчеризации, обеспечивающий *вытесняющее планирование, основанное на приоритетах.* Наконец, крайне желательны эффективная *поддержка сетевых коммуникаций* и наличие развитых механизмов *взаимодействия между процессами,* поскольку реальные технологические системы обычно управляются целым комплексом компьютеров и/или контроллеров. Весьма важно также, чтобы ОС поддерживала *множественные потоки управления* (не только *мультипрограммный,* но и *мультизадачный* режимы), а также симметричную мультипроцессорность.

И, наконец, при соблюдении всех перечисленных условий ОС должна быть способна *работать на ограниченных аппаратных ресурсах,* поскольку одна из её основных областей применения – это встроенные системы. К сожалению, данное условие обычно реализуется путём урезания стандартных сервисных средств.

Архитектура системы QNX

QNX – это ОС реального времени, позволяющая эффективно организовать распределённые вычисления. В системе реализована концепция связи между задачами на основе сообщений, посылаемых от одной задачи к другой, причём задачи эти могут находиться как на одном и том же компьютере, так и на удалённых, но связанных локальной вычислительной сетью. Реальное время и концепция связи между процессами в виде сообщений оказывают решающее влияние на разрабатываемое для QNX программное обеспечение и на программиста, стремящегося с максимальной выгодой использовать преимущества системы.

Микроядро имеет объем в несколько десятков килобайт (в одной из версий – 10 Кбайт, в другой – менее 32 Кбайт), то есть это одно из самых маленьких ядер среди всех существующих операционных систем. В этом объеме помещаются [54]:

- ♦ механизм передачи сообщений между процессами (IPC);
- ♦ редиректор[3] прерываний;
- ♦ блок планирования выполнения задач;
- ♦ сетевой интерфейс для перенаправления сообщений (менеджер Net).

---

[2] *GUI* (graphic user interface) – графический интерфейс пользователя.
[3] От *redirect* - перенаправлять.

Механизм передачи межпроцессных сообщений занимается пересылкой сообщений между процессами и является одной из важнейших частей операционной системы, так как всё общение между процессами, в том числе и системными, происходит через сообщения. Сообщение в QNX – это последовательность байтов произвольной длины (0–65 535 байтов) произвольного формата. Протокол обмена сообщениями выглядит таким образом. Например, задача блокируется для ожидания сообщения. Другая же задача посылает первой сообщение и при этом блокируется сама, ожидая ответа. Первая задача разблокируется, обрабатывает сообщение и отвечает, разблокируя при этом вторую задачу.

Сообщения и ответы, пересылаемые между процессами при их взаимодействии, находятся в теле отправляющего их процесса до того момента, когда они могут быть приняты. Это значит, что, с одной стороны, уменьшается вероятности повреждения сообщения в процессе передачи, а с другой – уменьшается объём оперативной памяти, необходимый для работы ядра. Кроме того, уменьшается число пересылок из памяти в память, что разгружает процессор. Особенностью процесса передачи сообщений является то, что в сети, состоящей из нескольких компьютеров под управлением QNX, сообщения могут прозрачно передаваться процессам, выполняющимся на любом из узлов. Определены в QNX ещё и два дополнительных метода передачи сообщений – метод представителей (Proxy) и метод сигналов (Signal).

Представители используются в случаях, когда процесс должен передать сообщение, но не должен при этом блокироваться на передачу. В таком случае вызывается функция qnx_proxy_attach( ) и создаётся представитель. Он накапливает в себе сообщения, которые должны быть доставлены другим процессам. Любой процесс, знающий идентификатор представителя, может вызвать функцию Trigger( ), после чего будет доставлено первое в очереди сообщение. Функция Trigger ( ) может вызываться несколько раз, и каждый раз представитель будет доставлять следующее сообщение. При этом представитель содержит буфер, в котором может храниться до 65 535 сообщений. Как известно, сигналы уже давно используются в ОС UNIX. Система QNX поддерживает множество сигналов, совместимых с POSIX, большое

количество сигналов, традиционно использовавшихся в UNIX (поддержка этих сигналов реализована для совместимости с переносимыми приложениями, и ни один из системных процессов QNX их не генерирует), а также несколько сигналов, специфичных для самой QNX. По умолчанию любой сигнал, полученный процессом, приводит к завершению процесса (кроме нескольких сигналов, которые по умолчанию игнорируются). Но процесс с приоритетом уровня «superuser» может защититься от нежелательных сигналов. В любом случае процесс может содержать обработчик для каждого возможного сигнала. Сигналы удобно рассматривать как разновидность программных прерываний.

Редиректор прерываний является частью ядра и занимается перенаправлением аппаратных прерываний в связанные с ними процессы. Благодаря такому подходу возникает один побочный эффект – с аппаратной частью компьютера работает ядро, оно перенаправляет прерывания процессам – обработчикам прерываний. Обработчики прерываний обычно встроены в процессы, хотя каждый из них исполняется асинхронно с процессом, в который он встроен. Обработчик исполняется в контексте процесса и имеет доступ ко всем глобальным переменным процесса. При работе обработчика прерываний прерывания разрешены, но обработчик приостанавливается только в том случае, если произошло более высокоприоритетное прерывание. Если это позволяется аппаратной частью, к одному прерыванию может быть подключено несколько обработчиков, и каждый из них получит управление при возникновении прерывания.

Этот механизм позволяет пользователю избежать работы с аппаратным обеспечением напрямую и тем самым избегать конфликтов между различными процессами, работающими с одним и тем же устройством. Для обработки сигналов от внешних устройств чрезвычайно важно минимизировать время между возникновением события и началом непосредственной его обработки. Этот фактор существен в любой области применения, от работы терминальных устройств до возможности обработки высокочастотных сигналов.

Блок планирования выполнения задач (диспетчер задач) занимается обеспечением многозадачности. В этой части ОС QNX предоставляет разработчику огром-

ный простор для выбора той методики выделения ресурсов процессора задаче, которая обеспечит наиболее подходящие условия для критических приложений или обеспечит такие условия для некритических приложений, что они выполнятся за разумное время, не мешая работе критических приложений.

К выполнению своих функций как диспетчера ядро приступает в следующих случаях:

- какой-либо процесс вышел из блокированного состояния;
- истёк квант времени для процесса, владеющего CPU;
- работающий процесс прерван каким-либо событием.

Диспетчер выбирает процесс для запуска среди неблокированных процессов в порядке значений их приоритетов, которые располагаются в диапазоне от 0 (наименьший) до 31 (наибольший). Обслуживание каждого из процессов зависит от метода диспетчеризации, с которым он работает (уровень приоритета и метод диспетчеризации могут динамически меняться во время работы). В QNX существуют три метода диспетчеризации: FIFO (первым пришел – первым обслужен), round–robin (процессу выделяется определенный квант времени для работы) и адаптивный, который является наиболее используемым.

Первый наиболее близок к кооперативной многозадачности. То есть процесс выполняется до тех пор, пока он не перейдёт в состояние ожидания сообщения, состояние ожидания ответа на сообщение или не отдаст управление ядру. При переходе в одно из таких состояний процесс помещается последним в очередь процессов с таким же уровнем приоритета, а управление передается процессу с наибольшим приоритетом.

Во втором варианте всё происходит так же, как и в предыдущем, с той разницей, что период, в течение которого процесс может работать без перерыва, ограничивается неким квантом времени. Процесс, работающий с адаптивным методом, в QNX ведет себя следующим образом:

- Когда процесс полностью использовал выделенный ему квант времени, его приоритет снижается на 1, если в системе есть процессы с тем же уровнем приоритета, готовые к исполнению.

♦ Если процесс с пониженным приоритетом остаётся не обслуженным в течение секунды, его приоритет увеличивается на 1.

Если процесс блокируется, ему возвращается оригинальное значение приоритета.

По умолчанию процессы запускаются в режиме адаптивной многозадачности. В этом же режиме работают все системные утилиты QNX. Процессы, работающие в разных режимах многозадачности, могут одновременно находиться в памяти и исполняться. Важный элемент реализации многозадачности – это приоритет процесса. Обычно приоритет процесса устанавливается при его запуске. Но есть дополнительная возможность, называемая «вызываемый клиентом приоритет». Как правило, она реализуется для серверных процессов (исполняющих запросы на какое-либо обслуживание). При этом приоритет процесса-сервера устанавливается только на время обработки запроса и становится равным приоритету процесса–клиента.

Сетевой интерфейс в системе QNX является неотъемлемой частью ядра. Он, конечно, взаимодействует с сетевым адаптером через сетевой драйвер, но базовые сетевые сервисы реализованы на уровне ядра. При этом передача сообщения процессу, находящемуся на другом компьютере, ничем не отличается с точки зрения приложения от передачи сообщения процессу, выполняющемуся на том же компьютере. Благодаря такой организации сеть превращается в однородную вычислительную среду. При этом для большинства приложений не имеет значения, с какого компьютера они были запущены, на каком исполняются и куда поступают результаты их работы. Такое решение принципиально отличает QNX от остальных ОС, которые тоже имеют все необходимые средства для работы в сети, и делает системы, работающие под её управлением, по-настоящему распределёнными.

Все сервисы QNX, не реализованные непосредственно в ядре, работают как стандартные процессы в полном соответствии с основными концепциями микроядерной архитектуры. С точки зрения операционной системы системные процессы ничем не отличаются от всех остальных. Как, впрочем, и драйверы устройств. Единственное, что нужно сделать, написав новый драйвер устройства в QNX, что-

бы он стал частью операционной системы, – это изменить конфигурационный файл системы так, чтобы драйвер запускался при загрузке.

Основные механизмы QNX для организации распредёленных вычислений

QNX является сетевой операционной системой и позволяет организовать эффективные распределённые вычисления. Для организации сети на каждой машине, называемой узлом, помимо ядра и менеджера процессов должен быть запущен уже упомянутый нами выше менеджер Net. Менеджер Net не зависит от аппаратной реализации сети. Данная аппаратная независимость обеспечивается за счёт использования сетевых драйверов. В QNX имеются драйверы для различных сетей, например Ethernet, Arcnet, Token Ring. Кроме этого, имеется возможность организации сети через последовательный канал или модем.

