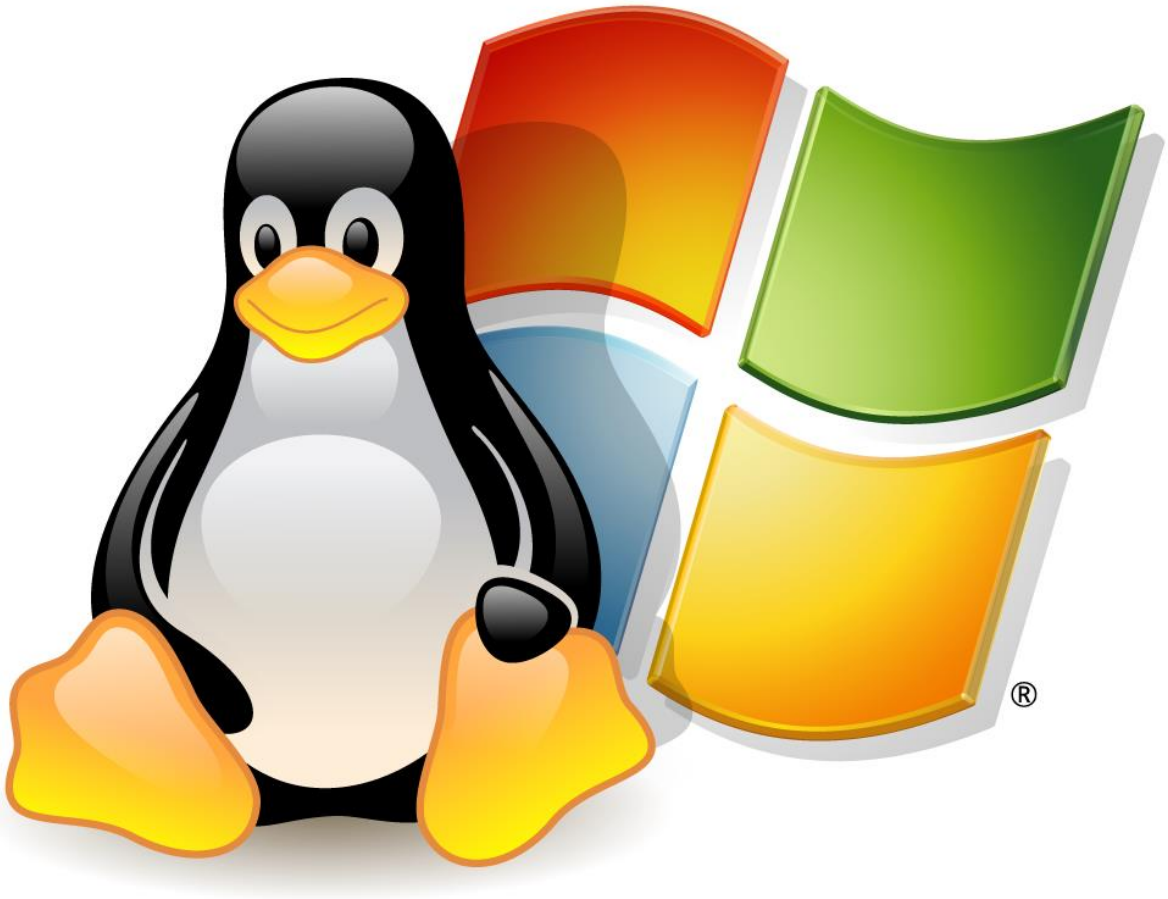
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Д.О. Пахмурин

ОПЕРАЦИОННЫЕ СИСТЕМЫ ЭВМ

Учебное пособие



Томск – 2013

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ СИСТЕМ УПРАВЛЕНИЯ И РАДИОЭЛЕКТРОНИКИ (ТУСУР)

Кафедра промышленной электроники

Д.О. Пахмурин

ОПЕРАЦИОННЫЕ СИСТЕМЫ ЭВМ

Учебное пособие
для студентов очной формы
обучения по направлению
210100.62 – Электроника и наноэлектроника (профиль "Промышленная электроника")

2013

Пахмурин Д.О.
Операционные системы ЭВМ: Учебное пособие. – Томск: Томский государственный университет систем управления и радиоэлектроники, 2013. – 254 с.: ил.

Рассмотрены основные понятия операционных систем, изложены история развития, принципы построения и функционирования UNIX и Windows. В учебном пособии уделено внимание таким аспектам, как межпроцессное взаимодействие, взаимоблокировки, управление памятью, ввод-вывод, файловые системы и информационная безопасность.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Подготовка современного инженера не мыслима без получения им соответствующих знаний в сфере компьютерных технологий. В современном мире без достаточно полноценного обучения работе с информацией, с принципами функционирования компьютерного оборудования не возможна качественная разработка каких-либо серьезных технических проектов. Освоение курса "Операционные системы ЭВМ" позволяет иметь более структурированный подход, в частности, к программированию микропроцессоров, что является очень востребованным направлением в настоящее время.

Целью изучения курса "Операционные системы ЭВМ" является формирование знаний по основным принципам организации операционных систем персональных ЭВМ и подготовка к эффективному практическому применению вычислительных систем.

Данный курс базируется на курсах "Информатика", "Математическое моделирование и программирование", "Цифровая и микропроцессорная техника" и "Компьютерные сети".

Курс имеет следующую структуру:

1. Основные понятия.
2. Процессы и потоки.
3. Межпроцессное взаимодействие.
4. Взаимоблокировки.
5. Управление памятью.
6. Ввод и вывод.
7. Файловые системы.
8. Мультимедийные операционные системы.
9. Многопроцессорные системы.
10. Безопасность.
11. Операционные системы семейства UNIX.
12. Операционные системы Windows семейства NT.

1. ОСНОВНЫЕ ПОНЯТИЯ

1.1. Понятие "операционная система"

Современный компьютер состоит из одного или нескольких процессоров, оперативной памяти, дисков, клавиатуры, монитора, принтеров, сетевых интерфейсов и других устройств ввода-вывода, то есть является сложной системой. Написание программ, которые отслеживают все компоненты, корректно используют их и при этом оптимально работают, представляет собой крайне трудную задачу. Еще много лет назад стало очевидно, что нужно как-то оградить программистов от тонкостей, связанных с аппаратным обеспечением. Постепенно был выработан следующий путь: поверх аппаратуры работает дополнительная программная прослойка, которая управляет всем оборудованием и предоставляет пользователю интерфейс, или виртуальную машину, более простую для понимания и программирования, чем аппаратура. Операционная система (ОС) и является этой программной прослойкой.

Место операционной системы в общей структуре компьютера показано на рисунке 1.1. Внизу находится аппаратное обеспечение, которое во многих случаях само состоит из двух или более уровней (или слоев). Самый нижний уровень содержит физические устройства, состоящие из интегральных микросхем, проводников, источников питания и т. п.

Рисунок 1.1 – Место операционной системы в структуре компьютера

Далее следует микроархитектурный уровень, на котором физические устройства группируются в функциональные блоки. Как правило, на микроархитектурном уровне находятся внутренние регистры центрального процессора (ЦП) и тракт данных, включающий арифметико-логическое устройство. На каждом такте процессора данные извлекаются из регистров и обрабатываются

арифметико-логическим устройством (к примеру, участвуют в операции арифметического или логического суммирования). Результат сохраняется в одном или нескольких регистрах. В некоторых компьютерах функционирование тракта данных находится под управлением особой программы, называемой микропрограммой. В остальных случаях управление обеспечивают аппаратные схемы. Тракт данных предназначен для выполнения наборов команд. Некоторые наборы могут быть выполнены в одном цикле такта данных, другие же требуют нескольких тактов. В распоряжении команд находятся различные аппаратные средства, в том числе регистры. Аппаратное обеспечение и команды, доступные программисту на языке ассемблера, образуют архитектуру набора команд (Instruction Set Architecture, ISA). Зачастую данный уровень называют машинным языком.

Обычно машинный язык содержит от 50 до 300 команд, служащих преимущественно для перемещения данных в пределах компьютера, выполнения арифметических операций и сравнения величин. Управление устройствами на этом уровне осуществляется путем загрузки определенных величин в специальные регистры устройств. Например, диску можно дать команду чтения, записав в его регистры адрес места на диске, адрес в основной памяти, число байтов для чтения и направление действия (чтение или запись). На практике нужно передавать больше параметров, а информация о статусе операции, возвращаемая диском, достаточно сложна. Кроме того, при программировании многих устройств ввода-вывода (Input/Output, I/O) очень важную роль играют временные соотношения.

Основное предназначение операционной системы – скрыть все эти сложности и предоставить программисту более удобную систему команд. Чтение блока из файла в этом случае представляется намного более простым действием, чем в случае, когда программисту приходится думать о перемещении головок диска, о задержках, связанных с их установкой в нужное место и т. д.

Поверх операционной системы на рисунке расположены остальные системные программы. Здесь находятся интерпретатор команд (оболочка), компиляторы, редакторы и т. д. Подобные программы не являются частью операционной системы. Под операционной системой обычно понимается то программное обеспечение, которое запускается в режиме ядра или, как его еще называют, режиме супервизора, привилегированном режиме. Операционная система

защищена от вмешательства пользователя с помощью аппаратных средств. Компиляторы и редакторы запускаются в пользовательском режиме.

Подобная классификация имеет весьма размытые границы во встраиваемых системах, допускающих отсутствие ядра, и в интерпретируемых системах. Тем не менее, в традиционных компьютерах операционная система является элементом, исполняемым в режиме ядра.

Поверх системных программ выполняются прикладные программы. Граница между системным и прикладным программным обеспечением размыта. Очевидно, что компоненты, исполняемые в режиме ядра, относятся к операционной системе, однако многие "пользовательские" программы также осуществляют системные функции либо, как минимум, тесно связаны с ними.

Таким образом, **операционная система компьютера** представляет собой комплекс взаимосвязанных программ, который действует как интерфейс между приложениями и пользователями с одной стороны, и аппаратурой компьютера с другой стороны.

1.2. История операционных систем

Понимание того, что представляет собой операционная система и каковы ее функции, пришло не сразу. В эволюции операционных систем можно выделить несколько периодов.

Первый период (1945-1955). В середине 40-х годов XX века были созданы первые ламповые вычислительные устройства. В то время одна и та же группа людей участвовала и в проектировании, и в эксплуатации, и в программировании вычислительной машины. Это была скорее научно-исследовательская работа в области вычислительной техники, а не использование компьютеров в качестве инструмента решения практических задач из других прикладных областей. Программирование осуществлялось исключительно на машинном языке. Об операционных системах не было и речи, все задачи организации вычислительного процесса решались вручную каждым программистом с пульта управления. Не было никакого другого системного программного обеспечения, кроме библиотек математических и служебных подпрограмм.

Второй период (1955-1965). С середины 50-х годов начался новый период в развитии вычислительной техники, связанный с появлением новой технической базы – полупроводниковых элементов. В эти годы появились первые

алгоритмические языки, а, следовательно, и первые системные программы – компиляторы. Высокая стоимость процессорного времени требовала уменьшения затрат времени между запусками программы. Появились первые системы пакетной обработки, которые просто автоматизировали запуск одной программы за другой и, тем самым, увеличивали коэффициент загрузки процессора. Системы пакетной обработки явились прообразом современных операционных систем, они стали первыми системными программами, предназначенными для управления вычислительным процессом.

Третий период (1965-1980). В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам. Для этого периода также характерно создание семейства программно-совместимых машин. Программная совместимость требовала и совместимости операционных систем. Такие операционные системы должны были бы работать и на больших, и на малых вычислительных системах, с большим и с малым количеством разнообразной периферии, в коммерческих областях и в области научных исследований.

Важнейшим достижением ОС данного поколения явилась реализация мультипрограммирования. *Мультипрограммирование* – это способ организации вычислительного процесса, в соответствии с которым на одном процессоре попеременно выполняются несколько программ. Другое нововведение – *спулинг* (spooling), который определялся в то время как способ организации вычислительного процесса, в соответствии с которым задания считывались с перфокарт на диск в том темпе, в котором они появлялись в помещении вычислительного центра, а затем, когда очередное задание завершалось, новое задание с диска загружалось в освободившийся раздел.

Наряду с мультипрограммной реализацией систем пакетной обработки появился новый вариант ОС – *системы разделения времени*. Этот вариант рассчитан на *многотерминальные системы*, когда каждый пользователь работает за своим терминалом. В мультипрограммных системах пакетной обработки пользователь был лишен возможности интерактивно взаимодействовать со своими программами. Вариант мультипрограммирования, применяемый в системах разделения времени, нацелен на создание для каждого отдельного пользователя иллюзии единоличного использования вычислительной машины за счет периодического выделения каждой программе своей доли процессорного времени. В системах разделения времени

эффективность использования оборудования ниже, чем в системах пакетной обработки, что явилось платой за удобства работы пользователя.

Многотерминальный режим использовался не только в системах разделения времени, но и в системах пакетной обработки. При этом не только оператор вычислительной машины, но и пользователи получали возможность формировать свои задания и управлять их выполнением со своего терминала. Такие операционные системы получили название *систем удаленного ввода заданий*.

В этот период произошло существенное изменение в распределении функций между аппаратными и программными средствами компьютера. Операционные системы стали неотъемлемым элементом компьютеров, играя роль "продолжения" аппаратуры.

Реализация мультипрограммирования потребовала внесения очень важных изменений в аппаратуру компьютера, непосредственно направленных на поддержку этого способа организации вычислительного процесса. При разделении ресурсов компьютера между программами необходимо обеспечить более быстрое переключение процессора с одной программы на другую, а также надежно защитить коды и данные одной программы от непреднамеренной и намеренной порчи другой программой. В процессорах появился привилегированный и пользовательский режимы работы, специальные регистры для быстрого переключения с одной программы на другую, средства защиты областей памяти, развитая система прерываний.

В привилегированном режиме, предназначенном для работы модулей операционной системы, процессор может выполнять все команды, в том числе и те из них, которые позволяют осуществлять распределение и защиту ресурсов компьютера. Программам, работающим в пользовательском режиме, некоторые команды процессора недоступны. Таким образом, только ОС может управлять аппаратными средствами и играть роль монитора и арбитра для пользовательских программ.

Система прерываний позволила синхронизировать работу различных устройств компьютера, работающих параллельно и асинхронно, таких как каналы ввода-вывода, диски, принтеры и т.п. *Аппаратная поддержка операционных систем* стала с тех пор неотъемлемым свойством практически любых компьютерных систем.

Четвертый период (1980-е годы). Этот период в эволюции операционных систем связан с появлением больших интегральных схем. К наиболее важным событиям этого десятилетия можно отнести разработку стека протоколов TCP/IP, становление Интернета, стандартизацию технологий локальных сетей, появление персональных компьютеров и операционных систем для них. Компьютеры стали широко применяться неспециалистами, что потребовало разработки "дружественного" программного обеспечения. Предоставление этих "дружественных" функций стало прямой обязанностью операционных систем. Стали бурно развиваться сети персональных компьютеров, работающих под управлением сетевых или распределенных операционных систем. В результате поддержка сетевых функций стала для ОС персональных компьютеров необходимым условием.

Современный период. В 90-е годы практически все операционные системы стали сетевыми. Сетевые функции сегодня встраиваются в ядро ОС, являясь ее неотъемлемой частью. Операционные системы получили средства работы со всеми основными технологиями локальных и глобальных сетей, а также для создания составных сетей.

Во второй половине 90-х годов все производители операционных систем резко усилили поддержку средств работы с Интернетом. В комплект поставки начали включать утилиты, реализующие популярные сервисы Интернета. Влияние Интернета проявилось и в том, что компьютер превратился из чисто вычислительного устройства в средство коммуникации с развитыми вычислительными и мультимедийными возможностями.

Особое внимание в настоящее время уделяется *корпоративным сетевым операционным системам*. Корпоративная операционная система отличается способностью хорошо и устойчиво работать в крупных сетях, которые характерны для большинства предприятий, имеющих отделения в разных городах и странах. Таким сетям присуща высокая степень гетерогенности программных и аппаратных средств. Поэтому корпоративная ОС должна беспроблемно взаимодействовать с операционными системами разных типов и работать на различных аппаратных платформах. Для корпоративной ОС также важно наличие средств централизованного администрирования и управления, позволяющих в единой базе данных хранить учетные записи о десятках тысячах пользователей, компьютеров, коммуникационных устройств и модулей программного обеспечения.

На современном этапе развития операционных систем на передний план вышли средства обеспечения *безопасности*. Это связано с возросшей ценностью информации, обрабатываемой компьютерами, а также с повышенным уровнем угроз, существующих при передаче данных по сетям. Многие ОС обладают сегодня развитыми средствами защиты информации, основанными на шифрации данных, аутентификации и авторизации.

Современным операционным системам присуща *многоплатформенность*, то есть способность работать на различных типах компьютеров. Многие ОС имеют специальные версии для поддержки кластерных архитектур, обеспечивающих высокую производительность и отказоустойчивость.

В последнее время получила дальнейшее развитие долговременная тенденция повышения *удобства работы человека с компьютером*. Эффективность работы человека становится основным фактором, определяющим эффективность вычислительной системы в целом.

1.3. Классификация операционных систем

Существует множество различных подходов к классификации операционных систем.

К основным признакам классификации ОС относятся:

- **Особенности алгоритмов управления ресурсами.** В зависимости от особенностей использования алгоритма управления процессором, операционные системы делят на:
  - *многозадачные и однозадачные*.

По числу одновременно выполняемых задач операционные системы могут быть разделены на два класса: однозадачные (например, MS-DOS, MSX) и многозадачные (ОС ЕС, OS/2, UNIX, Windows). *Однозадачные ОС* в основном выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным процесс взаимодействия пользователя с компьютером. Однозадачные ОС включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем. *Многозадачные ОС*, кроме вышеперечисленных функций, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

Важнейшим разделяемым ресурсом является процессорное время. Способ распределения процессорного времени между несколькими одновременно существующими в системе процессами (или нитями) во многом определяет специфику ОС. Среди множества существующих вариантов реализации многозадачности можно выделить две группы алгоритмов: *невытесняющая многозадачность* (NetWare, Windows 3.x) и *вытесняющая многозадачность* (Windows семейства NT, OS/2, UNIX). Основным различием между вытесняющим и невытесняющим вариантами многозадачности является степень централизации механизма планирования процессов.

В первом случае механизм планирования процессов целиком сосредоточен в операционной системе, а во втором – распределен между системой и прикладными программами. При невытесняющей многозадачности активный процесс выполняется до тех пор, пока он сам, по собственной инициативе, не отдаст управление операционной системе для того, чтобы та выбрала из очереди другой готовый к выполнению процесс. При вытесняющей многозадачности решение о переключении процессора с одного процесса на другой принимается операционной системой, а не самим активным процессом.

– *Многопользовательские и однопользовательские;*

По числу одновременно работающих пользователей ОС делятся на: однопользовательские (MS-DOS, Windows 3.x, ранние версии OS/2); многопользовательские (UNIX, Windows семейства NT). Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей.

Следует заметить, что не всякая многозадачная система является многопользовательской, и не всякая однопользовательская ОС является однозадачной.

– *Поддерживающие многопоточную обработку и не поддерживающие ее;*

Важным свойством операционных систем является возможность распараллеливания вычислений в рамках одной задачи. Многонитевая (многопоточная) ОС разделяет процессорное время не между задачами, а между их отдельными ветвями (нитями, потоками).

– *Многопроцессорные и однопроцессорные системы.*

Другим важным свойством ОС является отсутствие или наличие в ней средств поддержки многопроцессорной обработки – *мультипроцессирование*. Мультипроцессирование приводит к усложнению всех алгоритмов управления ресурсами. В наши дни становится общепринятым введение в ОС функций поддержки многопроцессорной обработки данных.

Такие функции появились в операционных системах Solaris 2.x фирмы Sun, Open Server 3.x компании Santa Crus Operations, OS/2 фирмы IBM, Windows NT фирмы Microsoft и NetWare 4.1 фирмы Novell. Многопроцессорные ОС могут классифицироваться по способу организации вычислительного процесса в системе с многопроцессорной архитектурой: *асимметричные ОС и симметричные ОС*. Асимметричная ОС целиком выполняется только на одном из процессоров системы, распределяя прикладные задачи по остальным процессорам. Симметричная ОС полностью децентрализована и использует весь пул процессоров, разделяя их между системными и прикладными задачами.

- **Особенности аппаратных платформ.** По типу аппаратуры различают операционные системы:
  - *персональных компьютеров;*
  - *миникомпьютеров;*
  - *мейнфреймов;*
  - *кластеров;*
  - *сетей ЭВМ.*

Среди перечисленных типов компьютеров могут встречаться как однопроцессорные варианты, так и многопроцессорные. В любом случае специфика аппаратных средств, как правило, отражается на специфике операционных систем. Очевидно, что ОС большой машины является более сложной и функциональной, чем ОС персонального компьютера. Так в ОС больших машин функции по планированию потока выполняемых задач, очевидно, реализуются путем использования сложных приоритетных дисциплин и требуют большей вычислительной мощности, чем в ОС персональных компьютеров. Аналогично обстоит дело и с другими функциями.

Мейнфрейм – высокопроизводительный компьютер со значительным объёмом оперативной и внешней памяти, предназначенный для организации централизованных хранилищ данных большой ёмкости и выполнения интенсивных вычислительных работ.

