Boxoft Image To PDF Demo. Purchase from www.Boxoft.com to remove the watermark

Современные операционные системы

2-е издание, исправленное

Назаров С.В.
Широков А.И.

Национальный Открытый Университет "ИНТУИТ"
2016

Современные операционные системы

2-е издание, исправленное

Назаров С.В.
Широков А.И.

Национальный Открытый Университет "ИНТУИТ"
2016

УДК 004.451(075)
ББК 14
Н19
Современные операционные системы / Назаров С.В., Широков А.И. - М.: Национальный Открытый Университет "ИНТУИТ", 2016 (Основы информационных технологий)
ISBN 978-5-9963-0416-5

В курсе представлены понятия и положения теории операционных систем. Даны основные определения и классификации, рассмотрены интерфейсы операционных систем, организация вычислительного процесса, вопросы управления памятью и устройствами компьютера, организации файловых систем. Уделено внимание совместимости операционных сред и средствам ее обеспечения, в том числе виртуальным машинам. Изложена история происхождения двух наиболее распространенных представителей этого класса программных систем: семейства UNIX/Linux и компании Microsoft. Рассмотрены стандарты и лицензии на программные продукты.

Курс освещает теоретические и практические вопросы построения современных операционных систем, сред и оболочек как отдельных компьютеров, так и корпоративных информационных систем, в том числе распределенных. Рассмотрены вопросы архитектуры современных ОС, организации мультипрограммных вычислительных процессов, распределения памяти, управления внешними устройствами и др. Уделено внимание перспективным тенденциям построения ОС, в том числе вопросам виртуализации и мобильности операционных систем. Авторы оригинально, в соответствии с современными концепциями теории построения, развития и применения операционных систем, решают вопросы изложения материала пособия.

(c) ООО "ИНТУИТ.РУ", 2010-2016
(c) Назаров С.В., Широков А.И., 2010-2016

Архитектура, назначение и функции операционных систем

Понятие операционной системы. Виртуальные машины. Операционная система, среда и операционная оболочка. Эволюция операционных систем. Назначение, состав и функции ОС. Архитектура операционной системы. Классификация операционных систем. Эффективность и требования, предъявляемые к ОС. Совместимость и множественные прикладные среды. Виртуальные машины как современный подход в реализации множественных прикладных сред. Эффекты виртуализации.

1.1. Понятие операционной системы. Виртуальные машины

Современный компьютер – сложнейшая аппаратно-программная система. Написание программ для компьютера, их отладка и последующее выполнение представляет собой сложную трудоемкую задачу. Основная причина этого – огромная разница между тем, что удобно для людей, и тем, что удобно для компьютеров. Компьютер понимает только свой, машинный язык (назовем его Я0), а для человека наиболее удобен разговорный или хотя бы язык описания алгоритмов – алгоритмический язык. Проблему можно решить двумя способами. Оба способа связаны с разработкой команд, которые были бы более удобны для человека, чем встроенные машинные команды компьютера. Эти новые команды в совокупности формируют некоторый язык, который назовем Я1.

Упомянутые два способа решения проблемы различаются тем, каким образом компьютер будет выполнять программы, написанные на языке Я1. Первый способ – замена каждой команды языка Я1 на эквивалентный набор команд в языке Я0. В этом случае компьютер выполняет новую программу, написанную на языке Я0, вместо программы, написанной на языке Я1. Эта технология называется трансляцией.

Второй способ – написание программы на языке Я0, которая берет программы, написанные на языке Я1, в качестве входных данных, рассматривает каждую команду по очереди и сразу выполняет

эквивалентный набор команд языка Я0. Эта технология не требует составления новой программы на Я0. Она называется интерпретацией, а программа, которая осуществляет интерпретацию, называется интерпретатором.

В подобной ситуации проще представить себе существование гипотетического компьютера или виртуальной машины, для которой машинным языком является язык Я1, чем думать о трансляции и интерпретации. Назовем такую виртуальную машину М1, а виртуальную машину с языком Я0 – М0. Для виртуальных машин можно будет писать программы, как будто они (машины) действительно существуют.

Очевидно, можно пойти дальше – создать еще набор команд, который в большей степени ориентирован на человека и в меньшей степени на компьютер, чем Я1. Этот набор формирует язык Я2 и, соответственно, виртуальную машину М2. Так можно продолжать до тех пор, пока не дойдем до подходящего нам языка уровня n.

Большинство современных компьютеров состоит из двух и более уровней. Уровень 0 – аппаратное обеспечение машины. Электронные схемы этого уровня выполняют программы, написанные на языке уровня 1. Следующий уровень – микроархитектурный уровень.

На этом уровне можно видеть совокупности 8 или 32 (иногда и больше) регистров, которые формируют локальную память и АЛУ (арифметико-логическое устройство). Регистры вместе с АЛУ формируют тракт данных, по которому поступают данные. Основная операция этого тракта заключается в следующем. Выбирается один или два регистра, АЛУ производит над ними какую-то операцию, а результат помещается в один из этих регистров. На некоторых машинах работа тракта контролируется особой программой, которая называется микропрограммой. В других машинах такой контроль выполняется аппаратным обеспечением.

Следующий (второй) уровень составляет уровень архитектуры системы команд. Команды используют регистры и другие возможности аппаратуры. Команды формируют уровень ISA (Instruction Set Architecture), называемый машинным языком. Обычно машинный язык содержит от 50 до 300 команд, служащих преимущественно для

перемещения данных по компьютеру, выполнения арифметических операций и сравнения величин.

Следующий (третий) уровень обычно – гибридный. Большинство команд в его языке есть также и на уровне архитектуры системы команд. У этого уровня есть некоторые дополнительные особенности: набор новых команд, другая организация памяти, способность выполнять две и более программы одновременно и некоторые другие. С течением времени набор таких команд существенно расширился. В нем появились так называемые макросы операционной системы или вызовы супервизора, называемые теперь системными вызовами.

Новые средства, появившиеся на третьем уровне, выполняются интерпретатором, который работает на втором уровне. Этот интерпретатор был когда-то назван операционной системой. Команды третьего уровня, идентичные командам второго уровня, выполняются микропрограммой или аппаратным обеспечением, но не операционной системой. Иными словами, одна часть команд третьего уровня интерпретируется операционной системой, а другая часть – микропрограммой. Вот почему этот уровень операционной системы считается гибридным.

Операционная система была создана для того, чтобы автоматизировать работу оператора и скрыть от пользователя сложности общения с аппаратурой, предоставив ему более удобную систему команд. Нижние три уровня (с нулевого по второй) конструируются не для того, чтобы с ними работал обычный программист. Они изначально предназначены для работы интерпретаторов и трансляторов, поддерживающих более высокие уровни. Эти трансляторы и интерпретаторы составляются системными программистами, которые специализируются на разработке и построении новых виртуальных машин.

Над операционной системой (ОС) расположены остальные системные программы. Здесь находится интерпретатор команд (оболочка), компиляторы, редакторы и т.д. Подобные программы не являются частью ОС (иногда оболочку пользователя считают операционной системой). Под операционной системой обычно понимается то программное обеспечение, которое запускается в режиме ядра или, как еще его называют, режиме супервизора. Она защищена от

вмешательства пользователя с помощью специальных аппаратных средств.

Четвертый уровень представляет собой символическую форму одного из языков низкого уровня (обычно ассемблер). На этом уровне можно писать программы в приемлемой для человека форме. Эти программы сначала транслируются на язык уровня 1, 2 или 3, а затем интерпретируются соответствующей виртуальной или фактически существующей (физической) машиной.

Уровни с пятого и выше предназначены для прикладных программистов, решающих конкретные задачи на языках высокого уровня (C, C++, C#, VBA и др.). Компиляторы и редакторы этих уровней запускаются в пользовательском режиме. На еще более высоких уровнях располагаются прикладные программы пользователей.

Большинство пользователей компьютеров имеют опыт общения с операционной системой, по крайней мере, в той степени, чтобы эффективно выполнять свои текущие задачи. Однако они испытывают затруднения при попытке дать определение операционной системе. В известной степени проблема связана с тем, что операционные системы выполняют две основные, но практически не связанные между собой функции: расширение возможностей компьютера и управление его ресурсами.

С точки зрения пользователя ОС выполняет функции расширенной машины или виртуальной машины, в которой легче программировать и легче работать, чем непосредственно с аппаратным обеспечением, составляющим реальный компьютер. Операционная система не только устраняет необходимость работы непосредственно с дисками и предоставляет простой, ориентированный на работу с файлами интерфейс, но и скрывает множество неприятной работы с прерываниями, счетчиками времени, организацией памяти и другими компонентами низкого уровня.

Однако концепция, рассматривающая операционную систему прежде всего как удобный интерфейс пользователя, – это взгляд сверху вниз. Альтернативный взгляд, снизу вверх, дает представление об операционной системе как о механизме, присутствующем в компьютере для управления всеми компонентами этой сложнейшей системы. В

соответствии с этим подходом работа операционной системы заключается в обеспечении организованного и контролируемого распределения процессоров, памяти, дисков, принтеров, устройств ввода-вывода, датчиков времени и т.п. между различными программами, конкурирующими за право их использовать.

1.2. Операционная система, среда и операционная оболочка

Операционные системы (ОС) в современном их понимании (их назначении и сущности) появились значительно позже первых компьютеров (правда, по всей видимости, и исчезнут в этой сущности в компьютерах будущего). *Почему и когда появились ОС?* Считается[2], что первая цифровая вычислительная машина ENIAC (Electronic Numerical Integrator and Computer) была создана в 1946 году по проекту "Проект PX" Министерства обороны США. На реализацию проекта затратили 500 тыс. долларов. Компьютер содержал 18000 электронных ламп, массу всякой электроники, включал в себя 12 десятиразрядных сумматоров, а для ускорения некоторых арифметических операций имел умножитель и "делитель-извлекатель" квадратного корня. Программирование сводилось к связыванию различных блоков проводами. Конечно, никакого программного обеспечения и тем более операционных систем тогда еще не существовало [10, 13].

Интенсивное создание различных моделей ЭВМ относится к началу 50-х годов прошлого века. В эти годы одни и те же группы людей участвовали и в проектировании, и в создании, и в программировании, и в эксплуатации ЭВМ. Программирование осуществлялось исключительно на машинном языке (а затем на ассемблере), не было никакого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм. Операционные системы еще не появились, и все задачи организации вычислительного процесса решались вручную каждым программистом с примитивного пульта управления ЭВМ.

С появлением полупроводниковых элементов вычислительные возможности компьютеров существенно выросли. Наряду с этим заметно прогрессировали достижения в области автоматизации

программирования и организации вычислительных работ. Появились алгоритмические языки (алгол, фортран, кобол) и системное программное обеспечение (трансляторы, редакторы связи, загрузчики и др.). Выполнение программ усложнилось и включало в себя следующие основные действия:

- загрузка нужного транслятора (установка нужных МЛ и др.);
- запуск транслятора и получение программы в машинных кодах;
- связывание программы с библиотечными подпрограммами;
- загрузка программы в оперативную память;
- запуск программы;
- вывод результатов работы программы на печатающее или другие периферийные устройства.

Для организации эффективной загрузки всех средств компьютера в штаты вычислительных центров ввели должности специально обученных операторов, профессионально выполняющих работу по организации вычислительного процесса для всех пользователей этого центра. Однако, как бы ни был подготовлен оператор, ему тяжело состязаться в производительности с работой устройств компьютера. И поэтому большую часть времени дорогостоящий процессор простаивал, а следовательно, использование компьютеров не было эффективным.

С целью исключения простоев были предприняты попытки разработки специальных программ – мониторов, прообразов первых операционных систем, которые осуществляли автоматический переход от задания к заданию. Считается, что первую операционную систему создала в 1952 году для своих компьютеров IBM-701 исследовательская лаборатория фирмы General Motors [2]. В 1955 году эта фирма и North American Aviation совместно разработали ОС для компьютера IBM-704.

В конце 50-х годов прошлого века ведущие фирмы изготовители поставляли операционные системы со следующими характеристиками:

- пакетная обработка одного потока задач;
- наличие стандартных программ ввода-вывода;
- возможность автоматического перехода от программы к программе;
- средства восстановления после ошибок, обеспечивающие

автоматическую "очистку" компьютера в случае аварийного завершения очередной задачи и позволяющие запускать следующую задачу при минимальном вмешательстве оператора;

- языки управления заданиями, предоставляющие пользователям возможность описывать свои задания и ресурсы, требуемые для их выполнения.

Пакет представляет собой набор (колоду) перфокарт, организованную специальным образом (задание, программа, данные). Для ускорения работы он мог переноситься на магнитную ленту или диск. Это позволяло сократить простои дорогой аппаратуры. Надо сказать, что в настоящее время в связи с прогрессом микроэлектронных технологий и методологий программирования значительно снизилась стоимость аппаратных и программных средств компьютерной техники. Поэтому сейчас основное внимание уделяется тому, чтобы сделать работу пользователей и программистов более эффективной, поскольку затраты труда квалифицированных специалистов сейчас представляют собой гораздо большую долю общей стоимости вычислительных систем, чем аппаратные и программные средства компьютеров.

Расположение операционной системы в иерархической структуре программного и аппаратного обеспечения компьютера можно представить, как показано на рис. 1.1.

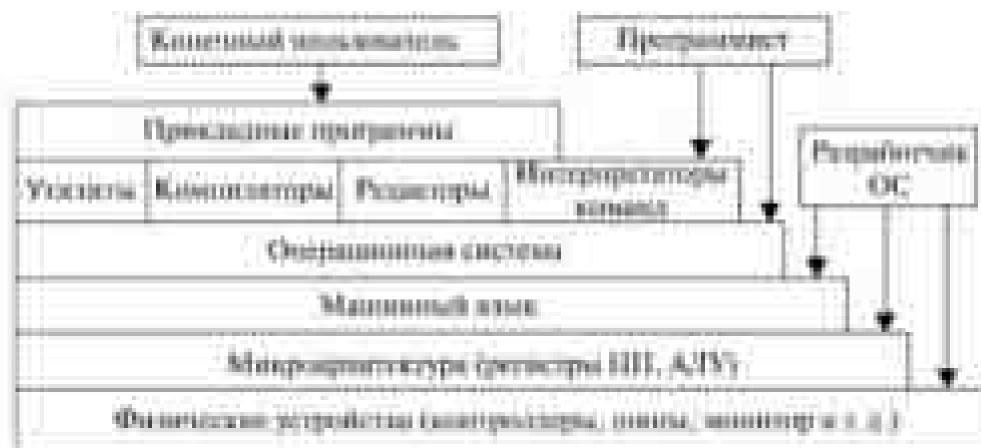


Рис. 1.1. Иерархическая структура программно-аппаратных средств компьютера

Самый нижний уровень содержит различные устройства компьютера, состоящие из микросхем, проводников, источников питания, электронно-лучевых трубок и т.п. Этот уровень можно разделить на подуровни, например контроллеры устройств, а затем сами устройства. Возможно деление и на большее число уровней. Выше расположен микроархитектурный уровень, на котором физические устройства рассматриваются как отдельные функциональные единицы.

На микроархитектурном уровне находятся внутренние регистры центрального процессора (их может быть несколько) и арифметико-логические устройства со средствами управления ими. На этом уровне реализуется выполнение машинных команд. В процессе выполнения команд используются регистры процессора и устройств, а также другие возможности аппаратуры. Команды, видимые для работающего на ассемблере программиста, формируют уровень ISA (Instruction Set Architecture – архитектура системы команд), часто называемый машинным языком.

Операционная система предназначена для того, чтобы скрыть все эти сложности. Конечный пользователь обычно не интересуется деталями устройства аппаратного обеспечения компьютера. Компьютер ему видится как набор приложений. Приложение может быть написано программистом на каком-либо языке программирования. Для упрощения этой работы программист использует набор системных программ, некоторые из которых называются утилитами. С их помощью реализуются часто используемые функции, которые помогают работать с файлами, управлять устройствами ввода-вывода и т.п. Программист применяет эти средства при разработке программ, а приложения во время выполнения обращаются к утилитам для выполнения определенных функций. Наиболее важной из системных программ является операционная система, которая освобождает программиста от необходимости глубокого знания устройства компьютера и предоставляет ему удобный интерфейс для его использования. Операционная система выступает в роли посредника, облегчая программисту, пользователям и программным приложениям доступ к различным службам и возможностям компьютера [10].

Таким образом, операционная система – это набор программ, контролирующих работу прикладных программ и системных

приложений и исполняющих роль интерфейса между пользователями, программистами, прикладными программами, системными приложениями и аппаратным обеспечением компьютера.

Образно можно сказать, что аппаратура компьютера предоставляет "сырую" вычислительную мощность, а задача операционной системы заключается в том, чтобы сделать использование этой вычислительной мощности доступным и по возможности удобным для пользователя. Программист может не знать детали управления конкретными ресурсами (например, диском) компьютера и должен обращаться к операционной системе с соответствующими вызовами, чтобы получить от нее необходимые сервисы и функции. Этот набор сервисов и функций и представляет собой операционную среду, в которой выполняются прикладные программы.

Таким образом, операционная среда – это программная среда, образуемая операционной системой, определяющая интерфейс прикладного программирования (API) как множество системных функций и сервисов (системных вызовов), которые предоставляются прикладным программам. Операционная среда может включать несколько интерфейсов прикладного программирования. Кроме основной операционной среды, называемой естественной (native), могут быть организованы путем эмуляции (моделирования) дополнительные программные среды, позволяющие выполнять приложения, которые рассчитаны на другие операционные системы и даже другие компьютеры.

Еще одно важное понятие, связанное с операционной системой, относится к реализации пользовательских интерфейсов. Как правило, любая операционная система обеспечивает удобную работу пользователя за счет средств пользовательского интерфейса. Эти средства могут быть неотъемлемой частью операционной среды (например, графический интерфейс Windows или текстовый интерфейс командной строки MS DOS), а могут быть реализованы отдельной системной программой – оболочкой операционной системы (например, Norton Commander для MS DOS). В общем случае под оболочкой операционной системы понимается часть операционной среды, определяющая интерфейс пользователя, его реализацию (текстовый, графический и т.п.), командные и сервисные возможности пользователя

по управлению прикладными программами и компьютером.

Перейдем к рассмотрению эволюции операционных систем.

1.3. Эволюция операционных систем

Рассматривая эволюцию ОС, следует иметь в виду, что разница во времени реализации некоторых принципов организации отдельных операционных систем до их общего признания, а также терминологическая неопределенность не позволяют дать точную хронологию развития ОС. Однако сейчас уже достаточно точно можно определить основные вехи на пути эволюции операционных систем.

Существуют также различные подходы к определению поколений ОС. Известно разделение ОС на поколения в соответствии с поколениями вычислительных машин и систем [5, 9, 10, 12]. Такое деление нельзя считать полностью удовлетворительным, так как развитие методов организации ОС в рамках одного поколения ЭВМ, как показал опыт их создания, происходит в достаточно широком диапазоне. Другая точка зрения не связывает поколение ОС с соответствующими поколениями ЭВМ. Так, например, известно определение поколений ОС по уровням входного языка ЭВМ, режимам использования центральных процессоров, формам эксплуатации систем и т.п. [5, 12].

Видимо, наиболее целесообразным следует считать выделение этапов развития ОС в рамках отдельных поколений ЭВМ и ВС.

Первым этапом развития системного программного обеспечения можно считать использование библиотечных программ, стандартных и служебных подпрограмм и макрокоманд. Концепция библиотек подпрограмм является наиболее ранней и восходит к 1949 году [4, 17]. С появлением библиотек получили развитие автоматические средства их сопровождения – программы-загрузчики и редакторы связей. Эти средства применялись в ЭВМ первого поколения, когда операционных систем как таковых еще не существовало.

Стремление устранить несоответствие между производительностью процессоров и скоростью работы электромеханических устройств ввода-вывода, с одной стороны, и использование достаточно

быстродействующих накопителей на магнитных лентах и барабанах (НМЛ и НМБ), а затем на магнитных дисках (НМД), с другой стороны, привело к необходимости решения задач буферизации и блокирования-деблокирования данных. Возникли специальные программы методов доступа, которые вносились в объекты модулей редактором связей (впоследствии стали использоваться принципы полибуферизации). Для поддержания работоспособности и облегчения процесса эксплуатации машин создавались диагностические программы. Таким образом было создано базовое системное программное обеспечение.

С улучшением характеристик ЭВМ и ростом их производительности стало ясно, что существующего базового программного обеспечения (ПО) недостаточно. Появились операционные системы ранней пакетной обработки – мониторы. В рамках системы пакетной обработки во время выполнения любой работы в пакете (трансляция, сборка, выполнение готовой программы) никакая часть системного ПО не находилась в оперативной памяти, так как вся память предоставлялась текущей работе. Затем появились мониторные системы, в которых оперативная память делилась на три области: фиксированная область мониторной системы, область пользователя и область общей памяти (для хранения данных, которыми могут обмениваться объектные модули).

Началось интенсивное развитие методов управления данными, возникла такая важная функция ОС, как реализация ввода-вывода без участия центрального процесса – так называемый спулинг (от англ. SPOOL – Simultaneous Peripheral Operation on Line).

Появление новых аппаратных разработок (1959-1963 гг.) – систем прерываний, таймеров, каналов – стимулировало дальнейшее развитие ОС [4, 5, 9]. Возникли исполнительные системы, которые представляли собой набор программ для распределения ресурсов ЭВМ, связей с оператором, управления вычислительным процессом и управления вводом-выводом. Такие исполнительные системы позволили реализовать довольно эффективную по тому времени форму эксплуатации вычислительной системы – однопрограммную пакетную обработку. Эти системы давали пользователю такие средства, как контрольные точки, логические таймеры, возможность построения программ оверлейной структуры, обнаружения нарушений

программам ограничений, принятых в системе, управление файлами, сбор учетной информации и др.

Однако однопрограммная пакетная обработка с ростом производительности ЭВМ не могла обеспечить экономически приемлемый уровень эксплуатации машин. Решением стало мультипрограммирование – способ организации вычислительного процесса, при котором в памяти компьютера находится несколько программ, попеременно выполняющихся одним процессором, причем для начала или продолжения счета по одной программе не требовалось завершения других. В мультипрограммной среде проблемы распределения ресурсов и защиты стали более острыми и трудноразрешимыми.

Теория построения операционных систем в этот период обогатилась рядом плодотворных идей. Появились различные формы мультипрограммных режимов работы, в том числе разделение времени – режим, обеспечивающий работу многотерминальной системы. Была создана и развита концепция виртуальной памяти, а затем и виртуальных машин. Режим разделения времени позволил пользователю интерактивно взаимодействовать со своими программами, как это было до появления систем пакетной обработки.

Одной из первых ОС, использующих эти новейшие решения, была операционная система MCP (главная управляющая программа), созданная фирмой Burroughs для своих компьютеров B5000 в 1963 году. В этой ОС были реализованы многие концепции и идеи, ставшие впоследствии стандартными для многих операционных систем:

- мультипрограммирование;
- мультипроцессорная обработка;
- виртуальная память;
- возможность отладки программ на исходном языке;
- написание операционной системы на языке высокого уровня.

Известной системой разделения времени того периода стала система CTSS (Compatible Time Sharing System) – совместимая система разделения времени, разработанная в Массачусетском технологическом институте (1963 год) для компьютера IBM-7094 [27]. Эта система была

использована для разработки в этом же институте совместно с Bell Labs и General Electric системы разделения времени следующего поколения MULTICS (Multiplexed Information And Computing Service). Примечательно, что эта ОС была написана в основном на языке высокого уровня EPL (первая версия языка PL/1 фирмы IBM).

Одним из важнейших событий в истории операционных систем считается появление в 1964 году семейства компьютеров под названием System/360 фирмы IBM, а позже – System/370 [1]. Это было первой в мире реализацией концепции семейства программно и информационно совместимых компьютеров, ставшей впоследствии стандартной для всех фирм компьютерной отрасли.

Нужно отметить, что основной формой использования ЭВМ, как в системах разделения времени, так и в системах пакетной обработки, стал многотерминальный режим. При этом не только оператор, но и все пользователи получали возможность формулировать свои задания и управлять их выполнением со своего терминала. Поскольку терминальные комплексы скоро стало возможным размещать на значительных расстояниях от компьютера (благодаря модемным телефонным соединениям), появились системы удаленного ввода заданий и телеобработки данных. В ОС добавились модули, реализующие протоколы связи [1], [2].

К этому времени произошло существенное изменение в распределении функций между аппаратными и программными средствами компьютера. Операционная система становится "неотъемлемой частью ЭВМ", как бы продолжением аппаратуры. В процессорах появился привилегированный (Супервизор в OS/360) и пользовательский (Задача в OS/360) режимы работы, мощная система прерываний, защита памяти, специальные регистры для быстрого переключения программ, средства поддержки виртуальной памяти и др.

В начале 70-х годов появились первые сетевые ОС, которые позволили не только рассредоточить пользователей, как в системах телеобработки данных, но и организовать распределенное хранение и обработку данных между компьютерами, соединенных электрическими связями. Известен проект ARPANET МО США. В 1974 году IBM объявила о создании собственной сетевой архитектуры SNA для своих

мэйнфреймов, обеспечивающий взаимодействие типа "терминал-терминал", "терминал-компьютер", "компьютер-компьютер". В Европе активно разрабатывались технологии построения сетей с коммутацией пакетов на основе протоколов X.25.

К середине 70-х годов наряду с мэйнфреймами широкое распространение получили мини-компьютеры (PDP-11, Nova, HP). Архитектура мини-компьютеров была значительно проще, многие функции мультипрограммных ОС мэйнфреймов были усечены. Операционные системы мини-ЭВМ стали делать специализированными (RSX-11M – разделение времени, RT-11 – ОС реального времени) и не всегда многопользовательскими.

Важной вехой в истории мини-компьютеров и вообще в истории операционных систем явилось создание ОС UNIX. Написал эту систему Кен Томпсон (Ken Thompson), один из специалистов по компьютерам в BELL Labs, работавший над проектом MULTICS. Собственно, его UNIX – это усеченная однопользовательская версия системы MULTICS. Первоначальное название этой системы – UNICS (UNiplexed Information and Computing Service – примитивная информационная и компьютерная служба). Так в шутку была названа эта система, поскольку MULTICS (MULTiplexed Information and Computing Service) – мультиплексная информационная и компьютерная служба. С середины 70-х годов началось массовое использование ОС UNIX, написанной на 90% на языке C. Широкое распространение C-компиляторов сделало UNIX уникальной переносимой ОС, а поскольку она поставлялась вместе с исходными кодами, она стала первой открытой операционной системой. Гибкость, элегантность, мощные функциональные возможности и открытость позволили ей занять прочные позиции во всех классах компьютеров – от персональных до супер-ЭВМ.

Доступность мини-компьютеров послужила стимулом для создания локальных сетей. В простейших ЛВС компьютеры соединялись через последовательные порты. Первое сетевое приложение для ОС UNIX – программа UUCP (Unix to Unix Copy Program) – появилось в 1976 году.

Дальнейшее развитие сетевых систем со стеком протоколов TCP/IP: в 1983 году он был принят МО США в качестве стандарта и использован в сети ARPANET. В этом же году ARPANET разделилась на MILNET (для

военного ведомства США) и новую ARPANET, которую стали называть Internet.

Все восьмидесятые годы характерны появлением все более совершенных версий UNIX: SunOS, HP-UX, Irix, AIX и др. Для решения проблемы их совместимости были приняты стандарты POSIX и XPG, определяющие интерфейсы этих систем для приложений.

Еще одним знаменательным событием для истории операционных систем было появление в начале 80-х годов персональных компьютеров. Они послужили мощным толчком для распространения локальных сетей, в результате поддержка сетевых функций стала для ОС ПК необходимым условием. Однако и дружественный интерфейс, и сетевые функции появились у ОС ПК не сразу [13].

Наиболее популярной версией ОС раннего этапа развития персональных компьютеров была MS-DOS компании Microsoft – однопрограммная, однопользовательская ОС с интерфейсом командной строки. Многие функции, обеспечивающие удобство работы пользователю, в этой ОС предоставлялись дополнительными программами – оболочкой Norton Commander, PC Tools и др. Наибольшее влияние на развитие программного обеспечения ПК оказала операционная среда Windows, первая версия которой появилась в 1985 году. Сетевые функции также реализовывались с помощью сетевых оболочек и появились в MS-DOS версии 3.1. В это же время появились сетевые продукты Microsoft – MS-NET, а позже – LAN Manager, Windows for Workgroup, а затем и Windows NT.

Другим путем пошла компания Novell: ее продукт NetWare – операционная система со встроенными сетевыми функциями. ОС NetWare распространялась как операционная система для центрального сервера локальной сети и за счет специализации функций файл-сервера обеспечивала высокую скорость удаленного доступа к файлам и повышенную безопасность данных. Однако эта ОС имела специфический программный интерфейс (API), что затрудняло разработку приложений.

В 1987 году появилась первая многозадачная ОС для ПК – OS/2, разработанная Microsoft совместно с IBM. Это была хорошо продуманная система с виртуальной памятью, графическим

интерфейсом и возможностью выполнять DOS-приложения. Для нее были созданы и получили распространение сетевые оболочки LAN Manager (Microsoft) и LAN Server (IBM). Эти оболочки уступали по производительности файловому серверу NetWare и потребляли больше аппаратных ресурсов, но имели важные достоинства. Они позволяли выполнять на сервере любые программы, разработанные для OS/2, MS-DOS и Windows, кроме того, можно было использовать компьютер, на котором они работали, в качестве рабочей станции. Неудачная рыночная судьба OS/2 не позволила системам LAN-Manager и LAN-Server захватить заметную долю рынка, но принципы работы этих сетевых систем во многом нашли свое воплощение в ОС 90-х годов – MS Windows NT.

В 80-е годы были приняты основные стандарты на коммуникационные технологии для локальных сетей: в 1980 г. – Ethernet, в 1985 г. – Token Ring, в конце 80-х – FDDI (Fiber Distributed Data Interface), распределенный интерфейс передачи данных по волоконно-оптическим каналам, двойное кольцо с маркером. Это позволило обеспечить совместимость сетевых ОС на нижних уровнях, а также стандартизировать операционные системы с драйверами сетевых адаптеров.

Для ПК применялись не только специально разработанные для них ОС (MS-Dos, NetWare, OS/2), но и адаптировались уже существующие ОС, в частности UNIX. Наиболее известной системой этого типа была версия UNIX компании Santa Cruz Operation (SCO UNIX).

В 90-е годы практически все операционные системы, занимающие заметное место на рынке, стали сетевыми. Сетевые функции встраиваются в ядро ОС, являясь ее неотъемлемой частью. В ОС используются средства мультиплексирования нескольких стеков протоколов, за счет которого компьютеры могут поддерживать одновременную работу с разнородными серверами и клиентами. Появились специализированные ОС, например, сетевая ОС IOS компании Cisco System, работающая в маршрутизаторах. Во второй половине 90-х годов все производители ОС усилили поддержку средств работы с интерфейсами. Кроме стека протоколов TCP/IP в комплект поставки начали включать утилиты, реализующие популярные сервисы Интернета: telnet, ftp, DNS, Web и др.

Особое внимание уделялось в последнем десятилетии и уделяется в настоящее время корпоративным сетевым операционным системам. Это одна из наиболее важных задач в обозримом будущем. Корпоративные ОС должны хорошо и устойчиво работать в крупных сетях, которые характерны для крупных организаций (предприятий, банков и т.п.), имеющих отделения во многих городах и, возможно, в разных странах. Корпоративная ОС должна без проблем взаимодействовать с ОС разного типа и работать на различных аппаратных платформах. Сейчас определились лидеры в классе корпоративных ОС – это MS Windows2000/2003/XP Professional x64 Edition/7 Enterprise, UNIX и Linux-системы, а также Novell NetWare 6.5.

1.4. Назначение состав и функции ОС

В настоящее время существует большое количество различных типов операционных систем, отличающихся областями применения, аппаратными платформами, способами реализации и др. Назначение операционных систем можно разделить на четыре основные составляющие [5, 10, 13].

1. Организация (обеспечение) удобного интерфейса между приложениями и пользователями, с одной стороны, и аппаратурой компьютера – с другой. Вместо реальной аппаратуры компьютера ОС представляет пользователю расширенную виртуальную машину, с которой удобнее работать и которую легче программировать. Вот список основных сервисов, предоставляемых типичными операционными системами.

1. Разработка программ: ОС представляет программисту разнообразные инструменты разработки приложений: редакторы, отладчики и т.п. Ему не обязательно знать, как функционируют различные электронные и электромеханические узлы и устройства компьютера. Часто пользователь не знает даже системы команд процессора, поскольку он может обойтись мощными высокоуровневыми функциями, которые представляет ОС.
2. Исполнение программ. Для запуска программы нужно выполнить ряд действий: загрузить в основную память программу и данные,

инициализировать устройства ввода-вывода и файлы, подготовить другие ресурсы. ОС выполняет всю эту рутинную работу вместо пользователя.

3. Доступ к устройствам ввода-вывода. Для управления каждым устройством используется свой набор команд. ОС предоставляет пользователю единообразный интерфейс, который скрывает все эти детали и обеспечивает программисту доступ к устройствам ввода-вывода с помощью простых команд чтения и записи. Если бы программист работал непосредственно с аппаратурой компьютера, то для организации, например, чтения блока данных с диска ему пришлось бы использовать более десятка команд с указанием множества параметров. После завершения обмена программист должен был бы предусмотреть еще более сложный анализ результата выполненной операции.
4. Контролируемый доступ к файлам. При работе с файлами управление со стороны ОС предполагает не только глубокий учет природы устройства ввода-вывода, но и знание структур данных, записанных в файлах. Многопользовательские ОС, кроме того, обеспечивают механизм защиты при обращении к файлам.
5. Системный доступ. ОС управляет доступом к совместно используемой или общедоступной вычислительной системе в целом, а также к отдельным системным ресурсам. Она обеспечивает защиту ресурсов и данных от несанкционированного использования и разрешает конфликтные ситуации.
6. Обнаружение ошибок и их обработка. При работе компьютерной системы могут происходить разнообразные сбои за счет внутренних и внешних ошибок в аппаратном обеспечении, различного рода программных ошибок (переполнение, попытка обращения к ячейке памяти, доступ к которой запрещен и др.). В каждом случае ОС выполняет действия, минимизирующие влияние ошибки на работу приложения (от простого сообщения об ошибке до аварийной остановки программы).
7. Учет использования ресурсов. Хорошая ОС имеет средства учета использования различных ресурсов и отображения параметров производительности вычислительной системы. Эта информация важна для настройки (оптимизации) вычислительной системы с целью повышения ее производительности.

В результате реальная машина, способная выполнять только небольшой набор элементарных действий (машинных команд), с помощью операционной системы превращается в виртуальную машину, выполняющую широкий набор гораздо более мощных функций. Виртуальная машина тоже управляется командами, но уже командами более высокого уровня, например: удалить файл с определенным именем, запустить на выполнение прикладную программу, повысить приоритет задачи, вывести текст файла на печать и т.д. Таким образом, назначение ОС состоит в предоставлении пользователю (программисту) некоторой расширенной виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальный компьютер, систему или сеть.

2. Организация эффективного использования ресурсов компьютера. ОС не только представляет пользователям и программистам удобный интерфейс к аппаратным средствам компьютера, но и является своеобразным диспетчером ресурсов компьютера. К числу основных ресурсов современных вычислительных систем относятся процессоры, основная память, таймеры, наборы данных, диски, накопители на магнитных лентах (МЛ), внешние накопители памяти(CD/DVD/Blu-Ray/USB), принтеры, сетевые устройства и др. Эти ресурсы распределяются операционной системой между выполняемыми программами. В отличие от программы, которая является статическим объектом, выполняемая программа – это динамический объект, он называется процессом и является базовым понятием современных ОС.

Управление ресурсами вычислительной системы с целью наиболее эффективного их использования является вторым назначением операционной системы. Критерии эффективности, в соответствии с которыми ОС организует управление ресурсами компьютера, могут быть различными. Например, в одних системах важен такой критерий, как пропускная способность вычислительной системы, в других – время ее реакции. Зачастую ОС должны удовлетворять нескольким, противоречащим друг другу критериям, что доставляет разработчикам серьезные трудности.

Управление ресурсами включает решение ряда общих, не зависящих от типа ресурса задач:

1. планирование ресурса – определение, какому процессу, когда и в каком качестве (если ресурс может выделяться частями) следует выделить данный ресурс;
2. удовлетворение запросов на ресурсы – выделение ресурса процессам;
3. отслеживание состояния и учет использования ресурса – поддержание оперативной информации о занятости ресурса и распределенной его доли;
4. разрешение конфликтов между процессами, претендующими на один и тот же ресурс.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, особенности которых, в конечном счете, определяют облик ОС в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Таким образом, управление ресурсами составляет важное назначение ОС. В отличие от функций расширенной виртуальной машины большинство функций управления ресурсами выполняются операционной системой автоматически и прикладному программисту недоступны.

3. Облегчение процессов эксплуатации аппаратных и программных средств вычислительной системы. Ряд операционных систем имеет в своем составе наборы служебных программ, обеспечивающие резервное копирование, архивацию данных, проверку, очистку и дефрагментацию дисковых устройств и др.

Кроме того, современные ОС имеют достаточно большой набор средств и способов диагностики и восстановления работоспособности системы. Сюда относятся:

- диагностические программы для выявления ошибок в конфигурации ОС;
- средства восстановления последней работоспособной конфигурации;
- средства восстановления поврежденных и пропавших системных файлов и др.

Следует отметить еще одно назначение ОС.

4. Возможность развития. Современные ОС организуются таким образом, что допускают эффективную разработку, тестирование и внедрение новых системных функций, не прерывая процесса нормального функционирования вычислительной системы. Большинство операционных систем постоянно развиваются (наглядный пример Windows). Происходит это в силу следующих причин.

1. Обновление и возникновение новых видов аппаратного обеспечения. Например, ранние версии ОС UNIX и OS/2 не использовали механизмы страничной организации памяти (что это такое, мы рассмотрим позже), потому что они работали на машинах, не обеспеченных соответствующими аппаратными средствами.
2. Новые сервисы. Для удовлетворения пользователей или нужд системных администраторов ОС должны постоянно предоставлять новые возможности. Например, может потребоваться добавить новые инструменты для контроля или оценки производительности, новые средства ввода-вывода данных (речевой ввод). Другой пример – поддержка новых приложений, использующих окна на экране дисплея.
3. Исправления. В каждой ОС есть ошибки. Время от времени они обнаруживаются и исправляются. Отсюда постоянные появления новых версий и редакций ОС. Необходимость регулярных изменений накладывает определенные требования на организацию операционных систем. Очевидно, что эти системы (как, впрочем, и другие сложные программные системы) должны иметь модульную структуру с четко определенными межмодульными связями (интерфейсами). Важную роль играет хорошая и полная документированность системы.

Перейдем к рассмотрению состава компонентов и функций ОС. Современные операционные системы содержат сотни и тысячи модулей (например, W2000 содержит 29 млн строк исходного кода на языке С). Функции ОС обычно группируются либо в соответствии с типами локальных ресурсов, которыми управляет ОС, либо в соответствии со специфическими задачами, применимыми ко всем ресурсам. Совокупности модулей, выполняющих такие группы функций, образуют подсистемы операционной системы.

Наиболее важными подсистемами управления ресурсами являются подсистемы управления процессами, памятью, файлами и внешними устройствами, а подсистемами, общими для всех ресурсов, являются подсистемы пользовательского интерфейса, защиты данных и администрирования.

Управление процессами. Подсистема управления процессами непосредственно влияет на функционирование вычислительной системы. Для каждой выполняемой программы ОС организует один или более процессов. Каждый такой процесс представляется в ОС информационной структурой (таблицей, дескриптором, контекстом процессора), содержащей данные о потребностях процесса в ресурсах, а также о фактически выделенных ему ресурсах (область оперативной памяти, количество процессорного времени, файлы, устройства ввода-вывода и др.). Кроме того, в этой информационной структуре хранятся данные, характеризующие историю пребывания процесса в системе: текущее состояние (активное или заблокированное), приоритет, состояние регистров, программного счетчика и др.

В современных мультипрограммных ОС может существовать одновременно несколько процессов, порожденных по инициативе пользователей и их приложений, а также инициированных ОС для выполнения своих функций (системные процессы). Поскольку процессы могут одновременно претендовать на одни и те же ресурсы, подсистема управления процессами планирует очередность выполнения процессов, обеспечивает их необходимыми ресурсами, обеспечивает взаимодействие и синхронизацию процессов.

Управление памятью. Подсистема управления памятью производит распределение физической памяти между всеми существующими в системе процессами, загрузку и удаление программных кодов и данных процессов в отведенные им области памяти, настройку адресно-зависимых частей кодов процесса на физические адреса выделенной области, а также защиту областей памяти каждого процесса. Стратегия управления памятью складывается из стратегий выборки, размещения и замещения блока программы или данных в основной памяти. Соответственно используются различные алгоритмы, определяющие, когда загрузить очередной блок в память (по запросу или с упреждением), в какое место памяти его поместить и какой блок

программы или данных удалить из основной памяти, чтобы освободить место для размещения новых блоков.

Одним из наиболее популярных способов управления памятью в современных ОС является виртуальная память. Реализация механизма виртуальной памяти позволяет программисту считать, что в его распоряжении имеется однородная оперативная память, объем которой ограничивается только возможностями адресации, предоставляемыми системой программирования.

Важная функция управления памятью – защита памяти. Нарушения защиты памяти связаны с обращениями процессов к участкам памяти, выделенной другим процессам прикладных программ или программ самой ОС. Средства защиты памяти должны пресекать такие попытки доступа путем аварийного завершения программы-нарушителя.

Управление файлами. Функции управления файлами сосредоточены в файловой системе ОС. Операционная система виртуализирует отдельный набор данных, хранящихся на внешнем накопителе, в виде файла – простой неструктурированной последовательности байтов, имеющих символьное имя. Для удобства работы с данными файлы группируются в каталоги, которые, в свою очередь, образуют группы – каталоги более высокого уровня. Файловая система преобразует символьные имена файлов, с которыми работает пользователь или программист, в физические адреса данных на дисках, организует совместный доступ к файлам, защищает их от несанкционированного доступа.

Управление внешними устройствами. Функции управления внешними устройствами возлагаются на подсистему управления внешними устройствами, называемую также подсистемой ввода-вывода. Она является интерфейсом между ядром компьютера и всеми подключенными к нему устройствами. Спектр этих устройств очень обширен (принтеры, сканеры, мониторы, модемы, манипуляторы, сетевые адаптеры, АЦП разного рода и др.), сотни моделей этих устройств отличаются набором и последовательностью команд, используемых для обмена информацией с процессором и другими деталями.

Программа, управляющая конкретной моделью внешнего устройства и

учитывающие все эти особенности, называются драйверами. Наличие большого количества подходящих драйверов во многом определяет успех ОС на рынке. Созданием драйверов занимаются как разработчики ОС, так и компании, выпускающие внешние устройства. ОС должна поддерживать четко определенный интерфейс между драйверами и остальными частями ОС. Тогда разработчики компаний-производителей устройств ввода-вывода могут поставлять вместе со своими устройствами драйверы для конкретной операционной системы.

Защита данных и администрирование. Безопасность данных вычислительной системы обеспечивается средствами отказоустойчивости ОС, направленными на защиту от сбоев и отказов аппаратуры и ошибок программного обеспечения, а также средствами защиты от несанкционированного доступа. Для каждого пользователя системы обязательна процедура логического входа, в процессе которой ОС убеждается, что в систему входит пользователь, разрешенный административной службой. Администратор вычислительной системы определяет и ограничивает возможности пользователей в выполнении тех или иных действий, т.е. определяет их права по обращению и использованию ресурсов системы.

Важным средством защиты являются функции аудита ОС, заключающиеся в фиксации всех событий, от которых зависит безопасность системы. Поддержка отказоустойчивости вычислительной системы реализуется на основе резервирования (дисковые RAID-массивы, резервные принтеры и другие устройства, иногда резервирование центральных процессоров, в ранних ОС – дуальные и дуплексные системы, системы с мажоритарным органом и др.). Вообще обеспечение отказоустойчивости системы – одна из важнейших обязанностей системного администратора, который для этого использует ряд специальных средств и инструментов [7, 10, 13].

Интерфейс прикладного программирования. Прикладные программисты используют в своих приложениях обращения к операционной системе, когда для выполнения тех или иных действий им требуется особый статус, которым обладает только ОС. Возможности операционной системы доступны программисту в виде набора функций, который называется интерфейсом прикладного

программирования (Application Programming Interface, API). Приложения обращаются к функциям API с помощью системных вызовов. Способ, которым приложение получает услуги операционной системы, очень похож на вызов подпрограмм.

Способ реализации системных вызовов зависит от структурной организации ОС, особенностей аппаратной платформы и языка программирования.

В ОС UNIX системные вызовы почти идентичны библиотечным процедурам. Ситуация в Windows иная (более подробно это рассмотрим далее).

Пользовательский интерфейс. ОС обеспечивает удобный интерфейс не только для прикладных программ, но и для пользователя (программиста, администратора). В ранних ОС интерфейс сводился к языку управления заданиями и не требовал терминала. Команды языка управления заданиями набивались на перфокарты, а результаты выполнения задания выводились на печатающее устройство.

Современные ОС поддерживают развитые функции пользовательского интерфейса для интерактивной работы за терминалами двух типов: алфавитно-цифрового и графического. При работе за алфавитно-цифровым терминалом пользователь имеет в своем распоряжении систему команд, развитость которой отражает функциональные возможности данной ОС. Обычно командный язык ОС позволяет запускать и останавливать приложения, выполнять различные операции с каталогами и файлами, получать информацию о состоянии ОС, администрировать систему. Команды могут вводиться не только в интерактивном режиме с терминала, но и считываться из так называемого командного файла, содержащего некоторую последовательность команд.

Программный модуль ОС, ответственный за чтение отдельных команд или же последовательности команд из командного файла, иногда называют командным интерпретатором (в MS-DOS – командным процессором).

Вычислительные системы, управляемые из командной строки, например UNIX-системы, имеют командный интерпретатор,

называемый оболочкой (Shell). Она, собственно, не входит в состав ОС, но пользуется многими функциями операционной системы. Когда какой-либо пользователь входит в систему, запускается оболочка. Стандартным терминалом для нее является монитор с клавиатурой. Оболочка начинает работу с печати приглашения (prompt) – знака доллара (или иного знака), говорящего пользователю, что оболочка ожидает ввода команды (аналогично управляется MS-DOS). Если теперь пользователь напечатает какую-либо команду, оболочка создает системный вызов и ОС выполнит эту команду. После завершения оболочка опять печатает приглашение и пытается прочесть следующую входную строку.

Ввод команд может быть упрощен, если операционная система поддерживает графический пользовательский интерфейс. В этом случае пользователь выбирает на экране нужный пункт меню или графический символ (так это происходит, например, в ОС Windows).

1.5. Архитектура операционной системы

Под архитектурой операционной системы понимают структурную и функциональную организацию ОС на основе некоторой совокупности программных модулей. В состав ОС входят исполняемые и объектные модули стандартных для данной ОС форматов, программные модули специального формата (например, загрузчик ОС, драйверы ввода-вывода), конфигурационные файлы, файлы документации, модули справочной системы и т.д.

На архитектуру ранних операционных систем обращалось мало внимания: во-первых, ни у кого не было опыта в разработке больших программных систем, а во-вторых, проблема взаимозависимости и взаимодействия модулей недооценивалась. В подобных монолитных ОС почти все процедуры могли вызывать одна другую. Такое отсутствие структуры было несовместимо с расширением операционных систем. Первая версия ОС OS/360 была создана коллективом из 5000 человек за 5 лет и содержала более 1 млн строк кода. Разработанная несколько позже операционная система Mastics содержала в 1975 году уже 20 млн строк [17]. Стало ясно, что разработка таких систем должна вестись на основе модульного программирования.

Большинство современных ОС представляют собой хорошо структурированные модульные системы, способные к развитию, расширению и переносу на новые платформы. Какой-либо единой унифицированной архитектуры ОС не существует, но известны универсальные подходы к структурированию ОС. Принципиально важными универсальными подходами к разработке архитектуры ОС являются [5, 10, 13, 17]:

- модульная организация;
- функциональная избыточность;
- функциональная избирательность;
- параметрическая универсальность;
- концепция многоуровневой иерархической вычислительной системы, по которой ОС представляется многослойной структурой;
- разделение модулей на две группы по функциям: ядро – модули, выполняющие основные функции ОС, и модули, выполняющие вспомогательные функции ОС;
- разделение модулей ОС на две группы по размещению в памяти вычислительной системы: резидентные, постоянно находящиеся в оперативной памяти, и транзитные, загружаемые в оперативную память только на время выполнения своих функций;
- реализация двух режимов работы вычислительной системы: привилегированного режима (режима ядра – Kernel mode), или режима супервизора (supervisor mode), и пользовательского режима (user mode), или режима задачи (task mode);
- ограничение функций ядра (а следовательно, и количества модулей ядра) до минимального количества необходимых самых важных функций.

Первые ОС разрабатывались как монолитные системы без четко выраженной структуры (рис. 1.2).

Для построения монолитной системы необходимо скомпилировать все отдельные процедуры, а затем связать их вместе в единый объектный файл с помощью компоновщика (примерами могут служить ранние версии ядра UNIX или Novell NetWare). Каждая процедура видит любую другую процедуру (в отличие от структуры, содержащей модули, в которой большая часть информации является локальной для модуля, и

процедуры модуля можно вызвать только через специально определенные точки входа).

Однако даже такие монолитные системы могут быть немного структурированными. При обращении к системным вызовам, поддерживаемым ОС, параметры помещаются в строго определенные места, такие как регистры или стек, а затем выполняется специальная команда прерывания, известная как вызов ядра или вызов супервизора. Эта команда переключает машину из режима пользователя в режим ядра, называемый также режимом супервизора, и передает управление ОС. Затем ОС проверяет параметры вызова, для того чтобы определить, какой системный вызов должен быть выполнен. После этого ОС индексирует таблицу, содержащую ссылки на процедуры, и вызывает соответствующую процедуру.

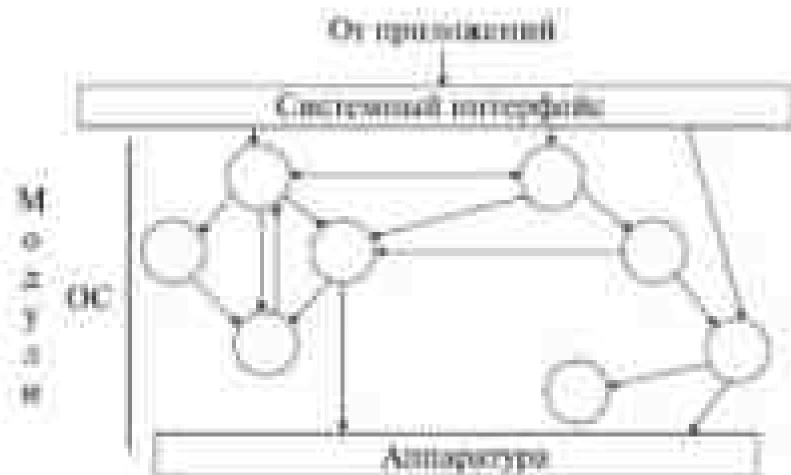


Рис. 1.2. Монолитная архитектура

Такая организация ОС предполагает следующую структуру [12]:

- главная программа, которая вызывает требуемые сервисные процедуры;
- набор сервисных процедур, реализующих системные вызовы;
- набор утилит, обслуживающих сервисные процедуры.

В этой модели для каждого системного вызова имеется одна сервисная

процедуры. Утилиты выполняют функции, которые нужны нескольким сервисным процедурам. Это деление процедур на три слоя показано на рис. 1.3.

Классической считается архитектура ОС, основанная на концепции иерархической многоуровневой машины, привилегированном ядре и пользовательском режиме работы транзитных модулей. Модули ядра выполняют базовые функции ОС: управление процессами, памятью, устройствами ввода-вывода и т.п. Ядро составляет сердцевину ОС, без которой она является полностью неработоспособной и не может выполнить ни одну из своих функций. В ядре решаются внутрисистемные задачи организации вычислительного процесса, недоступные для приложения.

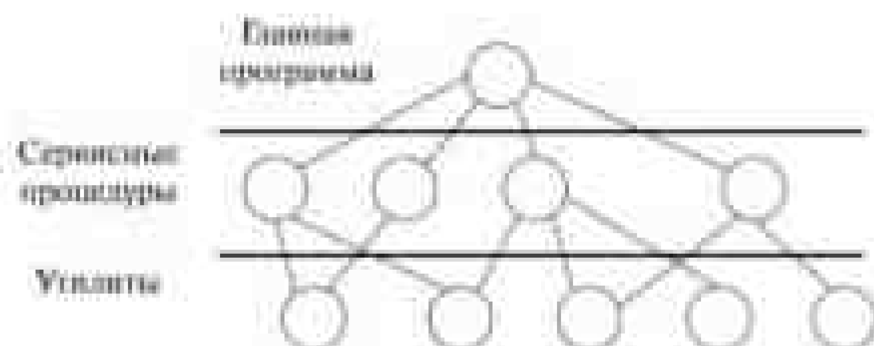


Рис. 1.3. Структурированная архитектура

Особый класс функций ядра служит для поддержки приложений, создавая для них так называемую прикладную программную среду. Приложения могут обращаться к ядру с запросами – системными вызовами – для выполнения тех или иных действий, например, открытие и чтение файла, получение системного времени, вывода информации на дисплей и т.д. Функции ядра, которые могут вызываться приложениями, образуют интерфейс прикладного программирования – API (Application Programming Interface).

Для обеспечения высокой скорости работы ОС модули ядра (по крайней мере, большая их часть) являются резидентными и работают в привилегированном режиме (Kernel mode). Этот режим, во-первых, должен обезопасить работу самой ОС от вмешательства приложений, и,

во-вторых, должен обеспечить возможность работы модулей ядра с полным набором машинных инструкций, позволяющих собственно ядру выполнять управление ресурсами компьютера, в частности, переключение процессора с задачи на задачу, управлением устройствами ввода–вывода, распределением и защитой памяти и др.

Остальные модули ОС выполняют не столь важные функции, как ядро, и являются транзитными. Например, это могут быть программы архивирования данных, дефрагментации диска, сжатия дисков, очистки дисков и т.п.

Вспомогательные модули обычно подразделяются на группы:

- утилиты – программы, выполняющие отдельные задачи управления и сопровождения вычислительной системы;
- системные обрабатывающие программы – текстовые и графические редакторы (Paint, Imaging в Windows 2000), компиляторы и др.;
- программы предоставления пользователю дополнительных услуг (специальный вариант пользовательского интерфейса, калькулятор, игры, средства мультимедиа Windows 2000);
- библиотеки процедур различного назначения, упрощения разработки приложений, например, библиотека функций ввода-вывода, библиотека математических функций и т.п.

Эти модули ОС оформляются как обычные приложения, обращаются к функциям ядра посредством системных вызовов и выполняются в пользовательском режиме (user mode). В этом режиме запрещается выполнение некоторых команд, которые связаны с функциями ядра ОС (управление ресурсами, распределение и защита памяти и т.п.).

В концепции многоуровневой (многослойной) иерархической машины структура ОС также представляется рядом слоев. При такой организации каждый слой обслуживает вышележащий слой, выполняя для него некоторый набор функций, которые образуют межслойный интерфейс. На основе этих функций следующий верхний по иерархии слой строит свои функции – более сложные и более мощные и т.д. Такая организация системы существенно упрощает ее разработку, т.к. позволяет сначала "сверху вниз" определить функции слоев и

межслойные интерфейсы, а при детальной реализации, двигаясь "снизу вверх", – наращивать мощность функции слоев. Кроме того, модули каждого слоя можно изменять без необходимости изменений в других слоях (но не меняя межслойных интерфейсов!).

Многослойная структура ядра ОС может быть представлена, например, вариантом, показанным на рис. 1.4.

Рис. 1.4. Многослойная структура ОС

В данной схеме выделены следующие слои.

1. Средства аппаратной поддержки ОС. Значительная часть функций ОС может выполняться аппаратными средствами [10]. Чисто программных ОС сейчас не существует. Как правило, в современных системах всегда есть средства аппаратной поддержки ОС, которые прямо участвуют в организации вычислительных процессов. К ним относятся: система прерываний, средства поддержки привилегированного режима, средства поддержки виртуальной памяти, системный таймер,

средства переключения контекстов процессов (информации о состоянии процесса в момент его приостановки), средства защиты памяти и др.

2. Машинно-зависимые модули ОС. Этот слой образует модули, в которых отражается специфика аппаратной платформы компьютера. Назначение этого слоя – "экранирование" вышележащих слоев ОС от особенностей аппаратуры (например, Windows 2000 – это слой HAL (Hardware Abstraction Layer), уровень аппаратных абстракций).
3. Базовые механизмы ядра. Этот слой модулей выполняет наиболее примитивные операции ядра: программное переключение контекстов процессов, диспетчеризацию прерываний, перемещение страниц между основной памятью и диском и т.п. Модули этого слоя не принимают решений о распределении ресурсов, а только обрабатывают решения, принятые модулями вышележащих уровней. Поэтому их часто называют исполнительными механизмами для модулей верхних слоев ОС.
4. Менеджеры ресурсов. Модули этого слоя выполняют стратегические задачи по управлению ресурсами вычислительной системы. Это менеджеры (диспетчеры) процессов ввода-вывода, оперативной памяти и файловой системы. Каждый менеджер ведет учет свободных и используемых ресурсов и планирует их распределение в соответствии запросами приложений.
5. Интерфейс системных вызовов. Это верхний слой ядра ОС, взаимодействующий с приложениями и системными утилитами, он образует прикладной программный интерфейс ОС. Функции API, обслуживающие системные вызовы, предоставляют доступ к ресурсам системы в удобной компактной форме, без указания деталей их физического расположения.

Повышение устойчивости ОС обеспечивается переходом ядра в привилегированный режим. При этом происходит некоторое замедление выполнения системных вызовов. Системный вызов привилегированного ядра инициирует переключение процессора из пользовательского режима в привилегированный, а при возврате к приложению – обратное переключение. За счет этого возникает дополнительная задержка в обработке системного вызова (рис. 1.5). Однако такое решение стало классическим и используется во многих ОС (UNIX, VAX, VMS, IBM OS/390, OS/2 и др.).

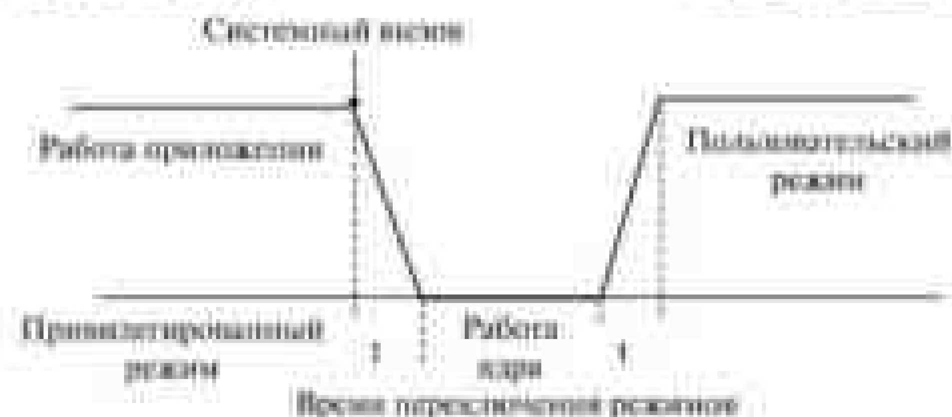


Рис. 1.5. Обработка системного вызова

Многослойная классическая многоуровневая архитектура ОС не лишена своих проблем. Дело в том, что значительные изменения одного из уровней могут иметь трудно предвидимое влияние на смежные уровни. Кроме того, многочисленные взаимодействия между соседними уровнями усложняют обеспечение безопасности. Поэтому, как альтернатива классическому варианту архитектуры ОС, часто используется микроядерная архитектура ОС.

Суть этой архитектуры состоит в следующем. В привилегированном режиме остается работать только очень небольшая часть ОС, называемая микроядром. Микроядро защищено от остальных частей ОС и приложений. В его состав входят машинно-зависимые модули, а также модули, выполняющие базовые механизмы обычного ядра. Все остальные более высокоуровневые функции ядра оформляются как модули, работающие в пользовательском режиме. Так, менеджеры ресурсов, являющиеся неотъемлемой частью обычного ядра, становятся "периферийными" модулями, работающими в пользовательском режиме. Таким образом, в архитектуре с микроядром традиционное расположение уровней по вертикали заменяется горизонтальным. Это можно представить, как показано на рис. 1.6.

Внешние по отношению к микроядру компоненты ОС реализуются как обслуживающие процессы. Между собой они взаимодействуют как равноправные партнеры с помощью обмена сообщениями, которые передаются через микроядро. Поскольку назначением этих компонентов

ОС является обслуживание запросов приложений пользователей, утилит и системных обрабатывающих программ, менеджеры ресурсов, вынесенные в пользовательский режим, называются серверами ОС, т.е. модулями, основным назначением которых является обслуживание запросов локальных приложений и других модулей ОС.

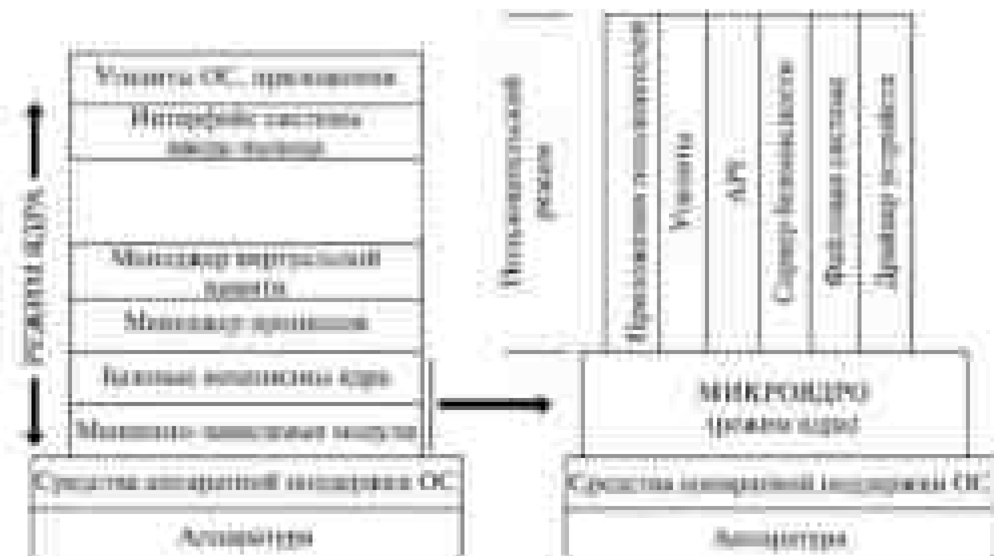


Рис. 1.6. Переход к микроядерной архитектуре

Схематично механизм обращений к функциям ОС, оформленным в виде серверов, выглядит, как показано на рис. 1.7.

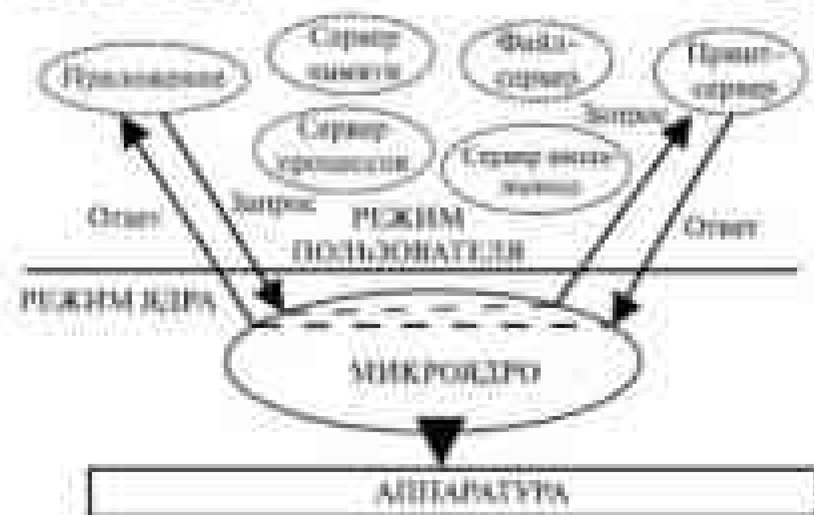


Рис. 1.7. Клиент-серверная архитектура

Схема смены режимов при выполнении системного вызова в ОС с микроядерной архитектурой выглядит, как показано на рис. 1.8. Из рисунка ясно, что выполнение системного вызова сопровождается четырьмя переключениями режимов (4 t), в то время как в классической архитектуре – двумя. Следовательно, производительность ОС с микроядерной архитектурой при прочих равных условиях будет ниже, чем у ОС с классическим ядром.

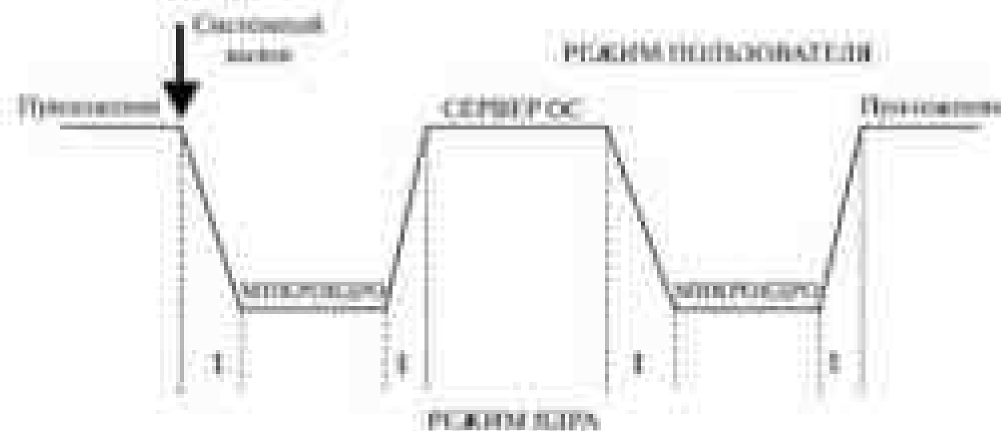


Рис. 1.8. Обработка системного вызова в микроядерной архитектуре

В то же время признаны следующие достоинства микроядерной архитектуры [17]:

- единообразные интерфейсы;
- простота расширяемости;
- высокая гибкость;
- возможность переносимости;
- высокая надежность;
- поддержка распределенных систем;
- поддержка объектно-ориентированных ОС.

По многим источникам вопрос масштабов потери производительности в микроядерных ОС является спорным. Многое зависит от размеров и функциональных возможностей микроядра. Избирательное увеличение функциональности микроядра приводит к снижению количества переключений между режимами системы, а также переключений адресных пространств процессов.

Может быть, это покажется парадоксальным, но есть и такой подход в микроядерной ОС, как уменьшение микроядра.

Для возможности представления о размерах микроядер операционных систем в ряде источников [17] приводятся такие данные:

- типичное микроядро первого поколения – 300 Кбайт кода и 140 интерфейсов системных вызовов;
- микроядро ОС L4 (второе поколение) – 12 Кбайт кода и 7 интерфейсов системных вызовов.

В современных операционных системах различают следующие виды ядер:

1. Наноядро (НЯ). Крайне упрощённое и минимальное ядро, выполняет лишь одну задачу – обработку аппаратных прерываний, генерируемых устройствами компьютера. После обработки посылает информацию о результатах обработки вышележащему программному обеспечению. НЯ используются для виртуализации аппаратного обеспечения реальных компьютеров или для реализации механизма гипервизора.

2. Микроядро (МЯ) предоставляет только элементарные функции управления процессами и минимальный набор абстракций для работы с оборудованием. Большая часть работы осуществляется с помощью специальных пользовательских процессов, называемых сервисами. В микроядерной операционной системе можно, не прерывая ее работы, загружать и выгружать новые драйверы, файловые системы и т. д. Микроядерными являются ядра ОС Minix и GNU Hurd и ядро систем семейства BSD. Классическим примером микроядерной системы является Symbian OS. Это пример распространенной и отработанной микроядерной (а начиная с версии Symbian OS v8.1, и наноядерной) операционной системы.
3. Экзоядро (ЭЯ) предоставляет лишь набор сервисов для взаимодействия между приложениями, а также необходимый минимум функций, связанных с защитой: выделение и высвобождение ресурсов, контроль прав доступа и т. д. ЭЯ не занимается предоставлением абстракций для физических ресурсов – эти функции выносятся в библиотеку пользовательского уровня (так называемую libOS). В отличие от микроядра ОС, базирующиеся на ЭЯ, обеспечивают большую эффективность за счет отсутствия необходимости в переключении между процессами при каждом обращении к оборудованию.
4. Монолитное ядро (МнЯ) предоставляет широкий набор абстракций оборудования. Все части ядра работают в одном адресном пространстве. МнЯ требует перекомпиляции при изменении состава оборудования. Компоненты операционной системы являются не самостоятельными модулями, а составными частями одной программы. МнЯ более производительно, чем микроядро, поскольку работает как один большой процесс. МнЯ является большинство Unix-систем и Linux. Монолитность ядер усложняет отладку, понимание кода ядра, добавление новых функций и возможностей, удаление ненужного, унаследованного от предыдущих версий кода. "Разбухание" кода монолитных ядер также повышает требования к объему оперативной памяти.
5. Модульное ядро (Мод. Я) – современная, усовершенствованная модификация архитектуры МЯ. В отличие от "классических" МнЯ, модульные ядра не требуют полной перекомпиляции ядра при изменении состава аппаратного обеспечения компьютера. Вместо этого они предоставляют тот или иной механизм подгрузки

модулей, поддерживающих то или иное аппаратное обеспечение (например, драйверов). Подгрузка модулей может быть как динамической, так и статической (при перезагрузке ОС после переконфигурирования системы). Мод. Я удобнее для разработки, чем традиционные монолитные ядра. Они предоставляют программный интерфейс (API) для связывания модулей с ядром, для обеспечения динамической подгрузки и выгрузки модулей. Не все части ядра могут быть сделаны модулями. Некоторые части ядра всегда обязаны присутствовать в оперативной памяти и должны быть жестко "вшиты" в ядро.

6. Гибридное ядро (ГЯ) – модифицированные микроядра, позволяющие для ускорения работы запускать "несущественные" части в пространстве ядра. Имеют "гибридные" достоинства и недостатки. Примером смешанного подхода может служить возможность запуска операционной системы с монолитным ядром под управлением микроядра. Так устроены 4.4BSD и MkLinux, основанные на микроядре Mach. Микроядро обеспечивает управление виртуальной памятью и работу низкоуровневых драйверов. Все остальные функции, в том числе взаимодействие с прикладными программами, осуществляются монолитным ядром. Данный подход сформировался в результате попыток использовать преимущества микроядерной архитектуры, сохраняя по возможности хорошо отлаженный код монолитного ядра.
7. Наиболее тесно элементы микроядерной архитектуры и элементы монолитного ядра переплетены в ядре Windows NT. Хотя Windows NT часто называют микроядерной операционной системой, это не совсем так. Микроядро NT слишком велико (более 1 Мбайт), чтобы носить приставку "микро". Компоненты ядра Windows NT располагаются в вытесняемой памяти и взаимодействуют друг с другом путем передачи сообщений, как и положено в микроядерных операционных системах. В то же время все компоненты ядра работают в одном адресном пространстве и активно используют общие структуры данных, что свойственно операционным системам с монолитным ядром.

1.6. Классификация операционных систем

Все многообразие существующих (и ныне не использующихся) ОС можно классифицировать по множеству различных признаков. Остановимся на основных классификационных признаках.

1. По назначению ОС делятся на универсальные и специализированные. Специализированные ОС, как правило, работают с фиксированным набором программ (функциональных задач). Применение таких систем обусловлено невозможностью использования универсальной ОС по соображениям эффективности, надежности, защищенности и т.п., а также вследствие специфики решаемых задач [10].

   Универсальные ОС рассчитаны на решение любых задач пользователей, но, как правило, форма эксплуатации вычислительной системы может предъявлять особые требования к ОС, т.е. к элементам ее специализации.

2. По способу загрузки можно выделить загружаемые ОС (большинство) и системы, постоянно находящиеся в памяти вычислительной системы. Последние, как правило, специализированные и используются для управления работой специализированных устройств (например, в БЦВМ баллистической ракеты или спутника, научных приборах, автоматических устройствах различного назначения и др.)
3. По особенностям алгоритмов управления ресурсами. Главным ресурсом системы является процессор, поэтому дадим классификацию по алгоритмам управления процессором, хотя можно, конечно, классифицировать ОС по алгоритмам управления памятью, устройствами ввода-вывода и т.д.

   - Поддержка многозадачности (многопрограммности). По числу одновременно выполняемых задач ОС делятся на 2 класса: однопрограммные (однозадачные) – например, MS-DOS, MSX, и многопрограммные (многозадачные) – например, ОС ЕС ЭВМ, OS/360, OS/2, UNIX, Windows разных версий.

     Однопрограммные ОС предоставляют пользователю виртуальную машину, делая более простым и удобным

процесс взаимодействия пользователя с компьютером. Они также имеют средства управления файлами, периферийными устройствами и средства общения с пользователем. Многозадачные ОС, кроме того, управляют разделением совместно используемых ресурсов (процессор, память, файлы и т.д.), что позволяет значительно повысить эффективность вычислительной системы.

- Поддержка многопользовательского режима. По числу одновременно работающих пользователей ОС делятся на однопользовательские (MS-DOS, Windows 3x, ранние версии OS/2) и многопользовательские (UNIX, Windows NT/2000/2003/XP/Vista).

  Главное отличие многопользовательских систем от однопользовательских – наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей. Следует заметить, что может быть однопользовательская мультипрограммная система.

- Виды многопрограммной работы. Специфику ОС во многом определяет способ распределения времени между несколькими одновременно существующими в системе процессами (или потоками). По этому признаку можно выделить 2 группы алгоритмов: не вытесняющая многопрограммность (Windows3.x, NetWare) и вытесняющая многопрограммность (Windows 2000/2003/XP, OS/2, Unix).

  В первом случае активный процесс выполняется до тех пор, пока он сам не отдает управление операционной системе. Во втором случае решение о переключении процессов принимает операционная система. Возможен и такой режим многопрограммности, когда ОС разделяет процессорное время между отдельными ветвями (потоками, волокнами) одного процесса.

- Многопроцессорная обработка. Важное свойство ОС – отсутствие или наличие средств поддержки многопроцессорной обработки. По этому признаку можно

выделить ОС без поддержки мультипроцессирования (Windows 3.x, Windows 95) и с поддержкой мультипроцессирования (Solaris, OS/2, UNIX, Windows NT/2000/2003/XP).

Многопроцессорные ОС классифицируются по способу организации вычислительного процесса на асимметричные ОС (выполняются на одном процессоре, распределяя прикладные задачи по остальным процессорам) и симметричные ОС (децентрализованная система).

4. По области использования и форме эксплуатации. Обычно здесь выделяют три типа в соответствии с использованными при их разработке критериями эффективности:
   - системы пакетной обработки (OS/360, ОС ЕС);
   - системы разделения времени (UNIX, VMS);
   - системы реального времени (QNX, RT11).

   Первые предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Критерий создания таких ОС – максимальная пропускная способность при хорошей загрузке всех ресурсов компьютера. В таких системах пользователь отстранен от компьютера.