В QNX последней, четвертой, версии полностью реализовано встроенное сетевое взаимодействие «точка-точка». Например, находясь на машине A, вы можете скопировать файл с гибкого диска, подключенного к машине B, на жёсткий диск, подключенный к машине C. По существу, сеть из машин QNX действует как один мощный компьютер. Любые ресурсы (модемы, диски, принтеры) могут быть добавлены к системе простым их подключением к любой машине в сети. QNX поддерживает одновременную работу в сетях Ethernet, Arcnet, Serial и Token Ring и обеспечивает более чем один-единственный путь для коммуникации, а также балансировку нагрузки в сетях. Если кабель или сетевая плата выходит из строя таким образом, что связь через эту сеть прекращается, то система будет автоматически перенаправлять данные через другую сеть. Это происходит в режиме «on-line», предоставляя пользователю автоматическую сетевую избыточность и увеличивая скорость коммуникаций во всей системе.

Каждому узлу в сети соответствует уникальный целочисленный идентификатор – логический номер узла. Любой поток в сети QNX имеет прозрачный доступ (при наличии достаточных привилегий) ко всем ресурсам сети, то же самое относится и к взаимодействию потоков. Для взаимодействия потоков, находящихся на разных узлах сети, используются те же самые вызовы ядра, что и для потоков, вы-

полняемых на одном узле. В том случае, если потоки находятся на разных узлах сети, ядро переадресует запрос менеджеру сети. Для организации обмена в сети используется надёжный и эффективный протокол транспортного уровня FLEET. Каждый из узлов может принадлежать одновременно нескольким QNX-сетям. В том случае если сетевое взаимодействие может быть реализовано несколькими путями, для передачи выбирается незагруженная и более скоростная сеть.

Сетевое взаимодействие является узким местом в большинстве операционных систем и обычно создаёт значительные проблемы для систем реального времени. Для того чтобы обойти это препятствие, разработчики QNX создали собственную специальную сетевую технологию FLEET и соответствующий протокол транспортного уровня FTL (FLEET transport layer). Этот протокол не базируется ни на одном из распространенных сетевых протоколов типа IPX или NetBios и обладает рядом качеств, которые делают его уникальным. Основные его качества зашифрованы в аббревиатуре FLEET, которая расшифровывается следующим образом.

Fault-Tolerant Networking	**QNX может одновременно использовать несколько физических сетей. При выходе из строя любой из них данные будут автоматически перенаправлены «на лету» через другую сеть**
Load-Balancing on the Fly	**При наличии нескольких физических соединений QNX автоматически распараллеливает передачу пакетов по соответствующим сетям**
Efficient Performance	**Специальные драйверы, разрабатываемые фирмой QSSL для широкого спектра оборудования, позволяют с максимальной эффективностью использовать сетевое оборудование**
Extensable Architecture	**Любые новые типы сетей могут быть поддержаны путем добавления соответствующих драйверов**
TransparentDistributed Processing	**Благодаря отсутствию разницы между передачей сообщений в пределах одного узла и между узлами нет необходимости вносить какие-либо изменения в приложения для того, чтобы они могли взаимодействовать через сеть**

Благодаря этой технологии сеть компьютеров с QNX фактически можно представлять как один виртуальный суперкомпьютер. Все ресурсы любого из узлов сети автоматически доступны другим, и для этого не нужны специальные «фокусы» с использованием технологии RPC. Это значит, что любая программа может быть запущена на любом узле, при этом её входные и выходные потоки могут быть направлены на любое устройство на любых других узлах [41].

Например, утилита make в QNX автоматически распараллеливает компиляцию пакетов из нескольких модулей на все доступные узлы сети, а затем собирает исполняемый модуль по мере завершения компиляции на узлах. Специальный драйвер, входящий в комплект поставки, позволяет использовать для сетевого взаимодействия любое устройство, с которым может быть ассоциирован файловый дескриптор, например последовательный порт, что открывает возможности для создания глобальных сетей.

Достигаются все эти удобства за счёт того, что поддержка сети частично обеспечивается и микроядром (специальный код в его составе позволяет QNX фактически объединять все микроядра в сети в одно ядро). Разумеется, за такие возможности приходится платить тем, что мы не можем получить драйвер для какой–либо сетевой платы от кого-либо ещё, кроме фирмы QSSL, то есть использоваться может только то оборудование, которое уже поддерживается. Однако ассортимент такого оборудования достаточно широк и периодически пополняется новейшими устройствами.

Когда ядро получает запрос на передачу данных процессу, находящемуся на удалённом узле, он переадресовывает этот запрос менеджеру Net, в подчинении которого находятся драйверы всех сетевых карт. Имея перед собой полную картину состояния всего сетевого оборудования, менеджер Net может отслеживать состояние каждой сети и динамически перераспределять нагрузку между ними. В случае когда одна из сетей выходит из строя, информационный поток автоматически перенаправляется в другую доступную сеть, что очень важно при построении высоконадёжных систем. Кроме поддержки своего собственного протокола, Net обеспечивает передачу пакетов TCP/IP, SMB и многих других, используя то же сетевое оборудование. Производительность QNX в сети приближается к производительности аппаратного обеспечения.

При проектировании системы реального времени, как правило, необходимо обеспечить одновременное выполнение нескольких приложений. В QNX/Neutrmo[1] параллельность выполнения достигается за счёт использования потоковой модели

[1] *Neutrino* – один из проектов микроядерной ОС.

POSIX, в которой процессы в системе представляются в виде совокупности потоков. Поток является минимальной единицей выполнения и диспетчеризации для ядра Neutrino, процесс определяет адресное пространство для потоков. Каждый процесс состоит минимум из одного потока. QNX представляет богатый набор функций для синхронизации потоков. В отличие от потоков само ядро не подлежит диспетчеризации. Код ядра исполняется только в том случае, когда какой-нибудь поток вызывает функцию ядра или при обработке аппаратного прерывания.

Напомним, что QNX базируется на концепции передачи сообщений. Передачу сообщений, а также их диспетчеризацию осуществляет ядро системы. Кроме того, ядро управляет временными прерываниями. Выполнение остальных функций обеспечивается задачами-администраторами. Программа, желающая создать задачу, посылает сообщение администратору задач (модуль task) и блокируется для ожидания ответа. Если новая задача должна выполняться одновременно с порождающей её задачей, администратор задач task создает её и, отвечая, выдает порождающей задаче идентификатор (id) созданной задачи. В противном случае никакого сообщения не посылается до тех пор, пока новая задача не закончится сама по себе. В этом случае в ответе администратора задач будут содержаться конечные характеристики закончившейся задачи.

Сообщения отличаются количеством данных, которые передаются от одной задачи точно к другой задаче. Данные копируются из адресного пространства первой задачи в адресное пространство второй, и выполнение первой задачи приостанавливается до тех пор, пока вторая задача не вернёт ответное сообщение. В действительности обе задачи кратковременно взаимодействуют во время выполнения передачи. Ничто, кроме длины сообщения (максимальная длина 65 535 байт), не заботит QNX при передаче сообщения. Существует несколько протоколов, которые могут быть использованы для сообщений.

Основные операции над сообщениями – это *послать, получить* и *ответить,* а также несколько их вариантов для обработки специальных ситуаций. Получатель всегда идентифицируется своим идентификатором задачи, хотя существуют способы ассоциировать имена с идентификатором задачи. Наиболее интересные вариан-

ты операций включают в себя возможность получать (копировать) только первую часть сообщения, а затем получать оставшуюся часть такими кусками, какие потребуются. Это может быть использовано для того, чтобы сначала узнать длину сообщения, а затем динамически распределить принимающий буфер. Если необходимо задержать ответное сообщение до тех пор, пока не будет получено и обработано другое сообщение, то чтение первых нескольких байт даёт вам компактный «обработчик», через который позже можно получить доступ ко всему сообщению. Таким образом, ваша задача предохраняется от того, чтобы хранить в себе большое количество буферов.

Другие функции позволяют программе получать сообщения только тогда, когда она уже ожидает их приема, а не блокироваться до тех пор, пока не прибудет сообщение, и транслировать сообщение к другой задаче без изменения идентификатора передатчика. Задача, которая транслировала сообщение, в транзакции невидима.

Кроме этого, QNX обеспечивает объединение сообщений в структуру данных, называемую очередью. Очередь – это просто область данных в третьей, отдельной задаче, которая временно принимает передаваемое сообщение и немедленно отвечает передатчику. В отличие от стандартной передачи сообщений, передатчик немедленно освобождается для того, чтобы продолжить свою работу. Задача администратора очереди хранит в себе сообщение до тех пор, пока приемник не будет готов прочитать его; это он делает путем запроса сообщения у администратора очереди. Любое количество сообщений (ограничено только возможностью памяти) может храниться в очереди. Они хранятся и передаются в том порядке, в котором они были приняты. Может быть создано любое количество очередей. Каждая очередь идентифицируется своим именем.

Помимо сообщений и очередей в QNX для взаимодействия задач и организации распределённых вычислений имеются так называемые порты, которые позволяют формировать сигнал одного конкретного условия, и механизм исключений, о котором мы уже упоминали выше.

Порт подобен флагу, известному всем задачам на одном и том же узле (но не на различных узлах). Он имеет точно два состояния, которые могут трактоваться как «присоединить» и «освободить», хотя пользователь может использовать свою интерпретацию; например, «занят» и «доступен». Порты используются для быстрой простой синхронизации между задачей и обработчиком прерываний устройства. Они нумеруются от нуля до максимум 32 (на некоторых типах узлов возможно и больше). Первые 20 номеров зарезервированы для использования операционной системой.

С портом могут быть выполнены три операции:

- ♦ присоединить порт;
- ♦ отсоединить порт;
- ♦ послать сигнал в порт.

Одновременно к порту может быть присоединена только одна задача. Если другая задача попытается «отсоединиться» от того же самого порта, то произойдёт отказ при вызове функции, и управление вернётся к задаче, которая в настоящий момент присоединена к этому порту. Это самый быстрый способ обнаружить идентификатор другой задачи, подразумевая, что задачи могут договориться использовать один номер порта. Напомним, что все рассматриваемые задачи должны находиться на одном и том же узле. При работе нескольких узлов специальные функции обеспечивают большую гибкость и эффективность.

Любая задача может посылать сигнал в любой порт независимо от того, была ли она присоединена к нему или нет (предпочтительно, чтобы не была присоединена). Сигнал подобен не блокирующей передаче пустого сообщения. То есть передатчик не приостанавливается, а приёмник не получает какие-либо данные; он только отмечает, что конкретный порт изменил своё состояние.