Другие требования предъявляются к операционным системам кластеров. *Кластер* – слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений и представляющихся пользователю единой системой. Наряду со специальной аппаратурой для функционирования кластерных систем необходима и программная поддержка со стороны операционной системы, которая сводится в основном к синхронизации доступа к разделяемым ресурсам, обнаружению отказов и динамической реконфигурации системы. Одной из первых разработок в области кластерных технологий были решения компании Digital Equipment на базе компьютеров VAX. Этой компанией было заключено соглашение с корпорацией Microsoft о разработке кластерной технологии, использующейся в Windows NT.

Несколько компаний предлагают кластеры на основе UNIX-машин. Наряду с ОС, ориентированными на совершенно определенный тип аппаратной платформы, существуют операционные системы, специально разработанные таким образом, чтобы они могли быть легко перенесены с компьютера одного типа на компьютер другого типа, так называемые мобильные ОС. Наиболее ярким примером такой ОС является популярная система UNIX. В этих системах аппаратно-зависимые места тщательно локализованы, так что при переносе системы на новую платформу переписываются только они. Средством, облегчающим перенос остальной части ОС, является написание ее на машинно-независимом языке, например, на С, который и был разработан для программирования операционных систем.

Сетевая ОС имеет в своем составе средства передачи сообщений между компьютерами по линиям связи, которые совершенно не нужны в автономной ОС. На основе этих сообщений сетевая ОС поддерживает разделение ресурсов компьютера между удаленными пользователями, подключенными к сети. Для поддержания функций передачи сообщений сетевые ОС содержат специальные программные компоненты, реализующие популярные коммуникационные протоколы, такие как IP, IPX, Ethernet и другие. Многопроцессорные системы требуют от операционной системы особой организации, с помощью которой сама операционная система, а также поддерживаемые ею приложения могли бы выполняться параллельно отдельными процессорами системы. Параллельная работа отдельных частей ОС создает дополнительные проблемы для разработчиков ОС, так как в этом случае гораздо сложнее обеспечить согласованный доступ

отдельных процессов к общим системным таблицам, исключить эффект гонок и прочие нежелательные последствия асинхронного выполнения работ.

- **Особенности областей использования.** В зависимости от областей использования различают:
  - *системы пакетной обработки;*
  - *системы разделения времени;*
  - *системы реального времени.*

Многозадачные ОС подразделяются на три типа в соответствии с использованными при их разработке критериями эффективности: системы пакетной обработки (например, ОС ЕС), системы разделения времени (UNIX, VMS), системы реального времени (QNX, RT/11). *Системы пакетной обработки* предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени. Для достижения этой цели в системах пакетной обработки используются следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие отличающиеся требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины; так, например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом.

Таким образом, выбор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается "выгодное" задание. Следовательно, в таких ОС невозможно гарантировать выполнение того или иного задания в течение определенного периода времени. В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит только в случае, если активная задача сама отказывается от процессора, например, из-за необходимости выполнить операцию ввода-вывода. Поэтому одна задача может надолго занять процессор, что делает невозможным выполнение интерактивных задач. Таким образом, взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки,

сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат.

Очевидно, что такой порядок снижает эффективность работы пользователя.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки – изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю системы разделения времени предоставляется терминал, с которого он может вести диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго, и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что каждый из них единолично использует машину. Ясно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая "выгодна" системе, и, кроме того, имеются накладные расходы вычислительной мощности на более частое переключение процессора с задачи на задачу. Критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя.

Системы реального времени применяются для управления различными техническими объектами, такими, например, как станок, спутник, научная экспериментальная установка или технологическими процессами, такими, как гальваническая линия, доменный процесс и т.п. Во всех этих случаях существует предельно допустимое время, в течение которого должна быть выполнена та или иная программа, управляющая объектом, в противном случае может произойти авария: спутник выйдет из зоны видимости, экспериментальные данные, поступающие с датчиков, будут потеряны, толщина гальванического покрытия не будет соответствовать норме. Таким образом, критерием эффективности для систем реального времени является их способность выдерживать заранее заданные интервалы времени между запуском программы и получением результата (управляющего воздействия).

Это время называется временем реакции системы, а соответствующее свойство системы – реактивностью. Для этих систем мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а

выбор программы на выполнение осуществляется, исходя из текущего состояния объекта, или в соответствии с расписанием плановых работ. Некоторые операционные системы могут совмещать в себе свойства систем разных типов, например, часть задач может выполняться в режиме пакетной обработки, а часть – в режиме реального времени или в режиме разделения времени. В таких случаях режим пакетной обработки часто называют *фоновым режимом*.

- **Особенности методов построения.** Различают следующие методы построения ОС:

– способы построения ядра системы – монолитное ядро или микроядерный подход;

– построение ОС на базе объектно-ориентированного подхода;

– наличие нескольких прикладных сред;

– распределенная организация операционной системы.

При описании операционной системы часто указываются особенности ее структурной организации и основные концепции, положенные в ее основу.

К таким базовым концепциям относятся способы построения ядра системы – монолитное ядро или микроядерный подход. Большинство ОС использует *монолитное ядро*, которое компонуется как одна программа, работающая в привилегированном режиме и использующая быстрые переходы с одной процедуры на другую, не требующие переключения из привилегированного режима в пользовательский и наоборот. Альтернативой является построение ОС на базе *микроядра*, работающего также в привилегированном режиме и выполняющего только минимум функций по управлению аппаратурой, в то время как функции ОС более высокого уровня выполняют специализированные компоненты ОС – серверы, работающие в пользовательском режиме. При таком построении ОС работает более медленно, так как часто выполняются переходы между привилегированным режимом и пользовательским, зато система получается более гибкой – ее функции можно наращивать, модифицировать или сужать, добавляя, модифицируя или исключая серверы пользовательского режима. Кроме того, серверы хорошо защищены друг от друга, как и любые пользовательские процессы.

Построение ОС на базе *объектно-ориентированного подхода* дает возможность использовать все его достоинства, хорошо зарекомендовавшие себя на уровне приложений, внутри операционной системы, а именно: аккумуляцию удачных решений в форме стандартных объектов, возможность создания новых объектов на

базе имеющихся с помощью механизма наследования, хорошую защиту данных за счет их инкапсуляции во внутренние структуры объекта, что делает данные недоступными для несанкционированного использования извне, структурированность системы, состоящей из набора хорошо определенных объектов.

Наличие *нескольких прикладных сред* дает возможность в рамках одной ОС одновременно выполнять приложения, разработанные для нескольких ОС. Многие современные операционные системы поддерживают одновременно прикладные среды MS-DOS, Windows, UNIX (POSIX), OS/2 или хотя бы некоторого подмножества из этого популярного набора. Концепция множественных прикладных сред наиболее просто реализуется в ОС на базе микроядра, над которым работают различные серверы, часть которых реализуют прикладную среду той или иной операционной системы. Распределенная организация операционной системы позволяет упростить работу пользователей и программистов в сетевых средах.

В *распределенной ОС* реализованы механизмы, которые дают возможность пользователю представлять и воспринимать сеть в виде традиционного однопроцессорного компьютера. Характерными признаками распределенной организации ОС являются: наличие единой справочной службы разделяемых ресурсов, единой службы времени, использование механизма вызова удаленных процедур (RPC) для прозрачного распределения программных процедур по машинам, многонитевой обработки, позволяющей распараллеливать вычисления в рамках одной задачи и выполнять эту задачу сразу на нескольких компьютерах сети, а также наличие других распределенных служб.

1.4. Обзор аппаратного обеспечения компьютера

Операционная система тесно связана с оборудованием компьютера, на котором она должна работать. Аппаратное обеспечение влияет на набор команд ОС и управление его ресурсами.

Концептуально простой персональный компьютер можно представить в виде абстрактной модели (рисунок 1.2)

Рассмотрим кратко ее основные компоненты.

Процессор. Для каждого центрального процессора существует набор команд, который он в состоянии выполнить. Например, процессор Pentium не может обработать команды процессора SPARC и наоборот. Кроме основных регистров,

используемых для хранения переменных и временных результатов, имеются специальные регистры, видимые для программиста. Это такие, как *счетчик команд (PC, program counter)*, содержащий адрес следующей, стоящей в очереди на выполнение команды, *указатель стека (SP, stack pointer)*, в котором хранится адрес вершины стека в памяти, и, наконец, *слово состояния процессора (PSW, Processor Status Word)* – в нем располагаются биты кода состояний и другая служебная информация.

Рисунок 1.2 – Некоторые компоненты персонального компьютера

Операционная система должна знать все обо всех регистрах. При временном мультиплексировании центрального процессора ОС часто останавливает работающую программу для запуска (или перезапуска) другой. Каждый раз при таком прерывании ОС должна сохранять все регистры процессора, чтобы позже, когда программа продолжит свою работу, их можно было восстановить.

Существует несколько вариантов выполнения команд процессором. Самый старый и сейчас не используемый вариант – *последовательный*, когда за один такт может быть считана, декодирована и выполнена только одна команда. Второй вариант – *конвейер*. При такой организации процессора у него существуют раздельные модули, занимающиеся выборкой, декодированием и выполнением команд. То есть во время выполнения команды с номером n он может декодировать команду с номером $n + 1$ и считывать команду $n + 2$. Одним из недостатков конвейера является то, что считанная команда должна быть выполнена, даже если в предыдущей команде был принят условный переход.

Более передовым вариантом организации работы процессора является *суперскалярный* центральный процессор. В этой структуре присутствует множество выполняющих узлов: один для целочисленных арифметических операций, другой –

для операций с плавающей точкой, третий – для логических операций. За один такт считывается две или более команды, которые декодируются и сбрасываются в буфер хранения, где они ждут своей очереди на выполнение. Когда выполняющее устройство освобождается, оно заглядывает в буфер хранения, интересуясь, есть ли там команда, которую она может обработать, и если да, то забирает ее и выполняет. Это ускоряет работу процессора, однако одновременно и усложняет ее, так как команды часто исполняются не в порядке их следования и должна быть гарантия того, что результат совпадет с тем, который выдала бы последовательная конструкция.

Большинство процессоров имеют два режима работы: режим ядра и пользовательский режим. Обычно режим задается битом слова состояния процессора (регистра PSW). Если процессор запущен в *режиме ядра*, он может выполнять все команды из набора инструкций и использовать все возможности аппаратуры. ОС работает в режиме ядра, предоставляя доступ ко всему оборудованию.

В противоположность этому программы пользователей работают в *пользовательском режиме*, разрешающем выполнение подмножества команд и делающем доступной лишь часть аппаратных средств. Как правило, все команды, включая ввод-вывод данных и защиту памяти, запрещены в пользовательском режиме. Установка бита режима ядра в регистре PSW, естественно, недоступна.

Для связи с ОС пользовательская система должна сформировать *системный вызов*, который обеспечивает переход в режим ядра и активирует функции ОС. Команда TRAP (эмулированное прерывание) переключает режим работы процессора из пользовательского в режим ядра и передает управление ОС. После завершения работы управление возвращается к пользовательской программе, к команде, следующей за системным вызовом.

Память. Второй основной составляющей любого компьютера является память. Системы памяти конструируются в виде иерархии слоев. Верхний слой состоит из внутренних *регистров* центрального процессора. Скорость доступа к ним максимальна (< 1 нс), а объем – минимален (< 1 Кбайт). Следующий слой – это *кэш-память*. Она может иметь несколько уровней, каждый из которых имеет меньшую скорость доступа при большем объеме. Еще один слой – это *оперативная память* (ОЗУ, RAM, Random Access Memory). Следующий слой – это *ПЗУ* (жесткий диск). Он представляет из себя механическую структуру, в связи с чем скорость доступа к

нему намного меньше, однако емкость очень большая. Кроме того, существуют внешние носители данных – *CD, DVD, BRD, дискеты, флэш-память, магнитные ленты (стримеры)*. Еще один вариант памяти в компьютере – это *CMOS-память*. Она питается от батарейки и обеспечивает сохранение конфигурационных параметров, используемых при загрузке компьютера, а также системное время.

Устройства ввода-вывода. Они состоят из двух частей – контроллера и самого устройства. *Контроллер* – это микросхема или набор микросхем, управляющая устройством. Он принимает команды ОС и выполняет их. Программа, которая общается с контроллером, отдает ему команды и получает ответы, называется *драйвером устройства*. Ввод и вывод данных можно осуществлять тремя различными способами. Простейший метод состоит в том, что пользовательская программа выдает системный запрос, который ядро транслирует в вызов процедуры соответствующего драйвера. Затем драйвер начинает процесс ввода-вывода. В это время драйвер выполняет очень короткий программный цикл, постоянно опрашивая готовность устройства, с которым он работает (обычно есть некий бит, который указывает на то, что устройство все еще занято). По завершении операции ввода-вывода драйвер помещает данные туда, куда требуется, и возвращается в исходное состояние. Затем операционная система возвращает управление программе, осуществлявшей вызов. Этот метод называется *ожиданием готовности* или *активным ожиданием* и имеет один недостаток: процессор должен опрашивать устройство до тех пор, пока оно не завершит свою работу.

При втором способе драйвер запускает устройство и просит его выдать *прерывание* по окончании ввода-вывода. После этого драйвер возвращает данные, ОС блокирует программу вызова, если это нужно, и начинает выполнять другие задания. Когда контроллер обнаруживает окончание передачи данных, он генерирует прерывание, чтобы сигнализировать о завершении операции.

Третий метод ввода-вывода информации заключается в использовании специального контроллера *прямого доступа к памяти* (DMA, Direct Memory Access), который управляет потоком битов между оперативной памятью и некоторыми контроллерами без постоянного вмешательства центрального процесса. Процессор вызывает микросхему DMA, говорит ей, сколько байтов нужно передать, сообщает адреса устройства и памяти, а также направление передачи данных и позволяет дальше действовать ей самой. По завершении работы DMA инициирует прерывание, которое обрабатывается так же, как было описано выше.

Шины. На данный момент структура, представленная на рисунке 1.2, претерпела значительные изменения. Одна шина уже перестала справляться с требованиями, предъявляемыми к скорости передачи данных. В связи с этим сейчас существует более разветвленная структура шин данных. Она представлена такими шинами, как ISA, IDE, USB, SCSI, IEEE 1394 и другие.

1.5. Основные понятия операционных систем

Для каждой операционной системы существует набор базовых понятий, которые являются самыми важными для понимания общей идеи. Кратко рассмотрим их.

Процесс. Это программа в момент выполнения. С каждым процессом связывается его *адресное пространство (образ памяти)* – список адресов в памяти от некоторого минимума (обычно нуля) до некоторого максимума, которые процесс может прочесть, и в которые он может писать. Адресное пространство содержит саму программу, данные к ней и ее стек. Со всяким процессом связывается некий *набор регистров*, включая счетчик команд, указатель стека и другие аппаратные регистры, плюс вся остальная информация, необходимая для запуска программы.

Во многих операционных системах вся информация о каждом процессе хранится в *таблице процессов* и представляет собой массив структур, по одной на каждый существующий в данный момент процесс.

Взаимоблокировка. Когда взаимодействуют два или более процесса, они могут попадать в патовые ситуации, из которых невозможно выйти без посторонней помощи.

Управление памятью. В очень простых ОС в конкретный момент времени в памяти может находиться только одна программа. Для запуска второй программы сначала нужно удалить из памяти первую и загрузить на ее место вторую. Более сложные системы позволяют одновременно находиться в памяти нескольким программам. Для того, чтобы они не мешали друг другу (и операционной системе), необходим некий защитный механизм. Хотя этот механизм располагается в аппаратуре, он управляется операционной системой.

Ввод-вывод данных. Каждая ОС имеет свою подсистему ввода-вывода для управления устройствами ввода-вывода. Некоторые из программ ввода-вывода являются независимыми от устройств, то есть их можно применить ко многим или ко всем устройствам ввода-вывода. Другая часть программного обеспечения ввода-

вывода, в которую входят драйверы устройств, предназначена для определенных устройств ввода-вывода.

Файловая система. Это еще одно ключевое понятие, поддерживаемое виртуально всеми ОС. Основной функцией ОС является скрытие особенностей дисков и других устройств ввода-вывода и предоставление пользователю понятной и удобной абстрактной модели независимых от устройств файлов. Системные вызовы необходимы для создания, удаления, чтения или записи файлов. Перед тем как прочитать файл, его нужно разместить на диске и открыть, а после прочтения его нужно закрыть. Все эти функции осуществляют системные вызовы.

Безопасность. Компьютеры содержат большое количество конфиденциальной информации. В задачу ОС входит управление системой защиты подобных файлов, так чтобы они, например, были доступны только пользователям, имеющим на это права. Кроме защиты файлов существует еще множество других вопросов безопасности: защита системы от нежелательных гостей, людей, вирусов.

1.6. Структура операционной системы

Наиболее общим подходом к структуризации ОС является разделение всех ее модулей на две группы: ядро (модули, выполняющие основные функции) и модули, выполняющие вспомогательные функции ОС.

Модули ядра выполняют такие базовые функции, как управление процессами, памятью, устройствами ввода-вывода и т.п. Остальные модули ОС выполняют весьма полезные, но менее обязательные функции. Например, к таким *вспомогательным модулям* относятся программы архивирования данных, дефрагментации диска, текстового редактора. Вспомогательные модули ОС оформляются либо в виде приложений, либо в виде библиотек процедур.

Операционная система имеет многослойную структуру (рисунок 1.3).

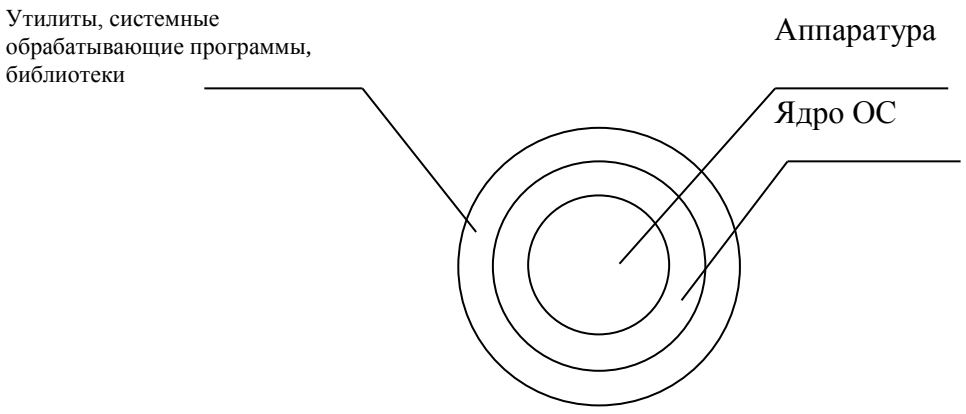


Рисунок 1.3 – Трехслойная схема вычислительной системы

При этом ядро может состоять из следующих слоев (рисунок 1.4):

1. средства аппаратной поддержки ОС;
2. машинно-зависимые компоненты ОС;
3. базовые механизмы ядра;
4. менеджеры ресурсов
5. интерфейс системных вызовов.

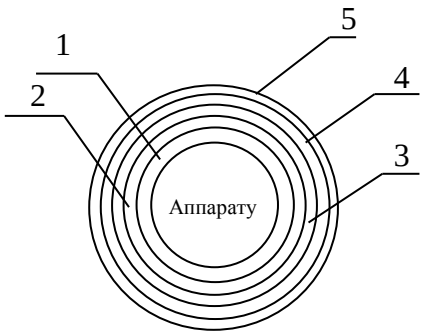


Рисунок 1.4 – Многослойная структура ядра ОС

Приведенное разбиение ядра ОС на слои является достаточно условным. В реальной системе количество слоев и распределение функций между ними может быть и иным.

2. ПРОЦЕССЫ И ПОТОКИ

2.1. Подсистема управления процессами и потоками

Одной из основных подсистем многозадачной ОС, непосредственно влияющей на функционирование вычислительной машины, является *подсистема управления процессами и потоками*, которая занимается их созданием и уничтожением, поддерживает взаимодействие между ними, а также распределяет процессорное время между несколькими одновременно существующими в системе процессами и потоками.

Подсистема управления процессами и потоками ответственна 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 в совместное использование с другими процессами. Некоторые из ресурсов выделяются процессу при его создании, а некоторые – динамически по запросам во время выполнения. Ресурсы могут быть приписаны 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.

Когда в системе одновременно выполняется несколько независимых задач, то возникают дополнительные проблемы. Хотя потоки возникают и выполняются асинхронно, у них может возникнуть необходимость во взаимодействии, например при обмене данными. Согласование скоростей потоков также очень важно для предотвращения эффекта "гонок" (когда несколько потоков пытаются изменить один и тот же файл), взаимных блокировок или других коллизий, которые возникают при совместном использовании ресурсов. Синхронизация потоков является одной из важных функций подсистемы управления процессами и потоками.

Итак, что же такое процессы и потоки. **Процесс** – это выполняемая программа, включая текущие значения счетчика команд, регистров и переменных. В простейшем случае процесс состоит из одного потока. В таких случаях понятие "поток" полностью поглощается понятием "процесс".

В чем же состоят принципиальные отличия в понятиях "процесс" и "поток"? Очевидно, что любая работа вычислительной системы заключается в выполнении некоторой программы. Поэтому и с процессом, и с потоком связывается определенный программный код, который для этих целей оформляется в виде исполняемого модуля. Чтобы этот программный код мог быть выполнен, его необходимо загрузить в оперативную память, возможно, выделить некоторое место на диске для хранения данных, предоставить доступ к устройствам ввода-вывода и т.д. В ходе выполнения программе может понадобиться доступ к информационным ресурсам, например, файлам, портам TCP/UDP, семафорам. И, конечно же, невозможно выполнение программы без предоставления ей процессорного времени, то есть времени, в течение которого процессор выполняет коды данной программы.