   Системы разделения времени обеспечивают удобство и эффективность работы пользователя, который имеет терминал и может вести диалог со своей программой.

   Системы реального времени предназначены для управления техническими объектами (станок, спутник, технологический процесс, например доменный и т.п.), где существует предельное время на выполнение программ, управляющих объектом.

5. По аппаратной платформе (типу вычислительной техники), для которой они предназначаются, операционные системы делят на следующие группы.
   - Операционные системы для смарт-карт. Некоторые из них могут управлять только одной операцией, например, электронным платежом. Некоторые смарт-карты являются

JAVA-ориентированным и содержат интерпретатор виртуальной машины JAVA. Апплеты JAVA загружаются на карту и выполняются JVM-интерпретатором. Некоторые из таких карт могут одновременно управлять несколькими апплетами JAVA, что приводит к многозадачности и необходимости планирования.

- Встроенные операционные системы. Управляют карманными компьютерами (Palm OS, Windows CE – Consumer Electronics – бытовая техника), мобильными телефонами, телевизорами, микроволновыми печами и т.п.
- Операционные системы для персональных компьютеров, например, Windows 9.x, Windows XP, Linux, Mac OSX и др.
- Операционные системы мини-ЭВМ, например, RT-11 для PDP-11 – ОС реального времени, RSX-11 M для PDP-11 – ОС разделения времени, UNIX для PDP-7.
- Операционные системы мейнфреймов (больших машин), например, OS/390, происходящая от OS/360 (IBM). Обычно ОС мейнфреймов предполагает одновременно три вида обслуживания: пакетную обработку, обработку транзакций (например, работа с БД, бронирование авиабилетов, процесс работы в банках) и разделение времени.
- Серверные операционные системы, например, UNIX, Windows 2000, Linux. Область применения – ЛВС, региональные сети, Intranet, Internet.
- Кластерные операционные системы. Кластер – слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений и представляющихся пользователю единой системой, например, Windows 2000 Cluster Server, Windows 2000 Server, Sun Cluster (базовая ОС – Solaris).

1.7. Эффективность и требования, предъявляемые к ОС

К операционным системам современных компьютеров предъявляется ряд требований. Главным требованием является выполнение основных функций эффективного управления ресурсами и обеспечения удобного интерфейса для пользователя и прикладных программ. Современная

ОС должны поддерживать мультипрограммную обработку, виртуальную память, свопинг, развитый интерфейс пользователя (многооконный графический, аудио -, менюориентированный и т.д.), высокую степень защиты, удобство работы, а также выполнять многие другие необходимые функции и услуги. Кроме этих требований функциональной полноты, к ОС предъявляется ряд важных эксплуатационных требований.

1. Эффективность. Под эффективностью вообще любой технической (да и не только технической) системы понимается степень соответствия системы своему назначению, которая оценивается некоторым множеством показателей эффективности [10].

   Поскольку ОС представляет собой сложную программную систему, она использует для собственных нужд значительную часть ресурсов компьютера. Часто эффективность ОС оценивают ее производительностью (пропускной способностью) – количеством задач пользователей, выполняемых за некоторый промежуток времени, временем реакции на запрос пользователя и др.

   На все эти показатели эффективности ОС влияет много различных факторов, среди которых основными являются архитектура ОС, многообразие ее функций, качество программного кода, аппаратная платформа (компьютер) и др.

2. Надежность и отказоустойчивость. Операционная система должна быть, по меньшей мере, так же надежна, как компьютер, на котором она работает. Система должна быть защищена как от внутренних, так и от внешних сбоев и отказов. В случае ошибки в программе или аппаратуре система должна обнаружить ошибку и попытаться исправить положение или, по крайней мере, постараться свести к минимуму ущерб, нанесенный этой ошибкой пользователям.

   Надежность и отказоустойчивость ОС, прежде всего, определяются архитектурными решениями, положенными в ее основу, а также отлаженностью программного кода (основные отказы и сбои ОС в основном обусловлены программными

ошибками в ее модулях. Кроме того, важно, чтобы компьютер имел резервные дисковые массивы, источники бесперебойного питания и др., а также программную поддержку этих средств.

3. Безопасность (защищенность). Ни один пользователь не хочет, чтобы другие пользователи ему мешали. ОС должна защищать пользователей и от воздействия чужих ошибок, и от попыток злонамеренного вмешательства (несанкционированного доступа). С этой целью в ОС как минимум должны быть средства аутентификации – определения легальности пользователей, авторизации – предоставления легальным пользователям установленных им прав доступа к ресурсам, и аудита – фиксации всех потенциально опасных для системы событий.

   Свойства безопасности особенно важны для сетевых ОС. В таких ОС к задаче контроля доступа добавляется задача защиты данных, передаваемых по сети.

4. Предсказуемость. Требования, которые пользователь может предъявить к системе, в большинстве случаев непредсказуемы. В то же время пользователь предпочитает, чтобы обслуживание не очень сильно менялось в течение предполагаемого времени. В частности, запуская свою программу в системе, пользователь должен иметь основанное на опыте работы с этой программой приблизительное представление, когда ему ожидать выдачи результатов.
5. Расширяемость. В отличие от аппаратных средств компьютера полезная жизнь операционных систем измеряется десятками лет. Примером может служить ОС UNIX, да и MS-DOS. Операционные системы изменяются со временем, как правило, за счет приобретения новых свойств, например, поддержки новых типов внешних устройств или новых сетевых технологий. Если программный код модулей ОС написан таким образом, что дополнения и изменения могут вноситься без нарушения целостности системы, то такую ОС называют расширяемой. Операционная система может быть расширяемой, если при ее создании руководствовались принципами модульности, функциональной избыточности, функциональной избирательности и параметрической универсальности.

6. Переносимость. В идеальном случае код ОС должен легко переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которые различаются не только типом процессора, но и способом организации всей аппаратуры компьютера) одного типа на аппаратную платформу другого типа. Переносимые ОС имеют несколько вариантов реализации для разных платформ, такое свойство ОС называется также многоплатформенностью. Достигается это свойство за счет того, что основная часть ОС пишется на языке высокого уровня (например С, С++ и др.) и может быть легко перенесена на другой компьютер (машинно-независимая часть), а некоторая меньшая часть ОС (программы ядра) является машинно-зависимой и разрабатывается на машинном языке другого компьютера.
7. Совместимость. Существует несколько "долгоживущих" популярных ОС (разновидности UNIX, MS-DOS, Windows3.x, Windows NT, OS/2), для которых наработана широкая номенклатура приложений. Для пользователя, переходящего с одной ОС на другую, очень привлекательна возможность – выполнить свои приложения в новой операционной системе. Если ОС имеет средства для выполнения прикладных программ, написанных для других операционных систем, то она совместима с этими системами. Следует различать совместимость на уровне двоичных кодов и совместимость на уровне исходных текстов. Кроме того, понятие совместимости включает также поддержку пользовательских интерфейсов других ОС.
8. Удобство. Средства ОС должны быть простыми и гибкими, а логика ее работы ясна пользователю. Современные ОС ориентированы на обеспечение пользователю максимально возможного удобства при работе с ними. Необходимым условием этого стало наличие у ОС графического пользовательского интерфейса и всевозможных мастеров – программ, автоматизирующих активизацию функций ОС, подключение периферийных устройств, установку, настройку и эксплуатацию самой ОС.
9. Масштабируемость. Если ОС позволяет управлять компьютером с различным числом процессоров, обеспечивая линейное (или почти такое) возрастание производительности при увеличении числа процессоров, то такая ОС является масштабируемой. В масштабируемой ОС реализуется симметричная

многопроцессорная обработка. С масштабируемостью связано понятие кластеризации – объединения в систему двух (и более) многопроцессорных компьютеров. Правда, кластеризация направлена не столько на масштабируемость, сколько на обеспечение высокой готовности системы.

Следует заметить, что в зависимости от области применения конкретной операционной системы может изменяться и состав предъявляемых к ней требований.

Производители могут предлагать свои ОС в различных, различающихся ценой и производительностью, конфигурациях. Например, Microsoft продает [[illegible]]:

- Windows 2003 Server (до 4-х процессоров) – для малого и среднего бизнеса;
- Windows 2003 Advanced Server (до 8 процессоров, 2-узловой кластер) – для средних и крупных предприятий;
- Windows 2003 DataCenter Server (16-32 процессора, 4-узловой кластер) – для особо крупных предприятий.

1.8. Совместимость и множественные прикладные среды

В то время как многие архитектурные особенности ОС непосредственно касаются только системных программистов, концепция множественных прикладных (операционных) средств непосредственно связана с нуждами конечных пользователей – возможностью операционной системы выполнять приложения, написанные для других операционных систем. Такое свойство операционной системы называется совместимостью.

Совместимость приложений может быть на двоичном уровне и на уровне исходных текстов [[illegible]]. Приложения обычно хранятся в ОС в виде исполняемых файлов, содержащих двоичные образы кодов и данных. Двоичная совместимость достигается в том случае, если можно взять исполняемую программу и запустить ее на выполнение в среде другой ОС.

Совместимость на уровне исходных текстов требует наличия соответствующего компилятора в составе программного обеспечения компьютера, на котором предполагается выполнять данное приложение, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция исходных текстов приложения в новый исполняемый модуль.

Совместимость на уровне исходных текстов важна в основном для разработчиков приложений, в распоряжении которых эти исходные тексты находятся. Но для конечных пользователей практическое значение имеет только двоичная совместимость, так как только в этом случае они могут использовать один и тот же продукт в различных операционных системах и на различных машинах.

Вид возможной совместимости зависит от многих факторов. Самый главный из них – архитектура процессора. Если процессор применяет тот же набор команд (возможно, с добавлениями, как в случае IBM PC: стандартный набор + мультимедиа + графика + потоковые) и тот же диапазон адресов, то двоичная совместимость может быть достигнута достаточно просто. Для этого необходимо соблюдение следующих условий:

- API, который использует приложение, должен поддерживаться данной ОС;
- внутренняя структура исполняемого файла приложения должна соответствовать структуре исполняемых файлов данной ОС..

Если процессоры имеют разную архитектуру, то, кроме перечисленных условий, необходимо организовать эмуляцию двоичного кода. Например, широко используется эмуляция команд процессора Intel на процессоре Motorola 680x0 компьютера Macintosh. Программный эмулятор в этом случае последовательно выбирает двоичную инструкцию процессора Intel и выполняет эквивалентную подпрограмму, написанную в инструкциях процессора Motorola. Так как у процессора Motorola нет в точности таких же регистров, флагов, внутреннего АЛУ и др., как в процессорах Intel, он должен также имитировать (эмулировать) все эти элементы с использованием своих регистров или памяти.

Это просто, но очень медленная работа, поскольку одна команда Intel выполняется значительно быстрее, чем эмулирующая ее последовательность команд процессора Motorola. Выходом в таких случаях является применение так называемых прикладных программных сред или операционных сред. Одной из составляющих такой среды является набор функций интерфейса прикладного программирования API, который ОС предоставляет своим приложениям. Для сокращения времени на выполнение чужих программ прикладные среды имитируют обращение к библиотечным функциям.

Эффективность этого подхода связана с тем, что большинство сегодняшних программ работают под управлением GUI (графических интерфейсов пользователя) типа Windows, MAC или UNIX Motif, при этом приложения тратят 60-80% времени на выполнение функций GUI и других библиотечных вызовов ОС. Именно это свойство приложений позволяет прикладным средам компенсировать большие затраты времени, потраченные на покомандное эмулирование программ. Тщательно спроектированная программная прикладная среда имеет в своем составе библиотеки, имитирующие библиотеки GUI, но написанные на "родном" коде. Таким образом, достигается существенное ускорение выполнения программ с API другой операционной системы. Иначе такой подход называют трансляцией – для того, чтобы отличать его от более медленного процесса эмулирования по одной команде за раз.

Например, для Windows-программы, работающей на Macintosh, при интерпретации команд процессора Intel производительность может быть очень низкой. Но когда производится вызов функции GUI, открытие окна и др., модуль ОС, реализующий прикладную среду Windows, может перехватить этот вызов и перенаправить его на перекомпилированную для процессора Motorola 680x0 подпрограмму открытия окна. В результате на таких участках кода скорость работы программы может достичь (а, возможно, и превзойти) скорость работы на своем родном процессоре.

Чтобы программа, написанная для одной ОС, могла быть выполнена в рамках другой ОС, недостаточно лишь обеспечивать совместимость API. Концепции, положенные в основу разных ОС, могут входить в

противоречии друг с другом. Например, в одной ОС приложению может быть разрешено управлять устройствами ввода-вывода, в другой – эти действия являются прерогативой ОС.

Каждая ОС имеет свои собственные механизмы защиты ресурсов, свои алгоритмы обработки ошибок и исключительных ситуаций, особую структуру процессора и схему управления памятью, свою семантику доступа к файлам и графический пользовательский интерфейс. Для обеспечения совместимости необходимо организовать бесконфликтное сосуществование в рамках одной ОС нескольких способов управления ресурсами компьютера.

Существуют различные варианты построения множественных прикладных сред, отличающиеся как особенностями архитектурных решений, так и функциональными возможностями, обеспечивающими разную степень переносимости приложений. Один из наиболее очевидных вариантов реализации множественных прикладных сред основывается на стандартной многоуровневой структуре ОС.

На рис. 1.9 ОС OS1 поддерживает кроме своих "родных" приложений приложения операционных систем OS2 и OS3. Для этого в её составе имеются специальные приложения, прикладные программные среды, которые транслируют интерфейсы "чужих" операционных систем API OS2 и API OS3 в интерфейс своей "родной" ОС – API OS1. Так, например, в случае если бы в качестве OS2 выступала ОС UNIX, а в качестве OS1 – OS/2, для выполнения системного вызова создания процесса fork () в UNIX-приложении программная среда должна обращаться к ядру операционной системы OS/2 с системным вызовом DOS ExecPgm ().

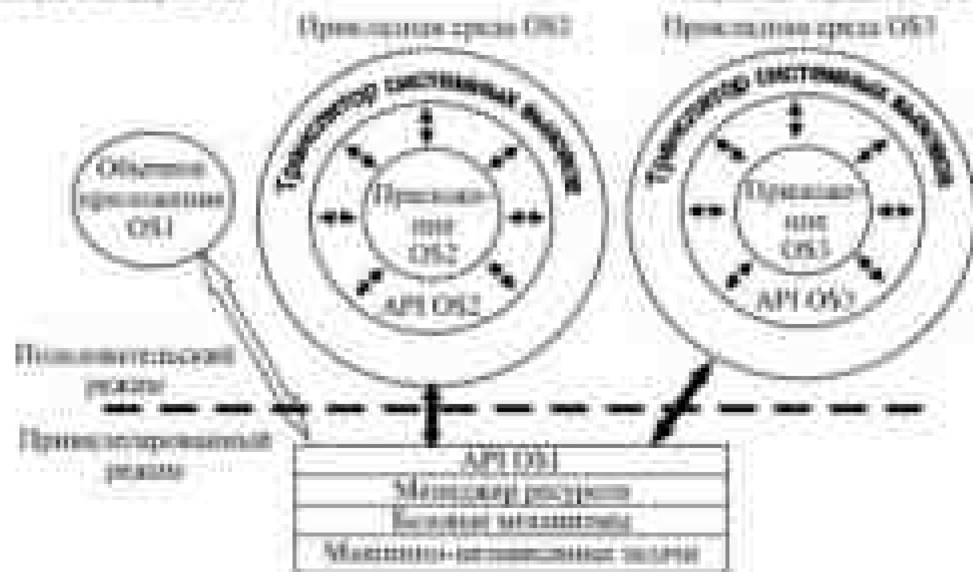


Рис. 1.9. Организация множественных прикладных сред

К сожалению, поведение почти всех функций, составляющих API одной ОС, как правило, существенно отличается от поведения соответствующих функций другой ОС. Например, чтобы функция создания процесса в OS/2 Dos ExecPgm () полностью соответствовала функции создания процесса fork () в UNIX-подобных системах, ее нужно было бы изменить и написать новую функциональность: поддержку возможности копирования адресного пространства родительского процесса в пространство процесса-потомка [17].

Еще один способ построения множественных прикладных сред основан на микроядерном подходе. При этом очень важно отметить базовое, общее для всех прикладных сред отличие механизмов операционной системы от специфических для каждой из прикладных сред высокоуровневых функций, решающих стратегические задачи. В соответствии с микроядерной архитектурой все функции ОС реализуются микроядром и серверами пользовательского режима. Важно, что прикладная среда оформляется в виде отдельного сервера пользовательского режима и не включает базовых механизмов.

Приложения, используя API , обращаются с системными вызовами к соответствующей прикладной среде через микроядро. Прикладная среда

обрабатывает запрос, выполняет его (возможно, обращаясь для этого за помощью к базовым функциям микроядра) и отсылает приложению результат. В ходе выполнения запроса прикладной среде приходится, в свою очередь, обращаться к базовым механизмам ОС, реализуемым микроядром и другими серверами ОС.

Такому подходу к конструированию множественных прикладных сред присущи все достоинства и недостатки микро-ядерной архитектуры, в частности:

- очень просто можно добавлять и исключать прикладные среды, что является следствием хорошей расширяемости микро-ядерных ОС;
- при отказе одной из прикладных сред остальные сохраняют работоспособность, что способствует надежности и стабильности системы в целом;
- низкая производительность микроядерных ОС сказывается на скорости работы прикладных средств, а значит, и на скорости работы приложений.

В итоге следует отметить, что создание в рамках одной ОС нескольких прикладных средств для выполнения приложений различных ОС представляет собой путь, который позволяет иметь единственную версию программы и переносить ее между различными операционными системами. Множественные прикладные среды обеспечивают совместимость на двоичном уровне данной ОС с приложениями, написанными для других ОС.

1.9. Виртуальные машины как современный подход к реализации множественных прикладных сред

Понятие "монитор виртуальных машин" (МВМ) возникло в конце 60-х годов как программный уровень абстракции, разделявший аппаратную платформу на несколько виртуальных машин. Каждая из этих виртуальных машин (ВМ) была настолько похожа на базовую физическую машину, что существующее программное обеспечение могло выполняться на ней в неизменном виде. В то время вычислительные задачи общего характера решались на дорогих мэйнфреймах (типа

IBM/360), и пользователи высоко оценили способность МВМ распределять дефицитные ресурсы среди нескольких приложений.

В 80-90-е годы существенно снизилась стоимость компьютерного оборудования и появились эффективные многозадачные ОС, что уменьшило ценность МВМ в глазах пользователей. Мэйнфреймы уступили место мини-компьютерам, а затем ПК, и нужда в МВМ отпала. В результате из компьютерной архитектуры попросту исчезли аппаратные средства для их эффективной реализации. К концу 80-х в науке и на производстве МВМ воспринимались не иначе как исторический курьез [10].

Сегодня МВМ – снова в центре внимания. Корпорации Intel, AMD, Sun Microsystems и IBM создают стратегии виртуализации, в научных лабораториях и университетах для решения проблем мобильности, обеспечения безопасности и управляемости развиваются подходы, основанные на виртуальных машинах. Что же произошло между отставкой МВМ и их возрождением?

В 90-е годы исследователи из Стэнфордского университета начали изучать возможность применения ВМ для преодоления ограничений оборудования и операционных систем. Проблемы возникли у компьютеров с массовой параллельной обработкой (Massively Parallel Processing, MPP), которые плохо поддавались программированию и не могли выполнять имеющиеся ОС. Исследователи обнаружили, что с помощью виртуальных машин можно сделать эту неудобную архитектуру достаточно похожей на существующие платформы, чтобы использовать преимущества готовых ОС. Из этого проекта вышли люди и идеи, ставшие золотым фондом компании VMware (www.vmware.com), первого поставщика МВМ для компьютеров массового применения.

Как ни странно, развитие современных ОС и снижение стоимости оборудования привели к появлению проблем, которые исследователи надеялись решить с помощью МВМ. Дешевизна оборудования способствовала быстрому распространению компьютеров, но они часто бывали недогруженными, требовали дополнительных площадей и усилий по обслуживанию. А следствиями роста функциональных возможностей ОС стали их неустойчивость и уязвимость.

Чтобы уменьшить влияние системных аварий и защититься от взломов,

системные администраторы вновь обратились к однозадачной вычислительной модели (с одним приложением на одной машине). Это привело к дополнительным расходам, вызванным повышенными требованиями к оборудованию. Перенос приложений с разных физических машин на ВМ и консолидация этих ВМ на немногих физических платформах позволили повысить эффективность использования оборудования, снизить затраты на управление и производственные площади. Таким образом, способность МВМ к мультиплексированию аппаратных средств – на этот раз во имя консолидации серверов и организации коммунальных вычислений – снова возродила их к жизни.

В настоящее время МВМ стал не столько средством организации многозадачности, каким он был когда-то задуман, сколько решением проблем обеспечения безопасности, мобильности и надежности. Во многих отношениях МВМ дает создателям операционных систем возможность развития функциональности, невозможной в нынешних сложных ОС. Такие функции, как миграция и защита, намного удобнее реализовать на уровне МВМ, поддерживающих обратную совместимость при развертывании инновационных решений в области операционных систем при сохранении предыдущих достижений.

Виртуализация – развивающаяся технология. В общих словах, виртуализация позволяет отделить ПО от нижележащей аппаратной инфраструктуры. Фактически она разрывает связь между определенным набором программ и конкретным компьютером. Монитор виртуальных машин отделяет программное обеспечение от оборудования и формирует промежуточный уровень между ПО, выполняемым виртуальными машинами, и аппаратными средствами. Этот уровень позволяет МВМ полностью контролировать использование аппаратных ресурсов гостевыми операционными системами (GuestOS), которые выполняются на ВМ.

МВМ создает унифицированное представление базовых аппаратных средств, благодаря чему физические машины различных поставщиков с разными подсистемами ввода-вывода выглядят одинаково и ВМ выполняются на любом доступном оборудовании. Не заботясь об отдельных машинах с их тесными взаимосвязями между аппаратными средствами и программным обеспечением, администраторы могут

рассматривать оборудование просто как пул ресурсов для оказания любых услуг по требованию.

Благодаря полной инкапсуляции состояния ПО на ВМ монитор МВМ может отображать ВМ на любые доступные аппаратные ресурсы и даже перенести с одной физической машины на другую. Задача балансировки нагрузки в группе машин становится тривиальной, и появляются надежные способы борьбы с отказами оборудования и наращивания системы. Если нужно отключить отказавший компьютер или ввести в строй новый, МВМ способен соответствующим образом перераспределить виртуальные машины. Виртуальную машину легко тиражировать, что позволяет администраторам по мере необходимости оперативно предоставлять новые услуги.

Инкапсуляция также означает, что администратор может в любой момент приостановить или возобновить работу ВМ, а также сохранить текущее состояние виртуальной машины либо вернуть ее к предыдущему состоянию. Располагая возможностью универсальной отмены, удается легко справиться с авариями и ошибками конфигурации. Инкапсуляция является основой обобщенной модели мобильности, поскольку приостановленную ВМ можно копировать по сети, сохранять и транспортировать на сменных носителях.

МВМ играет роль посредника во всех взаимодействиях между ВМ и базовым оборудованием, поддерживая выполнение множества виртуальных машин на единой аппаратной платформе и обеспечивая их надежную изоляцию. МВМ позволяет собрать группу ВМ с низкими потребностями в ресурсах на отдельном компьютере, снизив затраты на аппаратные средства и потребность в производственных площадях.

Полная изоляция также важна для надежности и обеспечения безопасности. Приложения, которые раньше выполнялись на одной машине, теперь можно распределить по разным ВМ. Если одно из них в результате ошибки вызовет аварию ОС, другие приложения будут от нее изолированы и продолжат работу. Если же одному из приложений угрожает внешнее нападение, атака будет локализована в пределах "скомпрометированной" ВМ. Таким образом, МВМ – это инструмент реструктуризации системы для повышения ее устойчивости и безопасности, не требующий дополнительных площадей и усилий по

администрированию, которые необходимы при выполнении приложений на отдельных физических машинах.

МВМ должен связать аппаратный интерфейс с ВМ, сохраняя полный контроль над базовой машиной и процедурами взаимодействия с ее аппаратными средствами. Для достижения этой цели существуют разные методы, основанные на определенных технических компромиссах. При поиске таких компромиссов принимаются во внимание основные требования к МВМ: совместимость, производительность и простота. Совместимость важна потому, что главное достоинство МВМ – способность выполнять унаследованные приложения. Производительность определяет величину накладных расходов на виртуализацию – программы на ВМ должны выполняться с той же скоростью, что и на реальной машине. Простота необходима, поскольку отказ МВМ приведет к отказу всех ВМ, выполняющихся на компьютере. В частности, для надежной изоляции требуется, чтобы МВМ был свободен от ошибок, которые злоумышленники могут использовать для разрушения системы.

Вместо того чтобы заниматься сложной переработкой кода гостевой операционной системы, можно внести некоторые изменения в основную операционную систему, изменяя некоторые наиболее "мешающие" части ядра. Подобный подход называется паравиртуализацией [10]. Ясно, что в этом случае адаптировать ядро ОС может только автор, и, например, Microsoft не проявляет желания адаптировать популярное ядро Windows 2000 к реалиям конкретных виртуальных машин.

При паравиртуализации разработчик МВМ переопределяет интерфейс виртуальной машины, заменяя непригодное для виртуализации подмножество исходной системы команд более удобными и эффективными эквивалентами. Заметим, что хотя ОС нужно портировать для выполнения на таких ВМ, большинство обычных приложений могут выполняться в неизменном виде.

Самый большой недостаток паравиртуализации – несовместимость. Любая операционная система, предназначенная для выполнения под управлением паравиртуализованного монитора МВМ, должна быть портирована в эту архитектуру, для чего нужно договариваться и

сотрудничестве с поставщиками ОС. Кроме того, нельзя использовать унаследованные операционные системы, а существующие машины не удается легко заменить виртуальными.

Чтобы добиться высокой производительности и совместимости при виртуализации архитектуры x86, компания VMware разработала новый метод виртуализации, который объединяет традиционное прямое выполнение с быстрой трансляцией двоичного кода "на лету". В большинстве современных ОС режимы работы процессора при выполнении обычных прикладных программ легко поддаются виртуализации, а следовательно, их можно виртуализировать посредством прямого выполнения. Непригодные для виртуализации привилегированные режимы может выполнять транслятор двоичного кода, исправляя "неудобные" команды x86. В результате получается высокопроизводительная виртуальная машина, которая полностью соответствует оборудованию и поддерживает полную совместимость ПО.

Преобразованный код очень похож на результаты паравиртуализации. Обычные команды выполняются в неизменном виде, а команды, нуждающиеся в специальной обработке (такие как POPF и команды чтения регистров сегмента кода), транслятор заменяет последовательностями команд, которые подобны требующимся для выполнения на паравиртуализованной виртуальной машине. Однако есть важное различие: вместо того, чтобы изменять исходный код операционной системы или приложений, транслятор двоичного кода изменяет код при его выполнении в первый раз.

Хотя трансляция двоичного кода требует некоторых дополнительных расходов, при нормальных рабочих нагрузках они незначительны. Транслятор обрабатывает лишь часть кода, и скорость выполнения программ становится сопоставимой со скоростью прямого выполнения – как только заполнится кэш-память трассировки.

Трансляция двоичного кода также помогает оптимизировать прямое выполнение. Например, если при прямом выполнении привилегированного кода часто происходит перехват команд, это может привести к существенным дополнительным расходам, поскольку при каждом перехвате управление передается от виртуальной машины к

монитору и обратно. Трансляция кода может устранить многие из таких переходов, что приведет к снижению накладных расходов на виртуализацию. Это особенно верно для центральных процессоров с длинными конвейерами команд, в частности, для современного семейства x86, в котором переход связан с высокими дополнительными расходами.

1.10. Эффекты виртуализации

Экспертиза современных продуктов и недавние исследования раскрывают некоторые интересные возможности развития МВМ и требования, которые они предъявят к технологиям виртуализации.

Администраторы центра данных могут с единой консоли быстро вводить в действие ВМ и управлять тысячами виртуальных машин, выполняющихся на сотнях физических серверов. Вместо того чтобы конфигурировать отдельные компьютеры, администраторы будут создавать по имеющимся шаблонам новые экземпляры виртуальных серверов и отображать их на физические ресурсы в соответствии с политиками администрирования. Уйдет в прошлое взгляд на компьютер как на средство предоставления конкретных услуг. Администраторы будут рассматривать компьютеры просто как часть пула универсальных аппаратных ресурсов (примером тому может служить виртуальный центр VMware VirtualCenter).

Отображение виртуальных машин на аппаратные ресурсы очень динамично. Возможность миграции работающих ВМ (подобные тем, которые обеспечивает технология VMotion компании VMware) позволяет ВМ быстро перемещаться между физическими машинами в соответствии с потребностями центра данных. МВМ сможет справиться с такими традиционными проблемами, как отказ оборудования, за счет простого перемещения ВМ с отказавшего компьютера на исправный. Возможность перемещения работающих ВМ облегчит решение аппаратных проблем, таких как планирование профилактического обслуживания, окончание срока действия лизингового договора и модернизация оборудования; администраторы станут устранять эти проблемы без перерывов в работе.

Еще недавно нормой являлась ручная миграция, но сейчас уже

распространены инфраструктуры виртуальных машин, которые автоматически выполняют балансировку нагрузки, прогнозируют отказы аппаратных средств и соответствующим образом перемещают ВМ, создают их и уничтожают в соответствии со спросом на конкретные услуги.

Решение проблем на уровне МВМ положительно сказывается на всех программах, выполняющихся на ВМ, независимо от их возраста (унаследованная или новейшая) и поставщиков. Независимость от ОС избавляет от необходимости покупать и обслуживать избыточную инфраструктуру. Например, из нескольких версий ПО службы поддержки или резервного копирования остается лишь одна – та, которая работает на уровне МВМ.

Виртуальные машины сильно изменили отношение к компьютерам. Уже сейчас простые пользователи умеют легко создавать, копировать и совместно использовать ВМ. Модели их применения значительно отличаются от привычных, сложившихся в условиях вычислительной среды с ограниченной доступностью аппаратных средств. А разработчики ПО могут применять такие продукты, как VMware Workstation, чтобы легко установить компьютерную сеть для тестирования или создать собственный набор испытательных машин для каждой цели.

Повышенная мобильность ВМ значительно изменила способы их применения. Такие проекты, как Collective и Internet Suspend/Resume, демонстрируют возможность перемещения всей вычислительной среды пользователя по локальной и территориально-распределенной сети. Доступность высокоемких недорогих сменных носителей, например, жестких дисков USB, означает, что потребитель может захватить свою вычислительную среду с собой, куда бы он ни направлялся.

Динамический характер компьютерной среды на базе ВМ требует и более динамичной топологии сети. Виртуальные коммутаторы, виртуальные брандмауэры и оверлейные сети становятся неотъемлемой частью будущего, в котором логическая вычислительная среда отделится от своего физического местоположения.

Виртуализация обеспечивает высокий уровень работоспособности и безопасности благодаря нескольким ключевым возможностям.

Локализация неисправностей. Большинство отказов приложений происходит из-за ошибок ПО. Виртуализация обеспечивает логическое разделение виртуальных разделов, поэтому программный сбой в одном разделе никак не влияет на работу приложения в другом разделе. Логическое разделение также позволяет защищаться от внешних атак, что повышает безопасность консолидированных сред.

Гибкая обработка отказов. Виртуальные разделы можно настроить так, чтобы обеспечить автоматическую обработку отказов для одного или нескольких приложений. Благодаря средствам обеспечения высокой степени работоспособности, заложенным сейчас в платформы на базе процессоров Intel® Itanium® 2 и Intel® Xeon™ MP, требуемый уровень услуг часто можно обеспечить, предусмотрев аварийный раздел на той же платформе, где работает основное приложение. Если требуется еще более высокий уровень работоспособности, аварийный раздел можно разместить на отдельной платформе.

Разные уровни безопасности. Для каждой виртуальной машины можно установить разные настройки безопасности. Это позволит IT-организациям обеспечить высокий уровень контроля за конечными пользователями, а также гибкое распределение административных привилегий.

МВМ имеют мощный потенциал для реструктуризации существующих программных систем в целях повышения уровня защиты, а также облегчают развитие новых подходов к построению безопасных систем. Сегодняшние ОС не обеспечивают надежной изоляции, оставляя машину почти беззащитной. Перемещение механизмов защиты за пределы ВМ (чтобы они выполнялись параллельно с ОС, но были изолированы от нее) позволяет сохранить их функциональные возможности и повысить устойчивость к нападениям.

Размещение средств безопасности за пределами ВМ – привлекательный способ изоляции сети. Доступ к сети предоставляется ВМ после проверки, гарантирующей, что она, с одной стороны, не представляет угрозы, а с другой – неуязвима для нападения. Управление доступом к сети на уровне ВМ превращает виртуальную машину в мощный инструмент борьбы с распространением злонамеренного кода.

Мониторы МВМ особенно интересны в плане управления

многочисленными группами программ с различными уровнями безопасности. Благодаря отделению ПО от оборудования ВМ обеспечивают максимальную гибкость при поиске компромисса между производительностью, обратной совместимостью и степенью защиты. Изоляция программного комплекса в целом упрощает его защиту. В сегодняшних ОС почти невозможно судить о безопасности отдельного приложения, поскольку процессы плохо изолированы от друг друга. Таким образом, безопасность приложения зависит от безопасности всех остальных приложений на машине.

Гибкость управления ресурсами, которую обеспечивают МВМ, может сделать системы более стойкими к нападениям. Возможность быстро тиражировать ВМ и динамически адаптироваться к большим рабочим нагрузкам станет основой мощного инструмента, позволяющего справиться с нарастающими перегрузками из-за внезапного наплыва посетителей на Web-сайты или атаки типа "отказ в обслуживании".

Модель распространения программных продуктов на основе ВМ потребует от поставщиков ПО корректировки лицензионных соглашений. Лицензии на эксплуатацию на конкретном процессоре или физической машине не приживутся в новых условиях, в отличие от лицензий на число пользователей или неограниченных корпоративных лицензий. Пользователи и системные администраторы будут отдавать предпочтение операционным средам, которые легко и без особых затрат распространяются в виде виртуальных машин.

Возрождение МВМ существенно изменило представления разработчиков программных и аппаратных средств о структурировании сложных компьютерных систем и управлении ими. Кроме того, МВМ обеспечивают обратную совместимость при развертывании инновационных решений в области операционных систем, которые позволяют решать современные задачи, сохраняя предыдущие достижения. Эта их способность станет ключевой при решении грядущих компьютерных проблем.

Виртуализация предоставляет также преимущества для сред разработки и тестирования ПО. Различные этапы цикла создания ПО, включая получение рабочей версии, можно выполнять в разных виртуальных разделах одной и той же платформы. Это помогает повысить степень

повысить использование аппаратного обеспечения и упростить управление жизненным циклом. Во многих случаях IT-организации получат возможность тестировать новые и модернизированные решения на имеющихся рабочих платформах, не прерывая производственный процесс. Это не только упрощает миграцию, но также позволяет сократить расходы, устраняя необходимость дублирования вычислительной среды.

Освобождая разработчиков и пользователей от ресурсных ограничений и недостатков интерфейса, виртуальные машины снижают уязвимость системы, повышают мобильность программного обеспечения и эксплуатационную гибкость аппаратной платформы.

Компьютерные системы существуют и продолжают развиваться благодаря тому, что разработаны по законам иерархии и имеют хорошо определенные интерфейсы, отделяющие друг от друга уровни абстракции. Использование таких интерфейсов облегчает независимую разработку аппаратных и программных подсистем силами разных групп специалистов. Абстракции скрывают детали реализации нижнего уровня, уменьшая сложность процесса проектирования.

Подсистемы и компоненты, разработанные по спецификациям разных интерфейсов, не способны взаимодействовать друг с другом. Например, приложения, распространяемые в двоичных кодах, привязаны к определенной ISA и зависят от конкретного интерфейса к операционной системе. Несовместимость интерфейсов может стать сдерживающим фактором, особенно в мире компьютерных сетей, в котором свободное перемещение программ столь же необходимо, как и перемещение данных.

Виртуализация позволяет обойти эту несовместимость. Виртуализация системы или компонента (например, процессора, памяти или устройства ввода/вывода) на конкретном уровне абстракции отображает его интерфейс и видимые ресурсы на интерфейс и ресурсы реальной системы. Следовательно, реальная система выступает в роли другой, виртуальной системы или даже нескольких виртуальных систем.

В отличие от абстракции, виртуализация не всегда нацелена на упрощение или сокрытие деталей. Например, при отображении виртуальных дисков на реальные программные средства виртуализации

используют абстракцию файла как промежуточный шаг. Операции записи на виртуальный диск преобразуются в операцию записи в файл (и следовательно, в операцию записи на реальный диск). Отметим, что в данном случае никакого абстрагирования не происходит – уровень детализации интерфейса виртуального диска (адресации секторов и дорожек) ничем не отличается от уровня детализации реального диска.

[1] По другим сведениям, первый компьютер был создан в Англии в 1943 году для расшифровки кодов немецких подводных лодок.

Основные семейства операционных систем

История семейства операционных систем UNIX/Linux. Генеалогия семейства операционных систем и некоторые известные версии UNIX. Операционные системы фирмы Microsoft. Отличия семейства UNIX/Linux от операционных систем Windows и MS DOS.

2.1. История семейства операционных систем UNIX/Linux

Изучение истории развития результатов творчества всегда интересно. Показательным в этом отношении является пример такого сложного и динамичного технологического объекта, как операционные системы. Подобные программные комплексы создаются годами и включают миллионы строк исходного кода. Они постоянно изменяются, а для успешной конкуренции их разработчикам приходится пополнять свои продукты новыми возможностями. Еще один важный момент из жизни операционных систем заключается в том, что аппаратура, для которой создаются эти программы, постоянно модернизируется и "обрастает" новыми функциями.

Предшественниками современных операционных систем можно назвать системы пакетной обработки, когда выполняемые задания вводились для выполнения поочередно. Сначала это исполнялось вручную, а затем появились средства автоматизации операций. Так возникли предпосылки разработки программных средств управления набором (пакетом) заданий. Важной вехой в этом развитии стал 1964 год, когда IBM анонсировала, а затем и выпустила OS/360. Естественным развитием идей более эффективного использования возможностей вычислительных машин стало появление систем разделения времени. На странице Википедии "Список операционных систем" приводится более чем 200 наименований, и они классифицируются по 9-ти типам. Среди них есть и такие, которые уже не существуют (вернее, уже не поддерживаются разработчиками). Там приводится даже более десятка вымышленных систем, упоминаемых в книгах, фильмах, шутках и т.д. На этом же интернет-ресурсе страница "Хронология операционных систем" начинается с BESYS (Bell System, 1957 год). Но в связи с этим следует упомянуть еще и операционную

систему для ЭВМ типа "мэйнфрейм", разработанную для модели IBM 704 в 1954 году. Ее создатель Джин Амдаль стал основателем компании Amdahl – мощного конкурента IBM на рынке мэйнфреймов [20].

Многие из представленных на странице "Хронология операционных систем" программных продуктов относятся к двум классам: проприетарные и свободные. Первые получили название от английского proprietary – "собственнический", т.е. относятся к программному обеспечению, которое имеет собственника. Такое программное обеспечение находится не в "общественном использовании", а в монопольном.

В этой части монографии анализируются пути развития двух представителей операционных систем: семейства UNIX/Linux и продуктов фирмы Microsoft. Первое из них имеет как проприетарные, так и свободно распространяемые версии. Вторые же являются антагонистом свободных программ.

Семейство операционных систем UNIX уникально по нескольким причинам [2, 14]:

- она является долгожителем и, претерпев многочисленные изменения, "пережила" разнообразную аппаратуру;
- при переносе UNIX на другие аппаратные платформы возникали интересные задачи, решение которых принесло много нового в компьютерные технологии;
- на одной из версий UNIX были реализованы протоколы обмена данными в компьютерных сетях с разной аппаратной платформой, что позволяет считать UNIX предвестницей сегодняшнего Интернета, а также основой для широкого развития локальных сетей;
- авторы ее первых версий создали язык программирования высокого уровня С, который можно назвать (с учетом его последующего совершенствования) самым распространенным среди разработчиков;
- использование этого языка дало возможность принять участие в разработке операционной системы тысячам специалистов;
- появившиеся в семействе UNIX свободно распространяемые операционные системы внесли много нового в представления о

том, как разрабатывать и распространять программы для компьютеров.

Очень большое влияние на все стороны информационных технологий оказала и продолжает оказывать операционная система Linux, первоначально являвшаяся лишь вариантом UNIX. Она завоевала широкую популярность и сегодня перенесена на разные аппаратные платформы, как и ее предшественница. В дальнейшем будем использовать термин "операционные системы семейства UNIX/Linux". Отметим, что часто Linux отделяют от UNIX, сравнивая достижения этой операционной системы со всеми остальными конкретными версиями этого семейства.

Рассмотрение истории и генеалогии UNIX/Linux интересно само по себе, но ее знание необходимо специалистам в области компьютерных технологий. Вот, например, что пишет по этому поводу автор книги, в которую вошли две программы подготовки системных администраторов операционной системы Solaris [7]. "Как системный администратор Вы должны понимать историю операционной системы UNIX – откуда она произошла, как создавалась и чего достигла на сегодняшний день". Но в материале данной книги поднимаются и другие вопросы, что делает ее полезной и другим специалистам. В первую очередь, это – разработчики программного обеспечения.

Имя UNIX возникло позже и имеет интересную историю. А началось с MULTICS (MULTiplexed Information and Computing Service), проекта, ориентированного на распространенные в 60-е и 70-е годы прошлого века компьютеры класса "мейнфрейм" (mainframe). Его авторы первоначально обратились к IBM, но фирма не согласилась на затраты. Разработка MULTICS велась для вычислительной машины GE-645 (General Electric). Для создания операционной системы в середине 60-х годов прошлого века объединились три фирмы: General Electric Company, Massachusetts Institute of Technology (MIT, Массачусетский технологический институт) и American Telephone and Telegraph (AT&T). Последняя была представлена в проекте несколькими сотрудниками подразделения Bell Laboratories. Среди них были Кен Томпсон (Ken Thompson) и Деннис Ритчи (Dennis M. Ritchie). По завершении проекта должна была появиться многозадачная, многопользовательская операционная система [13, 14].

Работа над программным комплексом MULTICS затянулась, и сотрудники Bell Labs вышли из проекта. Но в отличие от других Томпсон продолжил работу по написанию операционной системы в своей компании. Позже к нему присоединился сначала Ритчи, а затем и другие сотрудники отдела. Можно сказать, что UNIX начиналась группой программистов, но основную роль среди разработчиков первых версий играл Кен Томпсон. Сначала, правда, в ближайшем окружении Кена родилось другое название системы – UNICS (Uniplexed Information and Computing System). Оно напоминало об участии в проекте MULTICS, но не ориентировалось на многопользовательскую систему (MULTICS – MULTiplexed, но UNICS – Uniplexed). В скором времени UNICS превратилось в UNIX.

На интернет-ресурсах и в книгах [21, 22] приводится характеристика Кена Томпсона как одного из выдающихся программистов США. По адресу [23] можно найти перевод интересной статьи, в которой Кен Томпсон дает интервью журналу Computer, напечатанное в журнале "Открытые Системы". Персональная страничка Кена Томпсона находится по адресу [24]. На интернет-ресурсе [25] дана характеристика Деннису Ритчи. Персональная страничка Денниса Ритчи находится по адресу [26]. Интересным, на наш взгляд, является оценка вклада двух выдающихся деятелей компьютерного мира по адресу [27].

Вернемся к непосредственному рассмотрению истории создания операционной системы UNIX. Первые ее версии были написаны на языке программирования ассемблер для компьютеров PDP [2, 14]. Она содержала подсистемы управления процессами и файлами, а также небольшой набор утилит.

В эти годы Томпсон работал над транслятором для FORTRAN'а. Но у него получился новый язык программирования B. Последний был интерпретаторным, и, как следствие этого, не очень эффективным. Переработав его, Деннис Ритчи создал язык C, транслирующий исходный текст в машинный код, что повысило эффективность разрабатываемых программ [14]. Этот язык программирования занимает промежуточное положение между языками, близкими к машинным командам и позволяющими разрабатывать "быстрые" программы, и языками программирования высокого уровня (более удобными в использовании).

Приведем информацию из книги [28], описывающую, как появился язык программирования С. "Что это значит на самом деле, что скрывается за этими немного трафаретными словами: язык С разработан американским ученым Деннисом Ритчи? В действительности это означает, что в 1970 г. Деннис Ритчи был изобретен и реализован новый язык С. Ему суждено было большое будущее. Как это произошло? Язык С использует многие важные концепции и конструкции двух предшествовавших ему языков BCPL и B, а также добавляет типы данных и другие свойства".

Язык BCPL разработан в 1967 году Мартином Ричардсом как язык написания компиляторов программного обеспечения операционных систем. Автором языка B был Кен Томпсон – выдающийся программист. Он предусмотрел много возможностей в языке B и использовал его в 1970 году для создания одной из ранних версий операционной системы UNIX в Bell Laboratories на компьютере фирмы DEC PDP-7. Оба упомянутых языка – BCPL и B – были "нетипичными" языками программирования. Так, например, при обработке элемента данных целого или действительного типа значительная часть работы все еще падала на плечи программиста. Язык С приобрел широкую известность как язык разработки операционной системы UNIX. Сегодня фактически все новые операционные системы написаны на С или на С ++.

Возможно, UNIX так и не развилась бы, если бы ей не нашлось реального применения. Но в 1971 году в патентном отделе Bell была установлена именно она. Система стала решать реальные задачи для пользователей, а не ее разработчиков. Она была переписана на более мощный компьютер PDP 11. Со временем UNIX стала распространяться и в другие отделы Bell Labs [14]. Появление первых версий системы сопровождалось выпуском документации с соответствующим номером. Они получили название "редакции" (Edition).

Начиная с 1971 года таких редакций было выпущено 10, а последняя датируется 1989 годом. Семь первых из них были разработаны в Bell Labs. В книге [2] отмечены некоторые важные черты таких версий. В таблице после названия утилит в круглых скобках приводится номер, позволяющий точнее и быстрее найти информацию о ней (номер раздела стандартный для UNIX системы помощи man).

Таблица 2.1. Характеристика редакций UNIX AT&T

№ редакции	Год выпуска	Краткая характеристика
1	1971	Первая версия UNIX, написанная на ассемблере для PDP-11. Включала компилятор B и много известных команд и утилит, в том числе cat(1), chdir(1), chmod(1), cp(1), ed(1), find(1), mail(1), mkdir(1), mkfs(1M), mount(1M), mv(1), rm(1), rmdir(1), w(1), who(1). В основном использовалась как инструментальное средство обработки текстов для патентного отдела
3	1973	В системе появилась команда cc(1), запускавшая компилятор C. Число установленных систем достигло 16
4	1973	Первая система, в которой ядро написано на языке высокого уровня C
6	1975	Первая версия системы, доступная за пределами Bell Labs. Система полностью переписана на языке C. С этого времени начинается появление новых версий, разработанных за пределами Bell Labs, и рост популярности UNIX. В частности, эта версия системы была установлена Томпсоном в Калифорнийском университете в Беркли, и на ее основе вскоре была выпущена первая версия BSD (Berkeley Software Distribution) UNIX
7	1979	Эта версия включала командный интерпретатор Bourne Shell и компилятор C от Кернигана и Ритчи. Ядро было переписано для упрощения переносимости системы на другие платформы. Лицензия на эту версию была куплена фирмой Microsoft, которая разработала на ее базе операционную систему Xenix

Обратим внимание на то, что операционная система с самой первой версии содержит команды обслуживания файловой системы с каталогами (mkdir, rmdir, chdir), многих пользователей (w, who), а также

средства обмена информацией между последними (mail). Утилита mount позволяет включать в систему (монтировать) внешние носители информации. Эти команды "живут" и в современных версиях UNIX. Также обратите внимание, что с 1971 года в системе присутствуют средства работы с текстом. В частности, кроме редактора ed была разработана утилита форматирования текстов roff. Ее аналоги также используются и поныне.

В соответствии с законами США фирма AT&T, подразделением которой была Bell Labs, не имела права продавать программное обеспечение. Но с 1974 года система в виде исходных текстов стала передаваться разным организациям, в том числе университетам. Во время своего академического отпуска 1976 года Томпсон принял участие в проводимых в университете г. Беркли исследованиях по разработке UNIX. В этом ему активно помогали Билл Джой (Bill Joy) и Чак Халей (Chuck Haley) [14].

Джой сформировал собственный дистрибутив UNIX, названный BSD (Berkeley Software Distribution – дистрибутив программного обеспечения Беркли). С его именем связано появление текстового редактора vi, командного интерпретатора с (она выполняла функции оболочки операционной системы, а не компилятора языка программирования), использование виртуальной памяти (позволяющей загружать программы большего размера, чем свободная физическая память). Позже он стал одним из основателей Sun Microsystems, ныне одной из крупнейших компьютерных фирм [7, 15].

Распространяемая в виде исходных текстов UNIX стала быстро завоевывать популярность. Многие компьютерные фирмы начали разрабатывать свои версии этой операционной системы. Например, в 1977 году было уже более 500 работающих экземпляров UNIX [14].

Важным в истории UNIX является 1980 год, когда фирма BBN (Bolt, Beranek и Newman) подписала контракт с DARPA (Department of Advanced Research Projects Agency – Управление перспективных исследований и разработок, являющееся подразделением Министерства обороны США) на разработку и реализацию протоколов TCP/IP в BSD UNIX. Это можно считать началом разработок, явившихся предвестником технологий, которые приняты в Интернете и сегодня. Версия системы,

поддерживающая TCP/IP, также способствовала широкому распространению локальных сетей [14].

Популярность UNIX, поддержка передовых технологий, простота переноса на разные аппаратные платформы привели к тому, что создатели разных вариантов операционной системы начали вести настоящую конкурентную борьбу. В 1988 году фирмы AT&T и Sun объединились для разработки новой системы. В противовес этому несколько крупных фирм (IBM, DEC, HP и другие) основали альтернативный проект, назвав его OSF (Open Software Foundation). В результате появилась ОС с названием OSF/1 [16].

В 1991 году финский студент Линус Торвальдс (Linus Torvalds) написал первую версию операционной системы, названной Linux и распространяемой бесплатно. Тогда она представляла собой вариант UNIX для компьютеров IBM PC, но сегодня перенесена на многие аппаратные платформы. Свою разработку он начал будучи студентом, изучая учебные курсы по программированию на C и UNIX. Он занимался, используя операционную систему MINIX, созданную Эндрю С. Таненбаумом [17]. Такая система была описана в книге "Проектирование и реализация операционных систем". Она представляла собой миниатюрную UNIX-систему для IBM PC. Студента просто захватила концепция UNIX, ее простота и мощь. Свои разработки он обсуждал в Интернете со многими программистами. Можно сказать, что Linux является продуктом программистов всего мира, но руководящую роль в этом играет один человек – Линус Торвальдс.

Приведем по книге [15] абзац, относящийся к Linux. "Операционная система Linux – работа не одного человека. Линус Торвальдс – первоначальный архитектор – ее отец, если хотите. Возможно, самое большое проявление гения Линуса Торвальдса лежит в умении организовать совместную работу. Без оплаты труда, только ради удовольствия, он смог привлечь людей во всем мире к работе над не вполне обычным программным продуктом".

Линус Торвальдс – нетрадиционный человек. Достигнувшая успеха операционная система, как кажется, должна была принести ему хорошие условия жизни. Но он отказался от сотрудничества и с

представителями крупного бизнеса, и, что удивительно, со своими коллегами по разработке свободно распространяемых программ. Он имеет свой взгляд на развитие операционных систем и не часто идет на компромиссы.

Будучи не первой системой подобного класса, Linux быстро завоевала популярность, потеснив коммерческие операционные системы. Сам Торвальдс до сих пор занимается только основой системы – ядром. Доводят ее до пользователей фирмы, выпускающие инсталляторы. Первый имел имя SLS. Но успешно распространяемый и называемый старейшим был создан фирмой Slackware в 1993 году [8]. Версия Linux, поддерживающая графический интерфейс, была разработана в 1992 году. Такой режим стал возможным благодаря усилиям, прежде всего, Ореста Зборовски (Orest Zborowski)[17].

2.2. Генеалогия семейства операционных систем и некоторые известные версии UNIX

Продолжим рассмотрение истории UNIX, описывая, как появлялись различные варианты системы. Следует отметить, что среди них нет "эталона", который можно объявить "чистым" или наибольшим образом впитавшим ее достоинства. Но все они имеют много общего: среду программирования, архитектуру и интерфейс пользователя. Объясняется это достаточно просто – все эти операционные системы "из одного племени". Одни системы впитывали свойства других, как бы являясь их "дочерними" версиями. То общее, что есть у них – это заложенные в ядре возможности и методы их реализации.

Приведенные схемы имеют один вид соединений отдельных версий (элементов схем). Но это не означает, что все такие связи равнозначны. Некоторые версии просто изучались разработчиками на уровне исходных текстов, а другие включили в себя, возможно, без изменений, большие фрагменты исходных текстов программ. Многие из приведенных ниже схем взяты из книги [15].

Для понимания приведенного далее материала важно знать, как получали свои названия версии UNIX на первом этапе. Как было отмечено выше, выпускаемые в AT&T до 1979 года системы

сопровождались созданием документации соответствующего номера. Они назывались 'редакции', а на первой схеме, взятой из упомянутой в предыдущем абзаце книги, называются VERSION 1, …, VERSION 6. Последняя явилась предшественницей трех дочерних: 2.0, BSD и XENIX.

ЗАМЕЧАНИЕ. Многие источники вводят в рассмотрение еще одну версию – VERSION 7, считая, что от нее надо вести историю разделения на три упомянутых или некоторых из них.

AT&T 2.0 развивается и появляющиеся со временем новые версии получили названия System III, System V, а далее SVR2, SVR3, SVR4 (видимо S – System, V – 5, R – Release). Заметим, что версия System IV не была выпущена.

Как отмечалось ранее, название BSD связано с Berkeley Software Distribution (дистрибутив программного обеспечения Беркли). Сокращенные имена версий этого ключевого направления имеют такой вид V.RBSD (видимо V – Version, R – Release).

Фирма Microsoft, купив лицензию UNIX, создает XENIX. Попытка перенести UNIX VERSION 6 AT&T на персональный компьютер была предпринята в 1980 году, т.е. раньше выхода MS DOS [15]. В дальнейшем она была продана фирме SCO (Santa Cruz Operation).

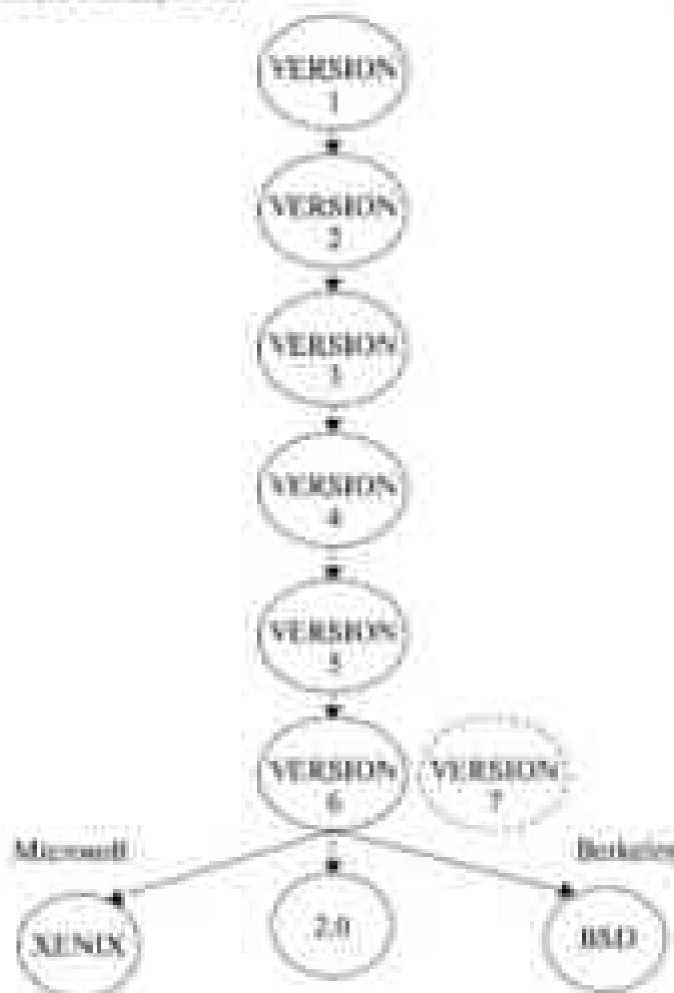


Рис. 2.1. Когда UNIX покинула Bell Labs, она распалась на три ключевых компонента

Следующая схема подтверждает тот факт, что многие варианты UNIX связаны между собой. Разрабатываемые в разных организациях версии объединяются, впитывая все лучшее не только от своих предшественников, но и от систем, разработанных параллельно другими производителями. Купив права на VERSION 6 (по некоторым источникам – VERSION 7), фирма Microsoft создала вариант операционной системы для аппаратной платформы Intel. Параллельно она разрабатывала MS DOS, которая коммерчески оказалась более успешной. Видимо, из-за этого Xenix была продана SCO. К этому

времени в Bell Labs продолжалось совершенствование своих версий. Две фирмы (AT&T Bell Labs и SCO), объединившись, выпустили версию, названную SVR3.2 (рис. 2.2).

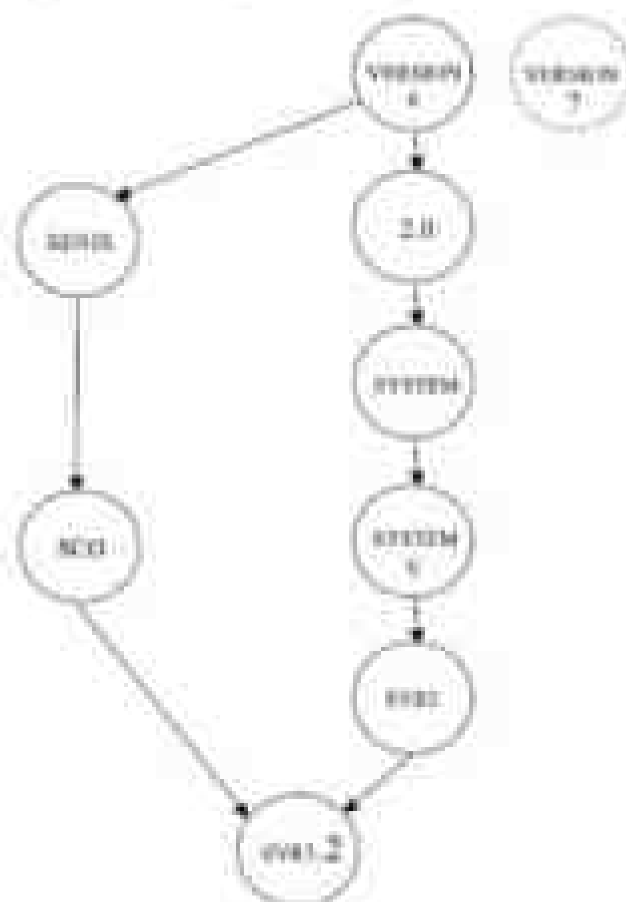


Рис. 2.2. Объединение лучших характеристик SCO Unix с AT&T SVR3 создало версию SVR3.2

Фирма IBM часто удивляет принимаемыми решениями. В свое время она отказалась от участия в проекте, предшествовавшем UNIX. Но со временем сама создает собственный вариант операционной системы AIX. Как видно из схемы, последняя объединяет достоинства SVR3 и 4.3BSD (рис. 2.3).

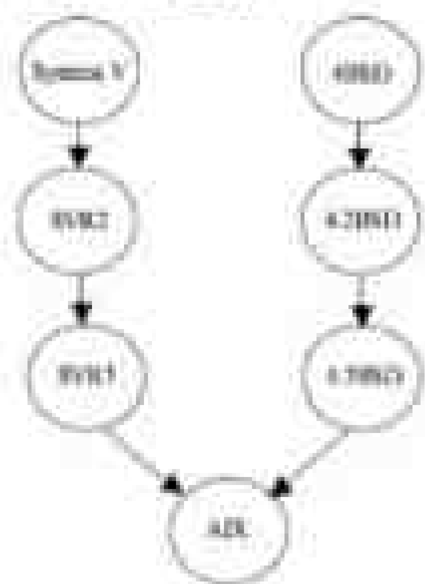


Рис. 2.3. Объединение 4.3BSD с SVR3 привело к созданию операционной системы AIX

Представленная далее схема (рис. 2.4) демонстрирует истоки появления операционной системы SVR4, ставшей одним из стандартов UNIX.

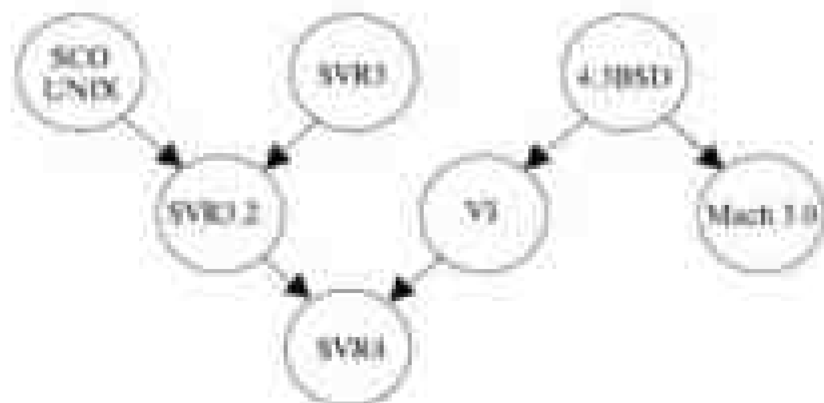


Рис. 2.4. Объединение SVR3.2 и V1 послужило созданию SVR4

На последней схеме отмечено, что после прекращения развития UNIX в университете Беркли ее последняя версия распадается на две ветви.*. . университет практически объявил о прекращении разработки версии

BSD. На сегодняшний день развиваются две фракции – Mach (основа NeXT) и V1" [19]. Также подчеркнем факт появления так называемой микроядерной архитектуры (Mach).

Прежде чем продолжить изложение материала, еще раз заметим, что история UNIX пересказана многократно. При этом некоторые факты в разных источниках противоречат друг другу. Например, в разных источниках по-разному сообщается, на основании какой версии были реализованы варианты BSD и Xenix или в каком году фирма AT&T потеряла права на UNIX. Есть и другие примеры противоречий. Но нам кажется, что все они не могут "смазать" общего представления об интересной и богатой событиями истории UNIX.

Следующая схема (рис. 2.5) демонстрирует этапы появления основных правопреемников исходных текстов программ AT&T UNIX.

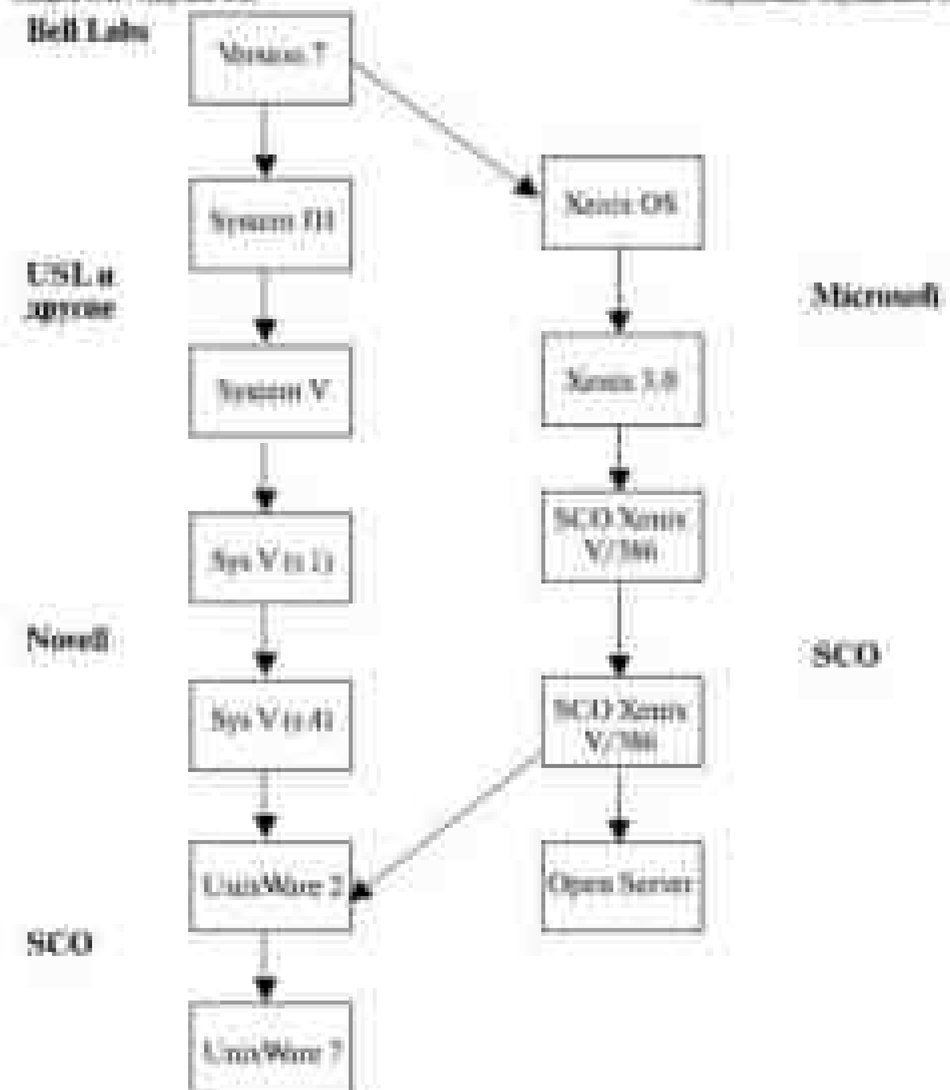


Рис. 2.5. Правопреемники исходных текстов UNIX

Стоявший у истоков создания версий BSD Билл Джой стал соучредителем фирмы Sun, выпускающей UNIX сначала с именем Sun OS, а теперь Solaris (рис. 2.6). В отличие от других фирм Sun гордится, среди прочего, еще и тем, что она одна из немногих крупнейших фирм компьютерной индустрии разрабатывает свою операционную систему для собственной аппаратной платформы (Solaris для процессоров SPARC).

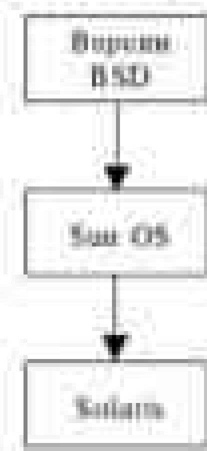


Рис. 2.6. Появление Solaris

Изучив MINIX, Линус Торвальдс пришел к разработке собственной системы, названной Linux (рис. 2.7). Во время разработки последней ее автор активно использовал Интернет для обсуждения возникающих проблем, принимаемых решений и перспектив развития.

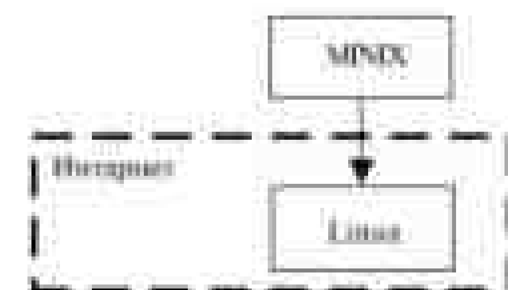


Рис. 2.7. Предшественницей Linux является Minix

На начальных этапах фирма Apple, основанная Стивом Джобсом (Steve Jobs), применяла операционную систему с общим именем System. Эта же фирма выпустила UNIX-подобную ОС AUX для процессоров Motorola. Покинув фирму, Джобс создавал операционную систему NeXTSTEP, а вернувшись в Apple – собственную ОС, названную Mac OS X. Она использовала исходные коды 4.4BSD UNIX. В новой системе применены идеи микроядра Mach 3.0. Естественно, Mac OS X создавалась с учетом опыта предыдущих разработок, в которых принимал участие Джобс (рис. 2.8).

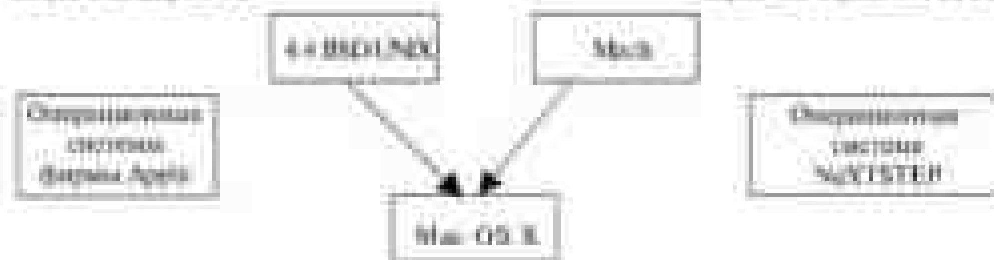


Рис. 2.8. История Mac OS X

Пожалуй, каждая из упомянутых здесь версий имеет не одного непосредственного "предка", а впитала в себя все лучшее из многих разработок, созданных к моменту ее появления. Например, генеалогическое дерево версий UNIX в статье [29] содержит около 60 элементов со множеством соединений. Отметим, что в этой схеме Xenix ведет свое начало от VERSION 7. А вот первая версия 1BSD происходит от VERSION 6, а 3BSD имеет такую "наследственность": сначала VERSION 7 и потом 32V. Видимо, это вносит путаницу в то, какая система является прямым наследником систем с именем BSD.

Приведем часть генеалогического дерева UNIX (рис. 2.9) с другого интернет-ресурса [30]. Отметим, что, на наш взгляд, название 4-го столбца (AT&T/USL) следует изменить, как минимум, на AT&T/USL/Novell.

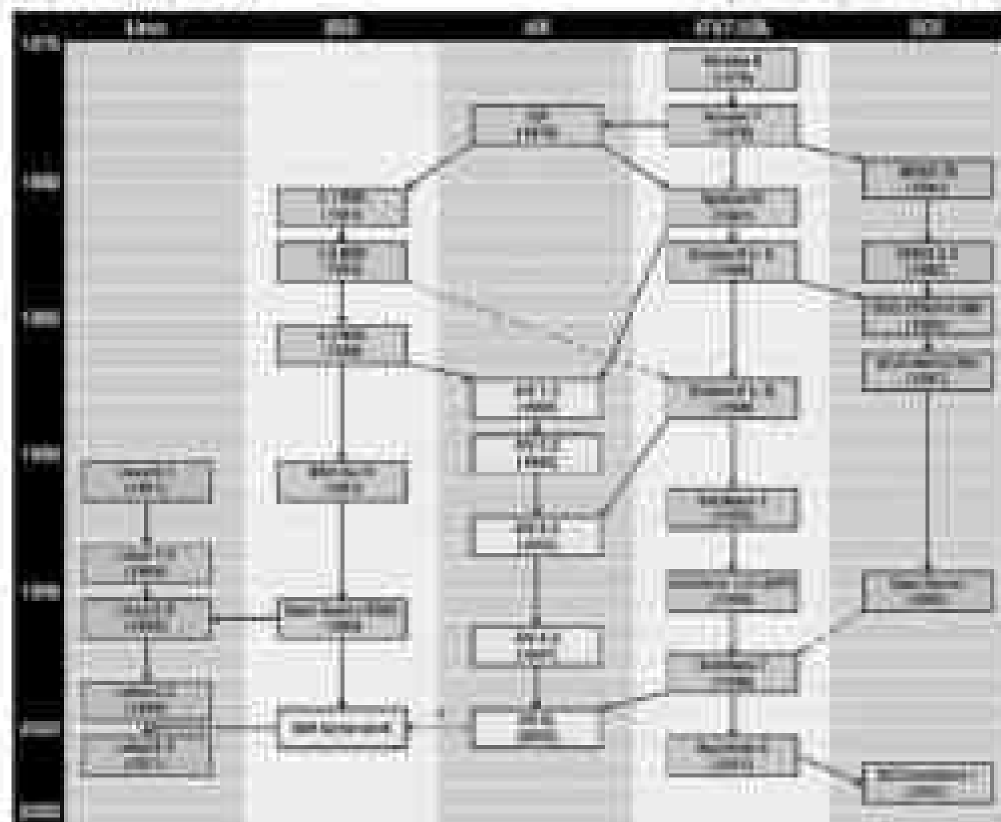


Рис. 2.9. Пример генеалогического дерева версий UNIX

Но самым полным генеалогическим, видимо, является дерево, опубликованное по адресу (ссылка: http://www.levenez.com/unix/ - http://www.levenez.com/unix/. Оно располагается более чем на 20 страницах формата А4, каждый из которых объединяет несколько десятков элементов.

В этом разделе приведем краткую информацию о нескольких известных версиях рассматриваемой операционной системы, предпринимая попытку дать более полный ответ на вопрос "Что представляет собой UNIX?". Решить, какие конкретные системы подпадают под "самые известные", трудно, а перечислить все – невозможно. Далее приводим те из них, которые чаще упоминаются в приведённом в конце пособия списке литературы.

На странице Википедии [21] приводятся такие варианты UNIX-

подобных операционных систем:

AUX	AIX	BSD	Dynix FreeBSD
GNU	GNU/Linux	HP-UX	IRIX
Mac OS X	Minix	NetBSD	NeXTSTEP
OpenBSD	PC-BSD	Plan 9 Plan B	QNX
SCO OpenServer	Solaris System V	Tru64	Xenix

AT&T – самая первая версия операционных систем семейства UNIX. Сначала она создавалась в Bell Labs, а затем в других организациях, образованных AT&T. В этой версии по мере развития опробованы и реализованы многие идеи, используемые в разных программных комплексах и сегодня. Удивляет, как уже с первых шагов в UNIX были выбраны решения, применяемые сегодня во многих операционных системах, и не только этого семейства. UNIX AT&T является преемницей MULTICS. Как сказано в материале С. Кузнецова [22], MULTICS стал "... неудачей с положительными последствиями".

Десять версий этого направления операционных систем создавались около 20 лет. Переданные в разные организации исходные тексты системы положили начало всем другим направлениям и версиям UNIX. Хотя работы над ней начались в Bell Labs AT&T, сейчас эта фирма не имеет к системе прямого отношения, продав права на нее.

Сделаем небольшое отступление о наиболее важных открытиях, сделанных в этой лаборатории. Представленный ниже материал взят из Википедии – свободной энциклопедии [23]. Bell Laboratories (известна также как Bell Labs, прежние названия – AT&T Bell Laboratories, Bell Telephone Laboratories) – бывшая американская корпорация, крупный исследовательский центр в области телекоммуникаций, электронных и компьютерных систем. Основана в 1925 году как исследовательский центр компании AT&T. В настоящее время является исследовательским центром корпорации Alcatel-Lucent. Штаб квартира Bell Labs расположена в Мюррей Хилле (Нью-Джерси, США).

Ниже перечислены наиболее известные разработки этой корпорации.

- В 1933 году Карл Янский обнаружил радиоволны, идущие из

центра галактики, – открытие радиоэлектроники.