Задача, присоединённая к порту, может ожидать прибытия сигнала или может периодически читать порт. QNX хранит информацию о сигналах, передаваемых в каждый порт, и уменьшает счётчик после каждой операции «приёма» сигнала («чтение» возвращает счётчик и устанавливает его в нуль). Сигналы всегда принимаются перед сообщениями, давая им тем самым больший приоритет над сообще-

ниями. В этом смысле сигналы часто используются обработчиками прерываний для того, чтобы оповестить задачу о внешних (аппаратных) событиях (действительно, обработчики прерываний не имеют возможности посылать сообщения и должны использовать сигналы).

В отличие от описанных выше методов, которые строго синхронизируются, «исключения» обеспечивают асинхронное взаимодействие. То есть исключение может прервать нормальное выполнение потока задачи. Они, таким образом, являются аварийными событиями. QNX резервирует для себя 16 исключений для того, чтобы оповещать задачи о прерывании с клавиатуры, нарушении памяти и подобных необычных ситуациях. Остальные 16 исключений могут быть определены и использованы прикладными задачами.

Системная функция может быть вызвана для того, чтобы позволить задаче управлять своей собственной обработкой исключений, выполняя свою собственную внутреннюю функцию во время возникновения исключения.

Заметим, что функция исключения задачи вызывается асинхронно операционной системой, а не самой задачей. Вследствие этого исключения могут иметь сильнодействующее побочное влияние на операции (например, передачу сообщений), которые выполняются в это же время. Обработчики исключений должны быть написаны очень аккуратно.

Одна задача может установить одно или несколько исключений на другой задаче. Они могут быть комбинацией системных исключений (определённых выше) и исключений, определяемых приложениями, что обеспечивает другие возможности для межзадачного взаимодействия.

Благодаря такому свойству QNX, как возможность обмена посланиями между задачами и узлами сети, программы не заботятся о конкретном размещении ресурсов в сети. Это свойство придает системе необычную гибкость. Так, узлы могут произвольно добавляться и изыматься из системы, не затрагивая системные программы. QNX приобретает эту конфигурационную независимость благодаря применению концепции «виртуальных» задач. У виртуальных задач непосредственный код и данные, будучи на одном из удаленных узлов, возникают и ведут себя так,

как если бы они были локальными задачами какого-то узла со всеми атрибутами и привилегиями. Программа, посылающая сообщение в сети, никогда не посылает его точно. Сначала она открывает «виртуальную цепочку». Виртуальная цепочка включает все виртуальные задачи, связанные между собой. На обоих концах такой связи имеются буферы, которые позволяют хранить самое большое послание из тех, которые цепочка может нести в данном сеансе связи. Сетевой администратор помещает в эти буферы все сообщения для соединенных задач. Виртуальная задача, таким образом, занимает всего лишь пространство, необходимое для буфера и входа в таблице задач. Чтобы открыть связь, необходимо знать идентификатор узла и задачи, с которой устанавливается связь. Для этого необходимо знать идентификатор задачи администратора, ответственного за данную функцию, или глобальное имя сервера. Не раскрывая здесь подробно механизм обмена посланиями, добавим лишь, что наша задача может вообще выполняться на другом узле, где, допустим, имеется более совершенный процессор.

Контрольные вопросы и задачи

Вопросы для проверки

1 Изложите основные архитектурные особенности ОС UNIX.

2 Перечислите и поясните основные понятия системы UNIX.

3 Что делает системный вызов fork( )? Каким образом осуществляется в UNIX запуск новой задачи?

4 Изложите основные моменты, связанные с защитой файлов в UNIX.

5 Расскажите об особенностях семафоров в UNIX. Зачем в семафорных операциях действия осуществляются сразу над множеством семафоров?

6 Что представляет собой вызов удаленной процедуры (RPC)?

7 Какие механизмы использует OS/2, чтобы уменьшить потребности в оперативной памяти и увеличить производительность системы?

8 Почему про QNX часто говорят «сетевая» ОС?

9 Что такое сетевой протокол FLEET?

10 Какие функции реализует ядро QNX?

11 В чём вы видите принципиальные отличия между ядром Windows NT 4.0, которое считают построенным по микроядерным принципам, от ядра QNX?

12 Расскажите об основных механизмах, которые имеются в QNX для организации распределённых вычислений.

ПРИЛОЖЕНИЕ А

Тексты программы параллельных взаимодействующих задач

Здесь приведены тексты программы, реализующей параллельное функционирование задач, взаимодействующих между собой в соответствии с рис. 6.6.

```
program MultiThreads;
uses Forms,
ThrForm in 'ThrForm.pas' {Form1} ,
Threads in 'Threads.pas' ;
($R *.RES}
begin
   Application.Initialize;
   Application.CreateForm (Tform1,Form1);
   Application.Run;
end.
unit ThrForm;
interface
uses
   Windows, Messages, SysUtils, Classes, Graphics, Controls, Forms, Dialogs, Gauges,
StdCtrls, Threads;
type
   Tform1 = class (TForm)
      StrButton1: TButton;
      GaugeA: Tgauge;
      GaugeB: Tgauge;
      GaugeC: Tgauge;
      GaugeD: Tgauge;
      GaugeE: Tgauge;
      GaugeF: Tgauge;
      GaugeG: Tgauge
      Memo1: Tmemo;
      Label1: Tlabel;
      Label2: Tlabel;
      Label3: Tlabel;
      Label4: Tlabel;
      Label5: Tlabel;
      Label6: Tlabel;
      Label7: Tlabel;
      ScrollBar1: TScrollBar;
      Label8: Tlabel;
      Label9: Tlabel;
```

```
    procedure StrButtonIClick(Sender: TObject); { Запуск задач }
    procedure FormCreate(Sender: TObject);
    procedure ScrollBarlChange(Sender: TObject};
private
        { Private declarations }
public
        { Public declarations }
    end;
var
Forml: TForml;
inplementation
{$R *.DFM}
procedure TForml.StrButtonlCLick(Sender: TObject);
{ Запуск задач }
begin
        StrButtonl.Enabled =false;
        Memol.Clear;
        GaugeA.Progress:=0;
        GaugeB.Progress:=0;
        GaugeC.Progress:=0;
        GaugeD.Progress:=0;
        GaugeE.Progress:=0;
        GaugeF. Progress:=0;
        GaugeG.Progress:=0;
         AllStarts(GaugeA,GaugeB,GaugeC,GaugeD,GaugeE,GaugeF,GaugeG);
End;
procedure Tform1.FormCreate(Sender: TObject);
begin
         StartBtn:=StrButton1;
         Delay:=ScrollBar1.Position;
         MessageList:=Memo1;
end;
procedure Tform1.ScrollBarlChange(Sender: TObject);
begin
        Delay:=ScrollBarl.Position;
end;
end.
unit Threads;
{ Описание тредов }
interface
uses
    Classes,Gauges,StdCtrls,Syncobjs;
type
    TParams = record     { Параметры для инициализации треда}
        OwnName:char;
        ParentName:char;
        Length:integer;
end;
ThreadProgress = class(TThread)       { Основной объект-родитель }
private
        FGauge:TGauge;                { Строка состояния }
        FLength:integer;              { Длина задачи }
        FProgress:Longint;            { Текущее состояние задачи }
```

```
        Finished:boolean;               { Признак завершения задачи ]
        FName:char;                     { Имя задачи }
        FParentName:char;               { Имя запустившей задачи }
        FText: string;                  { Строка для вывода сообщений }
protected
        procedure Execute; override;          { Выполнение задачи }
        procedure DoVisualProgress;           { Прорисовка строки состояния}
        procedure WriteToMemo;                { Вывод сообщения }
        procedure Finish; virtual; abstract:  { Конечная часть задачи }
public
        constructor create(Gauge:TGauge; Params:TParams);
end;
TThreadA = class(ThreadProgress)
protected
    procedure Finish; override ;
end;
TThreadB = class(ThreadProgress)
protected
    procedure Finish; override;
end;
TThreadC = class(ThreadProgress)
protected
    procedure Finish; override:
end;
TThreadD = class(ThreadProgress)
protected
    procedure Finish; override;
end;
TThreadE = class(ThreadProgress)
protected
    procedure Finish; override;
end;
TThreadF = class(ThreadProgress)
protected
    procedure Finish; override;
end;
TThreadG = class(ThreadProgress )
protected
    procedure Finish; override;
end;
procedure AllStarts(GaugeA,GaugeB,GaugeC,GaugeD,GaugeE,GaugeF,GaugeG:TGauge);
    var StartBtn:TButton;
    MessageList:TMemo;
    Delay:integer;
    Params:Tparams;
    ThrA:TThreadA;
    ThrB:TthreadB;
    ThrC:TThreadC;
    ThrD:TThreadD;
    ThrE:TthreadE;
    ThrF:TThreadF;
    ThrG:TThreadG;
    CriticalSection: TCriticalSection;  { Данный объект позволяет организовать монопольный
```

```
                                        доступ к общим ресурсам }
implementation
procedure AllStarts(GaugeA,GaugeB,GaugeC,GaugeD,GaugeE,GaugeF,GaugeG:TGauge);
   { Начало выполнения задач }
begin
   Params.OwnName:= ‘A’;
   Params.ParentName: =' U ';
   Params.Length:=2;
   ThrA:=TThreadA.Create(GaugeA,Params);
   Params.OwnName:='B';
   Params.ParentName :='U';
   Params.Length:=4;
   ThrB:=TThreadB.Create(GaugeB,Params);
   Params.OwnName:='C';
   Params.ParentName:=’A';
   Params.Length:=2;
   ThrC:=TthreadC.Create(GaugeC,Params);
   Params.OwnName:='D';
   Params.ParentName:='A';
   Params.Length:=2;
   ThrD:=TThreadD.Create(GaugeD,Params);
   Params.OwnName:=’E’;
   Params.ParentName:=’A';
   Params.Length:=4;
   ThrE:=TThreadE.Create(GaugeE,Params) ;
   Params.OwnName:=’F’;
   Params.ParentName:= ' – ';
   Params. Length:=2;
   ThrF:=TThreadF.Create(GaugeF,Params);
   Params.OwnName:='G';
   Params.ParentName:=' – ';
   Params.Length:=1;
   ThrG :=TThreadG.Create(GaugeG,Params) ;
   CriticalSectton:=TCriticalSection.Create;
   ThrA.Resume;
   ThrB.Resume;
end;
{ ThreadProgress }
procedure ThreadProgress.Execute;
var i:integer;
begin
   FText :='Запущена задача '+Fname+' (задачей ’  +FParentName+')';
   Synchronize(WriteToMemo);
   for i:=1 to FLength do
   begin
      FProgress:=FProgress+1;
      Synchronize{doVisualProgress):
   End;
   Finish;
   Destroy;
end;
constructor ThreadProgress.create(Gauge:Tgauge;Params:TParams);
begin
```

```
    inherited create(true);
    FGauge:=Gauge;
    FLength:=delay*Params.Length;
    FGauge.MaxValue:=Flength;
    FGauge.MinValue:=0;
    FGauge.Progress:=0;
    FProgress:=0;
    FName:=Params.OwnName;
    FParentName:=Params.ParentName;
    Finished:=false;
end;
procedure ThreadProgress.DoVisualProgress;
begin
    FGauge.Progress:=Fprogress;
end;
procedure ThreadProgress.WriteToMemo:
begin
    MessageList.Lines.Add(FText);
end;
{ TThreadA }
procedure TThreadA.Finish;
begin
    ThrC.Resume;
    ThrD.Resume;
    ThrE.Resume;
    FText:= 'Задача A завершила работу и запустила задачи C,D,E';
    Synchronize(WriteToMemo);
    Finished:=true;
end;
{ TThreadB }
procedure TThreadB.Finish;
begin
    FText:='Задача B завершила свою работу';
    Synchronize(WriteToMemo);
    Finished:=true;
    CriticalSection.Enter;       { Начало защищенного блока команд }
    try
        if (ThrC.Fimshed=false)and(ThrD.Finished=false) then
        begin
            ThrF.FParentName:='B';
            repeat
                Synchronize(DoVlsual progress);
            until (ThrC.Finished)and(ThrD.Finished);
            ThrF.Resume;
            FText:='Задача B запустила задачу F';
            Synchronize{WriteToMemo);
        end;
    finally
        CriticalSection.Leave;      { Конец защищенного блока команд)
    end;
end;

{ TthreadC }
```

```
procedure TThreadC.Finish;
begin
   Ftext:='Задача С завершила свою работу';
   Synchronize(WriteToMenro);
   Finished:=true;
   CriticalSection.Enter;
   try
      if (ThrB.Finished=fa1se)and(ThrD.Finished=false) then
      begin
         ThrF.FparentName:='C';
         repeat
            Synchronize(DoVisualprogress};
         until (ThrB.Finished)and(ThrD.Finished);
         ThrF.Resume;
         FText:='Задача С запустила задачу F';
         Synchronize(WriteToMemo) ;
      end;
   finally
      CriticalSection.Leave;
   end;
end;
{ TthreadD }
procedure TThreadD.Finish;
begin
      FText:=' Задача D завершила свою работу';
      Synchronize(WriteToMemo);
      Finlshed:=true;
      CriticalSection.Enter;
      try
         if(ThrC.Finished=false)and(ThrB.Finished=false) then
         begin
            ThrF.FParentName:=' D ';
            repeat
               Synchronize(DoVisualprogress);
            unitil(ThrC.Finished)and(ThrB.Finished);
            ThrF.Resume;
            FText:='Задача D запустила задачу F';
            Synchronize(WriteToMemo);
         end;
      finally
         CriticalSection.Leave;
      end;
end;
{ TThreadE }
procedure TThreadE.Finish;
begin
   Finished:=true;
   CnticalSection.Enter;
   try
      if ThrF.Finished then
      begin
         ThrG.FParentName:='E';
         ThrG.Resume;
```

```
            FText:='Задача E завершила работу и запустила задачу G'
            Synchronize(WriteToMemo);
        end
        else
            begin
                FText:='Задача E завершила свою работу';
                Synchronize(WriteToMemo);
            end;
    finally
        CriticalSection.Leave;
    end;
end;
{ TThreadF }
procedure TThreadF.Finish;
begin
    Finished:=true;
    CriticalSection.Enter;
    try
        if ThrE.Finished then
            begin
                ThrG.FParentName:='F';
                ThrG.Resume;
                FText:='Задача F завершила работу и запустила задачу G';
                Synchronize(WriteToMemo);
            end
            else
                begin
                    FText:='Задача F завершила свою работу';
                    Synchronize(WriteToMemo);
                end;
    finally
                CriticaiSection.Leave;
    end;
end;
{ TThreadG }
procedure TThreadG.Finish;
    begin
        Finished:=true;
        Ftext:='Задача G завершила работу';
        Synchronize(WriteToMemo);
        StartBtn.Enabled:=true;
        CriticalSection.Free;
    end;
end.
```