В ОС, где существуют и процессы, и потоки, процесс рассматривается операционной системой как заявка на потребление всех видов ресурсов, кроме одного – процессорного времени. Этот последний важнейший ресурс распределяется ОС между другими единицами работы – потоками, которые и получили свое название благодаря тому, что они представляют собой последовательности (потоки выполнения) команд.

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 ОС обеспечивает каждый процесс отдельным виртуальным адресным пространством, так что ни один процесс не может получить прямого доступа к командам и данным другого процесса. При необходимости взаимодействия процессы обращаются к ОС, которая, выполняя функции посредника, предоставляет им средства межпроцессной связи – конвейеры, почтовые ящики, разделяемые секции память и некоторые другие.

Процесс можно рассматривать как способ группирования родственных ресурсов в одну группу. У процесса есть адресное пространство, содержащее код программы и данные, а также другие ресурсы. Ресурсами могут быть открытые файлы, обработчики сигналов, учетная информация и многое другое. Гораздо проще управлять ресурсами, объединив их в форме процесса.

У *потока* же есть счетчик команд, отслеживающий порядок выполнения действий. У него есть регистры, в которых хранятся текущие переменные. У него есть стек, содержащий протокол выполнения процесса.

Таким образом, процессы используются для группирования ресурсов, а потоки являются объектами, поочередно выполняющимися на центральном процессоре.

Концепция потоков добавляет к модели процесса возможность одновременного выполнения в одной и той же среде процесса нескольких программ, в достаточной степени независимых. Несколько потоков, работающих параллельно в одном процессе, аналогичны нескольким процессам, идущим параллельно на одном компьютере.

Рассмотрим, какие элементы процесса совместно используются всеми потоками, а что является индивидуальным для каждого потока (таблица 2.1). Из таблицы видно, что все потоки совместно используют набор ресурсов процесса, а индивидуальными являются элементы, относящиеся к выполнению некоторой задачи.

Таблица 2.1 – Разделяемые и индивидуальные элементы потока

Элементы процесса, разделяемые потоками	**Элементы, индивидуальные для потока**
Адресное пространство	Счетчик команд
Глобальные переменные	Регистры
Открытые файлы	Стек
Дочерние процессы	Состояние
Сигналы и их обработчики	
Информация об использовании ресурсов	

Иногда операционная система не заботится о программных потоках. Другими словами, управление программными потоками происходит целиком в пользовательском режиме. Например, когда программный поток блокируется, то, перед тем как остановиться, он решает, какой программный поток должен выполняться следующим, и запускает его. Существуют и широко используются несколько библиотек для поддержки пользовательских программных потоков. Переключение с одного потока на другой в этом случае происходит существенно быстрее, чем при использовании процессов, так как требует сохранения только счетчика команд и регистров и может производиться без использования ядра.

В других случаях ОС учитывает существование множества потоков, и когда один программный поток переходит в состояние блокировки, система выбирает для запуска следующий поток в том же самом процессе или в другом. Чтобы поддерживать такую функциональность, ядро системы должно хранить таблицу всех программных потоков в системе, наподобие таблицы процессов.

2.2. Модель процесса

Следить за работой параллельно идущих процессов достаточно трудно, поэтому была разработана концептуальная модель последовательных процессов, упрощающая эту работу. В этой модели у каждого процесса есть собственный виртуальный центральный процессор. На самом деле реальный процессор переключается с процесса на процесс, но для лучшего понимания системы значительно проще рассматривать набор процессов, идущих параллельно (псевдопараллельно), чем пытаться представить себе процессор, переключающийся от программы к программе. Такое переключение, как уже говорилось, и называется многозадачностью или мультипрограммированием.

На рисунке 2.1, *а* представлена схема компьютера, работающего с четырьмя программами. На рисунке 2.1, *б* представлены четыре процесса, каждый со своей управляющей логикой (то есть логическим счетчиком команд), идущие независимо друг от друга. Разумеется, на самом деле существует только один физический счетчик команд, в который загружается логический счетчик команд текущего процесса. Когда время, отведенное текущему процессу, заканчивается, физический счетчик команд сохраняется в логическом счетчике команд процесса в памяти. На рисунке 2.1, *в* видно, что за достаточно большой промежуток времени изменилось состояние всех четырех процессов, но в каждый конкретный момент времени в действительности работает только один процесс.

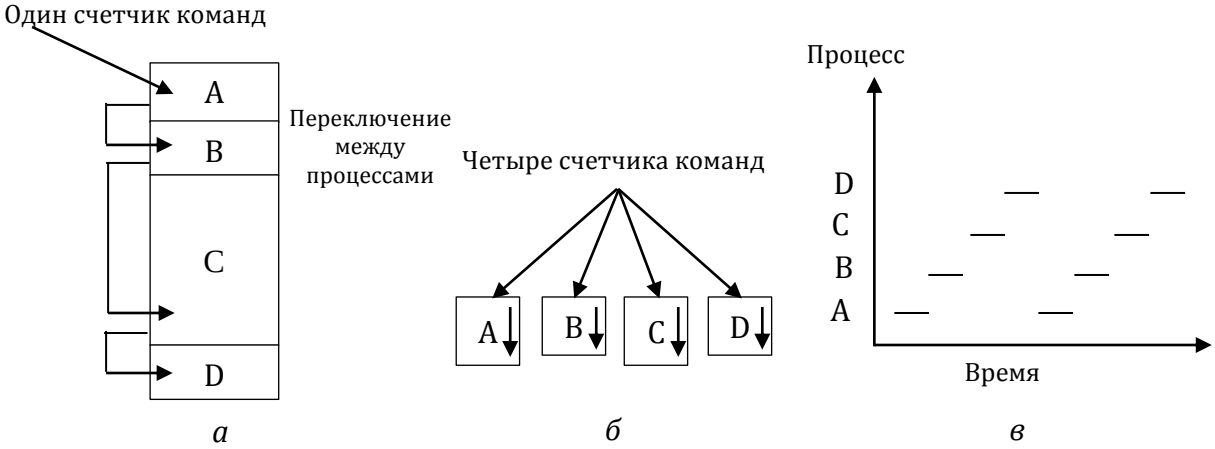


Рисунок 2.1 – Четыре программы в многозадачном режиме (*а*); принципиальная модель четырех независимых последовательных процессов (*б*); в каждый момент времени активна только одна программа (*в*)

Поскольку процессор переключается между программами, скорость, с которой процессор производит свои вычисления, будет непостоянной и, возможно, даже будет отличной при каждом новом запуске процесса. Поэтому не следует программировать процессы, исходя из каких-либо жестко заданных временных предположений. Представьте себе, например, процесс ввода вывода, запускающий накопитель на магнитной ленте для восстановления заархивированных файлов. Процесс выполняет холостой цикл задержки 10 000 раз, чтобы дать время накопителю разогнаться, а затем дает команду считать первый сектор. Если во время холостого цикла процессор решит переключиться на другую задачу, может случиться гак, что работающий с магнитофоном процесс запустится снова уже после того, как считывающая головка пройдет первую запись. Если у процесса есть критические временные рамки такого рода, то есть отдельные события должны укладываться в заданное количество миллисекунд, необходимы специальные меры, чтобы удостовериться в завершенности события. Однако обычно многозадачный режим процессора, а также относительные скорости различных процессов не влияют на работу большинства процессов.

Чтобы пояснить различие между процессом и программой, воспользуемся аналогией: представьте повара, пекущего торт на день рождения дочери. У него есть рецепт, кухня и ингредиенты для торта. Здесь рецепт – это программа, повар – процессор, ингредиенты – входные данные. Процессом является следующая последовательность действий: повар читает рецепт, смешивает продукты, печет торт.

Теперь представим, что на кухню забегает сын, которого укусила пчела. Повар отмечает, на чем он остановился (сохраняет текущее состояние процесса), находит справочник по оказанию первой помощи и действует по инструкции. Таким образом, процессор переключился с одного процесса на другой с большим приоритетом, и у каждого процесса есть своя программа (рецепт и справочник). После оказания первой помощи повар возвращается к торту, продолжая с той операции, на которой он остановился.

Эта аналогия показывает, что процесс – это активность некоторого рода. У него есть программа, входные и выходные данные, а также состояние. Один процессор может переключаться между различными процессами, используя некий алгоритм планирования для определения момента переключения от одного процесса к другому.

2.3. Создание, завершение и состояние процессов.

Любой процесс в операционной системе характеризуется своим состоянием. Количество состояний и переход между состояниями процесса различны в разных операционных системах. Рассмотрим типичные состояния процесса и переходы из одного состояния в другое (рисунок 2.2).

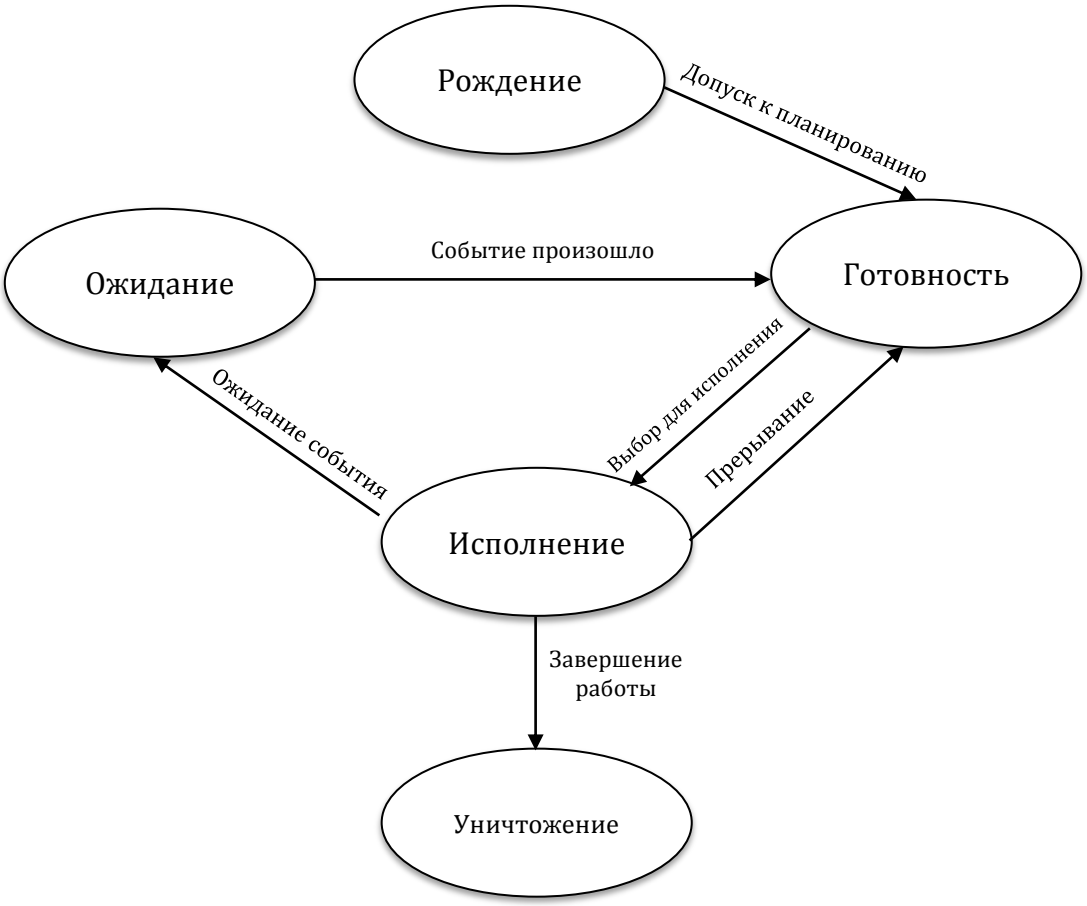


Рисунок 2.2 – Диаграмма состояний процесса

Представленная на рисунке диаграмма состояний процесса может отличаться от принятой в конкретной операционной системе. Но, тем не менее, она отражает типичный жизненный цикл процесса: рождение – готовность – выполнение – уничтожение.

При **рождении** процесс получает в свое распоряжение адресное пространство, в которое загружается программный код процесса. Ему выделяются стек и системные ресурсы, устанавливается начальное значение программного счетчика процесса и т.д.

Существуют четыре основных события, приводящих к созданию процессов:

1. Инициализация системы.

2. Исполнение запущенным процессом системного вызова создания процесса.

3. Запрос пользователя на создание процесса.

4. Инициализация пакетного задания.

При загрузке операционной системы обычно создаются несколько процессов. Некоторые из них являются процессами, взаимодействующими с пользователями и выполняющими для них определенную работу. Другие процессы являются фоновыми: они не связаны с конкретными пользователями, однако имеют определенное функциональное назначение. Процессы, выполняемые в фоновом режиме и осуществляющие определенную деятельность (например, печать), часто называют *демонами*. Процессы могут создаваться не только при загрузке операционной системы, но и во время ее работы. Запущенный процесс может создать один или несколько новых процессов, которые помогают ему выполнять свои функции. Таким образом, процессы в операционной системы образуют дерево.

В любом случае новый процесс создается путем выполнения соответствующего системного вызова существующим процессом. В качестве последнего может выступать пользовательский процесс, системный процесс, активизируемый клавиатурой или мышью, а также менеджер пакетной обработки. Процесс исполняет системный вызов, запрашивающий у операционной системы создание нового процесса, и указывает (прямо или косвенно), какая программа должна быть запущена в этом процессе.

Всякий новый процесс, появляющийся в системе, попадает в состояние **готовность**. Операционная система, пользуясь алгоритмом планирования, выбирает один из готовых процессов и переводит его в состояние **исполнение**. В этом состоянии происходит непосредственное выполнение программного кода процесса. Выйти из этого состояния процесс может по следующим причинам:

– операционная система прекращает выполнение процесса;

– процесс не может продолжать свою работу, пока не произойдет некоторое событие. В этом случае операционная система переводит процесс в состояние **ожидания**; После того, как событие произошло, операционная система возвращает процесс в состояние **готовность**.

– в результате возникновения прерывания в вычислительной системе (например, от таймера по истечении предусмотренного времени выполнения) операционная система возвращает процесс в состояние **готовность**.

После завершения своей деятельности процесс переходит в состояние **уничтожение**.

Процесс завершается одним из следующих четырех способов:

1. Нормальное завершение (добровольное).
2. Завершение вследствие ошибки (добровольное).
3. Завершение вследствие фатальной ошибки (принудительное).
4. Уничтожение другим процессом (принудительное).

По завершении работы операционная система освобождает все ресурсы, ассоциированные с данным процессом.

При управлении процессами операционная система использует два основных типа информационных структур: дескриптор процесса и контекст процесса. **Дескриптор процесса** содержит такую информацию о процессе, которая необходима ядру в течение всего жизненного цикла процесса независимо от того, в каком состоянии он находится, а также находится ли образ процесса в оперативной памяти или выгружен на диск. Под **образом процесса** понимается совокупность его кодов и данных.

Дескрипторы отдельных процессов объединены в список, образующий таблицу процессов. Память для таблицы процессов отводится динамически в области ядра. На основании информации, содержащейся в таблице процессов, операционная система осуществляет планирование и синхронизацию процессов. В дескрипторе прямо или косвенно (через указатели на связанные с процессом структуры) содержится информация о состоянии процесса, о расположении образа процесса в оперативной памяти и на диске, о значении отдельных составляющих приоритета, а также о его итоговом значении – глобальном приоритете, об идентификаторе пользователя, создавшего процесс, о родственных процессах, о событиях, осуществления которых ожидает данный процесс, и некоторая другая информация.

Контекст процесса содержит менее оперативную часть информации о процессе, необходимую для возобновления выполнения процесса с прерванного места: содержимое регистров процессора, коды ошибок выполняемых процессором системных вызовов, информация обо всех открытых данным процессом файлах и незавершенных операциях ввода-вывода и другие данные, характеризующие состояние вычислительной системы в момент прерывания. Контекст, также как и дескриптор процесса, доступен только программам ядра, то есть находится в виртуальном адресном пространстве операционной системы, однако хранится он не

в области ядра, а непосредственно примыкает к образу процесса и перемещается вместе с ним, если это необходимо, из оперативной памяти на диск.

Большинство современных операционных систем позволяют работать сразу с большим количеством процессов. Это позволяет лучше загружать ресурсы вычислительной системы: в то время как выполняющийся процесс может оказаться заблокированным в результате ожидания некоторого события или ресурса (например, ожидает данные из внешней памяти), другие процессы могут находиться в состоянии готовности, и какой-либо из них будет переведен в состояние выполнения. Таким образом, в результате изменения состояния процесса операционной системе приходится переключаться с одного процесса на другой. Для корректного переключения процессов необходимо сохранить контекст исполнявшегося процесса и восстановить контекст процесса, на который будет переключен процессор. Такая процедура сохранения/восстановления работоспособности процессов называется **переключением контекста**.

На рисунке 2.3. показано, как происходит переключение контекста при возникновении прерывания.

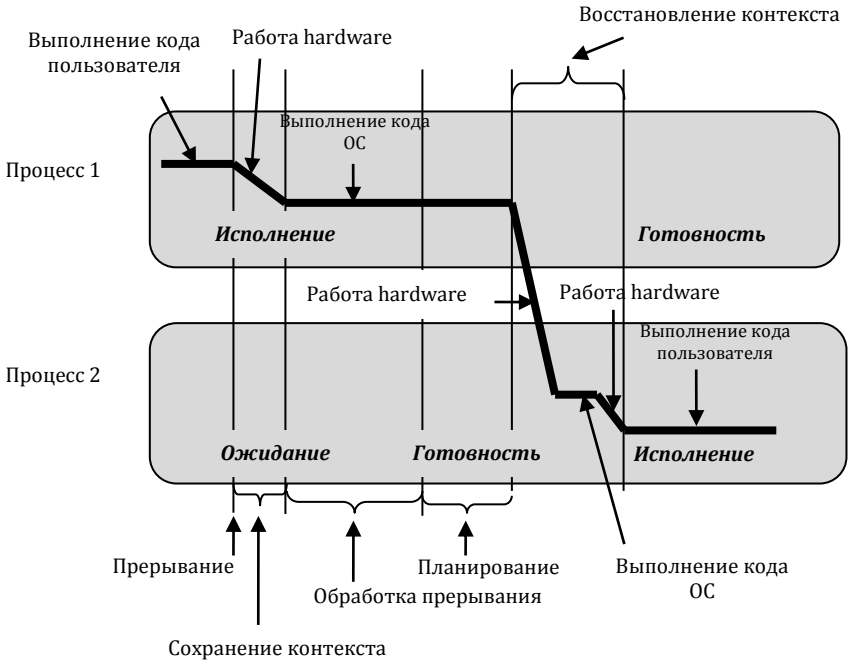


Рисунок 2.3 – Переключение контекста

Видно, что время, затрачиваемое на переключение контекста, складывается из времени сохранения контекста первого процесса, времени обработки самого прерывания, времени, затрачиваемого операционной системой на планирование (определение того процесса, который будет выполняться), и времени восстановления контекста второго процесса. Это время не используется

вычислительной системой для совершения полезной работы и представляет собой накладные расходы, снижающие производительность системы. Оно зависит от конкретных особенностей вычислительной системы и может составлять от 1 до 1000 микросекунд. Существенно сократить накладные расходы позволяет расширенная модель процессов, включающая в себя понятие потоков.

2.4. Использование и реализация потоков

Концепция потоков добавляет к модели процесса возможность одновременного выполнения в одной и той же среде процесса нескольких программ, в достаточной степени независимых. Несколько потоков, работающих параллельно в одном процессе, аналогичны нескольким процессам, идущим параллельно на одном компьютере. В первом случае потоки разделяют адресное пространство, открытые файлы и другие ресурсы. Во втором случае процессы совместно пользуются физической памятью, дисками, принтерами и другими ресурсами. Потоки обладают некоторыми свойствами процессов, поэтому их иногда называют упрощенными процессами. Термин **многопоточность** также используется для описания использования нескольких потоков в одном процессе.

На рисунке 2.4, *а* представлены три обычных процесса, у каждого из которых есть собственное адресное пространство и одиночный поток управления. На рисунке 2.4, *б* представлен один процесс с тремя потоками управления. В обоих случаях мы имеем три потока, но на рисунке 2,4, *а* каждый из них имеет собственное адресное пространство, а на рисунке 2.4, *б* потоки разделяют единое адресное пространство.