- В 1947 году изобретен транзистор. Джон Бардин, Вильям Брэдфорд Шокли и Уолтер Хаузер Браттейн были удостоены за это изобретение нобелевской премии по физике за 1956 год.
- В 1948 году Клод Шеннон опубликовал статью "A Mathematical Theory of Communication", одну из основополагающих работ в теории информации.
- В Bell Labs изобретены фотоэлементы.
- В 1970-х Брайан Керниган, Деннис Ритчи и Кен Томпсон разрабатывали первые версии операционной системы UNIX и язык C.
- В 1980 году разработан первый в мире 32-разрядный микропроцессор.
- В 1980-х Бьярне Страуструп разрабатывал язык C++.
- С конца 1980-х – начала 1990-х разрабатывается перспективная экспериментальная операционная система Plan 9.
- Разработка языка программирования AMPL.

Дадим краткую характеристику широко известных версий Unix-систем.

1. USL, Uniware.

Название этой версии связано с компанией USL, созданной AT&T после того, как она решила, что UNIX отвлекает ее от основного бизнеса. Из десяти версий UNIX AT&T только семь разрабатывались непосредственно в этой организации, а последние связаны с USL. Само название компании менялось, и она даже получала новых членов. Последняя версия является стандартом для операционных систем UNIX и называется System V Release 4.2 [17]. Она впоследствии была приобретена фирмой Novell, известной выпуском сетевой операционной системы для IBM PC с именем NetWare. На основе последней версии системы усилиями Novell и USL создается система UnixWare. Но и эта система поменяла хозяина и далее некоторое время распространялась фирмой SCO.

2. BSD.

Вторая и очень важная ветвь операционных систем UNIX. Имеет такую историю: находясь в творческом отпуске, Кен Томпсон установил UNIX

в Калифорнийском университете в городе Беркли. Заметим, что он закончил его в свое время. Как было сказано выше, два аспиранта, Билл Джой и Чак Халей, заинтересовавшиеся внутренним устройством UNIX, под его руководством стали дорабатывать систему, в результате чего появилась самостоятельная ветвь в семействе UNIX – BSD. Билл Джой (как было сказано выше, в дальнейшем один из соучредителей фирмы Sun Microsystems), разработал для системы много интересных новинок. Уже во второй дистрибутив BSD была добавлена поддержка виртуальной памяти, позволяющая выполнять программы большего размера, чем оперативная память [7].

Важным моментом в развитии этого варианта UNIX является тот факт, что именно на ней (впервые в версии 4.1) был реализован стек протоколов TCP/IP в исследовательской сети ARPANET. Таким образом, последняя приобрела все основные свойства, которыми обладает сегодняшний Интернет. Но реализация этого протокола в BSD сделала все версии сетевыми [12].

Создатели оригинальной BSD UNIX после прекращения деятельности Университета Беркли по разработке программного комплекса выпустили версии для аппаратной платформы Intel, среди которых, пожалуй, наиболее известна FreeBSD, еще существуют OpenBSD и NetBSD. Если Вы интересуетесь историей и версиями xBSD, то обратитесь к источнику [34].

3. Xenix.

Фирма Microsoft известна как разработчик операционной системы для аппаратной платформы IBM PC. В конце 70-х и начале 80-х годов на основе лицензии, купленной у AT&T, была создана система Xenix. Она не получила такого распространения, как думалось при ее создании. После выпуска делались заявления, что именно эта система является стратегическим курсом компании [35]. Но впоследствии она была переделана так, что могла работать на разнообразном оборудовании. Отметим, что разработчики первых версий MS DOS были, по-видимому, знакомы с идеями UNIX, предложив их для условий работы на аппаратуре IBM PC. Исходные тексты Xenix были проданы SCO, которая некоторое время поддерживала их, а затем прекратила. Некоторая часть исходных текстов Xenix перекочевала в программные

комплексы, в частности, SCO Open Server. Заметим, что Microsoft постоянно обращала свой взор на UNIX с разных сторон: как на систему, где возникают новые интересные идеи, как на конкурента, как на возможность на основе этой системы объединиться с другими компаниями для развития нового направления бизнеса.

4. SCO.

Версия с таким названием сегодня не распространяется. Но она была популярной. Компания Santa Cruz Operation (сокращенно SCO) купила у AT&T лицензию на UNIX. В 1988 году три фирмы (SCO, Microsoft и Interactive System) выпустили версию операционной системы для платформы Intel 386. В это время фирма SCO уже купила права на торговую марку UNIX. Сейчас фирма потеряла свою самостоятельность, и права на торговую марку принадлежат The Open Group.

Последние версии системы, поддерживаемые SCO, носили название SCO Open Server. Эта фирма разрабатывала операционные системы с разными названиями. Например, UnixWare она создавала совместно с Novell.

5. Sun OS. Solaris.

Вариант операционной системы с таким названием выпускается фирмой Sun Microsystems. Одним из ее основателей является Билл Джой, начавший разработку операционных систем в Калифорнийском университете после знакомства с Кеном Томпсоном. Solaris работает на разных аппаратных платформах и прежде всего – на SPARC (собственных процессорах фирмы Sun). Но эта операционная система перенесена и на компьютеры IBM PC и PowerPC. До Solaris фирма Sun выпускала UNIX с названием Sun OS. Появление системы с новым именем было связано со стремлением обеспечить стандарты операционных систем на разной аппаратуре.

Среди других достижений фирмы Sun Microsystems отметим разработку Java и в дальнейшем представление компьютерному сообществу его исходных кодов [36].

6. OSF/1.

Появление системы OSF/1 связано со стремлением ведущих компьютерных производителей создать противовес альянсу AT&T и Sun Microsystems. Название OSF является сокращением от Open Software Foundation. В OSF вошли IBM, HP, Digital Equipment Corporation (DEC) и другие [14]. Фирма DEC, ныне уже не существующая, известна, прежде всего, как производитель компьютеров PDP, на которых начинались обе важнейшие версии AT&T и BSD. Фирмы IBM и HP выпускают и поныне успешные версии UNIX. Альянс OSF объединился с X/Open для организации The Open Group, которая сегодня является, видимо, основным хранителем UNIX как таковой.

Видимо, система OSF/1 должна была претендовать на роль третьей важной ветви UNIX (в противовес AT&T и BSD). Трудно сказать, случилось ли это, но вклад в стандарты мира UNIX был, несомненно, сделан. К примеру, принятый альянсом стандарт на графический интерфейс Motif (разработанный в MIT) победил в конкуренции с разработкой Sun Open Look [7].

7. AIX

Собственно история операционных систем начинается с платформы IBM. В 1955 году для вычислительной машины IBM701 была создана развитая операционная система. Сама фирма сделала очень много для развития операционных систем и в дальнейшем. Скажем, к примеру, о легендарных операционных системах для мейнфреймов IBM 360/370, на которых были реализованы многозадачность и многопользовательский терминальный режим.

Сегодня вариант UNIX, разрабатываемый фирмой IBM для собственных аппаратных платформ, имеет название AIX. Оно происходит от Advanced Interactive Executive – улучшенная интерактивная операционная система. Первая версия AIX появилась в 1986 году на основе SVR3.2 AT&T, а последняя имеет название AIX 6. Эта система объединила в себе лучшие черты версий AT&T, BSD и OSF/1.

Справедливости ради отметим, что в последние годы на своей аппаратуре IBM кроме AIX активно поддерживает и Linux [37]. Но сегодня это только фрагмент, а несколько лет назад в категории "Программные продукты" Linux занимала верхнюю строчку.

Приведем несколько фактов из истории этой компании. Пожалуй, рассказывая об истории IBM, надо на первое место поставить перепись населения США в 1880 году, на которых был применен "электрический табулятор" Германа Холлерита, благодаря чему данные переписи были обработаны всего за 3 месяца вместо ожидаемых 24. Он основал фирму, которая в 1911 году объединилась с другими, образовав CTR (Computing Tabulating Recording). Для ее руководства в 1914 году был приглашен Томас Уотсон (Thomas Watson). Компания стала специализироваться на создании больших табуляционных машин и в 1921 году поменяла название на International Business Machines (IBM). Приведем несколько знаменательных для мира компьютерных технологий фактов, связанных с этой компанией (материалы взяты со странички "Голубого гиганта" Википедии).

- В 1943 году началась история компьютеров IBM – был создан "Марк I" весом около 4,5 тонн.
- Но в 1952 году появляется "IBM 701", первый большой компьютер на лампах.
- В 1957 году IBM ввела в обиход язык FORTRAN ("FORmula TRANslation"), применявшийся для научных вычислений и ставший одним из основных источников "проблемы 2000 года".
- В 1959 году появились первые компьютеры IBM на транзисторах.
- В 1964 году было представлено семейство IBM System/360, являющееся первыми универсальными компьютерами, первым спроектированным семейством компьютеров, первыми компьютерами с байтовой адресацией памяти и т.д.
- В 1971 году компания представила гибкий диск, который стал стандартом для хранения данных.
- 1981 год прочно вошел в историю человечества как год появления персонального компьютера "IBM PC".

Далее представлены фрагменты из раздела "Научные и технические разработки", указанного ранее источника Интернета об IBM.

- Фортран (Fortran) – первый реализованный язык программирования высокого уровня. Создан в период с 1954 по 1957 год группой программистов под руководством Джона Бэкуса в IBM.
- Хранение данных на жестком магнитном диске. В 1956 году IBM

инициировала первую в мире систему хранения данных на магнитных дисках (305 RAMAC).

- Фрактал. Фрактальная геометрия позволяет математически описывать различные виды неоднородностей, встречающихся в природе. Впервые введен ученым из исследовательского центра IBM имени Томаса Джона Уотсона Бенуа Мандельбротом в 1967 году в его статье в журнале Science.
- Кремний на изоляторе (КНИ) (англ. Silicon on insulator, SOI) – технология изготовления полупроводниковых приборов, основанная на использовании трехслойной подложки со структурой кремний-диэлектрик-кремний вместо обычно применяемых монолитных кремниевых пластин.
- Магнитная головка на эффекте гигантского магнитного сопротивления. Менее чем через 20 лет после открытия явления ГМС IBM разработала технологию производства магнитных головок с его использованием, что привело к революции в технологиях хранения данных.
- Высокотемпературная сверхпроводимость. Двое ученых IBM Йоханнес Георг Беднорц и Карл Александр Мюллер получили в 1987 году Нобелевскую премию по физике за их открытие в 1986 году сверхпроводимости керамических материалов на основе оксидов меди-лантана-бария.
- DES (Data Encryption Standard) – симметричный алгоритм шифрования, в котором один ключ используется как для шифрования, так и для расшифрования данных. DES разработан IBM и утвержден правительством США в 1977 году как официальный стандарт (FIPS 46-3).
- Реляционные базы данных. Концепция впервые опубликована в 1970 году Эдгаром Франком Коддом из Алмаденского исследовательского центра IBM в работе "A Relational Model of Data for Large Shared Data Banks".
- Суперкомпьютеры.
- DRAM (Dynamic Random Access Memory) – один из видов компьютерной памяти с произвольным доступом (RAM), наиболее широко используемый в качестве ОЗУ современных компьютеров. Эта концепция была впервые предложена Робертом Деннардом в 1966 году в исследовательском центре IBM имени Томаса Джона Уотсона и запатентована в 1968 году.
- Архитектура RISC (англ. Reduced Instruction Set Computing) –

вычислений с сокращенным набором команд. Первые работы были начаты в 1975 году в исследовательском центре IBM имени Томаса Джона Уотсона, прототип был готов в 1980 году.

Отметим и еще один замечательный факт – фирма была основным исполнителем в разработке процессоров Power PC (микропроцессоров RISC-архитектуры, разработанных 1991 Apple, IBM и Motorola).

8. HP-UX.

Второй по величине в мире компьютерный гигант разрабатывал систему с таким именем как серверную систему управляющую вычислительными сетями. Она поддерживается до настоящего времени. Создавалась операционная система в основном для собственной серверной аппаратной платформы HP9000. Ее первая версия родилась на основе VERSION 7 AT&T в 1982 году, а последняя имеет номер 11.

9. IRIX.

Фирма Silicon Graphics известна как производитель оборудования для графических работ на компьютере. С момента создания в начале 80-х годов долгое время фирма занимала лидирующее положение в области машинной графики. Перейдя в сектор подготовки компьютерных эффектов для кино и телевидения, она, можно сказать, участвовала в создании многих известных кинокартин. В выпускаемых компьютерах Silicon Graphics соединены процессоры фирмы MIPS с RISC архитектурой и собственная операционная система IRIX (клон UNIX). Ее последняя версия была выпущена в 2006 году и имеет номер 6.5 [38]. Кроме того, Silicon Graphics разработала библиотеку для моделирования трехмерной графики OpenGL, программный комплекс MAYA. Помимо программных комплексов, фирма разрабатывает и аппаратную часть графических станций.

10. AUX и Mac OS.

Версии с таким названием выпущены фирмой Apple. Ее основатель легендарный Стив Джобс (Steve Jobs), на наш взгляд, вполне заслуживает звания автора первого коммерчески успешного персонального компьютера. Хотя к 1977 году, моменту выпуска

компьютеров Apple, уже существовали такие приборы нескольких фирм, в том числе Atari и IBM, но эту модель можно считать первой наиболее успешной коммерческой моделью персонального компьютера. Далее был выпущен компьютер Lisa (Local Integrated Software Architecture) с реализацией того, что называют GUI. Этот проект был представлен в январе 1983 года. Для фирмы Apple следующим этапом стало появление компьютеров Macintosh, выпускаемых со своей операционной системой. Все перечисленные модели строились на процессорах Motorola 68000, которые по своим возможностям долгое время превосходили IBM PC с графическим интерфейсом Windows. Параллельно с основной операционной системой в Apple создается UNIX-подобная система AUX.

После ухода из Apple Джобс разрабатывал собственную операционную систему NeXTSTEP. Вернувшись в Apple в 2000 году, он сделал своей основной операционной системой Mac OS. Она является преемницей операционных систем, созданных под руководством Стива Джобса, и строится на основе микроядра Mach 3.0 и элементов UNIX BSD 4.4. Система активно развивается, и ее последняя версия имеет номер 10.6.

11. Версии UNIX для IBM PC.

До 1991 года было выпущено несколько версий UNIX для аппаратной платформы IBM PC. Но, пожалуй, только версия Linux смогла составить серьезную конкуренцию продуктам фирмы Microsoft – Windows. Прежде всего, Linux используется на серверах, но постепенно завоевывает рынок программ и для автоматизации деятельности в офисе, для графических работ на персональных компьютерах. Отметим, что кроме этой операционной системы на IBM PC применяется ОС Solaris (с апреля 2010 года принадлежащей Oracle). Последняя была разработана для аппаратной платформы Sun, но была адаптирована для процессоров Intel. Также на такой аппаратной платформе распространены продукты компаний, вышедших из BSD. Они называются Free BSD, OpenBSD, NetBSD.

Операционная система Linux создавалась для персональных компьютеров с процессорами Intel. Но постепенно она "перешла" и на другие аппаратные платформы (SPARC, Alpha, Power PC) [6]. Полный перечень аппаратных платформ, на которых уже работает Linux, можно

найти, например, по адресу в Интернете [39]. В последние годы Linux получает распространение и на карманных персональных компьютерах.

Необычность операционной системы Linux заключается в том, что ее основу до настоящего времени создает Линус Торвальдс. А вот продукт для потребителей разрабатывают многие фирмы, формируя дистрибутивы (инсталляторы). Мы уже отмечали, что первый успешный инсталлятор Slackware был выпущен Патриком Фолькердингом. Сделаем оговорку. Уже в 1992 году появился дистрибутив SLS (Softlanding Linux System) Питера Мак-Дональда, включавший в себя оконную систему X – то есть, теоретически, пригодный для конечного пользователя [40].

Интересную классификацию множества инсталляторов Linux предложил А. Федорчук в своей статье [44], положив в ее основу следующие признаки:

- программы инсталляции;
- средства установки пакетов программ;
- структура файловой системы;
- состав прикладных программ и утилит в инсталляторе.

По данной классификации дистрибутивы делятся на три группы, сходные с RedHat, Debian и Slackware.

Познакомиться с вариантами Linux на разных платформах и списком популярности дистрибутивов можно, например, по адресам [42, 43]. Приведем наиболее популярные дистрибутивы этой операционной системы.

В последние годы среди многих версий операционных систем семейства Linux одной из самых популярных является Ubuntu. Адрес русскоязычного ресурса – ссылка: http://ubuntu.ru - http://ubuntu.ru. На ресурсе Интернета ссылка: http://www.distrowatch.com - http://www.distrowatch.com, одном из источников, учитывающем пользовательскую популярность разновидностей Linux, дистрибутив Ubuntu занимает первое место. Его варианты выпускаются каждые 6 месяцев. Можно отослать заявку и дистрибутив будет доставлен по почте. Также можно скачать дистрибутив с бесплатных ресурсов Интернета.

Финансирует развитие Ubuntu Марк Ричард Шаттлворт (Mark Richard Shuttleworth) – миллионер и второй космический турист, родившийся в ЮАР.

Самый древний дистрибутив Slackware – до сих пор в строю, хотя на сегодняшний день не входит в десятку самых популярных. На его основе созданы другие дистрибутивы.

Red Hat долгое время была одной из наиболее распространенной системой Linux. В рамках дистрибутивов американской компании опробованы многие технологии. Но с 2003 года фирма Red Hat сменила политику выпуска дистрибутивов. Свободно распространяемой версией стала Fedora, а система Red Hat Enterprise Linux является корпоративным решением, который продается.

SUSE – этот дистрибутив имеет корни от самого первого дистрибутива SLS, не имевшего широкого распространения. В свое время он был очень распространен в Европе. Но в 2003 году этот дистрибутив был куплен американской фирмой Novell.

Дистрибутив с именем Debian находится в списке пионеров. Его создание началось в 1993 году. На его основе строились многие дистрибутивы, один из них – Ubuntu.

Отдельно скажем о русифицированных дистрибьюторах. Это Fedora (фирмы Red Hat ранее выпускающую версию с названием Red Hat Cyrillic Edition), Suse и Mandriva (долгое время имевший имя фирмы Mandrake), но как наиболее распространенные российские разработки следует отметить ASP Linux и Alt Linux.

2.3. Операционные системы фирмы Microsoft

Вначале дадим характеристику Microsoft, содержащуюся на странице Википедии об этой фирме.

Microsoft (Microsoft Corporation, читается "майкрософт", NASDAQ: MSFT) – крупнейшая (прибыль за 2008 год – 17.7 млрд долл. при обороте в 60,4 млрд долл.) транснациональная компания по производству программного обеспечения для различного рода вычислительной

техники — персональных компьютеров, игровых приставок, КПК, мобильных телефонов и прочего, разработчик наиболее широко распространенной на данный момент в мире программной платформы [4] – семейства операционных систем Windows. Подразделение компании также производит некоторые аксессуары для персональных компьютеров (клавиатуры, мыши и т. д.). Продукты Microsoft продаются более чем в 80-ти странах мира, программы переведены более чем на 45 языков.

Фирма Microsoft была основана двумя студентами: Биллом Гейтсом и Полом Алленом в 1975 году. Они прочитали статью о персональном компьютере Altair 8800 и разработали для него интерпретатор языка Basic. Его приобрел производитель аппаратуры. С этого началась компания, а ее учредители вместо учебы занялись бизнесом и значительно преуспели в этом.

История операционных систем для персональных компьютеров IBM PC начинается в 1981 году, когда на этом оборудовании была установлена MS DOS 1.0. Правда, эта операционная система не вполне может считаться разработанной в Microsoft. Ее прототип был разработан вне фирмы Microsoft в Seattle Computer Products и дополнен интерпретатором для Бейсика Билла Гейтса [17].

Первая операционная система Microsoft была построена после покупки лицензии у AT&T на UNIX. Так появилась операционная система Xenix, которую фирма разрабатывала несколько лет, но далее решила избавиться от нее, отдав предпочтение MS DOS.

Фирма Microsoft разработала и выпустила несколько десятков операционных систем для разной аппаратуры, но в основном для персональных компьютеров IBM PC. Их можно разделить на такие группы:

1. MS DOS. Серия операционных систем, поддерживающих только командную строку как интерфейс пользователя. Выпущены версии от 1.0 (1981 год) до 6.22 (1994 год). Многие компании (в числе которых IBM, DEC и даже МФТИ) создавали свои версии этой системы.

2 . Windows 1, 2, 3 и 3.11. Надстройки над операционными системами MS DOS, обеспечивающие режим графического интерфейса

пользователя. Они не были полноценными операционными системами, а являлись оболочками, обеспечивающими стандартизацию использования аппаратного обеспечения и единообразие интерфейсов для пользовательских программ. Первая из версий появилась в 1985 году, а последняя – в 1995 году.

Следует заметить, что имелся предшественник Windows – графическая оболочка компании Visi Corp под названием Visi On [48]. Приведем пример интерфейса этой оболочки 1983 года (рис. 2.10).

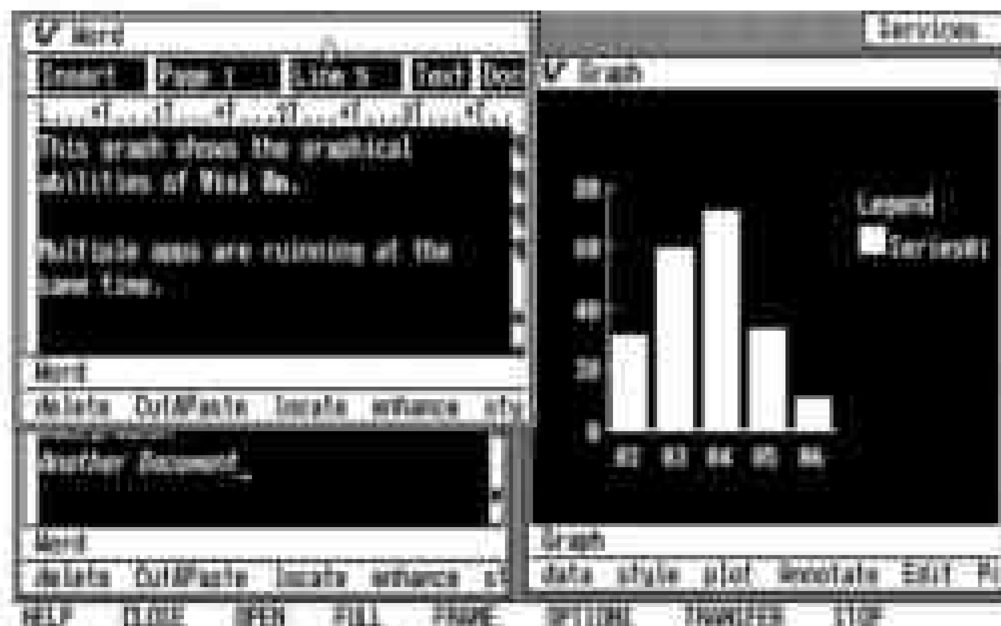


Рис. 2.10. Пример интерфейса графической оболочки Visi On

А вот как выглядел для пользователей экран среды Windows 1.0, выпущенной два года спустя в 1985 году (рис. 2.11).

Рис. 2.11. Пример интерфейса графической оболочки Windows1

3. Windows 9X. Эта серия операционных систем представлена такими версиями: Windows 95, Windows 98 и Windows Me. Они были предназначены для работы пользователей на персональных компьютерах IBM PC. Графический интерфейс этих систем оказал большое влияние на стандарты работы пользователей с персональным компьютером. Вид экрана пользователя приводится на рис. 2.12.

4. Windows NT. Сокращение в NT ее названии образовано от New Technology. Первая ее версия, созданная в 1993 году должна была вытеснить MS DOS, чего не произошло. Следующие версии должны были потеснить на рынке Windows 95, что случилось только в начале 2000 годов. Создавались варианты этой системы как для работы пользователя на локальном компьютере, так и для управления локальной сетью. Версии этого направления до определенного времени назывались NT, а с 2000 года получали разные имена: NT 5.0 – Windows 2000, NT 5.2 – Windows 2003, NT 6.0 – Windows Vista и Windows 2008, NT 6.1 – Windows 7.

5. Windows CE. Эти операционные системы начали разрабатываться в 1996 году. В настоящий момент они созданы для разнообразных мобильных устройств. Последняя версия в этой линейке – Windows mobile 6.

Можно отметить, что фирма Microsoft является монополистом на производство программного обеспечения для персональных компьютеров. Под ее эгидой создается самое разнообразное ПО – операционные системы, офисные приложения, средства разработки, системы управления фирмами и предприятиями (корпоративные системы). Попытки завоевать другие аппаратные платформы не увенчались успехом (кроме мобильных и переносных аппаратов). Есть определенные достижения у фирмы и на рынке суперкомпьютеров.

Рис. 2.12. Интерфейс операционной системы Windows95

В последнее время компания стала ответчицей в исках антимонопольной комиссии ЕС и конкурентов. Приведем два примера (Википедия).

В марте 2004 года Еврокомиссия признала американскую компанию виновной в использовании своего доминирующего положения на европейском рынке программного обеспечения и наложила на компанию штраф в размере 497 млн евро, потребовав от Microsoft

предоставить сторонним разработчикам информацию о своих продуктах, чтобы они смогли беспрепятственно выпускать совместимые программы. После того, как Microsoft не подчинилась данному решению, в июле 2006 года она вновь была оштрафована – на этот раз на 280,5 млн евро, после чего исполнила решение Еврокомиссии.

13 декабря 2007 года норвежская компания Opera Software ASA, разработчик веб-браузера Opera, заявила, что подала жалобу на Microsoft в Еврокомиссию. В жалобе Opera Software просит Microsoft дать пользователям "по-настоящему выбирать" браузер, поставляя с Windows браузеры конкурентов или отделив Internet Explorer от основной поставки. Кроме того, компания требует встроить поддержку открытых веб-стандартов в Internet Explorer.

2.4. Отличия семейства UNIX/Linux от операционных систем Windows и MS DOS

В этой книге мы ориентировались, в основном, на читателей, которые до настоящего времени использовали только операционные системы Windows. Хотим отметить, что до появления в 1981 году MS DOS система UNIX уже прошла значительный путь своей истории. Был момент, когда сама Microsoft стояла перед выбором: разрабатывать один из вариантов UNIX для IBM PC или продолжить собственную систему. Даже была куплена соответствующая лицензия и выпущен вариант UNIX – XENIX. Но потом выбор остался все же за MS DOS. В работе UNIX и MS DOS, а теперь UNIX в графическом режиме и современных версий Windows, есть много общего, иногда даже в мелочах. Сделав это вступление, приведем несколько пунктов, где семейство ОС UNIX/Linux существенно отличается от операционных систем фирмы Microsoft. Далее везде, где встречается термин "система", подразумевается семейство UNIX/Linux.

1. Исходные тексты компонентов системы доступны для просмотра и модификации. Чаще всего они располагаются в подкаталоге с именем source, который подчинен каталогу /usr.
2. Модифицировать систему можно перекомпилировав ядро – основу системы, которая непрерывно развивается и настраивается на конфигурацию вычислительной установки.

3. Существует несколько уровней настройки параметров работы системы:
   - работа с утилитами, в том числе в режиме графического интерфейса;
   - корректировка файлов конфигурации;
   - внесение изменений в исходные тексты и их дальнейшая перекомпиляция.
4. Первоначально загружается командный режим, а графический интерфейс требует дополнительного вызова. Последний имеет несколько методов реализации.
5. В инсталляторы системы Linux включается полный набор программного обеспечения, необходимый для работы как в качестве офисного или домашнего компьютера, так и сервера.
6. Интересной особенностью работы системы является возможность одновременной регистрации нескольких пользователей на виртуальных терминалах.
7. В системе существует множество оболочек (аналог командного интерпретатора command.com в MS DOS). В процессе работы можно получить их полный список (команда chsh – l) и выбрать любую (команда chsh).
8. Помимо работы с основной файловой системой, можно получать доступ к информации, подготовленной в других операционных системах.
9. Файловая система Linux на жестком диске может располагаться на нескольких разделах диска, а для области подкачки всегда выделяется отдельный дисковый раздел с типом файловой системы, отличной от основной. Также в отдельных разделах диска можно разместить следующую информацию (приводится список, доступный в ASPLinux [14]):
   - данные о загрузке (/boot);
   - области диска, куда заносится постоянно изменяемая системная информация, например, системные файлы, почтовые сообщения, (/var);
   - области диска выделяемые для работы обычным пользователям (/home);
   - информация предназначенная для всех пользователей (/usr).
10. Доступ к данным, получаемым с разнообразного оборудования, осуществляется не в одной из вершин верхнего уровня файловой системы, а в одной из вершин, подчиненных единственному

корня иерархической файловой системы (от имени /).

Стандарты и лицензии на программное обеспечение

Стандарты семейства UNIX. Стандарты языка программирования С. System V Interface Definition (SVID). Комитеты POSIX. X/Open, OSF и Open Group. *Лицензии на программное обеспечение и документацию.*

3.1. Стандарты семейства UNIX

Причиной появления стандартов на операционную систему UNIX стало то, что она была перенесена на многие аппаратные платформы. Ее первые версии работали на аппаратуре PDP, но в 1976 и 1978 годах система была перенесена на Interdata и VAX. С 1977 по 1981 годы оформились две конкурирующие ветви: UNIX AT&T и BSD. Наверное, цели разработки стандартов были разными. Одна из них – узаконить главенство своей версии, а другая – обеспечить переносимость системы и прикладных программ между различными аппаратными платформами. В связи с этим говорят и о мобильности программ. Такие свойства имеют отношение как к исходным текстам программ, так и исполняемым программам.

Дальнейший материал приводится в хронологическом порядке появления стандартов.

Стандарты языка программирования С

Этот стандарт не относится непосредственно к UNIX. Но поскольку С является базовым как для этого семейства, так и других ОС, упомянем о стандарте этого языка программирования. Начало ему было положено выходом в 1978 году первой редакции книги Б.Кернигана и Д.Ритчи. Этот стандарт часто называют K&R. Программисты, авторы этого труда, работали над UNIX вместе с Кеном Томпсоном. При этом первый из них предложил название системы, а второй изобрел этот язык программирования. Соответствующий текст доступен в Интернете [45].

Однако промышленный стандарт языка программирования С был выпущен в 1989 году ANSI и имел имя X3.159 - 1989. Вот что написано про этот стандарт [46]:

"Стандарт был принят для улучшения переносимости написанных на языке Си программ между различными типами ОС. Таким образом, в стандарт, кроме синтаксиса и семантики языка Си, вошли рекомендации по содержанию стандартной библиотеки. О наличии поддержки стандарта ANSI C говорит предопределенное символьное имя „STDC".

В 1988 году на основе этого стандарта языка программирования была выпущена вторая редакция книги Кернигана и Ритчи о С. Заметим, что фирмы, производящие программные продукты для разработки программ на языке С, могут формировать свой состав библиотек и даже несколько расширять состав других средств языка.

System V Interface Definition (SVID)

Другое направление развития стандартов UNIX связано с тем, что не только энтузиасты задумывались о создании "эталонов". Основные разработчики системы с появлением многих "вариантов" решили издавать собственные документы. Так появляются стандарты, выпускаемые USG, организацией, разрабатывающей документацию версий UNIX AT&T с того момента, когда для создания операционной системы была образована эта дочерняя компания. Первый документ появился в 1984 году на основе SVR2. Он имел название SVID (System V Interface Definition). Четырехтомное описание было выпущено после выхода в свет версии SVR4. Эти стандарты дополнялись набором тестовых программ SVVS (System V Verification Suite). Основное назначение этих средств состояло в том, чтобы разработчики имели возможность судить, может ли их система претендовать на имя System V [14].

Отметим, что положение дел со стандартом SVID в чем-то сходно со стандартом языка программирования С. Изданная авторами этого языка программирования книга является одним из эталонов, но не единственным. Выпущенный позже стандарт С является результатом коллективного труда, прошел этап обсуждения широкой общественности и, видимо, может претендовать на ведущую роль в списке стандартов. Так и SVVS является набором тестов, позволяющих судить, достойна ли система соответствовать имени System V, только одной из версий UNIX. При этом не учитывается весь опыт разработки

операционных систем от разных производителей.

Комитеты POSIX

Работа по оформлению стандартов UNIX началась группой энтузиастов в 1980 году. Была сформулирована цель - формально определить услуги, которые операционные системы обеспечивают приложениям. Такой стандарт программного интерфейса стал основой документа POSIX (Portable Operating System Interface for Computing Environment – переносимый интерфейс операционной системы для вычислительной среды) [14]. Первая рабочая группа POSIX была образована в 1985 году на основе UNIX-ориентированного комитета по стандартизации /usr/group, также называемой UniForum [47]. Название POSIX было предложено руководителем GNU Ричардом Столманом.

Ранние версии POSIX определяют множество системных сервисов, необходимых для функционирования прикладных программ, которые описаны в рамках интерфейса, специфицированного для языка С (интерфейс системных вызовов). Заложенные в нем идеи использовались комитетом ANSI (American National Standards Institute) при создании стандарта языка С, упомянутого ранее. Исходный состав функций, закладываемый в первые версии, опирался на UNIX AT&T (версия SVR4 [48]). Но в дальнейшем происходит отрыв спецификаций стандартов POSIX от этой конкретной ОС. Подход к организации системы на основе множества базовых системных функций был применен не только в UNIX (например, WinAPI фирмы Microsoft).

В 1988 году был опубликован стандарт 1003.1 – 1988, определяющий API (Application Programming Interface, программный интерфейс приложений). Через два года был принят новый вариант стандарта IEEE 1003.1 – 1990. В нем были определены общие правила программного интерфейса, как для системных вызовов, так и для библиотечных функций. Далее утверждаются дополнения к нему, определяющие сервисы для систем реального времени, "нитей" POSIX и др. Важным является стандарт POSIX 1003.2 – 1992 – определение командного интерпретатора и утилит.

Имеется перевод [1] этих двух групп документов, которые получили

такие названия: POSIX.1 (интерфейс прикладных программ) и POSIX.2 (командный интерпретатор и утилиты – интерфейс пользователя). В упомянутом переводе содержатся три главы: основные понятия, системные услуги и утилиты. Глава "Системные услуги" разделена на несколько частей, каждая из которых группирует сходные по функциям услуги. Например, в одном из разделов "Базовый ввод/вывод" седьмая часть, посвященная операциям с каталогами, описывает три функции (`opendir`, `readdir` и `closedir`). Они определяются в четырех пунктах: "Синтаксис", "Описание", "Возвращаемое значение" и "Ошибки".

Для тех, кто знаком с алгоритмическим языком программирования С, приведем пример фрагментов описания. Фактически такое описание дает представление о том, как специфицируется "Интерфейс системных вызовов". В пункте "Синтаксис" про функцию `readdir` приведены такие строки:

```
#include <sys/types.h>
#include <dirent.h>
struct dirent *readdir(DIR *dirp);
```

Второй пункт ("Описание") содержит следующий текст:

"Типы и структуры данных, используемые в определениях с каталогами, определяются в файле `dirent.h`. Внутренний состав каталогов определяется реализацией. При чтении с помощью функции `readdir` формируется объект типа `struct dirent`, содержащий в качестве поля символьный массив `d_name`, в котором находится завершенное символом NUL локальное имя файла.

`Readdir` читает текущий элемент каталога и устанавливает указатель позиции на следующий элемент. Открытый каталог задается указателем `dirp`. Элемент, содержащий пустые имена, пропускается".

А вот что приводится в пункте "Возвращаемое значение":

"`Readdir` при успешном завершении возвращает указатель на объект типа `struct dirent`, содержащий прочитанный элемент каталога. Прочитанный элемент может записываться в статическую память и

перекрываются очередным таким вызовом, примененным к тому же открытому каталогу. Вызов readdir для различных открытых каталогов не перекрывает читаемую информацию. В случае ошибки или достижения конца файла возвращается пустой указатель".

В пункте "Ошибки стандарта" указано следующее:

"readdir и closedir обнаруживают ошибку [EBADF] dirp не являются указателем на открытый каталог".

Этот пример показывает, как описываются представляемые приложением услуги. Требования к операционной системе (реализации) заключаются в том, что она "...должна поддерживать все обязательные служебные программы, функции, заголовочные файлы с обеспечением специфицированного в стандарте поведения. Константа _POSIX_VERSION имеет значение 200112L [49]".

В мире компьютерных технологий существует такое словосочетание: "программирование POSIX". Этому можно научиться, используя различные руководства по системному программированию UNIX и операционным системам (например, [21]). Есть отдельная книга с таким названием [2]. Заметим, что в предисловии к этой книге сказано, что она описывает "... стандарт втройне ...", так как она опирается на последнюю версию POSIX 2003 года, в основе которой три стандарта: IEEE Std 1003.1, технический стандарт Open Group и ISO/IEC 9945.

Как же проверить соответствие конкретной системы стандарту POSIX? Формализация такого вопроса не так проста, как кажется на первый взгляд. В современных версиях предлагается 4 вида соответствия (четыре семантических значения слова "соответствие": полное, международное, национальное, расширенное).

В рассматриваемых документах приводятся списки двух видов интерфейсных средств: обязательные (по возможности предполагается его компактность) и факультативные. Последние должны либо обрабатываться предписанным образом, либо возвращать фиксированное значение кода ENOSYS, означающего, что функция не реализована.

Отметим, что набор документов POSIX изменяется уже много лет. Но

разработчики новых версий всегда стараются максимально сохранить преемственность с предыдущими версиями, В более свежих редакциях может появиться что-то новое. Например, в документе 2004 года были объединены четыре части [50]:

- Base Definitions volume (XBD) – определения терминов, концепций и интерфейсов, общих для всех томов данного стандарта;
- System Interfaces volume (XSH) – интерфейсы системного уровня и их привязка к языку Си, где описываются обязательные интерфейсы между прикладными программами и операционной системой, в частности – спецификации системных вызовов;
- Shell and Utilities volume (XCU) – определение стандартных интерфейсов командного интерпретатора (т.н. POSIX-shell), а также базовой функциональности Unix-утилит;
- Rationale (Informative) volume (XRAT) – дополнительная, в том числе историческая, информация о стандарте.

Как и первые редакции, документ в своей основной части описывает группы представляемых услуг. Каждый элемент там описан в следующих пунктах: NAME (Имя), SINOPSIS (Синтаксис), DESCRIPTION (Описание), RETURN VALUE (Возвращаемое значение), ERRORS (Ошибки) и в заключении EXAMPLE (Примеры).

Современные версии стандарта определяют требования как к операционной системе, так и к прикладным программам. Приведем пример [51].

Функция readdir() должна возвращать указатель на структуру, относящуюся к очередному элементу каталога. Возвращаются ли элементы каталога с именами "точка" и "точка-точка", стандартом не специфицировано. В этом примере возможны четыре исхода, а требование к прикладной программе состоит в том, что она должна быть рассчитана на любой из них.

И в заключение приведем отрывок из курса лекций Сухомлинова ("ВВЕДЕНИЕ В АНАЛИЗ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ", Сухомлинов В.А. Часть V. Методология и система стандартов POSIX OSE), посвященным области применимости стандартов [52].

"Область применимости стандартов POSIX OSE (Open System Environment) – обеспечение следующих возможностей (называемых еще свойствами открытости) для разрабатываемых информационных систем:

- Переносимость приложений на уровне исходных текстов (Application Portability at the Source Code Level), т.е. предоставление возможности переноса программ и данных, представленных на исходных текстах языков программирования, с одной платформы на другую.
- Системная интероперабельность (System Interoperability), т.е. поддержка взаимосвязанности между системами.
- Переносимость пользователей (User Portability), т.е. обеспечение возможности для пользователей работать на различных платформах без переобучения.
- Адаптируемость к новым стандартам (Accommodation of Standards), связанным с достижением целей открытости систем.
- Адаптируемость к новым информационным технологиям (Accommodation of new System Technology) на основе универсальности классификационной структуры сервисов и независимости модели от механизмов реализации.
- Масштабируемость прикладных платформ (Application Platform Scalability), отражающая возможность переноса и повторного использования прикладного программного обеспечения применительно к разным типам и конфигурациям прикладных платформ.
- Масштабируемость распределенных систем (Distributed System Scalability), отражающая возможность функционирования прикладного программного обеспечения независимо от развития топологии и ресурсов распределенных систем.
- Прозрачность реализаций (Implementation Transparency), т.е. сокрытие от пользователей за интерфейсами систем особенностей их реализации.
- Системность и точность спецификаций функциональных требований пользователей (User Functional Requirements), что обеспечивает полноту и ясность определения потребностей пользователей, в том числе и определения состава применяемых стандартов."

Это позволяет решать следующие задачи:

- интеграция информационных систем из компонент различных изготовителей;
- эффективность реализаций и разработок, благодаря точности спецификаций и соответствию стандартным решениям, отражающим передовой научно-технический уровень;
- эффективность переноса прикладного программного обеспечения, благодаря использованию стандартизованных интерфейсов и прозрачности механизмов реализации сервисов систем.

Также в стандартах формально определяются такие важные понятия операционных систем: пользователь; файл; процесс; терминал; хост; узел сети; время; языково-культурная среда. Там не приводится формулировки такого определения, но вводятся применяемые к ним операции и присущие им атрибуты.

Всего в списке стандартов POSIX более трех десятков элементов. Их имена традиционно начинаются буквой "P", после которой расположено четырехзначное число с дополнительными символами. Существуют также групповые имена стандартов POSIX1, POSIX2 и т.д. Например, POSIX1 связан со стандартами на базовые интерфейсы ОС (P1003.1x, где вместо x либо пусто, либо символы от a до g; таким образом, в этой группе 7 документов), а POSIX3 – методы тестирования (два документа – P2003 и P2003n).

X/Open, OSF и Open Group

Основанная в 1984 году рядом компьютерных фирм организация X/Open своей основной задачей ставила согласование и утверждение для разных версий UNIX стандартов общего программного интерфейса и программной среды для приложений. В 1988 году появился документ XPG3 (X/Open Portability Guide3). Он включал POSIX 1003.1 – 1988 и стандарт на графическую систему X Window System (MIT). Этот документ был развит, включая последние идеи UNIX версии BSD и System V. Он получил название XPG4.2 [14].

X/Open объединила свои усилия с Open Software Foundation, создав The

Open Group, которая до настоящего времени работает над идеологией открытых систем. С момента создания ей принадлежит торговая марка UNIX. Фирма активно работает (вместе с IEEE, ISO и IEC) над последними стандартами операционных систем.

Заметим, что OSF была образована рядом организаций в ответ на объединение Sun Microsystems и AT&T для разработки операционной системы, претендующей на универсальность. Сегодня организация The Open Group является главным держателем стандартов UNIX. Она размещает на своем сайте много самой разнообразной информации, начиная от истории описания "What is UNIX" ("Что такое UNIX") и заканчивая серией стандартов Single Unix, Unix95, Unix98, Unix03, ISO/IEC 9945:2003, а также UNIX System API Table.

В этот неправительственный консорциум входит около 200 членов. В их числе правительственные организации, учебные заведения и представители бизнеса разных стран. Приведем (в алфавитном порядке) несколько представителей компьютерного бизнеса членов The Open Group:

AT&T	Hewlett-Packard	IBM
Intel	NEC Corporation	Oracle
Red Hat (R), Inc.	Sun Microsystems	SUSE LINUX Products GmbH

В заключение этого раздела заметим, что существует еще много менее значимых стандартов на программное обеспечение. Один из них, Linux Standard Base (LSB), создавался Free Standards Group. Но последняя объединилась с Open Source Development Labs (OSDL) и образовала новую организацию The Linux Foundation. Назовем и еще один документ – лицензию BSD (программная лицензия университета Беркли, применяемая для распространения UNIX-подобных операционных систем BSD).

3.2. Лицензии на программное обеспечение и документацию

С появлением Linux и подобных ей систем распространяемые свободно программы стали вытеснять коммерческие продукты. Для разных

платформ существует много бесплатных программ семейства UNIX/Linux. Большая часть из них разрабатывалась в соответствии со специальной лицензией GPL (General Public License), которая была издана в рамках проекта GNU, начатого в 1984 году Ричардом Столманом (Richard Stallman) [11].

Информацию о Ричарде Столмене как одном из плеяды "выдающихся программистов мира" можно посмотреть в Интернете [52]. Ричард Столмен окончил Гарвардский университет по специальности "физика". Затем поступил на работу в Массачусетский технологический институт, где участвовал в нескольких проектах по разработке программного обеспечения. К примеру, он написал включенный во многие версии UNIX текстовый редактор emacs. С 1984 года он работает над проектом, первоначальной целью которого было создание на основе идей UNIX свободно распространяемой (бесплатной) операционной системы. Для ее разработки были нужны другие программные средства, например, транслятор с одного из языков программирования и редактор текстов. Но они также должны были быть бесплатными, иначе их авторы могут заявить свои права на часть созданной операционной системы.

Мысли Столмена были перенаправлены на создание новых методов разработки программного обеспечения. Для этого была создана лицензия GPL, в рамках которой разрабатываются свободно распространяемые программы. Для развития такого направления основывается FSF (Free Software Foundation), который возглавил Столмен. Его идеи заключаются в том, что программы обязательно должны иметь открытые исходные тексты. Любой программист может воспользоваться фрагментом чужой программы, но открыв исходный текст, созданный им самим. Кроме изменений, связанных с возможностью использовать чужие фрагменты, такой метод разработки программ улучшает и процедуру тестирования программ.

Удачные алгоритмы применяются многими программистами и подвергаются многократной и разнообразной проверке. Вообще Столмен сравнивал такой способ разработки программ с обменом кулинарными рецептами. Заметим, что разрабатываемые в соответствии с GPL программы не обязательно должны быть бесплатными. Можно включить программы других авторов как часть своего продукта и продавать последний. Конечно, при этом надо

указать всех авторов всех частей проекта.

Что же такое свобода программного обеспечения по Столлману [1]?

1. Разрешается запускать программу и использовать ее по назначению в любых целях.
2. Разрешается изучать устройство программы, то, как она создана. При этом можно и даже необходимо использовать ее свободно предоставляемые исходники.
3. Разрешается копировать программу в любых количествах и распространять бесплатно всем, кому она нужна.
4. Разрешается изменять код программы, изменять ее в соответствии со своими представлениями и распространять как на коммерческой основе, так и на некоммерческой (платно или бесплатно).

Приведем и еще одну интерпретацию четырех пунктов "свободы" для разработчиков программ по Столлману [54]. Разработка свободно распространяемого ПО была очень важным шагом, но еще большей заслугой Р. Столлмена следует признать создание "Стандартной Общественной Лицензии GNU" (GNU General Public License, или GPL). На русский язык это название разные авторы переводят по-разному: "Универсальная общественная лицензия", "Обобщенная Публичная Лицензия" и т.п. Но считается, что юридическую силу имеет только английский вариант этой лицензии. Основная идея GPL состоит в том, что пользователь должен обладать следующими четырьмя правами (или четырьмя свободами):

- правом запускать программу для любых целей (свобода 0);
- правом изучать устройство программы и приспосабливать ее к своим потребностям (свобода 1), что предполагает доступ к исходному коду программы;
- правом распространять программу, имея возможность помочь другим (свобода 2);
- правом улучшать программу и публиковать улучшения в пользу всего сообщества (свобода 3), что тоже предполагает доступ к исходному коду программы.

Публичная лицензия первой версии была выпущена в 1989 году. Через

пару лет вышла ее вторая версия, а третья была написана в 2005 году, но окончательный вариант был принят в 2007 году. Эти лицензии обозначаются так: GPL vX (где X может быть 1, 2 или 3). Из-за ограниченности размера книги приводим только название частей второй версии GPL.

1. Определения.
2. Право на копирование и распространение.
3. Изменение программы.
4. Требование предоставления исходного кода.
5. Прекращение действия лицензии при нарушении ее условий.
6. Акты, означающие принятие лицензии.
7. Запрещение дополнительных ограничений при дальнейшем распространении.
8. Внешние ограничения не снимают обязательства выполнять условия лицензии.
9. Возможность географических ограничений.
10. Будущие версии GNU GPL.
11. Запросы на исключения из правил.
12. Отказ от предоставления гарантий.
13. Отказ от ответственности.

Иногда, в противовес правам на интеллектуальную собственность (в том числе и на программы), именуемым copyright, программы, распространяемые в соответствии с лицензией, разработанной Столманом, связывают с термином copyleft (копилефт-лицензии).

Также отметим, в противовес чисто коммерческому направлению разработки и распространения программного обеспечения, существует и другое направление – "открытые исходники" (Open Source). Его определение сформулировал Брюс Перенс (Bruce Perens) в 1997 году. С изменениями оно было опубликовано на сайте [54]. В Интернете об этом движении много самой разнообразной информации. Дадим только одну ссылку [55], содержащую адреса этой тематики в Рунете.

Отметим, что Open Source не эквивалентен GNU или FSF. Ярые последователи каждого из них часто высказывают свое несогласие между собой. Сам же разработчик Linux (Торвальдс) старается держаться "поодаль" от перечисленных и других подобных движений. Эти два

термина отличаются расстановкой приоритетов. Сторонники open source делают упор на эффективность открытых исходников как метода разработки, модернизации и сопровождения программ. Сторонники free software считают, что именно права на свободное распространение, модификацию и изучение программ являются достоинством свободного ПО.

Linux – один из самых ярких представителей программного продукта, реализованного по методу открытых исходников. Но в этой разработке есть и нечто большее. Об этом ярко написал Эрик С. Рэймонд в статье "Базар и Собор" (The Cathedral and the Bazaar). Русский перевод можно найти в [56]. Там в противовес централизованному методу разработки программ предлагается другой метод – параллельный. При его использовании, разрабатывая программу, надо публиковать ее исходный текст с ранних стадий. Тогда создаются условия участия в проекте, например, на уровне обсуждения идей или частичной отладки, многих программистов. Об этом можно прочитать и статью Безрукова [57, 58].

Open Source имеет как много сторонников, так и противников. Его сторонники собираются на различные мероприятия, обсуждают свои проблемы в открытой печати и Интернете. Среди противников, что естественно, находим, прежде всего, представителей компьютерного бизнеса. Глава Microsoft неоднократно высказывался об Open Source. Например, в интернете есть публикация "Гейтс о бесплатном ПО" [59].

Исходные тексты своих программ публикуют и самые мощные представители компьютерного бизнеса. Это сделали, к примеру, Sun и даже Microsoft. Правда, последнюю фирму вряд ли можно "заподозрить" в приверженности к Open Source. Просто они оказались вынужденными передать исходные тексты своих программ, например, операционной системы Windows, под давлением [60].

Заметим, что параллельно с выпуском GPL v2 был разработан и в 1991 году оформлен документ, названный GNU Lesser General Public License (англ. "Стандартная общественная лицензия ограниченного применения GNU", сокращенно – GNU LGPL). Она была основана на GNU Library General Public License (англ. "Стандартная общественная лицензия GNU для библиотек"). Эти лицензии действуют на свободное программное обеспечение и одобрены Фондом свободного программного

обеспечения. Их цель – достигнуть компромисса между GPL и простыми разрешительными лицензиями (например, BSD License, MIT License, Mozilla Public License). LGPL была написана в 1991 году и затем обновлена в 1999 и 2007 годах Ричардом Столлманом и Эбеном Моглином. На странице "Лицензии открытого ПО" Википедии приведен список из более 50 элементов. Естественно, это создает определенные трудности.

В семействе GNU есть еще одна лицензия. Ее имя FDL, а с описанием можно познакомиться по ссылке: http://ru.wikipedia.org/wiki/GNU_Free_Documentation_License - http://ru.wikipedia.org/wiki/GNU_Free_Documentation_License – "Свободная лицензия GNU на документацию". Может рассматриваться как дополнение к основной лицензии GPL.

Эта копилефт-лицензия первоначально разрабатывалась для пользовательских руководств, учебников и документации, сопровождающей программы для компьютеров. Как и основная лицензия GNU (GPL), предполагает возможность воспроизведения, распространения и изменения исходных документов (в том числе и в коммерческих целях). При этом обязательно указывать авторов первоисточника. Заметим, что последний может содержать неизменяемые разделы.

Интерфейсы операционных систем

Основные понятия, связанные с интерфейсом операционных систем. Графический интерфейс пользователя в семействе UNIX/Linux. К истории X Window system. Основные понятия системы X Window. X Window в Linux. Интегрированная графическая среда KDE. Интегрированная графическая среда GNOME.

4.1. Основные понятия, связанные с интерфейсом операционных систем

В области информационных технологий имеется несколько фундаментальных понятий. Одно из них – "интерфейс". Отметим, что оно может трактоваться с различных точек зрения. В предыдущей главе описано понятие "Интерфейс системных вызовов". Если искать такой термин в "Словарях" Yandex, то будет получено более десятка определений термина, большая часть которых дана в сочетании с другими терминами, например: "Интерфейс передачи данных", "Программный интерфейс", "Прикладной интерфейс". В словаре "Естественные науки" на ГЛОССАРИЙ.RU дается следующее определение фундаментальному понятию.

Интерфейс в широком смысле – определенная стандартами граница между взаимодействующими независимыми объектами. Интерфейс задает параметры, процедуры и характеристики взаимодействия объектов.

В "Издательском словаре-справочнике" [4.1] есть также определения основному термину "интерфейс". Это:

1. Система связей и взаимодействия устройств компьютера.
2. Средства взаимодействия пользователей с операционной системой компьютера, или пользовательской программой. Различают графический интерфейс пользователя (взаимодействие с компьютером организуется с помощью пиктограмм, меню, диалоговых окон и пр.) и интеллектуальный интерфейс (средства взаимодействия пользователя с компьютером на естественном языке пользователя).

Как видим, здесь этот термин имеет два значения. Но мы кратко остановимся на втором – "интерфейс пользователя". На уже упомянутом нами источнике ГЛОССАРИЙ.RU он определяется так: "Интерфейс пользователя – это элементы и компоненты программы, которые способны оказывать влияние на взаимодействие пользователя с программным обеспечением, в том числе:

- средства отображения информации, отображаемая информация, форматы и коды;
- командные режимы, язык пользователь-интерфейс;
- устройства и технологии ввода данных;
- диалоги, взаимодействие и транзакции между пользователем и компьютером;
- обратная связь с пользователем;
- поддержка принятия решений в конкретной предметной области;
- порядок использования программы и документация на нее".

По мере развития вычислительной техники методы и средства взаимодействия пользователя с операционной системой менялись. Широкое распространение цифровых вычислительных машин привело к режиму общения между человеком и ЭВМ на специальном языке. Сначала, в период пакетной обработки заданий, это реализовывалось с применением специальных носителей информации (например, перфокарт, на которые наносились задания для компьютера). Но в дальнейшем, с широким распространением терминалов и клавиатуры, основным стал командный режим работы пользователя, при котором взаимодействие строилось на основе системы встроенных команд. В свободной энциклопедии "Википедия" он определен так.

Интерфейс командной строки (англ. Command line interface, CLI) – разновидность текстового интерфейса (CUI) между человеком и компьютером, в котором инструкции компьютеру даются в основном путем ввода с клавиатуры текстовых строк (команд), в UNIX-системах возможно применение мыши. Также известен под названием "консоль".

Приведем приблизительный фрагмент экрана, который появляется в режиме командной строки (рис. 4.1).

Рис. 4.1.

Слева в строке появляется приглашение ([asplinux@asplinuxlive ~]), после него можно набрать команду, результаты которой выводятся далее. Приведем пример выполнения команды date в системе Linux (рис. 4.2).

Рис. 4.2.

Первые операционные системы фирмы Microsoft для персональных компьютеров IBM PC (они назывались MS DOS) также поддерживали командный режим, схожий с другими системами. Строка, в которой набирались команды, была схожей с приведенными выше. Сегодня командный режим операционных систем обеспечивается эмуляторами cmd.exe (для 32-х разрядного режима) или command.com (для 16-и разрядного режима). В графическом режиме семейства UNIX/Linux командная строка эмулируется программой Терминал (xterm).

Отметим, что для компьютеров с операционной системой MS DOS удачным дополнением реализации такого интерфейса пользователя стала легендарная программа Norton Commander. Она минимизировала действия по набору текста в командной строке, позволяя оперировать, прежде всего, выбором подходящей команды из меню. В этой программе также активно используются функциональные клавиши компьютера. Википедия эту систему описывает следующим образом:

"Norton Commander (NC) – популярный файловый менеджер для DOS, первоначально разработанный американским программистом John Socha (некоторые дополнительные компоненты были полностью или частично написаны другими людьми: Linda Dudinyak – Commander Mail,

вьюеры; Peter Braden – Commander Mail; Keith Ermel, Brian Yoder – вьюеры). Программа была выпущена компанией Peter Norton Computing (глава – Питер Нортон), которая позже была приобретена корпорацией Symantec".

Приведем пример снимка экрана этого файлового менеджера (рис. 4.3).

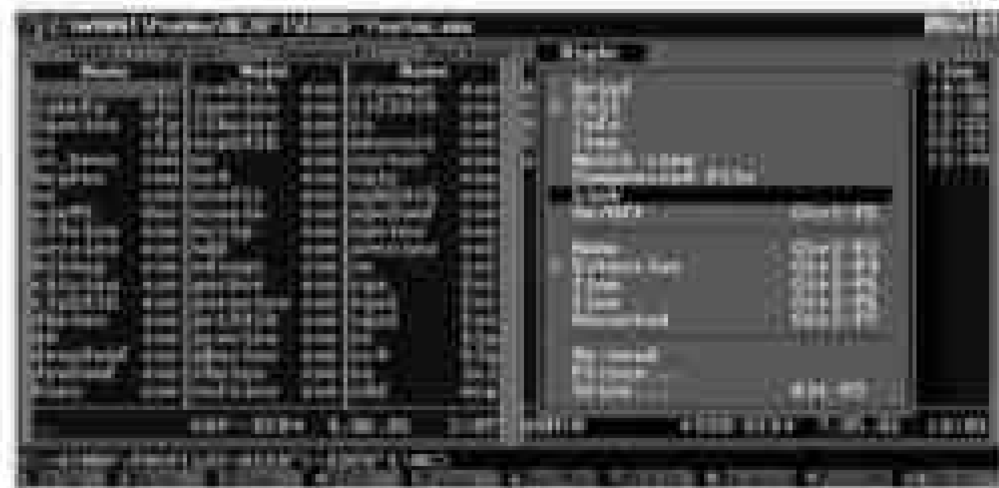


Рис. 4.3. Легендарный файловый менеджер Norton Commander

Популярность программы была настолько велика, что появились многочисленные клоны, которые более или менее точно копировали нортоновский интерфейс. К примеру, DOS Navigator, визуально схожий с Norton Commander-ом, предоставлял даже большие возможности. Для операционной системы Microsoft Windows появились Volkov Commander, FAR Manager, Total Commander и другие аналогичные программы. Впоследствии клоны появились и на других операционных системах: BSD, GNU/Linux - Midnight Commander, Krusader.

Norton Commander не только спровоцировал целую серию собственных клонов и реплик, но и внес в русский язык пару новых слов - "нортон" и "коммандер" стали в жаргоне пользователей ПК синонимами словосочетания "файловый менеджер".

Введенная программой парадигма работы с файлами (2 одинаковые панели, между которыми происходит операция; большинство команд выполняется по "горячим клавишам") до сих пор применяется в

подавляющим большинством файловых менеджеров.

Norton Commander также стал персонажем серий притч и анекдотов. Первая серия была написана Александром Голубевым, несколько последующих выпускались и дополнялись различными авторами, имена которых постепенно были утеряны, после чего эти рассказы перешли в состояние фольклора.

Также имеется музыкальная группа NortonCommander.

В разных версиях Linux используется аналог такой программы, называемой Midnight Commander. Приведем ее вид (рис. 4.4), когда она вызвана в режиме эмуляции командной строки.

Рис. 4.4. Программа Midnight Commander, выполненная в Терминале

Но идея разделения окна на две части, в которых представлено содержимое каталогов, осталась привлекательной и при появлении операционной системы только с графическим интерфейсом – Windows 95. Аналоги Norton Commander для этой и последующих версий многочисленны. В интегрированной графической среде UNIX аналогом NC является GNOME Commander. Приведем вид (рис. 4.5) файлового менеджера Total Commander (ранее известного как Windows Commander) операционной системы Windows XP [62].

Рис. 4.5. Файловый менеджер Total Commander

Но сегодня командный режим уходит в прошлое, уступая место другим. Кроме командного, определяются еще два современных вида интерфейса: WIMP и SILK.

WIMP-интерфейс (Window – окно, Image – образ, Menu – меню, Pointer – указатель). Характерной особенностью этого вида интерфейса является то, что диалог с пользователем ведется не с помощью команд, а с помощью графических образов – меню, окон, других элементов. Хотя и в этом интерфейсе подаются команды машине, но это делается "опосредованно", через графические образы. Этот вид интерфейса реализован на двух уровнях технологий: простой графический интерфейс и "чистый" WIMP-интерфейс.

SILK-интерфейс (Speech – речь, Image – образ, Language – язык, Knowledge – знание). Этот вид интерфейса наиболее приближен к обычной, человеческой форме общения. В рамках этого интерфейса идет обычный "разговор" человека и компьютера. При этом компьютер

находит для себя команды, анализируя человеческую речь и находя в ней ключевые фразы. Результат выполнения команд он также преобразует в понятную человеку форму. Этот вид интерфейса наиболее требователен к аппаратным ресурсам компьютера, и поэтому его применяют в основном для военных целей.

Долгое время возможности компьютеров, их технические характеристики предписывали пользователям работу в командном режиме как в основном. Первые персональные компьютеры также использовали его. Но в последние годы такой режим вытеснил другим – графическим. Он потребовал от компьютера больших ресурсов, но принес новое – удобство, разнообразный дизайн, многооконность (правда последняя может быть реализована и в командном режиме). Для обозначения графического режима используют аббревиатуру GUI (Graphics User Interface), что дословно переводят как "графический интерфейс пользователя", но часто при переводе заменяют на "многооконный графический интерфейс".

Первое появление графического интерфейса (рис. 4.6) следует связывать с фирмой XEROX. В ее лаборатории PARC (Palo Alto Research Center) в 1973 году создавался компьютер Alto. Последний был оснащен мышью и хорошим монитором. Считают, что этот компьютер обладал GUI, но широкого распространения не получил. Xerox все-таки решается вдохнуть жизнь в экспериментальный Alto, выпустив на рынок его полноценного коммерческого преемника – компьютер Star.

Рис. 4.6. Первый графический интерфейс от фирмы Xerox

Приведем высказывание из статьи Олега Свиргстина [63]: "Alto был первым в мире компьютером, на котором были практически реализованы метафора "рабочего стола" и графический пользовательский интерфейс, прежде существовавшие только в теоретических разработках".

Для операционных систем семейства UNIX, как и многих других, долгое время командный режим работы был основным. Пожалуй, сегодня он используется в основном для администрирования, его потеснил режим графического интерфейса. Фирма Microsoft более 10 лет (с 1981 года) обеспечивала персональным компьютерам IBM PC только командный режим, в то время как у конкурентов уже в 1984 году был реализован GUI. Правда, эта компания стремилась реализовать последний режим работы, что и было достигнуто в середине 90-х.

Приведем рисунок, иллюстрирующий этапы работы операционных систем Microsoft и UNIX в командном и графическом режимах. Из него видно, что для операционных систем UNIX/Linux до настоящего времени графический режим является надстройкой над командным, а для Windows – командный режим как основной прекратил существование в 1995 году (рис. 4.7).

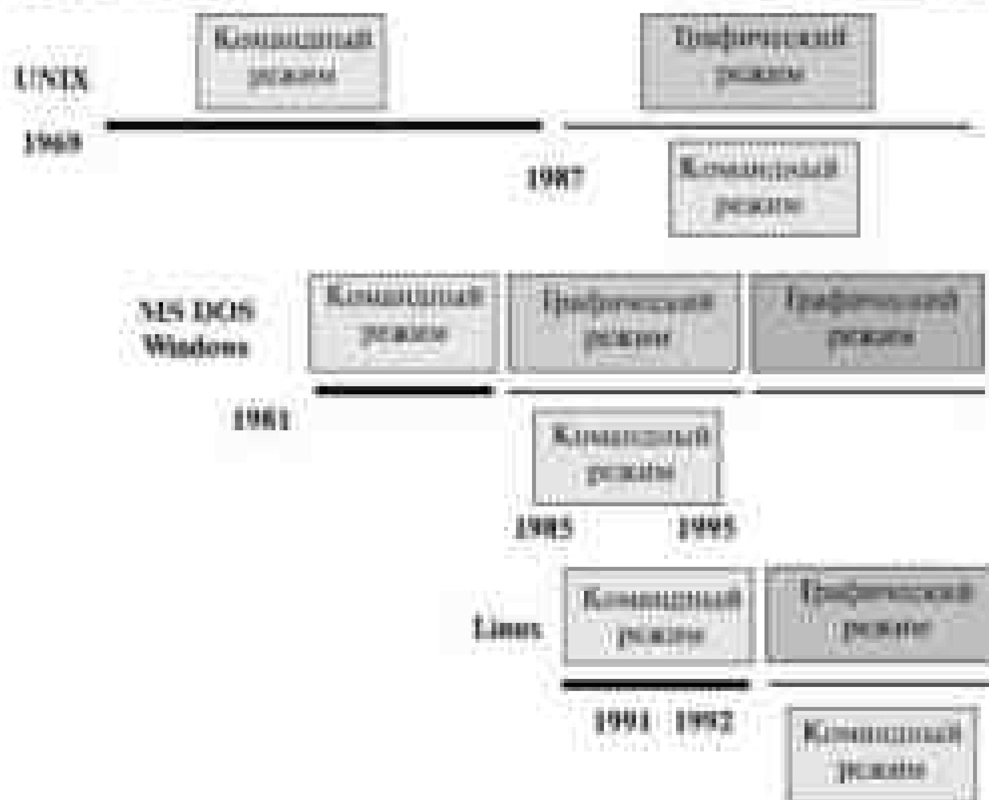


Рис. 4.7. Командный и графический интерфейс семейства UNIX/Linux и Windows.

Отметим, что операционная система MS DOS последние свои годы снабжалась надстройкой, обеспечивающей пользователям GUI. Названия этих графических оболочек были Windows1, Windows2, Windows3.

Из других графических интерфейсов назовем OPENSTEP, реализованный на компьютерах фирмы NeXT. Его создавал Стивен Джобс, основатель фирмы Apple, в период, когда он покинул ее и пытался завоевать мир новой разработкой. Этот интерфейс в дальнейшем был перенесен и на другие компьютерные платформы (рис. 4.8).

Обратите внимание на его отличия от того, что в это время предлагала фирма Microsoft со своей Windows95 (пример рабочего стола приведен в

главе 2, в части, посвященной операционным системам этой фирмы).

По адресу ссылки: http://www.guidebookgallery.org/guis - http://www.guidebookgallery.org/guis можно ознакомиться с "галереей" графических интерфейсов пользователей на разных компьютерных платформах. Приведем два снимка экрана, на которых представлен перечень всех элементов галереи (рис. 4.9).

Рис. 4.8. Графический интерфейс OPENSTEP for 1997 платформы

Отдельно остановимся на списке из 5 элементов Desktop раздела GUI (см. выше). Они содержат ссылки на описание систем, обеспечивающих графический интерфейс пользователя UNIX. Здесь коротко упомянем только два, остальные подробно рассматриваются дальше.

Рис. 4.9а: Галерея графических интерфейсов на разной аппаратуре (часть I)

На этих рисунках обратите внимание на более чем десяток типов рабочих столов (от Amiga OS до Xerox Star/View Point/Global View). Хотя рабочий стол Windows занимает одно из мест, но на сегодняшний день многие производители приняли его стандарты. В этом же ряду упомянуты системы, активно влиявшие на развитие операционных систем, но сегодня уже не существующие. Среди них:

- OS/2 от IBM, долгое время являвшаяся конкурентом Windows;
- BeOS, созданная корпорацией Be Inc. и обладавшая в момент своего выпуска многими пионерскими новинками. Это работа на 64-разрядной аппаратуре, удобный интерфейс пользователя и многое другое.

Рис. 4.86. Галерея графических интерфейсов на разной аппаратуре (часть 2)

OPEN LOOK представляет собой спецификацию графического интерфейса пользователя рабочих станций UNIX. Была создана в конце 1980-х годов Sun Microsystems и AT&T при участии Xerox. Эта спецификация была основной для операционной системы на ранних стадиях реализации графического интерфейса. Впоследствии утеряла свое значение в связи с появлением графического интерфейса Motif от OSF. Common Desktop Environment (CDE) – среда рабочего стола, основанная на системе Motif. Она была создана The Open Group вместе с рядом фирм: Hewlett-Packard, IBM, Novell. Некоторое время она была промышленным стандартом для UNIX-систем.

Режим GUI используется в разных операционных системах. Многие его разработчики пытались найти свой, наиболее привлекательный для пользователей "стиль". Со временем они вынуждены были оглядываться на то, что делают другие фирмы, или даже объединяться для стандартизации составляющих графического интерфейса.

Современное представление о графическом интерфейсе, на наш взгляд, объединяет все лучшее от разных производителей. Думается, поиски в этом направлении продолжатся и далее, хотя часто говорят о том, что с первых шагов становления графического интерфейса ничего принципиально не изменилось – все его основные элементы остаются прежними (рабочий стол, меню, иконки).

Пожалуй, следует отметить и еще одну тенденцию: последние варианты реализации графического интерфейса построены с "оглядкой" на то, что реализовано в операционных системах Windows. Это объясняется большой их долей (около 90%) на рынке персональных компьютеров.

Как уже говорилось выше, для систем UNIX долгое время – с начала 70-х годов и, пожалуй, до конца 80-х – единственным режимом был командный режим работы. Сегодня он уступил свое место графическому. В семействе операционных систем UNIX (напомним, работающих на разных аппаратных платформах) графический интерфейс пользователя поддерживается системой X Window System. Основной сайт с информацией о ней имеет адрес ссылки http://www.x.org - http://www.x.org. Последняя версия, представленная там, имеет имя X11R7.5.

4.2. Графический интерфейс пользователя в семействе UNIX/Linux

4.2.1. К истории X Window system

X Window system появилась в результате объединения усилий двух исследовательских групп: MIT группа, ответственная за сетевую программу (проект "Афина" – Project Athena) и *Лаборатория информатики* (Laboratory for Computer Science). До десятой версии X Window этот проект реализовали три программиста: Роберт Шейфлер (Robert Scheifler), Джим Геттис (Jim Gettys) и Рон Ньюман (Ron Newman). Двое из них работали в MIT, а третий в DEC [16, 17].

Первоначально разрабатываемая в MIT (Массачусетском технологическом институте) система X Window стала распространяться

свободно. Было создано несколько версий, и последняя из них, успешно используемая до настоящего времени, имела номер, присвоенный при создании и равный 11. Чаще других применяется версия 11, имеющая номер реализации 6. Поэтому на компьютерах с установленной системой Window часто встречаются каталоги, в названии которых есть символы X11R6 или X11.

В дальнейшем разработкой средств, обеспечивающих GUI для операционной системы UNIX, в режиме жесткой конкуренции занимались многие крупные компьютерные фирмы. При этом некоторые из них объединялись для совместных действий и даже создания стандартов.

В 1987 году ряд фирм решили создать единый стандарт оконного интерфейса для UNIX и для этого основали X Consortium ("Консорциум X"). В нем приняли участие IBM, DEC, HP и другие компании. Этот проект возник в противовес объединению AT&T и Sun. С 1997 X Consortium преобразовалась в "Открытую группу X" (X for the Open Group) [16]. Информацию о деятельности этой организации (ее современное имя X.Org Foundation) можно получить в Интернете [64].

В статье [65] приведены примеры четырех исторически появившихся видов графического интерфейса XWindows (OpenLook, Motif, KDE и трехмерный графический интерфейс). Там о них говорится следующее:

"Эволюция пользовательских интерфейсов, построенных на основе X Window, наглядно доказывает преимущества выбранного разработчиками системы подхода. Свобода в определении политик и простота использования механизмов позволили X Window пройти эволюционным путем от внешне примитивного вида OpenLook к де-факто стандартному экранному представлению примитивов пользовательского интерфейса Motif, к гибко настраиваемому современному виду KDE и, наконец, к прообразам трехмерного интерфейса".

Заметим, что трехмерный графический интерфейс появился сравнительно недавно. Но самые последние версии популярных операционных систем реализуют его. Это относится к разновидностям Linux, Mac OS и версий Microsoft начиная с Vista [66].

Не претендуя на полноту охвата вопроса, отметим, что трехмерные рабочие столы могут быть построены на разных эффектах. Одним из первых была реализована метафора рабочей комнаты со шкафчиками, ящичками и тому подобными элементами, которые открывались, выдвигались и т.д. Другой подход, видимо, основан на объемной фигуре, которую можно поворачивать и изменять в размерах. И еще одна идея 3D Desktop предполагает использование прозрачных окон, за которыми можно увидеть информацию расположенных за ними окон. С одной из наиболее распространенных версий Linux Mandriva сегодня поставляется Metisse (рис. 4.10). Последний основан на эффекте перспективы.

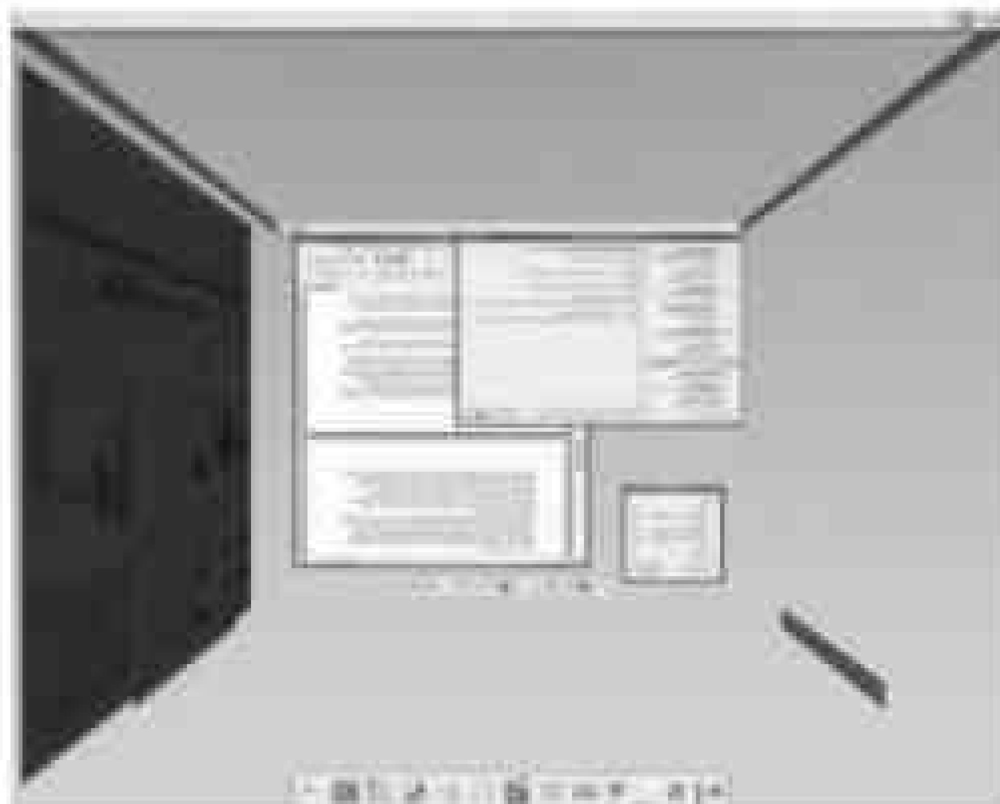


Рис. 4.10. Пример трехмерного графического интерфейса Metisse дистрибутива Linux Mandriva.