ПРИЛОЖЕНИЕ Б

Тексты программ комплекса параллельных взаимодействующих приложений

Здесь приведены тексты следующих программ: A, B, D и G. Эти программы отражают алгоритмы взаимодействия, заданные на рис. 6.7. Остальные программы содержат аналогичные алгоритмы взаимодействия.

Текст программы A

```
#define INCL_BASE
#define INCL_DOS
#include <os2.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "progr.h"
#include "func.h"
/*Глобальные переменные*/
 char myword[ ]=PROGR_A.EXE";
BYTE atrib[2];
/*Главная функция*/
int main( )
{
  /*Локальные переменные*/
   int i,j; PipeSize=250; strcpy(Word,myword); atrib[0]='  '; atrib[1]=0x02f;
  VioScrollUp(0,0,25,80,25,&atrib,0);
  /*Фиксация времени начала процесса*/
  DosGetDateTime(&Time);
  strcpy(BegTime,itoa(Time.hours,MyBuffer,10));strcat(BegTime,":");
  strcat(BegTime,itoa(Time.minutes,MyBuffer,10)); strcat(BegTime,":");
  strcat(BegTime,itoa(Time.seconds,MyBuffer,10)); strcat(BegTime,":");
  strcat(BegTime,itoa(Time.hundredths,MyBuffer,10));
  /*Установка одинакового приоритета всех подпроцессов*/
  rc=DosSetPriority(PRTYS_PROCESSTREE,2,10,0);
    if(rc!=0) (HaltSystem(“Set Priority",rc); }
  /* Создание файла, доступного всем подпроцессам */
  rc=DosOpen("result.txt",&FileOpen,&Action,0,0,0x0010|0x0002,0x0002|0x0040|0x4000.0);
    if (rc!=0) {HaltSystem(“DosOpen",rc); }
    /*Создание семафоров*/
  rc=DosCreateEventSem("\\SEM32\\PIPESEM",&PipeSem,DC_SEM_SHARED, 0);
    if (rc!=0) {HaltSystem("Create Sem PipeSem",rc); }
  rc=DosCreateEventSem("\\SEM32\\EXITSEM", &ExitSem, DC_SEM_SHARED, 0);
    if (rc!=0) { HaltSystem("Create Sem (ExtiSem)",rc);}
  rc=DosCreateEventSem("\\SEM32\\WRITEFILE", &FileSem, DC_SEM_SHARED, 0);
    if (rc!=0) { HaltSystem( "Create Sem (FileSem)",rc);}
  rc=DosCreateEventSem("\\SEM32\\POINT2SEM",&Point2Sem, DC_SEM_SHARED, 0);
    if (rc!=0) { HaltSystem("Create Sem (Point2Sem)",rc);}
```

```
rc=DosCreateEventSem("\\SEM32\\POINT3SEM",&Point3Sem, DC_SEM_SHARED, 0);
  if (rc!=0) { HaltSystem("Create Sem (Point3Sem)",rc);}
rc=DosCreateEventSem("\\SEM32\\POINTlSEM",&Point1Sem, DC_SEM_SHARED, 0);
  if (rc!=0) { HaltSystem("Create Sem (Point1Sem)",rc);}
rc=DosPostEventSem(PipeSem); if (rc!=0) {HaltSystem(“PostSem (PipeSem)",rc); }
  rc=DosPostEventSem(FileSem); if (rc!=0) {HaltSystem(“PostSem (FileSem)",rc); }
   /*Создание Pipe (транспортера)*/
rc=DosCreatePipe(&ReadHandle,&WriteHandle,PipeSize);
  if (rc!=0) { HaltSystem("Create Pipe",rc);}
  /*3адание строк аргументов*/
for (i=0;i<=strlen(myword);i++ ) { Argument[i]=myword[i]; }
j=i; Argument2=itoa(ReadHandle,MyBuffer,10);
for (i=0;i<=strlen(Argument2);i++) { Argument[j+1]=MyBuffer[i]; }
j+=1; Argument[j-1]='  ';
Argument2=itoa (WriteHandle, MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++)  { Argument [i+j],MyBuffer[i]; }
j+1; Argument [j-1]='  '; Argument2=itoa (FileOpen, MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++)  { Argument[i+j],MyBuffer[i]; }
/*Вывод сообщений и выполнение некоторого процесса (цикла)*/
strcpy(Pmessege,"Программа "); strcat(Pmessege,myword);
VioWrtNAttr(&atr,80*25,0,0,0); VioWrtCharStr(PMessege.sizeof(PMessege),2,1,0);
strcpy(Pmessege,"3апущена в "); strcat(Pmessege,BegTime);
VioWrtNAttr(&atr,80*25,0,0,0); VioWrtCharStr(PMessege,sizeof(PMessege),3,1,0);
strcpy(Fmessege1,myword); strcat(Fmessege1,"  "); strcat (FMessege1,PMessege);
strcpy(Pmessege,"Программой "); strcat(Pmessege,"                 ");
VioWrtNAttr(&atr,80*25,0,0,0); VioWrtCharStr(Pmessege,sizeof(PMessege),4,1,0) ;
strcpy(FMessege2,myword); strcat(FMessege2,"  "); strcat (FMessege2,PMessege);
for(i=0;i<22;i++)
{ VioWrtNAttr(&atr, 80*25,0,0,0); VioWrtCharStr("|", 1,5,1+i,0);
for (j=0;j<(12500*Speed);j++); }
/*Получение времени загрузки программ потомков*/
DosGetDateTime(&Time);
strcpy(MyTime,itoa(Time.hours,MyBuffer,10)); strcat(MyTime,":");
strcat(MyTime.itoa(Time.minutes,MyBuffer,10)); strcat(MyTime,":");
strcat(MyTime.itoa(Time.seconds,MyBuffer,10)); strcat(MyTime,":");
strcat(MyTime,itoa(Time.hundredths,MyBuffer,10)); strcpy(Inform.LaunchTime,MyTime);
strcpy(Inform.ParentName,myword); Inform.Number=4;
strcpy(PMessege,"Завершена в "); strcat(Pmessege,MyTime);
VioWrtNAttr(&atr,80*25,0,0,0); VioWrtCharStr(PMessege,sizeof(PMessege),6,10);
strcpy(FMessege3,myword); strcat(FMessege3,"  "); strcat (FMessege3,PMessege);
/*0жидание доступности записи в файл и немедленная запись в файл*/
do{ DosWaitEventSem(FileSem,-1);
rc=DosResetEventSem(FileSem,&BytesWritten);
} while (rc!=0);
DosWrite(FileOpen,(PVOID)&Fmessege1,sizeof(Fmessege1),&BytesWritten);
DosWrite(FileOpen,(PVOID)&FMessege2,sizeof(FMessege2),&BytesWritten);
DosWrite(FileOpen,(PVOID)&FMessege3,sizeof(FMessege3),&BytesWritten);
DosPostEventSem(FileSem);
/*3апись сообщения в Pipe: имя предка и время запуска программ*/
rc=DosWrite(WriteHandle,(PVOID)&Inform,sizeof (Inform), &BytesWritten);
if (rc!=0) { HaltSystem("(DosWrite)",rc); }
/*3апуск программ - потомков*/
rc=DosExecPgm(FailFileb,sizeof(FailFileb),1, Argument, 0,&ResCodeb,"progr_b.exe");
```

```
if (rc!=0) { HaltSystem("(DosExecPgm) B",rc);}
rc=DosExecPgm(FailFileb,sizeof (FailFileb),1,Argument, 0, &ResCodeb,"progr_c.exe");
if (rc!=0) { HaltSystem("(DosExecPgm) C",rc);}
rc=DosExecPgm(FalilFileb,spzeof(FailFileb),1,Argument,0, &ResCodeb,"progr_i.exe");
if (rc!=0) { HaltSystem("(DosExecPgm) I",rc);}
rc=DosExecPgm(FailFileb,sizeof(FailFileb),1, Argument,0,&ResCodeb,"progr_j.exe");
if (rc!=0) { HaltSystem("(DosExecPgm) J",rc);}
/*0жидание загрузки семафоров в программах-потомках*/
DosWaitEventSem(ExitSem,-1);
DosCloseEventSem(ExitSem);
return(0);}
```