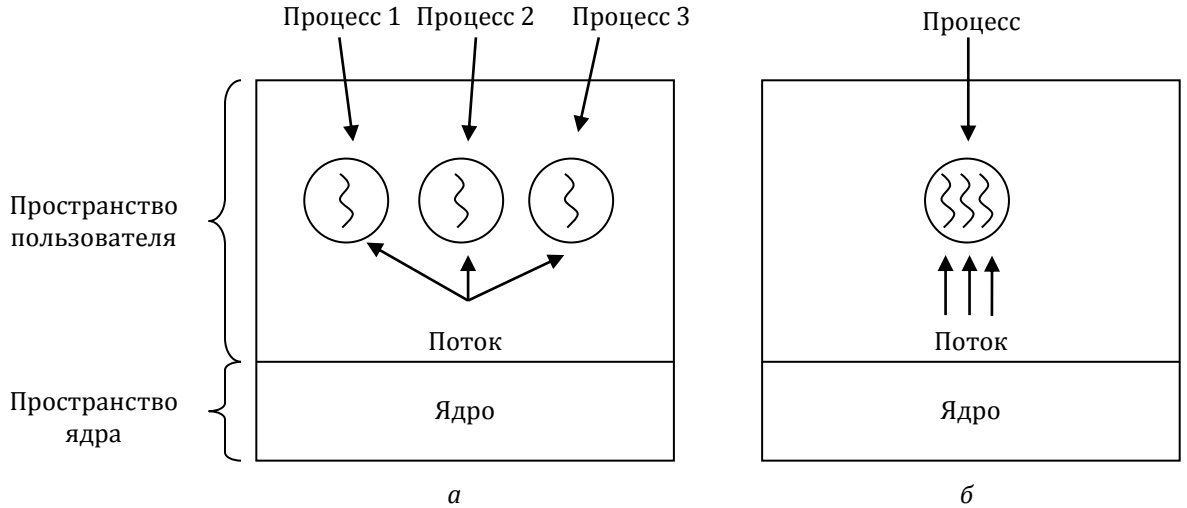


Рисунок 2.4 – Три процесса с одиночными потоками управления (*а*);
один процесс с тремя потоками управления (*б*)

При запуске многопоточного процесса в системе с одним процессором потоки работают поочередно. Иллюзия параллельной работы нескольких различных последовательных процессов создается путем постоянного переключения системы между процессами. Многопоточность реализуется примерно так же. Процессор быстро переключается между потоками, создавая впечатление параллельной работы потоков, хотя и на не столь быстром процессоре. В случае трех ограниченных производительностью процессора потоков в одном процессе все потоки будут работать параллельно, и каждому потоку будет соответствовать виртуальный процессор с быстродействием, равным одной трети быстродействия реального процессора.

Различные потоки в одном процессе не так независимы, как различные процессы. У всех потоков одно и то же адресное пространство, что означает совместное использование глобальных переменных. Поскольку любой поток имеет доступ к любому адресу ячейки памяти в адресном пространстве процесса, один поток может считывать, записывать или даже стирать информацию из стека другого потока. Защиты не существует, поскольку это невозможно и ненужно. В отличие от различных процессов, которые могут быть инициированы различными пользователями и преследовать несовместимые цели, один процесс всегда запущен одним пользователем, и потоки созданы таким образом, чтобы работать совместно, не мешая друг другу. Как уже говорилось, потоки разделяют не только адресное пространство, но и открытые файлы, дочерние процессы, сигналы и т. п. Таким образом, ситуацию на рисунке 2.4, *а* следует использовать в случае абсолютно несвязанных процессов, тогда как схема на рисунке 2.4, *б* будет уместна, когда потоки выполняют совместно одну работу.

Как и любой обычный процесс (то есть процесс с одним потоком), поток может находиться в одном из нескольких состояний: рабочем, заблокированном, готовности или завершенном. Действующий поток взаимодействует с процессором. Блокированный поток ожидает некоторого события, которое его разблокирует. Например, при выполнении системного запроса чтения с клавиатуры поток блокируется, пока не поступит сигнал с клавиатуры. Поток может быть разблокирован каким-либо внешним событием или другим потоком. Поток в состоянии готовности будет запущен, как только до него дойдет очередь. Переходы между состояниями потоков такие же, как на рисунке 2.2.

Важно понимать, что у каждого потока свой собственный стек. Стек каждого потока содержит по одному фрейму для каждой процедуры, вызванной, но еще не вернувшей управления. Во фрейме находятся локальные переменные процедуры и адрес возврата. Например, если процедура X вызывает процедуру Y, и она, в свою очередь, вызывает процедуру Z, то во время работы процедуры Z в стеке будут находиться фреймы для всех трех процедур. Каждый поток может вызывать различные процедуры и, соответственно, иметь различный протокол выполнения процесса – именно поэтому каждому потоку необходим собственный стек.

Теперь разберемся, почему же, собственно, потоки так необходимы. Основной причиной является выполнение большинством приложений существенного числа действий, некоторые из них могут время от времени блокироваться. Схему программы можно существенно упростить, если разбить приложение на несколько последовательных потоков, запущенных в квазипараллельном режиме.

Еще одним аргументом в пользу потоков является легкость их создания и уничтожения (поскольку с потоком не связаны никакие ресурсы). В большинстве систем на создание потока уходит примерно в 100 раз меньше времени, чем на создание процесса.

Третьим аргументом является производительность. Концепция потоков не дает увеличения производительности, если все они ограничены возможностями процессора. Но когда имеется одновременная потребность в выполнении большого объема вычислений и операций ввода-вывода, наличие потоков позволяет совмещать эти виды деятельности во времени, тем самым увеличивая общую скорость работы приложения.

И, наконец, концепция потоков полезна в системах с несколькими процессорами, где возможен настоящий параллелизм.

Существует 2 способа реализации пакета потоков: в пространстве пользователя и в ядре.

Первый метод состоит в размещении пакета потоков целиком в пространстве пользователя (рисунок 2.5, *а*). При этом ядро о потоках ничего не знает и управляет обычными однопоточными процессами. Наиболее очевидное преимущество этой модели состоит в том, что пакет потоков на уровне пользователя можно реализовать даже в операционной системе, не поддерживающей потоки.

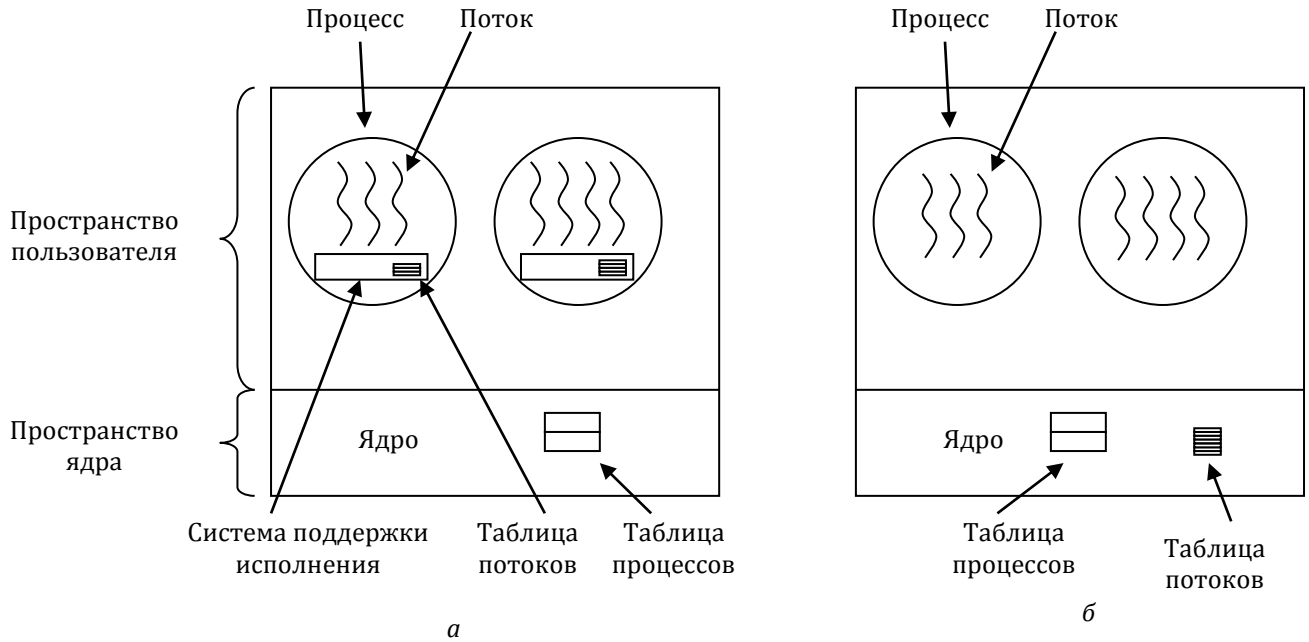


Рисунок 2.5 – Пакет потоков в пространстве пользователя (*а*);
пакет потоков, управляемый ядром (*б*)

Если управление потоками происходит в пространстве пользователя, то потоки работают поверх системы поддержки исполнения команд. При этом каждому процессу необходима собственная таблица потоков для отслеживания потоков в процессе. Эта таблица аналогична таблице процессов с той лишь разницей, что она отслеживает лишь характеристики потоков – счетчик команд, указатель вершины стека, регистры, состояние и т.п.

В тот момент, когда поток завершает на время свою работу, программа может сама сохранить информацию о потоке в таблице потоков, а после этого вызвать планировщик потоков для выбора следующего потока. При этом не требуется прерывание, переключение контекста, сохранение кэша и т.п., что существенно ускоряет переключение потоков.

Потоки, реализованные на уровне пользователя, имеют и другие преимущества. Так, они позволяют каждому процессу иметь собственный алгоритм планирования.

Однако, несмотря на более высокую производительность, с реализацией потоков на уровне пользователя связаны и некоторые проблемы. Первой из них является проблема реализации блокирующих системных запросов. Схожей проблемой является ошибка из-за отсутствия страницы в случае, если не вся программа находилась в основной памяти. В такой ситуации ядро блокирует весь процесс, несмотря на наличие остальных нормально функционирующих потоков.

Еще одной проблемой является тот факт, что при запуске одного потока ни один другой поток не будет запущен, пока первый поток добровольно не отдаст процессор.

И, наконец, самый серьезный недостаток потоков, реализованных на уровне пользователя, состоит в том, что программисты хотят использовать потоки именно в тех приложениях, в которых потоки часто блокируются, например, в многопоточном web-сервере. Эти потоки все время посылают системные запросы. И ядру, перехватившему управление, чтобы выполнить системный запрос, не составит труда заодно переключить потоки, если один из них блокирован.

В случае реализации потоков в ядре (рисунок 2.5, *б*) таблица потоков содержится в самом ядре дополнительно к обычной таблице процессов.

Все запросы, которые могут блокировать поток, реализуются как системные запросы, что требует значительно больших временных затрат, чем вызов процедуры системы поддержки исполнения программ. Когда поток блокируется, ядро запускает другой поток из этого же процесса (если есть поток в состоянии готовности) либо из другого.

Поскольку создание и завершение потоков в ядре требует относительно больших расходов, некоторые системы используют повторное использование потоков.

Управление потоками в ядре не требует новых не блокирующих системных запросов. Более того, если один поток вызвал ошибку из-за отсутствия страницы, ядро легко может проверить, есть ли в этом процессе потоки в состоянии готовности, и запустить один из них, пока требуемая страница считывается с диска.

Основным недостатком управления потоками в ядре является существенная цена системных запросов, поэтому постоянные операции с потоками приведут к увеличению накладных расходов.

С целью совмещения преимуществ реализации потоков на уровне ядра и на уровне пользователя были опробованы многие способы смешанной реализации. Один из методов заключается в использовании управления ядром и последующем мультиплексировании потоков на уровне пользователя.

В такой модели ядро знает только о потоках своего уровня и управляет ими. Некоторые из этих потоков могут содержать по нескольку потоков пользовательского уровня, мультиплексированных поверх них. Такие потоки создаются, завершаются и управляются также, как и потоки уровня пользователя в

процессе, запущенно в не поддерживающей многопоточность системе. Предполагается, что у каждого потока ядра есть набор потоков на уровне пользователя, которые используют его по очереди.

3. МЕЖПРОЦЕССНОЕ ВЗАИМОДЕЙСТВИЕ

3.1. Основные понятия

Процессам необходимо взаимодействовать между собой. Поэтому необходимо правильно организованное взаимодействие между процессами, по возможности не использующее прерываний.

Проблема межпроцессного взаимодействия разбивается на три пункта. Во-первых, это *необходимость передачи информации* от одного процесса другому. Во-вторых, *контроль над деятельностью процессов*: как гарантировать, что два процесса не пересекутся в критических ситуациях. И, в-третьих, *согласование действий процессов*: если процесс *A* должен поставлять данные, а процесс *B* выводить их на печать, то процесс *B* должен подождать и не начинать печатать, пока не поступят данные от процесса *A*. При этом второй и третий пункты можно в равной степени отнести и к потокам. Передача информации в случае потоков не является проблемой, так как у них общее адресное пространство.

В некоторых операционных системах процессы, работающие совместно, могут сообща использовать некое общее хранилище данных. Каждый из процессов может считывать из общего хранилища данных и записывать туда информацию. Это хранилище представляет собой участок в основной памяти (возможно, в структуре данных ядра) или файл общего доступа. Местоположение совместно используемой памяти не влияет на суть взаимодействия и возникающие проблемы. Рассмотрим межпроцессное взаимодействие на простом, но очень распространенном примере: спулер печати. Если процессу требуется вывести на печать файл, он помещает имя файла в специальный каталог спулера. Другой процесс, демон печати, периодически проверяет наличие файлов, которые нужно печатать, печатает файл и удаляет его имя из каталога.

Представьте, что каталог спулера состоит из большого числа сегментов, пронумерованных 0, 1, 2, и т.д., в каждом из которых может храниться имя файла. Также есть две совместно используемые переменные; *out*, указывающая на следующий файл для печати, и *in*, указывающая на следующий свободный сегмент. Эти две переменные можно хранить в одном файле (состоящем из двух слов), доступном всем процессам. Пусть в данный момент сегменты с 0 по 3 пусты (эти файлы уже напечатаны), а сегменты с 4 по 6 заняты (эти файлы ждут своей очереди

на печать). Более или менее одновременно процессы *A* и *B* решают поставить файл в очередь на печать.

В этом случае возможно возникновение следующей ситуации. Процесс *A* считывает значение (7) переменной *in* и сохраняет его в локальной переменной *next_free_slot*. После этого происходит прерывание по таймеру, и процессор переключается на процесс *B*. Процесс *B*, в свою очередь, считывает значение переменной *in* и сохраняет его (опять 7) в своей локальной переменной *next_free_slot*. В данный момент оба процесса считают, что следующий свободный сегмент – седьмой. Процесс *B* сохраняет в каталоге спулера имя файла и заменяет значение *in* на 8, затем продолжает заниматься своими задачами, не связанными с печатью.

Наконец управление переходит к процессу *A*, и он продолжает с того места, на котором остановился. Он обращается к переменной *next_free_slot*, считывает ее значение и записывает в седьмой сегмент имя файла (разумеется, удаляя при этом имя файла, записанное туда процессом *B*). Затем он заменяет значение *in* на 8. Структура каталога спулера не нарушена, так что демон печати не заподозрит ничего плохого, но файл процесса *B* не будет напечатан Пользователь, связанный с процессом *B* может в этой ситуации полдня описывать круги вокруг принтера, ожидая требуемой распечатки. Ситуации, в которых два (и более) процесса считывают или записывают данные одновременно и конечный результат зависит от того, какой из них был первым, называются **состояниями состязания**.

Основным способом предотвращения подобных проблем является **взаимное исключение** – запрет одновременной записи и чтения разделенных данных более, чем одним процессом. Это означает, что в тот момент, когда один процесс использует разделенные данные, другому процессу это делать будет запрещено. Выбор подходящей примитивной операции, реализующей взаимное исключение, является серьезным моментом разработки операционной системы.

Проблему исключения состояний состязания можно сформулировать на абстрактном уровне. Некоторый промежуток времени процесс занят внутренними расчетами и другими задачами, не приводящими к состояниям состязания. В другие моменты времени процесс обращается к совместно используемым данным или выполняет какое-то другое действие, которое может привести к состязанию. Часть программы, в которой есть обращение к совместно используемым данным, называется **критической областью** или **критической секцией**. Если нам удается

избежать одновременного нахождения двух процессов в критических областях, мы сможем избежать состязаний.

Для правильной совместной работы параллельных процессов и эффективного использования общих данных необходимо выполнение четырех условий:

– Два процесса не должны одновременно находиться в критических областях.

– В программе не должно быть предположений о скорости или количестве процессоров.

– Процесс, находящийся вне критической области, не может блокировать другие процессы.

– Невозможна ситуация, в которой процесс вечно ждет попадания в критическую область.

В абстрактном виде требуемое поведение процессов представлено на рисунке 3.1. Процесс *A* попадает в критическую область в момент времени T_1. Чуть позже, в момент времени T_2, процесс *B* пытается попасть в критическую область, но ему это не удается, поскольку в критической области уже находится процесс *A*, а два процесса не должны одновременно находиться в критических областях. Поэтому процесс *B* временно приостанавливается, до наступления момента времени T_3, когда процесс *A* выхолит из критической области. В момент времени T_4 процесс *B* также покидает критическую область, и мы возвращаемся в исходное состояние, когда ни одного процесса в критической области не было.

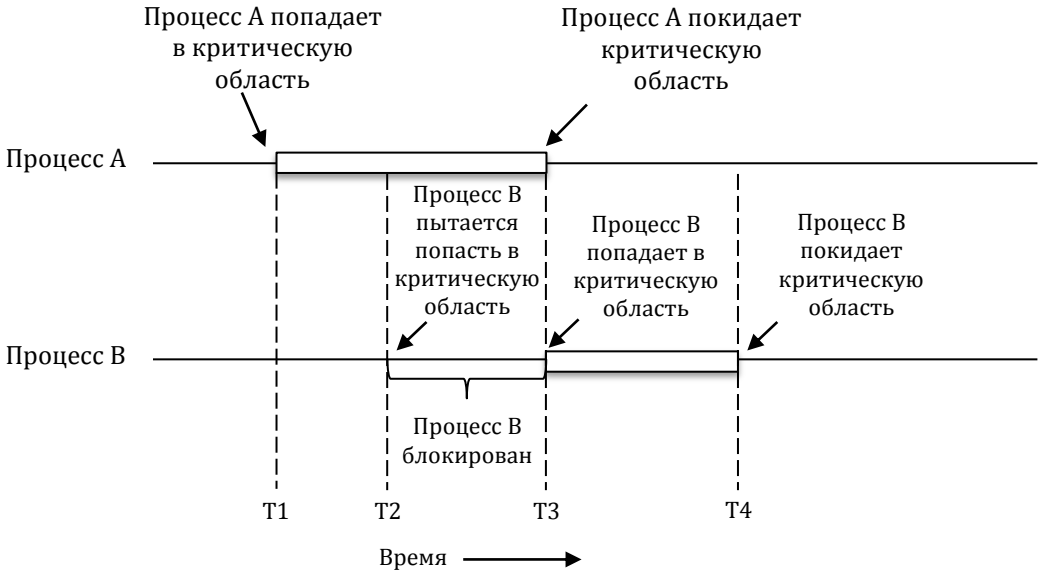


Рисунок 3.1 – Взаимное исключение с использованием критических областей

Существует несколько методов реализации взаимного исключения:

1. Запрещение прерываний при входе процесса в критическую область и разрешение по выходу из нее.

Это самое простое решение. Если прерывания запрещены, невозможно прерывание по таймеру. Поскольку процессор переключается с одного процесса на другой только по прерыванию, отключение прерываний исключает передачу процессора другому процессу. Таким образом, запретив прерывания, процесс может спокойно считывать и сохранять совместно используемые данные, не опасаясь вмешательства другого процесса.

И все же было бы неразумно давать пользовательскому процессу возможность запрета прерываний, Представьте себе, что процесс отключил все прерывания и в результате какого-либо сбоя не включил их обратно. Операционная система на этом может закончить свое существование. К тому же в многопроцессорной системе запрещение прерываний повлияет только на тот процессор, который выполнит инструкцию *disable*. Остальные процессоры продолжат работу и сохранят доступ к разделенным данным.

С другой стороны, для ядра характерно запрещение прерываний для некоторых команд при работе с переменными или списками. Возникновение прерывания в момент, когда, например, список готовых процессов находится в неопределенном состоянии, могло бы привести к состоянию состязания. Таким образом, запрет прерываний бывает полезным в самой операционной системе, но это решение неприемлемо в качестве механизма взаимного исключения для пользовательских процессов.

2. Переменные блокировки, которые процесс считывает в случае, если хочет попасть в критическую область.

Рассмотрим одну совместно используемую переменную блокировки, изначально равную 0. Если процесс хочет попасть в критическую область, он предварительно считывает значение переменной блокировки. Если переменная равна 0, процесс изменяет ее на 1 и входит в критическую область. Если же переменная равна 1, то процесс ждет, пока ее значение сменится на 0. Таким образом, 0 означает, что ни одного процесса в критической области нет, а 1 означает, что какой-либо процесс находится в критической области.

К сожалению, у этого метода те же проблемы, что и в примере с каталогом спулера. Представьте, что один процесс считывает переменную блокировки,

обнаруживает, что она равна 0, но прежде, чем он успевает изменить ее на 1, управление получает другой процесс, успешно изменяющий ее на 1. Когда первый процесс снова получит управление, он тоже заменит переменную блокировки на 1 и два процесса одновременно окажутся в критических областях.

Можно подумать, что проблема решается повторной проверкой значения переменной, прежде чем заменить ее, но это не так. Второй процесс может получить управление как раз после того, как первый процесс закончил вторую проверку, но еще не заменил значение переменной блокировки.

3. Строгое чередование, заключающееся в том, что считывается переменная, меняющая свое значение в случае выхода процесса из критической области.

Постоянная проверка значения переменной в ожидании некоторого значения называется **активным ожиданием**. Подобного способа следует избегать, поскольку он является бесцельной тратой времени процессора. Активное ожидание используется только в случае, когда есть уверенность в небольшом времени ожидания. Блокировка, использующая активное ожидание, называется **спин-блокировкой**.