Для операционных систем Mac OS 3D интерфейс реализуется в Aqua. Он основан на эффекте прозрачности (рис. 4.11).

Рис. 4.11. Пример трехмерного графического интерфейса Aqua Mac OS

Еще один пример трехмерного интерфейса от Mac OS, при котором каждый пользователь работает на своей грани куба (рис. 4.12).

Рис. 4.12. Пример трехмерного графического интерфейса Aqua Mac OS

Трехмерный интерфейс операционной системы Windows Vista получил название Aero. Он построен на эффекте 3D Flip.

Рис. 4.13. Пример трехмерного графического интерфейса Aero Windows Vista

4.2.2. Основные понятия системы X Window

X Window система (или просто X Window, а теперь часто и X) – графическая среда пользователя, поддерживающая одновременное выполнение многих программ в сети. В основе X Window – библиотека графических программ, используемых для создания GUI.

ЗАМЕЧАНИЕ. Отметим, что термину X Window дают разное определение. Поиск в Интернете позволяет получить их более десятка.

Достоинством системы X Window является ее мобильность (она не связана с конкретной операционной системой и не рассчитана на специфическое техническое обеспечение). Работа X-системы основана на специфической модели клиент/сервер.

В традиционной модели "клиент-сервер" с пользователем взаимодействует клиентская часть. В системе же X Window с пользователем взаимодействует X-сервер. Он отвечает за вывод информации на экран пользователя и получение от него команд. Такой сервер как бы "владеет" аппаратурой пользователя (называемой X-терминал) и предоставляет этот ресурс программам – клиентам. Именно они формируют изображение, выводимое на экране. При инициализации X Window одним из первых шагов будет загрузка X-сервера. Об этом можно узнать по появлению на сером экране в центре указателя мыши в виде крестика.

Но для окончательного вывода на экран сформированного программой клиентом изображения одного X-сервера мало. Для этого еще необходим менеджер окон.

Таким образом, система X Window представляет собой комплекс взаимодействующих компонент. Интересно, что существует несколько вариантов каждого элемента, из которых "собирается" конкретный экземпляр системы.

Следуя J.Hall [6], приведем схематическое изображение архитектуры графической системы (рис. 4.14).

Рис. 4.14. Архитектура X Window

Это упрощенная схема. Обязательными являются еще такие два компонента. Взаимодействие между графическими библиотеками и X-сервером реализуется по протоколу TCP/IP. Также важным элементом рассматриваемой системы являются шрифты, поэтому в системе можно выделить и еще один элемент – сервер шрифтов.

Приведем схему, взятую с сайта ссылке: http://www.answers.com/topic/x-window-system - http://www.answers.com/topic/x-window-system (рис. 4.15).

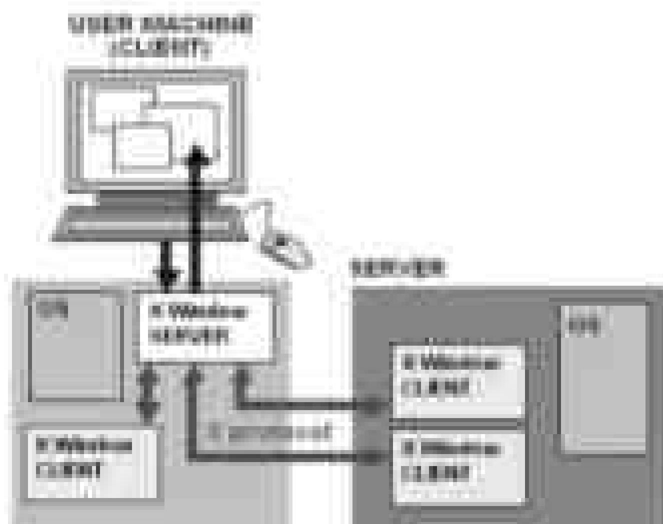


Рис. 4.15. X Windows сервер выполняется на клиенте

Из этой схемы видно, что программы, выполняющие роль X Window SERVER и X Window CLIENT, могут располагаться как на одном компьютере, так и на разных. Каждая из них может работать под управлением своей операционной системы. Взаимодействие между X-клиентом и X-сервером реализуется по специальному протоколу (X protocol). В этой схеме не обозначены драйверы устройств, обеспечивающие работу конкретной аппаратуры и вместе с X-сервером образующие X-терминал. За вывод информации отвечает такой компонент, как менеджер окон, обеспечивающий, по инициативе прикладных программ, вывод на экран множества перекрывающихся окон, расположенных в нужном месте экрана и имеющих требуемый размер. Этот компонент изображен на USER MACHINE (CLIENT).

Отметим, что общей идеологии X Window system не противоречит ситуация, при которой все компоненты располагаются на одном компьютере, что реализуется, к примеру, в Linux.

Таким образом, система X Window представляет комплекс взаимодействующих элементов, каждый из которых, в принципе, может быть заменен новым компонентом. Все это делает систему достаточно гибкой и легко модифицируемой.

Обратим ваше внимание, что графический режим в операционных системах семейства UNIX/Linux не является обязательным. Он вызывается из командной строки. Заметим, что таким же образом запускалась, например, графическая многооконная оболочка Windows3.X фирмы Microsoft. Из-за сложности процедуры запуска графический интерфейс активизируется целым набором действий. Для систем UNIX в таких случаях предусматривают создание специальных скриптов (сценариев). Долгое время традиционное название файла запуска было startx, а файл конфигурации параметров графического режима носил название XF86Config. Но сейчас это не является обязательным для всех систем.

4.2.3. X Window в Linux

Операционная система Linux в последние годы отходит от командного режима как основного и использует графический режим для разных действий: от работы пользователей с прикладными программами до настройки системы администратором. На первоначальном этапе своего развития Linux ориентировался на скромные ресурсы компьютера и, как следствие, командный режим как основной. Но современные версии этой операционной системы для реализации графического режима требуют больших ресурсов компьютера.

Долгое время в Linux использовалась версия X Window, ориентированная на IBM PC и названная XFree86. Она основывалась на стандарте X11R6, но имела ограничения на используемое оборудование. Как и многое в семействе UNIX/Linux, XFree86 постоянно развивается усилиями многих программистов в соответствии с принятыми стандартами. Последние ее версии имеют

номер 4.8.0 (декабрь 2008 года, ссылки: http://xfree86.org/releases/rel480.html - http://xfree86.org/releases/rel480.html). Для этого графического режима написано много самого разнообразного программного обеспечения. Большая его часть распространяется свободно и бесплатно, но ничем не уступает своим коммерческим аналогам. Это – офисные и графические программы, системы для управления предприятием и средства разработки.

Но с 1999 года параллельно с XFree86 возникает X.Org, основанная The Open Group. Первое время она не была популярной и использовала основные технические достижения XFree86. Но в последние годы ситуация изменилась. В начале 2004 года представители X.Org и freedesktop.org основали фонд X.Org Foundation. The Open Group передала ему управление доменным именем x.org. Это стало коренным изменением в разработке X. В то время как распорядители X с 1988 года (включая предыдущую X.Org) были организациями поставщиков, X.Org Foundation был основан самими разработчиками программного обеспечения, и в нем применялась открытая модель разработки, опирающаяся на вклад извне.

Графический интерфейс семейства UNIX/Linux похож на интерфейс других систем, но имеет отличия. Он поддерживает метафору рабочего стола. Но в отличие от некоторых систем имеет несколько рабочих столов, которые иногда называются еще и "рабочие места". Их количество можно изменять. Хотя графический UNIX зародился раньше, чем у других операционных систем, сейчас работа с использованием GUI аналогична у Linux и Windows. Пользователь работает с приложением в окне, имеющем прямоугольную форму. Последний содержит стандартные элементы – строка заголовка, главное меню, панели инструментов и т.д.

В X Window управление окнами приложений, их элементами выполняет компонент, называемый "менеджер окон" (иногда используют название "оконные менеджеры" или "диспетчер окон"). Может быть задействовано несколько диспетчеров окон.

Но сегодня пользователи редко выбирают менеджеры окон. Им представляются интегрированные графические среды. Две наиболее распространенные из них – KDE и GNOME – будут коротко

рассмотрены далее. Но сначала приведем список инструментов пользователя. На странице Википедии "Менеджер окон X Window System" приводятся такие списки. Интерфейс пользователя в UNIX-подобных системах:

1. среды рабочего стола: CDE, EDE, étoile, GNOME, JDS, KDE, LXDE, Mezzo, OpenWindows, ROX, Xfce, Xpde;
2. оконные менеджеры: AfterStep, Awesome, Blackbox, CTWM, dwm, Enlightenment, Fluxbox, FVWM, IceWM, JWM, Openbox, Sawfish, twm, Window Maker, wmii;
3. командные оболочки: ash, Bash, BusyBox, csh, dash, es shell, fish, ksh, jsh, rc, rsh, Sash, Scsh, sh, tcsh, Thompson shell, zsh и прочие.

Приводятся три категории: среды рабочего стола, оконные менеджеры и командные оболочки. Последние обеспечивают режим командной строки. Как видим, их много. Название первой образовано от английского shell (оболочка). В разных вариантах Linux распространена оболочка, имя которой Bash образовано от Born again shell (разработана Bourne).

А теперь, как было сказано ранее, приведем короткую информацию о двух интегрированных графических средах KDE и GNOME.

4.2.4. Интегрированная графическая среда KDE

Часто графическую среду KDE называют наиболее распространенной. Проект был основан в октябре 1996 года студентом Маттиасом Эттрихом, а в июле 1998 года выпущена версия 1.0. Сокращение образовано от K Desktop Environment. Она строится на основе инструментария разработки пользовательского интерфейса с именем Qt. Интересной особенностью последнего является свойство кроссплатформенности. Хотя эта среда разрабатывается для UNIX-подобных систем, но возможен ее запуск и на других платформах, например, с использованием Cygwin под Microsoft Windows.

KDE включает в себя набор тесно взаимосвязанных программ пользователя. В его рамках разрабатывается полнофункциональный офисный пакет KOffice, а также интегрированная среда разработки

KDevelop.

Основной адрес в Интернете команды KDE – ссылка: http://www.kde.org - http://www.kde.org, а в России – ссылка: http://kde.ru/ - http://kde.ru/. В 2010 году начат выпуск версии 4.0, содержащей следующие основные нововведения:

- переход на четвертую версию библиотеки элементов интерфейса Qt;
- новый стиль оформления – Oxygen;
- новый мультимедийный интерфейс API – Phonon;
- объединение SuperKaramba, рабочего стола и панели Kicker в одно приложение – Plasma.

Эта версия обеспечивает новые технологии не только для UNIX, но и для Microsoft Windows и Mac OS X. Узнать компьютер, на котором работает KDE, можно по его талисману – дракончику Konqi (рис. 4.16). Обратите внимание, что на изображении Konqi можно увидеть другой символ, который часто появляется при работе в среде KDE.

Рис. 4.16. Талисман KDE

Еще отметим, что по адресу ссылка: http://www.kde.ru/wiki/HomePage - http://www.kde.ru/wiki/HomePage расположены страницы русского проекта локализации KDE, где содержимое создается, изменяется, обсуждается и поддерживается пользователями, разработчиками и всеми остальными, кто как-либо причастен к этому проекту.

4.2.5. Интегрированная графическая среда GNOME

Название GNOME является акронимом от английского GNU Network Object Model Environment ("Сетевая объектная среда GNU"). На русскоязычном сайте [67], посвященном этой интегрированной среде, дается такой ответ на вопрос "Что такое GNOME?": в рамках проекта GNOME создаются две вещи – рабочая среда GNOME, простая в использовании и привлекательная на вид среда рабочего стола, а также платформа разработки GNOME – расширяемая среда для создания приложений, тесно интегрируемых с рабочим столом.

Основной сайт проекта GNOME располагается по адресу ссылки: http://www.gnome.org – http://www.gnome.org. Его история начинается с 1997 года и связана с именами Мигеля де Икасы и Федерико Мены. Основной целью было создать полностью свободную рабочую среду для операционной системы GNU/Linux [88], поскольку основной инструмент разработки Qt – другой интегрированной среды KDE – не был лицензирован на условиях GNU GPL. Отметим, что эти проблемы были ликвидированы в версии Qt 2.2 в 2000 году.

Среда рабочего стола GNOME была построена на основе GTK+, созданной при разработке мощного графического пакета GIMP. Кроме того, используется еще много различных технологий и библиотек. Описываемая интегрированная среда может быть запущена на большинстве UNIX-систем, адаптирована для работы под управлением Solaris, а также через специальный порт может быть запущена под Windows.

Среди других особенностей интегрированной графической среды отметим Java-апплеты – набор приложений, встраиваемых в панель рабочего стола (GNOME Panel) для выполнения различных функций (например, с именем "Часы" или "Расчистка рабочего стола"). Логотипом системы является следующее изображение (лапка Гнома).

Рис. 4.17. Логотип GNOME

За локализацию среды GNOME отвечает проект перевода GNOME [2] (англ. GNOME Translation Project). Перевод пользовательского интерфейса и документации производится с помощью инструментария gettext.

Статистика [2] для GNOME 2.30:

- на 32 языках переведено более 90 % строк пользовательского интерфейса;
- ещё на 33 языка переведено от 50 % до 90 % строк;
- на русский язык переведено 99 % строк пользовательского интерфейса и 46 % строк документации.

Последняя версия 2010 года имеет номер 2.30.

Организация вычислительного процесса

Концепция процессов и потоков. Задание, процессы, потоки (нити), волокна. Мультипрограммирование. Формы многопрограммной работы. Управление процессами и потоками. Создание процессов и потоков. Модели процессов и потоков. Планирование заданий, процессов и потоков. Взаимодействие и синхронизация процессов и потоков. Методы взаимоисключений. Семафоры и мониторы. Взаимоблокировки (тупики). Синхронизирующие объекты ОС. Аппаратно-программные средства поддержки мультипрограммирования. Системные вызовы.

5.1. Концепция процессов и потоков. Задание, процессы, потоки (нити), волокна

Одним из основных понятий, связанных с операционными системами, является процесс – абстрактное понятие, описывающее работу программы [10]. Все функционирующее на компьютере программное обеспечение, включая и операционную систему, можно представить набором процессов.

Задачей ОС является управление процессами и ресурсами компьютера или, точнее, организация рационального использования ресурсов в интересах наиболее эффективного выполнения процессов. Для решения этой задачи операционная система должна располагать информацией о текущем состоянии каждого процесса и ресурса. Универсальный подход к предоставлению такой информации заключается в создании и поддержке таблиц с информацией по каждому объекту управления.

Общее представление об этом можно получить из рис. 5.1, на котором показаны таблицы, поддерживаемые операционной системой: для памяти, устройств ввода-вывода, файлов (программ и данных) и процессов. Хотя детали таких таблиц в разных ОС могут отличаться, по сути, все они поддерживают информацию по этим четырем категориям. Располагающий одними и теми же аппаратными ресурсами, но управляемый различными ОС, компьютер может работать с разной степенью эффективности. Наибольшие сложности в управлении ресурсами компьютера возникают в мультипрограммных ОС.

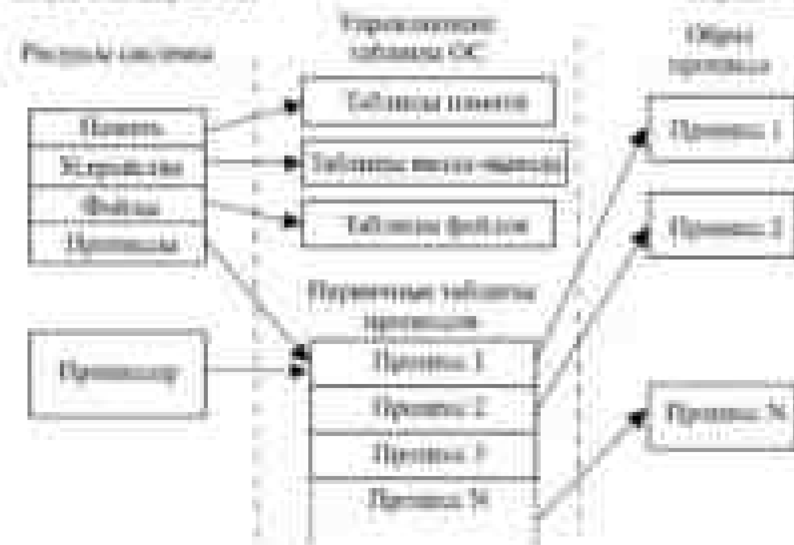


Рис. 5.1. Таблицы ОС

Мультипрограммирование (многозадачность, multitasking) – это такой способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ. Чтобы поддерживать мультипрограммирование, ОС должна определить для себя внутренние единицы работы, между которыми будут разделяться процессор и другие ресурсы компьютера. В ОС пакетной обработки, распространенных в компьютерах второго и сначала и третьего поколения, такой единицей работы было задание. В настоящее время в большинстве операционных систем определены два типа единиц работы: более крупная единица – процесс, или задача, и менее крупная – поток, или нить. Причем процесс выполняется в форме одного или нескольких потоков.

Вместе с тем, в некоторых современных ОС вновь вернулись к такой единице работы, как задание (Job), например, в Windows. Задание в Windows представляет собой набор из одного или нескольких процессов, управляемых как единое целое. В частности, с каждым заданием ассоциированы квоты и лимиты ресурсов, хранящиеся в соответствующем объекте задания. Квоты включают такие пункты, как максимальное количество процессов (это не позволяет процессам задания создавать бесконтрольное количество дочерних процессов), суммарное время центрального процессора, доступное для каждого

процесса в отдельности и для всех процессов вместе, а также максимальное количество используемой памяти для процесса и всего задания. Задания также могут ограничивать свои процессы в вопросах безопасности, например, получать или запрещать права администратора (даже при наличии правильного пароля).

Процессы рассматриваются операционной системой как заявки или контейнеры для всех видов ресурсов, кроме одного – процессорного времени. Это важнейший ресурс распределяется операционной системой между другими единицами работы – потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд. Каждый процесс начинается с одного потока, но новые потоки могут создаваться (порождаться) процессом динамически. В простейшем случае процесс состоит из одного потока, и именно таким образом трактовалось понятие "процесс" до середины 80-х годов (например, в ранних версиях UNIX). В некоторых современных ОС также положение сохранилось, т.е. понятие "поток" полностью поглощается понятием "процесс".

Как правило, поток работает в пользовательском режиме, но когда он обращается к системному вызову, то переключается в режим ядра. После завершения системного вызова поток продолжает выполняться в режиме пользователя. У каждого потока есть два стека, один используется в режиме ядра, другой – в режиме пользователя. Помимо состояния (текущие значения всех объектов потока) идентификатора и двух стеков, у каждого потока есть контекст (в котором сохраняются его регистры, когда он не работает), приватная область для его локальных переменных, а также может быть собственный маркер доступа (информация о защите). Когда поток завершает работу, он может прекратить свое существование. Процесс завершается, когда прекратит существование последний активный поток.

Взаимосвязь между заданиями, процессами и потоками показана на рис. 5.2.

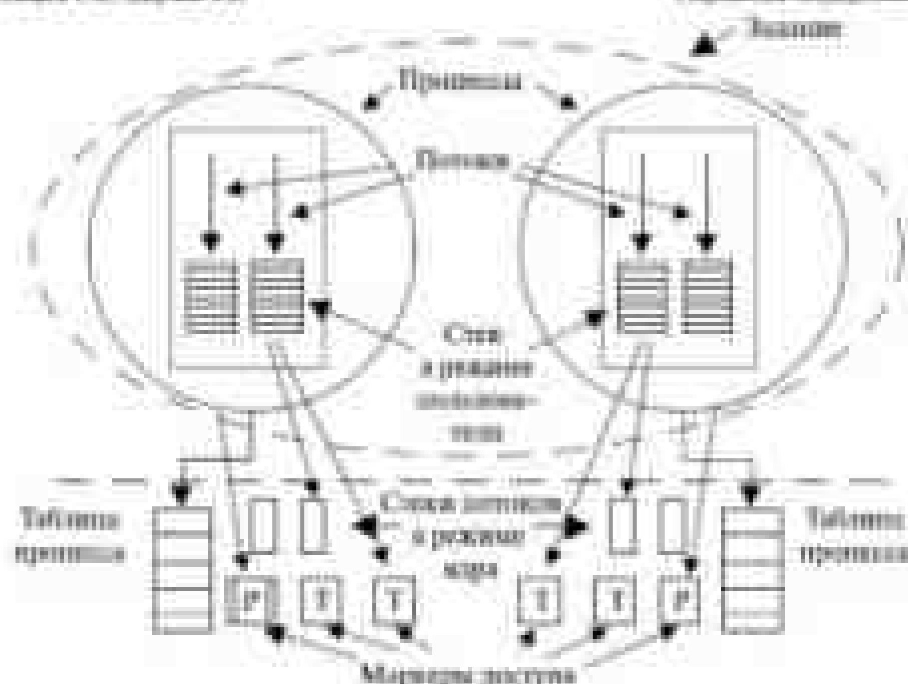


Рис. 5.2. Задания, процессы, потоки

Переключение потоков в ОС занимает довольно много времени, так как для этого необходимы переключение в режим ядра, а затем возврат в режим пользователя. Достаточно велики затраты процессорного времени на планирование и диспетчеризацию потоков. Для предоставления сильно облегченного псевдопараллелизма в Windows 2000 (и последующих версиях) используются волокна (Fiber), подобные потокам, но планируемые в пространстве пользователя создавшей их программой. У каждого потока может быть несколько волокон, с той разницей, что когда волокно логически блокируется, оно помещается в очередь блокированных волокон, после чего для работы выбирается другое волокно в контексте того же потока. При этом ОС "не знает" о смене волокон, так как все тот же поток продолжает работу.

Таким образом, существует иерархия рабочих единиц операционной системы, которая применительно к Windows выглядит следующим образом (рис. 5.3).

Возникает вопрос: зачем нужна такая сложная организация работ,

выполняемых операционной системой? Ответ нужно искать в развитии теории и практики мультипрограммирования, цель которой – в обеспечении максимально эффективного использования главного ресурса вычислительной системы – центрального процессора (нескольких центральных процессоров).

Поэтому прежде чем перейти к рассмотрению современных принципов управления процессором, процессами и потоками, следует остановиться на основных принципах мультипрограммирования.

Рис. 5.3. Иерархия рабочих единиц ОС

5.2. Мультипрограммирование. Формы многопрограммной работы

Мультипрограммирование призвано повысить эффективность использования вычислительной системы [3], [7]. Однако эффективность может пониматься по-разному. Наиболее характерными показателями эффективности вычислительных систем являются:

- пропускная способность – количество задач, выполняемых системой в единицу времени;

- удобство работы пользователей, заключающееся, в частности, в том, что они могут одновременно работать в интерактивном режиме с несколькими приложениями на одной машине;
- реактивность системы – способность выдерживать заранее заданные (возможно, очень короткие) интервалы времени между запуском программы и получением конечного результата.

В зависимости от выбора одного из этих показателей эффективности ОС делятся на системы пакетной обработки, системы разделения времени и системы реального времени (некоторые ОС могут поддерживать одновременно несколько режимов).

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов [1]. Максимальная пропускная способность компьютера достигается в этом случае минимизацией простоев его устройств и прежде всего процессора. Для достижения этой цели пакет заданий формируется так, чтобы получающаяся мультипрограммная смесь сбалансированно загружала все устройства машины. Например, в такой смеси желательно присутствие задач вычислительного характера и с интенсивным вводом-выводом. Однако в этом случае трудно гарантировать сроки выполнения того или иного задания.

В благоприятных случаях общее время выполнения смеси задач меньше, чем суммарное время их последовательного выполнения. При этом времени выполнения отдельной задачи может быть затрачено больше, чем при монопольном ее выполнении (рис. 5.4).

В системах разделения времени пользователям (в частном случае – одному) предоставляется возможность интерактивной работы сразу с несколькими приложениями. Для этого каждое приложение должно регулярно получать возможность "общения" с пользователем. Эта проблема решается за счет того, что ОС принудительно периодически приостанавливает приложения, не дожидаясь, когда они "добровольно" освободят процессор.

Рис. 5.4. Иллюстрация эффекта мультипрограммирования

Всем приложениям попеременно выделяются кванты времени процессора, таким образом, пользователи, запустившие программы на выполнение, получают возможность поддерживать с ними диалог (рис. 5.5) со своего терминала. Если время кванта выбрано достаточно небольшим, то у всех пользователей складывается впечатление единоличной работы на машине.

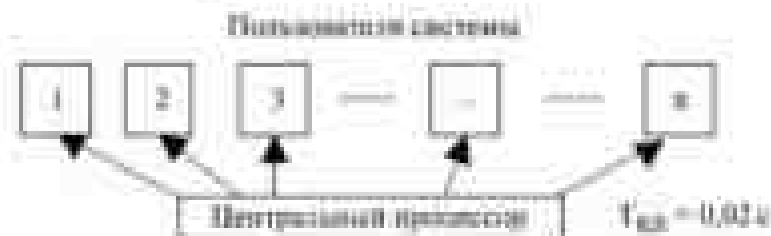


Рис. 5.5. Система разделения времени

Системы реального времени предназначены для управления техническими объектами (спутник, ракета, атомные электростанции,

станки, научные установки и др.), технологическими процессами (гальванические линии, доменный процесс и т.п.), системами обслуживания разного рода (резервирование авиабилетов, оплата покупок и счетов и др.). Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная программа управления объектом. В противном случае возможны нежелательные последствия вплоть до аварии.

Критерием эффективности ОС в этом случае является способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата. Это время называется временем реакции системы, а соответствующее свойство – реактивностью. Требования ко времени реакции зависят от специфики управляемого объекта или процесса. В системах реального времени мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ решения функциональных задач управления объектом или процессом. Выбор программы на выполнение осуществляется по прерываниям (исходя из текущего состояния объекта) или в соответствии с расписанием плановых работ.

В системе реального времени обычно закладывается запас вычислительной мощности на случай пиковой нагрузки, а также принимаются меры обеспечения высокой надежности работы системы (резервирование, дублирование, троирование с мажоритарным элементом и др.).

Интересная форма мультипрограммной работы связана с мультипроцессорной обработкой. Мультипроцессорная обработка – это способ организации вычислительного процесса в системе с несколькими процессорами, при котором несколько задач (процессов, потоков) могут одновременно выполняться на разных процессорах системы. Концепция мультипроцессирования не нова, она известна с 70-х годов, однако стала доступной в широком масштабе лишь в последнее десятилетие, особенно с появлением многопроцессорных ПК (часто в качестве серверов ЛВС).

В отличие от мультипрограммной обработки, в мультипроцессорных системах несколько задач выполняются одновременно, т.к. имеется несколько процессоров. Однако это не исключает мультипрограммной

обработки на каждом процессоре. При этом резко усложняются все алгоритмы управления ресурсами, т.е. операционная система. Современные ОС, как правило, поддерживают мультипроцессирование (Sun Solaris 2.x, Santa Cruz Operation Open Server 3.x, OS/2, Windows NT/2000/2003/XP, NetWare, начиная с версии 4.1 и др.).

Мультипроцессорные системы часто характеризуют как симметричные и как несимметричные. Эти термины относятся, с одной стороны, к архитектуре вычислительной системы, а с другой – к способу организации вычислительного процесса.

Симметричная архитектура мультипроцессорной системы предполагает однотипность и единообразие включения процессоров и большую разделяемую между этими процессорами память. Масштабируемость, т.е. возможность наращивания числа процессоров, в данном случае ограничена, т.к. все они используют одну и ту же оперативную память и, следовательно, должны располагаться в одном корпусе. В симметричных архитектурах вычислительных систем легко реализуется симметричное мультипроцессирование общей для всех процессоров операционной системой. При этом все процессоры равноправно участвуют и в управлении вычислительным процессом, и в выполнении прикладных задач. Разные процессоры могут в какой-то момент времени одновременно обслуживать как разные, так и одинаковые модули общей ОС. Для этого программы ОС должны быть реентерабельными (повторновходимыми).

Операционная система полностью децентрализована. Ее модули выполняются на любом доступном процессоре. Как только процессор завершает выполнение очередной задачи, он передает управление планировщику задач. Последний выбирает из общей для всех процессоров системной очереди задачу, которая будет выполняться на данном процессоре следующей.

В вычислительных системах с асимметричной архитектурой процессоры могут быть различными как по характеристикам (производительность, система команд), так и по функциональной роли в работе системы. Например, могут быть выделены процессоры для вычислений, ввода-вывода и др. Эта неоднородность ведет к структурным отличиям во фрагментах системы, содержащих разные

процессоры (разные схемы подключения, наборы периферийных устройств, способы взаимодействия процессоров с устройствами и др.).

Масштабирование в таких системах реализуется иначе, поскольку отсутствует требование единого корпуса. Система может состоять из нескольких устройств, каждое из которых содержит один или несколько процессоров. Масштабирование в данном случае называют горизонтальным, а мультипроцессорную систему – кластерной. В кластерной системе может быть реализовано только асимметричное мультипроцессирование с организацией вычислительного процесса по принципу "ведущий – ведомый". Этот наиболее простой способ может быть использован и в вычислительных системах с симметричной архитектурой. В таких системах ОС работает на одном процессоре, который называется ведущим и организует централизованное управление вычислительным процессом и распределением всех ресурсов системы.

5.3. Управление процессами и потоками

Одной из основных подсистем любой современной мультипрограммной ОС, непосредственно влияющей на функционирование компьютера, является подсистема управления процессами и потоками. Основные функции этой подсистемы [10, 12, 17]:

- создание процессов и потоков;
- обеспечение процессов и потоков необходимыми ресурсами;
- изоляция процессов;
- планирование выполнения процессов и потоков (вообще, следует говорить и о планировании заданий);
- диспетчеризация потоков;
- организация межпроцессного взаимодействия;
- синхронизация процессов и потоков;
- завершение и уничтожение процессов и потоков.

К созданию процесса приводят пять основных событий:

1. инициализация ОС (загрузка);

2. выполнение запроса работающего процесса на создание процесса;
3. запрос пользователя на создание процесса, например, при входе в систему в интерактивном режиме;
4. инициирование пакетного задания;
5. создание операционной системой процесса, необходимого для работы каких-либо служб.

Обычно при загрузке ОС создаются несколько процессов. Некоторые из них являются высокоприоритетными процессами, обеспечивающими взаимодействие с пользователями и выполняющими заданную работу. Остальные процессы являются фоновыми, они не связаны с конкретными пользователями, но выполняют особые функции – например, связанные с электронной почтой, Web-страницами, выводом на печать, передачей файлов по сети, периодическим запуском программ (например, *дефрагментации дисков*) и т.д. Фоновые процессы называют демонами.

Новый процесс может быть создан по запросу текущего процесса. Создание новых процессов полезно в тех случаях, когда выполняемую задачу проще всего сформировать как набор связанных, но, тем не менее, независимых взаимодействующих процессов. В интерактивных системах пользователь может запустить программу, набрав на клавиатуре команду или дважды щелкнув на значке программы. В обоих случаях создается новый процесс и запуск в нем программы. В системах пакетной обработки на мэйнфреймах пользователи посылают задание (возможно, с использованием удаленного доступа), а ОС создает новый процесс и запускает следующее задание из очереди, когда освобождаются необходимые ресурсы.

С технической точки зрения во всех перечисленных случаях новый процесс формируется одинаково: текущий процесс выполняет системный запрос на создание нового процесса. Подсистема управления процессами и потоками отвечает за обеспечение процессов необходимыми ресурсами. ОС поддерживает в памяти специальные информационные структуры, в которые записывает, какие ресурсы выделены каждому процессу. Она может назначить процессу ресурсы в единоличное пользование или совместное пользование с другими процессами. Некоторые из ресурсов выделяются процессу при его создании, а некоторые – динамически по запросам во время

выполнения. Ресурсы могут быть выделены процессу на все время его жизни или только на определенный период. При выполнении этих функций подсистема управления процессами взаимодействует с другими подсистемами ОС, ответственными за управление ресурсами, такими как подсистема управления памятью, подсистема ввода-вывода, файловая система.

Для того чтобы процессы не могли вмешаться в распределение ресурсов, а также не могли повредить коды и данные друг друга, важнейшей задачей ОС является изоляция одного процесса от другого. Для этого операционная система обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса.

В ОС, где существуют процессы и потоки, процесс рассматривается как заявка на потребление всех видов ресурсов, кроме одного – процессорного времени. Этот важнейший ресурс распределяется операционной системой между другими единицами работы – потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд. Переход от выполнения одного потока к другому осуществляется в результате планирования и диспетчеризации. Работа по определению момента, в который необходимо прервать выполнение текущего потока, и потока, которому следует предоставить возможность выполняться, называется планированием. Планирование потоков осуществляется на основе информации, хранящейся в описателях процессов и потоков. При планировании принимается во внимание приоритет потоков, время их ожидания в очереди, накопленное время выполнения, интенсивность обращения к вводу-выводу и другие факторы.

Диспетчеризация заключается в реализации найденного в результате планирования решения, т.е. в переключении процессора с одного потока на другой. Диспетчеризация проходит в три этапа:

- сохранение контекста текущего потока;
- загрузка контекста потока, выбранного в результате планирования;
- запуск нового потока на выполнение.

Когда в системе одновременно выполняется несколько независимых задач, возникают дополнительные проблемы. Хотя потоки возникают и выполняются синхронно, у них может возникнуть необходимость во взаимодействии, например, при обмене данными. Для общения друг с другом процессы и потоки могут использовать широкий спектр возможностей: каналы (в UNIX), почтовые ящики (Windows), вызов удаленной процедуры, сокеты (в Windows соединяют процессы на разных машинах). Согласование скоростей потоков также очень важно для предотвращения эффекта "гонок" (когда несколько потоков пытаются изменить один и тот же файл), взаимных блокировок и других коллизий, которые возникают при совместном использовании ресурсов.

Синхронизация потоков является одной из важнейших функций подсистемы управления процессами и потоками. Современные операционные системы предоставляют множество механизмов синхронизации, включая семафоры, мьютексы, критические области и события. Все эти механизмы работают с потоками, а не с процессами. Поэтому когда поток блокируется на семафоре, другие потоки этого процесса могут продолжать работу.

Каждый раз, когда процесс завершается, – а это происходит благодаря одному из следующих событий: обычный выход, выход по ошибке, выход по неисправимой ошибке, уничтожение другим процессом – ОС предпринимает шаги, чтобы "зачистить следы" его пребывания в системе. Подсистема управления процессами закрывает все файлы, с которыми работал процесс, освобождает области оперативной памяти, отведенные под коды, данные и системные информационные структуры процесса. Выполняется коррекция всевозможных очередей ОС и списков ресурсов, в которых имелись ссылки на завершаемый процесс.

Как уже отмечалось, чтобы поддержать мультипрограммирование, ОС должна оформить для себя те внутренние единицы работы, между которыми будет разделяться процессор и другие ресурсы компьютера. Возникает вопрос: в чем принципиальное отличие этих единиц работы, какой эффект мультипрограммирования можно получить от их применения и в каких случаях эти единицы работ операционной системы следует создавать?

Очевидно, что любая работа вычислительной системы заключается в

выполнения некоторой программы. Поэтому и с процессом, и с потоком связывается определенный программный код, который оформляется в виде исполняемого модуля. В простейшем случае процесс состоит из одного потока, и в некоторых современных ОС сохранилось такое положение. Мультипрограммирование в таких ОС осуществляется на уровне процессов. При необходимости взаимодействия процессы обращаются к операционной системе, которая, выполняя функции посредника, предоставляет им средства межпроцессной связи – каналы, почтовые ящики, разделяемые секции памяти и др.

Однако в системах, в которых отсутствует понятие потока, возникают проблемы при организации параллельных вычислений в рамках процесса. А такая необходимость может возникать. Дело в том, что отдельный процесс никогда не может быть выполнен быстрее, чем в однопрограммном режиме. Однако приложение, выполняемое в рамках одного процесса, может обладать внутренним параллелизмом, который, в принципе, мог бы ускорить его решение. Если, например, в программе предусмотрено обращение к внешнему устройству, то на время этой операции можно не блокировать выполнение всего процесса, а продолжить вычисления по другой ветви программы.

Параллельное выполнение нескольких работ в рамках одного интерактивного приложения повышает эффективность работы пользователя. Так, при работе с текстовым редактором желательно иметь возможность совмещения набора нового текста с такими продолжительными операциями, как переформатирование значительной части текста, сохранение его на локальном или удаленном диске.

Нетрудно представить будущую версию компилятора, способную автоматически компилировать файлы исходного кода в паузах, возникающих при наборе текста программы. Тогда предупреждения и сообщения об ошибках появлялись бы в режиме реального времени, и пользователь тут же видел бы, в чем он ошибся. Современные электронные таблицы пересчитывают данные в фоновом режиме, как только пользователь что-либо изменил. Текстовые процессоры разбивают текст на страницы, проверяют его на орфографические и грамматические ошибки, печатают в фоновом режиме, сохраняют текст

каждые несколько минут и т.д. Во всех этих случаях потоки используются как средство распараллеливания вычислений.

Эти задачи можно было бы возложить на программиста, который должен был бы написать программу-диспетчер, реализующую параллелизм в рамках одного процесса. Однако это весьма сложно, да и сама программа получилась бы весьма запутанной и сложной в отладке.

Другим решением является создание для одного приложения нескольких процессов для каждой из параллельных работ. Однако использование для создания процессов стандартных средств ОС не позволяет учесть тот факт, что процессы решают единую задачу и имеют много общего: работают с одними и теми же данными, используют один и тот же кодовый сегмент, имеют одни и те же права доступа к ресурсам вычислительной системы. А операционная система при таком подходе будет рассматривать эти процессы наравне со всеми остальными процессами и обеспечивать их изоляцию друг от друга. В данном случае это будет не только бесполезная, но и вредная работа, затрудняющая обмен данными между различными частями приложения. Кроме того, на создание каждого процесса ОС тратит определенные системные ресурсы, которые в данном случае неоправданно дублируются – каждому процессу выделяется собственное виртуальное адресное пространство, физическая память, закрепляются устройства ввода-вывода и т.п.

Из изложенного следует вывод, что операционной системе наряду с процессами нужен другой механизм распараллеливания вычислений, который учитывал бы тесные связи между отдельными ветвями вычислений одного и того же приложения. Для этих целей современные ОС предлагают механизм многопоточной обработки (multithreading).

Понятию "поток" соответствует последовательный переход процессора от одной команды к другой. Процессу ОС назначает адресное пространство и набор ресурсов, которые совместно используются всеми его потоками. В отличие от процессов, которые принадлежат, вообще говоря, конкурирующим приложениям, все потоки одного процесса всегда принадлежат одному приложению, поэтому ОС изолирует потоки в гораздо меньшей степени, чем процессы в традиционной

мультипрограммной системе. Все потоки одного процесса используют общие файлы, таймеры, устройства, одну и ту же область оперативной памяти, одно и то же адресное пространство.

Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждый поток может иметь доступ к любому виртуальному адресу, один поток может задействовать стек другого потока. Между потоками одного процесса нет полной защиты, во-первых, потому что это невозможно, и во-вторых, потому что не нужно. Чтобы организовать взаимодействие и обмен данными, потокам не требуется обращаться к ОС, им достаточно использовать общую память – один поток записывает данные, а другой читает их. С другой стороны, потоки разных процессов по-прежнему хорошо защищены друг от друга.

Таким образом, мультипрограммирование более эффективно на уровне потоков, а не процессов. Еще больший эффект многопоточной обработки достигается в мультипроцессорных системах, в которых потоки могут выполняться на разных процессорах действительно параллельно.

5.4. Создание процессов и потоков. Модели процессов и потоков

Создать процесс – это, прежде всего, создать описатель процесса: несколько информационных структур, содержащих все сведения (атрибуты) о процессе, необходимые операционной системе для управления им. В число таких сведений могут входить идентификатор процесса, данные о расположении в памяти исполняемого модуля, степень привилегированности процесса (приоритет и права доступа) и т.п.

Примерами таких описателей процесса являются [10, 17]:

- блок управления задачей (TCB – *Task Control Block*) в OS/360;
- управляющий блок процесса (*PCB* – *Process Control Block*) в OS/2;
- дескриптор процесса в UNIX;
- объект-процесс (object-process) в Windows NT/2000/2003.

Создание процесса включает загрузку кодов и данных исполняемой программы данного процесса с диска в оперативную память. Для этого нужно найти эту программу на диске, перераспределить оперативную память и выделить память исполняемой программе нового процесса. Кроме того, при работе программы обычно используется стек, с помощью которого реализуются вызовы процедур и передача параметров.

Множество, в которое входит программа, данные, стек и атрибуты процесса, называется образом процесса.

Типичные элементы образа процесса приведены ниже.

Информация	Описание
Данные пользователя	Изменяемая часть пользовательского адресного пространства (данные программы, пользовательский стек, модифицируемый код)
Пользовательская программа	Программа, которую необходимо выполнить
Системный стек	Один или несколько системных стеков для хранения параметров и адресов вызова процедур и системных служб
Управляющий блок процесса	Данные, необходимые операционной системе для управления процессом

Местонахождение образа процесса зависит от используемой схемы управления памятью. В большинстве современных ОС с виртуальной памятью образ процесса состоит из набора блоков (сегменты, страницы или их комбинации), не обязательно расположенных последовательно. Такая организация памяти позволяет иметь в основной памяти лишь часть образа процесса (активная часть), в то время как во вторичной памяти находится полный образ. Когда в основную память загружается часть образа, она туда не переносится, а копируется. Однако если часть образа в основной памяти модифицируется, она должна быть скопирована на диск.

При управлении процессами ОС использует два основных типа информационных структур: блок управления процессом (дескриптор

процесса) и контекст процесса. Дескрипторы процессов объединяются в таблицу процессов, которая размещается в области ядра. На основании информации, содержащейся в таблице процессов, ОС осуществляет планирование и синхронизацию процессов.

В дескрипторе (блоке управления) процесса содержится такая информация о процессе, которая необходима ядру в течение всего жизненного цикла процесса независимо от того, находится он в активном или пассивном состоянии и находится ли образ в оперативной памяти или на диске. Эту информацию можно разделить на три категории:

- информация по идентификации процесса;
- информация по состоянию процесса;
- информация, используемая при управлении процессом.

Каждому процессу присваивается числовой идентификатор, который может быть просто индексом в первичной таблице процессов. В любом случае должно существовать некоторое отображение, позволяющее операционной системе найти по идентификатору процесса соответствующие ему таблицы. При создании нового процесса идентификаторы указывают родительский и дочерние процессы. В операционных системах, не поддерживающих иерархию процессов, например, в Windows 2000, все созданные процессы равноправны, но один из 10-ти параметров, возвращаемых вызывающему (родительскому) процессу, представляет собой дескриптор нового процесса. Кроме того, процессу может быть присвоен идентификатор пользователя, который указывает, кто из пользователей отвечает за данное задание.

Информация по состоянию и управлению процессом включает следующие основные данные:

- состояние процесса, определяющее готовность процесса к выполнению (выполняющийся, готовый к выполнению, ожидающий какого-либо события, приостановленный);
- данные о приоритете (текущий приоритет, по умолчанию, максимально возможный);
- информация о событиях – идентификация события, наступление

которого позволит продолжить выполнение процесса;
- указатели, позволяющие определить расположение образа процесса в оперативной памяти и на диске;
- указатели на другие процессы (в частности, находящиеся в очереди на выполнение);
- флаги, сигналы и сообщения, имеющие отношение к обмену информацией между двумя независимыми процессами;
- данные о привилегиях, определяющих права доступа к определенной области памяти или возможности выполнять определенные виды команд, использовать системные утилиты и службы;
- указатели на ресурсы, которыми управляет процесс (например, перечень открытых файлов);
- сведения по истории использования ресурсов и процессора;
- информация, связанная с планированием. Эта информация во многом зависит от алгоритма планирования. Сюда относятся, например, такие данные, как время ожидания или время, в течение которого процесс выполнялся при последнем запуске, количество выполненных операций ввода-вывода и др.

Контекст процесса содержит информацию, позволяющую системе приостанавливать и возобновлять выполнение процесса с прерванного места.

В контексте процесса содержится следующая основная информация [10]:

- содержимое регистров процессора, доступных пользователю;
- содержимое счетчика команд;
- состояние управляющих регистров и регистров состояния;
- коды условий, отражающие результат выполнения последней арифметической или логической операции (например, знак равенства нулю, переполнения);
- указатели вершин стеков, хранящие параметры и адреса вызова процедур и системных служб.

Следует заметить, что часть этой информации, известная как "слово состояния программы" (*Program Status Word – PSW*), фиксируется в специальном регистре процессора (например, в регистре EFLAGS в

процессора Pentium).

Самую простую модель процесса можно построить исходя из того, что в любой момент времени процесс либо выполняется, либо не выполняется, т.е. имеет только два состояния. Если бы все процессы были бы всегда готовы к выполнению, то очередь по этой схеме могла бы работать вполне эффективно. Такая очередь работает по принципу обработки в порядке поступления, а процессор обслуживает имеющиеся в наличии процессы круговым методом (*Round-robin*). Каждому процессу отводится определенный промежуток времени, по истечении которого он возвращается в очередь.

Однако в таком простом примере подобная реализация не является адекватной: часть процессов готова к выполнению, а часть заблокирована, например, по причине ожидания ввода-вывода. Поэтому при наличии одной очереди диспетчер не может просто выбрать для выполнения первый процесс из очереди. Перед этим он должен будет просматривать весь список, отыскивая незаблокированный процесс, который находится в очереди дольше других. Отсюда представляется естественным разделить все невыполняющиеся процессы на два типа: готовые к выполнению и заблокированные. Полезно добавить еще два состояния, как показано на рис. 5.6.

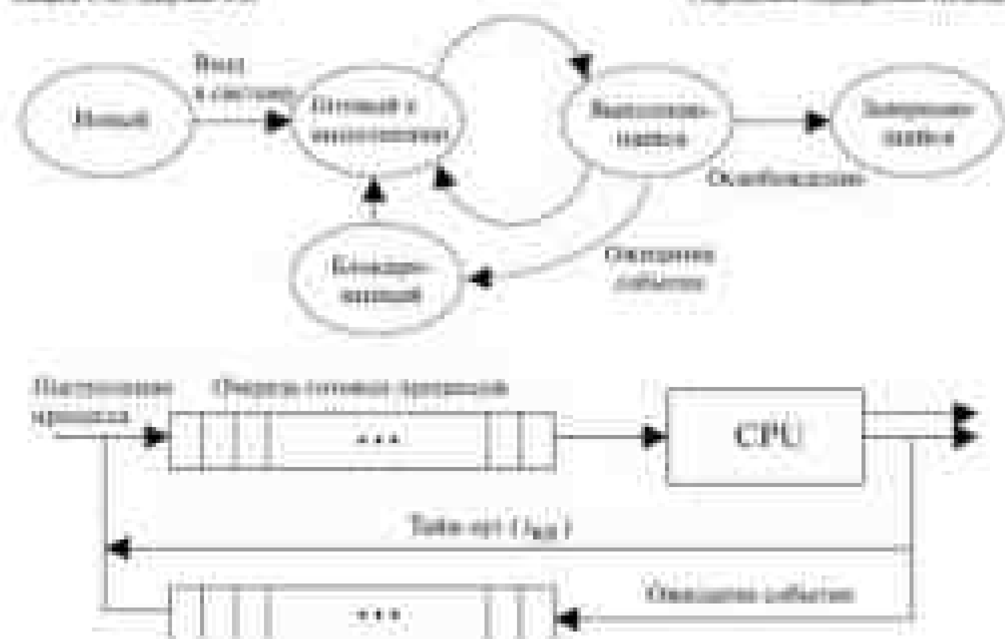


Рис. 5.6. Состояния процесса

В чем достоинства и недостатки такой модели и как устранить эти недостатки? Поскольку процессор работает намного быстрее выполнения операций ввода-вывода, то вскоре все находящиеся в памяти процессы оказываются в состоянии ожидания ввода-вывода. Таким образом, процессор может простаивать даже в многозадачной системе. Что делать? Можно увеличить емкость основной памяти, чтобы в ней умещалось больше процессов.

Но такой подход имеет два недостатка: во-первых, возрастает стоимость памяти, а во-вторых, "аппетит" программиста в использовании памяти возрастает пропорционально ее объему, так что увеличение объема памяти приводит к увеличению размера процессов, а не к росту их числа. Другое решение проблемы – свопинг, перенос части процессов из оперативной памяти на диск и загрузка другого процесса из очереди приостановленных (но не *блокированных!*) процессов, находящихся во внешней памяти. На этом мы прервем рассмотрение модели процессов и их выполнения. Как уже отмечалось, более эффективными являются многопоточные системы. В таких системах при создании процесса ОС создается для каждого процесса минимум один поток выполнения.

При создании потока, так же как и при создании процесса, ОС генерирует специальную информационную структуру – описатель потока, который содержит идентификатор потока, данные о правах доступа и приоритете, о состоянии потока и другую информацию. Описатель потока можно разделить на две части: атрибуты блока управления и контекст потока. Заметим, что в случае многопоточной системы процессы контекста не имеют, так как им не выделяется процессор.

Есть два способа реализации пакета потоков [17]:

- в пространстве пользователя или на уровне пользователя (User-level threads – ULT);
- в ядре или на уровне ядра (kernel-level threads – KLT).

Рассмотрим эти способы, их преимущества и недостатки.

В программе, полностью состоящей из ULT-потоков, все действия по управлению потоками выполняются самим приложением. Ядро о потоках ничего не знает и управляет обычными однопоточными процессами (рис. 5.7).

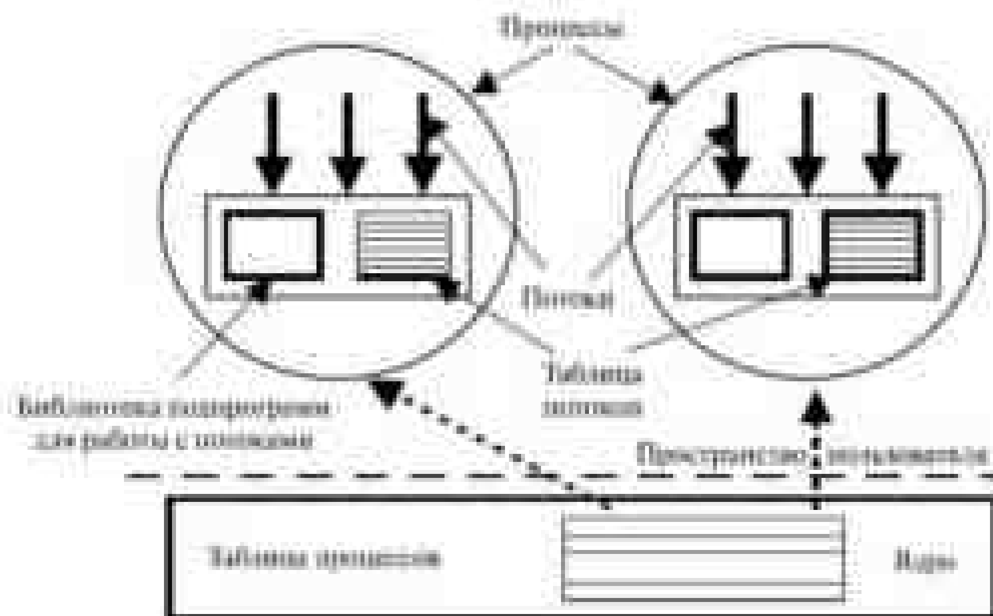


Рис. 5.7. Потоки в пространстве пользователя

Наиболее явное преимущество этого подхода состоит в том, что пакет потоков на уровне пользователя можно реализовать даже в ОС, не поддерживающей потоки.

Если управление потоками происходит в пространстве пользователя, каждому процессу необходима собственная таблица потоков. Она аналогична таблице процессов с той лишь разницей, что отслеживает такие характеристики потоков, как счетчик команд, указатель вершины стека, регистры состояния и т. п. Когда поток переходит в состояние готовности или блокировки, вся информация, необходимая для повторного запуска, хранится в таблице потоков.

По умолчанию приложение в начале своей работы состоит из одного потока и его выполнение начинается как выполнение этого потока. Такое приложение вместе с составляющим его потоком размещается в одном процессе, который управляется ядром. Выполняющийся поток может породить новый поток, который будет выполняться в пределах того же процесса. Новый поток создается с помощью вызова специальной подпрограммы из библиотеки, предназначенной для работы с потоками. Передача управления этой программе происходит в результате вызова соответствующей процедуры.

Таких процедур может быть по крайней мере четыре: *thread-create*, *thread-exit*, *thread-wait* и *thread-yield*, но обычно их больше. Библиотека подпрограмм для работы с потоками создает структуру данных для нового потока, а потом передает управление одному из готовых к выполнению потоков данного процесса, руководствуясь некоторым алгоритмом планирования. Когда управление переходит к библиотечной программе, контекст текущего процесса сохраняется в таблице потоков, а когда управление возвращается к потоку, его контекст восстанавливается. Все эти события происходят в пользовательском пространстве в рамках одного процесса. *Ядро* даже "не подозревает" об этой деятельности и продолжает осуществлять планирование процесса как единого целого и приписывать ему единое состояние выполнения.

Использование потоков на уровне пользователя имеет следующие преимущества [17]:

1. высокая производительность, поскольку для управления потоками процессу не нужно переключаться в режим ядра и обратно. Процедура, сохраняющая информацию о потоке, и планировщик являются локальными процедурами, их вызов существенно более эффективен, чем вызов ядра;
2. имеется возможность использования различных алгоритмов планирования потоков в различных приложениях (процессах) с учетом их специфики;
3. использование потоков на пользовательском уровне применимо для любой операционной системы. Для их поддержки в ядро системы не требуется вносить каких-либо изменений.

Однако имеются и недостатки по сравнению с использованием потоков на уровне ядра:

- в типичной ОС многие системные вызовы являются блокирующими. Когда в потоке, работающем на пользовательском уровне, выполняется системный вызов, блокируется не только этот поток, но и все потоки того процесса, к которому он относится;
- в стратегии с наличием потоков только на пользовательском уровне приложение не может воспользоваться преимуществом многопроцессорной системы, так как ядро закрепляет за каждым процессом только один процессор. Поэтому несколько потоков одного и того же процесса не могут выполняться одновременно. В сущности, получается мультипрограммирование в рамках одного процесса;
- при запуске одного потока ни один другой поток не будет запущен, пока первый добровольно не отдаст процессор. Внутри одного процесса нет прерываний по таймеру, в результате чего невозможно создать планировщик для поочередного выполнения потоков.

Рассмотрим теперь потоки на уровне ядра. В этом случае в области приложения система поддержки исполнения программ не нужна, нет необходимости и в таблицах потоков в каждом процессе. Вместо этого есть единая таблица потоков, отслеживающая все потоки в системе. Если потоку необходимо создать новый поток или завершить имеющийся, он выполняет запрос ядра, который создает или завершает

потока, внося изменения в таблицу потоков (рис. 5.8).

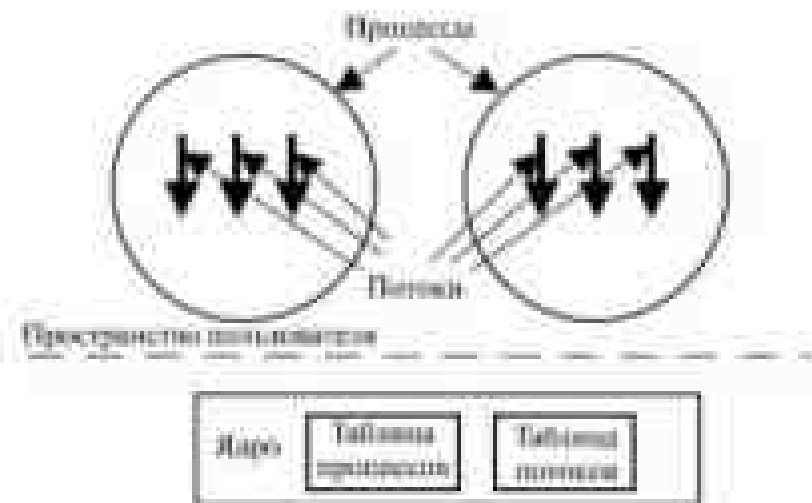


Рис. 5.8. Потоки в пространстве ядра

Любое приложение можно запрограммировать как многопоточное, при этом все потоки приложения поддерживаются в рамках единого процесса. Ядро поддерживается информацией контекста процесса как единого целого, а также контекстами каждого отдельного потока процесса. Планирование осуществляется ядром, исходя из состояния потоков. С помощью такого подхода удается избавиться от основных недостатков потоков пользовательского уровня.

Возможно планирование работы нескольких потоков одного и того же процесса на нескольких процессорах;

1. реализуется мультипрограммирование в режимах нескольких процессов (вообще – всех);
2. при блокировке одного из потоков процесса ядро может выбрать для выполнения другой поток этого же процесса;
3. процедуры ядра могут быть многопоточными.

Главный недостаток связан с необходимостью двукратного переключения режимов пользовательский – ядро, ядро – пользовательский для передачи одного потока к другому в рамках одного и того же процесса.

5.5. Планирование заданий, процессов и потоков

Основная цель планирования вычислительного процесса заключается в распределении времени процессора (нескольких процессоров) между выполняющимися заданиями пользователей таким образом, чтобы удовлетворить требованиям, предъявляемым пользователями к вычислительной системе. Такими требованиями могут быть, как это уже отмечалось, пропускная способность, время отклика, загрузка процессора и др.

Все виды планирования, используемые в современных ОС, в зависимости от временного масштаба, делятся на долгосрочное, среднесрочное, краткосрочное и планирование ввода-вывода. Рассматривая частоту работы планировщика, можно сказать, что долгосрочное планирование выполняется сравнительно редко, среднесрочное несколько чаще. Краткосрочный планировщик, называемый часто диспетчер (dispatcher), обычно работает, определяя, какой процесс или поток будет выполняться следующим. Ниже приведен перечень функций, выполняемых планировщиками каждого вида.

Вид планирования	Выполняемые функции
Долгосрочное	Решение о добавлении задания (процесса) в пул выполняемых в системе
Среднесрочное	Решение о добавлении процесса к числу процессов, полностью или частично размещенных в основной памяти
Краткосрочное	Решение о том, какой из доступных процессов (потоков) будет выполняться процессором
Планирование ввода-вывода	Решение о том, какой из запросов процессов (потоков) на операцию ввода-вывода будет выполняться свободным устройством ввода-вывода

Место планирования в графе состояний и переходов процессов показано на рис. 5.9. В большинстве операционных систем универсального назначения планирование осуществляется динамически

(on-line), т.е. решения принимаются во время работы системы на основе анализа текущей ситуации, не используя никаких предложений о мультипрограммной смеси. Найденное оперативно решение в таких условиях редко бывает оптимальным.

Рис. 5.9. Место планирования в графе процессов

Другой тип планирования – статический (предварительный), может быть использован только в специализированных системах с заданным набором задач (заранее определенным), например, в управляющих вычислительных системах или системах реального времени. В этом случае статический планировщик (или предварительный планировщик) принимает решение не во время работы системы, а заранее (off-line). Результатом его работы является расписание – таблица, в которой указано, какому процессу, когда и на какое время должен быть предоставлен процессор. При этом накладные расходы ОС на исполнение расписания значительно меньше, чем при динамическом планировании.

Краткосрочный планировщик (диспетчер) реализует найденное решение, т.е. переключает процессор с одного процесса (потока) на

другой. Он вызывается при наступлении события, которое может приостановить текущий процессор или предоставить возможность прекратить выполнение данного процесса (потока) в пользу другого. Примерами этих событий могут быть:

- прерывание таймера;
- прерывание ввода-вывода;
- вызовы операционной системы;
- сигналы.

Среднесрочное планирование является частью системы свопинга. Обычно решение о загрузке процесса в память принимается в зависимости от степени многозадачности (например, OS MFT, OS MVT). Кроме того, в системе с отсутствием виртуальной памяти среднесрочное планирование тесно связано с вопросами управления памятью.

Диспетчеризация сводится к следующему:

- сохранение контекста текущего потока, который требуется сменить;
- загрузка контекста нового потока, выбранного в результате планирования;
- запуск нового потока на выполнение.

Поскольку переключение контекстов существенно влияет на производительность вычислительной системы, программные модули ОС выполняют эту операцию при поддержке аппаратных средств процессора.

В мультипрограммной системе поток (процесс, если операционная система работает только с процессами) может находиться в одном из трех основных состояний:

- выполнение – активное состояние потока, во время которого поток обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;
- ожидание – пассивное состояние потока, находясь в котором, поток заблокирован по своим внутренним причинам (ждет

осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого потока или освобождения какого-либо необходимого ему ресурса);

- готовность – также пассивное состояние потока, но в этом случае поток заблокирован в связи с внешним по отношению к нему обстоятельством (имеет все требуемые ресурсы, готов выполняться, но процессор занят выполнением другого потока).

В течение своей жизни каждый поток переходит из одного состояния в другое в соответствии с алгоритмом планирования потоков, принятым в данной операционной системе.

В состоянии выполнения в однопроцессорной системе может находиться не более одного потока, а в остальных состояниях – несколько. Эти потоки образуют очереди, соответственно, ожидающих и готовых потоков. Очереди организуются путем объединения в списки описателей отдельных потоков. С самых общих позиций все множество алгоритмов планирования можно разделить на два класса: вытесняющие и не вытесняющие алгоритмы планирования.

Не вытесняющие (*non-preemptive*) алгоритмы основаны на том, что активному потоку позволяется выполняться, пока он сам, по своей инициативе, не отдаст управление операционной системе, для того чтобы она выбрала из очереди готовый к выполнению поток.

Вытесняющие (*preemptive*) алгоритмы – это такие способы планирования потоков, в которых решение о переключении процессора с выполнения одного потока на выполнение другого потока принимается операционной системой, а не активной задачей.

В первом случае механизм планирования распределяется между операционной системой и прикладными программами. Во втором случае функции планирования потоков целиком сосредоточены в операционной системе.

Недостатком первого типа алгоритмов планирования является необходимость разработки таких приложений, которые будут "дружественными" по отношению к другим выполняемым одновременно с ним программам. Для этого в приложениях должны быть

предусмотрены частые передачи управления операционной системе. Крайним проявлением недружественности приложения является его зависание, которое приводит к общему краху системы. Потому распределение функций планирования между ОС и приложениями достаточно сложно в программистском отношении и используется, как правило, в специализированных системах с фиксированным набором задач. В то же время существенным преимуществом невытесняющего планирования является более высокая скорость переключения потоков.

Однако почти во всех ОС (UNIX, Windows NT/2000/2003, OS/2, VAX/VMS и др.) реализованы вытесняющие алгоритмы планирования. В основе многих таких алгоритмов лежит концепция квантования. В соответствии с ней каждому потоку поочередно для выполнения предоставляется ограниченный непрерывный период процессорного времени – квант.

Смена активного потока происходит, если:

- поток завершается и покинул систему;
- произошла ошибка;
- поток перешел в состояние ожидания;
- исчерпан квант времени, отведенный данному потоку.

Поток, который исчерпал свой квант, переводится в состояние готовности и ожидает, когда ему будет предоставлен новый квант процессорного времени, а на выполнение в соответствии с определенным правилом выбирается новый поток из очереди готовых потоков.

Кванты, выделяемые потокам, могут быть равными или различными (типичное значение десятки – сотни мс). Например, первоначально каждому потоку назначается достаточно большой квант, а величина каждого следующего кванта уменьшается до некоторой заранее заданной величины. В таком случае преимущество получают короткие задачи, которые успевают выполниться в течение первого кванта (второго и т.д.), а длительные вычисления будут проводиться в фоновом режиме.

Некоторые потоки, получив квант времени, используют его не полностью, например, из-за необходимости выполнить ввод или вывод

данных. В результате может возникнуть ситуация, когда потоки с интенсивным вводом-выводом используют только небольшую часть выделенного им процессорного времени. Можно исправить эту "несправедливость", изменив алгоритм планирования, например, так: создать две очереди потоков, очередь 1 – для потоков, которые пришли в состояние готовности в результате исчерпания кванта времени, и очередь 2 – для потоков, у которых завершилась операция ввода-вывода. При выборе потока для выполнения сначала просматривается вторая очередь, и если она пуста, квант выделяется потоку из первой очереди.

Отметим три замечания об алгоритмах, основанных на квантовании.

Первое. Переключение контекстов потоков связано с потерями процессорного времени, которые не зависят от величины кванта, но зависят от частоты переключения. Поэтому чем больше квант, тем меньше суммарные затраты процессорного времени на переключение потоков.

Второе. С увеличением кванта может быть ухудшено качество обслуживания пользователей, связанное с ростом времени реакции системы.

Третье. В алгоритмах, основанных на квантовании, ОС не имеет никаких сведений о решаемых задачах (длинные или короткие, интенсивен "ввод-вывод" или нет, важно быстрое исполнение или нет и т.п.). Дифференциация обслуживания при квантовании базируется на "истории существования" потока в системе.

Важной концепцией, лежащей в основе многих вытесняющих алгоритмов планирования, является приоритетное обслуживание. Оно предполагает наличие у потоков некоторой изначально известной характеристики – приоритета, на основании которого определяется порядок выполнения потоков. Чем выше приоритет, тем выше привилегии потока, тем меньше времени поток находится в очередях. Приоритет задается числом (целым или дробным, положительным или отрицательным).

В большинстве ОС, поддерживающих потоки, приоритет потока связан с приоритетом процесса, в рамках которого выполняется поток. Приоритет процесса назначается операционной системой при его

создании, его значение включается в описатель процесса и используется при назначении приоритета потокам этого процесса. При назначении приоритетов вновь созданному процессу ОС учитывается, является ли этот процесс системным или прикладным, каков статус пользователя, запустившего процесс (администратор, пользователь, гость и т.п.), было ли явное указание пользователя на присвоение процессу определенного уровня приоритета. Поток может быть инициирован не только по команде пользователя, но и в результате выполнения системного вызова другим потоком. В этом случае ОС учитывает значения параметров системного вызова.

Изменения приоритета могут происходить по инициативе самого потока, когда он обращается с соответствующим вызовом к ОС, или по инициативе пользователя, когда он выполняет соответствующую команду. Кроме этого, сама ОС может изменять приоритеты потоков в зависимости от ситуации, складывающейся в системе. В последнем случае приоритеты называются динамическими в отличие от неизменяемых, фиксированных приоритетов. Возможности пользователей влиять на приоритеты процессов и потоков ограничены ОС. Обычно это разрешается администраторам, и то в определенных пределах. В большинстве случаев ОС присваивает приоритеты потокам по умолчанию.

Существует две разновидности *приоритетного планирования*: с относительными и абсолютными приоритетами. В обоих случаях на выполнение выбирается поток, имеющий наивысший приоритет. Но определение момента смены активного потока решается по-разному. В системах с относительными приоритетами активный поток выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ожидания (по вводу-выводу, например), или не завершится, или не произойдет ошибка. В системах с абсолютными приоритетами выполнение активного потока прерывается еще и по причине появления потока, имеющего более высокий приоритет, чем у активного потока. В этом случае прерванный поток переходит в состояние готовности.

В качестве примера рассмотрим организацию приоритетного обслуживания в Windows 2000/2003/XP/Vista. Здесь приоритеты организованы в виде двух групп, или классов: реального времени и

переменные. Каждая из групп состоит из 16 уровней приоритетов (рис. 5.10). Потоки, требующие немедленного внимания, находятся в классе реального времени, который включает такие функции, как осуществление коммуникаций и задачи реального времени [17].

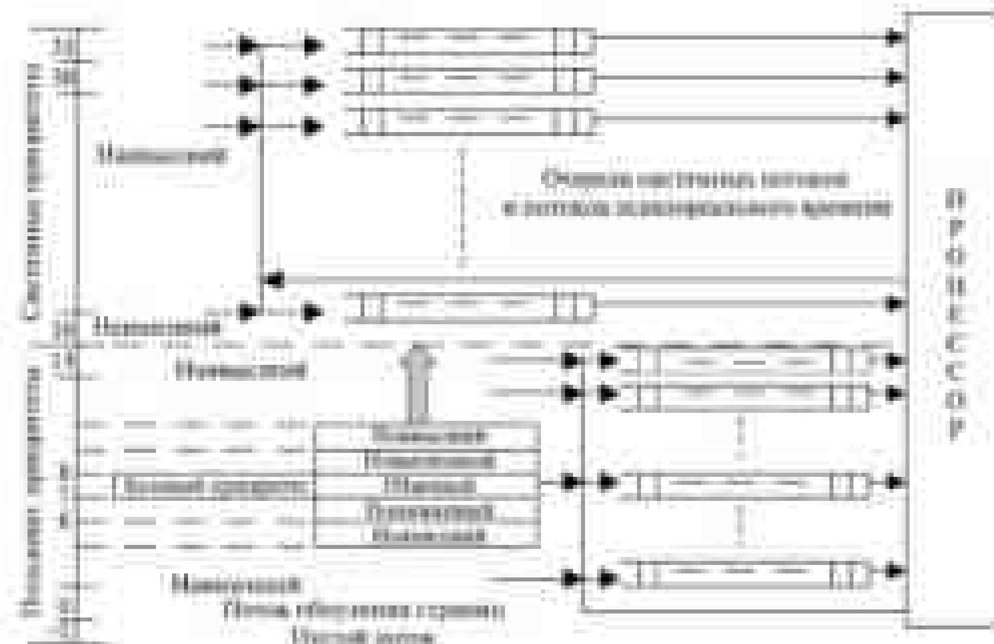


Рис. 5.10. Планирование в Windows

В целом, поскольку W2K использует вытесняющий планировщик с учетом приоритетов, потоки с приоритетами реального времени имеют преимущество по отношению к прочим потокам. В однопроцессорной системе, когда становится готовым к выполнению поток с более высоким приоритетом, чем выполняющийся в настоящий момент, текущий поток вытесняется и начинает выполняться поток с более высоким приоритетом.

Приоритеты из разных классов обрабатываются несколько по-разному. В классе приоритетов реального времени все потоки имеют фиксированный приоритет (от 16 до 31), который никогда не изменяется, и все активные потоки с определенным уровнем приоритета располагаются в круговой очереди данного класса (кв=20 мс для W2K Professional, 120 мс - для однопроцессорных серверов).

В классе переменных приоритетов поток начинает работу с базового приоритета процесса, который может принимать значение от 1 до 15. Каждый поток, связанный с процессом имеет, свой базовый приоритет, равный базовому приоритету процесса, или отличающийся от него не более чем на 2 уровня в большую или меньшую сторону. После активации потока его динамический приоритет может колебаться в определенных пределах – он не может упасть ниже наименьшего базового приоритета данного класса, т.е. 15 (для потоков с приоритетом 16 и выше никогда не делается никаких изменений приоритетов).

Когда же увеличивается приоритет потока? Во-первых, когда завершается операция ввода-вывода и освобождается ожидающий ее поток, его приоритет увеличивается, чтобы дать шанс этому потоку запуститься быстрее и снова запустить операцию ввода-вывода. Суть в том, чтобы поддержать занятость устройств ввода-вывода. Величина, на которую увеличивается приоритет, зависит от устройства ввода-вывода. Как правило, это 1 – для диска, 2 – для последовательной линии, 6 – для клавиатуры и 8 – для звуковой карты.

Во-вторых, если поток ждал семафора, мьютекса или другого события, то когда он отпускается, к его приоритету добавляется 2 единицы, если это поток переднего плана (т.е. управляет окном, к которому направляется ввод с клавиатуры), и одна единица – в противном случае. Таким образом, интерактивный процесс получает преимущество перед большим количеством других процессов. Наконец, если поток графического интерфейса пользователя просыпается, потому что стал доступен оконный ввод, он также получает прибавку к приоритету.

Эти увеличения приоритета не вечны. Они незамедлительно вступают в силу, но если поток использует полностью свой следующий квант, он теряет один пункт приоритета. Если он использует еще квант, то перемещается еще на уровень ниже и т.д. вплоть до своего базового уровня.

Последняя модификация алгоритма планирования W2K заключается в том, что когда окно становится окном переднего плана, все его потоки получают более длительные кванты времени (величина прибавки хранится в системном реестре).

Поток, созданный в системе, может находиться в одном из 6 состояний

в соответствии с графом, приведенным на рис. 5.11.

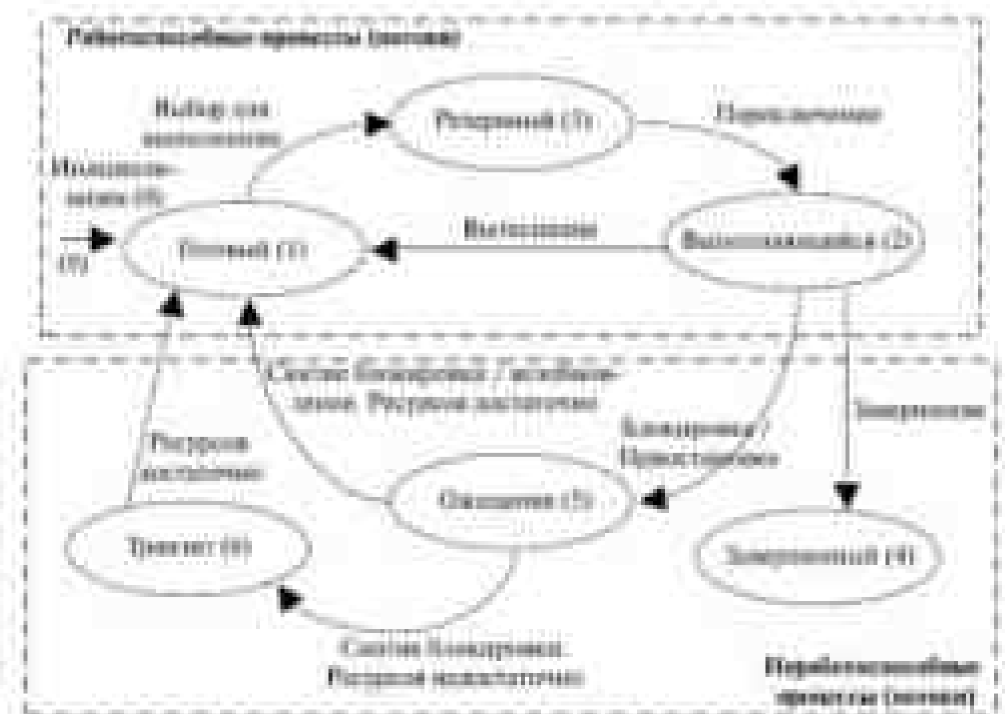


Рис. 5.11. Состояния потоков в Windows

5.6. Взаимодействие и синхронизация процессов и потоков

В мультипрограммных однопроцессорных системах процессы чередуются, обеспечивая эффективное выполнение программ. В многопроцессорных системах возможно не только чередование, но и перекрытие процессов. Обе эти технологии, которые можно рассматривать как примеры параллельных вычислений, порождают одинаковые проблемы. Выполнение процессов и потоков в мультипрограммной среде всегда имеет асинхронный характер – невозможно предсказать относительную скорость выполнения процессов. Момент прерывания потоков, время нахождения их в очередях к разделяемым ресурсам, порядок выбора потоков для выполнения – все эти события являются результатом стечения многих обстоятельств и являются случайными, это справедливо как по

отношению к потокам одного процесса, выполняющим общий программный код, так и по отношению к потокам разных процессов, каждый из которых выполняет собственную программу.