Текст программы B

```
#define INCL_BASE
#define INCL_DOS
#include <os2.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "progr.h"
#include "func.h"
/*Глобальные переменные*/
char myword[ ]="PROGR_B.EXE";
/*Главная функция*/
int main(int argc, char* argv[ ], char* envp[ ])
{
  /*Локальные переменные*/
  int i,j;  strcpy(Word,myword);
  /*Инициализация семафоров*/
  rc=DosOpenEventSem("\\SEM32\\PIPESEM", &PipeSem);
     if (rc!=0) { HaltSystem(“Open Sem (PipeSem) ", rc); }
  rc=DosOpenEventSem("\\SEM32\\EXITSEM", &ExitSem);
     if (rc!=0) { HaltSystem("Open Sem (ExitSem) ", rc);}
  rc=DosOpenEventSem("\\SEM32\\WRITEFILE", &FileSem);
     if (rc!=0) { HaltSystem("Open Sem (FileSem) ", rc);}
  rc=DosOpenEventSem("\\SEM32\\POINT1SEM", &Point1Sem);
     if (rc!=0) { HaltSystem("Create Sem (Point1Sem) ", rc);}
  rc=DosOpenEventSem("\\SEM32\\POINT2SEM", &Point2Sem);
     if (rc!=0) { HaltSystem("Create Sem (Point2Sem) ", rc);}
  rc=DosOpenEventSem("\\SEM32\\POINT3SEM", &Point3Sem);
     if (rc!=0) { HaltSystem("Create Sem (Point3Sem) ", rc);}
  /*Проверка количества аргументов */
     if (argc!=4) { HaltSystem( "Ошибка командной строки",rc);}
  /*Инициализация переменных для записи в транспортер и файл*/
  WriteHandle=atoi(argv[2]); ReadHandle=atoi(argv[1]); FileOpeп=atoi(argv[3]);
/*0жидание свободного транспортера*/
do { DosWaitEventSem(PipeSem, -1);
```

```
      rc=DosResetEventSem(PipeSem, &NPost);
      } while (rc!=0);
/* Работа с транспортером*/
rc=DosRead(ReadHandle, (PVOID)&OldInform, sizeof(OldInform), &BytesReaden);
DosPostEventSem(PipeSem); if (rc!=0) { HaltSystem("Read Pipe", rc);}
/*Уведомление предка о завершении инициализации*/
if (OldInform.Number==1)
      { rc=DosPostEventSem(ExitSem);
        if (rc!=0) { HaltSystem(“PostSem (ExitSem) ", rc);} }
         else { do { OldInform.Number--;
                     DosWaitEventSem(PipeSem,-1) ;
                     Rc=DosResetEventSem(PipeSem,&NPost);
      } white (rc!=0);
rc=DosWrite(WriteHandle,(PVOID)&OldInform,sizeof(OldInform),&BytesWritten);
      DosPostEventSem(PipeSem);
      if (rc!=0) { HaltSystem("Write Pipe", rc);}   }
/*3адание строк аргументов*/
for (i=0;i<=strlen(myword);i++ ) { Argument [i]=myword[i]; }
j=i; Argument2=itoa(ReadHandle.MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++) { Argument[j+i]='MyBuffer[i]; }
j+=i; Argument[j-1]='  '; Argument2=itoa(WriteHandle, MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++) { Argument[i+j]=MyBuffer[i]; }
j+=i; Argument[j-1]='  '; Argument2=itoa(FileOpen, MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++)  { Argument [i+j], MyBuffer[i]; }
/*Выполнение процесса - цикл */
strcpy(Pmessege, "Программа "); strcat(Pmessege,myword);
VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr(Pmessege, sizeof(PMessege), 2, 27,0);
strcpy(Pmessege,"Запущена в "); strcat(Pmessege,OldInform.LaunchTime);
VioWrtNAttr(&atr,80*25, 0, 0, 0); VioWrtCharStr(PMessege.sizeof(PMessege), 3, 27, 0);
strcpy(Fmessege1, myword); strcat(Fmessege1, " "); strcat (Fmessegel, PMessege);
strcpy(Pmessege, "Программой "); strcat(PMessege.OldInform.ParentName);
VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr(Pmessege, sizeof(PMessege), 4, 27, 0);
strcpy(FMessege2,myword); strcat(FMessege2, " "); strcat (FMessege2, PMessege);
for(i=0;i<22;i++)
      { VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr("|", 1, 5, 27+i, 0);
        for (j=0;j<(25000*Speed);j++);    };
 /*Получение времени загрузки программ-потомков*/
DosGetDateTime(&Time);
strcpy(MyTime, itoa(Time.hours, MyBuffer, 10)); strcat(MyTime, ":");
strcat(MyTime, itoa(Time.minutes, MyBuffer, 10)); strcat(MyTime, ":");
strcat(MyTime, itoa(Time.seconds, MyBuffer, 10)); strcat(MyTime, ";");
strcat(MyTime, itoa(Time.hundredths, MyBuffer, 10));
strcpy(NewInform.LaunchT1me,MyT1me);
strcpy(NewInform.ParentName, myword); NewInform.Number=3;
strcpy(PMessege."Завершена в "); strcat(PMessege.MyTime);
VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr(Pmessege, sizeof(PMessege), 6, 27, 0);
strcpy(FMessege3, myword); strcat(FMessege3, " "); strcat (FMessege3, PMessege);
 /*0жидание доступности записи в файл и немедленная запись в файл*/
 do { DosWaitEventSem(FileSem, -1);
        rc=DosResetEventSem(FileSem, &BytesWritten);
      } while (rc!=0);
DosWrite(FileOpen, (PVOID)&Fmessege1,sizeof(Fmessege1), &BytesWritten);
```

```
  DosWrite(FileOpen, (PVOID)&FMessege2, sizeof(FMessege2), &BytesWritten);
  DosWrite(FileOpen, (PVOID)&FMessege3, sizeof(FMessege3), &BytesWritten);
  DosPostEventSem(FileSem);
  /*Алгоритм прохождения точки 2*/
  rc=DosPostEventSem(Point1Sem);
  if (rc==0) {
  /*Запись сообщения в Pipe; имя предка и время запуска программ*/
  rc=DosWrite(WriteHandle, (PVOID)&NewInform, sizeof(NewInform), &BytesWritten);
    if (rc!=0) { HaltSystem("(DosWrite)", rc); }
    /*Создание семафора ожидания инициализации ресурсов потомками*/
  r=DosCreateEventSem("\\SEM32\\EXITSEM2", &ExitSem2, DC_SEM_HARED, 0);
       if (rc!=0) { HaltSystem("Create Sem (ExitSem2) ", rc);}
  /*Запуск программ-потомков*/
  rc=DosExecPgm(FailFileb.sizeof (FailFileb), 1, Argument, 0, &ResCodeb, "progr_d.exe");
       if (rc!=0) { HaltSystem("(DosExecPgm)", rc);}
  rc=DosExecPgm(FailFileb,sizeof(FailFileb), 1, Argument, 0, &ResCodeb, "progr_e.exe");
       if (rc!=0) { HaltSystem("(DosExecPgm)", rc);}
  rc=DosExecPgm(FailFileb, sizeof(FailFileb), 1, Argument, 0, &ResCodeb, "progr_f.exe");
       if (rc!=0) { HaltSystem("(DosExecPgm)", rc);}
  /*0жидание инициализации ресурсов потомками*/
  DosWaitEventSem(ExitSem2,-1);
  DosCIoseEventSem(ExitSem2);
}
return(0);}
```