Рассмотрим, как работает этот метод. Пусть целая переменная *turn*, изначально равная 0, отслеживает, чья очередь входить в критическую область. Вначале процесс 0 проверяет значение *turn*, считывает 0 и входит в критическую область. Процесс 1 также проверяет значение *turn*, считывает 0 и после этого входит в цикл, непрерывно проверяя, когда же значение turn будет равно 1.

Когда процесс 0 покидает критическую область, он меняет значение *turn* на 1, позволяя процессу 1 попасть в критическую область. Предположим, что процесс 1 быстро покидает свою критическую область, так что оба процесса теперь находятся вне критической области, и значение *turn* равно 0. Теперь процесс 0 выполняет весь цикл быстро, выходит из критической области и устанавливает значение *turn* равным 1. В этот момент значение *turn* равно 1, и оба процесса находятся вне критической области.

Неожиданно процесс 0 завершает работу вне критической области и возвращается к началу цикла. Но войти в критическую область он не может, поскольку значение *turn* равно 1 и процесс 1 находится вне критической области. Процесс 0 зависнет в своем цикле, ожидая, пока процесс 1 изменит значение *turn* на 0. Получается, что метод поочередного доступа к критической области не слишком удачен, если один процесс существенно медленнее другого.

Эта ситуация нарушает третье из сформулированных нами условий: один процесс блокирован другим, не находящимся в критической области. Возвратимся к примеру с каталогом спулера: если заменить критическую область процедурой считывания и записи в каталог спулера, процесс 0 не сможет послать файл на печать, поскольку процесс 1 занят чем-то другим.

Фактически этот метод требует, чтобы два процесса попадали в критические области строго по очереди. Ни один из них не сможет попасть в критическую область (например, послать файл на печать) два раза подряд. Хотя этот алгоритм и исключает состояния состязания, его нельзя рассматривать всерьез, поскольку он нарушает третье условие успешной работы двух параллельных процессов с совместно используемыми данными.

Существует различные решения, исправляющие этот недостаток. Однако все они в своей основе содержат метод активного ожидания. А такой метод бесцельно расходует время процессора и может иметь некоторые неожиданные последствия. Рассмотрим два процесса: *H*, с высоким приоритетом, и *L*, с низким приоритетом. Правила планирования в этом случае таковы, что процесс *H* запускается немедленно, как только он оказывается в состоянии ожидания. В какой-то момент, когда процесс *L* находится в критической области, процесс *H* оказывается в состоянии ожидания (например, он закончил операцию ввода-вывода). Процесс *H* попадает в состояние активного ожидания, но поскольку процессу *L* во время работающего процесса *H* никогда не будет предоставлено процессорное время, у процесса *L* не будет возможности выйти из критической области, и процесс *H* навсегда останется в цикле. Эту ситуацию иногда называют **проблемой инверсии приоритета**.

Теперь рассмотрим некоторые *примитивы межпроцессного взаимодействия*, применяющиеся вместо циклов ожидания, в которых лишь напрасно расходуется процессорное время. Эти примитивы блокируют процессы в случае запрета на вход в критическую область. Одной из простейших является пара примитивов *sleep* и *wakeup*. Примитив *sleep* – системный запрос, в результате которого вызывающий процесс блокируется, пока его не запустит другой процесс. У запроса *wakeup* есть один параметр – процесс, который следует запустить. Также возможно наличие одного параметра у обоих запросов – адреса ячейки памяти, используемой для согласования запросов ожидания и запуска.

В качестве примера использования этих примитивов рассмотрим проблему **производителя и потребителя**, также известную как проблема **ограниченного буфера**. Два процесса совместно используют буфер ограниченного размера. Производитель – это процесс, помещающий данные в буфер, а потребитель – считывающий их оттуда.

Трудности возникают, когда производитель хочет поместить данные в переполненный буфер. Для производителя решением является ожидание, пока потребитель полностью или частично не очистит буфер. Аналогично, если потребитель хочет забрать данные из буфера, а буфер пуст, потребитель уходит в состояние ожидания и выходит из него, как только производитель положит что-нибудь в буфер и разбудит его.

Это решение кажется достаточно простым, но оно приводит к состояниям состязания, как и пример с каталогом спулера. Было предложено использовать переменную *count,* которая предназначена для отслеживания количества элементов в буфере. Если максимальное число элементов, хранящихся в буфере, равно *N*, программа производителя должна проверить, не равно ли *N* значение *count* прежде, чем поместить в буфер следующую порцию данных. Если значение *count* равно *N*, то производитель уходит в состояние ожидания; в противном случае производитель помещает данные в буфер и увеличивает значение *count*.

Код программы потребителя прост: сначала проверить, не равно ли значение *count* нулю. Если равно, то уйти в состояние ожидания; иначе забрать порцию данных из буфера и уменьшить значение *count*. Каждый из процессов также должен проверять, не следует ли активизировать другой процесс, и в случае необходимости проделывать это.

Возникновение состояния состязания возможно, поскольку доступ к переменной *count* не ограничен. Может возникнуть следующая ситуация: буфер пуст, и потребитель только что считал значение переменной *count*, чтобы проверить, не равно ли оно нулю. В этот момент планировщик передал управление производителю, производитель поместил элемент в буфер и увеличил значение count, проверив, что теперь оно стало равно 1. Зная, что перед этим оно было равно 0, и потребитель находился в состоянии ожидания, производитель активизирует его с помощью вызова *wakeup.*

Но потребитель не был в состоянии ожидания, так что сигнал активизации пропал впустую. Когда управление перейдет к потребителю, он вернется к

считанному когда-то значению count, обнаружит, что оно равно 0, и уйдет и состояние ожидания. Рано или поздно производитель наполнит буфер и также уйдет в состояние ожидания. Оба процесса так и останутся в этом состоянии.

Суть проблемы в данном случае состоит в том, что сигнал активизации пришедший к процессу, не находящемуся в состоянии ожидания, пропадает. Если бы не это, проблемы бы не было. Чтобы этого избежать было предложено использовать **бит ожидания активизации**. Если сигнал активизации послан процессу, не находящемуся в состоянии ожидания, этот бит устанавливается. Позже, когда процесс пытается уйти в состояние ожидания, бит ожидания активизации сбрасывается, но процесс остается активным. Этот бит исполняет роль копилки сигналов активизации. Несмотря на то, что введение бита ожидания запуска спасло положение в этом примере, легко сконструировать ситуацию с несколькими процессами, в которой одного бита будет недостаточно. Мы можем добавить еще один бит, или 8, или 32, но это не решит проблему.

В связи с этим было предложено использовать целую переменную для подсчета сигналов запуска, сохраненных на будущее. Это так называемый **семафор**, значение которого может быть нулем (в случае отсутствия сохраненных сигналов активизации) или положительным числом, соответствующим количеству отложенных активизирующих сигналов.

Были предложены две операции, *down* и *up*. Операция *down* сравнивает значение семафора с нулем. Если его значение больше нуля, то операция *down* его уменьшает (то есть расходует один сигнал активации) и просто возвращает управление. Если значение равно нулю, то процедура *down* не возвращает управление процессу, а процесс переводится в состояние ожидания. Все операции проверки значении семафора, его изменения и перевода процесса в состояние ожидания выполняются как единое и неделимое **элементарное действие**. Этим гарантируется, что после начала операции ни один процесс не получит доступа к семафору до окончания или блокирования операции.

Операция *up* увеличивает значение семафора. Если с этим семафором связаны один или несколько ожидающих процессов, которые не могут завершить операцию, более раннюю, чем *down*, один из них выбирается системой и ему разрешается завершить свою операцию *down*. Таким образом, после операции *up*, примененной к семафору, связанному с несколькими ожидающими процессами, значение семафора так и останется равным нулю, но число ожидающих процессов уменьшится на

единицу. Операция увеличения значения семафора и активизации процесса тоже неделима.

Однако такой механизм может приводить к взаимоблокировкам в случае заполнения буфера в тех ситуациях, когда две процедуры *down* в программе производителя меняются местами. Поэтому был предложен примитив синхронизации более высокого уровня, называемый монитором.

Монитор – это набор процедур, переменных и других структур данных, объединенных в особый модуль или пакет. Процессы могут вызвать процедуры монитора, но у процедур, объявленных вне монитора, нет прямого доступа к внутренним структурам данных мониторов.

Реализации взаимных исключений способствует важное свойство монитора: при обращении к нему в любой момент времени активным может быть только один процесс. Мониторы являются структурным компонентом языка программирования, поэтому компилятор знает, что обрабатывать вызовы процедур монитора следует иначе, чем вызовы остальных процедур. Обычно при вызове процедуры монитора первые несколько команд процедуры проверяют, нет ли в мониторе активного процесса. Если активный процесс есть, вызывающему процессу придется подождать, в противном случае запрос удовлетворяется.

Необходимо отметить, что и семафоры, и мониторы были разработаны для решения задачи взаимного исключения в системе с одним или несколькими процессорами, имеющими доступ к общей памяти. Поэтому они не подходят для работы в системах с несколькими процессами с собственной памятью у каждого, а также для реализации обмена информации между компьютерами.

Для этих целей используется метод межпроцессного взаимодействия, называемый **передача сообщений**. Он использует два примитива: *send* и *receive*. Первый запрос посылает сообщение заданному адресату, а второй получает сообщение от указанного источника (или от любого источника, если это не имеет значения). Если сообщения нет, второй запрос блокируется до поступления сообщения или немедленно возвращает код ошибки.

Если производитель будет работать быстрее, чем потребитель, то все сообщения будут ожидать потребителя в заполненном виде. При этом производитель блокируется в ожидании пустого сообщения. Если наоборот, то все сообщения будут пустыми, а потребитель будет блокирован в ожидании полного сообщения.

Эта проблема может быть решена различными путями. Например, можно присвоить каждому процессу уникальный адрес и адресовать сообщение непосредственно процессам. Другой подход состоит в использовании структуры данных, называемой **почтовым ящиком**, то есть буфером для определенного количества сообщений, тип которых задается при создании ящика. При использовании почтовых ящиков в качестве параметров адреса процедур *send* и *receive* задаются почтовые ящики, а не процессы. Если процесс пытается послать сообщение в полный почтовый ящик, ему приходится подождать, пока хотя бы одно сообщение не будет удалено из ящика.

В задаче производителя и потребителя оба они создадут почтовые ящики, достаточно большие, чтобы хранить *N* сообщений. Производитель будет посылать сообщения с данными в почтовый ящик потребителя, а потребитель будет посылать пустые сообщения в почтовый ящик производителя. С использованием почтовых ящиков метод буферизации очевиден: в почтовом ящике получателя хранятся сообщения, которые были посланы процессу-получателю, но еще не получены.

3.2. Классические проблемы межпроцессного взаимодействия

В литературе по ОС описано несколько классических проблем межпроцессного взаимодействия. Например, проблемы обедающих философов, читателей и писателей, а также – спящего брадобрея.

Рассмотрим *проблему обедающих философов*: пять философов сидят за круглым столом (рисунок 3.2), у каждого есть тарелка со спагетти, которые настолько скользкие, что каждому философу нужно две вилки, чтобы с ними управиться. Между каждыми двумя тарелками лежит одна вилка. Жизнь философа состоит из чередующихся периодов поглощения пищи и размышлений. Когда философ голоден, он пытается получить 2 вилки, левую и правую, в любом порядке. Вопрос состоит в следующем: можно ли написать алгоритм, который моделирует эти действия для каждого философа и никогда не застревает.

Для решения этой проблемы используется массив для отслеживания душевного состояния каждого философа – ест, размышляет или голодает (пытаясь получить вилки). Философ может начать есть только, если ни один из его соседей не ест. В программе используется массив семафоров, по одному на каждого философа, чтобы блокировать голодных философов, если их вилки заняты.

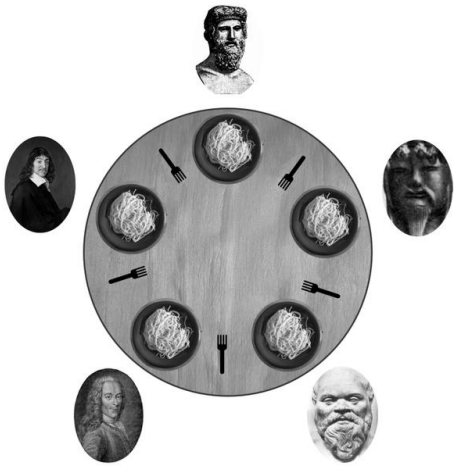


Рисунок 3.2 – Проблема обедающих философов

Проблема обедающих философов полезна для моделирования процессов, соревнующихся за монопольный доступ к ограниченному количеству ресурсов, например, к устройствам ввода-вывода.

Другой известной задачей является *проблема читателей и писателей*, моделирующая доступ к базе данных. Представьте себе базу данных бронирования билетов на самолет, к которой пытается получить доступ множество процессов. Можно разрешить одновременное считывание данных из базы, но если процесс записывает информацию в базу, доступ остальных процессов должен быть прекращен, даже доступ на чтение.

Эта задача решается таким образом, что первый читающий процесс выполняет операцию *down* на семафоре, чтобы получить доступ к базе данных. Последующие читатели просто увеличивают значение счетчика. По мере ухода читателей из базы значение счетчика уменьшается, и последний читающий процесс выполняет на семафоре операцию *up*, позволяя блокированному пишущему процессу получить доступ к базе.

Здесь нужно иметь ввиду один момент. Пишущий процесс не получит доступ к базе, пока в ней есть хоть один читающий процесс. Однако любой читающий процесс будет иметь доступ на чтение. В результате возможна ситуация, при которой пишущий процесс никогда не сможет попасть в базу. Чтобы этого избежать, нужно сделать так, чтобы, как только появляется ожидающий доступа пишущий процесс, все вновь поступающие читающие процесс становились в очередь за ним.

Действие еще одной классической проблемной ситуации межпроцессного взаимодействия разворачивается в парикмахерской (*проблема спящего брадобрея*) (рисунок 3.3). Предположим, что там есть один брадобрей, его кресло и *n* стульев для

посетителей. Если желающих воспользоваться его услугами нет, то брадобрей спит в своем кресле. Если приходит клиент, то он должен разбудить брадобрея. Если клиент приходит и видит, что брадобрей занят, то он либо садится на стул, (если место есть), либо уходит (если места нет). Необходимо запрограммировать брадобрея и посетителей так, чтобы избежать состояния состязания.

Рисунок 3.3 – Проблема спящего брадобрея

Конфликтные ситуации связаны с тем фактом, что действия и парикмахера, и клиента (проверка приёмной, вход в парикмахерскую, занятие места в приёмной, и т. д.) занимают неизвестное количество времени и/или могут происходить одновременно. Например, клиент может войти и заметить, что парикмахер работает, тогда он идет в приёмную. Пока он идет, парикмахер заканчивает стрижку, которую он делает и идет, чтобы проверить приемную, причём делает это быстрее направляющегося туда клиента. Так как в приёмной пока ещё никого нет (клиент ещё не дошел), он возвращается к своему месту и спит. Парикмахер теперь ждет клиента, а клиент ждет парикмахера. В другом примере два клиента могут прибыть в то же самое время, когда в приемной есть единственное свободное место. Они замечают, что парикмахер работает, идут в приёмную, и оба пытаются занять единственный стул.

Для решения этой задачи используется три семафора – для подсчета ожидающих посетителей; количество брадобреев (0 или 1), простаивающих в ожидании клиента; для реализации взаимного исключения. Также используется переменная, предназначенная для подсчета ожидающих клиентов.

Когда брадобрей приходит, он засыпает, пока не придет первый клиент. Приходя в парикмахерскую, посетитель запрашивает доступ для входа в

критическую область. Если вслед за ним появится еще один посетитель, то он будет ждать, пока первый не освободит доступ. Если свободный стул есть, то посетитель увеличивает значение целочисленной переменной. Затем он выполняет процедуру *up* на первом семафоре, предназначенном для подсчета ожидающих посетителей. По окончании стрижки посетитель выходит из процедуры и покидает парикмахерскую. Здесь нет цикла посетителя, так как каждого стригут только один раз. Цикл брадобрея существует, и брадобрей пытается найти следующего посетителя. Если ему это удается, то он стрижет его, если нет, то – засыпает.

3.3. Введение в планирование

Когда компьютер работает в многозадачном режиме, на нем могут быть активными процессы, пытающиеся одновременно получить доступ к процессору. Эта ситуация возникает при наличии двух и более процессов в состоянии готовности. Если доступен только один процессор, то необходимо выбирать между процессами. Отвечающая за это часть ОС называется **планировщиком**, а используемый алгоритм – **алгоритмом планирования**.

Когда процессору нужно выбрать между процессом, перерисовывающим экран после того, как пользователь закрыл окно приложения, и процессом, отсылающим почту, впечатление пользователя о реакции компьютера будет существенно зависеть от этого выбора. Ведь если перерисовка экрана во время отправки почты займет 2 с, пользователь решит, что система очень медленная, тогда как двухсекундная отсылка почты даже не будет замечена. В этом случае планирование процессов очень важно.

Помимо правильного выбора следующего процесса, планировщик также должен заботиться об эффективном использовании процессора, поскольку переключение между процессами требует затрат. Во-первых, необходимо переключиться из режима пользователя в режим ядра. Затем следует сохранить состояние текущего процесса, включая сохранение регистров в таблице процессов, чтобы их можно было загрузить заново позже. В большинстве систем также необходимо сохранить карту памяти (то есть биты-признаки обращения к страницам памяти). Затем нужно выбрать следующий процесс, запустив алгоритм планирования. После этого необходимо перезапустить блок управления памятью с картой памяти нового процесса. И, наконец, нужно запустить новый процесс. Помимо этого, переключение между процессами обычно делает бесполезной

информацию, содержащуюся в кэше памяти, что приводит к двойной перезагрузке кэша из основной памяти (при входе в ядро и при выходе из ядра). Все это занимает много процессорного времени, особенно при слишком частом переключении между процессами, поэтому в этом рекомендуется соблюдать умеренность.

Необходимо отметить, что все процессы делятся на ограниченные возможностями процессора (те, которые большую часть времени заняты вычислениями) и ограниченные возможностями устройств ввода-вывода. Основным параметром здесь является не время ожидания, а время вычислений. Процессы, ограниченные возможностями устройств ввода-вывода, называются так, поскольку загруженность процессора вычислениями между запросами ввода-вывода мала, а не потому, что время выполнения ввода-вывода особо велико. На считывание блока данных с диска уходит одно и то же время, не зависящее от времени, затрачиваемого на обработку этих данных.

При этом с увеличением скорости процессоров процессы становятся все более ограниченными возможностями устройств ввода-вывода. Это связано с тем, что процессоры совершенствуются существенно быстрее, чем диски. Таким образом, планирование процессов, ограниченных возможностями устройств ввода-вывода, будет иметь большее значение в будущем. В таких процессах следует избегать простоя относительно медленных дисков.

Ключевым вопросом планирования является выбор момента принятия решения. Существует множество ситуаций, в которых необходимо планирование. Во-первых, когда создается новый процесс, необходимо решить, какой процесс запустить: родительский или дочерний. Во-вторых, когда процесс завершает работу, необходимо выбрать, какой процесс выполнять следующим. В-третьих, когда процесс блокируется на операции ввода-вывода, семафоре или по какой-либо другой причине, необходимо выбрать и запустить другой процесс. В-четвертых, необходимость планирования может возникнуть при появлении прерывания ввода-вывода.

Все алгоритмы планирования могут быть разделены на две группы: **алгоритмы без переключений (неприоритетные)** – выбирают процесс и позволяют ему работать до блокировки или пока он сам не отдаст процессор и **алгоритмы планирования с переключениями (приоритетные)** – выбирают процесс и позволяют ему работать определенное время. Приоритетное

планирование требует прерываний по таймеру. При его отсутствии возможно только неприоритетное планирование.

В различных средах требуются различные алгоритмы планирования. Это связано с тем, что различные операционные системы и различные приложения ориентированы на разные задачи. Другими словами, то, для чего следует оптимизировать планировщик, различно в разных системах. Можно выделить три среды:

- системы пакетной обработки данных;
- интерактивные системы;
- системы реального времени.

В *системах пакетной обработки* данных нет пользователей, сидящих за терминалами и ожидающих ответа. В таких системах приемлемы алгоритмы без переключений или с переключениями, но с большим временем, отводимым каждому процессу. Такой метод уменьшает количество переключений между процессами и улучшает эффективность.

В *интерактивных системах* необходимы алгоритмы планирования с переключениями, чтобы предотвратить захват процессора одним процессом.

В *системах реального времени* приоритетность, как это ни странно, не всегда обязательна, поскольку процессы знают, что их время ограничено и быстро выполняют работу, а затем блокируются. Отличие от интерактивных систем в том, что в системах реального времени работают только программы, предназначенные для содействия конкретным приложениям. Интерактивные системы универсальны. В них могут работать произвольные программы, не сотрудничающие друг с другом и даже враждебные по отношению друг к другу.

Рассмотрим вкратце алгоритмы планирования, применяемые *в системах пакетной обработки данных*. В таких системах применяются следующие алгоритмы:

– "Первым пришел – первым обслужен". Это самый простой из алгоритмов планирования. Процессам предоставляется доступ к процессору в том порядке, в котором они его запрашивают. Чаще всего формируется единая очередь ждущих процессов. Как только появляется первая задача, она немедленно запускается и работает столько, сколько необходимо. Остальные задачи ставятся в конец очереди. Когда текущий процесс блокируется, запускается следующий в очереди, а когда блокировка снимается, процесс попадает в конец очереди.