Способы взаимодействия процессов (потоков) можно классифицировать по степени осведомленности одного процесса о существовании другого [33].

1. Процессы не осведомлены о наличии друг друга (например, процессы разных заданий одного или различных пользователей). Это независимые процессы, не предназначенные для совместной работы. Хотя эти процессы и не работают совместно, ОС должна решать вопросы конкурентного использования ресурсов. Например, два независимых приложения могут затребовать доступ к одному и тому же диску или принтеру. ОС должна регулировать такие обращения.
2. Процессы косвенно осведомлены о наличии друг друга (например, процессы одного задания). Эти процессы не обязательно должны быть осведомлены о наличии друг друга с точностью до идентификатора процесса, однако они разделяют доступ к некоторому объекту, например, буферу ввода-вывода, файлу или БД. Такие процессы демонстрируют сотрудничество при разделении общего объекта.
3. Процессы непосредственно осведомлены о наличии друг друга (например, процессы, работающие последовательно или поочередно в рамках одного задания). Такие процессы способны общаться один с другим с использованием идентификаторов процессов и изначально созданы для совместной работы. Эти процессы также демонстрируют сотрудничество при работе.

Таким образом, потенциальные проблемы, связанные с взаимодействием и синхронизацией процессов и потоков, могут быть представлены следующей таблицей.

Степень осведомленности	Взаимосвязь	Влияние одного процесса на другой	Потенциальные проблемы

Процессы не осведомлены друг о друге	Конкуренция	• Результат работы одного процесса не зависит от действий других. • Возможно влияние одного процесса на время работы других	• Взаимоискл • Взаимоблок • Голодание
Процессы косвенно осведомлены о наличии друг друга	Сотрудничество с использованием разделения	• Результат работы одного процесса может зависеть от информации, полученной от других. • Возможно влияние одного процесса на время работы других	• Взаимоискл • Взаимоблок • Голодание • Синхрониз
		• Результат работы одного	

Процессы непосредственно осведомлены о наличии друг друга	Сотрудничество с использованием связи	процесса зависит от информации, полученной от других процессов. • Возможно влияние одного процесса на время работы других	• Взаимоблоки (расходуемы ресурсы) • Голодание

При необходимости использовать один и тот же ресурс параллельные процессы вступают в конфликт (конкурируют) друг с другом. Каждый из процессов не подозревает о наличии остальных и не повергается никакому воздействию с их стороны. Отсюда следует, что каждый процесс не должен изменять состояние любого ресурса, с которым он работает. Примерами таких ресурсов могут быть устройства ввода-вывода, память, процессорное время, часы.

Между конкурирующими процессами не происходит никакого обмена информацией. Однако выполнение одного процесса может повлиять на поведение конкурирующего процесса. Это может, например, выразиться в замедлении работы одного процесса, если ОС выделит ресурс другому процессу, поскольку первый процесс будет ждать завершения работы с этим ресурсом. В предельном случае блокированный процесс может никогда не получить доступ к нужному ресурсу и, следовательно, никогда не сможет завершиться.

В случае конкурирующих процессов (потоков) возможно возникновение трех проблем. Первая из них – необходимость взаимных исключений (*mutual exclusion*). Предположим, что два или большее количество процессов требуют доступ к одному неразделяемому ресурсу, как например принтер (рис. 3.12). О таком ресурсе будем говорить как о *критическом ресурсе*, а о части программы, которая его использует, – как о *критическом разделе* (*critical section*) программы. Крайне важно,

чтобы в критической ситуации в любой момент могла находиться только одна программа. Например, во время печати файла требуется, чтобы отдельный процесс имел полный контроль над принтером, иначе на бумаге можно получить чередование строк двух файлов.

Рис. 5.12. Критическая секция

Осуществление взаимных исключений создает две дополнительные проблемы. Одна из них – взаимоблокировки (deadlock) или тупики. Рассмотрим, например, два процесса – P1 и P2, и два ресурса – R1 и R2. Предположим, что каждому процессу для выполнения части своих функций требуется доступ к обоим ресурсам. Тогда возможно возникновение следующей ситуации: ОС выделяет ресурс R1 процессу P2, а ресурс R2 – процессу P1. В результате каждый процесс ожидает получения одного из двух ресурсов. При этом ни один из них не освобождает уже имеющийся ресурс, ожидая получения второго ресурса для выполнения функций, требующих наличие двух ресурсов. В результате процессы оказываются взаимно заблокированными.

Очень удобно моделировать условия возникновения тупиков, используя направленные графы [17] (предложено Holt, 1972). Графы имеют 2 вида узлов: процессы-кружочки и ресурсы-квадратики. Ребро, направленное от квадрата (ресурса) к кружку (процессу), означает, что ресурс был запрошен, получен и используется. В нашем примере это будет изображено так, как показано на рис. 5.13 а).

Ребро, направленное от процесса (кружка) к ресурсу (квадрату), означает, что процесс в данный момент заблокирован и находится в состоянии ожидания доступа к этому ресурсу. В нашем примере граф

надо достроить, как показано на рис. 5.13 б) или в). Цикл в графе означает наличие взаимной блокировки процессов.

Существует еще одна проблема у конкурирующих процессов – голодание. Предположим, что имеется 3 процесса (P1, P2, P3), каждому из которых периодически требуется доступ к ресурсу R. Представим ситуацию, в которой P1 обладает ресурсом, а P2 и P3 приостановлены в ожидании освобождения ресурса R. После выхода P1 из критического раздела доступ к ресурсу будет получен одним из процессов P2 или P3.

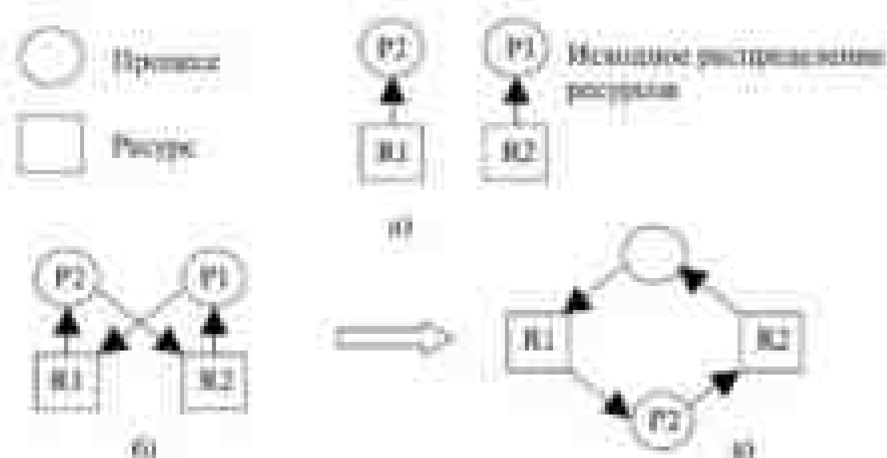


Рис. 5.13. Тупиковая ситуация

Пусть ОС предоставила доступ к ресурсу процессу P3. Пока он работает с ресурсом, доступ к ресурсу вновь требуется процессу P1. В результате по освобождении ресурса R процессом P3 может оказаться, что ОС вновь предоставит доступ к ресурсу процессу P1. Тем временем процессу P3 вновь требуется доступ к ресурсу R. Таким образом, теоретически возможна ситуация, в которой процесс P2 никогда не получит доступ к требуемому ему ресурсу, несмотря на то, что никакой взаимной блокировки в данном случае нет.

Рассмотрим случай сотрудничества с использованием разделения. Этот случай охватывает процессы (потоки), взаимодействующие с другими процессами (потоками), без наличия явной информации о них. Например, несколько потоков могут обращаться к разделяемым

переменным (глобальным) или совместно используемым файлам или базам данных. Поскольку данные хранятся в ресурсах (устройствах памяти), в этом случае также возможны проблемы взаимоблокировок, взаимоисключения и голодания. Единственное отличие в том, что доступ к данным может осуществляться в двух режимах – чтения и записи, и взаимоисключающими должны быть только операции записи.

Однако в этом случае вносится новое требование: синхронизации процессов для обеспечения согласованности данных.

Пусть имеются два процесса, представленные последовательностью неделимых (атомарных) операций:

P: a b c и Q: d e f

где a, b, c, d, e, f – атомарные операции.

При последовательном выполнении активностей мы получаем следующую последовательность атомарных действий:

PQ: a b c d e f

Что произойдет при исполнении этих процессов псевдопараллельно, в режиме разделения времени? Процессы могут расслоиться на неделимые операции с различным их чередованием, то есть может произойти то, что на английском языке принято называть словом interleaving. Возможные варианты чередования:

a b c d e f
a b d c e f
a b d e c f
a b d e f c
a d b c e f
……
d e f a b c

В данном случае атомарные операции активностей могут чередоваться всевозможными способами с сохранением своего порядка расположения внутри процессов. Так как псевдопараллельное выполнение двух процессов приводит к чередованию их неделимых

операций, результат псевдопараллельного выполнения может отличаться от результата последовательного выполнения. Пусть есть два процесса P и Q, состоящие из двух атомарных операций:

P: x=2; y=x-1;   Q: x=3; y=x+1

Что мы получим в результате их псевдопараллельного выполнения, если переменные x и y являются общими для процессов? Легко видеть, что возможны четыре разных набора значений для пары {x, y}: {3, 4}, {2, 1}, {2, 3} и {3, 2}. Будем говорить, что набор процессов детерминирован, если всякий раз при псевдопараллельном исполнении для одного и того же набора входных данных он дает одинаковые выходные данные. В противном случае он недетерминирован. Выше приведен пример недетерминированного набора программ. Понятно, что детерминированный набор активностей можно безбоязненно выполнять в режиме разделения времени. Для недетерминированного набора такое исполнение нежелательно.

Можно ли до получения результатов, заранее, определить, является ли набор активностей детерминированным или нет? Для этого существуют достаточные условия *Бернстайна* [17]. Изложим их применительно к программам с разделяемыми переменными.

Введем наборы входных и выходных переменных программы. Для каждой атомарной операции наборы входных и выходных переменных – это наборы переменных, которые атомарная операция считывает и записывает. Набор входных переменных программы R(P) (R от слова read) суть объединение наборов входных переменных для всех ее неделимых действий. Аналогично, набор выходных переменных программы W(P) (W от слова write) суть объединение наборов выходных переменных для всех ее неделимых действий. Например, для программы

P: x = u + v; y = x * w;

получаем R(P) = {u, v, x, w}, W(P) = {x, y}. Заметим, что переменная x присутствует как в R(P), так и в W(P).

Теперь сформулируем условия *Бернстайна*.

Если для двух данных процессов P и Q:

- пересечение W(P) и W(Q) пусто,
- пересечение W(P) с R(Q) пусто,
- пересечение R(P) и W(Q) пусто,

тогда выполнение P и Q детерминировано.

Если эти условия не соблюдены, возможно, что параллельное выполнение P и Q детерминировано, но возможно, что и нет. Случай двух процессов естественным образом обобщается на их большее количество.

Условия Бернстайна информативны, но слишком жестки. По сути дела, они требуют практически невзаимодействующих процессов. Однако хотелось бы, чтобы детерминированный набор образовывали процессы, совместно использующие информацию и обменивающиеся ею. Для этого нам необходимо ограничить число возможных чередований атомарных операций, исключив некоторые чередования с помощью механизмов синхронизации выполнения программ и обеспечив тем самым упорядоченный доступ программ к некоторым данным.

Про недетерминированный набор программ говорят, что он имеет race condition (состояние гонки, состояние состязания). В приведенном выше примере процессы состязаются за вычисление значений переменных x и y.

Задачу упорядоченного доступа к разделяемым данным (устранение race condition), в том случае, если нам не важна его очередность, можно решить, если обеспечить каждому процессу эксклюзивное право доступа к этим данным. Каждый процесс, обращающийся к разделяемым ресурсам, исключает для всех других процессов возможность одновременного с ним общения с этими ресурсами, если это может привести к недетерминированному поведению набора процессов. Такой прием называется взаимоисключением (mutual exclusion). Если очередность доступа к разделяемым ресурсам важна для получения правильных результатов, то одними взаимоисключениями уже не обойтись.

При сотрудничестве с использованием связи различные процессы принимают участие в общей работе, которая их объединяет. Связь обеспечивает возможность синхронизации, или координации, различных действий процессов. Обычно можно считать, что связь состоит из сообщений определенного вида. Примитивы для отправки и получения сообщений могут быть предоставлены языком программирования или ядром операционной системы.

Поскольку в процессе передачи сообщений не происходит какого-либо совместного использования ресурсов, взаимоисключения не требуется, хотя проблемы взаимоблокировок и голодания остаются актуальными. В качестве примера взаимоблокировки можно привести ситуацию, при которой каждый из двух процессов заблокирован ожиданием сообщения от другого процесса. Голодание можно проиллюстрировать следующим образом. Пусть есть три процесса P1, P2, P3, и те, в свою очередь, пытаются связаться с процессом P1. Может возникнуть ситуация, когда P1 и P2 постоянно связываются *друг с другом*, а P3 остается заблокированным, ожидая связи с процессом P1.

5.7. Методы взаимоисключений

Организация взаимоисключения для критических участков, конечно, позволит избежать возникновения *race condition*, но не является достаточной для правильной и эффективной параллельной работы кооперативных процессов. Сформулируем пять условий, которые должны выполняться для хорошего программного алгоритма организации взаимодействия процессов, имеющих критические участки, если они могут проходить их в произвольном порядке [10, 17].

1. Задача должна быть решена чисто программным способом на обычной машине, не имеющей специальных команд взаимоисключения. При этом предполагается, что основные инструкции языка программирования (такие примитивные инструкции, как load, store, test) являются атомарными операциями.
2. Не должно существовать никаких предположений об относительных скоростях выполняющихся процессов или числе процессоров, на которых они исполняются.

3. Если процесс Pi исполняется в своем критическом участке, то не существует никаких других процессов, которые исполняются в своих соответствующих критических секциях. Это условие получило название условия взаимоисключения (*mutual exclusion*).
4. Процессы, которые находятся вне своих критических участков и не собираются входить в них, не могут препятствовать другим процессам входить в их собственные критические участки. Если нет процессов в критических секциях и имеются процессы, желающие войти в них, то только те процессы, которые не исполняются в remainder section, должны принимать решение о том, какой процесс войдет в свою критическую секцию. Такое решение не должно приниматься бесконечно долго. Это условие получило название условия прогресса (progress).
5. Не должно возникать бесконечного ожидания для входа процесса в свой критический участок. От того момента, когда процесс запросил разрешение на вход в критическую секцию, и до того момента, когда он это разрешение получил, другие процессы могут пройти через свои критические участки лишь ограниченное число раз. Это условие получило название условия ограниченного ожидания (bound waiting).

Надо заметить, что описание соответствующего алгоритма в нашем случае означает описание способа организации пролога и эпилога для критической секции. Критический участок должен сопровождаться прологом и эпилогом, которые не имеют отношения к активности одиночного процесса. Во время выполнения пролога процесс должен, в частности, получить разрешение на вход в критический участок, а во время выполнения эпилога – сообщить другим процессам, что он покинул критическую секцию.

Наиболее простым решением поставленной задачи является организация пролога и эпилога запретом на прерывания:

```
while (some condition)
 {
   запретить все прерывания
   critical section
   разрешить все прерывания
   remainder section
```

```
    }
```

Поскольку выход процесса из состояния исполнения без его завершения осуществляется по прерыванию, внутри критической секции никто не может вмешаться в его работу. Если прерывания запрещены, невозможно прерывание по таймеру. Отключение прерываний исключает передачу процессора другому процессу. Таким образом, при запрете прерываний процесс может считывать и сохранять совместно используемые данные, не опасаясь вмешательства другого процесса. Однако этот способ практически не применяется, так как опасно доверять управление системой пользовательскому процессу – он может надолго занять процессор, а результат сбоя в критической ситуации может привести к краху ОС и, следовательно, всей системы. Кроме того, нужного результата можно не достичь в многопроцессорной системе, так как запрет прерываний будет относиться только к одному процессу, остальные процессоры продолжают работу и сохраняют доступ к разделенным данным.

Тем не менее, запрет и разрешение прерываний часто применяются как пролог и эпилог к критическим секциям внутри самой операционной системы, например, при обновлении содержимого PSW (Program Status Word).

Для синхронизации потоков одного процесса программист может использовать глобальные блокирующие переменные. С этими переменными, к которым все потоки процесса имеют прямой доступ, программист работает, не обращаясь к системным вызовам ОС.

Каждому набору критических данных ставится в соответствие двоичная переменная. Поток может войти в критическую секцию только тогда, когда значение этой переменной-замка равно 0, одновременно изменяя ее значение на 1 – закрывая замок. При выходе из критической секции поток сбрасывает ее значение в 0 – замок открывается.

```
shared int lock = 0;
while (some condition)
  {
    while(lock); lock = 1;
    critical section
    lock = 0;
```

```
    remainder section
  }
```

К сожалению, внимательное изучение показывает, что такое решение не удовлетворяет условию взаимоисключения, так как действие `while(lock); lock = 1;` не является атомарным. Допустим, что поток `P0` протестировал значение переменной lock и принял решение двигаться дальше. В этот момент, еще до присваивания переменной lock значения 1, планировщик передал управление потоку `P1`. Он тоже изучает содержимое переменной lock и тоже принимает решение войти в критический участок. Мы получаем два процесса, одновременно выполняющих свои критические секции.

Попробуем решить задачу сначала для двух процессов. Очередной подход будет также использовать общую для них обоих переменную с начальным значением 0. Только теперь она будет играть не роль замка для критического участка, а явно указывать, кто может следующим войти в него. Для i-го процесса это выглядит так:

```
shared int turn = 0;
while (some condition)
  {
    while(turn != i);
    critical section
    turn = 1-i;
    remainder section;
  }
```

Легко видеть, что взаимоисключение гарантируется, процессы входят в критическую секцию строго по очереди: `P0`, `P1`, `P0`, `P1`, `P0`, ... Но наш алгоритм не удовлетворяет условию прогресса. Например, если значение `turn` равно 1 и процесс `P0` готов войти в критический участок, он не может сделать этого, даже если процесс `P1` находится в `remainder section`.

Недостаток предыдущего алгоритма заключается в том, что процессы ничего не знают о состоянии друг друга в текущий момент времени. Давайте попробуем исправить эту ситуацию. Пусть два процесса имеют разделяемый массив флагов готовности входа процессов в критический

участок.

```
shared int ready[2] = {0, 0};
```

Когда i-й процесс готов войти в критическую секцию, он присваивает элементу массива ready[i] значение, равное 1. После выхода из критической секции он, естественно, сбрасывает это значение в 0. Процесс не входит в критическую секцию, если другой процесс уже готов к входу в критическую секцию или находится в ней.

```
shared int turn = 0;
while (some condition)
  {
    ready[i] = 1;
    while(ready[1-i]);
    critical section
    ready[i] = 0;
    remainder section
  }
```

Полученный алгоритм обеспечивает взаимоисключение, позволяет процессу, готовому к входу в критический участок, войти в него сразу после завершения эпилога в другом процессе, но все равно нарушает условие прогресса. Пусть процессы практически одновременно подошли к выполнению пролога. После выполнения присваивания ready[0] = 1 планировщик передал процессор от процесса 0 процессу 1, который также выполнил присваивание ready[1] = 1. После этого оба процесса бесконечно долго ждут друг друга на входе в критическую секцию. Возникает ситуация, которую принято называть тупиковой (deadlock).

Первое решение проблемы, удовлетворяющее всем требованиям и использующее идеи ранее рассмотренных алгоритмов, было предложено датским математиком Деккером (Dekker). В 1981 году Петерсон (Peterson) предложил более изящное решение. Пусть оба процесса имеют доступ к массиву флагов готовности и к переменной очередности.

```
shared int ready[2] = {0, 0};
shared int turn;
```

```
while (some condition)
  {
    ready[i] = 1;
    turn =1- i;
    while(ready[1-i] && turn == 1-i);
    critical section
    ready[i] = 0;
    remainder section
  }
```

При исполнении пролога критической секции процесс Pi заявляет о своей готовности выполнить критический участок и одновременно предлагает другому процессу приступить к его выполнению. Если оба процесса подошли к прологу практически одновременно, то они оба объявят о своей готовности и предложат выполняться друг другу. При этом одно из предложений всегда последует после другого. Тем самым работу в критическом участке продолжит процесс, которому было сделано последнее предложение.

Наличие аппаратной поддержки взаимоисключений позволяет упростить алгоритмы и повысить их эффективность точно так же, как это происходит и в других областях программирования. Мы уже обращались к аппаратным средствам для решения задачи реализации взаимоисключений, когда говорили об использовании механизма запрета-разрешения прерываний.

Многие вычислительные системы помимо этого имеют специальные команды процессора, которые позволяют проверить и изменить значение машинного слова или поменять местами значения двух машинных слов в памяти, выполняя эти действия как атомарные операции. Рассмотрим, как концепции таких команд могут быть использованы для реализации взаимоисключений.

О выполнении команды `Test-and-Set`, осуществляющей проверку значения логической переменной с одновременной установкой ее значения в 1, можно думать как о выполнении функции

```
int Test_and_Set (int *target)
  {
    int tmp = *target;
```

```
    *target = 1;
    return tmp;
  }
```

С использованием этой атомарной команды мы можем модифицировать алгоритм для переменной-замка так, чтобы он обеспечивал взаимоисключения

```
shared int lock = 0;
while (some condition)
  {
    while(Test_and_Set(&lock));
    critical section
    lock = 0;
    remainder section
  }
```

К сожалению, даже в таком виде полученный алгоритм не удовлетворяет условию ограниченного ожидания для алгоритмов. Недостатком рассмотренного способа взаимоисключения является необходимость постоянного опроса другими потоками, требующими тот же ресурс, блокирующей переменной, когда один из потоков находится в критической секции. На это будет бесполезно тратиться процессорное время. Для устранения этого недостатка во многих ОС предусматриваются системные вызовы для работы с критическими секциями.

Большинство алгоритмов, рассмотренных выше, хотя и являются корректными, но достаточно громоздки и не обладают элегантностью. Более того, процедура ожидания входа в критический участок включает в себя достаточно длительное вращение процесса в пустом цикле, с потреблением впустую драгоценного времени процессора. Существуют и другие серьезные недостатки у алгоритмов, построенных средствами обычных языков программирования.

5.8. Семафоры и мониторы

Одним из первых механизмов, предложенных для синхронизации поведения процессов, стали семафоры, концепцию которых описал

Дейкстрой (Dijkstra) в 1965 году. Семафор представляет собой целую переменную, принимающую неотрицательные значения, доступ любого процесса к которой, за исключением момента ее инициализации, может осуществляться только через две атомарные операции: P (от датского слова proberen – проверять) и V (от verhogen – увеличивать). Классическое определение этих операций выглядит следующим образом:

```
P(S): пока S == 0 процесс блокируется;
S = S - 1;
V(S): S = S + 1;
```

Эта запись означает следующее: при выполнении операции P над семафором S сначала проверяется его значение. Если оно больше 0, то из S вычитается 1. Если оно меньше или равно 0, то процесс блокируется до тех пор, пока S не станет больше 0, после чего из S вычитается 1. При выполнении операции V над семафором S к его значению просто прибавляется 1.

Подобные переменные-семафоры могут с успехом использоваться для решения различных задач организации взаимодействия процессов. В ряде языков программирования они были непосредственно введены в синтаксис языка (например, в ALGOL-68), в других случаях применяются через использование системных вызовов. Соответствующая целая переменная располагается внутри адресного пространства ядра операционной системы. Операционная система обеспечивает атомарность операций P и V, используя, например, метод запрета прерываний на время выполнения соответствующих системных вызовов. Если при выполнении операции P заблокированными оказались несколько процессов, то порядок их разблокирования может быть произвольным, например, FIFO.

Одной из типовых задач, требующих организации взаимодействия процессов, является задача producer-consumer (производитель-потребитель). Пусть два процесса обмениваются информацией через буфер ограниченного размера. Производитель закладывает информацию в буфер, а потребитель извлекает ее оттуда. Грубо говоря, на этом уровне деятельность потребителя и производителя можно описать следующим образом:

```
Producer: while(1)
  {
    produce_item;
    put_item;
  }
    Consumer:
    while(1)
 {
  get_item;
  consume_item;
  }
```

Если буфер забит, то производитель должен ждать, пока в нем появится место, чтобы положить туда новую порцию информации. Если буфер пуст, то потребитель должен дожидаться нового сообщения. Как можно реализовать эти условия с помощью семафоров? Возьмем три семафора `empty`, `full` и `mutex`. Семафор `full` будем использовать для гарантии того, что потребитель будет ждать, пока в буфере появится информация.

Семафор `empty` будем использовать для организации ожидания производителя при заполненном буфере, а семафор `mutex` – для организации взаимоисключения на критических участках, которыми являются действия `put_item` и `get_item` (операции "положить информацию" и "взять информацию" не могут пересекаться, поскольку возникает опасность искажения информации). Тогда решение задачи выглядит так:

```
Semaphore mutex = 1;
Semaphore empty = N, где N – емкость буфера;
Semaphore full = 0;
Producer: while(1)
  {
    produce_item;
    P(empty);
    P(mutex);
    put_item;
    V(mutex);
    V(full);
```

```
}
Consumer: while(1)
{
P(full);
P(mutex);
get_item;
V(mutex);
V(empty);
consume_item;
}
```

Легко убедиться, что это действительно корректное решение поставленной задачи. Попутно заметим, что семафоры использовались здесь для достижения двух целей: организации взаимоисключения на критическом участке и синхронизации скорости работы процессов.

Хотя решение задачи producer-consumer с помощью семафоров выглядит достаточно элегантно, программирование с их использованием требует повышенной осторожности и внимания, чем отчасти напоминает программирование на языке Ассемблера. Допустим, что в рассмотренном примере мы случайно поменяли местами операции P, сначала выполнив ее для семафора mutex, а уже затем для семафоров full и empty. Допустим теперь, что потребитель, войдя в свой критический участок (mutex сброшен), обнаруживает, что буфер пуст. Он блокируется и начинает ждать появления сообщений. Но производитель не может войти в критический участок для передачи информации, так как тот заблокирован потребителем. Получаем тупиковую ситуацию.

В сложных программах произвести анализ правильности использования семафоров с карандашом в руках становится очень непростым занятием. В то же время обычные способы отладки программ зачастую не дают результата, поскольку возникновение ошибок зависит от *interleaving*'а атомарных операций, и ошибки могут быть трудно воспроизводимы. Для того чтобы облегчить труд программистов, в 1974 году Хоаром (Hoare) был предложен механизм еще более высокого уровня, чем семафоры, получивший название мониторов. Рассмотрим конструкцию, несколько отличающуюся от оригинальной.

Мониторы представляют собой тип данных, который может быть с успехом внедрен в объектно-ориентированные языки программирования. Монитор обладает своими собственными переменными, определяющими его состояние. Значения этих переменных извне монитора могут быть изменены только с помощью вызова функций-методов, принадлежащих монитору. В свою очередь, эти функции-методы могут использовать в своей работе только данные, находящиеся внутри монитора, и свои параметры. На абстрактном уровне можно описать структуру монитора следующим образом:

```
monitor monitor_name
 {
  описание переменных ;
  void m1(...){...
 }
 void m2(...)
 {
  ...
 }
  ...
  void mn(...)
 {
  ...
 }
 {
  блок инициализации внутренних переменных ;
 }
```

Здесь функции m1,..., mn представляют собой функции-методы монитора, а блок инициализации внутренних переменных содержит операции, которые выполняются только один раз: при создании монитора или при самом первом вызове какой-либо функции-метода до ее исполнения.

Важной особенностью мониторов является то, что в любой момент времени только один процесс может быть активен, т. е. находиться в состоянии "готовность" или "исполнение", внутри данного монитора. Поскольку мониторы представляют собой особые конструкции языка программирования, компилятор может отличить вызов функции,

принадлежащей монитору, от вызовов других функций и обработать его специальным образом, добавив к нему пролог и эпилог, реализующий взаимоисключение. Так как обязанность конструирования механизма взаимоисключений возложена на компилятор, а не на программиста, работа программиста при использовании мониторов существенно упрощается, а вероятность появления ошибок становится меньше.

Однако одних только взаимоисключений недостаточно для того, чтобы в полном объеме реализовать решение задач, возникающих при взаимодействии процессов. Нам нужны еще и средства организации очередности процессов, подобно семафорам `full` и `empty` в предыдущем примере. Для этого в мониторах было введено понятие *условных переменных* (*condition variables*), над которыми можно совершать две операции – `wait` и `signal`, до некоторой степени похожие на операции `P` и `V` над семафорами.

Если функция монитора не может выполняться дальше, пока не наступит некоторое событие, она выполняет операцию wait над какой-либо условной переменной. При этом процесс, выполнивший операцию `wait`, блокируется, становится неактивным, и другой процесс получает возможность войти в монитор.

Когда ожидаемое событие происходит, другой процесс внутри функции-метода совершает операцию `signal` над той же самой условной переменной. Это приводит к пробуждению ранее заблокированного процесса, и он становится активным. Если несколько процессов дожидались операции `signal` для этой переменной, то активным становится только один из них. Что нужно предпринять для того, чтобы не оказалось двух процессов, разбудившего и пробужденного, одновременно активных внутри монитора? *Хор* предложил, чтобы пробужденный процесс подавлял исполнение разбудившего процесса, пока он сам не покинет монитор. Несколько позже Хансен (Hansen) предложил другой механизм: разбудивший процесс покидает монитор немедленно после исполнения операции signal. Рассмотрим подход Хансена. Применим концепцию мониторов к решению задачи "производитель-потребитель".

```
monitor ProducerConsumer
{
```

```
    condition full, empty;
    int count;
    void put()
  {
    if(count == N) full.wait;
    put_item;
    count += 1;
   if(count == 1) empty.signal;
   }
    void get()
   {
   if (count == 0) empty.wait;
  get_item();
  count -= 1;
  if(count == N-1) full.signal;
   }
   {
    count = 0;
   }
    }
     Producer:
     while(1)
    {
    produce_item;ProducerConsumer.put();
    }
   Consumer:
   while(1)
    {
      ProducerConsumer.get();
      consume_item;
 }
```

Легко убедиться, что приведенный пример действительно решает поставленную задачу.

5.9. Взаимоблокировки (тупики)

Коффман и другие исследователи доказали, что для возникновения

тупиковой ситуации должны выполняться четыре условия [37].

1. Условие взаимного исключения. Каждый ресурс в данный момент или отдан ровно одному процессу, или доступен.
2. Условие удерживания и ожидания. Процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы.
3. Условие отсутствия принудительной выгрузки ресурсов. У процесса нельзя забрать принудительно ранее полученные ресурсы. Процесс, владеющий ими, должен сам освободить ресурсы.
4. Условие циклического ожидания. Должна существовать круговая последовательность из двух и более процессов, каждый из которых ждет доступа к ресурсу, удерживаемому следующим членом последовательности.

Для того чтобы произошла взаимоблокировка, должны выполняться все эти четыре условия. Если хотя бы одно отсутствует, тупиковая ситуация невозможна.

При столкновении с взаимоблокировками используются четыре стратегии.

- Пренебрежение проблемой в целом.
- Обнаружение и восстановление. Позволить взаимоблокировке произойти, обнаружить ее и предпринять какие-либо действия.
- Избегать тупиковых ситуаций с помощью аккуратного распределения ресурсов.
- Предотвращать с помощью структурного опровержения одного из четырех условий, необходимых для взаимоблокировки.

Если взаимоблокировка случается в среднем раз в пять лет, а сбои ОС, компиляторов и неисправности аппаратуры происходят в среднем один раз в неделю, то подходит первая стратегия. К этому надо добавить, что большинство операционных систем потенциально страдают от взаимоблокировок, которые не обнаруживаются, не говоря уже об автоматическом выходе из тупика.

Вторая техника представляет собой обнаружение и восстановление.

При использовании этого метода система не пытается предотвратить попадания в тупиковые ситуации. Вместо этого она позволяет произойти взаимоблокировке, старается определить, когда это случилось, и затем совершает некоторые действия по возврату системы к состоянию, имевшему место до того, как система попала в тупик.

Рассмотрим методы обнаружения взаимоблокировок.

Обнаружение взаимоблокировки при наличии одного ресурса каждого типа достаточно просто. Для такой системы можно построить граф ресурсов и процессов, о котором уже говорилось, и если в графе нет циклов, система в тупик не попала.

Например, пусть система из семи процессов (A, B, C, D, E, F, G) и шести ресурсов (R, S, T, V, W, U) в некоторый момент соответствует следующему списку [17].

1. Процесс A занимает ресурс R и хочет получить ресурс S.

Вопрос: заблокирована ли эта система, и если да, то какие процессы в этом участвуют?

Чтобы ответить на этот вопрос, нужно составить граф ресурсов и процессов (рис. 5.14).

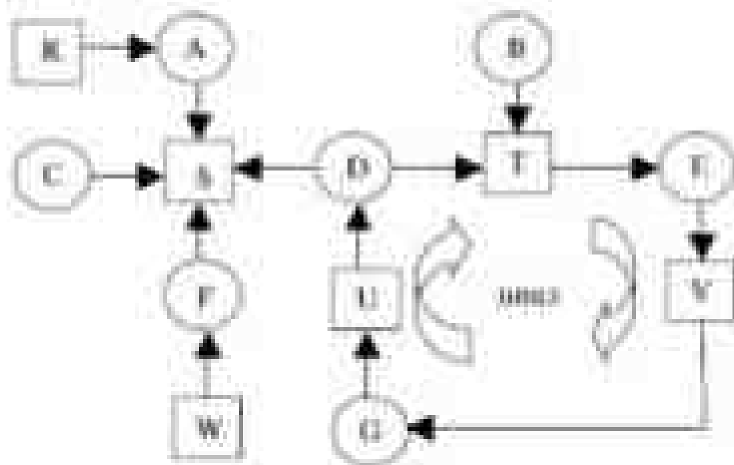


Рис. 5.14. Граф ресурсов и процессов

Этот граф содержит цикл, указывающий, что процессы D, E, G заблокированы (зрительно легко видно). Однако в этом случае в операционной системе необходима реализация формального алгоритма, выявляющего тупики.

Рассмотрим возможность обнаружения взаимоблокировок при наличии нескольких ресурсов каждого типа. Пусть имеется множество процессов $P=\{P_1, P_2, \ldots, P_n\}$, всего n процессов, и множество ресурсов $E=\{E_1, E_2, \ldots, E_m\}$, где m – число классов ресурсов. В любой момент времени некоторые из ресурсов могут быть заняты и, соответственно, недоступны. Пусть A – вектор доступных ресурсов $A=\{A_1, A_2, \ldots, A_m\}$. Очевидно, что $A_j \le E_j$, $j = 1, 2, \ldots, m$.

Введем в рассмотрение две матрицы:

$C=\{c_{i,j} \mid i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, m\}$ – матрица текущего распределения ресурсов, где $c_{i,j}$ – количество ресурсов j-ого класса, которые занимает процесс P_i;

$R=\{r_{i,j} \mid i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, m\}$ – матрица требуемых (запрашиваемых) ресурсов, $r_{i,j}$ – количество ресурсов j-ого класса, которые хочет получить процесс P_i.

Справедливо m соотношений по ресурсам:

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j,\ j = 1, 2, \ldots, m.$$

Алгоритм обнаружения взаимоблокировок основан на сравнении векторов доступных и требуемых ресурсов. В исходном состоянии все процессы не маркированы (не отмечены). По мере реализации алгоритма на процессы будет ставиться отметка, служащая признаком того, что они могут закончить свою работу и, следовательно, не находятся в тупике. После завершения алгоритма любой немаркированный процесс находится в тупиковой ситуации.

Алгоритм обнаружения тупиков состоит из следующих шагов.

1. Отыскивается процесс P_i, для которого i-я строка матрицы R меньше вектора A, т.е. $R_i <= A$, или $r_{i,j} <= A_j$, j=1, 2, …, m.
2. Если такой процесс найден, это означает, что он может завершиться, а следовательно – освободить занятые ресурсы. Найденный процесс маркируется, и далее прибавляется i-я строка матрицы C к вектору A, т.е. $A_j = A_j + c_{i,j}$, j=1, 2, …, m. Возвращаемся к шагу 1.
3. Если таких процессов не существует, работа алгоритма заканчивается. Немаркированные процессы попадают в тупик.

Когда нужно искать возникновение тупиков? Можно, конечно, проверять систему каждый раз, когда запрашивается очередной ресурс, это позволит обнаружить тупик максимально рано, но приведет к большим издержкам процессорного времени. Поэтому период проверки нужно выбрать: например, каждые K (сколько – нужно определить!) минут или когда процессор слабо загружен.

Предположим, обнаружен тупик. Какие методы можно использовать для его ликвидации? Здесь возможно несколько подходов.

Первый – принудительная выгрузка ресурсов: способность забирать ресурс у процесса, отдавать его другому процессу, а затем возвращать назад так, что исходный процесс не замечает того, в значительной мере зависит от свойств ресурса. Выйти из тупика, таким образом, зачастую трудно или невозможно.

Второй подход – восстановление через откат. В этом способе процессы должны периодически создавать контрольные точки, позволяющие запустить процесс с его предыстории. Когда взаимоблокировка обнаружена, достаточно просто понять, какие ресурсы нужны процессам. Чтобы выйти из тупика, процесс, занимающий необходимый ресурс, откатывается к тому моменту времени, перед которым он получил данный ресурс, для чего запускается одна из его контрольных точек. Вся работа, выполненная после этой контрольной точки, теряется. Если возобновленный процесс вновь пытается получить данный ресурс, ему придется ждать, когда ресурс станет доступным.

Третий подход – восстановление путем уничтожения одного или более процессов. Это грубейший, но простейший выход из тупика. Проблема – решить, какой процесс уничтожать.

Идеальной была бы такая организация вычислительного процесса, при которой не возникали бы тупики за счет безопасного распределения ресурсов. Подобные алгоритмы базируются на концепции безопасных состояний.

5.10. Синхронизирующие объекты ОС

Рассмотренные способы синхронизации, основанные на глобальных переменных процесса, обладают существенным недостатком – они не подходят для синхронизации потоков различных процессов. В таких случаях ОС должна предоставлять потокам системные объекты синхронизации, которые были бы видны для всех потоков, даже если они принадлежат разным процессам и работают в разных адресных пространствах.

Примерами таких синхронизирующих объектов являются системные семафоры, мьютексы, события, таймеры и др. Набор таких объектов определяется конкретной ОС. Чтобы разные процессы могли разделять синхронизирующие объекты, используются различные методы. Некоторые ОС возвращают указатель на объект. Этот указатель может быть доступен всем родственным процессам, наследующим характеристики общего родительского процесса. В других ОС процессы в запросах на создание объектов синхронизации указывают имена, которые должны им быть присвоены. Далее эти имена используются различными процессами для манипуляций объектами синхронизации. В этом случае работа с синхронизирующими объектами подобна работе с файлами. Их можно создавать, открывать, закрывать, уничтожать.

Для синхронизации могут быть использованы такие объекты ОС, как файлы, процессы и потоки. Все эти объекты могут находиться в двух состояниях: сигнальном и несигнальном – свободном. Смысл, вкладываемый в понятие "сигнальное состояние", зависит от типа объекта. Так, например, поток переходит в сигнальное состояние, когда он завершается. Процесс переходит в сигнальное состояние, когда завершились все его потоки. Файл переходит в сигнальное состояние,

когда завершается операция ввода-вывода для этого файла. Для остальных объектов сигнальное состояние устанавливается в результате выполнения специальных системных вызовов. Приостановка и активизация потоков осуществляется в зависимости от состояния синхронизирующих объектов ОС.

Потоки с помощью специального системного вызова (Wait (X), где X – указатель на объект синхронизации) сообщают операционной системе о том, что они хотят синхронизировать свое выполнение с состоянием объекта X. Системный вызов, с помощью которого поток может перевести объект синхронизации в сигнальное состояние, назовем Set(X).

Поток, выполнивший системный вызов Wait(X), переводится операционной системой в состояние ожидания до тех пор, пока объект X не перейдет в сигнальное состояние. Поток может ждать сигнального состояния не одного объекта, а нескольких. Может случиться, что установки некоторого объекта в сигнальное состояние ожидают несколько потоков.

Синхронизация тесно связана с планированием потоков. Во-первых, любое обращение потока к системному вызову Wait(X) приводит к переводу его в очередь ожидающих потоков, а из очереди готовых потоков выбирается и активизируется новый поток.

Во-вторых, при переходе объекта в сигнальное состояние (в результате выполнения некоторого потока – системного или прикладного) ожидающий этот объект поток переводится в очередь готовых к выполнению потоков. Таким образом, в обоих случаях происходит перепланирование потоков, в том числе изменение их приоритетов и квантов времени, если это предусмотрено в ОС.

Круг событий, с которыми потоку может потребоваться синхронизировать свое выполнение, не исчерпывается завершением потока, процесса или операции ввода-вывода. Поэтому в ОС имеются и другие, более универсальные объекты синхронизации, такие как событие (event), мьютекс (mutex), системный семафор и др.

Мьютекс (mutex – сокращение от *mutual exclusion* – взаимное исключение) – упрощенный семафор, не способный считать; он может

управлять лишь взаимным исключением доступа к совместно используемым ресурсам или кодам. Реализация мьютекса полезна в случае потоков, действующих только в пространстве пользователя.

Объект "событие" обычно используется не для доступа к данным, а для того, чтобы оповестить другие потоки о том, что некоторые действия завершены.

Сигналы дают возможность задаче реагировать на событие, источником которого может быть ОС или другая задача. Синхронные сигналы чаще всего приходят от системы прерываний процессора и свидетельствуют о действиях процесса, блокируемых аппаратурой, например, деление на нуль, ошибка адресации, нарушение защиты памяти и т.д. Примером асинхронного сигнала является сигнал с терминала. Во многих ОС предусматривается оперативное снятие процесса с выполнения (Ctrl + Break) для выработки сигнала ОС и направления его процессу.

Обработка сигналов аналогична обработке аппаратных прерываний ввода-вывода. Сигналы обеспечивают логическую связь между процессами, а также между процессами и пользователями (терминалами). Поскольку посылка сигнала предусматривает знание идентификатора процесса, взаимодействие посредством сигналов возможно только для членов группы процессов, состоящей из родительского и дочерних процессов. Процесс может послать сигнал всей своей группе за один системный вызов.

Другим средством взаимодействия процессов является канал (труба) – псевдофайл, который может использоваться для связи двух процессов. Когда процесс А хочет отправить данные процессу В, он пишет их в канал, как если бы это был выходной файл. Процесс В читает данные из канала, как если бы он был входным файлом. Подобное средство взаимодействия используется в операционной системе UNIX.

Почтовые ящики, используемые в Windows 2000, в некоторых аспектах подобны каналам, однако в отличие от каналов являются однонаправленными. Они позволяют отправляющему процессу использовать широковещание для рассылки сообщений сразу многим получателям.

Для прямой и непрямой адресации достаточно двух примитивов, чтобы

описать передачу сообщений по линии связи – send и receive. В случае прямой адресации их можно обозначать так:

`send(P, message)` – послать сообщение message процессу P;

`receive(Q, message)` – получить сообщение message от процесса Q.

В случае непрямой адресации мы будем обозначать их так:

`send(A, message)` – послать сообщение message в почтовый ящик A;

`receive(A, message)` – получить сообщение message из почтового ящика A.

Примитивы send и receive уже имеют скрытый от наших глаз механизм взаимоисключения. Более того, в большинстве систем они уже имеют и скрытый механизм блокировки при чтении из пустого буфера и при записи в полностью заполненный буфер. Реализация решения задачи producer-consumer для таких примитивов становится тривиальной. Надо отметить, что, несмотря на простоту использования, передача сообщений в пределах одного компьютера происходит существенно медленнее, чем работа с семафорами и мониторами.

Сокеты (ОС Windows 2000) подобны каналам с тем отличием, что они при нормальном использовании соединяют процессы на разных компьютерах. Например, один процесс пишет в сокет, а другой на удаленной машине читает из него. В принципе, сокеты можно использовать для соединения процессов на одной машине, но это связано с большими накладными расходами.

Вызов удаленной процедуры (Remote Procedure Call, RPC) представляет собой способ, которым процесс A просит процесс B вызвать процедуру в адресном пространстве процесса B от имени процесса A и вернуть результат процессу A.

Наконец, процессы могут совместно использовать память для одновременного отображения одного и того же файла. Все, что один процесс будет писать в этот файл, будет появляться в адресном

пространстве других процессов. С помощью такого механизма легко реализовать общий буфер, применяемый в задаче производителя и потребителя. Запись в этот файл одного из процессов мгновенно становится видной остальным.

5.11. Аппаратно-программные средства поддержки мультипрограммирования

Ни одна операционная система не будет эффективно работать без аппаратно-программной системы прерывания. Тем более невозможно организовать мультипрограммный режим без хорошо организованной системы прерывания. Действительно, периодические прерывания от таймера вызывают смену процессов в мультипрограммной ОС; прерывания от устройств ввода-вывода управляют потоками данных, которыми вычислительная система обменивается с внешним миром; сигналы от управляемых объектов позволяют оперативно реагировать на различные ситуации, складывающиеся в процессах управления этими объектами; сигналы от схем контроля устройств компьютера позволяют включить резервные устройства; ошибки при выполнении программ позволяют принять действия по их устранению или приводят к аварийному завершению программ и т.д.

В зависимости от источника прерывания делятся на три класса (правда, это весьма условно, разные авторы классифицируют прерывания различно):

- внешние;
- внутренние;
- программные.

Внешние прерывания могут возникать в результате действий пользователя (клавиатура, мышь), поступления сигналов от периферийных устройств (принтеры, диски, управляемых объектов и т.п.) и других внешних устройств, подключенных к компьютеру. Данный класс прерываний является асинхронным по отношению к потоку команд прерываемой программы. Обычно опрос системы прерываний производится после завершения текущей команды, а текущий процесс продолжается, уже начиная со следующей команды.

Внутренние прерывания (исключения – exception) происходят синхронно выполнению программы при появлении аварийной ситуации в ходе исполнения некоторой инструкции программы. Примерами исключений являются деление на нуль, ошибки защиты памяти, обращения по несуществующему адресу, попытка выполнить привилегированную инструкцию в пользовательском режиме и т.п. Исключения возникают в ходе выполнения такта команды.

Программные прерывания отличаются от предыдущих классов тем, что они по своей сути не являются "истинными" (непредсказуемыми) прерываниями. Программное прерывание "запланировано" программистом и возникает при выполнении особой команды процессора, имитирующей прерывание, т.е. переход на новую последовательность инструкций. Например, такой командой в процессоре Pentium является INT, в процессорах Motorola – trap. Причинами использования таких команд являются:

- желание получить более компактный код программы;
- необходимость перехода из пользовательского режима в привилегированный;
- обращение к услугам ОС – системный вызов.

Прерываниям приписывается приоритет, с помощью которого они ранжируются по степени важности и срочности. Операционные системы имеют специальные модули для работы с прерываниями – обработчики прерываний, или процедуры обслуживания прерываний (Interrupt Service Routine, ISR). Аппаратные прерывания обрабатываются драйверами соответствующих внешних устройств, исключения – специальными модулями ядра, а программные прерывания – процедурами ОС, обслуживающими системные вызовы. Кроме этих модулей, в ОС может быть диспетчер прерываний, координирующий работу отдельных обработчиков прерываний.

Аппаратная поддержка прерываний имеет свои особенности, зависящие от типа процессора и других аппаратных компонентов (контроллер внешнего устройства, шины подключения внешних устройств, контроллеры прерываний и др.). Существуют два основных способа, с помощью которых шины выполняют прерывания: векторный (vectored) и опрашиваемый (polled). В обоих случаях процессору

предоставляется информация об уровне приоритета прерывания на шине подключения внешних устройств. В случае векторных прерываний в процессор передается информация о начальном адресе программы обработки прерываний – обработчика прерывания.

При использовании опрашиваемых прерываний процессор получает от запросившего прерывание устройства только информацию об уровне приоритета прерывания. С каждым уровнем может быть связано несколько устройств и, соответственно, несколько обработчиков прерываний. В этом случае при возникновении прерывания процессор вызывает поочередно всех обработчиков прерываний данного уровня приоритета, пока один из обработчиков не подтвердит, что прерывание пришло от обслуживаемого им устройства.

Возможен комбинированный подход, сочетающий векторный и опрашиваемый типы прерываний. Такой подход реализован в ПК на основе процессоров Intel Pentium. Вектор прерываний в процессоре Pentium поставляет контроллер прерываний, который отображает поступающий от шины сигнала IRQ (Interrupt request) на определенный номер вектора. Вектор прерываний представляет собой число, указывающее на одну из 256 программ обработки прерываний, адреса которых хранятся в таблице обработчиков прерываний. В том случае, когда к каждой линии IRQ подключается только одно устройство, процедура прерываний работает как чисто векторная. Однако при совместном использовании одного уровня IRQ несколькими устройствами обработка прерываний реализуется по схеме опрашиваемых прерываний, т.е. в данном случае необходимо выполнить опрос всех устройств, подключенных к данному уровню IRQ (Interrupt Request).

Обычно в ОС поддерживаются механизм приоритезации и маскирование прерываний. Каждый класс прерываний имеет свой уровень приоритета. Приоритеты могут обслуживаться как относительные и как абсолютные. Маскирование позволяет запретить прерывания любого приоритета на некоторый промежуток времени. В целом эти механизмы позволяют организовать гибкое обслуживание прерываний.

Обобщенно последовательность действий аппаратных и программных

средств по обработке прерываний можно представить следующим образом.

1. При возникновении сигнала (аппаратные прерывания) или условия (для внутренних прерываний) происходит первичное аппаратное распознавание типа прерывания. Если прерывания в данный момент запрещены, то процессор продолжает текущую программу. В противном случае вызывается диспетчер прерываний. Он запрещает на некоторое время все прерывания и устанавливает причину прерывания. После этого диспетчер сравнивает приоритет источника прерывания с текущим приоритетом потока команд, выполняемого процессором. Заметим, что в это время процессор мог выполнять поток задания пользователя или другого обработчика прерываний, имеющего некоторый приоритет. Однако по отношению к обработчикам прерываний любой пользовательский поток имеет более низкий приоритет, так что любой запрос на прерывание всегда может прервать выполнение этого потока.

   Если прерывания разрешены и поступивший запрос на прерывание имеет приоритет более высокий, чем текущий поток, то будет производиться вызов процедуры обработки прерывания, адрес которой содержится в ОП в таблице векторов прерываний.

2. Автоматически сохраняется некоторая часть контекста прерванного потока, которая позволит ядру возобновить исполнение потока процесса после обработки прерывания (обычно это счетчик команд, слово состояния процессора – регистр EFLAGS в Pentium, регистры общего назначения). Может быть сохранен и полный контекст, если ОС обслуживает данное прерывание со сменой процесса. Однако это происходит не всегда, например, обслуживание прерывания по вводу-выводу (прием очередной порции данных от контроллера внешнего устройства) чаще всего выполняется без смены текущего процесса.
3. Одновременно с загрузкой адреса процедуры обработки прерываний в счетчик команд производится загрузка нового *PSW* (слова состояния процессора), которое определяет режимы работы процессора при обработке прерывания. Прерывания

обрабатывается в привилегированном режиме модулями ядра ОС, так как при этом нужно выполнить ряд критических операций, от которых зависит жизнеспособность системы.

4. Временно запрещаются прерывания данного типа, чтобы не образовалась очередь вложенных друг в друга потоков одной и той же процедуры. Делается это обычно маскированием прерываний. Многие процессоры автоматически устанавливают признак запрета прерываний в начале цикла обработки прерывания, в противном случае это делает программа обработки прерывания.
5. После того как прерывание обработано ядром операционной системы, прерванный контекст восстанавливается (частично аппаратно – PSW, содержимое счетчика команд, частично программно – извлечением данных из стека) и работа потока возобновляется с прерванного места. Снимается блокировка повторных прерываний данного типа.

Если прерывание поступило в тот момент, когда процессор выполнял инструкции другого обработчика прерываний, то сравниваются приоритеты нового прерывания и текущий приоритет. Если приоритеты нового запроса выше, то выполнение текущего обработчика приостанавливается и он помещается в соответствующую очередь обработчиков прерываний. В противном случае, в очередь помещается обработчик нового запроса.

Диспетчеризация прерываний является важнейшей функцией ОС. По сути, диспетчер прерываний реализует верхний уровень планирования всех работ, выполняющихся в системе. На этом уровне распределяется процессорное время между потоками поступающих запросов на прерывания различных типов – внешних, внутренних и программных. Оставшееся процессорное время распределяется диспетчером потоков на основании дисциплины квантования или других дисциплин.

5.12. Системные вызовы

Системный вызов позволяет приложению обратиться к операционной системе с просьбой выполнить то или иное действие, оформленное как процедуры кодового сегмента ОС. В этом плане для прикладного

программиста ОС представляется некоторой библиотекой, имеющей набор различных функций, с помощью которых можно упростить прикладную программу или выполнить действия, запрещенные в пользовательском режиме, например, обмен данными с устройством ввода-вывода.

Реализация системных вызовов должна удовлетворять следующим требованиям [10, 17]:

- обеспечивать переключение в привилегированный режим;
- обладать высокой скоростью вызова процедур ОС;
- обеспечивать по возможности единообразное обращение к системным вызовам для всех аппаратных платформ, на которых работает ОС;
- допускать простое расширение системных вызовов;
- обеспечивать контроль со стороны ОС за корректным использованием системных вызовов.

Первое требование – переключение в привилегированный режим выполняется через механизм программных прерываний.

Для обеспечения высокой скорости было бы полезно использовать векторные свойства системы программных прерываний, т.е. закрепить за каждым системным вызовом определенный вектор. Однако этот децентрализованный способ передачи управления требует наличия свободного элемента в таблице прерываний, которого может не оказаться. К этому же таблица прерываний обычно ограничена в размерах.

Поэтому в большинстве ОС системные вызовы обслуживаются по централизованной схеме, основанной на существовании диспетчера системных вызовов (рис. 5.15). При любом системном вызове приложение выполняет программное прерывание с определенным и единственным номером вектора (например, INT 2Eh при работе на платформе Pentium). Перед выполнением прерывания приложение передает операционной системе номер системного вызова, который является индексом в таблице адресов процедур ОС, реализующих системные вызовы. Кроме того, передаются параметры (аргументы) системного вызова (эти данные могут передаваться через регистр

общего назначения или стек пользователя).

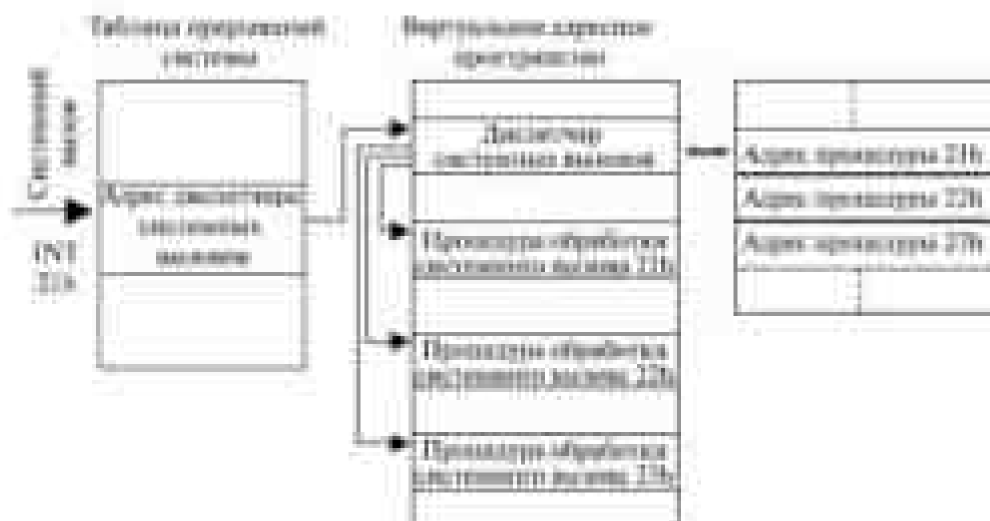


Рис. 5.15. Диспетчер системных вызовов

Любой системный вызов приводит к запуску диспетчера системных вызовов, который представляет собой простую программу, сохраняющую содержимое регистров в системном стеке (поскольку в результате программного прерывания процессор переходит в привилегированный режим) и проверяющую, попадает ли запрошенный номер вызова в поддерживаемый ОС диапазон (т.е. не выходит ли номер за границы таблицы). Если все условия соблюдены, диспетчер передает управление процедуре ОС, адрес которой задан в таблице адресов системных вызовов.

Процедура реализации системного вызова извлекает из системного стека аргументы и выполняет заданное действие (чтение системных часов, чтение из файла, запрос на выделение дополнительной памяти и т.д.). После завершения работы системного вызова управление возвращается диспетчеру, при этом он получает код завершения этого вызова. Диспетчер восстанавливает регистры процессора, помещает в определенный регистр код возврата и выполняет инструкцию возврата из прерывания, которая восстанавливает непривилегированный режим работы процессора.

Для приложений системный вызов внешне ничем не отличается от

вызова библиотечной функции языка C, выполняющейся в пользовательском режиме. И такая ситуация действительно существует – для всех системных вызовов в библиотеках, предоставляемых компилятором C, имеются так называемые "заглушки" (stub – остаток, огрызок). Каждая заглушка оформлена как C-функция, при этом она содержит несколько ассемблерных строк, нужных для выполнения инструкции программного прерывания. Таким образом, пользовательская программа вызывает заглушку, а та, в свою очередь, вызывает процедуру ОС.

Прикладной программист имеет дело с набором функций прикладного программного интерфейса API, например, Win32 API. Количество вызовов в Win32 API исчисляется тысячами, причем многие из них являются библиотечными функциями, работающими в пользовательском пространстве, т.е. не являются настоящими системными вызовами.

Операционная система выполняет системные вызовы в синхронном и асинхронном режимах. В первом случае процесс, сделавший такой вызов, приостанавливается до тех пор, пока системный вызов не выполнит свою работу. После этого планировщик переводит процесс в состояние готовности и при очередном выполнении процесс может воспользоваться результатами завершившегося системного вызова.

Асинхронный системный вызов не приводит к переводу процесса в режим ожидания и после выполнения некоторых начальных системных действий, например, запуска операции ввода-вывода, управление возвращается прикладному процессу. Такой режим работы характерен для ОС на основе микроядра.

Управление памятью. Методы, алгоритмы и средства

Организация памяти современного компьютера. Функции ОС по управлению памятью. Распределение памяти. Страничная организация виртуальной памяти. Оптимизация функционирования страничной виртуальной памяти. Сегментная организация виртуальной памяти. Сегментно-страничная виртуальная память.

6.1. Организация памяти современного компьютера

Со времен создания ЭВМ фон Неймана основная память в компьютерной системе организована как линейное (одномерное) адресное пространство, состоящее из последовательности слов, а позже байтов [10]. Аналогично организована и внешняя память. Хотя такая организация и отражает особенности используемого аппаратного обеспечения, она не соответствует способу, которым обычно создаются программы. Большинство программ организованы в виде модулей, некоторые из которых неизменны (только для чтения, только для исполнения), а другие содержат данные, которые могут быть изменены.

Если операционная система и аппаратное обеспечение могут эффективно работать с пользовательскими программами и данными, представленными модулями, то это обеспечивает ряд преимуществ.

1. Модули могут быть созданы и скомпилированы независимо друг от друга, при этом все ссылки из одного модуля в другой разрешаются системой во время работы программы.
2. Разные модули могут получать разные степени защиты (только чтение, только исполнение) за счет весьма умеренных накладных расходов.
3. Возможно применение механизма, обеспечивающего совместное использование модулей разными процессами (для случая сотрудничества процессов в работе над одной задачей).

Память – важнейший ресурс вычислительной системы, требующий эффективного управления. Несмотря на то, что в наши дни память среднего домашнего компьютера в тысячи раз превышает память больших ЭВМ 70-х годов, программы увеличиваются в размере

быстрее, чем память. Достаточно сказать, что только операционная система занимает сотни Мбайт (например, Windows 2000 – до 30 млн строк), не говоря о прикладных программах и базах данных, которые могут занимать в вычислительных системах десятки и сотни Гбайт.

Перефразированный закон Паркинсона гласит: "Программы расширяются, стремясь заполнить весь объем памяти, доступный для их поддержки" (сказано это было об ОС). В идеале программисты хотели бы иметь неограниченную в размере и скорости память, которая была бы энергонезависимой, т.е. сохраняла свое содержимое при выключении электричества, а также недорого бы стоила. Однако реально пока такой памяти нет. В то же время на любом этапе развития технологии производства запоминающих устройств действуют следующие достаточно устойчивые соотношения:

- чем меньше время доступа, тем дороже бит;
- чем выше емкость, тем ниже стоимость бита;
- чем выше емкость, тем больше время доступа.

Чтобы найти выход из сложившейся ситуации, необходимо опираться не на отдельно взятые компоненты или технологию, а выстроить иерархию запоминающих устройств, показанную на рис. 5.1. При перемещении слева направо происходит следующее:

- снижается стоимость бита;
- возрастает емкость;
- возрастает время доступа;
- снижается частота обращений процессора к памяти.

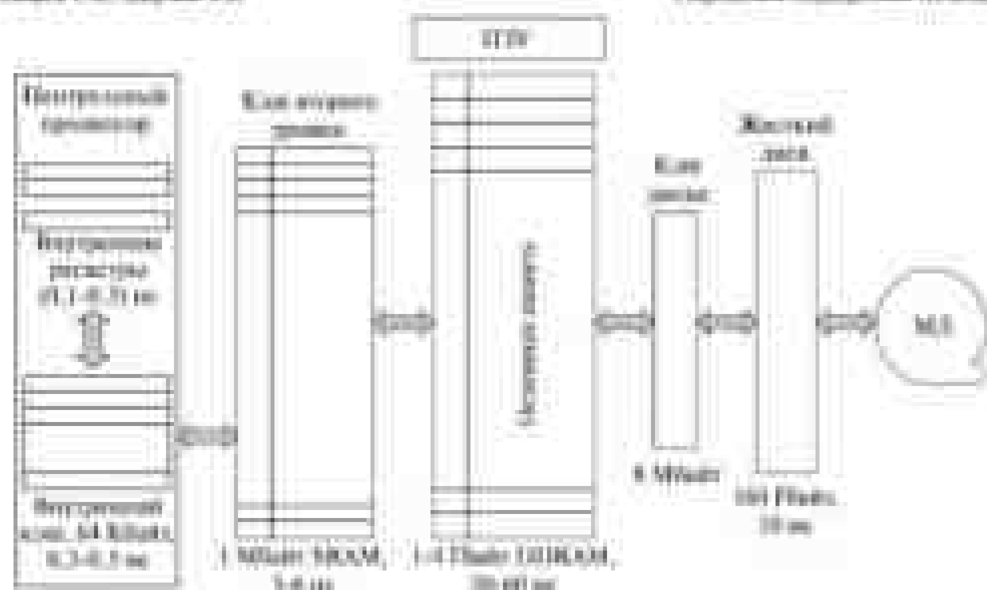


Рис. 6.1. Иерархия памяти

Предположим, процессор имеет доступ к памяти двух уровней. На первом уровне содержится N_1 слов, и он характеризуется временем доступа $T_1 = 1$ нс. К этому уровню процессор может обращаться непосредственно. Однако если требуется получить слово, находящееся на втором уровне, то его сначала нужно передать на первый уровень. При этом передается не только требуемое слово, а блок данных, содержащий это слово. Поскольку адреса, к которым обращается процессор, имеют тенденцию собираться в группы (циклы, подпрограммы), процессор обращается к небольшому повторяющемуся набору команд. Таким образом, работа процессора с вновь полученным блоком памяти будет проходить достаточно длительное время.

Обозначим через $T_2 = 10$ нс время обращения ко второму уровню памяти, а через P – отношение числа нахождений нужного слова в быстрой памяти к числу всех обращений. Пусть в нашем примере $P = 0{,}95$ (т.е. 95% обращений приходится на быструю память, что вполне реально), тогда среднее время доступа к памяти можно записать так:

$$T_{ср} = 0{,}95 \cdot 1нс + 0{,}05 \cdot (1нс + 10нс) = 1{,}55нс$$

Этот принцип можно применять не только к памяти с двумя уровнями. Реально так и происходит. Объем оперативной памяти существенно сказывается на характере протекания вычислительного процесса, так как он ограничивает число одновременно выполняющихся программ, т.е. уровень мультипрограммирования. Если предположить, что процесс проводит часть p своего времени в ожидании завершения операции ввода-вывода, то степень загрузки Z центрального процессора (ЦП) в идеальном случае будет выражаться зависимостью

$Z = 1 - p^n$, где n – число процессов.

На рис. 6.2 показана зависимость $Z=p(n)$ для различного времени ожидания завершения операции ввода-вывода (20%, 50% и 80%) и числа процессов n. Большое количество задач, необходимое для высокой загрузки процессора, требует большого объема оперативной памяти. В условиях, когда для обеспечения приемлемого уровня мультипрограммирования имеющейся памяти недостаточно, был предложен метод организации вычислительного процесса, при котором образы некоторых процессов целиком или частично временно выгружаются на диск.

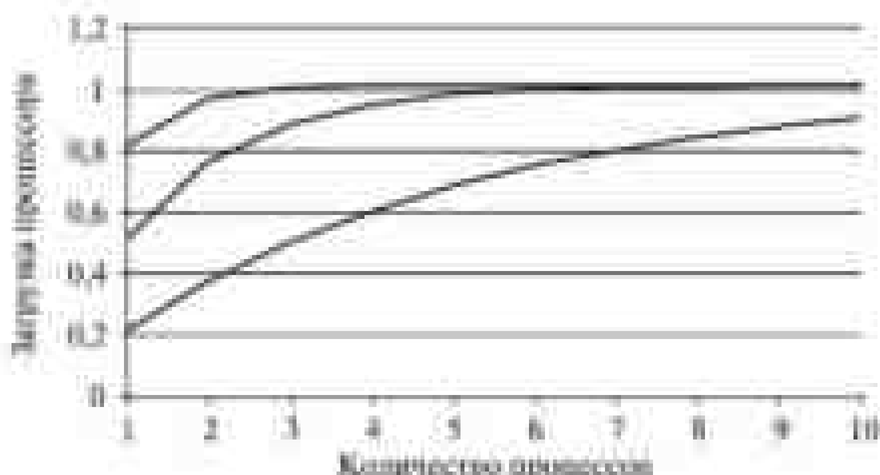


Рис. 6.2. Загрузка процессора при различном числе процессов

Очевидно, что имеет смысл временно выгружать неактивные процессы, находящиеся в ожидании каких-либо ресурсов, в том числе очередного кванта времени центрального процессора. К моменту, когда

пройдет очередь выполнения выгруженного процесса, его образ возвращается с диска в оперативную память. Если при этом обнаруживается, что свободного места в оперативной памяти не хватает, то на диск выгружается другой процесс.

Такая подмена (виртуализация) оперативной памяти дисковой памятью позволяет повысить уровень мультипрограммирования, поскольку объем оперативной памяти теперь не столь жестко ограничивает число одновременно выполняемых процессов. При этом суммарный объем оперативной памяти, занимаемой образами процессов, может существенно превосходить имеющийся объем оперативной памяти.

В данном случае в распоряжение прикладного программиста предоставляется виртуальная оперативная память, размер которой намного превосходит реальную память системы и ограничивается только возможностями адресации используемого процесса (в ПК на базе Pentium 2^{32} = 4 Гбайт). Вообще виртуальным (кажущимся) называется ресурс, обладающий свойствами (в данном случае большой объем ОП), которых в действительности у него нет.

Виртуализация оперативной памяти осуществляется совокупностью аппаратных и программных средств вычислительной системы (схемами процессора и операционной системой) автоматически без участия программиста и не сказывается на логике работы приложения.

Виртуализация памяти возможна на основе двух возможных подходов [17]:

- свопинг (swapping) – образы процессов выгружаются на диск и возвращаются в оперативную память целиком;
- виртуальная память (virtual memory) – между оперативной памятью и диском перемещаются части образов (сегменты, страницы, блоки и т.п.) процессов.

Недостатки свопинга:

- избыточность перемещаемых данных и отсюда замедление работы системы и неэффективное использование памяти;
- невозможность загрузить процесс, виртуальное пространство

которого превышает имеющуюся в наличии свободную память.

Достоинство свопинга по сравнению с виртуальной памятью – меньшие затраты времени на преобразование адресов в кодах программ, поскольку оно делается один раз при загрузке с диска в память (однако это преимущество может быть незначительным, т.к. выполняется при очередной загрузке только часть кода и полностью преобразовывать код, может быть, и не надо).

Виртуальная память не имеет указанных недостатков, но ее ключевой проблемой является преобразование виртуальных адресов в физические (почему это проблема, будет ясно дальше, а пока можно отметить существенные затраты времени на этот процесс, если не принять специальных мер).

6.2. Функции ОС по управлению памятью

Под памятью (memory) в данном случае подразумевается оперативная (основная) память компьютера. В однопрограммных операционных системах основная память разделяется на две части. Одна часть для операционной системы (резидентный монитор, ядро), а вторая – для выполняющейся в текущий момент времени программы. В многопрограммных ОС "пользовательская" часть памяти – важнейший ресурс вычислительной системы – должна быть распределена для размещения нескольких процессов, в том числе процессов ОС. Эта задача распределения выполняется операционной системой динамически специальной подсистемой управления памятью (memory management). Эффективное управление памятью жизненно важно для многозадачных систем. Если в памяти будет находиться небольшое число процессов, то значительную часть времени процессы будут находиться в состоянии ожидания ввода-вывода и загрузка процессора будет низкой.

В ранних ОС управление памятью сводилось просто к загрузке программы и ее данных из некоторого внешнего накопителя (перфоленты, магнитной ленты или магнитного диска) в ОЗУ. При этом память разделялась между программой и ОС. На рис. 6.3 показаны три варианта такой схемы. Первая модель раньше применялась на мэйнфреймах и мини-компьютерах. Вторая схема сейчас используется

на некоторых карманных компьютерах и встроенных системах, третья модель была характерна для ранних персональных компьютеров с MS-DOS.

Рис. 6.3. Варианты распределения памяти

С появлением мультипрограммирования задачи ОС, связанные с распределением имеющейся памяти между несколькими одновременно выполняющимися программами, существенно усложнились.

Функциями ОС по управлению памятью в мультипрограммных системах являются:

- отслеживание (учет) свободной и занятой памяти;
- первоначальное и динамическое выделение памяти процессам приложений и самой операционной системе и освобождение памяти по завершении процессов;
- настройка адресов программы на конкретную область физической памяти;
- полное или частичное вытеснение кодов и данных процессов из ОП на диск, когда размеры ОП недостаточны для размещения всех процессов, и возвращение их в ОП;
- защита памяти, выделенной процессу, от возможных вмешательств со стороны других процессов;
- дефрагментация памяти.

Перечисленные функции особого пояснения не требуют, остановимся

только на этапе преобразования адресов программы при ее загрузке в ОП.

Для идентификации переменных и команд на разных этапах жизненного цикла программы используются символьные имена, виртуальные (математические, условные, логические – все это синонимы) и физические адреса (рис. 6.4).

Рис. 6.4. Типы адресов

Символьные имена присваивает пользователь при написании программ на алгоритмическом языке или ассемблере. Виртуальные адреса вырабатывает транслятор, переводящий программу на машинный язык. Поскольку во время трансляции неизвестно, в какое место оперативной памяти будет загружена программа, транслятор присваивает переменным и командам виртуальные (условные) адреса, считая по умолчанию, что начальным адресом программы будет нулевой адрес.

Физические адреса соответствуют номерам ячеек оперативной памяти, где в действительности будут расположены переменные и команды.

Совокупность виртуальных адресов процесса называется виртуальным адресным пространством. Диапазон адресов виртуального пространства у всех процессов один и тот же и определяется разрядностью адреса процессора (для Pentium адресное пространство составляет объем, равный 2^{32} байт, с диапазоном адресов от $0000\,0000_{16}$ до $FFFFFFFF_{16}$).

Существует два принципиально отличающихся подхода к преобразованию виртуальных адресов в физические. В первом случае такое преобразование выполняется один раз для каждого процесса во время начальной загрузки программы в память. Преобразование осуществляет перемещающий загрузчик на основании имеющихся у него данных о начальном адресе физической памяти, в которую предстоит загружать программу, а также информации, предоставляемой транслятором об адресно-зависимых элементах программы.

Второй способ заключается в том, что программа загружается в память в виртуальных адресах. Во время выполнения программы при каждом обращении к памяти операционная система преобразует виртуальные адреса в физические.

6.3. Распределение памяти

Существует ряд базовых вопросов управления памятью, которые в различных ОС решаются по-разному. Например, следует ли назначать каждому процессу одну непрерывную область физической памяти или можно выделять память участками? Должны ли сегменты программы, загруженные в память, находиться на одном месте в течение всего периода выполнения процесса или их можно время от времени сдвигать? Что делать, если сегменты программы не помещаются в имеющуюся память? Как сократить затраты ресурсов системы на управление памятью? Имеется и ряд других не менее интересных проблем управления памятью [5, 10, 13, 17].

Ниже приводится классификация методов распределения памяти, в которой выделено два класса методов – с перемещением сегментов процессов между ОП и ВП (диском) и без перемещения, т.е. без привлечения внешней памяти (рис. 6.5). Данная классификация

учитывает только основные признаки методов. Для каждого метода может быть использовано несколько различных алгоритмов его реализации.

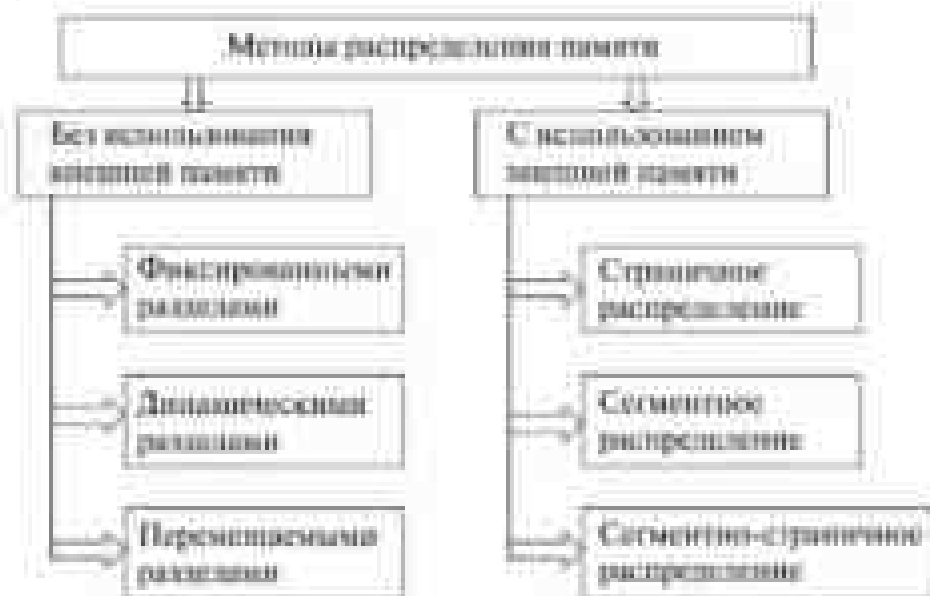


Рис. 6.5. Классификация методов распределения памяти

На рис. 6.6 показаны два примера фиксированного распределения. Одна возможность состоит в использовании разделов одинакового размера. В этом случае любой процесс, размер которого не превышает размера раздела, может быть загружен в любой доступный раздел. Если все разделы заняты и нет ни одного процесса в состоянии готовности или работы, ОС может выгрузить процесс из любого раздела и загрузить другой процесс, обеспечивая тем самым процессор работой.