Текст программы D

```
#define INCL_BASE
#define INCL_DOS
#include <os2.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "progr.h"
#include "func.h"
/*Глобальные переменные*/
char myword[ ]="PROGR_D.EXE";
/*Главная функция*/
int main(int argc, char *argv[ ], char *envp[ ])
{
  /*Локальные переменные*/
  int i, j; strcpy(Word, myword);
  /Инициализация семафоров*/
  rc=DosOpenEventSem("\\SEM32\\PIPESEM", &PipeSem);
       if (rc!=0) { HaltSystem("Open Sem (PipeSem) ", rc); }
  rc=DosOpenEventSem("\\SEM32\\WRITEFILE", &FileSem);
       if (rc!=0) { Hal tSystem( "Open Sem (FileSem) ", rc);}
  rc=DosOpenEventSem("\\SEM32\\POINT2SEM", &Point2Sem);
       if (rc!=0) { HaltSystem("Create Sem (Point2Sem) ", rc);}
  rc=DosOpenEventSem("\\SEM32\\POINT3SEM", &Point3Sem);
       if (rc!=0) { HaltSystem("Create Sem (Point3Sem) ", rc);}
  rc=DosOpenEventSem("\\SEM32\\POINT1SEM", &Point1Sem);
```

```
     if (rc!=0) { HaltSystem("Create Sem (Potnt1Sem) ", rc);}
rc=DosOpenEventSem("\\SEM32\\EXITSEM2", &ExitSem2);
     if (rc!=0) { HaltSystem("Create Sem (ExitSem2) ", rc);}
/*Проверка количества аргументов */
if (argc!=4) { HaltSystem( "Ошибка командной строки", rc);}
/*Инициализация переменных для записи в транспортер и файл*/
FileOpen=atoi(argv[3]); WriteHandle=atoi(argv[2]); ReadHandle=atoi(argv[1]);
/*0жидание свободного транспортера*/
do { DosWaitEventSem(PipeSem, -1);
rc=DosResetEventSem(PipeSem, &NPost);
} while (rc!=0);
/* Работа с Pipe каналом*/
rc=DosRead(ReadHandle, (PVOID)&0ldInform, sizeof(0ldInform), &BytesReaden);
DosPostEventSem(PipeSem); if (rc!=) { HaltSystem("Read Pipe", rc);}
/*Уведомление предка о завершении инициализации*/
if (0ldInform.Number==1)
  { rc=DosPostEventSem(ExitSem2);
     if (rc!=0) { HaltSystem(“PostSem (ExitSem2) ", rc);} }
     else { do { 0ldInform.Number--;
                  DosWaitEventSem(PipeSem, -1);
                  Rc=DosResetEventSem(PipeSem,&NPost);
                } while (rc!=0);
rc=DosWrite(WriteHandle,(PVOID)&OldInform, sizeof(OldInform), &BytesWritten);
DosPostEventSem(PipeSem);
     if (rc!=0) { HaltSystem("Write Pipe", rc);}   }
/*3адание строк аргументов*/
for (i=0;i<=strlen(myword);i++ ) { Argument[i]=myword[i]; }
j=1; Argument2=itoa(ReadHandle, MyBuffer,10);
for (i=0;i<=strlen(Argument2);i++) { Argument [j+i]=MyBuffer[i]; }
j+=i; Argument[j-1]='  ';  Argument2=itoa (WriteHandle, MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++) { Argument[i+j]=MyBuffer[i]; }
j+=i; Argument[j-1]='  ‘; Argument2=itoa (FileOpen, MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++) { Argument[i+j]=MyBuffer[i]; }
/*Выполнение процесса - цикл */
strcpy(Pmessege, "Программа "); strcat(Pmessege,myword);
VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr(Pmessege, sizeof(PMessege), 8, 54,0);
strcpy(PMessege, "3апущена в "); strcat(Pmessege, 0ldInform.LaunchTime);
VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr(PMessege, sizeof(PMessege), 9, 54, 0);
strcpy(Fmessege1, myword); strcat(Fmessege1, " "); strcat(Fmessege1, PMessege);
strcpy(Pmessege, "Программой "); strcat(Pmessege, OldInform.ParentName);
VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr(Pmessege, sizeof(PMessege), 10, 54, 0);
strcpy(FMessege2, myword); strcat(FMessege2, " "); strcat (FMessege2, PMessege);
     for(i=0;i<22;i++)
         { VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr("|", 1, 11, 54+i, 0);
           for (j=0;j<(15000*Speed);j++);  };
/Получение времени загрузки программ-потомков*/
DosGetDateTime(&Time);
strcpy(MyTime, itoa(Time.hours, MyBuffer, 10)); strcat(MyTime, ":");
strcat(MyTime, itoa(Time.minutes, MyBuffer, 10)); strcat(MyTime, ":");
strcat(MyTime, itoa(Time.seconds, MyBuffer, 10)); strcat(MyTime,":");
strcat(MyTime, itoa(Time.hundredths, MyBuffer,10));
strcpy(NewInform.LaunchTime, MyTime);
strcpy(NewInform.ParentName, myword); NewInform.Number=2;
```

```
  strcpy(Pmessege, "Завершена в "); strcat(PMessege, MyTime);
  VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr(Pmessege, sizeof(PMessege), 12, 54, 0);
  strcpy(FMessege3, myword); strcat(FMessege3, " "); strcat (FMessege3, PMessege);
  /*0жидание доступности записи в файл и немедленная запись в файл*/
  do{ DosWaitEventSem(FileSem, -1);
       rc=DosResetEventSem(FileSem, &BytesWritten);
  } while (rc!=0);
  Doswrite(FileOpen, (PVOID)&Fmessege1, sizeof(Fmessege1), &BytesWritten);
  Doswrite(FileOpen, (PVOID)&FMessege2, sizeof(FMessege2), &BytesWritten);
  DosWrite(FileOpen, (PVOID)&FMessege3, sizeof(FMessege3), &BytesWritten);
  DosPostEventSem(FileSem):
  /* Алгоритм прохождения точки 3*/
  rc=DosPostEventSem(Point2Sem) ;
  if (rc==0)
     { do{ DosQueryEventSem(Point2Sem, &BytesWritten):
     }while (BytesWritten<=2);
  /*3апись сообщения в Pipe: имя предка(Progr_A) и время запуска программ*/
  rc=DosWrite(WriteHandle, (PVOID)&NewInform, sizeof(NewInform), &BytesWritten);
       if (rc!=0) { HaltSystem("(DosWrite)", rc); }
  /*Создание семафора ожидания инициализации ресурсов потомками*/
  re=DosCreateEventSem("\\SEM32\\EXITSEM3", &ExitSem3, DC_SEM_SHARED, 0);
  if (re!=0) { HaltSystem("Create Sem (ExitSem3) ", rc);}
  /*3апуск программ-потомков*/
  rc=DosExecPgm(FailFileb, sizeof(FailFileb), 1, Argument, 0, &ResCodeb, "progr_g.exe");
       if (rc!=0) {HaltSystem("(DosExecPgm)", rc);}
  rc=DosExecPgm(FailFileb, sizeof(FailFileb), 1, Argument, 0, &ResCodeb, "progr_h. exe");
       if (rc!=0) { HaltSystem("(DosExecPgm)", rc);}
  /*0жидание инициализации ресурсов потомками*/
  DosWaitEventSem(ExitSem3, -1);
  DosCloseEventSem(ExitSem3);
  }return(0);}
```