– "Кратчайшая задача – первая". Этот алгоритм предполагает, что временные отрезки работы известны заранее. Если в очереди есть несколько одинаково важных задач, планировщик выбирает первой самую короткую задачу.

– Наименьшее оставшееся время выполнения. В соответствии с этим алгоритмом планировщик каждый раз выбирает процесс с наименьшим оставшимся временем выполнения. В этом случае также необходимо заранее знать время выполнения задач. Когда поступает новая задача, ее полное время выполнения сравнивается с оставшимся временем выполнения текущей задачи. Если время выполнения новой задачи меньше, текущий процесс приостанавливается и управление передается новой задаче. Эта схема позволяет быстро обслуживать короткие запросы.

Теперь обратимся к алгоритмам планирования, используемым *в интерактивных системах*. Все они также могут использоваться в качестве планировщика процессора в системах пакетной обработки. К алгоритмам для интерактивных систем относятся:

– Циклическое планирование. Это один из наиболее старых, простых, справедливых и часто используемых алгоритмов планирования. Каждому процессу предоставляется некоторый интервал времени процессора, так называемый **квант** времени. Если к концу кванта времени процесс все еще работает, он прерывается, а управление передается другому процессу. Разумеется, если процесс блокируется или прекращает работу раньше, переход управления происходит в этот момент. Реализация циклического планирования проста. Планировщику нужно всего лишь поддерживать список процессов в состоянии готовности. Когда процесс исчерпал свой лимит времени, он отправляется в конец списка. Важным моментом в этом алгоритме является длина кванта времени. Слишком малый квант приведет к частому переключению процессов и небольшой эффективности, но слишком большой квант может привести к медленному реагированию на короткие интерактивные запросы. Значение кванта около 20-50 мс часто является разумным компромиссом.

– Приоритетное планирование. В циклическом алгоритме планирования есть важное допущение о том, что все процессы равнозначны. В ситуации компьютера с большим числом пользователей это может быть не так. Основная идея проста: каждому процессу присваивается приоритет, и управление передается готовому к работе процессу с самым высоким приоритетом. Чтобы предотвратить бесконечную

работу процессов с высоким приоритетом, планировщик может уменьшать приоритет процесса с каждым тактом часов (то есть при каждом прерывании по таймеру). Если в результате приоритет текущего процесса окажется ниже, чем приоритет следующего процесса, произойдет переключение. Возможно предоставление каждому процессу максимального отрезка времени работы, Как только время кончилось, управление передается следующему по приоритету процессу.

– "Самый короткий процесс – следующий". Этот алгоритм основывается на оценке длины процесса, базирующейся на предыдущем поведении процесса. При этом запускается процесс, у которого оцененное время самое маленькое.

– Гарантированное планирование. Если процессором пользуются n пользователей, им будет предоставлено $1/n$ мощности процессора. И в системе с одним пользователем и n запущенными процессами каждому достанется $1/n$ циклоп процессора.

– Лотерейное планирование. В основе алгоритма лежит раздача процессам лотерейных билетов на доступ к различным ресурсам, в том числе и к процессору. Когда планировщику необходимо принять решение, выбирается случайным образом лотерейный билет, и его обладатель получает доступ к ресурсу. Что касается доступа к процессору, "лотерея" может происходить 50 раз в секунду, и победитель получает 20 мс времени процессора. Более важным процессам можно раздать дополнительные билеты, чтобы увеличить вероятность выигрыша. Каждый процесс получит процент ресурсов, примерно равный проценту имеющихся у него билетов.

– Справедливое планирование. До сих пор предполагалось, что каждый процесс управляется независимо от того, кто его хозяин. Поэтому если пользователь 1 создаст 9 процессов, а пользователь 2 – 1 процесс, то с использованием циклического планирования или в случае равных приоритетов пользователю 1 достанется 90 % процессора, а пользователю 2 – всего 10. Чтобы избежать подобных ситуаций, некоторые системы обращают внимание на хозяина процесса перед планированием. В такой модели каждому пользователю достается некоторая доля процессора, и планировщик выбирает процесс в соответствии с этим фактом. Если в нашем примере каждому из пользователей было обещано по 50 % процессора, то им достанется по 50 % процессора, независимо от количества процессов.

Алгоритмы планирования в системах реального времени будут рассмотрены позднее в теме "Мультимедийные операционные системы".

4. ВЗАИМОБЛОКИРОВКИ

4.1. Основные понятия

В компьютерных системах существует большое количество ресурсов, каждый из которых в конкретный момент времени может использоваться только одним процессом. К таким ресурсам относятся устройства ввода-вывода, элементы внутренних таблиц системы и т.д. Поэтому все ОС обладают способностью предоставлять процессу эксклюзивный доступ (по крайней мере, временный) к определенным ресурсам.

Часто для выполнения прикладных задач процесс нуждается в исключительном доступе не к одному, а к нескольким ресурсам. Предположим, есть процессы *A* и *B*, каждый из которых должен записать отсканированный документ на компакт-диск. Но запрограммированы они по-разному – процесс *A* сначала получает доступ к сканеру, а процесс *B* – к устройству записи компакт-дисков. Затем процесс *A* обращается к приводу CD-ROM, но запрос отклоняется, пока привод занят процессом *B*. При этом процесс *B* не освобождает устройство, одновременно пытаясь получить доступ к сканеру. В результате возникает ситуация, когда оба процесса блокируют друг другу и будут вечно оставаться в этом состоянии. Такая ситуация называется **тупиком**, **тупиковой ситуацией** или **взаимоблокировкой** (рисунок 4.1).

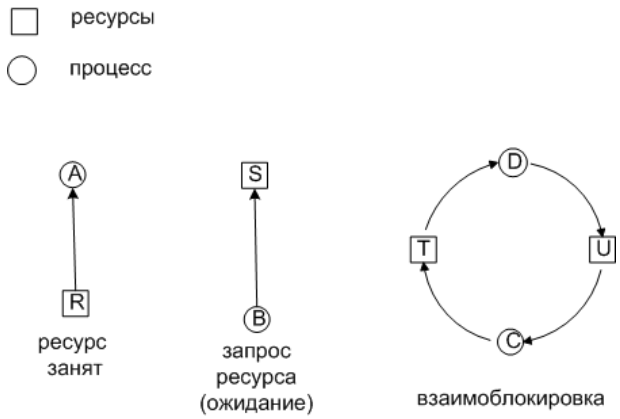


Рисунок 4.1 – Условные обозначения взаимоблокировок. Графы ресурсов

Взаимоблокировки могут возникать между различными машинами. Во многих офисах есть локальная сеть, включающая в себя множество компьютеров. Часто такие устройства, как сканеры, устройства для записи компакт-дисков, принтеры и накопители на магнитной лепте, присоединены к сети как ресурсы совместного доступа, то есть доступны любому пользователю на любой машине. Если эти ресурсы позволяется резервировать удаленно (то есть с домашней машины пользователя), может возникнуть аналогичный описанному выше вид тупиковых

ситуаций. Более сложные ситуации могут стать причиной тупиков, вовлекающих три, четыре и более устройств пользователей.

Взаимоблокировки могут произойти во множестве других ситуаций помимо запросов выделенных устройств ввода-вывода. В системах баз данных программа может оказаться вынужденной заблокировать несколько записей, чтобы избежать состояния конкуренции. Если процесс *A* блокирует запись *R*1, процесс *B* блокирует запись *R*2, а затем каждый процесс попытается заблокировать чужую запись, мы также окажемся в тупике. Таким образом, взаимоблокировки появляются при работе как с аппаратными, так и с программными ресурсами.

Таким образом, **взаимоблокировка** – ситуация, когда каждый процесс из группы ожидает события, которое может вызвать только другой процесс из той же группы.

Так как все процессы находятся в состоянии ожидания, ни один из них не будет причиной какого-либо события, которое могло бы активировать любой другой процесс в группе, и все процессы продолжают ждать до бесконечности. В этой модели мы предполагаем, что процессы имеют только один поток и что нет прерываний, способных активизировать заблокированный процесс. Условие отсутствия прерываний необходимо, чтобы предотвратить ситуацию, когда тот или иной заблокированный процесс активизируется, скажем, по сигналу тревоги и затем приводит к событию, которое освободит другие процессы в группе.

В большинстве случаев событием, которого ждет каждый процесс, является возврат какого-либо ресурса, в данный момент занятого другим участником группы. Другими словами, каждый участник в группе процессов, зашедших в тупик, ждет доступа к ресурсу, принадлежащему заблокированному процессу. Ни один из процессов не может работать, ни один из них не может освободить какой-либо ресурс и ни один из них не может возобновиться. Количество процессов и количество и вид ресурсов, имеющихся и запрашиваемых, здесь не важны. Результат остается тем же самым для любого вида ресурсов, аппаратных и программных.

4.2. Выгружаемые и невыгружаемые ресурсы

Итак, что же такое ресурсы? Система может зайти в тупик, когда процессам предоставляются исключительные права доступа к устройствам, файлам и т. д. Чтобы обобщить все объекты предоставления доступа, вводится понятие ресурса. Ресурсом может быть аппаратное устройство (например, накопитель на магнитной

ленте) или часть информации (например, закрытая запись в базе данных). В компьютере существует масса различных ресурсов, к которым могут происходить обращения. Кроме того, в системе может оказаться несколько идентичных экземпляров какого-либо ресурса, например три накопителя на магнитных лентах. Если в системе есть несколько экземпляров ресурса, то в ответ на обращение к нему может предоставляться любая из доступных копий. Короче говоря, **ресурс** – это все то, что может использоваться только одним процессом в любой момент времени.

Ресурсы бывают двух типов: выгружаемые и невыгружаемые.

Выгружаемый ресурс – ресурс, который можно безболезненно забирать у владеющего им процесса. Образцом такого ресурса является память. Пусть процесс *A* запрашивает и получает принтер, затем начинает вычислять данные для печати. Еще не закончив расчеты, он превышает свой квант времени и выгружается на диск в область подкачки.

Теперь работает процесс *B* и безуспешно пытается обратиться к принтеру. Это потенциально тупиковая ситуация, потому что процесс *A* использует принтер, а процесс *B* – память, и ни один из них не может продолжать работу без ресурса, удерживаемого другим. К счастью, можно выгрузить память у процесса *B*, переместив его на диск в область подкачки и скачав с диска в память процесс *A*. В результате взаимоблокировки не происходит.

Невыгружаемый ресурс – ресурс, который нельзя забрать от текущего владельца, не уничтожив результаты вычислений. Так, если в момент записи компакт-диска внезапно отнять у процесса устройство для записи, то в результате мы получим испорченный компакт-диск. Таким образом, взаимоблокировки касаются только невыгружаемых ресурсов.

Последовательность событий, необходимых для использования ресурса, включает в себя:

1. Запрос ресурса.
2. Использование ресурса.
3. Возврат ресурса.

Если ресурс недоступен, когда он требуется, то запрашивающий его процесс вынужден ждать. В некоторых операционных системах при неудачном обращении к ресурсу процесс автоматически блокируется и возобновляется только после того, как ресурс становится доступным. В других системах запрос ресурса, получивший

отказ, возвращает код ошибки, тогда вызывающий процесс может подождать немного и повторить попытку заново.

Процесс, чье обращение к ресурсу оказалось неудачным, обычно дальше попадает в короткий цикл: запрос ресурса, затем режим ожидания, потом очередная попытка. Хотя этот процесс не блокирован, он во всех смыслах ведет себя, как заблокированный, поскольку не может выполнить никакой полезной работы.

Истинная природа запросов ресурсов сильно зависит от системы. В некоторых системах существует системный вызов *request*, позволяющий процессам запрашивать ресурсы явно. В других случаях единственный вид ресурсов, известных операционной системе, – это специальные файлы, которые в каждый данный момент времени могут открыть только один процесс. Они открываются с помощью обычного вызова *open*. Если файл уже используется, вызывающая программа блокируется до тех пор, пока текущий владелец файла не закроет его.

4.3. Обнаружение и устранение взаимоблокировок

Для возникновения ситуации взаимоблокировки должны выполняться следующие условия:

1. Условие взаимного исключения – каждый ресурс в данный момент или отдан ровно одному процессу, или доступен.
2. Условие удержания и ожидания – процессы, в данный момент удерживающие полученные ранее ресурсы, могут запрашивать новые ресурсы.
3. Условие отсутствия принудительной выгрузки ресурса – у процесса нельзя принудительным образом забрать ранее полученные ресурсы. Он должен сам освободить их.
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 выполниться все эти четыре условия, Если хоть одно из них отсутствует, тупиковая ситуация невозможна.

Для того чтобы бороться с взаимоблокировками необходимо каким-то образом их моделировать. Для моделирования взаимоблокировок используются направленные графы, предложенные Холтом. Графы имеют два вида узлов: процессы, показанные кружочками, и ресурсы, нарисованные квадратиками. Ребро,

направленное от узла ресурса (квадрат) к узлу процесса (круг), означает, что ресурс ранее был запрошен процессом, получен и в данный момент используется этим процессом (см. рисунок 4.1). Ребро, направленное от процесса к ресурсу, означает, что процесс в данный момент блокирован и находится в состоянии ожидания доступа к этому ресурсу. Цикл в графе означает наличие взаимоблокировки, циклично включающей процессы и ресурсы (предполагается, что в системе есть по одному ресурсу каждого вида). Процесс *C* ожидает ресурс *T*, удерживаемый в настоящее время процессом *D*. Процесс *D* вовсе не намеревается освобождать ресурс *T*, потому что он ждет ресурс *U*, используемый процессом *C*. Оба процесса будут ждать до бесконечности.

Графы ресурсов являются инструментом, позволяющим нам увидеть, станет ли заданная последовательность запросов/возвратов ресурсов причиной взаимоблокировки. Мы всего лишь шаг за шагом осуществляем запросы и возвраты ресурсов и после каждого шага проверяем граф на содержание циклов. Если они есть, мы зашли в тупик; если нет, значит, взаимоблокировки тоже нет.

При столкновении с взаимоблокировками используются четыре стратегии.

1. Пренебрежение проблемой в целом (страусовый алгоритм). Если вы проигнорируете проблему, возможно, затем она проигнорирует вас.

2. Обнаружение и восстановление. Позволить взаимоблокировке произойти, обнаружить ее и предпринять какие-либо действия.

3. Динамическое избежание тупиковых ситуаций с помощью аккуратного распределения ресурсов.

4. Предотвращение с помощью структурного опровержения одного из четырех условий, необходимых для взаимоблокировки.

Рассмотрим по очереди эти алгоритмы

Страусовый алгоритм. Самым простым подходом является "страусовый алгоритм": воткните голову в песок и притворитесь, что проблема вообще не существует. Различные люди отзываются об этой стратегии по-разному. Математики считают ее полностью неприемлемой и говорят, что взаимоблокировки нужно предотвращать любой ценой. Инженеры спрашивают, как часто встает подобная проблема, как часто система попадает в аварийные ситуации по другим причинам и насколько серьезны последствия взаимоблокировок. Если взаимоблокировки случаются в среднем один раз в пять лет, а сбои операционной системы, ошибки компилятора и поломки компьютера из-за неисправности

аппаратуры происходят раз в неделю, то большинство инженеров не захотят добровольно уступать в производительности и удобстве для того, чтобы ликвидировать возможность взаимоблокировок. Кроме того, большинство операционных систем потенциально страдают от взаимоблокировок, которые даже не обнаруживаются, не говоря уже об автоматическом выходе из тупика.

Суммарное количество процессов в системе определяется количеством записей в таблице процесса. Таким образом, ячейки таблицы процесса являются ограниченным ресурсом. Если системный вызов *fork* получает отказ, потому что таблица целиком заполнена, разумно будет, что программа, вызывающая *fork*, подождет какое-то время и повторит попытку.

Теперь предположим, что система UNIX имеет 100 ячеек процессов. Работают десять программ, каждой необходимо создать 12 (под) процессов. После образования каждым процессом девяти процессов 10 исходных и 90 новых процессов заполнят таблицу целиком. Теперь каждый из десяти исходных процессов попадает в бесконечный цикл, состоящий из попыток разветвления и отказов, то есть возникает взаимоблокировка. Вероятность того, что произойдет подобное, минимальна, но это *могло бы* случиться. Должны ли мы отказаться от процессов и вызова *fork*, чтобы устранить данную проблему?

Максимальное количество открытых файлов также ограниченно размером таблицы, следовательно, когда таблица заполняется целиком, возникает та же самая проблема. Пространство для подкачки файлов на диск является еще одним ограниченным ресурсом. Фактически почти каждая таблица в операционной системе представляет собой ресурс, имеющий пределы. Должны ли мы упразднить их все из-за того, что может произойти ситуация, когда в группе из n процессоров каждый может потребовать $1/n$ от целого, а затем попытаться получить еще часть?

Большая часть операционных систем, включая UNIX и Windows, игнорируют эту проблему. Они исходят из предположения, что большинство пользователей скорее предпочтут иметь дело со случающимися время от времени взаимоблокировками, чем с правилом, по которому всем пользователям разрешается только один процесс, один открытый файл и т. д. Если бы можно было легко устранить взаимоблокировки, не возникло бы столько разговоров на эту тему. Сложность заключается в том, что цена достаточно высока, и в основном она, как мы вскоре увидим, исчисляется в наложении неудобных ограничений на процессы. Таким образом, мы столкнулись с неприятным выбором между удобством и

корректностью и множеством дискуссий о том, что более важно и для кого. При всех этих условиях трудно найти верное решение.

Вторая техника представляет собой **обнаружение и восстановление**. При использовании этого метода система не пытается предотвратить попадание в тупиковые ситуации. Вместо этого она позволяет взаимоблокировке произойти, старается определить, когда это случилось, и затем совершает некие действия по возврату системы к состоянию, имевшему место до того, как система попала в тупик.

Обнаружение взаимоблокировок основано на определенных алгоритмах, различных для случаев, когда имеется по одному ресурсу каждого типа или по несколько. Если имеется один ресурс каждого типа, то строится граф ресурсов и определяется, содержит ли он циклы. Если содержит, то есть взаимоблокировка (рисунок 4.2).

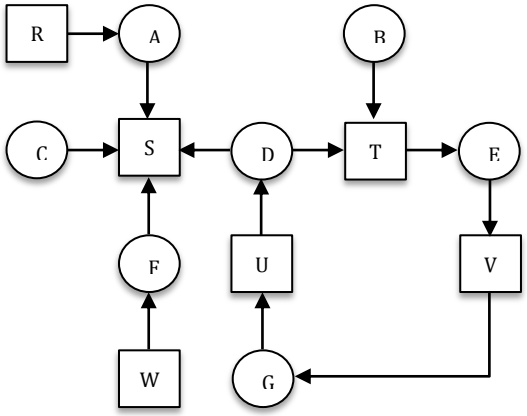


Рисунок 4.2 – Граф ресурсов при наличии блокировки

Если имеется несколько ресурсов одного типа, то строятся матрицы существующих ресурсов, доступных ресурсов, текущего распределения и запросов (рисунок 4.3). Путем их сравнения и определяется, можно ли выйти из состояния взаимоблокировки или нельзя.

Существующие ресурсы
$(E_1 \quad E_2 \quad E_3 \quad \dots \quad E_m)$

Матрица текущего распределения

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

Строка *n* в данный момент предоставлена процессу *n*

Доступные ресурсы
$(A_1 \quad A_2 \quad A_3 \quad \dots \quad A_m)$

Матрица текущего распределения

$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$

Строка *2* нужна процессу *2*

Рисунок 4.3 – Структуры данных, необходимые для алгоритма обнаружения взаимоблокировок

Пример использования матриц ресурсов показан на рисунке 4.4.

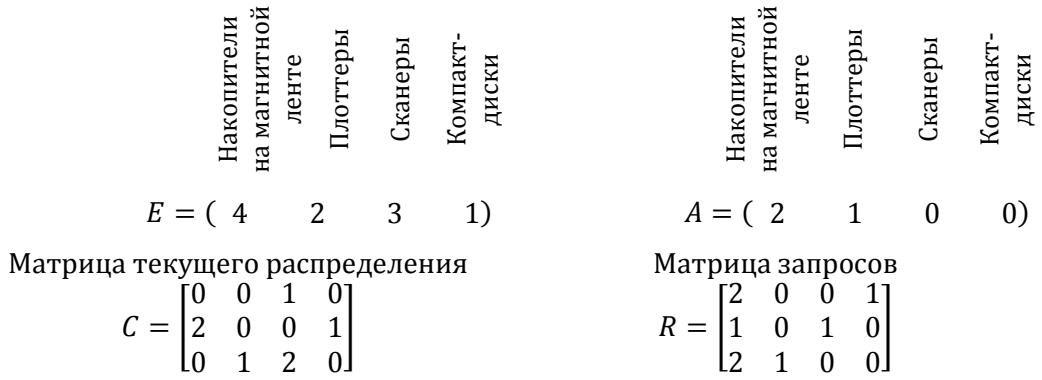


Рисунок 4.4 – Пример использования алгоритма обнаружения взаимоблокировок

Здесь возникает вопрос – когда нужно искать возникновение взаимоблокировок? Можно проверять систему каждый раз, когда запрашивается очередной ресурс. Но это будет слишком загружать процессор. Есть альтернативные подходы – проверять наличие взаимоблокировок каждые n минут или, например, когда степень занятости процессора меньше некоторого граничного значения. Учет загрузки процессора имеет смысл, потому что при достаточно большом количестве заблокированных процессов работоспособных процессов в системе останется немного, и процессор часто будет незанятым.