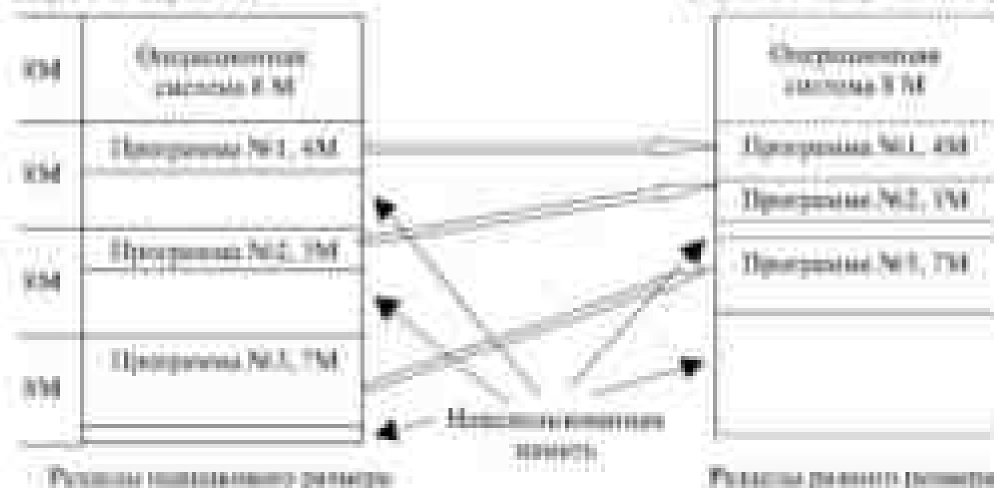


Рис. 6.6. Варианты фиксированного распределения памяти

При использовании разделов с одинаковым размером имеются две проблемы.

1. Программа может быть слишком велика для размещения в разделе. В этом случае программист должен разрабатывать программу, использующую оверлеи, чтобы в любой момент времени требовался только один раздел памяти. Когда требуется модуль, отсутствующий в данный момент в ОП, пользовательская программа должна сама его загрузить в раздел памяти программы. Таким образом, в данном случае управление памятью во многом возлагается на программиста.
2. Использование ОП крайне неэффективно. Любая программа, независимо от ее размера, занимает раздел целиком. При этом могут оставаться неиспользованные участки памяти большого размера. Этот феномен появления неиспользованной памяти называется внутренней фрагментацией (internal fragmentation).

Бороться с этими трудностями (хотя и не устранить полностью) можно посредством использования разделов разных размеров. В этом случае программа размером до 8 Мбайт может обойтись без оверлеев, а разделы малого размера позволяют уменьшить внутреннюю фрагментацию при загрузке небольших программ.

В том случае, когда разделы имеют одинаковый размер, размещение процессов тривиально – в любой свободный раздел. Если все разделы заняты процессами, которые не готовы к немедленной работе, любой из них может быть выгружен для освобождения памяти для нового процесса.

Когда разделы имеют разные размеры, есть два возможных подхода к назначению процессов разделам памяти. Простейший путь состоит в том, чтобы каждый процесс размещался в наименьшем разделе, способном вместить данный процесс (в этом случае в задании пользователя указывался размер требуемой памяти). При таком подходе для каждого раздела требуется очередь планировщика, в которой хранятся выгруженные из памяти процессы, предназначенные для данного раздела памяти. Достоинство такого способа в возможности распределения процессов между разделами ОП так, чтобы минимизировать внутреннюю фрагментацию.

Недостаток заключается в том, что отдельные очереди для разделов могут привести к неоптимальному распределению памяти системы в целом. Например, если в некоторый момент времени нет ни одного процесса размером от 7 до 12 Мбайт, то раздел размером 12 Мбайт будет пустовать, в то время как он мог бы использоваться меньшими процессами. Поэтому более предпочтительным является использование одной очереди для всех процессов. В момент, когда требуется загрузить процесс в ОП, выбирается наименьший доступный раздел, способный вместить данный процесс.

В целом можно отметить, что схемы с фиксированными разделами относительно просты, предъявляют минимальные требования к операционной системе; накладные расходы работы процессора на распределение памяти невелики. Однако у этих схем имеются серьезные недостатки.

1. Количество разделов, определенное в момент генерации системы, ограничивает количество активных процессов (т.е. уровень мультипрограммирования).
2. Поскольку размеры разделов устанавливаются заранее во время генерации системы, небольшие задания приводят к неэффективному использованию памяти. В средах, где заранее

известны потребности в памяти всех задач, применение рассмотренной схемы может быть оправдано, но в большинстве случаев эффективность этой технологии крайне низка.

Для преодоления сложностей, связанных с фиксированным распределением, был разработан альтернативный подход, известный как динамическое распределение. В свое время этот подход был применен фирмой IBM в операционной системе для мэйнфреймов в OS/MVT (мультипрограммирование с переменным числом задач – Multiprogramming With a Variable number of Tasks). Позже этот же подход к распределению памяти использован в ОС ЕС ЭВМ [12].

При динамическом распределении образуется переменное количество разделов переменной длины. При размещении процесса в основной памяти для него выделяется строго необходимое количество памяти. В качестве примера рассмотрим использование 64 Мбайт (рис. 6.7) основной памяти. Изначально вся память пуста, за исключением области, задействованной ОС. Первые три процесса загружаются в память, начиная с адреса, где заканчивается ОС, и используют столько памяти, сколько требуется данному процессу. После этого в конце ОП остается свободный участок памяти, слишком малый для размещения четвертого процесса. В некоторый момент времени все процессы в памяти оказываются неактивными, и операционная система выгружает второй процесс, после чего остается достаточно памяти для загрузки нового, четвертого процесса.

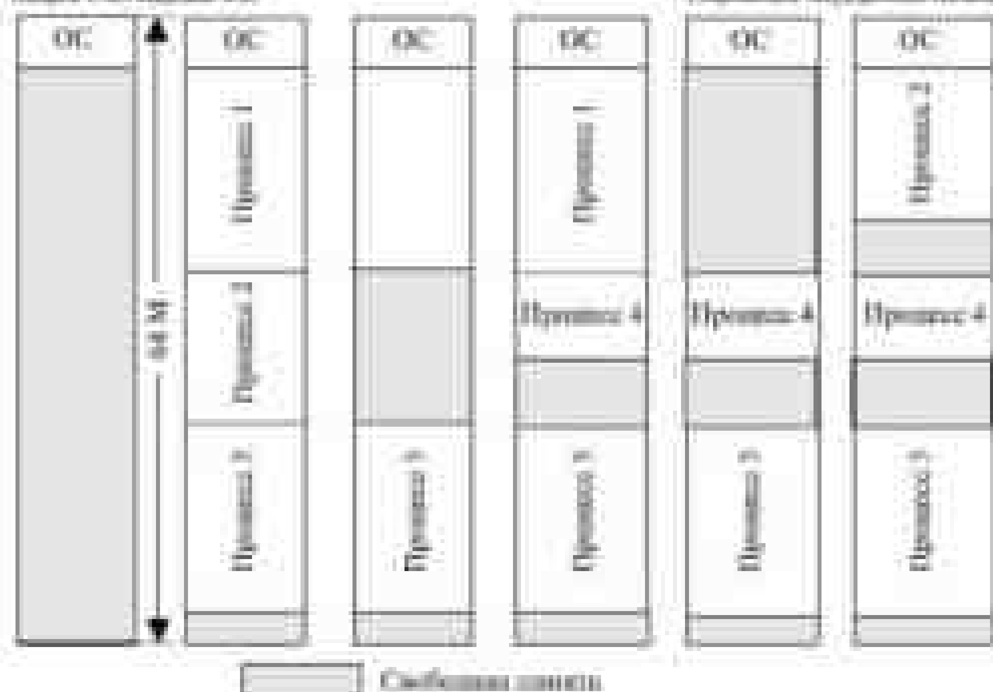


Рис. 6.7. Вариант использования памяти

Поскольку процесс 4 меньше процесса 2, появляется еще свободный участок памяти. После того как в некоторый момент времени все процессы оказались неактивными, но стал готовым к работе процесс 2, свободного места в памяти для него не находится, а ОС вынуждена выгрузить процесс 1, чтобы освободить необходимое место и разместить процесс 2 в ОП. Как показывает данный пример, этот метод хорошо начинает работу, но плохо продолжает. В конечном счете, он приводит к наличию множества мелких свободных участков памяти, в которых нет возможности разместить какой-либо новый процесс. Это явление называется внешней фрагментацией (external fragmentation), что отражает тот факт, что сильно фрагментированной становится память, внешняя по отношению ко всем разделам.

Один из методов преодоления внешней фрагментации – уплотнение (compaction) процессов в ОП. Осуществляется это перемещением всех занятых участков так, чтобы вся свободная память образовала единую свободную область. В дополнение к функциям, которые ОС выполняет при распределении памяти динамическими разделами, в данном случае

она должна еще время от времени копировать содержимое разделов из одного места в другое, корректируя таблицы свободных и занятых областей. Эта процедура называется уплотнением или сжатием.

Перечислим функции операционной системы по управлению памятью в этом случае.

1. Перемещение всех занятых участков в сторону старших или младших адресов при каждом завершении процесса или для вновь создаваемого процесса в случае отсутствия раздела достаточного размера.
2. Коррекция таблиц свободных и занятых областей.
3. Изменение адресов команд и данных, к которым обращаются процессы при их перемещении в памяти, за счет использования относительной адресации.
4. Аппаратная поддержка процесса динамического преобразования относительных адресов в абсолютные адреса основной памяти.
5. Защита памяти, выделяемой процессу, от взаимного влияния других процессов.

Уплотнение может выполняться либо при каждом завершении процесса, либо только тогда, когда для вновь создаваемого процесса нет свободного раздела достаточного размера. В первом случае требуется меньше вычислительной работы при корректировке таблиц свободных и занятых областей, а во втором – реже выполняется процедура сжатия.

Так как программа перемещается по оперативной памяти в ходе своего выполнения, в данном случае невозможно выполнить настройку адресов с помощью перемещающего загрузчика. Здесь более подходящим оказывается динамическое преобразование адресов. Достоинствами распределения памяти перемещаемыми разделами является эффективное использование оперативной памяти, исключение внутренней и внешней фрагментации, недостатком – дополнительные накладные расходы ОС.

При использовании фиксированной схемы распределения процесс всегда будет назначаться одному и тому же разделу памяти после его выгрузки и последующей загрузки в память. Это позволяет применять простейший загрузчик, который замещает при загрузке процесса все

относительные ссылки абсолютными адресами памяти, определенными на основе базового адреса загруженного процесса.

Ситуация усложняется, если размеры разделов равны (или неравны) и существует единая очередь процессов, – процесс по ходу работы может занимать разные разделы. Такая же ситуация возможна и при динамическом распределении. В этих случаях расположение команд и данных, к которым обращается процесс, не является фиксированным и изменяется всякий раз при выгрузке, загрузке или перемещении процесса. Для решения этой проблемы в программах используются относительные адреса. Это означает, что все ссылки на память в загружаемом процессе даются относительно начала этой программы. Таким образом, для корректной работы программы требуется аппаратный механизм, который бы транслировал относительные адреса в физические в процессе выполнения команды, обращающейся к памяти.

Применяемый обычно способ трансляции показан на рис. 5.8. Когда процесс переходит в состояние выполнения, в специальный регистр процесса, называемый базовым, загружается начальный адрес процесса в основной памяти. Кроме того, используется "граничный" (bounds) регистр, в котором содержится адрес последней ячейки программы. Эти значения заносятся в регистры при загрузке программы в основную память. При выполнении процесса относительные адреса в командах обрабатываются процессором в два этапа. Сначала к относительному адресу прибавляется значение базового регистра для получения абсолютного адреса. Затем полученный абсолютный адрес сравнивается со значением в граничном регистре. Если полученный абсолютный адрес принадлежит данному процессу, команда может быть выполнена. В противном случае генерируется соответствующее данной ошибке прерывание.

ОС
Начальный адрес процесса
Относительный адрес команды (данных)
Управляющий блок процесса
Базовый регистр ОС
Программа
Сумматор
ОС
Граничный регистр
Компаратор
Данные
Прерывание ОС
Абсолютный адрес
Стек

Рис. 6.8. Преобразование адресов

6.4. Страничная организация виртуальной памяти

Большинство систем виртуальной памяти используют технику, называемую страничной организацией памяти [32, 37]. Любой процесс, реализуемый в компьютере, может обратиться к множеству адресов в памяти. Адреса могут формироваться с применением индексации, базовых регистров, сегментных регистров и другими путями. Эти программно формируемые адреса, называемые виртуальными адресами, формируют виртуальное адресное пространство. На компьютерах без виртуальной памяти виртуальные адреса подаются непосредственно на шину памяти и вызывают для чтения или записи слово в физической памяти с тем же самым адресом.

Когда используется виртуальная память, виртуальные адреса не передаются напрямую шиной памяти. Вместо этого они передаются диспетчеру памяти (*MMU – Memory Management Unit*), который отображает виртуальные адреса на физические адреса памяти, как показано на рис. 6.9. Здесь диспетчер памяти показан как часть

микросхемы процессора, как обычно и бывает чаще всего. Но логически он мог бы быть отдельной микросхемой, как было в недавнем прошлом.

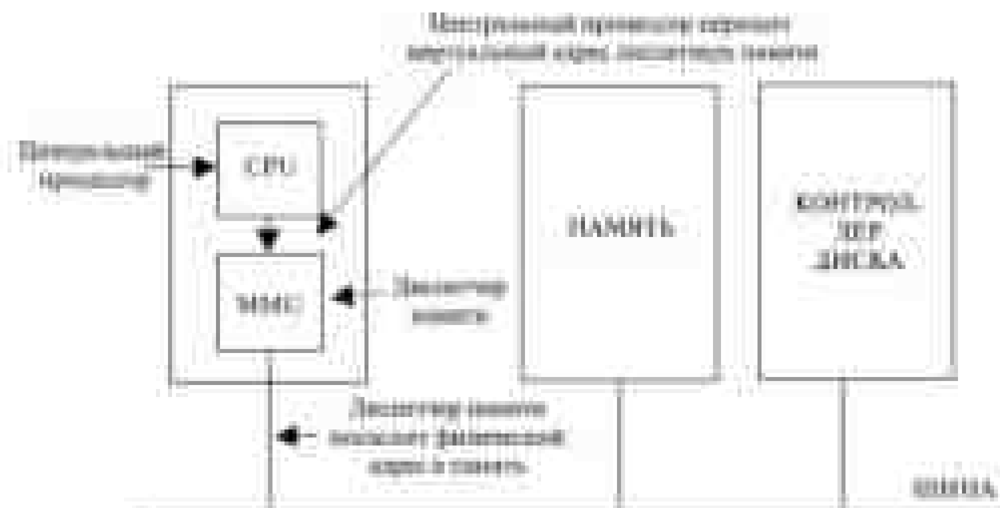


Рис. 6.9. Диспетчер памяти

Все имеющиеся в настоящее время множество реализаций виртуальной памяти различается в основном способом структуризации виртуального адресного пространства.

Сам термин "виртуальная память" ассоциируется с системами, использующими страничную организацию. Впервые сообщение о виртуальной памяти на основе страничной организации появилось в 1962 году в работе Kilburn J. и др. "One-Level Storage System", и вскоре после этого виртуальная память стала широко применяться в коммерческих системах.

В настоящее время выделяют три метода реализации виртуальной памяти.

1. Страничная виртуальная память организует перемещение данных между основной памятью и диском страницами – частями виртуального адресного пространства фиксированного и сравнительно небольшого размера.
2. Сегментная виртуальная память предусматривает перемещение данных сегментами – частями виртуального адресного

пространства произвольного размера, полученного с учетом смыслового значения данных.

3. Сегментно-страничная виртуальная память использует двухуровневое деление: виртуальное адресное пространство делится на сегменты, а затем сегменты делятся на страницы. Единицей перемещения данных является страница.

Для временного хранения сегментов и страниц на диске отводится специальная область либо специальный файл (страничный файл или файл подкачки – paging file). Текущий размер страничного файла является важным параметром, оказывающим влияние на возможности операционной системы: чем больше страничный файл, тем больше приложений может одновременно выполнять ОС (при фиксированном размере оперативной памяти). Однако необходимо понимать, что увеличение числа одновременно работающих приложений за счет увеличения размера страничного файла замедляет их работу, так как значительная часть времени при этом тратится на перемещение данных на диск и обратно.

Размер страничного файла в современных ОС является настраиваемым параметром, который выбирается администратором системы для достижения компромисса между уровнем программирования и быстродействием системы.

При страничной организации виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые виртуальными страницами (virtual pages). В общем случае размер виртуального адресного пространства не кратен размеру страницы, поэтому последняя страница дополняется фиктивной областью.

Вся оперативная память машины также делится на части такого же размера, называемые физическими страницами (или блоками, или кадрами). Размер страницы выбирается равным степени двойки: 1024, 2048, 4096 байт и т.д. Это позволяет упростить механизм преобразования адресов.

При создании процесса ОС загружает в оперативную память несколько его виртуальных страниц (начальные страницы кодового

сегмента и сегмента данных). Копия всего виртуального адресного пространства процесса находится на диске. Смежные виртуальные страницы не обязательно находятся в смежных физических страницах. Для каждого процесса ОС создает таблицу страниц – информационную структуру, содержащую записи обо всех виртуальных страницах процесса (рис. 6.10).

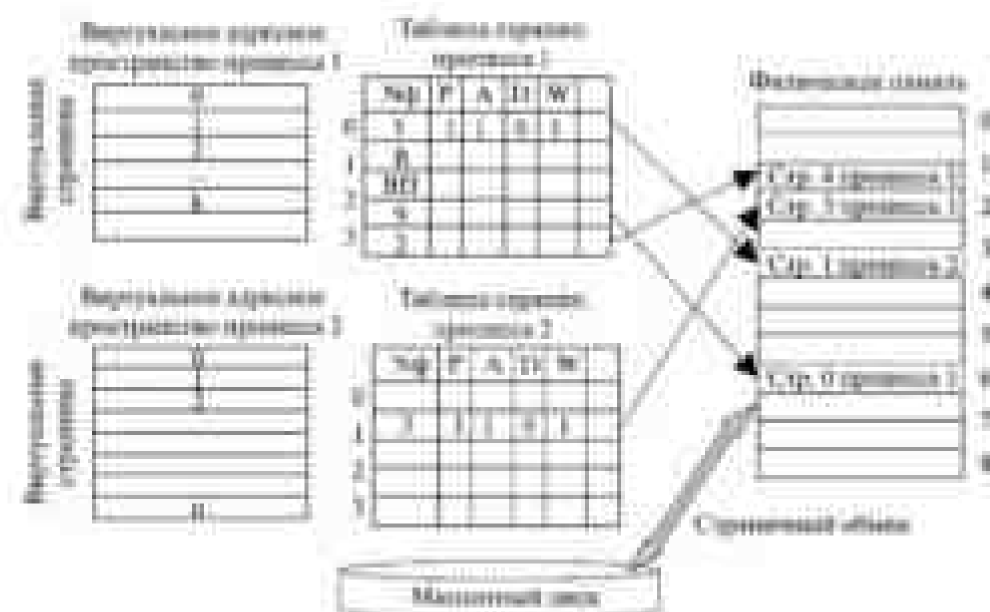


Рис. 6.10. Таблицы страниц виртуальной памяти

Запись таблицы (дескриптор страницы) включает следующую информацию:

1. номер физической страницы (N ф.с.), в которую загружена данная виртуальная страница;
2. признак присутствия P, устанавливаемый в единицу, если данная страница находится в оперативной памяти;
3. признак модификации страницы D, который устанавливается в единицу всякий раз, когда производится запись по адресу, относящемуся к данной странице;
4. признак обращения A к странице, называемый также битом доступа, который устанавливается в единицу при каждом обращении по адресу, относящемуся к данной странице;

3. другие управляющие биты, служащие, например, для целей защиты или совместного использования памяти на уровне страниц.

Перечисленные признаки в большинстве моделей процессоров устанавливаются аппаратно схемами процессора при выполнении операций с памятью. Информация из таблицы страниц используется для решения вопроса о необходимости перемещения той или иной страницы между памятью и диском, а также для преобразования виртуального адреса в физический. Сами таблицы страниц, так же как и описываемые ими страницы, размещаются в оперативной памяти.

Поскольку процесс может задействовать больший объем виртуальной памяти (например, в Windows 2000 он равен 2^{32} = 4 Гбайт), при использовании страницы объемом 4 Кбайт (2^{12}) потребуется 2^{20} записей в таблице страниц для каждого процесса. Понятно, что выделять такое количество оперативной памяти под таблицы страниц нецелесообразно. Для преодоления этой проблемы большинство схем виртуальной памяти хранят таблицы страниц не в реальной, а в виртуальной памяти. Это означает, что сами таблицы страниц становятся объектами страничной организации. При работе процесса как минимум часть его таблицы страниц должна располагаться в основной памяти, в том числе запись о странице, выполняющейся в настоящий момент. Адрес таблицы страниц включается в контекст процесса. При активизации очередного процесса ОС загружает адрес его таблицы страниц в специальный регистр.

При каждом обращении к памяти выполняется поиск номера виртуальной страницы, содержащей требуемый адрес, затем по этому номеру определяется нужный элемент таблицы страниц и из него извлекается описывающая страницу информация. Далее анализируется признак присутствия, и если данная виртуальная страница находится в оперативной памяти, то выполняется преобразование виртуального адреса в физический. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит страничное прерывание.

Выполняющий процесс переводится в состояние ожидания, активизируя процесс из очереди процессов, находящихся в состоянии готовности. Параллельно программа обработки страничного

прерывания находит на диске требуемую виртуальную страницу и пытается ее загрузить в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно. Если же свободных страниц нет, то на основании принятой в данной системе стратегии замещения страниц решается вопрос о том, какую страницу следует выгрузить из оперативной памяти.

После того как выбрана страница, которая должна покинуть оперативную память, обнуляется ее бит присутствия и анализируется ее признак модификации. Если удаляемая страница за время последнего пребывания в оперативной памяти была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то принимается во внимание, что на диске уже имеется предыдущая копия этой виртуальной страницы, и никакой записи на диск не производится. Физическая страница объявляется свободной. Из соображений безопасности в некоторых системах освобождаемая страница обнуляется, чтобы невозможно было использовать содержимое выгруженной страницы. Для хранения информации о положении вытесненной страницы в страничном файле ОС может задействовать специальные поля таблицы страниц.

Виртуальный адрес при страничном распределении может быть представлен в виде пары (p, s_v), где p – номер виртуальной страницы процесса (нумерация страниц начинается с 0), а s_v – смещение в пределах виртуальной страницы (рис. 6.11). Физический адрес также может быть представлен в виде пары (n, s_f), где n – номер физической страницы, а s_f – смещение в пределах физической страницы. Задача подсистемы виртуальной памяти состоит в отображении пары значений (p, s_v) в пару (n, s_f).

Рис. 6.11. Преобразование виртуального адреса

Чтобы понять механизм реализации этого отображения, следует остановиться на двух базисных свойствах страничной организации. Как уже отмечалось, объем страницы, как виртуальной, так и физической, выбирается равным степени двойки – 2^k (k = 9 и более). Отсюда следует, что смещение S_v и S_f может быть получено отделением k младших разрядов в двоичной записи виртуального и, соответственно, физического адреса страницы. При этом оставшиеся старшие разряды адреса представляют собой двоичную запись номера виртуальной и, соответственно, физической страницы. Дополнив эти номера k нулями, можно получить начальный адрес виртуальной и физической страниц.

Второе свойство – линейность адресного пространства виртуальной и физической страницы – приводит к тому, что $S_f = S_v$. Отсюда следует простая схема преобразования виртуального адреса в физический.

При обращении к памяти по некоторому виртуальному адресу (P_v, S_v) аппаратные схемы процессора выполняют следующие действия.

1. Из специального регистра процессора извлекается начальный адрес AT таблицы страниц активного процесса. С помощью

сумматора $\sum$ по значениям AT, P, L (длина отдельной записи в таблице страниц) определяется адрес нужной записи в таблице страниц:

$$A = AT + (P * L).$$

2. Считывается номер соответствующей физической страницы – N.
3. К номеру физической страницы присоединяется смещение $E_{см}$.

В итоге полученный физический адрес оперативной памяти представляется парой значений (N, $E_{см}$).

Рассмотрим пример, поясняющий основные характеристики организации страничной виртуальной памяти. Пусть компьютер имеет оперативную память объемом $E_{оп}$ = 256 Мбайт, размер страницы выбран равным $E_{стр}$ = 4 Кбайт. В этом случае количество физических страниц равно

$$N1 = E_{оп} / E_{стр} = 256*20^{20} / 4*2^{10} = 64.000 \text{ страниц.}$$

Для отображения физического адреса произвольного байта оперативной памяти потребуется $N = \log_2 256*2^{20} = 28$ двоичных разрядов.

Число разрядов для отображения смещения в странице $M = \log_2 4$ Кбайт $= \log_2 4096 = 12$.

Если процессор имеет 32-разрядную структуру, то на номер виртуальной страницы отводится 32–12=20 двоичных разрядов. Таким образом, число виртуальных страниц равно $N_в$ = 220 (примерно 1 млн виртуальных страниц).

Для каждой виртуальной страницы в таблице страниц должна быть запись, содержащая номер виртуальной страницы (20 двоичных разрядов), начальный адрес соответствующей ей физической страницы плюс дополнительные разряды, характеризующие свойства страницы (присутствие, модификация, обращение и т.п.), на которые потребуется 1 байт. Поскольку адрес начала физической страницы кратен 4096, то на

нужно достаточно $28 - 12 = 16$ двоичных разрядов (остальные 12 разрядов заполняются нулями). Таким образом, одна запись таблицы страниц займет $20 + 16 + 8 = 44$ двоичных разряда или 6 байт. Общий объем таблицы страниц составит $6 * 1М = 6$ Мбайт.

Реально при выборе структуры записи таблицы страниц нужно учитывать следующие факторы. Современные компьютеры позволяют наращивать объем оперативной памяти (например, в ПК она может почти достигать объема виртуальной памяти и даже более). Поэтому на адрес физической страницы в нашем примере следует выделить $32 - 12 = 20$ двоичных разрядов. С другой стороны, нет необходимости в записи (дескрипторе) виртуальной страницы иметь поле с номером виртуальной страницы (20 разрядов), так как адрес нужной записи можно вычислить, как это было рассмотрено выше. Следовательно, в нашем примере длина записи должна быть равной $32 - 12 + 8 = 28$ двоичным разрядам, т.е. с округлением до целого числа байт – 4 байт. Таким образом, для каждого выполняющегося в компьютере процесса ОС должна создать страничную таблицу размером $4 * 1М$ байт $= 4 * 2^{20} = 4$ Мбайт.

Процедура преобразования виртуального адреса в физический без принятия специальных мер (кэширование активных страниц) занимает один цикл оперативной памяти, который затрачивается на считывание номера физической страницы из таблицы страниц. Поэтому любое обращение к ОП будет занимать 2 цикла вместо одного при работе без виртуальной памяти. Другим фактором, влияющим на производительность систем, являются затраты времени на обработку страничных прерываний. При неправильно выбранной стратегии замещения страниц может возникнуть ситуация, когда система тратит большую часть времени впустую на подкачку страниц из оперативной памяти на диск и обратно.

6.5. Оптимизация функционирования страничной виртуальной памяти

В настоящее время известно несколько методов повышения эффективности функционирования страничной виртуальной памяти. К ним относятся [17]:

1. более сложная структуризация виртуального адресного пространства, например, двухуровневая (типичная для 32-битовой адресации);
2. использование специального высокоскоростного кэша для хранения части записей таблицы страниц, который обычно называют буфером быстрого преобразования адреса, или буфером поиска трансляции (translation lookaside buffer – TLB);
3. выбор оптимального размера страницы виртуальной памяти;
4. эффективное управление страничным обменом.

Остановимся на возможностях реализации этих методов.

Рассмотрим вариант двухуровневой страничной организации. При такой схеме имеется каталог таблиц страниц, в котором каждая запись указывает на таблицу страниц. Таким образом, если размер каталога – X, а максимальный размер таблицы страниц – Y, то процесс может состоять максимум из X и Y страниц. Обычно максимальный размер таблицы страниц определяется условием ее размещения на одной странице (такой подход используется в процессоре Pentium).

На рис. 8.12 приведен пример двухуровневой схемы, типичной для 32-битовой адресации. Подобная схема позволяет существенно сократить размер пользовательской таблицы страниц, размещаемой в основной памяти (с 4 Мбайт до 4 Кбайт). В данном случае виртуальное адресное пространство пользовательского процесса может составлять 2^{32} = 4 Гбайт. При объеме страницы 2^{12} = 4 Кбайт в этом пространстве размещается $2^{32} / 2^{12} = 2^{20}$ страниц. Таким образом, пользовательская таблица страниц будет иметь 2^{20} 4-байтных записей общим объемом 4 Мбайт. Разместить такие таблицы для нескольких процессов в ОП нереально. В двухуровневой схеме это и не требуется. В основной памяти постоянно находится корневая таблица, содержащая 1024 записей, указывающих на начальный адрес пользовательской таблицы страниц (ее объем, как указано выше, 4 Мбайт). Указатель на начальный адрес корневой таблицы (активного процесса) заносится в регистр процессора. Первые 10 бит виртуального адреса используются для индексации в корневой таблице для поиска записей о странице пользовательской таблицы.

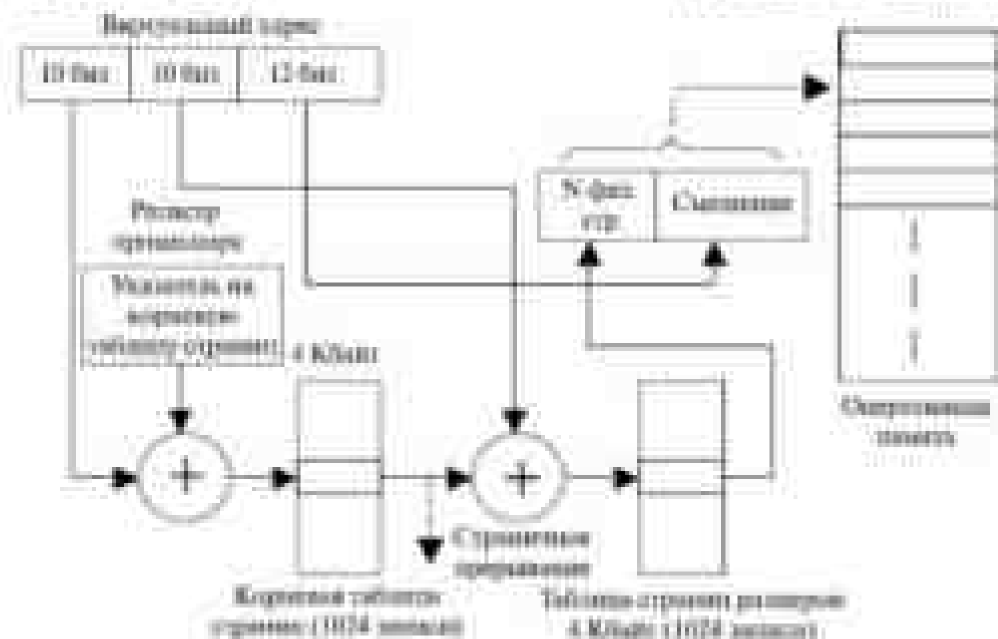


Рис. 6.12. Двухуровневая схема таблиц страниц

Если страница находится в ОП, то следующие 12 бит виртуального адреса используются для задания смещения в физической странице ОП. В противном случае генерируется страничное прерывание, но уже из-за отсутствия нужной страницы процесса в ОП.

Таким образом, двухуровневая схема, сокращая объем памяти для хранения таблицы страниц, в общем случае замедляет преобразование виртуального адреса за счет большего числа возможных страничных прерываний (даже если нет страничного прерывания, требуется три цикла ОП вместо двух при одноуровневой страничной организации).

Как уже отмечалось, простая схема страничной виртуальной памяти, по сути, удваивает время обращения к памяти. Для преодоления этой проблемы большинство реально применяющихся схем виртуальной памяти использует специальный высокоскоростной кэш для записей таблицы страниц, который обычно называют буфером быстрой трансляции адресов – TLB. Этот кэш функционирует так же как и обычный кэш памяти и содержит те записи таблицы страниц, которые использовались последними. Организация аппаратной поддержки использования TLB показана на рис. 6.13.

Получив виртуальный адрес, процессор просматривает TLB. Если требуемая запись найдена, процессор получает адрес физической страницы и формирует реальный адрес. Если запись в TLB не найдена, то процессор берет номер виртуальной страницы в качестве индекса для таблицы страниц процесса и просматривает соответствующую запись. Если бит присутствия в ней установлен, значит, искомая страница находится в основной памяти, и процессор получает номер физической страницы из записи таблицы страниц, а затем формирует реальный физический адрес. Одновременно вносится использованная запись таблицы страниц в TLB.

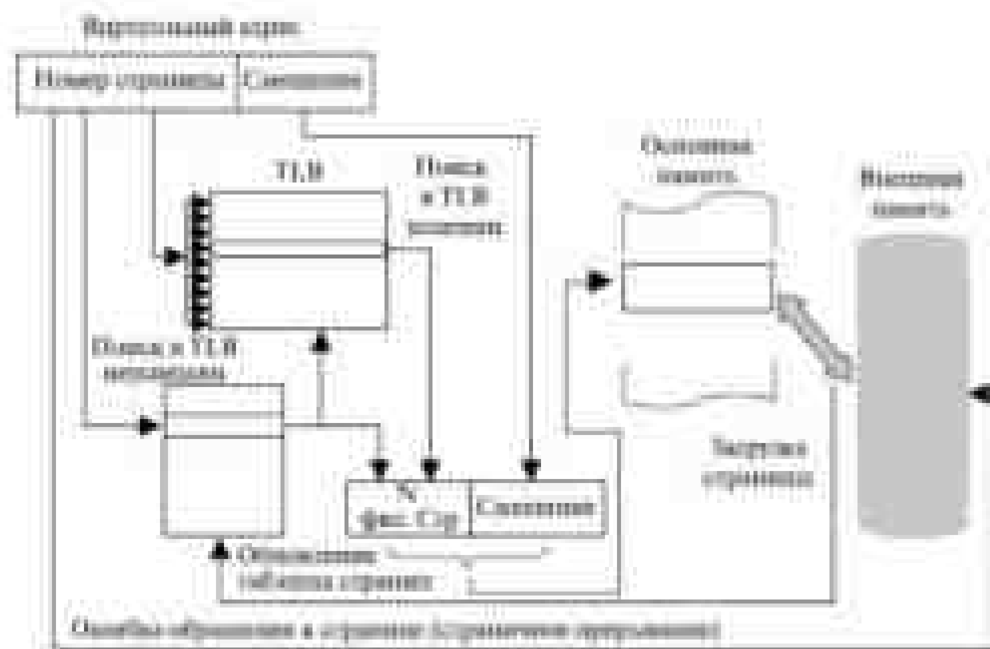


Рис. 6.13. Буфер быстрой переадресации

Если бит присутствия данной виртуальной страницы не установлен, это означает, что искомой страницы в оперативной памяти нет. В этом случае процессор генерирует сигнал страничного прерывания, активизирующий операционную систему, которая загружает требуемую страницу в оперативную память и обновляет таблицу страниц.

Практика использования виртуальной памяти показала, что для нее справедлив закон локализации большинства обращений в небольшом

количество недавно использованных страниц. При этом соответствующие записи будут находиться в кэше, так что с помощью TLB существенно повышается эффективность работы виртуальной памяти.

В организации TLB имеется ряд особенностей. Поскольку TLB содержит только некоторые из записей таблицы страниц (32 в процессоре Pentium), индексация записей в TLB на основе номера страницы не представляется возможной. Вместо этого каждая запись TLB должна наряду с полной информацией из записи таблицы страниц включать также номер виртуальной страницы. Процессор аппаратно способен одновременно опрашивать все записи TLB для определения того, какая из них соответствует заданному номеру страницы. Такой подход известен как ассоциативное отображение (associative mapping), в отличие от прямого отображения, или индексирования, применяемого для поиска в таблице страниц, как показано на рис. 6.14.

Рис. 6.14. Ассоциативная память

Конструкция TLB должна также предусматривать способы организации записей в кэш и принятия решения о том, какая из старых записей должна быть удалена при внесении в кэш новой записи.

Следует подчеркнуть, что механизм виртуальной памяти должен взаимодействовать с кэшем оперативной памяти (кроме TLB). Это взаимодействие показано на рис. 6.15. Сначала происходит обращение к TLB для выяснения наличия в нем соответствующей записи таблицы страниц. При положительном результате путем объединения номера физической страницы, получаемого из TLB, и смещения генерируется физический адрес. Если требуемой записи в TLB нет, она выбирается из таблицы страниц. После получения физического адреса в обеих ситуациях выполняется обращение к кэшу для выяснения наличия в нем блока с требуемым физическим адресом. Если ответ положительный, то требуемое значение (код или данные) передается процессору. В противном случае производится выборка слова из основной памяти и обновляется содержимое кэша основной памяти.

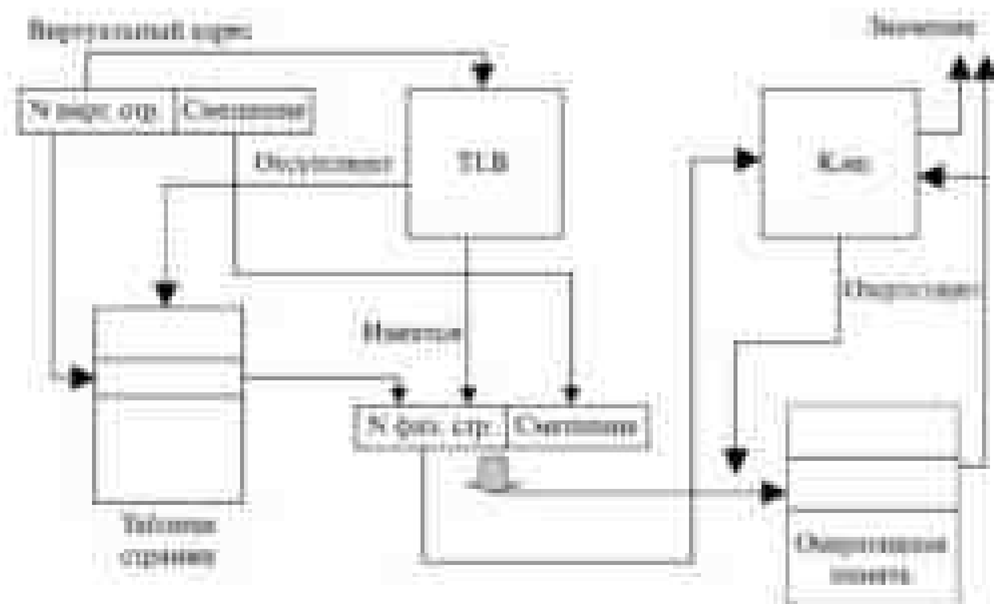


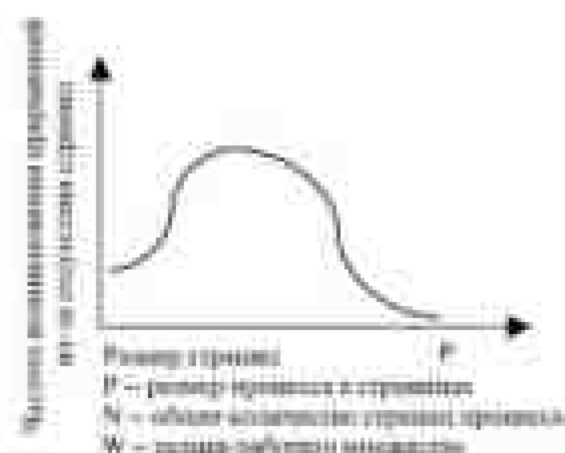
Рис. 6.15. Использование кэша оперативной памяти

При выборе размера страницы нужно учитывать несколько факторов. Один из них – внутренняя фрагментация, которая напрямую зависит от размера страницы. Внутренняя фрагментация уменьшается с уменьшением размера страницы. Однако, чем меньше страницы, тем больше их требуется для процесса, что означает увеличение размера таблицы страниц. При этом для больших программ в загруженной

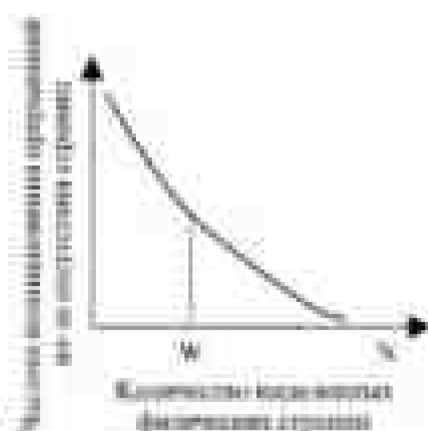
многозадачной среде это приведет к тому, что часть страничных таблиц активных процессов будет находиться в виртуальной памяти, и при отсутствии страницы будет возникать двойное прерывание: первое – для получения требуемой записи из таблицы страниц, второе – для получения доступа к требуемой странице процесса.

Такое двойное прерывание существенно снизит производительность виртуальной памяти. Кроме того, следует учитывать и факт повышения скорости работы диска при передаче больших блоков данных. Таким образом, страницы небольшого размера нецелесообразны, поскольку уменьшение внутренней фрагментации в этом случае не столь значительно по сравнению со снижением производительности виртуальной памяти.

Проблема выбора размера страницы усложняется еще и тем, что размер страницы влияет и на частоту возникновения прерывания из-за отсутствия страницы в основной памяти. На рис. 5.15 а) показан примерный график изменения частоты страничных прерываний из-за отсутствия страницы с учетом принципа локализации. Если размер страницы очень мал, в памяти размещается относительно большое число страниц процесса. Через некоторое время страницы в памяти будут содержать части процесса, сосредоточенные вблизи последних обращений, и частота прерываний из-за отсутствия страницы должна быть невелика.



а)



б)

Рис. 6.16. Изменение частоты страничных прерываний

По мере увеличения размера страницы каждая отдельная страница будет содержать данные, которые располагаются все дальше и дальше от последних выполненных обращений к памяти. Действие принципа локализации ослабевает, и наблюдается рост количества прерываний из-за отсутствия страницы. С дальнейшим ростом размера страницы он (размер) становится сравнимым с размером процесса (точка P на графике) и прерывания становятся реже, а достижения размера этого процесса прекращаются.

Следует учитывать также влияние количества физических страниц, распределенных процессу. На рис. 6.16 б) показано, что для фиксированного размера страницы частота прерываний из-за отсутствия страницы уменьшается с ростом числа страниц, находящихся в основной памяти.

Реально размеры страниц различных компьютеров составляют следующие значения: 512 байт (семейство VAX, IBM AS/400), 4 Кбайт (IBM 370, MIPS), 8 Кбайт (DEC Alpha), от 4 Кбайт до 4 Мбайт (Pentium).

Решение об используемом размере страниц связано также с размером физической памяти и размером программы. Нужно также учитывать тот факт, что современные технологии программирования приводят к снижению локализации ссылок процесса. Например, объектно-ориентированные технологии стимулируют применение множества мелких модулей кода и данных с обращениями к большому количеству объектов за относительно короткое время (если программа на языке С для небольших задач занимает 3 – 4 Кбайт, то та же программа на Visual C++ займет сотни Кбайт). Многопоточные приложения приводят к внезапным изменениям в потоке команд и обращениям к памяти, разбросанным по сильно отличающимся адресам.

Новые тенденции в программировании приводят к тому, что снижается результативность поиска в TLB с ростом размеров процесса и уменьшением локализации обращений в программе. Таким образом, TLB может стать узким местом, ограничивающим производительность виртуальной памяти. Чтобы повысить производительность TLB, нужно увеличивать его емкость. Однако увеличение размера TLB связано с другими аспектами аппаратного решения вопросов обращения к памяти

– такими как размер кэша основной памяти и количество обращений к памяти при выполнении одной команды. Это приводит к выводу о невозможности роста размера TLB такими же темпами, как увеличение размеров памяти. Альтернативой может быть использование больших размеров страниц (в этом случае размер TLB может быть меньше, а TLB ссылается на больший блок данных). Однако в этом случае кэш ОП должен тоже быть большим. Кроме того, большие размеры страниц приводят к значительной внутренней фрагментации.

Учитывая все эти обстоятельства, в рядя компьютеров применяются множественные размеры страниц (что, однако, весьма сложно как в аппаратном аспекте, так и в программном в части операционной системы). Множественные размеры страниц обеспечивают гибкость, необходимую для использования TLB. Большие непрерывные области адресного пространства процесса, например программный код, могут отображаться с использованием небольшого количества страниц, в то время как стеки потоков могут использоваться для отображения страницы малого размера.

Одна из основных задач ОС – управление виртуальной памятью. При выборе стратегии решения этой задачи ключевым вопросом становится производительность: требуется сократить количество прерываний из-за отсутствия страницы в основной памяти, поскольку их обработка приводит к существенным накладным расходам. Кроме того, ОС должна активизировать готовый к работе процесс на время выполнения медленных операций ввода-вывода.

Для управления страничным обменом нужно решать следующие задачи [18, 17]:

- когда передавать страницу в основную память;
- где размещать страницу в физической памяти;
- какую страницу основной памяти выбрать для замещения, если в основной памяти нет свободной физической страницы;
- сколько страниц процесса следует загрузить в основную память;
- когда измененная страница должна быть записана во вторичную память;
- сколько процессов размещать в основной памяти.

В соответствии с этими задачами ниже перечислены стратегии ОС для управления виртуальной памятью.

Наименование	Возможные алгоритмы (решения)
Стратегия выборки (когда?)	По требованию, предварительная выборка
Стратегия размещения (где?)	Первый подходящий раздел (для сегментной виртуальной памяти). Любая страница физической памяти (для сегментно-страничной и страничной организации виртуальной памяти)
Стратегия замещения (какие?)	Оптимальный выбор, дольше всех неиспользовавшиеся. Первым вошел – первым вышел (FIFO), часовой, буферизация страниц
Управление резидентным множеством (сколько?)	Фиксированный размер, переменный размер, локальная и глобальная области видимости
Стратегия очистки (когда?)	По требованию, предварительная очистка
Управление загрузкой (сколько?)	Рабочее множество, критерий L=S и 50%

Стратегия выборки определяется, когда страница должна быть передана в основную память. Два основных варианта – по требованию и предварительная. В первом случае страница передается в основную память только тогда, когда выполняется обращение к ячейке памяти, расположенной на этой странице. Если все прочие элементы системы управления памятью работают хорошо, то происходит следующее. Когда процесс только запускается, возникает поток прерываний обращений к страницам, но далее срабатывает принцип локализации, все большее количество обращений выполняется к недавно загруженным страницам, и количество прерываний существенно снижается.

В случае предварительной выборки загружается несколько страниц,

Такая выборка использует особенности работы дисковых устройств, заключающиеся в том, что несколько последовательно расположенных страниц загружаются значительно быстрее, чем загрузка этих же страниц по одной в течение некоторого промежутка времени.

Предварительная выборка планируется программистом при разработке программы. Тем не менее, выгодность предварительной выборки не доказана.

Стратегия *размещения* определяет, где именно в физической памяти будут располагаться части процесса. Для систем, использующих сегментно-страничную или чисто страничную организации виртуальной памяти, стратегия размещения не актуальна, поскольку применение TLB и аппаратное обеспечение в памяти одинаково результативны при любых сочетаниях адресов виртуальных и физических страниц.

В многопроцессорных системах с неоднородным доступом к памяти (различные расстояния между процессорами и модулями памяти) стратегия размещения становится очень важной и требует всестороннего исследования.

Стратегия замещения определяет выбор страниц в основной памяти для замещения их загружаемыми из вторичной памяти страницами. Эта стратегия связана с решением следующих вопросов:

- какое количество страниц в основной памяти должно быть выделено каждому активному процессу;
- должны ли замещаемые страницы относиться к одному процессу или в качестве кандидатов на замещение должны рассматриваться все страницы оперативной памяти;
- какие именно страницы из рассматриваемого множества следует выбрать для замещения.

Первые два вопроса относятся к стратегии управления резидентным множеством, их рассмотрим далее. Третий вопрос напрямую связан со стратегией замещения. Все используемые стратегии замещения направлены на то, чтобы выгрузить страницу, обращений к которой в ближайшем будущем не последует. Большинство стратегий замещения

пытается определить будущее поведение программы на основе ее прошлого поведения. Независимо от стратегий управления резидентным множеством имеется ряд основных алгоритмов, применяемых для выбора замещаемой страницы:

- оптимальный алгоритм;
- алгоритм дольше всех не использующейся страницы;
- алгоритм "первым вошел - первым вышел" (FIFO);
- часовой алгоритм и др.

Оптимальный алгоритм состоит в выборе замещения той страницы, обращение к которой будет через наибольший промежуток времени по сравнению со всеми остальными страницами. Понятно, что реализовать такой алгоритм невозможно, поскольку для этого системе требуется знать все будущие события. Однако он является стандартом, с которым сравниваются все алгоритмы.

Алгоритм FIFO рассматривает физические страницы процесса как циклический буфер, с циклическим удалением страниц из него. Это один из простейших в реализации алгоритмов. Логика его работы заключается в том, что замещается страница, находящаяся в памяти дольше других. Однако далеко не всегда эта страница редко используется.

Хотя алгоритм дольше всех не использующейся страницы близок к оптимальному, он труден в реализации и приводит к значительным накладным расходам. Разработано достаточно много алгоритмов, основанных на данной стратегии, многие из них представляют собой варианты схемы, известной как часовая стратегия (clock policy).

В простейшей схеме часовой стратегии с каждой физической страницей связан один бит, который называется битом использования (рис. 6.17). Когда виртуальные страницы загружаются впервые в физическую страницу, бит использования переводится в 1. При последующих обращениях к странице, вызвавших прерывание из-за отсутствия страницы, этот бит устанавливается равным 1. При работе алгоритма замещения множество страниц, являющихся кандидатами на замещение (текущий процесс, локальная область видимости или глобальная область видимости[1]), рассматривается как циклический буфер, с

которым связан указатель.

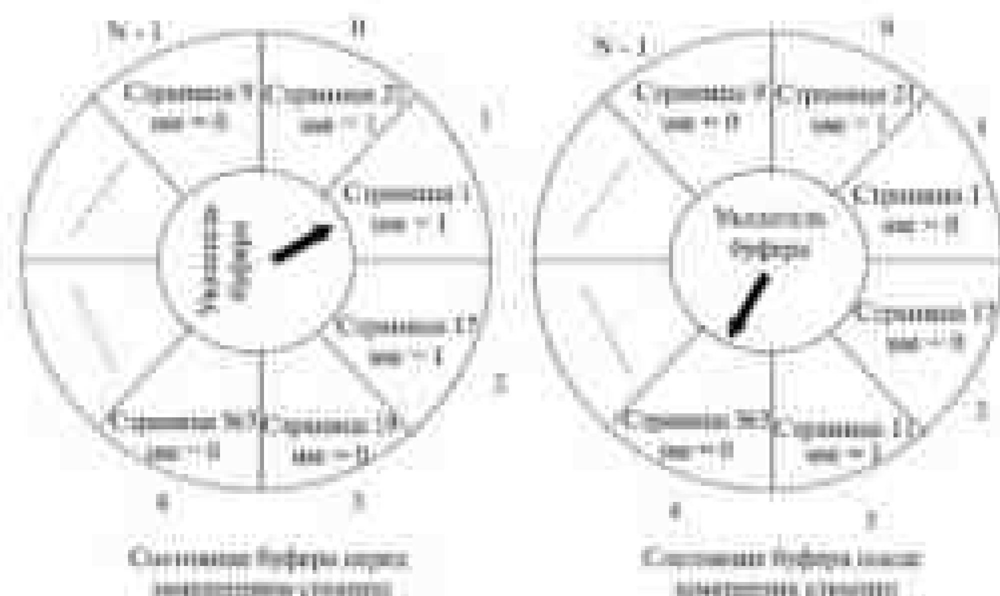


Рис. 6.17. Часовая стратегия замещения

При замещении страницы указатель перемещается к следующему кадру в буфере. Когда наступает время замещения страницы, ОС сканирует буфер для поиска кадра, бит использования которого равен 0. При этом когда в процессе поиска встречается кадр с битом использования, равным 1, он сбрасывается в 0. Первый же встреченный кадр с нулевым битом использования выбирается для замещения. Если все кадры имеют бит использования, равный 1, указатель совершает полный круг и возвращается к начальному положению, заменяя страницу в этом кадре. Буфер кадров страниц представлен в виде круга, напоминающего часы, откуда и происходит название стратегии.

На рис. 6.17 приведен простейший пример использования часовой стратегии. Для замещения доступны n-1 кадров основной памяти, представленных в виде циклического буфера. Непосредственно перед тем как заместить страницу в буфере загружаемой из вторичной памяти страницей 11, указатель буфера указывает на кадр 1, содержащий страницу 1. Приступаем к выполнению часового алгоритма. Поскольку бит использования страницы 17 в кадре 2 равен 1, эта страница не замещается. Бит ее использования сбрасывается, а указатель

перемещается к следующему кадру 3. Здесь находится страница 19, бит использования которой равен 0. Эта страница выбирается для замещения. На ее место загружается страница 13, бит использования которой переводится в 1. Указатель переводится на следующий кадр 4, и на этом выполнение алгоритма завершается. Повысить эффективность часового алгоритма можно путем увеличения количества используемых при его работе битов [17].

6.6. Сегментная организация виртуальной памяти

При страничной организации виртуальное адресное пространство делится на равные части механически без учета смыслового значения данных. Для многих задач наличие двух и более отдельных виртуальных адресных пространств может оказаться намного лучше, чем одно.

Например, у компилятора есть много таблиц, которые формируются по мере трансляции, включая в себя [10]:

1. исходный текст, сохраненный для печати листинга;
2. символьную таблицу, содержащую имена и атрибуты переменных;
3. таблицу, содержащую константы;
4. дерево грамматического разбора, содержащее синтаксический анализ программы;
5. стек, используемый для процедурных вызовов внутри компилятора.

Во время компиляции каждая из первых четырех таблиц непрерывно растет. Последняя таблица при компиляции непредсказуемо увеличивается или уменьшается. В одномерной памяти эти пять таблиц должны размещаться в смежных частях виртуального адресного пространства, как показано на рис. 6.13. В одномерном адресном пространстве при росте таблиц одна может "упереться" в другую. Можно было бы программным путем забирать память у одних таблиц и передавать другим. Но такая работа аналогична управлению собственными оверлеями, что представляет собой неудобство и большую скучную (возможно, неэкономичную) работу.

Рис. 6.18. Сложности размещения в одном виртуальном адресном пространстве

Необходим метод, освобождающий программиста от управления расширяющимися и сокращающимися таблицами тем же самым способом, которым виртуальная память устраняет беспокойство организации оверлейных программ. Простое и предельно общее решение заключается в том, чтобы обеспечить машину множеством полностью независимых адресных пространств, называемых сегментами.

Каждый сегмент содержит линейную последовательность адресов от 0 до некоторого максимума. Различные сегменты могут быть различной длины. Более того, длины сегментов могут изменяться во время выполнения. Поскольку каждый сегмент составляет отдельное адресное пространство, разные сегменты могут расти и сокращаться независимо друг от друга.

Чтобы определить адрес в такой сегментированной или двумерной памяти, программа должна указать адрес, состоящий из двух частей: номер сегмента и адрес внутри сегмента. Максимальный размер сегмента определяется разрядностью виртуального адреса, например, при 32-разрядном микропроцессоре он равен 2^{32} = 4 Гбайт. При этом максимально возможное виртуальное адресное пространство

представляет набор из N виртуальных сегментов (заметим, что общего для сегментов линейного виртуального адреса не существует).

Стоит подчеркнуть, что сегмент – это логический объект, о чем программист знает и поэтому использует его как логический объект.

Помимо простоты управления увеличивающимися или сокращающимися структурами данных, сегментированная память обладает и другими преимуществами.

К ним относятся:

- простота компоновки отдельно скомпилированных процедур (обращение к начальной точке процедуры осуществляется адресом вида (n,0), где n – номер сегмента);
- легкость обеспечения дифференцируемого доступа к различным частям программы (например, запретить обращаться для записи в сегмент программы);
- простота организации совместного использования фрагментов программ различными процессами, например, библиотеки совместного доступа могут быть оформлены в виде отдельного сегмента, который может быть включен в виртуальное адресное пространство нескольких процессов.

Сравнение страничной организации памяти и сегментации приведено ниже.

Вопрос	Страничная	Сегментация
Нужно ли программисту знать о том, что используется эта техника?	Нет	Да
Сколько в системе линейных адресных пространств?	Одно	Много
Может ли суммарное адресное		

пространство превышать размеры физической памяти?	Да	Да
Возможно ли разделение процедур и данных, а также раздельная защита для них?	Нет	Да
Легко ли размещаются таблицы с непостоянными размерами?	Нет	Да
Облегчен ли совместный доступ пользователей к процедурам?	Нет	Да
Зачем была придумана эта техника?	Чтобы получить большое линейное адресное пространство без затрат на физическую память	Для разбиения программ и данных на независимые адресные пространства, облегчения защиты и совместного доступа

При загрузке процесса в оперативную память помещается только часть его сегментов, полная копия виртуального адресного пространства находится в дисковой памяти. Для каждого загружаемого сегмента ОС подыскивает непрерывный участок свободной памяти достаточного размера. Смежные в виртуальной памяти сегменты могут занимать несмежные участки оперативной памяти. Если во время выполнения процесса происходит обращение к отсутствующему в основной памяти сегменту, происходит прерывание. Операционная система в данном случае работает аналогично подобному процессу в страничной виртуальной памяти.

На этапе создания процесса во время загрузки его образа в оперативную память ОС создает таблицу сегментов процесса, аналогичную таблице

страниц, в которой для каждого сегмента указывается:

- базовый физический адрес начала сегмента в оперативной памяти;
- размер сегмента;
- правила доступа к сегменту;
- признаки модификации, присутствия и обращения к данному сегменту, а также некоторая другая информация.

Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре. Обычно программы в этих сегментах являются реентабельными (reentrant able), т.е. обладают свойством повторной входимости кода. Код таких программ не изменяется процессом.

Механизм преобразования виртуального адреса при сегментной организации очень схож с преобразованием виртуального адреса при страничной организации. Однако факт произвольного размера сегментов приводит к тому, что нельзя обойтись конкатенацией номера сегмента и смещения. В данном случае физический адрес получается сложением базового адреса сегмента, который определяется по номеру сегмента n из таблицы сегментов, и смещения S. Схема преобразования виртуального адреса при сегментной организации памяти приведена на рис. 6.19.

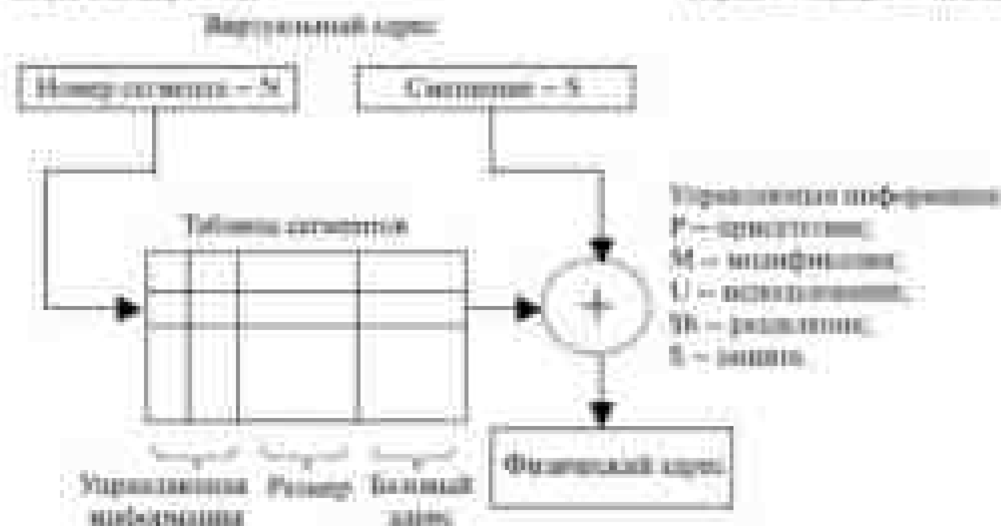


Рис. 6.19. Преобразование виртуального адреса

Использование операции сложения вместо конкатенации замедляет процедуру преобразования виртуального адреса в физический. Другим недостатком сегментной организации виртуальной памяти является большая избыточность перемещения данных между диском и оперативной памятью, поскольку перемещаются целиком большие сегменты. Во многих случаях было бы достаточно загружать и выгружать не весь сегмент, а одну или несколько страниц. Однако наиболее существенный недостаток сегментной организации виртуальной памяти – внешняя фрагментация, которая возникает из-за произвольных размеров сегментов. Заметим, что внутренняя фрагментация, характерная для страничной организации виртуальной памяти, в данном случае отсутствует.

6.7. Сегментно-страничная виртуальная память

Данный метод организации виртуальной памяти направлен на сочетание достоинств страничного и сегментного методов управления памятью. В такой комбинированной системе адресное пространство пользователя разбивается на ряд сегментов по усмотрению программиста. Каждый сегмент в свою очередь разбивается на страницы фиксированного размера, равные странице физической памяти. С точки зрения программиста, логический адрес в этом случае

состоит из номера сегмента и смещения в нем. С позиции операционной системы смещение в сегменте следует рассматривать как номер страницы определенного сегмента и смещение в ней (рис. 6.20).

Рис. 6.20. Сегментно-страничная организация памяти

С каждым процессом связана одна таблица сегментов и несколько (по одной на сегмент) таблиц страниц. При работе определенного процесса в регистре процессора хранится начальный адрес соответствующей таблицы сегментов. Получив виртуальный адрес, процессор использует его часть, представляющую номер сегмента, в качестве индекса в таблице сегментов для поиска таблицы страниц данного сегмента. После этого часть адреса, представляющая собой номер страницы, используется для поиска номера физической страницы в таблице страниц. Затем часть адреса, представляющая смещение, используется для получения искомого физического адреса путем добавления к начальному адресу физической страницы.

Сегментация удобна для реализации защиты и совместного использования сегментов разными процессами. Поскольку каждая запись таблицы сегментов включает начальный адрес и значение

длины, программа не в состоянии непреднамеренно обратиться к основной памяти за границами сегмента. Для того чтобы отличать разделяемые сегменты от индивидуальных, записи таблицы сегментов содержат 1-битовое поле, имеющее два значения: shared (разделяемый) или private (индивидуальный). Для осуществления совместного использования сегмента он помещается в виртуальное адресное пространство нескольких процессов, при этом параметры отображения этого сегмента настраиваются так, чтобы они соответствовали одной и той же области оперативной памяти (делается это указанием одного и того же базового физического адреса сегмента).

Возможен и более экономичный для ОС метод создания разделяемого виртуального сегмента – помещение его в общую часть виртуального адресного пространства, т.е. в ту часть, которая обычно задействуется для модулей ОС. В этом случае настройка соответствующей записи для разделяемого сегмента выполняется только один раз, а все процессы пользуются такой настройкой и совместно используют часть оперативной памяти.

Оба рассмотренных подхода к разделению сегмента можно иллюстрировать схемами, показанными ниже на рис. 6.21.

По второй схеме организована виртуальная память систем, работающих на процессоре Pentium. В Windows 2000 поддерживается 16 К независимых сегментов. У каждого процесса 4 Гбайт виртуального адресного пространства (из них 2 Гбайт отводится под ОС и 2 Гбайт – пользовательским программам). Основа виртуальной памяти Windows 2000 представляется двумя таблицами: локальной таблицей дескрипторов LDT (Local Descriptor Table) и глобальной таблицей дескрипторов GDT (Global Descriptor Table). У каждого процесса есть своя собственная таблица LDT, но глобальная таблица дескрипторов одна, ее совместно используют все процессы. Таблица LDT описывает сегменты, локальные для каждой программы, включая ее код, данные, стек и т.д.; таблица GDT несет информацию о системных сегментах, включая саму операционную систему.

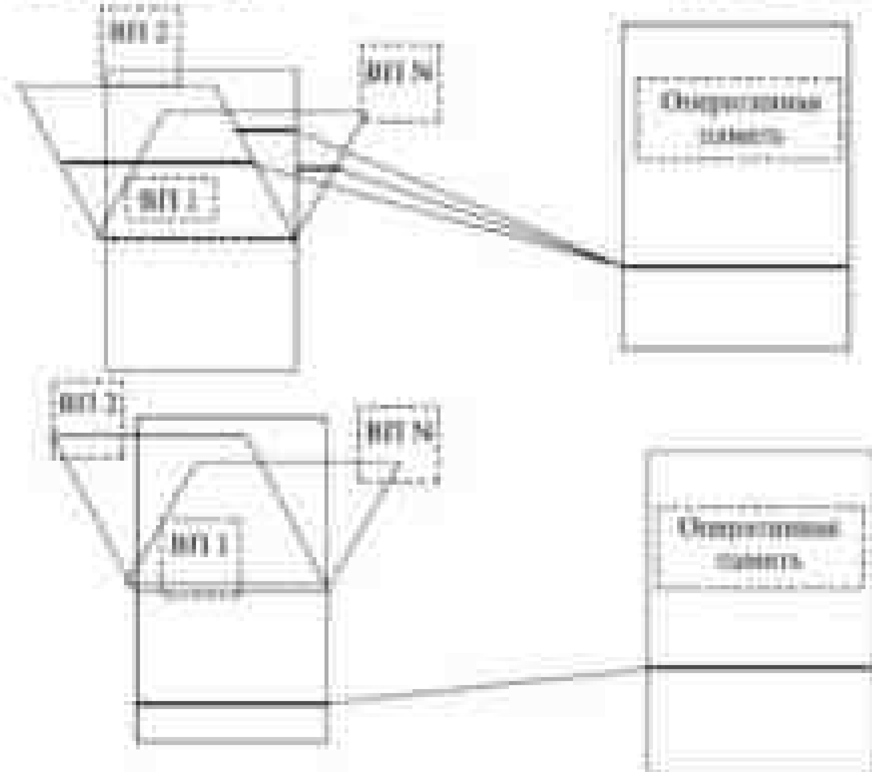


Рис. 6.21. Разделяемые сегменты

В каждый момент времени в специальных регистрах GDTR и LDTR хранится информация о местоположении и размерах глобальной таблицы GDT и активной таблицы LDT. Регистр LDTR указывает на расположение сегмента LDT в оперативной памяти косвенно – он содержит индекс дескриптора в таблице GTD, в котором содержится адрес таблицы LDT и ее размер.

Процесс обращается к физической памяти по виртуальному адресу, представляющему собой пару – селектор и смещение. Селектор определяет номер сегмента, а смещение – положение искомого адреса относительно начала сегмента. Селектор состоит из трех полей (рис. 6.22). Индекс задает пользовательский номер дескриптора в таблице GDT или LDT (всего 2^{13} = 8 K сегментов). Таким образом, виртуальное адресное пространство процесса состоит из 8 K локальных и 8 K глобальных сегментов, всего из 16 K сегментов. Учитывая, что каждый

сегмент имеет максимальный размер 4 Гбайт при чисто сегментной чистосегментной? организации виртуальной памяти (без включения страничного механизма), процесс может работать в виртуальном адресном пространстве в 64 Тбайт.

Рис. 6.22. Сегментно-страничная организация памяти в Windows

Поле из двух битов селектора задает требуемый уровень привилегий, и используется механизм защиты. В системах на базе микропроцессора Pentium поддерживается 4 уровня защиты, где уровень 0 является наиболее привилегированным, а уровень 3 – наименее привилегированным. Эти уровни образуют так называемые кольца защиты (рис. 6.23).

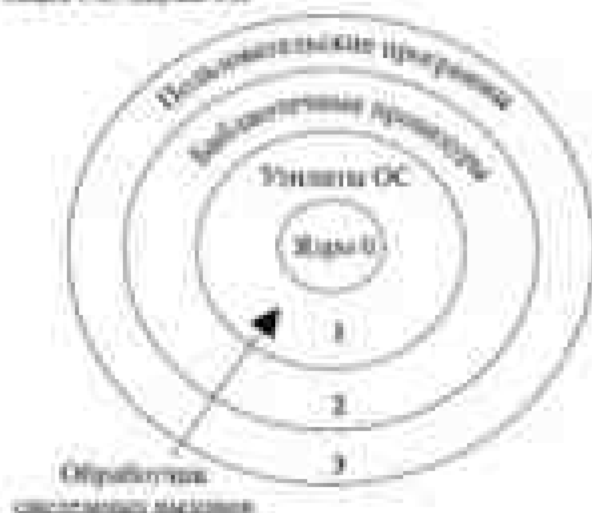


Рис. 6.23. Кольца защиты в Windows

Система защиты манипулирует несколькими переменными, характеризующими уровень привилегий:

- DPL (Descriptor Privilege Level) – задается полем DPL в дескрипторе сегмента;
- RPL (Requested Privilege Level) – запрашиваемый уровень привилегий, задается полем RPL селектора сегмента;
- CPL (Current Privilege Level) – текущий уровень привилегий выполняемого кода, задается полем RPL селектора кодового сегмента;
- EPL (Effective Privilege Level) – эффективный уровень привилегий запроса.

Под запросом понимается любое обращение к памяти. Уровни привилегий DPL и RPL назначаются операционной системой при создании новых процессов и во время их загрузки в память. Уровень привилегий определяет не только возможности доступа к сегментам и дескрипторам, но и разрешенный набор инструкций. В каждый момент времени работающая программа находится на определенном уровне, что отмечается 2-битовым полем в регистре слова состояния программы (PSW). Уровень привилегий кодового сегмента DPL определяет текущий уровень привилегий CPL, фиксируемый в PSW.

Контроль доступа процесса к сегментным данным осуществляется на основе сопоставления эффективного уровня привилегий EPL запроса и уровня привилегий DPL дескриптора сегмента данных. Доступ может быть разрешен, если:

$$EPL <= DPL,$$

где $EPL = \max(CPL, RPL)$.

Значение RPL – уровня запрашиваемых привилегий – определяется полем RPL селектора, указывающего на запрашиваемый сегмент.

1) Концепция области видимости рассматривается в подразделе "Стратегии управления резидентным множеством".

Подсистема ввода-вывода. Файловые системы

Устройства ввода-вывода. Назначение, задачи и технологии подсистемы ввода-вывода. Согласование скоростей обмена и кэширование данных. Разделение устройств и данных между процессами. Обеспечение логического интерфейса между устройствами и системой. Поддержка широкого спектра драйверов. Динамическая загрузка и выгрузка драйверов. Поддержка синхронных и асинхронных операций ввода-вывода. Многослойная (иерархическая) модель подсистемы ввода-вывода. Драйверы. Файловые системы. Основные понятия. Архитектура файловой системы. Организация файлов и доступ к ним. Каталоговые системы. Физическая организация файловой системы. Физическая организация и адресация файла. Физическая организация FAT-системы. Файловые операции. Контроль доступа к файлам.

7.1. Устройства ввода-вывода

Внешние устройства, выполняющие операции ввода-вывода, можно разделить на три группы:

- устройства, работающие с пользователем. Используются для связи пользователя с компьютером. Сюда относятся принтеры, дисплеи, клавиатура, манипуляторы (мышь, трекбол, джойстики) и т.п.;
- устройства, работающие с компьютером. Используются для связи с электронным оборудованием. К ним можно отнести дисковые устройства и устройства с магнитными лентами, датчики, контроллеры, преобразователи;
- коммуникации. Используются для связи с удаленными устройствами. К ним относятся модемы и адаптеры цифровых линий.

По другому признаку устройства ввода-вывода можно разделить на блочные и символьные [33]. Блочными являются устройства, хранящие информацию в виде блоков фиксированного размера, причем у каждого блока есть адрес и каждый блок может быть прочитан независимо от остальных блоков. Символьные устройства принимают или передают поток символов без какой-либо блочной структуры (принтеры, сетевые карты, мыши и т.д.)

Однако некоторые из устройств не попадают ни в одну из этих категорий, например, часы, мониторы и др. И все же модель блочных и символьных устройств является настолько общей, что может использоваться в качестве основы для достижения независимости от устройств некоторого программного обеспечения операционных систем, имеющего дело с вводом-выводом. Например, файловая система имеет дело с абстрактными блочными устройствами, а зависимую от устройств часть оставляет программному обеспечению нижнего уровня.

Следует также отметить существенные различия между устройствами ввода-вывода, принадлежащими к разным классам, и в рамках каждого класса. Эти различия касаются следующих характеристик:

- скорость передачи данных (различия на несколько порядков);
- применение. Каждое действие, поддерживаемое устройством, оказывает влияние на программное обеспечение и стратегии операционной системы (например, диск, используемый для хранения файлов или для страниц виртуальной памяти, требует различного программного обеспечения);
- сложность управления. Для принтера требуется относительно простой интерфейс управления, для диска – намного сложнее. Влияние этих отличий на ОС сглаживается усложнением контроллеров ввода-вывода;
- единицы передачи данных. Данные могут передаваться блоками или потоками байтов или символов;
- представления данных. Различные устройства используют разные схемы кодирования данных, включая разную кодировку символов и контроль четности;
- условия ошибки. Природа ошибок, способ сообщения о них, их последствия и возможные ответы резко отличаются при переходе от одного устройства к другому.

Такое разнообразие внешних устройств приводит, по сути, к невозможности разработки единого и согласованного подхода к проблеме ввода-вывода как с точки зрения операционной системы, так и с точки зрения пользовательских процессов.

Устройства ввода-вывода, как правило, состоят из электромеханической

и электронной части. Обычно их выполняют в форме отдельных модулей – собственно устройство и контроллер (адаптер). В ПК контроллер принимает форму платы, вставляемой в слот расширения. Плата имеет разъем, к которому подключается кабель, ведущий к самому устройству. Многие контроллеры способны управлять двумя, четырьмя и даже более идентичными устройствами. Интерфейс между контроллером и устройством является официальным стандартом (ANSI, IEEE или ISO) или фактическим стандартом, и различные компании могут выпускать отдельно контроллеры и устройства, удовлетворяющие данному интерфейсу. Так, многие компании производят диски, соответствующие интерфейсу IDE или SCSI, а наборы схем системной логики материнских плат реализуют IDE и SCSI-контроллеры.

Интерфейс между контроллером и устройством часто является интерфейсом очень низкого уровня, т.е. очень специфичным, зависящим от типа внешнего устройства. Например, видеоконтроллер считывает из памяти байты, содержащие символы, которые следует отобразить, и формирует сигналы управления лучом электронной трубки, сигналы строчной и кадровой развертки и т.п.

Каждый контроллер взаимодействует с драйвером системным программным модулем, предназначенным для управления данным устройством. Для работы с драйвером контроллер имеет несколько регистров, кроме того, он может иметь буфер данных, из которого операционная система может читать данные, а также записывать данные в него. Каждому управляющему регистру назначается номер порта ввода-вывода. Используя регистры контроллера, ОС может узнать состояние устройства (например, готово ли оно к работе), а также выдавать команды управления устройством (принять или передать данные, включиться, выключиться и т.п.).