Текст программы G

```
#define INCL_BASE
#define INCL_DOS
#include <os2.h>
#include <string.h>
#include <stdio.h>
#include <stdlib.h>
#include "progr.h"
#include "func.h"
/*Глобальные переменные*/
char myword[]="PROGR_G.EXE":
/*Главная функция*/
int main(int argc, char* argv[ ], char*envp[ ])
{
  /*Локальные переменные*/
  int i,j; strcpy(Word, myword);
  /Инициализация семафоров*/
  rc=DosOpenEventSem("\\SEM32\\PIPESEM", &PipeSem);
    if (rc!=0) { HaltSystem(“Open Sem (PipeSem) ", rc); }
  rc=DosOpenEventSem("\\SEM32\\EXITSEM3", &ExitSem3);
```

```
  if (rc!=0) { HaltSystem(“Open Sem (ExitSem3) ", rc);}
rc=DosOpenEventSem("\\SEM32\\WRITEFILE", &FileSem);
  if (rc!=0) { Hal tSystem( "Open Sem (FileSem) ", rc);}
rc=DosOpenEventSem("\\SEM32\\POINT2SEM", &Point2Sem);
      if (rc!=0) { HaltSystem("Create Sem (Point2Sem) ", rc);}
rc=DosOpenEventSem("\\SEM32\\POINT3SEM", &Point3Sem);
if (rc!=0) { HaltSystem("Create Sem (Point3Sem) ", rc);}
rc=DosOpenEventSem("\\SEM32\\POINT1SEM", &Point1Sem);
      if (rc!=0) { HaltSystem("Create Sem (Point1Sem) ", rc);}
/* Проверка количества аргументов */
if (argc!=4) { HaltSystem( "Ошибка командной строки", rc);}
/*Инициализация переменных для записи в транспортер и файл*/
WriteHandle=atoi(argv[2]); ReadHandle=atoi(argv[1]);FileOpen=atoi(argv[3]);
/*0жидание свободного транспортера*/
do { DosWaitEventSem(PipeSem, -1);
       rc=DosResetEventSem(PipeSem, &NPost);
     } while (rc!=0);
/* Работа с Pipe каналом*/
rc=DosRead(ReadHandle, (PVOID)&0ldInform, sizeof(0ldInform), &BytesReaden);
DosPostEventSem(PipeSem);
if (rc!=0) (HaltSystem(“Read Pipe", rc);}
/*Уведомление предка о завершении инициализации*/
if (OldInform.Number==1)
     { rc=DosPostEventSem(ExitSem3);
       if (rc!=0) { HaltSystem(“PostSem (ExitSem3) ", rc);} }
       else { do { Old Inform.Number--;
                   DosWaitEventSem(PipeSem, -1);
                   Rc=DosResetEventSem(PipeSem, &NPost);
             } while (rc!=0);
       rc=DosWrite(WriteHandle, (PVOID)&0ldInform, sizeof(0ldInform), &BytesWritten);
       DosPostEventSem(PipeSem);
       if (rc!=0) { HaltSystem("Write Pipe", rc);}   }
/*3адание строк аргументов*/
for (i=0;i<=strlen(myword);i++ ) { Argument[i]=myword[i]; }
j=i; Argument2=itoa(ReadHandle, MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++) { Argument[j+i]=MyBuffer[i]; }
j+=i; Argument[j-i]='  '; Argument2=itoa(WriteHandle, MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++) { Argument [i+j]=MyBuffer[i]; }
j+=i; Argument[j-i]='  '; Argument2=itoa(FileOpen, MyBuffer, 10);
for (i=0;i<=strlen(Argument2);i++) { ArgumentEH-jl-MyBufferCi]; }
/*Выполнение процесса - цикл */
strcpy(Pmessege, "Программа "); strcat(Pmessege, myword);
VioWrtNAttr(&atr, 80*25, 0, 0, 0); VioWrtCharStr(Pmessege, sizeof(PMessege), 14, 54, 0);
strcpy(Pmessege, "Запущена в "); strcat(Pmessege, OldInform.LaunchTime);
VioWrtNAttr(&atr, 80*25, 0, 0, 0);
VioWrtCharStr(Pmessege, sizeof(PMessege), 15, 54, 0);
strcpy(Fmessege1, myword); strcat(Fmessege1, "  "); strcat (Fmessege1, PMessege);
strcpy(Pmessege, "Программой "); strcat(Pmessege, 0ldInform.ParentName);
VioWrtNAttr(&atr, 80*25, 0, 0, 0);
VioWrtCharStr(Pmessege, sizeof(PMessege), 16, 54, 0);
strcpy(FMessege2, myword); strcat(FMessege2, " "); strcat (FMessege2, PMessege);
for(i=0;i<22;i++)
   { VioWrtNAttr(&atr, 80*25, 0, 0, 0);VioWrtCharStr("|",1, 17, 54+i, 0) ;
```

```
      for (j=0;j<(25000*Speed);j++); };
 /*Получение времени загрузки программ-потомков*/
 DosGetDateTime(&Time);
 strcpy(MyTime, itoa(Time.hours, MyBuffer, 10)); strcat(MyTime, ":");
 strcat(MyTime, itoa(Time.minutes, MyBuffer, 10)); strcat(MyTime,":");
 strcat(MyTime, itoa(Time.seconds, MyBuffer, 10)); strcat(MyTime,":");
 strcat(MyTime, itoa(Time.hundredths, MyBuffer,10));
 strcpy(NewInform.LaunchTt me, MyTime);
 strcpy(NewInform.ParentName, myword); NewInform.Number=1;
 strcpy(Pmessege, "Завершена в "); strcat(Pmessege, MyTime);
 ViowrtNAttr(&atr, 80*25, 0, 0, 0);
 VioWrtCharStr(Pmessege, sizeof(PMessege), 18, 54, 0);
 strcpy(FMessege3, myword); strcat(FMessege3, " ");
 strcat (FMessege3, PMessege);
 /*0жидание доступности записи в файл и немедленная запись в файл*/
 do{ DosWaitEventSem(FileSem, -1);
      rc=DosResetEventSem(FileSem, &BytesWritten);
    } while (rc!=0);
 DosWrite(FileOpen, (PVOID)&Fmessege1, sizeof(Fmessege1), &BytesWritten);
 DosWrite(FileOpen, (PVOID)&FMessege2, sizeof(FMessege2), &BytesWritten);
 DosWrite(FileOpen, (PVOID)&FMessege3, sizeof(FMessege3), &BytesWritten);
 DosPostEventSem(FileSem);
 /*Алгоритм прохождения точки 4*/
 do { DosWaitEventSem(Point1Sem, -1);
       rc=DosResetEventSem(Point1Sem,&BytesReaden);
      } while (rc!=0);
 DosPostEventSem(Point3Sem);
 DosQueryEventSem(Point3Sem, &BytesWritten);
 DosPostEventSem(Point1Sem);
 if (BytesWritten==4)
    {
     /*Запись сообщения в Pipe; имя предка(Progr_A) и время запуска программ*/
    rc=DosWrite(WriteHandle, (PVOID)&NewInform, sizeof(NewInform), &BytesWritten);
      if (rc!=0) { HaltSystem("(DosWrite)", rc); }
 /*Создание семафора ожидания инициализации ресурсов потомками*/
 rc=DosCreateEventSem("\\SEM32\\EXITSEM4", &ExitSem4, DC_SEM_SHARED, 0);
 if (rc!=0) { HaltSystem("Create Sem (ExitSem4) ", rc);}
 /*Запуск программ-потомков*/
 rc=DosExecPgm(FailFileb, sizeof(FailFileb), 1, Argument, 0, &ResCodeb, "progr_k.exe") ;
      if (rc!=0) { HaltSystem("(DosExecPgm)", rc);}
 /*0жидание инициализации ресурсов потомками*/
 DosWaitEventSem(ExitSem4, -1);
 DosCloseEventSem(ExitSem4);
 }return(0);}
```

Список литературы

1 *Абрамова Н. А.* и др. Новый математический аппарат для анализа внешнего поведения и верификации программ – М.: Институт проблем управления РАН, 1998. - 109 с.

2 *Александров Е. К., Рудня Ю. Л.* Микропроцессор 80386: как он работает и как работают с ним: Учебное пособие / Под ред. проф. Д. В. Пузанкова. – С-Пб.: Элмор, 1994. – 274 с.

3 *Эпплман Д.* Win32 API и Visual Basic. – СПб.: Питер, 2001.

4 *Кэнту М.* Delphi 5 для профессионалов. – СПб.: Питер, 2001.

5 *Афанасьев А. Н.* Формальные языки и грамматики: Учебное пособие. – Ульяновск: УлГТУ, 1997. – 84 с.

6 *Ахо А.*, *Ульман Дж.* Теория синтаксического анализа, перевода и компиляции. – М.: Мир, 1978. – т.1, 612 с. – т.2, 487 с.

7 *Бартеньев О. В.* Фортран для студентов. – М.: Диалог-МИФИ, 1999. – 342 с.

8 *Павловская Т. А.* C/C++: Учебник. – СПб.: Питер, 2001.

9 *Богумирский Б. С.* Руководство пользователя ПЭВМ: В 2-х ч. – С-Пб.: Ассоциация OILCO, 1992. – 357 с.

10 *Бранденбау Дж.* JavaScript: сборник рецептов для профессионалов. – СПб.: Питер, 2000. – 416 с.

11 *Браун С.* Операционная система UNIX. – М.: Мир, 1986. – 463 с.

12 *Бржезовский А. В., Корсакова Н. В.*, *Филъчаков В. В.* Лексический и синтаксический анализ. Формальные языки и грамматики. – Л.: ЛИАП, 1990. – 31 с.

13 *Бржезовский А. В.*, *Фильчаков В. В.* Концептуальный анализ вычислительных систем. – СПб.: ЛИАП, 1991. – 78 с.

14 *Вирт Н.* Алгоритмы и структуры данных. – М.: Мир, 1989. – 360 с.

15 *Волкова И. Л.*, *Руденко Т. В.* Формальные языки и грамматики. Элементы теории трансляции. – М.: Диалог-МГУ, 1999. – 62 с.

16 *Гончаров А.* Самоучитель HTML. – СПб.: Питер, 2000. – 240 с.

17 *Гордеев А. В., Молчанов А. Ю.* Применение сетей Петри для анализа вычислительных процессов и проектирования вычислительных систем: Учебное пособие. – Л.: ЛИАП, 1993. – 80 с.

18 *Гордеев А. В.*, *Никитин А. В., Фильчаков В. В.* Организация пакетов прикладных программ: Учебное пособие. – Л.: ЛИАП, 1988. – 78 с.

19 *Гордеев А. В., Кучин Н. В.* Проектирование взаимодействующих процессов в операционных системах: Учебное пособие. – Л.: ЛИАП, 1991. – 72 с.

20 *Гордеев А. В., Решетникова Н. Н., Соловьев А. П.* Дисковая операционная система реального времени. – СПб.: ГААП, 1994. – 44 с.

21 *Гордеев* А. *В., Штепен В. А.* Управление процессами в операционных системах реального времени: Учебное пособие. – Л.: ЛИАП, 1988. – 76 с.

22 *Григорьев В. Л.* Микропроцессор i486. Архитектура и программирование (в 4-х кн.). – М.: Гранал, 1993.

23. *Грис Д.* Конструирование компиляторов для цифровых вычислительных машин. – М.: Мир, 1975. – 544 с.

24 *Гудмэн Дж.* Секреты жесткого диска. – Киев: Диалектика, 1994. – 256 с.

25 *Д. Ван Тассел.* Стиль, разработка, эффективность, отладка и испытание программ. – М.: Мир, 1985. – 332 с.

26 *Дворянкин А. И,* Основы трансляции: Учебное пособие. – Волгоград: ВолгГТУ, 1999. - 80 с.

27 *Дейкстра Е.* Взаимодействующие последовательные процессы//Языки программирования (под ред. Ф. Женюи). – М.: Мир, 1972.

28. *Дейтел Г.* Введение в операционные системы. В двух томах/Пер, с англ. Л. А. Теплицкого, А. Б. Ходулева, В. С. Штаркмана под ред. В. С. Штаркмана. – М.: Мир, 1987.

29 *Джордейн Р.* Справочник программиста персональных компьютеров типа IBM PC, XT и AT/Пер. с англ. – М.: Финансы и статистика, 1991. – 544 с.

30 *Донован Дж.* Системное программирование. – М.: Мир, 1975. – 540 с.

31 *Дунаев С.* UNIX System V. Release 4.2. Общее руководство. – М.: Диалог-МИФИ, 1995. - 287 с.

32 *Жаков В. И.*, *Коровинский В. В.*, *Фильчаков В. В.* Синтаксический анализ и генерация кода. – СПб.: ГААП, 1993. – 26 с.

33 *Жаков В. И.*, *Корсакова Н. В.*, *Никитин А. В., Фильчаков В. В.* Структуры данных: Учебное пособие. – Л.: ЛИАП. 1989. – 76 с.

34 *Немнюгин* С., *Перколаб Л.* Изучаем Turbo Pascal. – СПб.: Питер, 2000.