Предположим, что наш алгоритм обнаружения взаимоблокировок закончился успешно и нашел тупик. Что дальше? Необходимы методы для восстановления и получения в итоге снова работающей системы.

Существует 3 основных способа ликвидации взаимоблокировок:

1. Восстановление при помощи *принудительной выгрузки ресурса*.

Иногда можно временно отобрать ресурс у его текущего владельца и отдать его другому процессу. Во многих случаях требуется ручное вмешательство, особенно в операционных системах пакетной обработки, работающих на мэйнфреймах.

Например, чтобы забрать лазерный принтер у использующего его процесса, оператор может взять все уже отпечатанные листы и сложить их в стопку, соблюдая последовательность их появления из принтера. Затем процесс можно приостановить (пометить как неработоспособный). В этот момент принтер можно предоставить другому процессу. Когда он закончит работу, можно сложить стопку напечатанных листов обратно на выходной поднос принтера и возобновить первоначальный процесс,

Способность забирать ресурс у процесса, отдавать его другому процессу и затем возвращать назад так, что исходный процесс этого не замечает, в

значительной мере зависит от свойств ресурса. Выйти из тупика таким образом зачастую трудно или невозможно. Выбор приостанавливаемого процесса главным образом зависит от того, какой процесс владеет ресурсами, которые легко могут быть у него отняты.

2. Восстановление через *откат*.

Если разработчики системы и машинные операторы знают о том, что есть вероятность появления взаимоблокировок, они могут организовать работу таким образом, чтобы процессы периодически создавали контрольные точки. Создание процессом контрольной точки означает, что состояние процесса записывается в файл, в результате чего впоследствии процесс может быть возобновлен из этого файла. Контрольные точки содержат не только образ памяти, но и состояние ресурсов, то есть информацию о том, какие ресурсы в данный момент предоставлены процессу. Для большей эффективности новая контрольная точка должна записываться не поверх старой, а в новый файл, так что во время выполнения процесса образуется целая последовательность контрольных точек.

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 копии, теряется (например, выходные данные, напечатанные позднее контрольной копии, отбрасываются и позже печатаются заново). В результате процесс вновь запускается с более раннего момента, когда он не занимал тот ресурс, который теперь предоставляется одному из процессов, попавших в тупик. Если возобновленный процесс снова пытается получить данный ресурс, ему придется ждать того момента, когда ресурс опять станет доступен.

3. Восстановление путем *уничтожения процессов*.

Грубейший, но одновременно и простейший способ выхода из ситуации взаимоблокировки заключается в уничтожении одного или нескольких процессов. Можно уничтожить процесс, находящийся в цикле взаимоблокировки. При небольшом везении другие процессы смогут продолжить работу. Если первое удаление не помогает, процедуру можно повторять до тех пор, пока цикл наконец не будет разорван.

Можно, наоборот, в качестве жертвы выбрать процесс, не находящийся в цикле, чтобы он освободил свои ресурсы. При этом подходе уничтожаемый процесс выбирается с особой тщательностью, потому что он должен занимать ресурсы, которые нужны некоторым процессам в цикле. Например, один процесс может использовать принтер и требовать плоттер, другой, наоборот, получил плоттер и запрашивает принтер. Оба попали в тупик. Третий процесс может удерживать другие принтер и плоттер и успешно работать Уничтожение третьего процесса приведет к освобождению этих ресурсов и разрушит взаимоблокировку первых двух процессов.

Там, где это возможно, лучше всего уничтожать те процессы, которые можно запустить с самого начала без всяких болезненных эффектов. Например, процедуру компиляции всегда можно повторить заново, поскольку она всего лишь читает исходный файл и создает объектный файл. Если процедуру компиляции уничтожить в процессе работы, первый ее запуск не повлияет на второй.

С другой стороны, процесс, который обновляет базу данных, не всегда можно успешно выполнить во второй раз. Если процесс прибавляет 1 к какой-нибудь записи в базе данных, то его запуск, потом уничтожение, затем повторный запуск приведут к прибавлению к записи 2, что неверно.

Перейдем теперь к алгоритму **избегания взаимоблокировок**. Рассматривая обнаружение взаимоблокировок, мы неявно предполагали, что когда процесс запрашивает ресурсы, он требует их все сразу. Однако в большинстве систем ресурсы запрашиваются поочередно, по одному. Система должна уметь решать, является ли предоставление ресурса безопасным или нет, и предоставлять его процессу только в первом случае. Таким образом, возникает новый вопрос: существует ли алгоритм, который всегда может избежать ситуации взаимоблокировки, все время делая правильный выбор? Ответом является условное "да" – мы можем избежать тупиков, но только если заранее будет доступна определенная информация.

Основные алгоритмы, позволяющие предотвращать взаимоблокировки, базируются на концепции безопасных состояний. В любой момент времени существует текущее состояние, составленное из величин E, A, C и R. Говорят, что **состояние безопасно**, если оно не находится в тупике и существует некоторый порядок планирования, при котором каждый процесс может работать до завершения, даже если все процессы вдруг захотят немедленно получить свое

максимальное количество ресурсов. Проще всего проиллюстрировать эту идею на примере с одним ресурсом. На рисунке 4.5, *а* у нас есть состояние, в котором процесс *A* занимает 3 экземпляра ресурса, но ему в итоге могут потребоваться 9 экземпляров. Процесс *B* в настоящий момент занял 2 экземпляра, но позже ему могут понадобиться всего 4. Процесс *C* владеет двумя, но может потребовать еще 5 штук. В системе есть всего 10 экземпляров данного ресурса, 7 из них уже распределены, три пока свободны.

	Имеет	Max
A	3	9
B	2	4
C	2	7

Свободно: 3

а

	Имеет	Max
A	3	9
B	4	4
C	2	7

Свободно: 1

б

	Имеет	Max
A	3	9
B	0	–
C	2	7

Свободно: 5

в

	Имеет	Max
A	3	9
B	0	–
C	7	7

Свободно: 0

г

	Имеет	Max
A	3	9
B	0	–
C	0	–

Свободно: 7

д

Рисунок 4.5 – Безопасное состояние

Состояние на рисунке 4.5, *а* безопасно, потому что существует такая последовательность предоставления ресурсов, которая позволяет завершиться всем процессам. Планировщик может просто запустить в работу только процесс *B* на то время, пока он запросит и получит два дополнительных экземпляра ресурса, что приведет к состоянию, изображенному на рисунке 4.5, *б*. Когда процесс *B* закончится, мы получим состояние рисунке 4.5, *в*. Затем планировщик может запустить процесс *C*, что со временем приведет нас к ситуации рисунке 4.5, *г*. По завершении процесса *C* мы получим рисунке 4.5, *д*. Теперь процесс *A,* наконец, может занять необходимые ему шесть экземпляров ресурса и также успешно завершиться. Таким образом, состояние на рисунке 4.5, *а* является безопасным, потому что система может избежать тупика с помощью аккуратного планирования процессов.

Теперь предположим, что исходное состояние системы продемонстрировано на рисунке 4.6, *а*, но в данный момент процесс *A* запрашивает и получает еще один ресурс, приводя к рисунку 4.6, *б*. Сможем ли мы найти последовательность, которая гарантирует работу системы? Давайте попытаемся. Планировщик может дать проработать процессу *B* до того момента, пока он не запросит все свои ресурсы, как показано на рисунке 4.6, *в*.

В итоге процесс *B* успешно завершается, и мы получаем ситуацию рисунка 4.6, *г*. В этом месте мы застряли: в системе осталось только четыре свободных экземпляра ресурса, а каждому из активных процессов необходимо пять. И не

существует последовательности действий, гарантирующей успешное завершение всех процессов. Следовательно, решение о предоставлении ресурса, которое передвинуло систему из положения рисунка 4.6, *а* к рисунку 4.6, *б*, привело ее из безопасного в небезопасное состояние. Если из ситуации рисунка 4.6, *б* запустить процесс *A* или *C*, мы не выйдем из тупика. Теперь, оглядываясь назад, можно уверенно сказать, что нельзя было выполнять запрос процесса *A*.

	Имеет	Max
A	3	9
B	2	4
C	2	7

Свободно: 3

а

	Имеет	Max
A	4	9
B	2	4
C	2	7

Свободно: 2

б

	Имеет	Max
A	4	9
B	4	4
C	2	7

Свободно: 0

в

	Имеет	Max
A	4	9
B	0	–
C	2	7

Свободно: 4

г

Рисунок 4.6 – Небезопасное состояние

Следует отметить, что небезопасное состояние само по себе не является тупиком. Начав с рисунка 4.6, *б* система может проработать некоторое время. Фактически даже может успешно завершиться один процесс. Кроме того, возможна ситуация, что процесс *A* сможет освободить один ресурс до следующего своего запроса, позволяя успешно завершиться процессу *C*, а системе избежать взаимной блокировки. Таким образом, разница между безопасным и небезопасным состоянием заключается в следующем: в безопасном состоянии система может гарантировать, что все процессы закончат свою работу, а в небезопасном состоянии такой гарантии дать нельзя.

4.4. Предотвращение взаимоблокировок

Уклонение от взаимоблокировок, в сущности, невозможно, потому что оно требует наличия никому не известной информации о будущих процессах. Тогда возникает справедливый вопрос: как же реальные системы избегают попадания в тупики? Для того чтобы ответить на этот вопрос, вернемся назад к четырем условиям возникновения взаимоблокировок, сформулированным ранее в п.4.3, и посмотрим, смогут ли они дать нам ключ к разрешению проблемы. Если мы сможем гарантировать, что хотя бы одно из этих условий никогда не будет выполнено, тогда взаимоблокировки станут конструктивно невозможными.

Сначала попробуем разобраться с условием взаимного исключения. Если в системе нет ресурсов, отданных в единоличное пользование одному процессу, мы

никогда не попадем в тупик. Но в равной степени понятно, что если позволить двум процессам одновременно печатать данные на принтере, воцарится хаос. Используя подкачку выходных данных для печати, несколько процессов могут одновременно генерировать свои выходные данные. В такой модели только один процесс, который фактически запрашивает физический принтер, является демоном принтера. Так как демон не запрашивает никакие другие ресурсы, для принтера мы можем исключить тупики.

К сожалению, не все устройства поддерживают подкачку данных (таблицу процессов невозможно подкачивать с диска). Кроме того, конкуренция за дисковое пространство для подкачки сама по себе может привести к тупику. Что получится, если два процесса заполнили своими выходными данными каждый по половине дискового пространства, отведенного под подкачку данных, и ни один из них не закончил вычисления? Демон может быть запрограммирован так, что начнет печать, не дожидаясь подкачки всех выходных данных, и принтер тогда простоит впустую в том случае, если вычисляющий процесс решил подождать несколько часов после первого пакета выходных данных. По этой причине обычно демоны программируют так, что они начинают печать только после того, как файл выходных данных целиком станет доступен. В этом случае мы получаем два процесса, каждый из которых обработал часть выходных данных, но не все и не может продолжать вычисления дальше. Ни один из двух процессов никогда не завершится, так что произошла взаимоблокировка на диске.

Тем не менее, в этом проглядывает росток часто применяющегося решения. Избегайте выделения ресурса, когда это не является абсолютно необходимым, и пытайтесь обеспечить ситуацию, в которой фактически претендовать на ресурс может минимальное количество процессов.

Второе из условий, кажется, все же подает надежду. Если мы сможем уберечь процессы, занимающие некоторые ресурсы, от ожидания остальных ресурсов, мы устраним ситуацию взаимоблокировки. Один из способов достижения этой цели состоит в требовании, следуя которому любой процесс должен запрашивать все необходимые ресурсы до начала работы. Если все ресурсы доступны, процесс получит все, что ему нужно, и сможет работать до успешного завершения. Если один или несколько ресурсов заняты, процессу ничего не предоставляется, и он непременно попадает в состояние ожидания.

Первая проблема при этом подходе заключается в том, что многие процессы не знают, сколько ресурсов им понадобится, до тех пор, пока не начнут работу. Другая проблема состоит в том, что при этом методе ресурсы не будут использоваться оптимально. Возьмем, например, процесс, который читает данные с входной ленты, анализирует их в течение часа и затем пишет выходную ленту, а заодно и чертит результаты на плоттере. Если все ресурсы нужно запрашивать заранее, то процесс в течение часа не позволит работать накопителю на магнитной ленте и принтеру.

И все-таки некоторые пакетные системы на мэйнфреймах требуют, чтобы пользователи объявляли список всех ресурсов в первой строке каждого задания. Затем система немедленно запрашивает все ресурсы и сохраняет их до окончания задачи. Этот способ накладывает ограничения на деятельность программиста и занимается расточительством ресурсов, зато предотвращает безвыходные тупиковые ситуации.

Немного отличный метод, позволяющий нарушить условие удержания и ожидания, заключается в наложении следующего требования на процесс, запрашивающий ресурс: процесс сначала должен временно освободить все используемые им в данный момент ресурсы. Затем этот процесс пытается сразу получить все необходимое.

Попытка исключить третье условие (нет принудительной выгрузки ресурса) подает еще меньше надежд, чем устранение второго условия. Если процесс получил принтер и в данный момент печатает выходные данные, насильственное изъятие принтера по причине недоступности требуемого плоттера в лучшем случае сложно, в худшем – невозможно.

Остается только одно условие. Циклическое ожидание можно устранить несколькими способами. Один из них: просто следовать правилу, гласящему, что процессу дано право только на один ресурс в конкретный момент времени. Если нужен второй ресурс, процесс обязан освободить первый. Но подобное ограничение неприемлемо для процесса, копирующего огромный файл с магнитной ленты на принтер.

Другой способ уклонения от циклического ожидания заключается в поддержке обшей нумерации всех ресурсов. Тогда действует следующее правило: процессы могут запрашивать ресурс, когда хотят этого, но все запросы должны быть

сделаны в соответствии с нумерацией ресурсов. Процесс может запросить сначала устройство с меньшим номером, затем – с большим, но не наоборот.

Вариантом этого алгоритма является схема, в которой отбрасывается требование приобретения ресурсов в строго возрастающем порядке, но сохраняется условие, что процесс не может запросить ресурсы с меньшим номером, чем уже у него имеющиеся. Если процесс на начальной стадии запрашивает ресурсы 9 и 10, затем освобождает их, то это равнозначно тому, как если бы он начал работу заново, поэтому нет причины теперь запрещать ему запрос ресурса 1.

Несмотря на то, что систематизация ресурсов с помощью их нумерации устраняет проблему взаимоблокировки, бывают ситуации, когда невозможно найти порядок, удовлетворяющий всех. Когда ресурсы включают в себя области таблицы процессов, дисковое пространство для подкачки данных, закрытые записи базы данных и другие абстрактные ресурсы, число потенциальных ресурсов и вариантов их применений может быть настолько огромным, что никакая систематизация не сможет работать.

Таким образом, все методы предотвращения взаимоблокировок можно свести в таблицу 4.1.

Таблица 4.1. Методы предотвращения взаимоблокировок

Условие	Метод
Взаимное исключение	Организовывать подкачку данных
Удержание и ожидание	Запрос всех ресурсов на начальной стадии
Нет принудительной выгрузки ресурса	Отобрать ресурсы
Циклическое ожидание	Пронумеровать ресурсы и упорядочить

5. УПРАВЛЕНИЕ ПАМЯТЬЮ

5.1. Модели организации памяти

Одной из задач ОС является координация использования всех составляющих памяти. Часть ОС, отвечающая за управление памятью, называется **модулем управления памятью** или **менеджером памяти**.

Системы управления памятью можно разделить на 2 класса: перемещающие процессы между оперативной памятью и диском во время их выполнения (то есть осуществляющие подкачку процессов целиком (*swapping*) или использующие страничную подкачку (*paging*)) и те, которые этого не делают. Второй вариант проще, поэтому начнем с него.

Самая простая из возможных схем управления памятью заключается в том, что в каждый конкретный момент времени работает только одна программа, при этом память разделяется между программами и ОС (рисунок 5.1).

Рисунок 5.1 – Три простейшие модели организации памяти

Когда система организована таким образом, в каждый конкретный момент времени может работать только один процесс. Как только пользователь набирает команду, ОС копирует запрашиваемую программу с диска в память и выполняет ее, а после окончания процесса ждет новой команды. Получив команду, она загружает новую программу в память, записывая ее поверх предыдущей.

Большинство современных систем позволяет одновременный запуск нескольких процессов. Самый легкий способ достижения многозадачности представляет собой простое разделение памяти на *n* разделов. Такое разбиение можно выполнить, например, вручную при запуске системы.

Когда задание поступает в память, его можно расположить во входной очереди к наименьшему разделу, достаточно большому, чтобы вместить это задание. Так как размер разделов неизменен, все пространство, неиспользуемое работающим процессом, пропадает (рисунок 5.2, *а*).

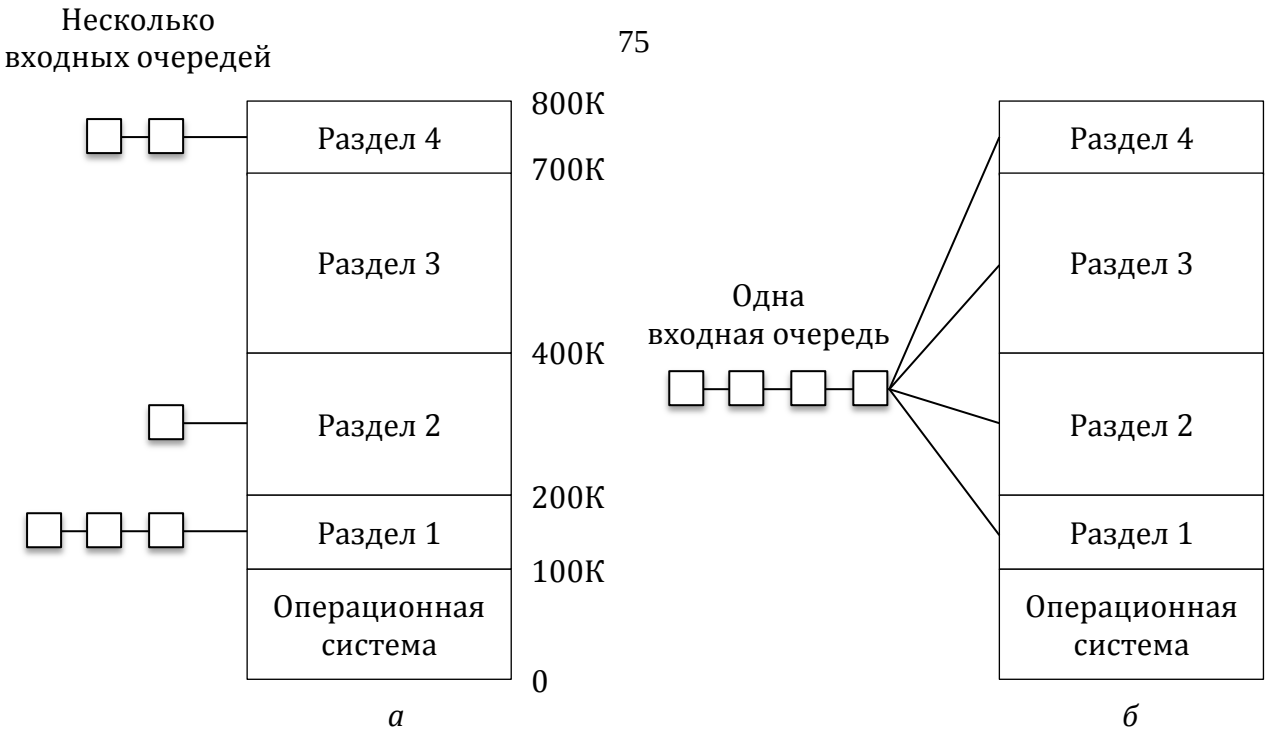


Рисунок 5.2 – Фиксированные разделы памяти с отдельными входными очередями для каждого раздела (*а*); с одной очередью на вход (*б*)

Недостаток сортировки входящих работ по отдельным очередям становится очевидным, когда к большому разделу нет очереди, в то время, как к маленькому выстроилось довольно много задач. Альтернативная схема заключается в организации общей очереди для всех разделов (рисунок 5.2, *б*). Как только раздел освобождается, задачу, находящуюся ближе всего к началу очереди и подходящую для выполнения в этом разделе, можно загрузить в него и начать обработку. Поскольку нежелательно тратить большие разделы на маленькие задачи, существует другая стратегия – каждый раз после освобождения раздела происходит поиск в очереди наибольшего из помещающихся в него заданий. Однако это приводит к тому, что мелкие задачи простаивают. Чтобы этого избежать, можно создать хотя бы один маленький раздел памяти, который позволит выполнять мелкие задания без долгого ожидания.

При другом подходе задачу, имеющую право быть выбранной для обработки, можно пропустить не более определенного числа раз.

5.2. Подкачка

Организация памяти в виде фиксированных разделов проста и эффективна для работы с пакетными системами. Каждое задание после того, как доходит до начала очереди, загружается в раздел памяти и остается там до своего завершения. До тех пор, пока в памяти может храниться достаточное количество задач для

обеспечения постоянной занятости центрального процессора, нет причин что-либо усложнять.

В случае с системами разделения времени или ПК, ориентированными на работу с графикой, оперативной памяти оказывается недостаточно, чтобы вместить все текущие активные процессы. Тогда избыток процессов приходится хранить на диске, а для обработки динамически переносить их в память.