7.2. Назначение, задачи и технологии подсистемы ввода-вывода

Обмен данными между пользователями, приложениями и периферийными устройствами компьютера выполняет специальная подсистема ОС – подсистема ввода-вывода. Собственно, для выполнения этой задачи и были разработаны первые системные

программы, выступающие прототипами операционных систем.

Основными компонентами подсистемы ввода-вывода являются драйверы, управляющие внешними устройствами, и файловая система. В работе подсистемы ввода-вывода активно участвует диспетчер прерываний. Более того, основная нагрузка диспетчера прерываний обусловлена именно подсистемой ввода-вывода, поэтому диспетчер прерываний иногда считают частью подсистемы ввода-вывода.

Файловая система – это основное хранилище информации в любом компьютере. Она активно использует остальные части подсистемы ввода-вывода. Кроме того, модель файла лежит в основе большинства механизмов доступа к периферийным устройствам.

На подсистему ввода-вывода возлагаются следующие функции [5, 17]:

- организация параллельной работы устройств ввода-вывода и процессора;
- согласование скоростей обмена и кэширование данных;
- разделение устройств и данных между процессами (выполняющимися программами);
- обеспечение удобного логического интерфейса между устройствами и остальной частью системы;
- поддержка широкого спектра драйверов с возможностью простого включения в систему нового драйвера;
- динамическая загрузка и выгрузка драйверов без дополнительных действий с операционной системой;
- поддержка нескольких различных файловых систем;
- поддержка синхронных и асинхронных операций ввода-вывода.

Эволюция ввода-вывода может быть представлена следующими этапами [17].

1. Процессор непосредственно управляет периферийным устройством.
2. Устройство управляется контроллером. Процессор использует программируемый ввод-вывод без прерываний (переход к абстракции интерфейса ввода-вывода).
3. Использование контроллера прерываний. Ввод-вывод,

управляемый прерываниями.

4. Использование модуля (канала) прямого доступа к памяти. Перемещение данных в память (из нее) без применения процессора.
5. Использование отдельного специализированного процессора ввода-вывода, управляемого центральным процессором.
6. Использование отдельного компьютера для управления устройствами ввода-вывода при минимальном вмешательстве центрального процессора.

Проследив описанный путь развития устройств ввода-вывода, можно заметить, что вмешательство процессора в функции ввода-вывода становится все менее заметным. Центральный процессор все больше освобождается от задач, связанных с вводом-выводом, что приводит к повышению общей производительности компьютерной системы.

Для персональных компьютеров операции ввода-вывода могут выполняться тремя способами.

1. С помощью программируемого ввода-вывода. В этом случае, когда процессору встречается команда, связанная с вводом-выводом, он выполняет ее, посылая соответствующие команды контроллеру ввода-вывода. Это устройство выполняет требуемое действие, а затем устанавливает соответствующие биты в регистрах состояния ввода-вывода и не посылает никаких сигналов, в том числе сигналов прерываний. Процессор периодически проверяет состояние модуля ввода-вывода с целью проверки завершения операции ввода-вывода.

   Таким образом, процессор непосредственно управляет операциями ввода-вывода, включая опознание состояния устройства, пересылку команд чтения-записи и передачу данных. Процессор посылает необходимые команды контроллеру ввода-вывода и переводит текущий процесс в состояние ожидания завершения операции ввода-вывода. Недостатки такого метода – большие потери процессорного времени, связанные с управлением вводом-выводом.

2. Ввод-вывод, управляемый прерываниями. Процессор посылает

необходимые команды контроллеру ввода-вывода и продолжает выполнять текущий процесс, если нет необходимости в ожидании выполнения операции ввода-вывода. В противном случае текущий процесс приостанавливается до получения сигнала прерывания о завершении ввода-вывода, а процессор переключается на выполнение другого процесса. Наличие прерываний процессор проверяет в конце каждого цикла выполняемых команд.

Такой ввод-вывод намного эффективнее, чем программируемый ввод-вывод, так как при этом исключается ненужное ожидание с бесполезным простоем процессора. Однако и в этом случае ввод-вывод потребляет еще значительное количество процессорного времени, потому что каждое слово, которое передается из памяти в модуль ввода-вывода (контроллер) или обратно, должно пройти через процессор.

3. Прямой доступ к памяти (direct memory access – DMA). В этом случае специальный модуль прямого доступа к памяти управляет обменом данных между основной памятью и контроллером ввода-вывода. Процессор посылает запрос на передачу блока данных модулю прямого доступа к памяти, а прерывание происходит только после передачи всего блока данных.

В настоящее время в персональных и других компьютерах используется третий способ ввода-вывода, поскольку в структуре компьютера имеется DMA-контроллер или подобное ему устройство, обслуживающее, как правило, запросы по передаче данных от нескольких устройств ввода-вывода на конкурентной основе.

DMA-контроллер имеет доступ к системной шине независимо от центрального процессора, как показано на рис. 7.1. Контроллер содержит несколько регистров, доступных центральному процессору для чтения и записи (регистр адреса памяти, счетчик байтов, управляющие регистры). Управляющие регистры задают порт ввода-вывода, который должен быть использован, направление переноса данных (чтение или запись в устройство ввода-вывода), единицу переноса (побайтно, пословно), а также число байтов, которые следует перенести за одну операцию.

Перед выполнением операции обмена ЦП программирует DMA-контроллер, устанавливая его регистры (шаг 1 на рис. 7.1). Затем ЦП дает команду дисковому контроллеру прочитать вносити данные во внутренний буфер и проверить контрольную сумму. После этого процессор продолжает свою работу. Когда данные получены и проверены контроллером диска, DMA может начинать работу.

DMA-контроллер начинает перенос данных, посылая дисковому контроллеру по шине запрос чтения (шаг 2). Адрес памяти уже находится на адресной шине, так что контроллер знает, куда пересылать следующее слово из своего буфера. Запись в память является еще одним стандартным циклом шины (шаг 3). Когда запись закончена, контроллер диска посылает сигнал подтверждения контроллеру DMA (шаг 4). Затем контроллер DMA увеличивает используемый адрес памяти и уменьшает значение счетчика байтов. После этого шаги 2, 3 и 4 повторяются, пока значение счетчика не станет равным нулю. По завершении цикла копирования контроллер DMA инициирует прерывание процессора, сообщая ему о завершении операции ввода-вывода.

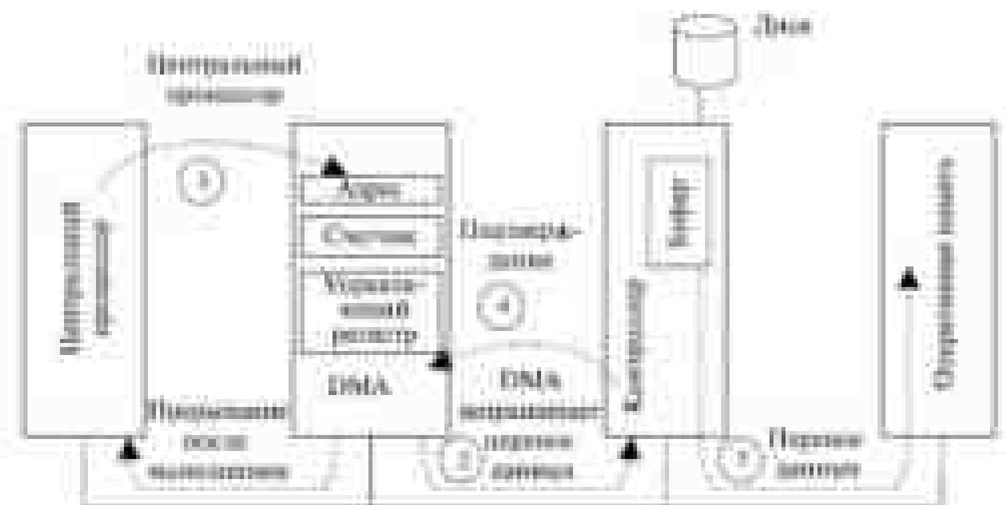


Рис. 7.1. Работа DMA

Необходимо обратить внимание на работу шины в этом процессе обмена данными. Шина может работать в двух режимах: пословном и поблочном. В первом случае контроллер DMA выставляет запрос на перенос одного слова и получает его. Если процессору также нужна эта

шины (не забывайте, в основном он работает с кэш-памятью), ему приходится подождать. Этот механизм называется захватом цикла, потому что контроллер устройства периодически забирает случайный цикл шины у центрального процессора, слегка тормозя его.

Ниже на рис. 7.2 показаны позиции цикла команд, в которых работа процессора может быть приостановлена. В любом случае приостановка процессора происходит только при необходимости использования шины. После этого устройство DMA выполняет передачу слова и возвращает управление процессору. Однако это не является прерыванием: процессор не сохраняет контекст с переходом к выполнению другого задания. Он просто делает паузу на время одного цикла шины.

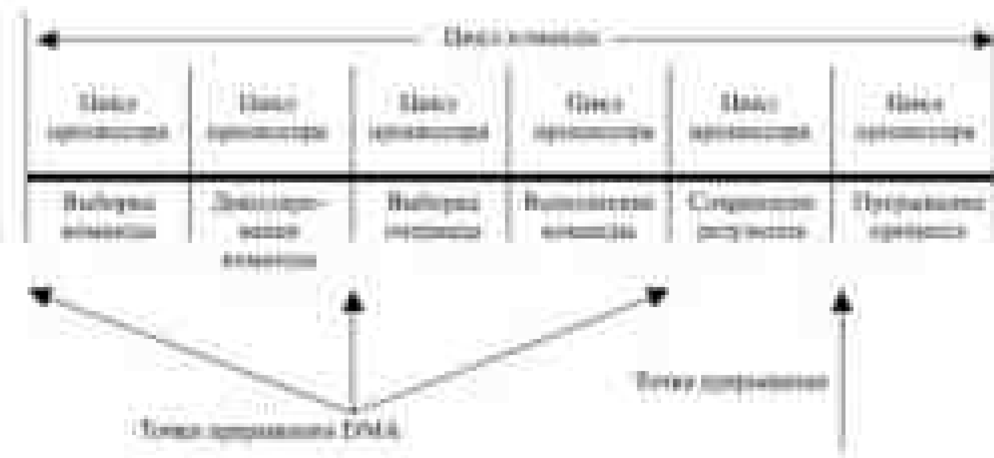


Рис. 7.2. Точки прерывания DMA

В блочном режиме работы контроллер DMA занимает шину на серию пересылок (пакет). Этот режим более эффективен, однако при переносе большого блока центральный процессор и другие устройства могут быть заблокированы на существенный промежуток времени.

При большом количестве устройств ввода-вывода от подсистемы ввода-вывода требуется спланировать в реальном масштабе времени (в котором работают внешние устройства) запуск и приостановку большого количества разных драйверов, обеспечив при этом время реакции каждого драйвера на независимые события контроллеров внешних устройств. С другой стороны, необходимо минимизировать

загрузку процессора задачами ввода-вывода.

Решение этих задач достигается на основе многоуровневой приоритетной схемы обслуживания прерываний. Для обеспечения приемлемого уровня реакции все драйверы распределяются по нескольким приоритетным уровням в соответствии с требованиями по времени реакции и временем использования процессора. Для реализации приоритетной схемы задействуется общий диспетчер прерываний ОС.

7.3. Согласование скоростей обмена и кэширования данных

При обмене данными всегда возникает задача согласования скоростей работы устройств. Решение этой задачи достигается буферизацией данных [10, 17]. В подсистеме ввода-вывода часто используется буферизация в оперативной памяти. Однако буферизация только на основе оперативной памяти часто оказывается недостаточной из-за большой разницы скоростей работы оперативной памяти и внешнего устройства обмена, оперативной памяти может просто не хватить. В этих случаях часто используют в качестве буфера дисковый файл, называемый спул-файлом. Типичный пример применения спулинга – вывод данных на принтер (для печатаемых документов объем в несколько Мбайт – не редкость, поэтому временное хранение такого файла в течение десятков минут в оперативной памяти нецелесообразно).

Другим решением проблемы является использование большой буферной памяти в контроллерах внешних устройств. Такой подход полезен в тех случаях, когда помещение данных на диск слишком замедляет обмен (или когда данные выводятся на сам диск). Например, в контроллерах графических дисплеев применяется буферная память, соизмеримая по обмену с оперативной памятью, и это существенно ускоряет вывод графики на экран.

При рассмотрении различных методов буферизации нужно учитывать, что существует, как отмечалось, два типа устройств – блочные и символьные. Первые сохраняют информацию блоками фиксированного

размеры и передают побайтно (диски, ленты). Вторые выполняют передачу в виде неструктурированных потоков байтов (терминалы, принтеры, манипулятор мыши, сканеры и др.).

Возможные схемы буферизации ввода-вывода приведены на рис. 7.3.

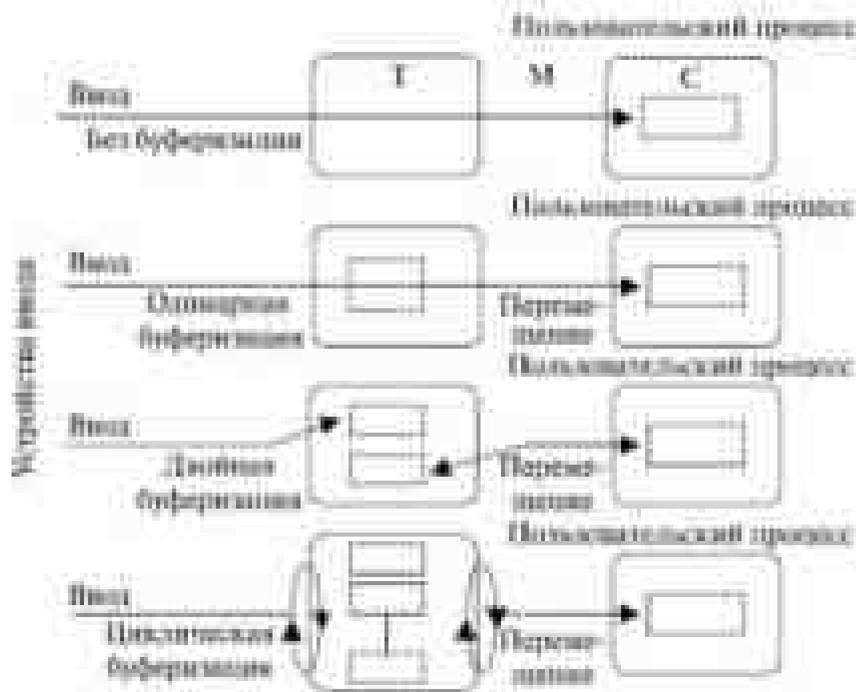


Рис. 7.3. Варианты буферизации

Простейший тип поддержки со стороны ОС – одинарный буфер. В тот момент, когда пользовательский процесс выполняет запрос ввода-вывода, операционная система назначает ему буфер в системной части оперативной памяти. Работа одинарного буфера для блочно-ориентированных устройств может быть описана следующим образом. Сначала осуществляется передача вводных данных в системный буфер. Когда она завершается, процесс перемещает блок в пользовательское пространство и немедленно производит запрос следующего блока. Такая процедура называется опережающим считыванием, или упреждающим вводом.

Подобный подход по сравнению с отсутствием буферизации

обеспечивает повышение быстродействия, поскольку пользовательский процесс может обрабатывать один блок данных в то время, когда происходит считывание следующего блока.

Пусть T – время, необходимое для ввода одного блока, а C – для вычислений, выполняющихся между запросами на ввод-вывод. Без буферизации время выполнения, приходящееся на один блок, будет равно T + C. При использовании одинарной буферизации время будет равно max (C, T) + M, где M – время перемещения данных из системного буфера в пользовательскую память. В большинстве случаев T + C > max (C, T) + M.

Схема одинарного буфера может быть применена и при потоково-ориентированном вводе-выводе – построчно или побайтно (в строчных принтерах, терминалах и др.). Например, при операции вывода пользовательский процесс может разместить в буфере строку и продолжить работу. Улучшить схему одинарной буферизации можно путем использования двух буферов. Теперь процесс выполняет передачу данных в один буфер (или считывает из него), в то время как ОС освобождает (или заполняет) другой. Эта технология известна как двойная буферизация, или сменный буфер.

Время выполнения при блочно-ориентированной передаче можно грубо оценить как max (C, T). Таким образом, если C <= T, то блочно-ориентированное устройство может работать с максимальной скоростью. Если C > T, то двойная буферизация избавляет процесс от необходимости ожидания завершения ввода-вывода.

Двойной буферизации может оказаться недостаточно, если процесс часто выполняет ввод или вывод. Решить проблему помогает наращивание количества буферов. Если буферов больше двух, схема именуется циклической буферизацией.

Буферизация данных позволяет не только согласовать скорости работы процессора и внешних устройств, но и решить другую задачу – сократить количество реальных операций ввода-вывода за счет кэширования данных. Дисковый кэш является непременным атрибутом подсистем ввода-вывода практически всех операционных систем и значительно сокращает время доступа к хранимым данным.

7.4. Разделение устройств и данных между процессами

Устройства ввода-вывода могут предоставляться процессам как в монопольном, так и разделяемом режиме. При этом ОС должна обеспечивать контроль доступа теми же способами, что и при доступе процессов к другим ресурсам вычислительной системы, – путем проверки прав пользователя или группы пользователей, от имени которых действует процесс, на выполнение той или иной операции над устройством.

ОС может контролировать доступ не только к устройству в целом, но и к отдельным порциям данных, хранимых этим устройством. Диск является типичным примером такого устройства, где важно контролировать доступ к файлам и каталогам. В последнем случае непременным является задание режима совместного использования устройства в целом.

Одно и то же устройство в разные периоды времени может работать как в разделяемом, так и в монопольном режимах. Тем не менее, существуют устройства, для которых характерен один из этих режимов, например, последовательные порты и алфавитно-цифровые терминалы чаще используются в монопольном режиме, а диск – в режиме совместного доступа.

В случае совместного использования ОС должна оптимизировать последовательность операций ввода-вывода для различных процессов в целях повышения общей производительности. Например, при обмене данными нескольких процессов с диском можно так упорядочить последовательность операций, что непроизводительные затраты времени на перемещение головок существенно уменьшатся (при этом для отдельных процессов возможно некоторое замедление операции ввода-вывода).

При разделении устройства между процессами может возникнуть необходимость в разграничении данных процессов друг от друга. Обычно такая потребность появляется при совместном использовании последовательных устройств, которые, в отличие от устройств прямого доступа, не адресуются. Типичный представитель таких устройств – принтер. Для таких устройств организуется очередь заданий на вывод,

при этом каждое задание представляет собой порцию данных, которую нельзя разрывать, например, документ для печати.

Для хранения очереди заданий используется спул-файл, который согласует скорость работы принтера и оперативной памяти и позволяет организовать разбиение данных на логические порции. Процессы могут одновременно выполнять вывод на принтер, помещая данные в свой раздел спул-файла.

7.5. Обеспечение логического интерфейса между устройствами и системой

Разнообразие устройств ввода-вывода делает актуальной функции операционной системы по созданию экранирующего логического интерфейса между периферийными устройствами и приложениями.

Практически все современные ОС поддерживают в качестве такого интерфейса файловую модель периферийных устройств, когда любое устройство выглядит для прикладного программиста последовательным набором байт, с которым можно работать с помощью унифицированных системных вызовов (например, read, write), задавая имя файла-устройства и смещение от начала последовательности байт.

Привлекательность модели файла-устройства состоит в ее простоте и унифицированности для устройств любого типа, однако во многих случаях для программирования операций ввода-вывода некоторого устройства она является слишком бедной. Поэтому данная модель часто используется в качестве базиса, над которым подсистема ввода-вывода строит более содержательную модель устройства конкретного типа.

7.6. Поддержка широкого спектра драйверов

Разнообразный набор драйверов для широкого круга популярных периферийных устройств – непременное условие популярности ОС у пользователей.

Для разработки драйверов производителями внешних устройств необходимо наличие четкого, удобного, открытого и хорошо

документированного интерфейса между драйверами и другими компонентами ОС. Драйвер взаимодействует, с одной стороны, с модулями ядра ОС (модулями подсистемы ввода-вывода, модулями системных вызовов, модулями подсистем управления процессами и памятью), а с другой стороны – с контроллерами внешних устройств. Поэтому существует два вида интерфейсов: интерфейс "драйвер-ядро" (Driver Kernel Interface, DKI) и интерфейс "драйвер-устройство" (Driver Device Interface).

Интерфейс "драйвер-ядро" должен быть стандартизован в любом случае. Подсистема ввода-вывода может поддерживать несколько различных интерфейсов DKI/DDI, предоставляя специфический интерфейс для устройств определенного класса. К наиболее общим классам относятся блочные устройства, например, диски, и символьные устройства, такие как клавиатура и принтеры. Может существовать класс сетевых адаптеров и др. В большинстве современных ОС определен стандартный интерфейс, который должны поддерживать все блочные драйверы, и второй стандартный интерфейс, поддерживаемый всеми символьными адаптерами. Эти интерфейсы включают наборы процедур, которые могут вызываться остальной операционной системой для обращения к драйверу. К этим процедурам относятся, например, процедуры чтения блока или записи символьной строки.

Кроме того, подсистема ввода-вывода поддерживает большое количество системных функций, которые драйвер может вызывать для выполнения некоторых типовых действий. Например, это операции обмена с регистрами контроллера, ведения буферов промежуточного хранения данных ввода-вывода, взаимодействия с DMA-контроллером и контроллером прерываний и др.

У драйверов устройств есть множество функций, перечисленных ниже [17].

1. Обработка запросов записи-чтения от программного обеспечения управления устройствами. Постановка запросов в очередь.
2. Проверка входных параметров запросов и обработка ошибок.
3. Инициализация устройства и проверка статуса устройства.
4. Управление энергопотреблением устройства.

5. Регистрация событий в устройстве.
6. Выдача команд устройству и ожидание их выполнения, возможно, в блокированном состоянии, до поступления прерывания от устройства.
7. Проверка правильности завершения операции.
8. Передача запрошенных данных и статуса завершенной операции.
9. Обработка нового запроса при незавершенном предыдущем запросе (для реентерабельных драйверов).

Наиболее очевидная функция состоит в обработке абстрактных запросов чтения и записи независимого от устройств программного обеспечения, расположенного над ними. Но, кроме этого, они должны выполнять еще несколько функций. Например, драйвер должен при необходимости инициализировать устройство. Ему может понадобиться управлять энергопотреблением устройства и регистрацией событий.

Многие драйверы обладают сходной общей структурой. Типичный драйвер начинает работу с проверки входных параметров. Если они не удовлетворяют определенным критериям, драйвер возвращает ошибку. В противном случае драйвер преобразует абстрактные термины в конкретные. Например, дисковый драйвер преобразует линейный номер кластера в номер головки, дорожки и сектора.

Затем драйвер может проверить, не используется ли это устройство в данный момент. Если устройство занято, запрос может быть поставлен в очередь. Если устройство свободно, проверяется статус устройства, чтобы понять, может ли запрос быть обслужен прямо сейчас. Может оказаться необходимым включить устройство или запустить двигатель, прежде чем начнется перенос данных. Как только устройство включено и готово, начинается собственно управление устройством.

Управление устройством подразумевает выдачу ему серии команд. Именно в драйвере определяется последовательность команд в зависимости от того, что должно быть сделано. Определившись с командой, драйвер начинает записывать их в регистры контроллера устройства. После записи каждой команды в контроллер, возможно, будет нужно проверить, принял ли контроллер команду и готов ли принять следующую. Такая последовательность действий продолжается до тех пор, пока контроллеру не будут переданы все команды.

Некоторые контроллеры способны принимать связные списки команд, находящихся в памяти. Они сами считывают и выполняют их без дальнейшей помощи ОС.

После того как драйвер передал все команды контроллеру, ситуация может развиваться по двум сценариям. Во многих случаях драйвер устройства должен ждать, пока контроллер не выполнит для него определенную работу, поэтому он блокируется до тех пор, пока прерывание от устройства не разблокирует его. В других случаях операция завершается без задержек и драйверу не нужно блокироваться. В любом случае по завершении выполнения операции драйвер должен проверить, завершилась ли операция без ошибок. Если все в порядке, драйверу, возможно, придется передать данные (например, только что прочитанный блок) независимому от устройств программному обеспечению. Наконец, драйвер возвращает информацию о состоянии для информирования вызывающей программы о статусе завершения операции. Если в очереди находились другие запросы, один из них теперь может быть выбран и запущен. В противном случае драйвер блокируется в ожидании следующего запроса.

Для поддержки процесса разработки драйверов операционной системы выпускается так называемый пакет DDK (Driver Development Kit), представляющий собой набор инструментальных средств-библиотек, компиляторов и отладчиков.

7.7. Динамическая загрузка и выгрузка драйверов

Так как набор потенциально поддерживаемых данный ОС периферийных устройств всегда шире набора устройств, которыми ОС должна управлять при установке на конкретной машине, то ценным свойством ОС является возможность динамически загружать в оперативную память требуемый драйвер (без остановки ОС) и выгружать его, если надобность в драйвере отпала. Такое свойство ОС может существенно сэкономить системную область памяти.

Альтернативой динамической загрузке драйверов при изменении текущей конфигурации внешних устройств компьютера является повторная компиляция кода ядра с требуемым набором драйверов, что создает между всеми компонентами ядра статические связи вместо

динамически. Например, таким образом решалась данная проблема в ранних версиях ОС UNIX. При статических связях между ядром и драйверами структура ОС упрощается, но этот подход требует наличия исходных кодов модулей ОС, доступность которых скорее является исключением (для некоммерческих версий UNIX). Кроме того, в этом варианте работающую версию ОС надо остановить и заменить новой, что не всегда допустимо в некоторых применениях.

Поэтому поддержка динамической загрузки драйверов является практически обязательным требованием для современных универсальных ОС.

7.8. Поддержка синхронных и асинхронных операций ввода-вывода

Операция ввода-вывода может выполняться по отношению к программному модулю, запросившему операцию, в синхронном или асинхронном режимах [10]. Синхронный режим означает, что программный модуль приостанавливает свою работу до тех пор, пока операция ввода-вывода не будет завершена (рис. 7.4, верхняя диаграмма). При асинхронном режиме программный модуль продолжает выполняться в мультипрограммном режиме одновременно с операцией ввода-вывода (рис. 7.4, нижняя диаграмма).

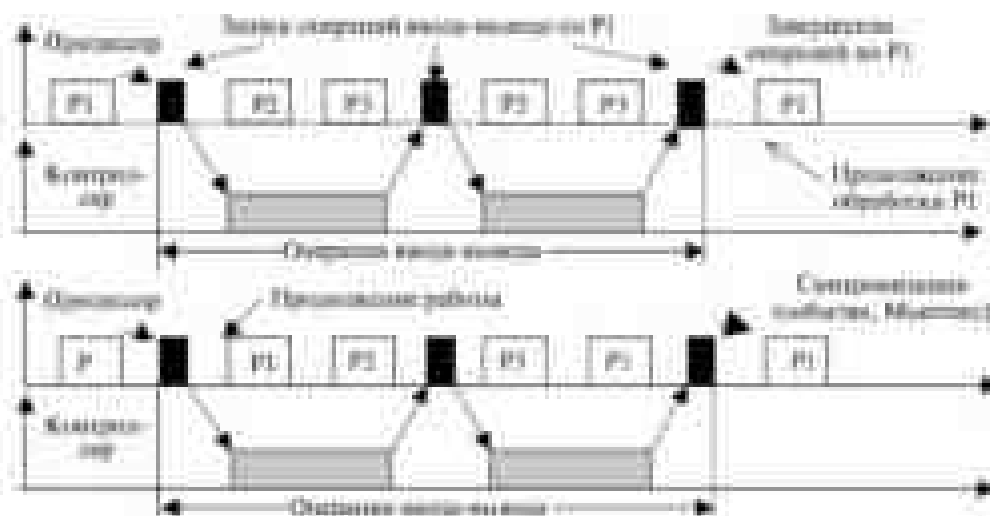


Рис. 7.4. Варианты выполнения операций ввода-вывода

Отличие заключается в том, что операция ввода-вывода может быть инициирована не только пользовательским процессом – в этом случае операция выполняется в рамках системного вызова, – но и кодом ядра, например, кодом подсистемы виртуальной памяти для считывания отсутствующей страницы.

Системные вызовы ввода-вывода чаще оформляются как синхронные процедуры в связи с тем, что такие операции длятся долго и пользовательскому процессу или потоку все равно придется ждать получения результатов операции, для того чтобы продолжить свою работу.

Внутренние вызовы операций ввода-вывода из модулей ядра обычно выполняются в виде асинхронных процедур, так как кодам ядра нужна свобода в выборе дальнейшего поведения после запроса ввода-вывода.

7.9. Многослойная (иерархическая) модель подсистемы ввода-вывода

При большом разнообразии устройств ввода-вывода, обладающих существенно различными характеристиками, иерархическая структура подсистемы ввода-вывода позволяет соблюсти баланс между двумя противоречивыми требованиями. С одной стороны, необходимо учесть все особенности каждого устройства, а с другой стороны – обеспечить единое логическое представление и унифицированный интерфейс для устройств всех типов. При этом нижние слои подсистемы ввода-вывода должны включать индивидуальные драйверы, написанные для конкретных физических устройств, а верхние слои должны обобщать процедуры управления этими устройствами, предоставляя общий интерфейс если не для всех устройств, то, по крайней мере, для группы устройств, обладающих некоторыми общими характеристиками, например, для принтеров определенного производителя или для всех матричных принтеров и т.п.

Многослойность структуры, безусловно, облегчает решение большинства перечисленных в предыдущем разделе задач подсистемы ввода-вывода. Обобщенная структура подсистемы ввода-вывода

показана на рис. 7.5 [13]. Как видно из рисунка, программное обеспечение подсистемы ввода-вывода делится не только на горизонтальные слои, но и на вертикальные. В данном случае в качестве примера приведены три вертикальные подсистемы управления дисками, графическими устройствами и сетевыми адаптерами. Естественно, таких подсистем может быть больше. Например, сюда можно добавить подсистему управления текстовыми терминалами или подсистему управления специализированными устройствами, такими как аналого-цифровые и цифро-аналоговые преобразователи.

В каждой вертикальной подсистеме – несколько слоев модулей. Нижний слой образуют аппаратные драйверы, управляющие аппаратурой внешних устройств, осуществляя обмен байтами и блоками байтов. Как правило, этот слой программного обеспечения не имеет дела с вопросами логической организации данных, например, с файлами или сложными графическими объектами. Функции вышележащих слоев в значительной степени зависят от типа вертикальной подсистемы.

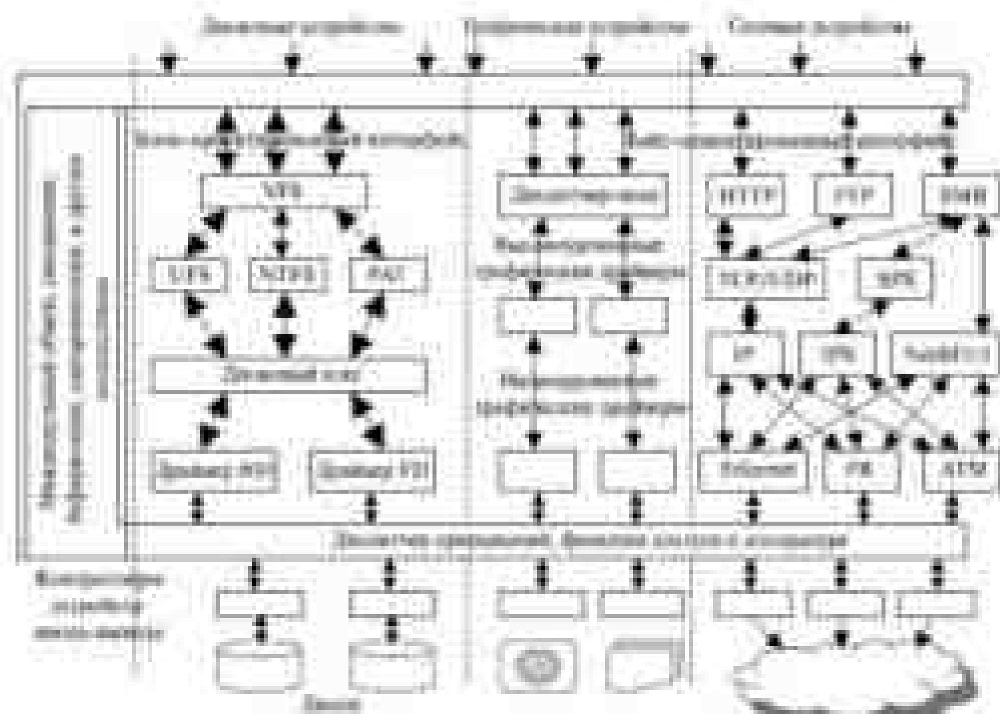


Рис. 7.5. Иерархическая структура подсистемы ввода-вывода

Наряду с модулями, отражающими специфику внешних устройств, в подсистеме ввода-вывода имеются модули универсального назначения. Эти модули организуют согласованную работу всех остальных компонентов подсистемы ввода-вывода, взаимодействие с пользовательскими процессами и другими подсистемами ОС. Эти организующие функции распределяются по всем уровням, образуя оболочку называемую менеджером ввода-вывода.

Верхний слой менеджера составляют системные вызовы ввода-вывода, которые принимают от пользовательских процессов запросы на ввод-вывод и переадресуют их отвечающим за определенный класс устройств модулям и драйверам, а также возвращают процессам результаты операций ввода-вывода. Таким образом, этот слой поддерживает пользовательский интерфейс ввода-вывода, создавая для прикладных программистов максимум удобств по манипулированию внешними устройствами и расположенными на них данными.

Нижний слой менеджера реализует непосредственное взаимодействие с контроллерами внешних устройств, экранируя драйверы от особенностей аппаратной платформы компьютера – шин ввода-вывода, системы прерываний и т.п. Этот слой принимает от драйверов запросы на обмен данными с регистрами контроллеров в некоторой обобщенной форме с использованием независимых от шины ввода-вывода адресации и формата, а затем преобразует эти запросы в зависящий от аппаратной платформы формат.

Диспетчер прерываний может входить в состав менеджера ввода-вывода или представлять отдельный модуль ядра. В последнем случае менеджер ввода-вывода выполняет для диспетчера прерываний первичную обработку запросов прерываний, передавая диспетчеру обобщенные сведения об источнике запроса.

Важной функцией менеджера ввода-вывода является создание некоторой среды для остальных компонентов системы, которая бы облегчала их взаимодействие друг с другом. Эта задача решается созданием стандартного внутреннего интерфейса взаимодействия модулей ввода-вывода между собой. Это облегчает включение новых драйверов и файловых систем в состав ОС. Кроме того, разработчики драйверов и других программных компонентов освобождаются от

написания общих процедур, таких как буферизация данных и синхронизация нескольких модулей между собой при обмене данными. Все эти функции берет на себя менеджер ввода-вывода.

Еще одной функцией менеджера ввода-вывода является организация взаимодействия модулей ввода-вывода с модулями других подсистем ОС, таких как подсистема управления процессами, виртуальной памятью и другими.

7.10. Драйверы

Первоначально термин "драйвер" применялся в достаточно узком смысле – под драйвером понимается программный модуль, который:

- входит в состав ядра ОС, работая в привилегированном режиме;
- непосредственно управляет внешним устройством, взаимодействуя с его контроллером с помощью команд ввода-вывода компьютера;
- обрабатывает прерывания от контроллера устройства;
- предоставляет прикладному программисту удобный логический интерфейс работы с устройством, экранируя от него низкоуровневые детали управления устройством и организации его данных;
- взаимодействует с другими модулями ядра ОС с помощью строго оговоренного интерфейса, описывающего формат передаваемых данных, структуру буферов, способы включения драйвера в состав ОС, способы вызова драйвера, набор общих процедур подсистемы ввода-вывода, которыми драйвер может пользоваться и т.п.

Согласно этому определению драйвер вместе с контроллером устройства и прикладной программой воплощали идею многослойного подхода к организации программного обеспечения. Контроллер представляет нижний слой управления устройством, выполняющий операции в терминах блоков и агрегатов устройства (например, передвижение головки дисковода, побитную передачу байта по двухпроводному кабелю). Драйвер выполнял более сложные операции, преобразуя данные, адресуемые в терминах номеров цилиндров,

головок и секторов диска, в линейную последовательность блоков. В результате прикладная программа работала с данными, преобразованными в достаточно понятную форму – файлами, таблицами баз данных, текстовыми окнами на мониторе и т.п., не вдаваясь в детали представления этих данных в устройствах ввода-вывода.

В описанной схеме драйверы не делились на слои. Постепенно, по мере развития операционных систем и усложнения структуры подсистемы ввода-вывода, наряду с традиционными драйверами в ОС появились так называемые высокоуровневые драйверы, которые располагаются в общей модели подсистемы ввода-вывода над традиционными драйверами. Появление таких драйверов можно считать развитием идеи многоуровневой организации подсистемы ввода-вывода, когда ее функции декомпозируются между несколькими модулями в соседних слоях иерархии (таких примеров много, например семиуровневая модель сетевых протоколов).

Традиционные драйверы, которые стали называть аппаратными, низкоуровневыми или драйверами устройств, освобождаются от высокоуровневых функций и занимаются только низкоуровневыми операциями. Эти низкоуровневые операции составляют фундамент, на котором можно построить тот или иной набор операций в драйверах более высоких уровней.

При таком подходе повышается гибкость и расширяемость функций по управлению устройством. Например, если различным приложениям необходимо работать с различными логическими моделями одного и того же физического устройства, то для этого в системе достаточно установить несколько драйверов на одном уровне, работающих над одним аппаратным драйвером. Несколько драйверов, управляющих одним устройством, но на разных уровнях, можно рассматривать как один многоуровневый драйвер.

На практике используют от двух до пяти уровней драйверов, поскольку с увеличением числа уровней снижается скорость выполнения операций ввода-вывода.

Высокоуровневые драйверы оформляются по тем же правилам и придерживаются тех же внутренних интерфейсов, что и аппаратные

драйверы. Как правило, высокоуровневые драйверы не вызываются по прерываниям, так как взаимодействуют с устройством через посредничество аппаратных драйверов.

В модулях подсистемы ввода-вывода, кроме драйверов, могут присутствовать и другие модули, например, дисковый кэш. Достаточно специфичные функции кэша делают нецелесообразным оформление его в виде драйвера, взаимодействующего с другими модулями ОС только с помощью услуг менеджера ввода-вывода. Другим примером модуля, который чаще всего не оформляется в виде драйвера, является диспетчер окон графического интерфейса. Иногда этот модуль вообще выносится из ядра ОС и реализуется в виде пользовательского интерфейса. Таким образом, был реализован диспетчер окон в Windows NT 3.5 и 3.51, но этот микроядерный подход заметно замедляет графические операции, поэтому в Windows 4.0 диспетчер окон и высокоуровневые графические драйверы, а также графическая библиотека GDI были перенесены в пространство ядра.

Аппаратные драйверы после запуска операции ввода-вывода должны своевременно реагировать на завершение контроллером заданного действия путем взаимодействия с системой прерываний. Драйверы более высоких уровней вызываются не по прерываниям, а по инициативе аппаратных драйверов или драйверов вышележащего уровня. Не все процедуры аппаратного драйвера нужно вызывать по прерываниям, поэтому драйвер обычно имеет определенную структуру, в которой выделяется секция обработки прерываний (Interrupt Service Routine, ISR), которая и вызывается от соответствующего устройства диспетчером прерываний.

В унификацию драйверов большой вклад внесла ОС UNIX, в которой все драйверы были разделены на два класса: блок-ориентированные (Block-oriented) и байт-ориентированные (Character-oriented) драйверы. Это более общее деление, чем деление на вертикальные подсистемы. Например, драйверы графических устройств и сетевых устройств относятся к классу байт-ориентированных.

Блок-ориентированные драйверы управляют устройствами прямого доступа, которые хранят информацию в блоках фиксированного размера, каждый из которых имеет свой адрес. Адресуемость блоков

приводит к тому, что для дисков, являющихся устройствами прямого доступа, появляется возможность кэширования данных в оперативной памяти. Это обстоятельство значительно влияет на общую организацию ввода-вывода для блок-ориентированных драйверов.

Устройства, с которыми работают байт-ориентированные драйверы, не адресуют данные и не позволяют производить операции поиска данных, они генерируют или потребляют последовательность байтов (терминалы, принтеры, сетевые адаптеры и т.п.).

Однако не все устройства, управляемые подсистемой ввода-вывода, можно разделить на блок- и байт-ориентированные. Для таких устройств (например, таймер) нужен специфический драйвер.

В свое время ОС UNIX сделала очень важный шаг по унификации операций и структуризации программного обеспечения ввода-вывода. В ОС UNIX все устройства рассматриваются как виртуальные (специальные) файлы, что дает возможность использовать общий набор базовых операций ввода-вывода для любых устройств независимо от их специфики. Подобная идея реализована позже в MS-DOS, где последовательные устройства – монитор, принтер и клавиатура – считаются файлами со специальными именами: con, prn, com.

7.11. Файловые системы. Основные понятия

Цели и задачи файловой системы

Любое компьютерное приложение получает, хранит и выводит данные. Во время работы процесс может хранить ограниченное количество данных в собственном адресном пространстве, поскольку его емкость ограничена рамками виртуального адресного пространства. Для некоторых приложений, например, систем резервирования авиабилетов, систем банковского учета и др., однако, только виртуального адресного пространства будет недостаточно.

Кроме того, после завершения работы процесса информация, хранящаяся в его адресном пространстве, теряется. В это же время для ряда приложений (например, баз данных) ее надо хранить длительное время, а иногда даже вечно. Исчезновение данных после завершения

процесса для таких приложений неприемлемо. Информация должна сохраняться и при аварийном завершении процесса в случае сбоя компьютера.

Третья проблема состоит в том, что часто необходимо разным процессам одновременно получать доступ к одним и тем же данным (или части данных). Для решения этой проблемы необходимо отделить информацию от процесса.

Таким образом, необходимо хранить данные на устройствах компьютеров (диски, ленты и др.) с соблюдением следующих требований [13].

1. Устройства должны позволять хранить очень большие объемы данных. К таким устройствам относятся жесткие магнитные диски, магнитные ленты, оптические и магнитооптические диски.
2. Информация должна длительно и надежно сохраняться после прекращения работы процесса, использующего эту информацию. Долговременность хранения обеспечивается применением запоминающих устройств, не зависящих от электропитания, а высокая надежность определяется соответствующей организацией операционной системы.
3. Несколько процессов должны иметь возможность получения одновременного доступа к информации, т.е. должно быть обеспечено совместное использование данных.

Решение этих проблем состоит в хранении информации, организованной в файлы. Файл — это именованная совокупность данных, хранящаяся на каком-либо носителе информации.

При рассмотрении отдельных файлов и их совокупностей используются следующие понятия.

1. Поле (Field) — основной элемент данных. Поле содержит единственное значение, такое как имя служащего, дата, значение некоторого показателя и т.п. Поле характеризуется длиной и типом данных и может быть фиксированной или переменной длины, т.е. состоять из нескольких подполей: имя поля, значение, длина поля.

2. Запись (Record) – набор связанных между собой полей, которые могут быть обработаны как единое целое некоторой прикладной программой (например, запись о сотруднике, содержащая такие поля, как имя, должность, оклад и т.д.). В зависимости от структуры записи могут быть фиксированной или переменной длины.
3. Файл (File) – совокупность однородных записей. Файл рассматривается как единое целое приложениями и пользователями. Обращение к файлу осуществляется по его имени. Пользователь (программист) должен иметь удобные средства работы с файлами, включая каталоги-справочники, объединяющие файлы в группы, средства поиска файла по различным признакам, набор команд для создания, модификации и удаления файлов. Файл может быть создан одним пользователем, а затем использоваться другим, при этом создатель файла или администратор могут определять права доступа к нему других пользователей. В некоторых системах управления доступом осуществляется на уровне записи, а иногда и на уровне поля.
4. База данных (database) – набор связанных между собой данных, представленных совокупностью файлов одного или нескольких типов. Обычно существует отдельная система управления базой данных (СУБД), независимая от операционной системы, но, тем не менее, она почти всегда использует некоторые программы управления файлами.

Обычно единственным способом работы с файлами является применение системы управления файлами или иначе – файловой системы (ФС).

Файловая система – это часть операционной системы, включающая:

- совокупность всех файлов на носителе информации (магнитном или оптическом диске, магнитной ленте и др.);
- наборы структур данных, используемых для управления файлами, каталоги и дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске и др.);
- комплекс системных программных средств, реализующих различные операции над файлами (создание, уничтожение,

чтение, запись и др.).

Задачи, решаемые файловой системой, во многом определяются способом организации вычислительного процесса (наиболее простые – в однопрограммных и однопользовательских ОС, наиболее сложные – в сетевых ОС.).

В мультипрограммных, многопользовательских ОС задачами файловой системы являются [10]:

- соответствие требованиям управления данными и требованиям со стороны пользователей, включающими возможность хранения данных и выполнения операций с ними;
- гарантирование корректности данных, содержащихся в файле;
- оптимизация производительности, как с точки зрения системы (пропускная способность), так и с точки зрения пользователя (время отклика);
- поддержка ввода-вывода для различных типов устройств хранения информации;
- минимизация или полное исключение возможных потерь или повреждений данных;
- защита файлов от несанкционированного доступа;
- обеспечение поддержки совместного использования файлов несколькими пользователями (в том числе средства блокировки файла и его частей, исключение тупиков, согласование копий и т.п.);
- обеспечение стандартизированного набора подпрограмм интерфейса ввода-вывода.

Минимальным набором требований к файловой системе со стороны пользователя диалоговой системы общего назначения можно считать следующую совокупность возможностей, предоставляемых пользователю:

1. создание, удаление, чтение и изменение файлов;
2. контролируемый доступ к файлам других пользователей;
3. управление доступом к своим файлам;
4. реструктурирование файлов в соответствии с решаемой задачей;
5. перемещение данных между файлами;

6. резервирование и восстановление файлов в случае повреждения;
7. доступ к файлам по символьным именам.

7.12. Архитектура файловой системы

Файловая система позволяет программам обходиться набором достаточно простых операций для выполнения действий над некоторым абстрактным объектом, представляющим файл. При этом программистам не нужно иметь дело с деталями действительного расположения данных на диске, буферизацией данных и другими низкоуровневыми проблемами передачи данных с запоминающего устройства. Все эти функции файловая система берет на себя. Файловая система распределяет дисковую память, поддерживает именование файлов, отображает имена файлов в соответствующие адреса во внешней памяти, обеспечивает доступ к данным, поддерживает разделение, защиту и восстановление данных.

Таким образом, файловая система играет роль промежуточного слоя, экранирующего все сложности физической организации долговременного хранилища данных и создающего для программ более простую логическую модель этого хранилища, а также предоставляет им набор удобных в использовании команд для манипулирования файлами.

Классическая схема организации программного обеспечения файловой системы представлена на рис. 7.6.

Рис. 7.6. Организация программного обеспечения файловой системы

На нижнем уровне драйверы устройств непосредственно связаны с периферийными устройствами или их контроллерами либо каналами. Драйвер устройства отвечает за начальные операции ввода-вывода устройства и за обработку завершения запроса ввода-вывода. При файловых операциях контролируемыми устройствами являются дисководы и стримеры (накопители на МЛ). Драйверы устройств рассматриваются как часть операционной системы.

Следующий уровень называется базовой файловой системой, или уровнем физического ввода-вывода. Это первичный интерфейс с окружением (периферией) компьютерной системы. Он оперирует блоками данных, которыми обменивается с дисками, магнитной лентой и другими устройствами. Поэтому он связан с размещением и буферизацией блоков в оперативной памяти. На этом уровне не выполняется работа с содержимым блоков данных или структурой файлов. Базовая файловая система обычно рассматривается как часть операционной системы (в MS-DOS эти функции выполняет BIOS, не относящийся к ОС).

Диспетчер базового ввода-вывода отвечает за начало и завершение файлового ввода-вывода. На этом уровне поддерживаются управляющие структуры, связанные с устройством ввода-вывода,

планированием и статусом файлов. Диспетчер осуществляет выбор устройства, на котором будет выполняться операция файлового ввода-вывода, планирование обращения к устройству (дискам, лентам), назначение буферов ввода-вывода и распределение внешней памяти. Диспетчер базового ввода-вывода является частью ОС.

Логический ввод-вывод предоставляет приложениям и пользователям доступ к записям. Он обеспечивает возможности общего назначения по вводу-выводу записей и поддерживает информацию о файлах. Наиболее близкий к пользователю уровень ФС часто называется методом доступа. Он обеспечивает стандартный интерфейс между приложениями и файловыми системами и устройствами, содержащими данные. Различные методы доступа отражают различные структуры файлов и различные пути доступа и обработки данных.

7.13. Организация файлов и доступ к ним

Типы, именование и атрибуты файлов

Файловые системы поддерживают несколько функционально различных типов файлов, в числе которых входят обычные файлы, содержащие информацию произвольного характера (текст, графика, звук и др.), файлы-каталоги, специальные файлы, именованные конвейеры, отображаемые в память файлы и др.

Обычные файлы, или просто файлы, или регулярные файлы, содержат информацию, которую в них заносит пользователь или которая образуется в результате работы системных и пользовательских программ. Большинство ОС не контролируют содержимое и структуру регулярных файлов, которые в основном являются ASCII-файлами либо двоичными файлами. ASCII-файлы состоят из текстовых строк. Они могут отображаться на экране и выводиться на печать без какого-либо преобразования, и могут редактироваться практически любым текстовым редактором. Двоичные файлы имеют определенную внутреннюю структуру, которая известна программе, использующей данный файл. При выводе двоичного файла на принтер получается случайный набор символов.

Каталоги – это системные файлы, обеспечивающие поддержку

структуры файловой системы. Они содержат системную справочную информацию о наборе файлов, сгруппированных пользователем по какому-либо неформальному признаку (договоры, рефераты, курсовые проекты и т.п.). Во многих ОС в каталог могут входить другие файлы, в том числе другие каталоги, за счет чего образуется древовидная структура, удобная для поиска требуемого файла. Каталоги устанавливают соответствие между именами файлов и их характеристиками, используемыми файловой системой для управления файлами. В число таких характеристик входит тип файла, права доступа к файлу, его расположение на диске, размер, дата и время создания и др.

Специальные файлы – это фиктивные файлы, ассоциированные с устройствами ввода-вывода, которые используются для унификации механизма доступа к последовательным устройствам ввода-вывода, таким как терминалы, принтеры и др. (например, MS-DOS рассматривает монитор и клавиатуру как файлы со стандартным именем con – консоль, а принтер – как файл prn). Блочные специальные файлы используются для моделирования дисков.

Именованные конвейеры (каналы) представляют собой циклические буферы, позволяющие выходной файл одной программы соединить со входным файлом другой программы.

Наконец, отображаемые файлы – это обычные файлы, отображенные на адресное пространство процесса по указанному виртуальному адресу.

Файлы относятся к абстрактному механизму. Они предоставляют способ сохранять информацию на запоминающем устройстве и считывать ее позднее снова. При этом от пользователя должны скрываться такие детали, как способ и место хранения информации, а также детали работы устройства.

Наиболее важной характеристикой любого механизма абстракции является именование управляемых объектов. Правила именования файлов меняются от одной ОС к другой, но, как правило, все современные операционные системы поддерживают использование в качестве имен файлов 8-символьные текстовые строки. Часто в именах разрешается использование цифр и специальных символов. В некоторых файловых системах различаются прописные и строчные символы, тогда как в других, например, MS-DOS, – нет.

Во многих операционных системах имя файла состоит из двух частей, разделенных точкой. Часть имени после точки называется расширением файла и обычно означает его тип. Так, в MS-DOS имя файла может содержать от 1 до 8 символов, а расширение от 0 (отсутствует) до 3.

В некоторых ОС, например, Windows, расширение указывает на программу, создавшую файл. Другие ОС, например, UNIX, не принуждают пользователя строго придерживаться расширений. Некоторые типичные расширения файлов приведены ниже.

Расширение	Значение
file.bak	Резервная копия файла
file.cpp	Исходный текст программы на C++
file.gif	Изображение формата GIF
file.hlp	Файл справки
file.html	Документ в формате HTML
file.jpg	Неподвижное изображение стандарта JPEG
file.mp3	Музыка в формате MPEG-1 уровень 3
file.mpg	Фильм в формате MPEG
file.obj	Объектный файл

В иерархически организованных файловых системах обычно используются три типа имен файлов: простые, составные и относительные.

Простое (короткое) символьное имя идентифицирует файл в пределах одного каталога. Несколько файлов могут иметь одно и то же простое имя, если они принадлежат разным каталогам.

Составное (полное) символьное имя представляет собой цепочку, содержащую имя диска и имена всех каталогов, через которые проходит путь от корневого каталога до данного файла.

Относительное имя файла определяется через текущий каталог, т.е. каталог, в котором в данный момент времени работает пользователь. Таким образом, относительных имен у файла может быть достаточно много, и все они являются частью полного имени.

Понятие файла включает не только хранимые им данные и имя, но и информацию, описывающую свойства файла. Эта информация составляет атрибуты файла. Список атрибутов может быть различным в различных ОС. Пример возможных атрибутов приведен ниже.

Атрибут	Значение
Тип файла	Обычный, каталог, специальный и т.д.
Владелец файла	Текущий владелец
Создатель файла	Идентификатор пользователя, создавшего файл
Пароль	Пароль для получения доступа к файлу
Время	Создания, последнего доступа, последнего изменения
Текущий размер файла	Количество байт в записи
Максимальный размер	Количество байт, до которого можно увеличивать размер файла
Флаг "только чтение"	0 – чтение / запись, 1 – только чтение
Флаг "скрытый"	0 – нормальный, 1 – не показывать в перечне файлов каталога
Флаг "системный"	0 – нормальный, 1 – системный
Флаг "архивный"	0 – заархивирован, 1 – требуется архивация
Флаг ASCII / двоичный	0 – ASCII, 1 – двоичный
Флаг произвольного доступа	0 – только последовательный доступ, 1 – произвольный доступ
Флаг "временный"	0 – нормальный, 1 – удаление после окончания работы процесса
Позиция ключа	Смещение до ключа в записи
Длина ключа	Количество байт в поле ключа

Пользователь может получить доступ к атрибутам, используя средства, предоставленные для этой цели файловой системой. Обычно разрешается читать значение любых атрибутов, а изменять – только некоторые.

Значения атрибутов файлов могут содержаться в каталогах, как это сделано, например, в MS-DOS (рис. 7.7). Другим вариантом является размещение атрибутов в специальных таблицах, в этом случае в каталогах содержатся ссылки на эти таблицы.

Имя файла	Расширение	Атрибуты	Резерв	Время	Дата	Номер первого кластера	Размер
8 байт	3 байта	1 байт	10 байт	2 байта	2 байта	2 байта	4 байта

Рис. 7.7. Атрибуты файлов MS DOS

Логическая организация файла

В общем случае данные, содержащиеся в файле, имеют некоторую логическую структуру. Эта структура (организация) файла является базой при разработке программы, предназначенной для обработки этих данных. Поддержание структуры данных может быть целиком возложено на приложение либо в той или иной степени эту работу может взять на себя файловая система.

В первом случае, когда все действия, связанные со структуризацией и интерпретацией содержимого файла, целиком относятся к ведению приложения, файл представляется файловой системе неструктурированной последовательностью данных. Приложение формирует запросы к файловой системе на ввод-вывод, используя общие для всех приложений системные средства, например, указывая смещение от начала файла и количество байт, которые необходимо считать или записать. Поступивший к приложению поток байт интерпретируется в соответствии с заложенной в программе логикой. Следует подчеркнуть, что интерпретация данных никак не связана с действительным способом их хранения в файловой системе.

Модель файла, в соответствии с которой содержимое файла представляется неструктурированной последовательностью байт, стала популярной вместе с ОС UNIX, а теперь широко используется в

современных ОС. Неструктурированная модель файла позволяет легко организовать разделение файла между несколькими приложениями, поскольку разные приложения могут по-своему структурировать и интерпретировать данные, содержащиеся в файле.

Другая модель файла – структурированный файл. В этом случае поддержание структуры файла поручается файловой системе. Файловая система видит файл как упорядоченную последовательность логических записей. ФС предоставляет приложению доступ к записи, а вся дальнейшая обработка данных, содержащихся в этой записи, выполняется приложением.

Известно пять фундаментальных способов организации файлов [11]:

- смешанный файл,
- последовательный файл,
- индексно-последовательный файл,
- индексируемый файл,
- файл прямого доступа.

При выборе способа организации файла нужно учитывать несколько критериев:

- быстрота доступа,
- легкость обновления,
- экономность хранения,
- простота обслуживания,
- надежность.

Смешанный файл. Это наименее сложная форма организации файла. Данные накапливаются в порядке поступления. Запись состоит из одного пакета данных. Записи могут иметь различные или одинаковые поля, расположенные в различном порядке (рис. 7.11). Каждое поле описывает само себя, включая как имя, так и значение. Длина каждого поля должна быть указана явно либо посредством применения разделителей.

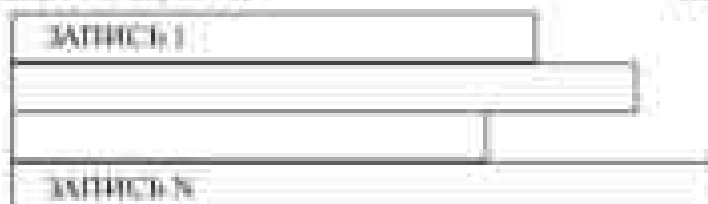


Рис. 7.8. Смешанный файл

Поскольку смешанный файл не имеет никакой структуры, доступ к записи осуществляется полным перебором всех записей файла. Смешанные файлы применяются в том случае, когда данные накапливаются и сохраняются перед обработкой, или если данные неудобны для организации. Файлы этого типа рационально используют дисковое пространство, хорошо подходят для полного набора. Обновление записей достаточно сложно, так же как и вставка записи.

Последовательный файл. Для записей используется фиксированный формат. Все записи имеют одинаковую длину (но иногда и не одинаковую) и состоят из одинакового количества полей фиксированной длины, организованных в определенном порядке (рис. 7.9). Поскольку длина и позиция каждого поля известны, сохранению подлежат только значения полей. Атрибутами файловой структуры является имя и длина каждого поля.

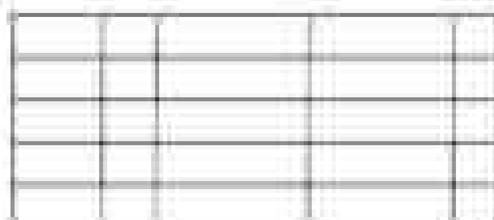


Рис. 7.9. Последовательный файл

Одно определенное поле (или несколько полей) называется ключевым. Оно однозначно идентифицирует запись, так как это поле различно для каждой записи. Более того, записи сохраняются в "ключевой" последовательности: в алфавитном порядке для текстового ключа и в числовом – для числового. Последовательные файлы часто используются пакетными приложениями и обычно являются

оптимальным вариантом, если это приложение выполняет обработку всех записей. Удобно и то, что такой файл можно хранить как на ленте, так и на магнитном диске.

Для диалоговых приложений последовательный файл малоэффективен, поскольку для нахождения нужной записи требуется последовательный перебор записей файла. Правда, если в оперативную память загрузить весь файл, возможен более эффективный метод поиска. Дополнения к файлу или изменения в записях создают проблемы.

Обычно последовательный файл сохраняется с последовательной организацией записей внутри блока, т.е. физическая организация файла в точности соответствует логической. Новые записи размещаются в отдельном смешанном файле, называемом журнальным файлом, или файлом транзакций. Периодически в пакетном режиме выполняется слияние основного и журнального файлов в новый файл с корректной последовательностью ключей.

Альтернативной организацией может быть физическая организация в виде списка с использованием указателей. В каждом физическом блоке сохраняется одна или несколько записей, и каждый блок содержит указатель на следующий блок. Для вставки новых записей достаточно изменить указатели, и нет необходимости в том, чтобы новые записи занимали определенную физическую позицию. Это удобство достигается за счет определенных накладных расходов и дополнительной работы. Если в последовательном файле записи имеют одну и ту же длину, то можно вычислить адрес требуемой записи по ее номеру, номеру текущей записи и длине записи. Если записи имеют переменную длину, такой подход неприменим.

Индексно-последовательный файл. Одним из методов преодоления недостатков последовательного файла является индексно-последовательная организация файла. В этом случае файл состоит из трех частей (файлов): главный файл, содержащий записи с последовательно идущими ключами, индексный файл, содержащий индексное поле, и указатель в главный с ключами, файл переполнения (рис. 7.10).

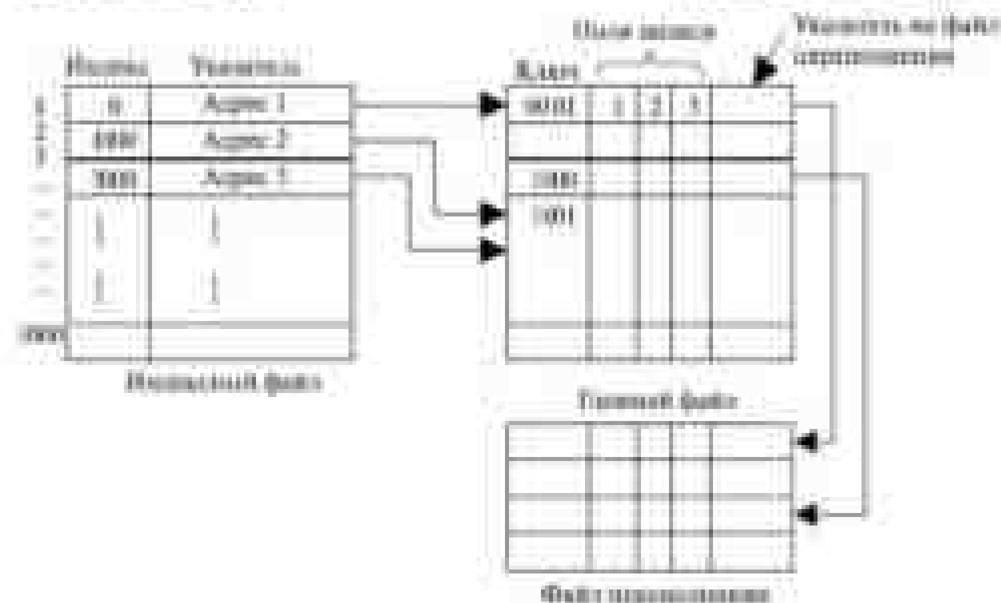


Рис. 7.10. Индексно-последовательный файл

Для поиска нужной записи по ее ключу сначала выполняется поиск в индексном файле. После того как в нем найдено наибольшее значение ключа, которое не превышает искомое, продолжается поиск в главном файле. Например, пусть последовательный файл (главный) содержит 1 млн записей. Для поиска определенного ключевого значения необходимо в среднем 0,5 млн операций доступа к записям. Если создать индексный файл, содержащий 1000 элементов, то потребуется в среднем 500 операций доступа к индексному файлу, после чего еще нужно в среднем 500 операций доступа к главному файлу. В результате средняя длина поиска уменьшилась с 0,5 млн до 1000. Еще лучшего результата можно достичь, используя многоуровневую индексацию. При этом нижний уровень индексного файла рассматривается как последовательный файл, для которого создается индексный файл верхнего уровня.

Дополнения к файлу обрабатываются следующим образом. В каждой записи главного файла содержится дополнительное поле, невидимое для приложения и являющееся указателем на файл переполнения. Если в файле производится вставка новой записи, она добавляется в файл переполнения. Запись в главном файле, непосредственно

предшествующая новой записи в логической последовательности, обновляется и указывает на новую запись в файле переполнения. Время от времени выполняется слияние индексно-последовательного файла с файлом переполнения.

Индексированный файл. Индексно-последовательный файл сохраняет одно ограничение последовательного файла: эффективная работа с файлом ограничена работой с ключевым полем. Если необходимо производить поиск записи по какой-либо иной характеристике, отличной от ключевого поля, то оказываются непригодными обе организации последовательного файла, в то время как в некоторых приложениях эта гибкость крайне желательна.

Для достижения гибкости необходимо применение большого количества индексов, по одному для каждого типа поля, которое может быть объектом поиска. В обобщенном индексированном файле доступ к записям осуществляется только по их индексам. В результате к размещению записей нет никаких ограничений до тех пор, пока указатель по крайней мере в одном индексе ссылается на эту запись. Кроме того, в таком файле легко реализуются записи переменной длины.

Используется два типа индексов. Полный индекс содержит по одному элементу для каждого типа записей главного файла. Сам по себе индекс организовывается в виде последовательного файла для облегчения поиска. Частный индекс содержит элементы для записей, в которых имеется интересующее пользователя поле. При добавлении новой записи в главный файл необходимо обновлять все индексные файлы.

Индексированные файлы применяются теми приложениями, в которых время доступа к информации является критической характеристикой и редко требуется обработка всех записей в файле.

Файл прямого доступа. Такой файл использует возможность прямого доступа к блоку с известным адресом при хранении файлов на диске. В каждой записи в этом случае также имеется ключевое поле.

7.14. Каталоговые системы

Связующим звеном между системой управления файлами и набором файлов служит файловый каталог. Простейшая форма системы каталогов состоит в том, что имеется один каталог, в котором содержатся все файлы. Каталог содержит информацию о файлах, включая атрибуты, местоположение, принадлежность. Пользователи обращаются к файлам по символьным именам. Однако способности человеческой памяти ограничивают количество имен объектов, к которым пользователь может обращаться по именам. Иерархическая организация пространства имен позволяет значительно расширить эти границы. Именно поэтому каталоговые системы имеют иерархическую структуру. Граф, описывающий иерархию каталогов, может быть деревом или сетью. Каталоги образуют дерево, если файлу разрешено входить только в один каталог (рис. 7.11), и сеть, если файл может входить в несколько каталогов.

Например, в Ms-Dos и Windows каталоги образуют древовидную структуру, а в UNIX – сетевую. В общем случае вычислительная система может иметь несколько дисковых устройств, даже в ПК всегда имеется несколько дисков: гибкий, винчестер, CD-ROM (DVD). Как организовать хранение файлов в этом случае?

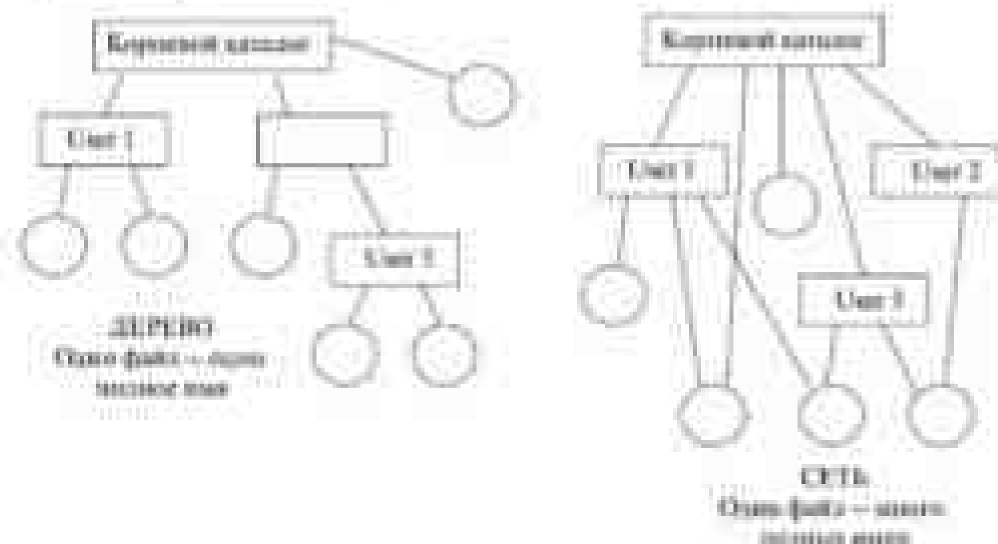


Рис. 7.11. Каталоговые системы

Первое решение состоит в том, что на каждом из устройств размещается

автономная файловая система, т.е. файлы, находящиеся на этом устройстве, описываются деревом каталогов, никак не связанным с деревьями каталогов на других устройствах. В таком случае для однозначной идентификации файла пользователь вместе с составным символьным именем файла должен указывать идентификатор логического устройства. Примером такого автономного существования может служить MS-DOS, Windows 95/98/Me/XP.

Другим решением является такая организация хранения файлов, при которой пользователю предоставляется возможность объединять файловые системы, находящиеся на разных устройствах, в единую файловую систему, описываемую единым деревом каталогов. Такая операция называется монтированием.

В ОС UNIX монтирование осуществляется следующим образом. Среди всех имеющихся логических дисковых устройств выделяется одно, называемое системным. Пусть имеются две файловые системы, расположенные на разных логических дисках, причем один из дисков является системным (рис. 7.12).

Файловая система, расположенная на системном диске, называется корневой. Для связи иерархий файлов в корневой файловой системе выбирается некоторый существующий каталог, в данном примере – каталог /ос. После выполнения монтирования выбранный каталог /ос становится корневым каталогом второй файловой системы. Через этот каталог монтируемая файловая система подсоединяется как поддерево к общему дереву.

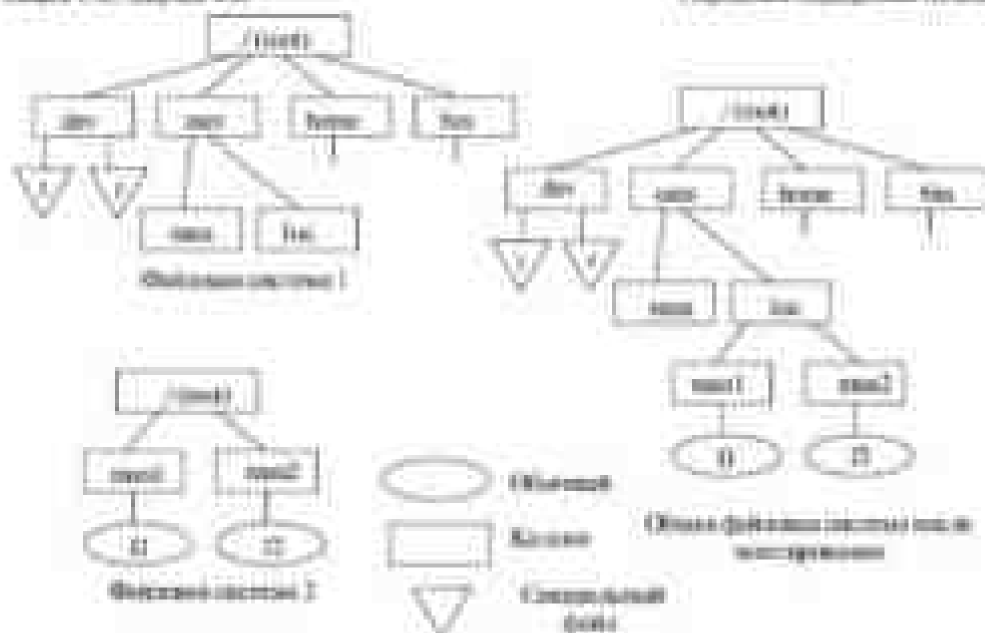


Рис. 7.12. Монтирование

7.15. Физическая организация файловой системы

Информационная структура магнитных дисков

Представление пользователей о файловой системе как об иерархически организованном множестве информационных блоков имеет мало общего с порядком хранения файлов на диске. Файл, имеющий образ цельного, непрерывающегося набора байт, на самом деле разбросан своими частями по всему диску, причем это разбиение никак не связано с логической структурой файла. Точно так же логически объединенные файлы из одного каталога совсем не обязательно соседствуют на диске. Принципы размещения файлов, каталогов и системной информации на реальном устройстве описываются физической организацией файловой системы. При этом ясно, что разные файловые системы имеют разную физическую организацию.

Основным устройством для хранения файлов являются жесткие и гибкие магнитные диски. Жесткий диск состоит из одной или нескольких стеклянных или металлических пластин, каждая из которых

покрыта с одной стороны или двух сторон магнитным материалом.

На каждой стороне каждой пластины размечены тонкие концентрические кольца – дорожки (traks), на которых хранятся данные. Нумерация дорожек начинается с 0 от внешнего края к центру диска. Когда диск вращается, магнитные головки, имеющиеся над (под) каждой поверхностью диска, считывают или записывают двоичные данные на магнитные дорожки. Головки могут позиционироваться над каждой дорожкой, если на одну поверхность диска в устройстве имеется одна головка. Некоторые диски имеют по отдельной головке на каждую дорожку, тогда позиционирование головок не требуется, что повышает быстродействие диска.

Совокупность дорожек одного радиуса на всех поверхностях пластин пакета называется цилиндром (cylinder). Каждая дорожка разбивается на фрагменты, называемые секторами (sectors) или блоками (blocks), так что все дорожки имеют равное число секторов, в которые можно максимально записать одно и то же число байт. Сектор имеет фиксированный для данной системы размер, выражающийся степенью двойки (чаще всего 512 байт).

Сектор – наименьшая адресуемая единица обмена данными диска с оперативной памятью. Для того чтобы контроллер мог найти на диске нужный сектор, ему необходимо задать все составляющие адреса сектора: номер цилиндра, номер поверхности и номер сектора. Типичный запрос включает чтение (запись) нескольких секторов, содержащих наряду с требуемыми избыточные данные.

Операционная система при работе с диском использует, как правило, единицу дискового пространства, называемую кластером (cluster) и содержащую несколько секторов в числе, кратном степени двойки. Это связано с тем, что применение более мелкой единицы дискового пространства – сектора – усложняет учет свободного и занятого пространства диска при современных больших емкостях дисков, исчисляющихся десятками и сотнями Гбайт.

Дорожки и секторы создаются в результате выполнения процедуры физического (низкоуровнего) форматирования диска, предшествующей использованию диска. Для определения границ блоков на диск записывается идентификационная информация. Низкоуровневый

формат диска не зависит от типа ОС, которая с этим диском будет работать.

Разметку диска под конкретный тип файловой системы выполняют процедуры высокоуровнего, или логического, форматирования. При высокоуровневом форматировании определяется *размер кластера*, записываются информация, необходимая для работы файловой системы, и загрузчик ОС – небольшая программа, которая начинает процесс инициализации операционной системы после включения питания.

Прежде чем форматировать диск под определенную файловую систему, он может быть разбит на разделы. Раздел – это непрерывная часть физического диска, которую операционная система представляет пользователю как логическое устройство (логический диск или логический раздел). На каждом разделе может создаваться только одна файловая система.

В IBM-совместимых ПК сектор 1 диска называется главной загрузочной записью (*MBR – Master Boot Record*) и используется для загрузки компьютера. В конце *MBR* содержится таблица разделов. В ней хранятся начальные и конечные адреса (номера блоков) каждого раздела. Один из разделов помечен в таблице как активный. При загрузке компьютера BIOS считывает и исполняет *MBR*-запись, после чего загрузчик в *MBR*-записи определяет активный раздел диска, считывает его первый блок (загрузчик) и исполняет его. Программа, находящаяся в загрузочном блоке, загружает операционную систему, содержащуюся в этом разделе. Для единообразия каждый дисковый раздел начинается с загрузочного блока, даже если в нем не содержится операционной системы. К тому же в этом разделе может быть в дальнейшем установлена операционная система, поэтому зарезервированный загрузочный блок называется полезным.