35 *Калверт Ч.* Delphi 4. Энциклопедия пользователя. – Киев: ДиаСофт, 1998.

36 *Костер Х.* Основы Windows NT и NTFS /Пер. с англ. – М.: Изд. отдел «Русская редакция» ТОО «Channel Trading Ltd.», 1996. – 440 с.

37 *Кейлингерт П.* Элементы операционных систем. Введение для пользователей/ Пер. с англ. Б. Л. Лисса и С. П. Тресковой. – М.: Мир, 1985. – 295 с.

38 *Кейслер С.* Проектирование операционных систем дня малых ЭВМ. – М.: Мир, 1986. – 680 с.

39 *Керниган Б., Пайк Р.* UNIX – универсальная среда программирования. – М.: Финансы и статистика, 1992. – 420 с.

40 *Клочко В. И.* Теория вычислительных процессов и структур: Учебное пособие. – Краснодар: Изд-во КубГТУ, 1999. – 118 с.

41 *Коваленко И. Н.* QNX: Золушка в семье UNIX.–

http://www.lgg.ru/~nigl/QNX/ doc/Kovalenko_cinderella.html, 1995.

42 *Компаниец Р. И.*, *Маньков Е. В., Филатов Н. Е.* Системное программирование. Основы построения трансляторов/Учебное пособие для высших и средних учебных заведений. – СПб.: КОРОНА принт, 2000. – 256 с.

43 Концептуальное моделирование информационных систем/Под ред. В. В. Фильчакова. – СПб.: СПВУРЭ ПВО, 1998. – 356 с.

44 *Краковяк С.* Основы организации и функционирования ОС ЭВМ. – М.: Мир, 1988. – 480 с.

45 *Льюис* Ф. и др. Теоретические основы построения компиляторов: – М.: Мир, 1979. – 483 с.

46 *МайерсДж.* Надёжность программного обеспечения. – М.: Мир, 1987. – 360 с.

47 *Мельников Б. Ф.* Подклассы класса контекстно-свободных языков. – М.: Изд-во МГУ, 1995. – 174 с.

48 Микропроцессоры 80x86, Pentium: Архитектура, функционирование, программирование, оптимизация кода/В. М. Михальчук, А. А. Ровдо, С. В. Рыжиков. – Мн.: Битрикс, 1994. – 400 с.

49. *Мурата Т.* Сети Петри: Свойства, анализ, приложения (обзор) // ТИИЭР, 1989, № 4. С. 41-85.

50 *Мэдник С*, *Донован Дж.* Операционные системы. – М.: Мир, 1978. – 792 с.

51 *Непомнящий В. А*., *Рякин О. М.* Прикладные методы верификации программ/ Под ред. А. П. Ершова. – М.: Радио и связь, 1988. – 256 с.

52 *Нортон П.* Персональный компьютер фирмы IBM и операционная система MS-DOS /Пер. с англ. – М.: Радио и связь, 1992. – 416 с.

53 *Нортон П*., *Гудмен Дж.* Внутренний мир персональных компьютеров. Изд. 8-е. Избранное от Питера Нортона /Пер. с англ. – К.: Диасофт, 1999. – 584 с.

54 *Обухов И*. QNX: Как надо делать операционные системы / PC Week RE № 7, 1998. С. 58–59.

55 *Озеров В.* Советы по Дельфи (Версия 1.3.1 от 1.07.2000). –

http://www. webmachine.ru/delphi.

56 *Олифер Н. А.*, *Олифер В. Г*. Сетевые операционные системы. – СПб.: Питер, 2001.

57 Операционная система СМ ЭВМ РАФОС: Справочник /Под ред. В. П. Семика. – М.: Финансы и статистика, 1984. – 207 с.

58 Операционные системы – от PC до PS/2 / Ж. Фодор, Д. Бонифас, Ж. Танги. Пер. с франц. – М.: Мир, 1992. – 319 с.

59 *Орловский Г. В.* Введение в архитектуру микропроцессора 80386. – СПб: Сеанс-Пресс Ltd, Инфокон, 1992. – 240 с.

60 ОС QNX: Обзор системы. –

http://www.lgg. ru/~nigl/QNX/doc/about_qnx.html.

61 Red Hat Linux 6.2/Под ред. А. Пасечника – СПб.: Питер, 2000.

62 *Петзолд Ч.* Программирование для Windows 95 /Пер. с англ. – СПб.: BHV, 1997. В 2-х т.

63 *Петруцос Э*., *Хау К.* Visual Basic 6 и VBA.– СПб. и др.: Питер, 2000.– 425 с.

64 *Питерсон Дж.* Теория сетей Петри и моделирование систем/Пер, с англ. – М.: Мир, 1984. – 264 с.

65 *Полетаева И. А.* Методы трансляции: Конспект лекций. – Новосибирск: Изд-во НГТУ, 1998. – Ч. 2. – 51 с.

66 *Пратт Т.*, *Зелковиц М.* Языки программирования: реализация и разработка. – СПб.: Питер, 2001.

67 *Рассел Ч., Кроуфорд Ш.* UNIX и Linux: книга ответов. – СПб.: Питер, 1999. – 297с.

68 *Рейчард К., Фостер -Джонсон Э.* UNIX: справочник. – СПб.: Питер, 2000. – 384 с.

69 Ресурсы Microsoft Windows NT Workstation 4.0 /Пер. с англ. – СПб.: BHV, 1998. – 800 с.

70 *Робачевский А. М.* Операционная система UNIX.– СПб.: BHV, 1997. – 528 с.

71 *Рогаткин Д., Федоров А.* Borland Pascal в среде Windows. – Киев: Диалектика, 1993. – 511 с.

72 *Рудаков П. И., Федотов М. А*. Основы языка Pascal: Учебный курс. – М.: Радио и связь: Горячая линия – Телеком, 2000. – 205 с.

73 *Рудаков П. И., Финогенов К. Г.* Программируем на языке ассемблера IBM PC. Ч. 3. Защищенный режим. – М.: Энтроп, 1996. – 320 с.

74 *Серебряков В. И.* Лекции по конструированию компиляторов. – М.: МГУ, 1997. – 171 с.

75 *Соловьев Г. Н.*, *Никитин В. Д.* Операционные системы ЭВМ: Учебное пособие. – М.: Высшая школа, 1989. – 255 с.

76 *Страуструп Б.* Язык программирования Си++. – М.: Радио и связь, 1991. – 348 с.

77 *Стрыгин В. В.*, *Щарев Л. С.* Основы вычислительной, микропроцессорной техники и программирования. – М.: Высшая школа, 1989. – 478 с.

78 *Студнев* Л. Boot-менеджеры – кто они и откуда?//Byte Россия, 1998, № 4. С. 70–75.

79 *Тревеннор А.* Операционные системы малых ЭВМ /Пер. с англ. А. Г. Васильева. – М.: Финансы и статистика, 1987. – 188 с.

80 *Уинер Р.* Язык Турбо С. – М.: Мир, 1991. – 380 с.

81 *Уокерли Дж.* Архитектура и программирование микро-ЭВМ. – М.: Мир, 1984. – 486 с.

82 *Федоров В. В.* Основы построения трансляторов: Учебное пособие. – Обнинск: ИАТЭ, 1995. – 105 с.

83 *Финогенов К. Г.* Основы языка ассемблера. – М.: Радио и связь, 1999. – 288 с.

84 *Фролов А. В., Фролов Г. В.* Защищённый режим процессоров Intel 80286, 80386, 80486. Практическое руководство по использованию защищённого режима. – М.: Диалог-МИФИ, 1993. – 240 с.

85 *Фролов А. В., Фролов Г. В.* Операционная система OS/2 Warp. –М.: Диалог-МИФИ (Библиотека системного программиста; т. 20), 1995. – 272 с.

86 *Фролов А. В., Фролов Г. В.* Программирование для IBM OS/2 Warp: Ч.1. – М.: Диалог-МИФИ, 1996. – 288 с.

87 *Фролов А. В*., *Фролов Г. В.* Программирование для Windows NT. – М.: Диалог-МИФИ (Библиотека системного программиста; т. 26, 27), 1996.

88 *Хоар Ч.* Взаимодействующие последовательные процессы. – М.: Мир, 1989. – 264 с.

89 *Цикритзис Д., Бернстайн Ф.* Операционные системы /Пер. с англ. В. Л. Ушковой и Н. Б. Фейгельсон. – М.: Мир, 1977. – 336 с.

90 *Чернышев А. В.* Инструментальные средства программирования из состава ОС UNIX и их применение в повседневной практике. – М.: Изд-во МГУП, 1999. – 191 с.

91 *Шапошников И. В.* Интернет-программирование. – СПб.: БХВ – Санкт-Петербург, 2000. – 214 с.

92 *Шоу А*. Логическое проектирование операционных систем /Пер. с англ. В. В. Макарова и В. Д. Никитина. – М.: Мир, 1981. – 360 с.

93 *Юров В.* Assembler: Учебник. – СПб. и др.: Питер, 2000. – 622 с.

94 *Ющенко* С. *В.* ОС QNX – реальное время, реальные возможности /Мир ПК, № 5-6,1995.

95 Windows 2000 для системного администратора. Microsoft Windows 2000: Server и Professional. Русские версии / Под общ. ред. А. Н. Чекмарева и Д. Б. Вишнякова. – СПб.: BHV, 2000. – 1056 с.

96 OS/2 Warp изнутри. В 2-х томах. / М. Минаси, Б. Камарда и др. Пер. с англ. С. Сокорновой. – С-Пб.: Питер, 1996. Т. 1. – 528 с. Т. 2 – 512 с.

97 Understanding Windows NT POSIX Compatibility by Ray Cort Microsoft Corporate Technology Team, Created: May/June 1993.

98 http://www.borland.ru/.

99 http://www.corba.ru/.

100 http://www.gnu.org/.

101 http://java.sun.com/.

102 http://www.linux.zp.ua:8100/philosophy/philosophy.ru.html.

103 http://www.microsoft.com/RUS/lnternet/intranets.html.

104 http://www.microsoft.com/RUS/mssol99/prod/dev/vc.htm.

105 http://www.microsoft.com/RUS/mssol99/prod/dev/vb.htm.

106 http://www.microsoft.com/rus/net/.

107 http://www.perl.org/.

108 http://www.php.net/.

СОДЕРЖАНИЕ