Таблица разделов располагается в *MBR* по смещению 0x1BE и содержит четыре элемента. Структура записи элемента таблицы разделов приведена ниже.

Наименование записи элемента таблицы разделов	Длина, байт
Флаг активности раздела	1

Номер головки начала раздела	1
Номера сектора и цилиндра загрузочного сектора раздела	2
Кодовый идентификатор операционной системы	1
Номер головки конца раздела	1
Номера сектора и цилиндра последнего сектора раздела	2
Младшее и старшее двухбайтовые слова относительного номера начального сектора	4
Младшее и старшее двухбайтовые слова размера раздела в секторах	4

Каждый элемент таблицы описывает один раздел, причем двумя способами: через координаты *C-H-S* начального и конечного секторов, а также через номер первого сектора в спецификации *LBA* (*Logical Block Addressing*) и общее число секторов в разделе [10]. Последние два байта *MBR* имеют значение 55AAh, т.е. чередующиеся значения 0 и 1. Эта сигнатура выбрана для того, чтобы проверить работоспособность всех линий передачи данных. Значение 55AAh, присвоенное последним двум байтам, имеется во всех загрузочных секторах.

Разделы диска могут быть двух типов: первичные (primary) и расширенные (extended). Максимальное число первичных разделов равно четырем. Из них только один может быть активным. Именно загрузчику, расположенному в активном разделе, передается управление при включении компьютера с помощью внесистемного загрузчика. Согласно принятым спецификациям на одном жестком диске может быть только один расширенный раздел, который может быть разделен на *логические диски* (рис. 7.13). *Расширенный раздел* содержит вторичную запись *MBR*, в состав которой вместо таблицы разделов входит аналогичная ей таблица логических дисков (Logical Disks Table, LDT). Эта таблица описывает размещение и характеристики раздела, содержащего единственный логический диск, а также может специфицировать следующую запись SMBR (Secondary MBR).

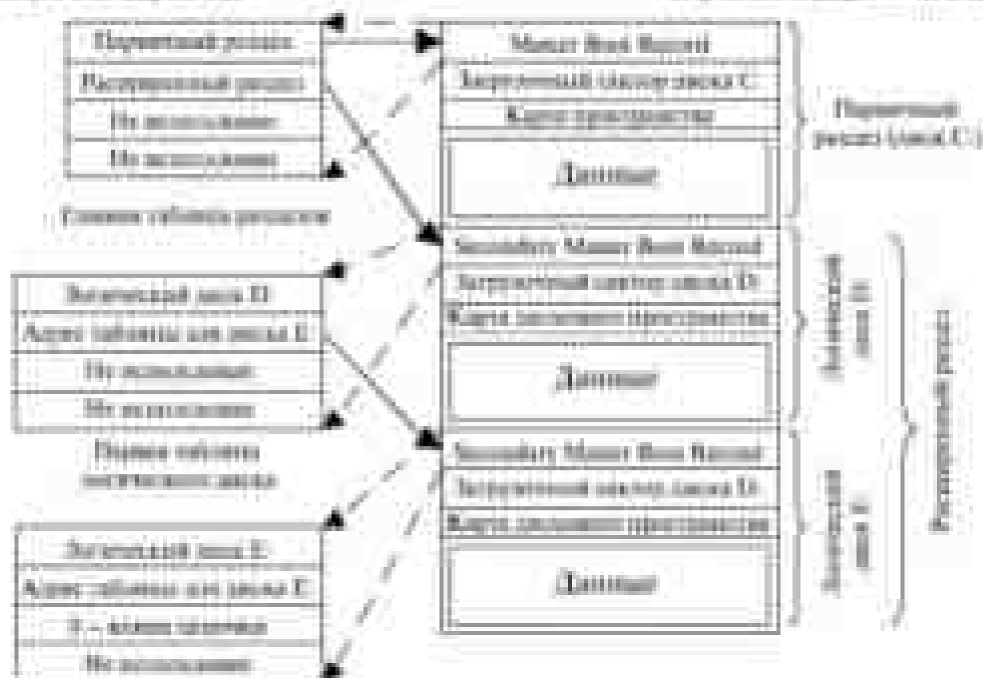


Рис. 7.13. Разделы диска

Во всем остальном строение раздела диска меняется от системы к системе. Часто файловая система содержит некоторые элементы, показанные на рис. 7.14. Один из таких элементов называется суперблоком и содержит ключевые параметры файловой системы, и считывается в память при загрузке компьютера. Следом располагается информация о свободных блоках файловой системы. За этими данными может следовать информация об i-узлах, содержащих информацию о файлах. Следом может размещаться каталог и затем – остальные файлы и каталоги.

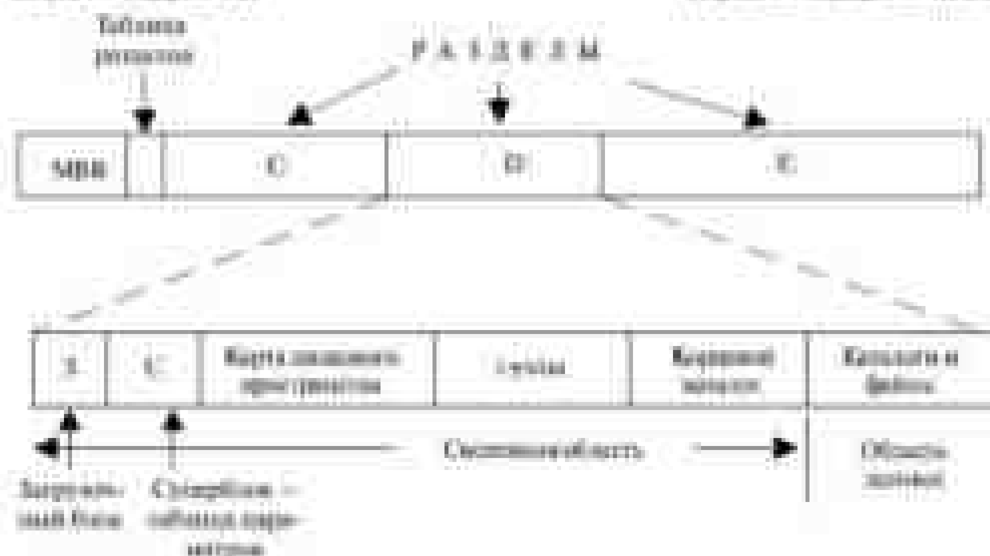


Рис. 7.14. Структура раздела

На разных логических устройствах одного и того же физического диска могут располагаться файловые системы разного типа. Все разделы одного диска имеют одинаковый размер блока, определенный для данного диска в результате низкоуровневого форматирования. Однако в результате высокоуровневого форматирования в разных разделах одного и того же диска могут быть установлены различные файловые системы с различными размерами кластеров.

7.16. Физическая организация и адресация файла

Физическая организация определяет способ размещения файлов на диске и учет соответствия блоков диска файлам. Основными критериями эффективности физической организации файлов являются:

- скорость доступа к данным;
- объем адресной информации файла;
- степень фрагментированности дискового пространства;
- максимально возможный размер файла.

Наиболее часто используются следующие схемы размещения файлов:

- непрерывное размещение (непрерывные файлы);
- связный список блоков (кластеров) файла;
- связный список индексов блоков (кластеров) файла;
- перечень номеров блоков (кластеров) файла в структурах, называемых i-узлами (index-node – индекс-узел).

Простейший вариант физической организации – непрерывное размещение в наборе соседних кластеров (рис. 7.15а). Достоинство этой схемы – высокая скорость доступа и минимальный объем адресной информации, поскольку достаточно хранить номер первого кластера и объем файла. Размер файла при такой организации не ограничивается.

Однако у этой схемы имеется серьезный недостаток – фрагментация, возрастающая по мере удаления и записи файлов. Кроме того, возникает вопрос, какого размера область нужно выделить файлу, если при каждой модификации он может увеличить свой размер.

И все-таки есть ситуации, в которых непрерывные файлы могут эффективно использоваться и действительно широко применяются – на компакт-дисках. Здесь все размеры файлов заранее известны и не могут меняться.

Второй метод размещения файлов состоит в представлении файла в виде связного списка кластеров дисковой памяти (рис. 7.15б). Первое слово каждого кластера используется как указатель на следующий кластер. В этом случае адресная информация минимальна, поскольку расположение файла задается номером его первого кластера.

Рис. 7.15. Варианты физической организации файлов

Кроме того, отсутствует фрагментация на уровне кластеров, а файл легко может изменить размер наращиванием или удалением цепочки кластеров. Однако доступ к такому файлу может оказаться медленным, так как для получения доступа к кластеру n операционная система должна прочитать первые n-1 кластеры. Кроме того, размер кластера уменьшается на несколько байтов, требуемых для хранения. Указатели это не очень важно, но многие программы читают и пишут блоками, кратными степени двойки.

Оба недостатка предыдущей схемы организации файлов могут быть устранены, если указатели на следующие кластеры хранить в отдельной таблице, загружаемой в память. Таким образом, образуется связный список не самих блоков (кластеров) файла, а индексов, указывающих на эти блоки (рис. 7.16).

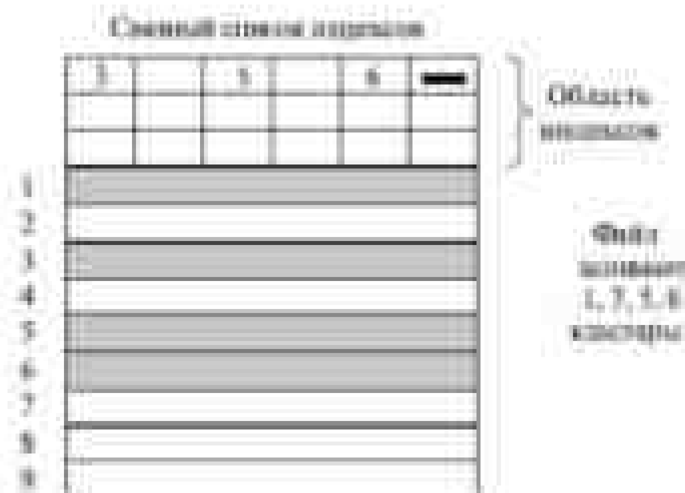


Рис. 7.16. Вариант физической организации файлов

Такая таблица, называемая FAT-таблицей (File Allocation Table), используется в файловых системах MS-DOS и Windows (FAT 16 и FAT 32). Файлу выделяется память на диске в виде связного списка кластеров. Номер первого кластера запоминается в записи каталога, где хранятся характеристики этого файла. С каждым кластером диска связывается индекс. Индексы располагаются в FAT-таблице в отдельной области диска. Когда память свободна, все индексы равны нулю. Если некоторый кластер N назначен файлу, то индекс этого кластера либо

становится равным номеру M следующего кластера файла, либо принимает специальное значение, являющееся признаком того, что кластер является последним для файла. Вообще индексы могут содержать следующую информацию о кластере диска (для FAT 32):

- не используется (Unused) – 0000.0000;
- используется файлом (Cluster in use by a file) – значение, отличное от 000.000, FFFF.FFFF и FFFF.FFF7;
- плохой кластер (*Bad cluster*) – FFFF.FFF7;
- последний кластер файла (Last cluster in a file) – FFFF.FFFF.

При такой организации сохраняются все достоинства второго метода организации файлов: отсутствие фрагментации, отсутствие проблем при изменении размера. Кроме того, данный способ обладает дополнительными преимуществами: для доступа к произвольному кластеру файла не требуется последовательно считывать его кластеры, достаточно прочитать FAT-таблицу, отсчитать нужное количество кластеров файла по цепочке и определить номер нужного кластера. Во-вторых, данные файла заполняют кластер целиком в объеме, кратном степени двойки.

Еще один способ заключается в простом перечислении номеров кластеров, занимаемых файлом (рис. 7.17). Этот перечень и служит адресом файла. Недостаток такого подхода – длина адреса зависит от размера файла. Достоинства – высокая скорость доступа к произвольному кластеру благодаря прямой адресации, отсутствие внешней фрагментации.

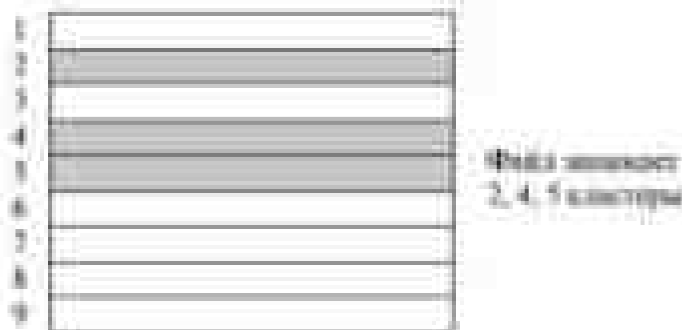


Рис. 7.17. Вариант физической организации файлов

Эффективный метод организации файлов, используемый в Unix-подобных операционных системах, состоит в связывании с каждым файлом структуры данных, называемой i-узлом. Такой узел содержит атрибуты файла и адреса кластеров файла (рис. 7.18). Преимущество такой схемы перед FAT-таблицей заключается в том, что каждый конкретный i-узел должен находиться в памяти только тогда, когда открыт соответствующий ему файл. Если каждый узел занимает n байт, а одновременно может быть открыто k файлов, то массив i-узлов займет в памяти k * n байт, что значительно меньше, чем FAT-таблица.

Это объясняется тем, что размер FAT-таблицы растет линейно с размером диска и даже быстрее, чем линейно, так как с увеличением количества кластеров на диске может потребоваться увеличить разрядность числа для хранения их номеров.

Рис. 7.18. Вариант физической организации файлов

Достоинством i-узлов является также высокая скорость доступа к произвольному кластеру файла, так как здесь применяется прямая адресация. Фрагментация на уровне кластеров также отсутствует.

Однако с такой схемой связана проблема, заключающаяся в том, что при выделении каждому файлу фиксированного количества адресов кластеров этого количества может не хватить. Выход из этой ситуации может быть в сочетании прямой и косвенной адресации. Такой подход реализован в файловой системе ufs, используемой в ОС UNIX, схема адресации в которой приведена на рис. 7.19.

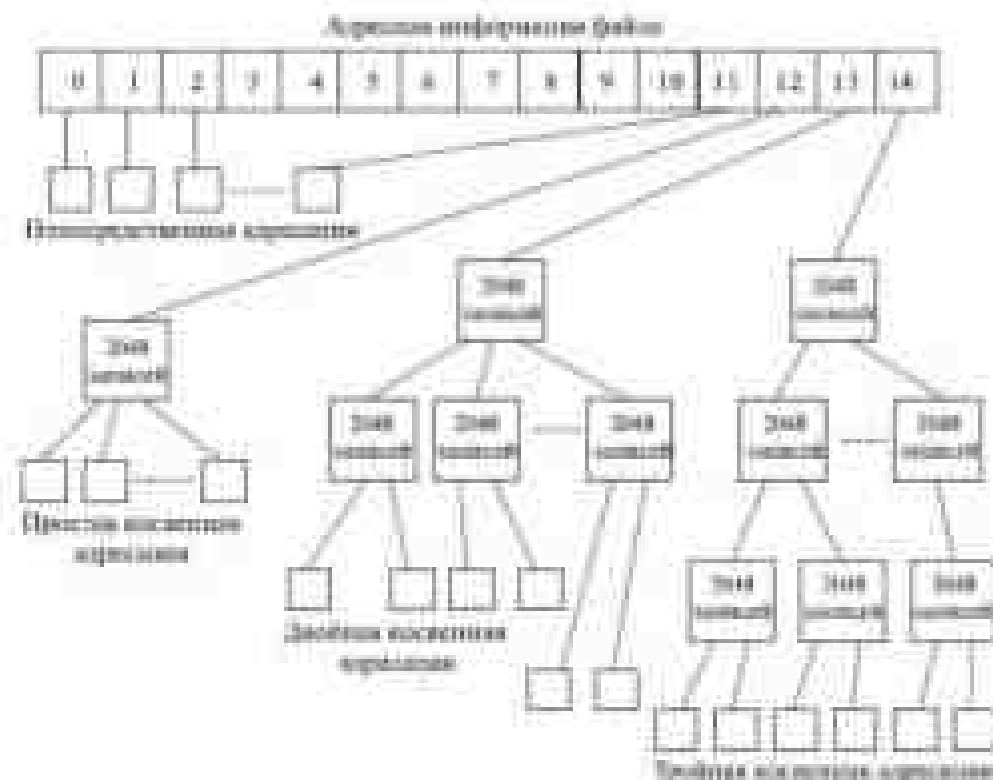


Рис. 7.19. Файловая система ufs

Для хранения адреса файла выделено 15 полей, каждое из которых состоит из 4 байт. Если размер файла меньше или равен 12 кластерам, то номера этих кластеров непосредственно перечисляются в первых двенадцати полях адреса. Если кластер имеет размер 8 Кбайт, то можно адресовать файл размером до 8 Кбайт * 12 = 98304 байт. Если размер кластеры превышает 12 кластеров, то следующее 13 поле содержит адрес кластера, в котором могут быть расположены номера следующих кластеров, и размер файла может возрасти до 8192 * (12 + 2048) = 16.875.520 байт.

Следующий уровень адресации, обеспечиваемый 14-м полем, позволяет адресовать до 8192 * (12 + 2048 + 20482) = 3,4376*1010 байт. Если и этого недостаточно, используется следующее 15-е поле. В этом случае максимальный размер файла может составить 8192 * (12 + 2048 + 20482 + 20483) = 7,0403*1013 байт.

При этом объем самой адресной информации составит всего 0,05% от объема адресуемых данных (ядрен!!!).

Метод перечисления адресов кластеров файла задействован и в файловой системе NTFS, применяемой в Windows NT/2000/2003. Для сокращения объема адресной информации в NTFS адресуются не кластеры файла, а непрерывные области, состоящие из смежных кластеров диска. Каждая такая область называется экстентом (extent) и описывается двумя числами: номером начального кластера и количеством кластеров.

7.17. Физическая организация FAT-системы

Для обеспечения доступа приложений к файлам операционная система с файловой системой FAT использует следующие структуры:

- загрузочные секторы главного и дополнительных разделов;
- загрузочные секторы логических дисков (разделов);
- корневой каталог;
- область данных;
- цилиндр для выполнения диагностических операций чтения-записи.

На дискетах, в отличие от жесткого диска, нет загрузочных секторов главного и дополнительных разделов и диагностического цилиндра. Эти структуры создаются программой Fdisk, которая не применяется для дискет, так как они на разделы не разбиваются. Чтобы установить на один жесткий диск несколько операционных систем, его надо разбить на разделы. В загрузочном секторе главного раздела создается таблица списка разделов.

Загрузочный сектор главного раздела (называемый главной загрузочной записью – *Master Boot Record – MBR*) является первым сектором на

жестком диске (цилиндр 0, головка 0, сектор 1) и состоит из двух элементов [10]:

- таблица главного раздела, содержащая список разделов (максимум четыре) и расположение загрузочных секторов соответствующих логических дисков (первая и последняя головки, первый и последний цилиндры с соответствующими номерами секторов, а также количество секторов);
- главный загрузочный код – небольшая программа, которая выполняется системой BIOS. Основная функция этого кода – передача управления в раздел, который обозначен как активный (загрузочный).

Загрузочный сектор раздела содержит:

- блок параметров диска, в котором содержится информация о разделе (размер, количество секторов, размер кластера, метка тома и др.);
- загрузочный код – программу, с которой начинается процесс загрузки операционной системы (для Ms-Dos и Windows 9x – файл Io.sys).

Загрузочные секторы логических дисков создаются программой Format. Они похожи на загрузочные диски разделов. Однако при загрузке выполняется код только того сектора, который находится в активном разделе.

Логический диск, отформатированный программой Fdisk, состоит из следующих областей (рис. 7.20):

- загрузочный сектор;
- основная FAT-таблица, содержащая информацию о размещении файлов и каталогов на диске;
- копия FAT-таблицы;
- корневой каталог – фиксированная область (16 Кбайт для жесткого диска), позволяющая хранить 512 записей о файлах и каталогах (каждая запись состоит из 32 байтов);
- область данных для размещения всех файлов и каталогов, кроме

корневого каталога.

Первые две записи FAT зарезервированы и содержат информацию о самой FAT, все остальные указывают на соответствующие кластеры диска. Индексный указатель принимает значение, характеризующее состояние связанного с ним кластера (для FAT 16):

- кластер свободен (0000h);
- кластер используется (любое значение, кроме специальных);
- последний кластер файла (FFF8h – FFFFh);
- кластер поврежден (FFF7h);
- резервный кластер (FFF6h).

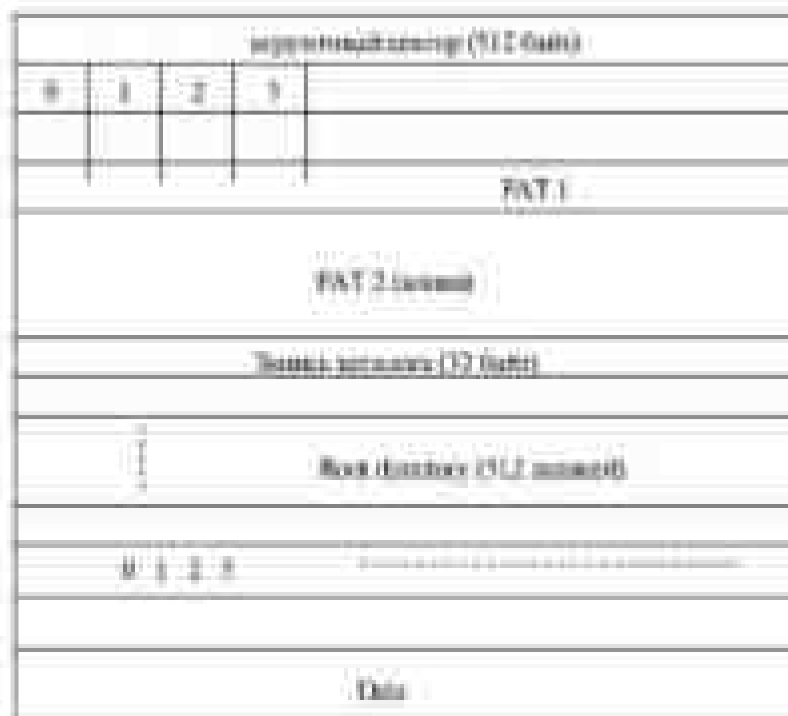


Рис. 7.20. FAT-система

Размер FAT-таблицы определяется количеством кластеров. Разрядность индексного указателя FAT-таблицы должна быть такой, чтобы можно было задать максимальный номер кластера диска определенного объема. В соответствии с разрядностью дискового указателя существуют

нескольких разновидностей FAT: FAT12, *FAT16*, FAT32 (соответственно 2^{12}, 2^{16} и 2^{32} кластеров). Тип используемой FAT определяется программой Fdisk, хотя и записывается она в процессе форматирования высокого уровня программы Format. На всех дискетах применяется FAT 12, на жестких дисках до 512 Мбайт – *FAT16*, на жестких дисках, имеющих большую емкость при использовании Windows 95 OSR2 и Windows98 – FAT 32 (вообще *размер кластера* может быть от 1 до 128 секторов или от 512 байт до 64 Кбайт). Максимальный размер раздела *FAT16* ограничен объемом 4 Гбайт (2^{16} = 65536 кластеров по 64 Кбайт). Максимальный размер раздела FAT 32 практически не ограничен (232 кластеров по 32 Кбайт).

За копией FAT-таблицы следует корневой каталог – база данных, содержащая информацию о записанных на диске данных. Каждая запись в ней имеет длину 32 байта и содержит всю информацию о файле, которой располагает операционная система. Формат записи приведен ниже.

Смещение			Описание
Hex	Dec	Длина поля	
00h	0	8 байт	Имя файла
08h	8	3 байт	Расширение файла
0Bh	11	1 байт	Атрибуты файла
0Ch	12	10 байт	Зарезервировано
16h	22	2 байт	Время создания
18h	24	2 байт	Дата создания
1Ah	26	2 байт	Начальный кластер
1Ch	28	4 байт	Размер файла в байтах

Информация о расположении файла, то есть о расположении остальных кластеров, содержится в FAT-таблице. В процессе работы системы кластеры файла могут оказаться не в смежных областях, а будут чередоваться с кластерами других файлов. Однако эту цепочку кластеров легко выделить, зная начальный кластер файлов. На рис. 7.21 показан пример размещения двух файлов.

В корневом каталоге хранятся записи не только о файлах, но и подкаталогах. Эти записи имеют точно такую же структуру, что и записи корневого каталога. Признак подкаталогов указывается в атрибутах файла, т.е. можно считать, что подкаталог – это специальный файл. Структура атрибутивного байта показана ниже.

Позиция бита в шестнадцатеричном формате	Значение	Описание	
7	6	5	4 3 2 1 0
0	0	0	0 0 0 0 1 01h Только чтение
0	0	0	0 0 0 1 0 02h Скрытый
0	0	0	0 0 1 0 0 04h Системный
0	0	0	0 1 0 0 0 08h Метка тома
0	0	0	1 0 0 0 0 10h Подкаталог
0	0	1	0 0 0 0 0 20h Архивный (измененный)
0	1	0	0 0 0 0 0 40h Зарезервировано
1	0	0	0 0 0 0 0 80h Зарезервировано

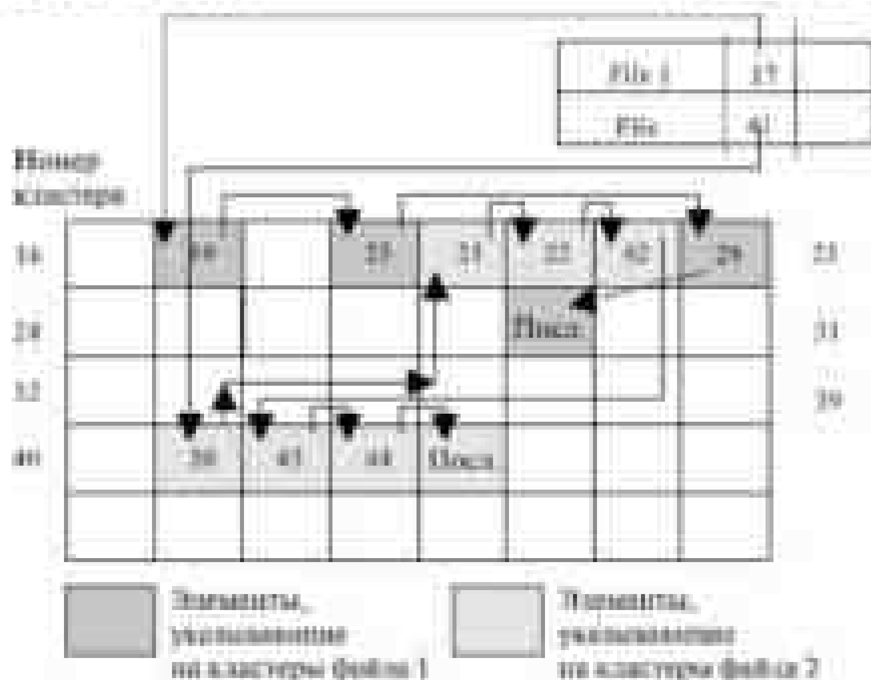


Рис. 7.21. Пример размещения двух файлов

Файловые системы FAT 12 и *FAT16* оперируют с именами файлов, составленных по схеме 8.3 (имя, расширение). В Windows 95 с появлением 32-разрядной виртуальной FAT-VFAT (Virtual file allocation table) поддерживаются имена длиной 255 символов (заметим, что изменился лишь программный код, поддерживающий *FAT16*, он стал 32-м). Для обеспечения обратной совместимости ОС создает его псевдоним, удовлетворяющий стандарту 8.3. Делается это следующим образом.

1. Первые 3 символа после последней точки в длинном имени файла становятся расширением псевдонима.
2. Первые шесть символов длинного имени файла, за исключением пробелов, которые игнорируются, преобразуются в символы верхнего регистра и становятся шестью символами стандартного имени файла. Недопустимые символы (+ , ; = [ ]), которые могут использоваться в Windows 95, преобразуются в символы подчеркивания.
3. Добавляются символы ~1 (седьмой и восьмой) к псевдониму

имени файла.

Если первые шесть символов нескольких файлов одни и те же, то добавляются символы ~2, ~3 и т.д.

VFAT хранит псевдонимы длинных имен в поле стандартных имен файлов записи каталога файла. Таким образом, все версии DOS и Windows могут получить доступ к файлу под длинным именем с помощью его псевдонима. Остается проблема: как хранить 255 символов имени файла в 32 байт записи каталога? Разработчики файловой системы решили эту проблему следующим образом: были добавлены дополнительные записи каталога для хранения длинных имен файлов. Чтобы предыдущие версии не повредили эти дополнительные записи каталога, *VFAT* устанавливает для них атрибуты, которые нельзя использовать для обычного файла: только для чтения, скрытый, системный и метка тома. Такие атрибуты DOS игнорирует, и следовательно, длинные имена файлов остаются нетронутыми. Подобным же образом решается проблема длинных имен в Windows NT/2000/2003/XP, применяющих для хранения имени двухбайтовый формат на каждый символ – Unicode.

Как уже отмечалось, выбор типа FAT-системы во многом определяется емкостью жесткого диска. При использовании FAT16 нельзя создать раздел емкостью более 2-х Гбайт. Для устранения этого ограничения фирма Microsoft разработала FAT 32. Она работает как FAT 16, но имеет отличие в организации хранения данных. Кроме того, FAT 32 можно установить с помощью программы FDisk. Впервые FAT 32 была реализована в Windows 95 OEM *Service Release* 2 (OSR2). Она встроена и в Windows 98/Me/NT/2000.

Основное преимущество FAT 32 – возможность использования 32-разрядных записей вместо 16-разрядных, что приводит к увеличению кластеров (вместо 216=65536) до 268 435 456 в разделе. Это значение в Windows 95 OSR2 эквивалентно 2^{28}, а не 2^{32}, поскольку 4 бита из 32 зарезервированы для других целей.

При работе в FAT 32 размер раздела может достигать 2 Тбайт при кластере размером 8, 16 или 32 Кбайт. Новая файловая система может иметь 232 кластеров размером 512 байт, а размер единичного файла

может составить 4 Гбайт. Реально FAT 32 поддерживает максимальный размер тома до 32 Гбайт. Это связано с тем, что в Windows 2000 это ограничение обусловлено программой Format. Вообще максимально возможный том – 2 Тбайт при кластере 32 Кбайт.

Существует важное отличие FAT 32 от ее предшественниц – положение корневого каталога: он может располагаться в любом месте раздела и иметь любой раздел. Это обеспечивает динамическое изменение размера раздела. Независимые разработчики использовали это свойство. Так, фирма Power-Quest создала программу Partion Magic, позволяющую перераспределять разделы после их создания.

Файловая система FAT 32 также использует преимущество двух копий FAT. Как и в FAT 16, в FAT 32 первая копия является основной и периодически копирует данные в дополнительную копию FAT. При проблемах с главной копией FAT система переключается в дополнительную копию, которая становится главной.

Примечание: программа Fdisk автоматически определяет размер кластера на основе выбранной файловой системы и размером раздела. Однако существует недокументированный параметр команды Format, позволяющий явно указать размер кластера: Format/z:n, где n –размер кластера в байтах, кратный 512.

7.18. Файловые операции

Набор файловых операций

Файловая система ОС должна предоставлять пользователям набор операций для работы с файлами, оформленный в виде системных вызовов. В различных ОС имеются различные наборы файловых операций. Наиболее часто встречающимися системными вызовами для работы с файлами являются [12, 17]:

1. Create (создание). Файл создается без данных. Этот системный вызов объявляет о появлении нового файла и позволяет установить некоторые его атрибуты;
2. Delete (удаление). Ненужный файл удаляется, чтобы освободить пространство на диске;

3. Open (открытие). До использования файла его нужно открыть. Данный вызов позволяет прочитать атрибуты файла и список дисковых адресов для быстрого доступа к содержимому файла;
4. Close (закрытие). После завершения операций с файлом его атрибуты и дисковые адреса не нужны. Файл следует закрыть, чтобы освободить пространство во внутренней таблице;
5. Read (чтение). Файл читается с текущей позиции. Процесс, работающий с файлом, должен указать (открыть) буфер и количество читаемых данных;
6. Write (запись). Данные записываются в файл в текущую позицию. Если она находится в конце файла, его размер автоматически увеличивается. В противном случае запись производится поверх существующих данных;
7. Append (добавление). Это усеченная форма предыдущего вызова. Данные добавляются в конец файла;
8. Seek (поиск). Данный системный вызов устанавливает файловый указатель в определенную позицию;
9. Get attributes (получение атрибутов). Процессам для работы с файлами бывает необходимо получить их атрибуты;
10. Set attributes (установка атрибутов). Этот вызов позволяет установить необходимые атрибуты файлу после его создания;
11. Rename (переименование). Этот системный вызов позволяет изменить имя файла. Однако такое действие можно выполнить копированием файла. Поэтому данный системный вызов не является необходимым;
12. Execute (выполнить). Используя этот системный вызов, файл можно запустить на выполнение.

Рассмотрим примеры файловых операций в ОС Windows 2000 и UNIX. Как и в других ОС, в Windows 2000 есть свой набор системных вызовов, которые она может выполнять. Однако корпорация Microsoft никогда не публиковала список системных вызовов Windows, кроме того, она постоянно меняет их от одного выпуска к другому [12]. Вместо этого Microsoft определила набор функциональных вызовов, называемый Win 32 API (Win 32 Application Programming Interface). Эти вызовы опубликованы и полностью документированы. Они представляют собой библиотечные процедуры, которые либо обращаются к системным вызовам, чтобы выполнить требуемую работу, либо выполняют ее прямо в пространстве пользователя.

Философия Win 32 API заключается в предоставлении всеобъемлющего интерфейса, с возможностью выполнить одно и то же требование несколькими (тремя-четырьмя) способами. В ОС UNIX все системные вызовы формируют минимальный интерфейс: удаление даже одного из них приведет к снижению функциональности ОС.

Многие вызовы API создают объекты ядра того или иного типа (файлы, процессы, потоки, каналы и т.д.). Каждый вызов, создающий объект, возвращает вызывающему процессу результат, называемый дескриптором (небольшое целое число). Дескриптор используется впоследствии для выполнения операций с объектами. Он не может быть передан другому процессу и использован им. Однако при определенных обстоятельствах дескриптор может быть дублирован и передан другому процессу защищенным способом, что предоставляет второму процессу контролируемый доступ к объекту, принадлежащему первому процессу. С каждым объектом ассоциирован дескриптор безопасности, описывающий, кто и какие действия может, а какие не может выполнять с данным объектом.

Основные функции Win 32 API для файлового ввода-вывода и соответствующие системные вызовы ОС UNIX приведены ниже.

Функции Win 32 API	Системные вызовы UNIX	Описание
`CreateFile`	`open`	Создать или открыть файл; вернуть дескриптор файла
`DeleteFile`	`unlink`	Удалить существующий файл
`CloseHandle`	`close`	Закрыть файл
`ReadFile`	`read`	Прочитать данные из файла
`WriteFile`	`write`	Записать данные в файл
`SetFilePointer`	`lseek`	Установить указатель в файле в определенную позицию
`GetFileAttributes`	`stat`	Вернуть атрибуты файла
`LockFile`	`fcntl`	Заблокировать область файла для обеспечения взаимного исключения

UnlockFile	fcntl	Отменить блокировку области файла

Аналогично файловым операциям обстоит дело с операциями управления каталогами. Основные функции Win 32 API и системные вызовы UNIX для управления каталогами приведены ниже.

Функция Win 32 API	Системные вызовы UNIX	Описание
CreateDirectory	mkdir	Создать новый каталог
RemoveDirectory	rmdir	Удалить пустой каталог
FindFirstFile	opendir	Инициализация, чтобы начать чтение записей каталога
FindNextFile	readdir	Прочитать следующую запись каталога
MoveFile	rename	Переместить файл из одного каталога в другой
SetCurrentDirectory	chdir	Изменить текущий рабочий каталог

Способы выполнения файловых операций

Чаще всего с одним и тем же файлом пользователь выполняет не одну, а последовательность операций. Независимо от набора этих операций операционной системе необходимо выполнить ряд постоянных (универсальных) для всех операций действий.

1. По символьному имени файла найти его характеристики, которые хранятся в файловой системе на диске.
2. Скопировать характеристики в оперативную память, поскольку только в этом случае программный код может их использовать.
3. На основании характеристик файла проверить права пользователя на выполнение запрошенной операции.
4. Очистить область памяти, отведенную под временное хранение характеристик файла.

Кроме того, каждая операция включает ряд уникальных для нее действий, например, чтение определенного набора кластеров диска, удаление файла, изменение его атрибутов и т.п.

ОС может выполнять последовательность действий над файлами двумя способами (см. рис. рис. 7.22).

1. Для каждой операции выполняются как универсальные, так и уникальные действия. Такая схема иногда называется схемой без запоминания состояния операции (stateless).
2. Все универсальные действия выполняются в начале и конце последовательности операций, а для каждой промежуточной операции выполняются только уникальные действия.

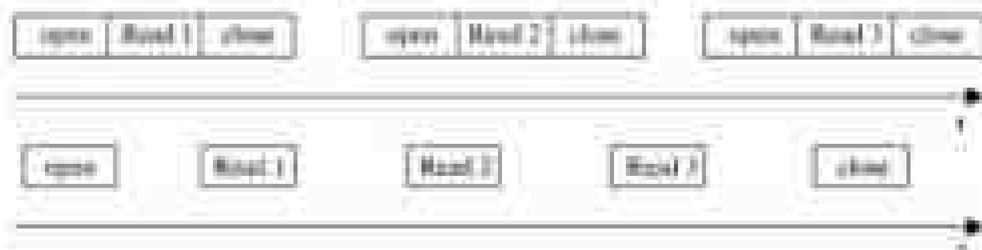


Рис. 7.22. Варианты выполнения последовательности действий над файлами

Подавляющее большинство файловых систем поддерживает второй способ, как более экономичный и быстрый. Однако первый способ более устойчив к сбоям в работе системы, так как каждая операция является самодостаточной и не зависит от результата предыдущей. Поэтому первый способ иногда применяется в распределенных сетевых файловых системах, когда сбои из-за потери пакетов или отказов одного из сетевых узлов более вероятны, чем при локальном доступе к данным.

При втором способе в ФС вводятся два специальных системных вызова: open и close. Первый выполняется перед началом любой последовательности операций с файлом, а второй – после окончания работы с файлом.

Основной задачей вызова open является преобразование символьного имени файла в его уникальное числовое имя, копирование

характеристик файла из дисковой области в буфер оперативной памяти и проверка прав пользователя на выполнение запрошенной операции. Вызов close освобождает буфер с характеристиками файла и делает невозможным продолжение операций с файлами без его повторного открытия.

Приведем несколько примеров системных вызовов для работы с файлами. Системный вызов creat в ОС UNIX работает с двумя аргументами: символьным именем открываемого файла и режимом защиты. Так команда

```
fd = creat ("abc", mode);
```

создает файл abc с режимом защиты, указанным в переменной mode. Биты mode определяют круг пользователей, которые могут получать доступ к файлам, и уровень предоставляемого им доступа. Системный вызов create не только создает новый файл, но также открывает его для записи. Чтобы последующие системные вызовы могли получать доступ к файлу, успешный системный вызов create возвращает небольшое неотрицательное целое число – дескриптор файла – fd. Если системный вызов выполняется с существующим файлом, длина этого файла уменьшается до 0, а все содержимое теряется.

Чтобы прочитать данные из существующего файла или записать в него данные, файл сначала нужно открыть с помощью системного вызова open с двумя аргументами: символьным именем файла и режимом открытия файла (для записи, чтения или того и другого), например:

```
fd = open ("file", how);
```

Системные вызовы creat и open возвращают наименьший неиспользуемый в данный момент дескриптор файла. Когда программа начинает выполнение стандартным образом, файлы с дескрипторами 0, 1 и 2 уже открыты для стандартного ввода, стандартного вывода и стандартного потока сообщений об ошибках.

В стандарте языка Си отсутствуют средства ввода-вывода. Все операции ввода-вывода реализуются с помощью функций, находящихся в библиотеке языка, поставляемой в составе системы программирования Си. На стандартный поток ввода ссылаются через

указатель `stdin`, вывода – `stdout`, сообщений об ошибках – `stderr`. По умолчанию потоку ввода stdin ставится в соответствие клавиатура, а потокам `stdout` и `stderr` – экран дисплея.

Для ввода-вывода данных с помощью стандартных потоков в библиотеке Си определены функции:

- `getchar ( ) /putchar ( )` – ввод-вывод отдельного символа;
- `gets ( ) / puts ( )` – ввод-вывод строки;
- `scanf ( ) / printf ( )` – ввод-вывод в режиме форматирования данных.

Процесс в любое время может организовать ввод данных из стандартного файла ввода, выполнив системный вызов:

read (stdin, buffer, nbytes);

Аналогично организуется вывод в стандартный файл вывода

write (stdout, buffer, nbytes).

При работе в Windows 2000 с помощью функции `CreateFile` можно создать файл и получить дескриптор к нему. Эту же функцию следует применять и для открытия уже существующего файла, так как в Win 32 API нет специальной функции `File Open`. Параметры функций, как правило, многочисленны, например, функция `CreateFile` имеет семь параметров:

1. указатель на имя файла, который нужно создать или открыть;
2. флаги (биты), указывающие, может ли с этим файлом выполняться чтение, запись или то и другое;
3. флаги, указывающие, может ли этот файл одновременно открываться несколькими процессами;
4. указатель на описатель защиты, сообщающий, кто может получать доступ к файлу;
5. флаги, сообщающие, что делать, если файл существует или, наоборот, не существует;
6. флаги, управляющие архивацией, сжатием и т.д.;

7. дескриптор файла, чьи атрибуты должны быть клонированы для нового файла.

Fd = CreateFile ("data", GENERIC_READ, 0, NULL, OPEN_EXISTING, 0,

7.19. Контроль доступа к файлам

Файлы – один из видов разделяемых ресурсов, доступ к которым ОС должна контролировать. Существуют и другие виды ресурсов, с которыми пользователи работают в режиме совместного использования: принтеры, модемы, графопостроители и т.п. Во всех этих случаях пользователи или процессы пытаются выполнить с разделяемым ресурсом определенные операции, а ОС должна решить, имеют ли пользователи на это право. Пользователи являются субъектами доступа, а разделяемые ресурсы – объектами. Пользователь осуществляет доступ к объектам не непосредственно, а с помощью прикладных процессов, которые запускаются от его имени.

Для каждого типа объекта существует набор операций, которые можно с ним выполнить. Система контроля доступа ОС должна предоставлять средства для задания прав пользователей по отношению к объектам дифференцированно по операциям.

В качестве субъектов доступа могут выступать как отдельные пользователи, так и группы пользователей. Объединение пользователей с одинаковыми правами в группу и задание прав доступа в целом для группы является одним из основных приемов администрирования в больших системах.

У каждого объекта доступа существует владелец. Владелец объекта имеет право выполнять с ним любые допустимые для данного объекта операции. Во многих ОС существует особый пользователь – администратор "superuser", который имеет все права по отношению к объектам системы, не обязательно являясь их владельцем. Эти права (полномочия) необходимы администратору для управления политикой доступа.

Различают два основных подхода к определению прав доступа [13].

1. Избирательный доступ – ситуация, когда владелец объекта определяет допустимые операции с объектом. Этот подход называется также произвольным доступом, так как позволяет администратору и владельцам объекта определить права доступа произвольным образом, по их желанию. Однако администратор по умолчанию наделен всеми правами.
2. Мандатный доступ (от mandatory – принудительный) – подход к определению прав доступа, при котором система (администратор) наделяет пользователя или группу определенными правами по отношению к каждому разделяемому ресурсу. В этом случае группы пользователей образуют строгую иерархию, причем каждая группа пользуется всеми правами группы более низкого уровня иерархии.

Мандатные системы доступа считаются более надежными, но менее гибкими. Обычно они применяются в системах с повышенными требованиями к защите информации.

Каждый пользователь (группа) имеет символьное имя, а также уникальный числовой идентификатор. При выполнении процедуры логического входа в систему пользователь сообщает свое символьное имя или пароль. Все идентификационные данные, а также сведения о вхождении пользователя в группы хранятся в специальном файле (UNIX) или базе данных (Windows NT).

Вход пользователя в систему порождает процесс – оболочку, который поддерживает диалог с пользователем и запускает для него другие процессы. Любой порождаемый процесс наследует идентификаторы пользователя и групп от процесса родителя.

В разных ОС для одних и тех же типов ресурсов может быть определен свой список дифференцируемых операций доступа. Для файловых объектов этот список может включать операции, которые рассмотрены выше.

Набор файловых операций может включать всего несколько укрупненных операций, например, для файлов и каталогов: читать, писать и выполнять.

Возможна комбинация двух подходов – детальный уровень и укрупненный. Например, в Windows NT/2000/2003 администратор работает на укрупненном уровне, а при желании может перейти на детальный.

В самом общем случае права доступа могут быть описаны матрицей прав доступа, в которой столбцы соответствуют всем файлам системы, а строки – всем пользователям. На пересечении строк и столбцов указываются разрешенные операции. Однако реально для тысяч и десятков тысяч файлов в системе пользоваться такой матрицей неудобно. Поэтому она хранится по частям, т.е. для каждого файла и каталога создается список управления доступом (Access Control List, ACL), в котором описываются права на выполнение операций пользователей и групп пользователей по отношению к этому файлу или каталогу. Список управления доступом является частью характеристик файла или каталога и хранится на диске в соответствующей области. Не все файловые системы поддерживают списки управления доступом, например, FAT не поддерживает, поскольку разрабатывалась для однопрограммной, однопользовательской ОС MS-DOS.

Обобщенно формат списка управления доступом (ACL) можно представить в виде набора идентификаторов пользователей и групп пользователей, в котором для каждого идентификатора указывается набор разрешенных операций над объектом. Сам список ACL состоит из элементов управления доступом (Access Control Element, ACE), которые соответствуют одному идентификатору. Список ACL с добавлением идентификатора владельца называют характеристиками безопасности.

Рассмотрим организацию контроля доступа в ОС Windows NT/2000/XP. Система управления доступом в этой операционной системе отличается высокой степенью гибкости, которая достигается за счет большого разнообразия субъектов и объектов доступа и детализации операций доступа.

Для разделяемых ресурсов в Windows XP применяется общая модель объекта, которая содержит такие характеристики безопасности, как набор допустимых операций, идентификатор владельца, список управления доступом.

Проверка прав доступа для объектов любого типа выполняется централизованно с помощью монитора безопасности (Security Reference Monitor), работающего в привилегированном режиме.

Для системы безопасности Windows характерно большое количество различных встроенных (предопределенных) субъектов доступа – отдельных пользователей и групп (Administrator, System, Guest, группы Users, Administrators, Account, Operators и др.). Смысл этих встроенных пользователей и групп состоит в том, что они наделены определенными правами. Это облегчает работу администратора по созданию эффективной системы разграничения доступа. Во-первых, за счет того, что нового пользователя можно внести в какую-то группу. Во-вторых, можно добавлять (изымать) права встроенных групп. Наконец, можно создавать новые группы с уникальным набором прав.

Все объекты при создании снабжаются дескрипторами безопасности, содержащими список управления доступом и список пользователей и групп, имеющих доступ к данному объекту. Владелец объекта, обычно пользователь, который его создал, обладает возможностью изменять ACL объекта, чтобы позволить или не позволить другим осуществлять доступ к объекту. Он может выполнить требуемую операцию с объектом, став его владельцем (такая возможность предусмотрена), а затем как владелец получить полный набор разрешений. Однако вернуть владение предыдущему владельцу объекта администратор не может, поэтому пользователь всегда может узнать о том, что с его файлом (принтером и т.п.) работал администратор.

В Windows NT/2000/XP администратор может управлять доступом пользователей к каталогам и файлам только в разделах диска, в которых установлена файловая система NTFS. Разделы FAT не поддерживаются, так как в этой ФС у файлов и каталогов отсутствуют атрибуты для хранения списков управления доступом.

Разрешения в Windows бывают индивидуальные (специальные) и стандартные. Индивидуальные относятся к элементарным операциям над каталогами и файлами, а стандартные разрешения являются объединением нескольких индивидуальных разрешений. На рис. 7.23 и рис. 7.24 приведены шесть стандартных разрешений (элементарных операций), смысл которых отличается для каталогов и файлов.

Рис. 7.23. Стандартные разрешения для каталогов

Рис. 7.24. Стандартные разрешения для файлов

На рис. 7.25 показана возможность установки индивидуальных разрешений для файлов.

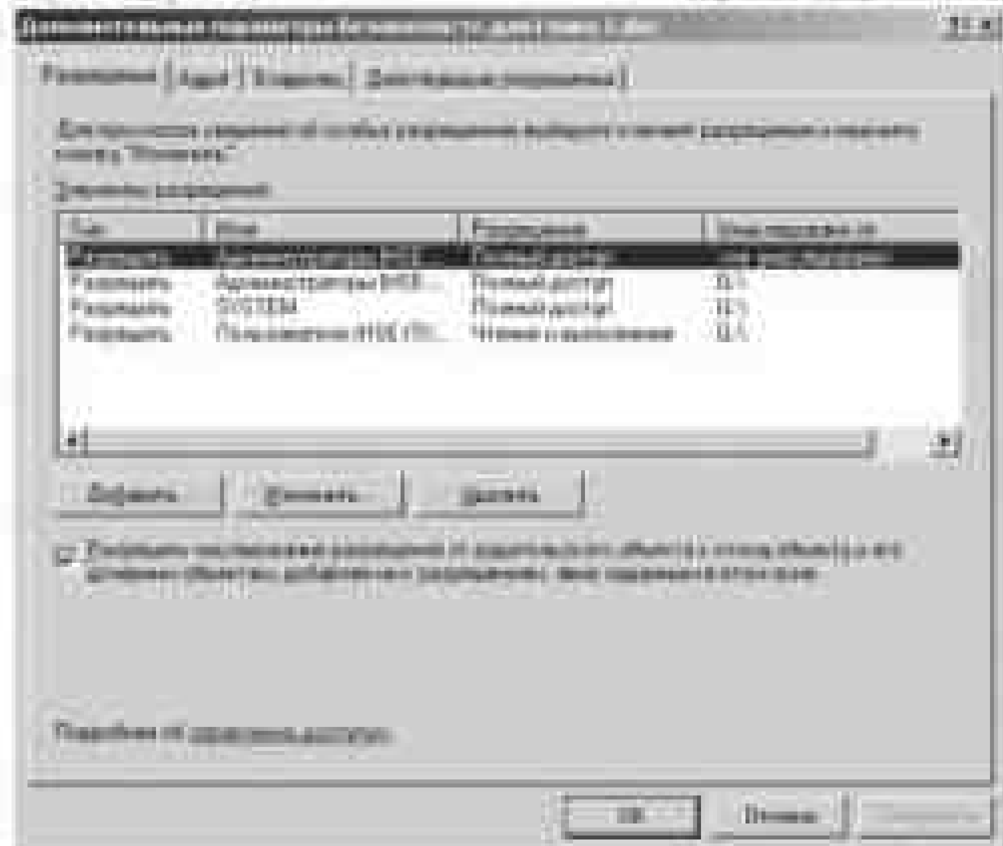


Рис. 7.25. Индивидуальные разрешения для файла

Основные события в истории семейства UNIX/Linux

В этом приложении приведены в хронологическом порядке основные события в истории семейства системы UNIX/Linux.

Представленные факты выбраны из разных книг, ссылки на которые приведены в списке литературы, статей и публикаций в Интернете, но основной материал взят из следующих источников:

- страницы History and Timeline с сайта *Open Group* (ссылка: http://www.unix.org/what_is_unix/history_timeline.html - http://www.unix.org/what_is_unix/history_timeline.html);
- В. Кравчук. Основы операционной системы UNIX. Ее адрес в Интернете ссылка: http://www.citforum.ru/operating_systems/unix/kravchuk/2.shtml - http://www.citforum.ru/operating_systems/unix/kravchuk/2.shtml;
- В. Костромин. Свободная система для свободных людей. Ее адрес в Интернете ссылка: http://linuxcenter.ru/lib/history/lh-01.phtml - http://linuxcenter.ru/lib/history/lh-01.phtml;
- статьи "Хроника событий накануне конфликта между SCO и IBM". Ее адрес в Интернете ссылка: http://naffiman.dyns.com/files/HTM/SCO.htm - http://naffiman.dyns.com/files/HTM/SCO.htm.

Годы	События
1969	Программисты *Bell Labs* вышли из состава разработчиков ОС *Multics*. Система *Multics* была представлена общественности.
1970	С 1 января 1970 года ведется отсчет "Времени UNIX".
1971	В *Bell Labs* выпущена версия UNIX V1 и первое издание "Справочного руководства по системе Unix".
1972	Разработан язык программирования С.
	В *Bell Labs* выпущена версия UNIX V4, переписанная на языке С.

1973	Первое публичное представление UNIX. На четвертом симпозиуме по принципам ОС, проводимом ассоциацией ACM в крупнейшем исследовательском центре IBM (4th ACM Symposium on Operating Systems Principles, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, October 15-17, 1973), был сделан доклад "The UNIX Time-Sharing System" (ссылка http://mfbrain.dyns.com/files/HTM/SCO.htm - http://mfbrain.dyns.com/files/HTM/SCO.htm).
1974	В июле 1974 года Томпсон и Ритчи опубликовали в журнале Communications of the ACM историческую статью "UNIX Timesharing Operating System", которая положила начало новому этапу в истории системы. ОС UNIX заинтересовались в университетах (Учебное пособие С.Д.Кузнецова ссылка: http://www.citforum.ru/operating_systems/unix/contents.shtml - http://www.citforum.ru/operating_systems/unix/contents.shtml).
1975	В Bell Labs выпущена UNIX V6, известная как "Исследовательский UNIX".
1976	Кен Томпсон провел свой академический отпуск в университете г. Беркли и принял участие в проводившихся там исследованиях. (Учебное пособие С.Д.Кузнецова ссылка: http://www.citforum.ru/operating_systems/unix/contents.shtml - http://www.citforum.ru/operating_systems/unix/contents.shtml). В первый раз UNIX была перенесена на другую машину (Interdata 8/32, Ричард Миллер, Австралия).
1977	Появилась BSD 1.0 Калифорнийского университета в Беркли.
1978	Unix "переехала" и на другую 32-разрядную машину – это была VAX 11/780).
1979	Выпуск V7 (последний настоящий UNIX). В состав новой версии системы (V7) входил компилятор нового диалекта языка С *PCC* (Portable C-Compiler), новый командный интерпретатор sh, называемый также в честь своего создателя Bourne-shell (Учебное пособие С.Д.Кузнецова ссылка:

	http://www.citforum.ru/operating_systems/unix/contents.shtml - http://www.citforum.ru/operating_systems/unix/contents.shtml)
1980	В 1980 году при финансовой поддержке Министерства обороны США и по его же инициативе, начаты работы по внедрению стека протоколов TCP/IP. Работы завершились в 1981 году выпуском 4.1BSD (ссылка: http://vap.org.ru/daemons/03.shtml - http://vap.org.ru/daemons/03.shtml). Microsoft приобретает у AT&T лицензию на выпуск собственного диалекта UNIX. Microsoft выпускает собственный диалект UNIX – XENIX OS (ссылка: http://mdforum.dyns.com/files/HTM/SCO.htm - http://mdforum.dyns.com/files/HTM/SCO.htm).
1982	Выпущена System III AT&T UNIX. Выпущена HP-UX (на базе System III) от Hewlett-Packard.
1983	Авторы UNIX К. Томпсон и Д. Ритчи получили премию Тьюринга. Версия 4.2BSD, выпущенная в 1983 году, уже имела поддержку технологии Ethernet и могла интегрироваться в сеть ARPANET (http://vap.org.ru/daemons/03.shtml). Выпущена операционная системаSunOS 1.0 (на базе 4.1BSD) от Sun Microsystems.
1984	В MIT запущен проект X Window System. Выпущена SVR2 AT&T UNIX(первые попытки стандартизации). Выпущены SCO, RS/6000, XENIX – первый коммерческий UNIX на Intel-архитектуре. Начат проект GNU и создан *Free Software Foundation* (*FSF*).

1985	В Университете Карнеги – Меллона в 1985 году было разработано микроядро *Mach*, использованное в NeXT OS (NeXT), MachTen (Mac), OS/2, AIX (для IBM), OSF/1, Digital UNIX (для Alpha), Windows NT и *BeOS* (Энциклопедия компьютеров от KM.RU ссылка: http://www.megakm.ru/pc/encyclop.asp?topic=pc_264&rubr=pc_264 - http://www.megakm.ru/pc/encyclop.asp?topic=pc_264&rubr=pc_264). В IEEE (Institute of *Electrotechnical* and Electronics Engineers) приняты первые стандарты POSIX (Portable operating system interfaces). Появляется ОС *Minix*.
1986	Появилась операционная система AIX от IBM. Появилась операционная система *AUX* от Apple.
1987	Спецификация System V AT&T UNIX была значительно переработана и обогащена дополнительными возможностями. Выходит версия System V Release 3 (SVR3) Появилась операционная система IRIX (SVR3.0) от Silicon Graphics. Другим важным событием стало соглашение AT&T с ведущими UNIX-производителями Sun и Microsoft в 1987 г. о так называемой унификации UNIX. Проект предусматривал создание четвертого издания System V (*SVR4*), которая объединила характеристики Xenix Microsoft (другое название UNIX для микрокомпьютеров, основанной на седьмом издании и испытавшей сильное влияние System V), SunOS (система UNIX фирмы Sun Microsystems, основанной на BSD и System V 3.2) (Учебное пособие С.Д. Кузнецова ссылка: http://www.citforum.ru/operating_systems/unix/contents.shtml - http://www.citforum.ru/operating_systems/unix/contents.shtml).

	AT&T в первый раз лицензировала имя UNIX.
1988	Создание *SVR4 AT&T UNIX* на базе System V, BSD и SunOS. Появление компьютера NeXT с ОС NeXTSTEP (4.3BSD + Mach 2.0).
1989	Ключевым этапом в развитии ветви System V стал 1989 год, год выхода System V Release 4 (SVR4). Важным шагом было решение об объединении возможностей различных UNIX, подобных ОС: BSD, SunOS и System V "под одной крышей" (ссылка: http://vap.org.ru/danmonik/03.shtml - http://vap.org.ru/danmonik/03.shtml). Выпущена *Motif* 1.0.
1990	Выпущен *XPG3*. Появилась *OSF/1* от *Open Software Foundation*. Документ POSIX 1003.1 с редакционными изменениями был принят в качестве стандарта ISO (ссылка: http://www.linuxcenter.ru/lib/history/unix_genove.phtml?style=print&; - http://www.linuxcenter.ru/lib/history/unix_genove.phtml?style=print&;).
1991	Выделение из *AT&T* отдельного подразделения USL (Unix System Laboratories), владеющего кодом *AT&T UNIX System V*. Появление ОС Linux (на базе Minix).
	Novell объявляет о намерении приобрести USL. Появление 4.4BSD.

1992	Закрытие CSRG в Беркли (разработчика последней версий BSD UNIX). Появление UnixWare 1 (реализация SVR4.2).
1993	AT&T продала права на UNIX вместе с лабораторией USL компании Novell. Начало истории развития проекта FreeBSD. Выпуск Linux с версией ядра 0.99.
1994	Novell передает права на товарный знак UNIX международной организации по стандартизации X/Open Company. Группа бывших сотрудников Novell во главе с основателем Novell Реймондом Нурда (Raymond J. Noorda) образуют компанию Caldera Systems International. Одна из ее основных задач – распространение Linux. Начало документированной истории Linux на Руси, когда в журнале "Монитор" была опубликована статья Владимира Водолазкого. Название ее звучало примерно так как легко и без головной боли установить Linux на персональный компьютер.
1995	Появляется спецификация UNIX 95 (Single UNIX Specification) от X/Open. Novell продает UnixWare и весь исходный код AT&T компании SCO. Появляются OpenBSD и NetBSD. У истоков отечественной истории Linux стоят также статьи Петра Врублевского и Виктора Хименко, опубликованные в журнале "Мир ПК" в середине 1995 года и посвященные установке SLS и

	Slackware, соответственно.
1998	Формируется Open Group как слияние компаний OSF и X/Open.
	Появляется спецификация UNIX 98 от The Open Group.
1998	Впервые о Linux'е для конечного пользователя можно говорить, вероятно, с 1998 года, когда Жиль Дюваль создал дистрибутив Mandrake (ныне – Mandriva). Основной его идеей было объединение Linux'а и графической среды KDE (Linux: предыстория и генезис. Алексей Федорчук, ссылка: http://linuxcenter.ru/lib/history/linuxhistory_1.phtml - http://linuxcenter.ru/lib/history/linuxhistory_1.phtml).
1999	Появляется Mac OS X.
2000	Компания SCO продала все свои ОС компании Caldera (Caldera OpenLinux).
	Появляется спецификация Single UNIX Specification Version 3, соединившая IEEE POSIX, The Open Group and промышленный стандарт Linux.
2001	Завершается сделка по покупке SCO со стороны Caldera. Часть компании SCO, ставшей уже бывшей, продолжает развивать продукт Tarantella. Сам бренд SCO, а также SCO OpenServer и база программного кода UNIX System V отходит к Caldera (ссылка: http://rallison.tripod.com/files/HTM/SCO.htm - http://rallison.tripod.com/files/HTM/SCO.htm).
2002	Caldera объявляет об изменении названия на SCO Group (ссылка: http://rallison.tripod.com/files/HTM/SCO.htm - http://rallison.tripod.com/files/HTM/SCO.htm).
2003	Появляется спецификация UNIX03 POSIX The Open Group
2004	Начало проекта Ubuntu

2007 Выпуск iPhone OS X

Первенство технологических достижений двух основных версий UNIX

Это приложение демонстрирует в хронологическом порядке свойства версий двух основных направлений операционных систем семейства UNIX.

Как на один из примеров взаимного проникновения идей в разные версии, обратите внимание на строку SVR4, в которой появилась быстрая файловая система, заимствованная у BSD. И конечно, таблица поражает количеством новых идей, реализуемых в версиях по мере их совершенствования. Основа таблицы взята из пособия С. Кузнецова, опубликованного в Интернете (ссылка: http://www.citforum.ru/operating_systems/unix/contents.shtml - http://www.citforum.ru/operating_systems/unix/contents.shtml).

Характерные свойства версий AT&T UNIX начиная с 1982 года.

Года, версия системы	Характерные свойства
1982 System III	Именованные программные каналы
	Очереди запуска
1983 System V	Хеш-таблицы
	Кэши буферов и inode
	Семафоры
	Разделяемая память
	Очереди сообщений
1984 SVR2	Блокирование записей и файлов
	Подкачка по требованию
	Копирование по записи (write on copy)
1987 SVR3	Межпроцессные взаимодействия (IPC)
	Разделение удаленных файлов (RFS)
	Развитые операции обработки сигналов
	Разделяемые библиотеки
	Переключатель файловых систем (FSS)

	Интерфейс транспортного уровня (TLI)
	Возможности коммуникаций на основе потоков
1989 SVR4	Поддержка обработки в реальном времени
	Классы планирования процессов
	Динамически выделяемые структуры данных
	Развитые возможности открытия файлов
	Управление виртуальной памятью (VM)
	Возможности виртуальной файловой системы (VFS)
	Быстрая файловая система (BSD)
	Развитые возможности потоков
	Прерываемое ядро
	Квоты файловых систем
	Интерфейс драйвера с ядром системы
1989	Прекращение выпуска версий UNIX AT&T

Приведем описание новшеств, появлявшихся в разных версиях операционной системы BSD и служащей характеристикой ее достижений.

Характерные свойства версий BSD

Годы, версия системы	Характерные свойства
1975 (1978) BSD	Текстовый редактор ex, впоследствии названный vi
	Командный процессор C
1983 3BSD	Виртуальная память
	Страничное замещение по требованию
1983 4.2BSD	Поддержка семейства сетевых протоколов TCP/IP
	Поддержка работы в сетях, в т.ч. Ethernet
	Эта версия обеспечила подключение к сети ARPANET
	Файловая система fast.
1986-1990	

4.3BSD	
	Виртуальная файловая система VFS
	Отладчик ядра
	Более мощная поддержка сети
1993 4.4BSD	Виртуальная память как в Mach 2.5
	Журналируемая файловая система LSF
1992	Закрытие CSRG в Беркли (подразделения, разрабатывающего последние версии BSD UNIX)

Список литературы

1. Беляков М.И., Рабовер Ю.И., Фридман А.Л. Мобильная операционная система, М., «Радио и связь», 1991, 208 с

2. Бурк Робин, Хорват Б. Дэвид и др. UNIX для системных администраторов. Энциклопедия пользователя, К. Издательство «ДиаСофт». 1998-864 с

3. Галатенко В.А. Программирование в стандарте POSIX. Курс лекций. Часть первая. 2004 М.Ж, ИНТУИТ.РУ (Интернет Университет Информационных Технологий). 560 стр

4. Дейтел Г. Введение в операционные системы М.:Мир, 1987. – том 2 – 400 с

5. Карпов В.Е., Коньков К.А. Основы операционных систем. Интернет-университет Информационных технологий. Москва, 2004, 632 с

6. Костромин В.А. Самоучитель Linux для пользователя. СПб.: БХВ-Петербург, 2002.-672 с.: с ил. (Издательский дом «Питер»)

7. Коллинз Билл, [illegible] Сертификация системного администратора. К.: «ТИД ДС», 2003. – 528 стр

8. Леви Д. Linux для начинающих. М. Бестселлер, 2003.- 256 стр

9. Мэдник С., Донован Дж. Операционные системы. М.: Мир, 1978. – 792 с

10. Назаров С.В. Операционные среды, системы и оболочки. Основы структурной и функциональной организации. М.:КУДИЦ-ПРЕСС, 2007.- 504 с

11. Назаров С.В., Барсуков А.Г. Генерация операционной системы ОС ЕС. М.: Финансы и статистика, 1985. – 175 с

12. Назаров С.В., Гудыно Л.П., Кириченко А.А. Операционные системы. Практикум. М.:КУДИЦ-ПРЕСС, 2008. — 464 с

13. Олифер В.Г.,Олифер Н.А. Сетевые операционные системы. СПб.Питер, 2003.-544 с.:ил

14. Робачевский А.М. Операционная система UNIX. СПб. BHV, 1999. 544 с

15. Стоффер Д. Бизнес путь. Секреты мега-бренда Новой экономики. СПб.: Издательство Крылов, 2003. – 352 с

16. Таккет Дж. Использование Linux. (Специальное издание) К.; М.; СПб Издательский дом «Вильямс», 1998. – 576 с.: ил

17. Таненбаум Э. Современные операционные системы. 2-е изд. СПб.: Питер, 2006. – 1040 с

18. Торвальдс Л., Даймонд Д. Ради удовольствия (рассказ нечаянного революционера). М.: Изд-во ЭКСМО, 2002. – 288 с

19. Хейвер Крис и др. Внутренний мир UNIX. К.: Издательство «ДиаСофт», 1998. – 432 с

20. Руслан Богатырев. [illegible]. URL: http://www.osp.ru/os/2001/12/180729_p12.html . Открытые системы . – Электронный журнал 2001 – 28 декабря

21. Интернет энциклопедия «Википедия». URL: http://ru.wikipedia.org/wiki/Томпсон,_Кен . Статья Кен Томпсон

22. Ольга Чуприкова. Известные программисты. URL: http://[illegible] . Страничка Кена Томпсона. Студенческий сайт Вологодского государственного педагогического университета, факультет прикладной математики и компьютерных технологий. стр 2009

23. Интервью с Кеном Томпсоном. URL: http://www.osp.ru/os/1999/04/179809 . Открытые системы. Электр. журнал 1999 – 17 апр

24. *Персональная страничка Кена Томпсона*, URL: http://www.cs.bell-labs.com/who/ken/ , [illegible], 2001, August 21

25. *Интернет энциклопедия «Википедия»*, URL: http://ru.wikipedia.org/wiki/Деннис_Ритчи , Статья Деннис Ритчи

26. *Персональная страничка Денниса Ритчи*, URL: http://www.cs.bell-labs.com/who/dmr/ , 2006, May

27. Ольга Чупрякова, *Известные программисты*, URL: http://students.uni-vologda.ac.ru/pages/[illegible]/programers/index.html , Страничка Денниса Ритчи Студенческий сайт Вологодского государственного педагогического университета, факультет прикладной математики и компьютерных технологий, год 2009

28. Климов Ю.В. *Дистанционное обучение программистов. Программирование на C. Урок 1. Краткая история развития языка C*. URL: http://cpp.lem.ru/lessons/c/les01.html

29. Vladimir A. Popov. *Весьма полные и краткие сведения о операционной системе Free BSD. Часть 1*. URL: http://vap.org.ru/[illegible].html , cop. 200[illegible]-2010

30. *Department of Linguistics and Oriental Languages of San Diego State University*, URL: http://www.rohan.sdsu.edu/~gawron/compling/unix.html , Unix Cop 2008

31. *Интернет энциклопедия «Википедия»*, Статья Unix, URL: http://ru.wikipedia.org/wiki/Unix

32. С. Д. Кузнецов, Основы операционной системы UNIX, URL: http://www.citforum.ru/operating_systems/unix/contents.shtml , Cop. 1997-2000 CIT, 2001-2009 CIT Forum

33. *Интернет энциклопедия «Википедия»*, URL: http://ru.wikipedia.org/wiki/Bell_Labs , Статья Bell_Labs

34. Крис Касперски, «BSD в ракурсе исторического процесса», URL: http://www.insidepro.com/kk/117/117r.shtml , cop. 2003-2010 InsidePro Software

35. Сергей Шамарин, [illegible], URL: http://[illegible] , Электронный журнал Компьютеры и программы. 23.2002

36. Sun открыла большую часть исходного кода Java, URL: http://www.cybersecurity.ru/news/24105.html , CyberSecurity / Новости / 2007 – 11 мая

37. The International Business Machine (IBM), URL: http://www.ibm.com/developerworks/ru/linux

38. *The Silicon Graphics, Inc* (SGI), URL: http://www.sgi.com/products/software/irix/

39. *The Linux Org*, URL: http://www.linux.org/hardware , cop. 1994-2008

40. Алексей Федорчук, *Linux: предыстория и генезис*, URL: http://linuxcenter.ru/lib/history/linuxhistory_1.phtml , The linuxcenter (ЛинуксЦентр), 2006 – 10 февраля

41. Алексей Федорчук, *Начало дистрибуции*, URL: http://linuxcenter.ru/lib/history/linuxhistory_3.phtml , The linuxcenter (ЛинуксЦентр), 2006 – 10 февраля

42. *The Linux Org* , URL: http://www.linux.org/dist , cop. 1994-2008

43. *The distrowatch*, URL: http://www.distrowatch.com , cop. 2001-2010

44. Operating System Interface Designs, URL: http://www.[illegible].com/2009/01/12/operating-system-interface-designs , Posted by Ludwig in Miscellaneous on 01 12th, 2009

45. Керниган, Ритчи, Язык C, URL: http://[illegible]-2002-[illegible]

kernigan-rich/p0.html , ITBooks , copy. 2008 ITBooks

46. Martin Chalant, Путеводитель по стандартам, URL: http://[illegible].shtml , The OpenNET (форум OpenNET). Created 1996-2010

47. Сергей Кузнецов, Открытые системы, процессы стандартизации и профили стандартов, URL: http://citforum.ru/database/articles/art_19.shtml , Copy. 1997-2008 CIT, 2001-2008 CIT Forum

48. Эрик С. Реймонд, Искусство программирования для UNIX, URL: http://itbooks.ru/prog/unixprog/[illegible].html , ITBooks , copy. 2008 ITBooks

49. В.А. Галатенко, Программирование в стандарте POSIX, URL: http://www.intuit.ru/department/se/pposix/ , ИНТУИТ 2006 – 15 июня

50. Алексей Федорчук, Введение в POSIX'ивизм. Часть 2. Вопросы истории POSIX'ивизма, URL: http://www.linuxcenter.ru/lib/books/posixbook/ch03.phtml , The Linuxcenter (ЛинуксЦентр). 2005 – 8 июня

51. Сергей Романюк, Стандарты POSIX, URL: http://citforum.ru/operating_systems/articles/posix.shtml , Открытые системы , Электр. журнал 1998 9 октября

52. Сухомлинов В.А, ВВЕДЕНИЕ В АНАЛИЗ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ Часть V Методология и система стандартов POSIX OSE, URL: http://old.[illegible]/lectures/AnalizIT/

53. Интернет энциклопедия «Википедия», URL: http://ru.wikipedia.org/wiki/Richard_Stallman , Статья Ричард Столлман

54. В.А. Костромин, Свободные системы для свободных людей, URL: http://linux.[illegible].phtml?part= , The Linuxcenter (ЛинуксЦентр). 2005 март

55. The Open Source, URL: http://www.opensource.org

56. Интернет энциклопедия «Википедия», URL: http://ru.wikipedia.org/wiki/OpenSource , Статья Open Source

57. Эрик С. Реймонд, Собор и Базар, URL: http://lib.ru/LINUXGUIDE/[illegible]-pictures.html , Lib.Ru: Библиотека Максима Мошкова

58. Николай Безруков, Разработка программ с открытыми исходниками как особый вид исследовательской деятельности, URL: http://[illegible].ru/articles/16 , 2000 – 17 февраля

59. Николай Безруков, Повторный взгляд на Собор и Базар, URL: http://[illegible].ru/articles/18 , 2000 – 17 февраля

60. Россия и свободное программное обеспечение, URL: http://[illegible]?ID=[illegible] , AlgoNet, Новости, / Сентябрь 2005

61. Борис Викторов, Кто в России пишет под Windows, URL: http://www.[illegible]/2003/01/25/[illegible].html , 2003 25 января

62. Словари Yandex, URL: http://slovari.yandex.ru/[illegible]%20[illegible]/Интерфейс

63. FreeDownload, URL: http://freedownloads.[illegible] , copy. 2005-2006

64. Смирнова Олег, Билл Гейтс и Стив Джобс: общаются в интервью, URL: http://deepapple.ru/articles/[illegible].html , DeepApple 2007 [illegible] июня

65. *The X Org*, URL: http://www.x.org

66. Андрей Зубинский. Windows. *Восточные проблемы*. Часть 1, URL: http://itc.ua/article.phtml?ID=8934 , Издательский Дом ITC, 2002, 6 февраля

67. Сергей Голубев, Трёхмерные [illegible] рабочего стола, URL: http://www.pcmag.ru/solutions/detail.php?ID=8660, PC Magazine , 2005, 23 мая

68. The GNOME, URL: http://www.gnome.org.ru

Содержание

Титульная страница	2
Выходные данные	3
Лекция 1. Архитектура, назначение и функции операционных систем	4
Лекция 2. Основные семейства операционных систем	66
Лекция 3. Стандарты и лицензии на программное обеспечение	102
Лекция 4. Интерфейсы операционных систем	116
Лекция 5. Организация вычислительного процесса	143
Лекция 6. Управление памятью. Методы, алгоритмы и средства	215
Лекция 7. Подсистема ввода-вывода. Файловые системы	264
Лекция 8. Основные события в истории семейства UNIX/Linux	337
Лекция 9. Первенство технологических достижений двух основных версий UNIX	345
Список литературы	348