Существует два основных подхода к управлению памятью в таких системах. Самый простой вариант – это **свопинг** (swapping) или **обычная подкачка**, когда каждый процесс полностью переносится в память, работает некоторое время и затем целиком возвращается на диск. Второй вариант – **виртуальная память**, позволяющая программам работать даже тогда, когда они только частично находятся в оперативной памяти.

Работа системы свопинга представлена на рисунке 5.3. На начальной стадии в памяти находится только процесс *A*. Затем создаются или загружаются с диска процессы *B* и *C*. На рисунке 5.3, *г* процесс *A* выгружается на диск. Затем появляется процесс *D*, а процесс *B* завершается. Наконец, процесс *A* снова возвращается в память. Так как теперь процесс *A* имеет другое размещение в памяти, его адреса должны быть перенастроены или программно во время загрузки в память, или (более заманчивый вариант) аппаратно во время выполнения программы.

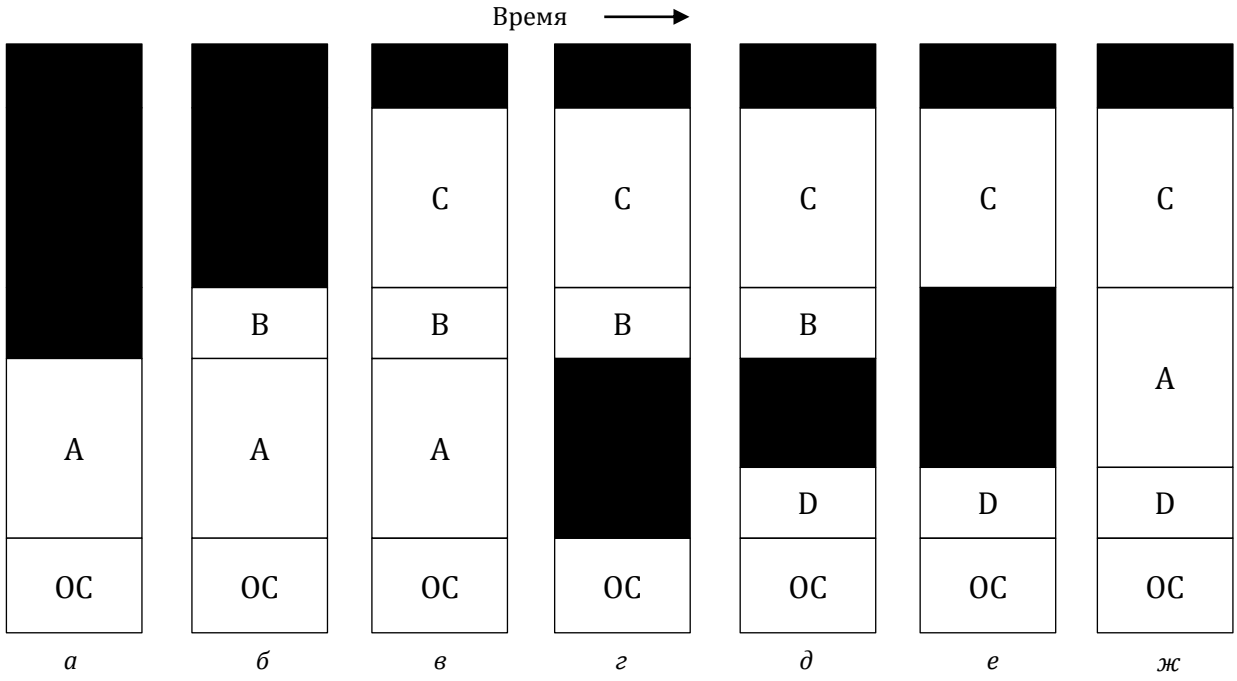


Рисунок 5.3 – Изменение распределения памяти по мере поступления процессов

Основная разница между фиксированными разделами на рисунке 5.2 и непостоянными разделами на рисунке 5.3 заключается в том, что во втором случае количество, размещение и размер разделов изменяются динамически по мере поступления и завершения процессов, тогда как в первом варианте они фиксированы. Гибкость схемы, в которой нет ограничений, связанных с определенным количеством разделов, и каждый раздел может быть очень большим или совсем маленьким, улучшает использование памяти, однако усложняет операции размещения процессов и освобождения памяти, а также отслеживание происходящих изменений.

Когда в результате подкачки процессов с диска в памяти появляется множество неиспользованных фрагментов, их можно объединить в один большой участок, передвинуть все процессы в сторону младших адресов настолько, насколько это возможно. Такая операция называется **уплотнением** или **сжатием памяти**. Обычно ее не выполняют, потому что на нее уходит много времени работы процессора.

Еще один момент, на который стоит обратить внимание: сколько памяти должно быть предоставлено процессу, когда он создается или скачивается с диска? Если процесс имеет фиксированный, никогда не изменяющийся размер, размещение происходит просто: операционная система предоставляет точно необходимое количество памяти, ни больше, ни меньше, чем нужно.

Однако если область данных процесса может расти, например, в результате динамического распределения памяти, как происходит во многих языках программирования, проблема предоставления памяти возникает каждый раз, когда процесс пытается увеличиться. Когда участок неиспользованной памяти расположен рядом с процессом, его можно отдать в пользу процесса, таким образом, позволив процессу вырасти на размер этого участка. Если же процесс соседствует с другим процессом, для его увеличения нужно или переместить достаточно большой свободный участок памяти, или перекачать на диск один или больше процессов, чтобы создать незанятый фрагмент достаточного размера. Если процесс не может расти в памяти, а область на диске, предоставленная для подкачки, переполнена, процесс будет вынужден ждать освобождения памяти или же будет уничтожен.

Если предположить, что большинство процессов будут увеличиваться во время работы, вероятно, сразу стоит предоставлять им немного больше памяти, чем требуется, а всякий раз, когда процесс скачивается на диск или перемещается в

памяти, обрабатывать служебные данные, связанные с перемещением или подкачкой процессов, больше не умещающихся в предоставленной им памяти. Но когда процесс выгружается на диск, должна скачиваться только действительно используемая часть памяти, так как очень расточительно также перемещать и дополнительную память. На рисунке 5.4, *а* можно увидеть конфигурацию памяти с предоставлением пространства для роста двух процессов.

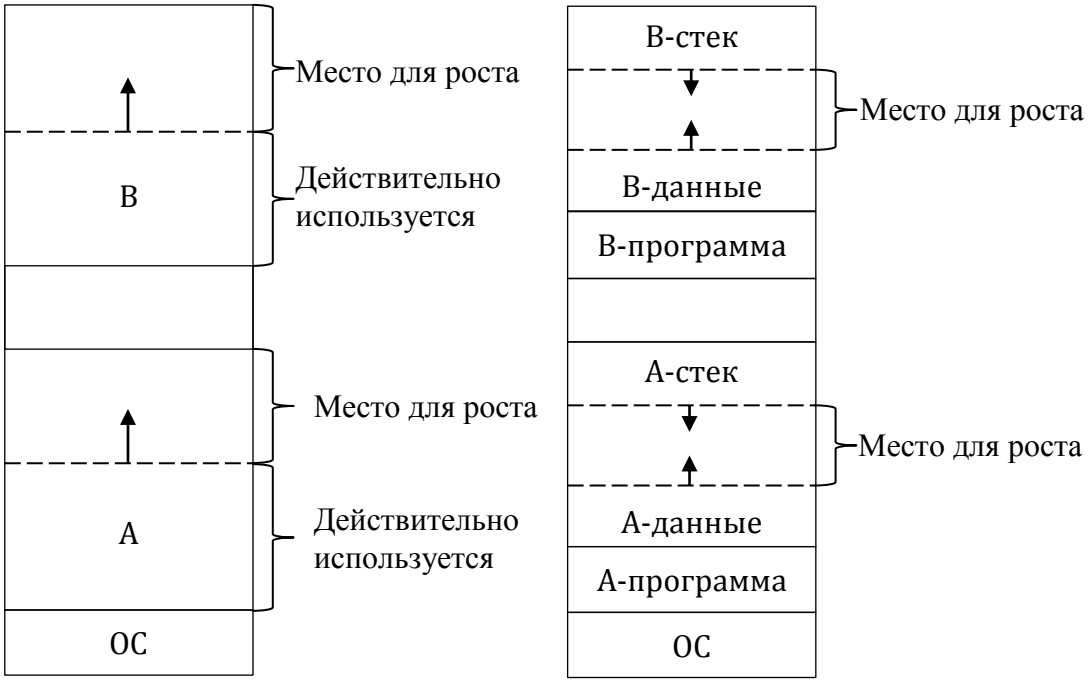


Рисунок 5.4 – Предоставление пространства для роста области данных (*а*); предоставление для роста стека и области данных (*б*)

Если процесс может иметь два увеличивающихся сегмента, например сегмент данных, используемый для динамически назначаемых и освобождаемых переменных, и сегмент стека для обычных локальных переменных и возвращаемых адресов, предлагается альтернативная схема распределения памяти (рисунок 5.4, *б*). У каждого процесса вверху предоставленной ему области памяти находится стек, который расширяется вниз, и сегмент данных, расположенный отдельно от текста программы, который увеличивается вверх. Область памяти между ними разрешено использовать для любого сегмента. Если ее становится недостаточно, то процесс нужно или перенести на другое, большее свободное место, или выгрузить на диск до появления свободного пространства необходимого размера, или уничтожить.

5.3. Виртуальная память

Рассмотрим второй подход к управлению памятью – **виртуальную память**. Ее основная идея заключается в том, что объединенный размер программы, данных и стека может превысить количество доступной физической памяти. Операционная система хранит части программы, использующиеся в настоящий момент, в оперативной памяти, остальные – на диске. Например, программа размером 16 Мбайт сможет работать на машине с 4 Мбайт памяти, если тщательно продумать, какие 4 Мбайт должны храниться в памяти в каждый момент времени. При этом части программы, находящиеся на диске и в памяти, будут меняться местами по мере необходимости.

Виртуальная память может также работать в многозадачной системе при одновременно находящихся в памяти частях многих программ. Когда программа ждет перемещения в память очередной ее части, она находится в состоянии ожидания ввода-вывода и не может работать, поэтому центральный процессор может быть отдан другому процессу тем же самым способом, как в любой другой многозадачной системе.

Большинство систем виртуальной памяти используют технику, называемую **страничной организацией памяти** (paging). Эта техника заключается в следующем. ОС программно формирует **виртуальные адреса**, составляющие **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 осуществляет отображение виртуальных адресов на физические адреса памяти

На рисунке 5.5 приведен пример такого отображения. Пусть у нас есть компьютер, который может формировать 16-разрядные адреса, от 0 до 64 К. Это виртуальные адреса. Однако у этого компьютера только 32 Кбайт физической памяти, поэтому, хотя программы размером 64 Кбайт могут быть написаны, они не могут целиком быть загружены в память и запущены. Полная копия образа памяти программы размером до 64 Кбайт должна присутствовать на диске, но в таком виде, чтобы ее можно было по мере надобности перекачать в память по частям.

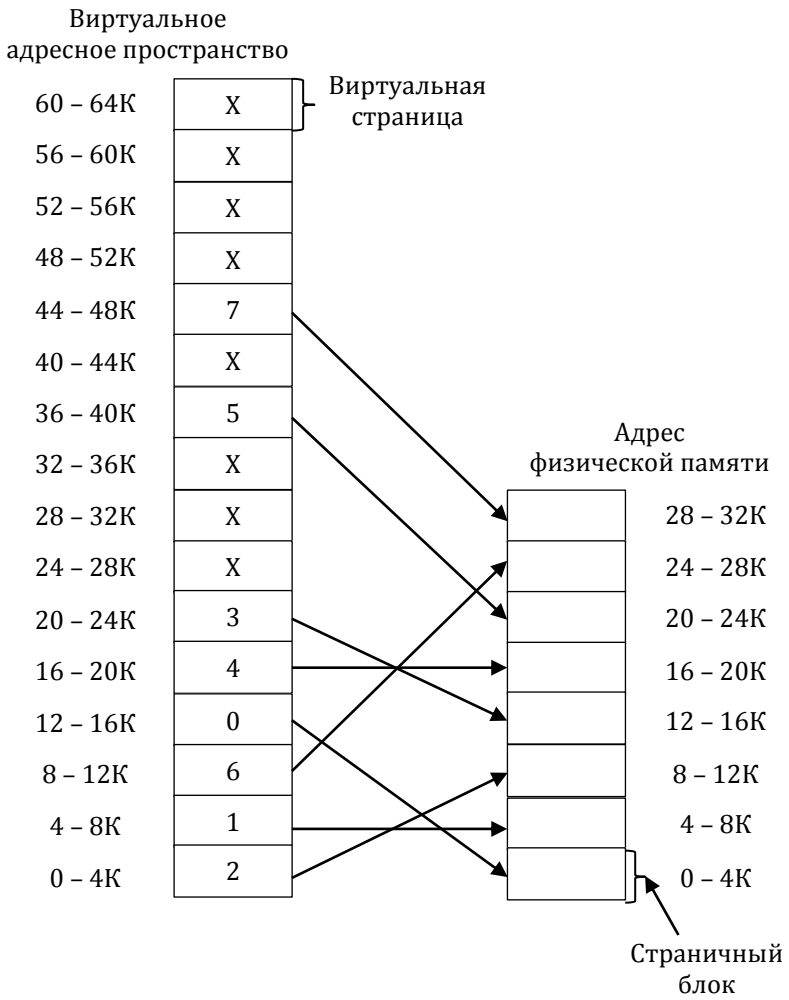


Рисунок 5.5 – Связь между виртуальными и физическими адресами

Пространство виртуальных адресов разделено на единицы, называемые **страницами**. Соответствующие единицы в физической памяти называются **страничными блоками** (page frame). Страницы и их блоки всегда имеют одинаковый размер. В этом примере они равны 4 Кбайт, но в реальных системах используются размеры страниц от 512 байт до 64 Кбайт. Имея 64 Кбайт виртуального адресного пространства и 32 Кбайт физической памяти, мы получаем 16 виртуальных страниц и 8 страничных блоков. Передача данных между ОЗУ и диском всегда происходит в страницах.

Когда программа пытается получить доступ к адресу 0, например, используя команду

MOV REG.0

виртуальный адрес 0 передается диспетчеру памяти (MMU). Диспетчер памяти видит, что этот виртуальный адрес попадает на страницу 0 (от 0 до 4095), которая отображается страничным блоком 2 (от 8192 до 12287). Диспетчер переводит виртуальный адрес 0 в физический адрес 8192 и выставляет последний

на шину. Память ничего не знает о диспетчере памяти и видит просто запрос на чтение или запись слова по адресу 8192, который и выполняет. Таким образом, диспетчер памяти эффективно отображает все виртуальные адреса между 0 и 4095 на физические адреса от 8192 до 12287.

Сама по себе возможность отображения 16 виртуальных страниц на любой из восьми страничных блоков с помощью установки соответствующей карты в диспетчере памяти не решает проблемы, заключающейся в том, что размер виртуального адресного пространства больше физической памяти. Так как у нас есть только восемь физических страничных блоков, только восемь виртуальных страниц на рисунке 5.5 воспроизводятся в физической памяти. Другие страницы, обозначенные на рисунке крестиками, не отображаются. В фактическом аппаратном обеспечении страницы, физически присутствующие в памяти, отслеживаются с помощью бита присутствия/отсутствия.

Что происходит, если программа пытается воспользоваться неотображаемой страницей, например, с помощью инструкции

MOV REG.32780

которая обращается к байту 12 на виртуальной странице 8 (начинающейся с адреса 32768)? Диспетчер памяти замечает, что страница не отображается (обозначена крестиком на рисунке), и инициирует прерывание центрального процессора, передающее управление операционной системе. Такое прерывание называется **ошибкой из-за отсутствия страницы** или **страничным прерыванием** (page fault). При этом ОС выбирает малоиспользуемый страничный блок и записывает его содержимое на диск. Затем она считывает с диска страницу, на которую произошла ссылка, в только что освободившийся блок, изменяет карту отображения и запускает заново прерванную команду.

Например, если операционная система решает удалить из оперативной памяти страничный блок 1, она загружает виртуальную страницу 8 по физическому адресу 4 К и производит два изменения в карте диспетчера памяти. Во-первых, отмечается содержимое виртуальной страницы 1 как неотображаемое для того, чтобы перехватывать в будущем любые попытки обращения к виртуальным адресам между 4 К и 8 К. Затем заменяется крест в записи для виртуальной страницы 8 на номер 1, так что когда прерванная команда будет выполняться заново, она отобразит виртуальный адрес 32780 на физический адрес 4108.

5.4. Алгоритмы замещения страниц

Когда происходит страничное прерывание, ОС должна выбрать страницу для удаления из памяти, чтобы освободить место для страницы, которую нужно перенести в память. Если удаляемая страница была изменена за время своего присутствия в памяти, то ее необходимо переписать на диск, чтобы обновить копию, хранящуюся там. Однако если страница не была модифицирована, ее не надо переписывать и загружаемая страница записывается поверх выгружаемой.

Хотя, в принципе, можно осуществлять выбор выгружаемой страницы случайно, однако производительность системы значительно повышается, когда предпочтение отдается редко используемой странице. Существует достаточно много алгоритмов выбора замещаемой страницы. Рассмотрим наиболее важные из них.

1. Оптимальный алгоритм – каждая страница в памяти помечается количеством команд, которые будут выполняться перед первым обращением к этой странице. Для выгрузки выбирается страница с наибольшей меткой. Однако такой механизм невозможно осуществить. Он может использоваться только на основе сведений о предыдущих обращениях к страницам памяти, причем именно этой программы с именно этими входными данными.

2. Алгоритм NRU – не использовавшаяся в последнее время страница (Not Recently Used). Все страницы делятся на 4 класса:

- Класс 0 – не было обращений и изменений;
- Класс 1 – не было обращений, есть изменения;
- Класс 2 – было обращение, нет изменений;
- Класс 3 – были обращение и изменение

В этом алгоритме ОС выбирает для удаления случайную страницу в непустом классе с наименьшим номером.

3. Алгоритм FIFO – первым прибыл – первым обслужен. ОС хранит список всех страниц, находящихся в памяти, в котором первая страница является старейшей, а страница в хвосте списка попала в него совсем недавно.

При возникновении прерывания удаляется страница в голове списка, а новая добавляется в конец. Однако может быть удалена старейшая страница, которая часто используется.

4. Алгоритм "вторая попытка" – аналогичен предыдущему, но у старейшей страницы изучается бит *R*. Если он равен 0, значит страница находится в памяти

давно и при этом не используется. Если он равен 1, то ему присваивается значение 0 и страница ставится в конец списка, а время ее загрузки обновляется. Однако это не очень эффективно, так как страницы постоянно передвигаются по списку.

5. Алгоритм "часы" – страничные блоки хранятся в кольцевом списке в форме часов. Стрелка указывает на старейшую страницу. Когда происходит страничное прерывание, проверяется страница, на которую указывает стрелка. Если ее бит *R* равен 0, страница выгружается, на ее место становится новая страница, а стрелка перемещается к следующей странице. Если бит *R* равен 1, то он сбрасывается, а стрелка перемещается к следующей странице и так пока не найдется страница, у которой бит *R* = 0.

6. Алгоритм LRU – страница, не использовавшаяся дольше всего (Least Recently Used). В этом алгоритме предполагается, что страницы, к которым происходило частое обращение, будут и дальше востребованы, а те, к которым и раньше редко обращались, вряд ли потребуются в ближайшее время.

7. Алгоритм "рабочий набор" – с самого начала после запуска процесса, когда в памяти отсутствуют любые страницы, в нее загружаются те, которые требуются для работы в конкретный момент, а не все подряд. В дальнейшем выбирается страница, наименее часто используемая в рабочем наборе (если время обращения к ней отличается от текущего виртуального времени больше, чем на время *T*). Однако при каждом страничном прерывании приходится проверять весь рабочий набор, что слишком накладно.

8. Алгоритм WSClock (Workset Clock)– это своего рода гибрид алгоритма "рабочий набор" и алгоритма "часы". Также создается рабочий набор страниц, однако обход их осуществляется по принципу алгоритма "часы". Если страница имеет бит *R*, равный 1, то она пропускается с обнулением значения, а если бит *R* равен 0, то определяется возраст страницы и есть ли в ней изменения. Если возраст больше времени *T* и она без изменений, то поверх нее загружается новая страница, а если изменения были, то она ставится в очередь на копирование на диск, а алгоритм продолжает искать страницу, которую можно сразу заместить.

5.5. Вопросы разработки систем со страничной организацией памяти

При разработке систем со страничной организацией памяти необходимо учитывать ряд моментов, чтобы получить хорошую производительность.

Во-первых, это *политика распределения памяти*. Если используется *локальная*, то поиск страниц на замещение производится среди страниц, занятых тем же процессом. Если *глобальная*, то среди всех страниц. Более предпочтительной является глобальная политика распределения памяти ввиду большей эффективности и производительности.

Во-вторых, это *регулирование загрузки*. Если сумма рабочих наборов всех процессов будет превышать размеры памяти, можно ожидать пробуксовки. Уменьшить количество конкурирующих за использование памяти процессов можно, выгрузив некоторые из них на диск и освободив все занимаемые ими страницы.

В-третьих, это *размер страницы*. Оптимального решения не существует, но есть доводы как в пользу маленького размера страниц, так и в пользу большого. Чаще всего данные не заполняют полностью целое количество страниц – всегда последняя страница будет наполовину пустой. Если в памяти n сегментов данных, а размер страницы равен p байтам, то $np/2$ байт будет потрачено впустую. Поэтому чем меньше страницы, тем лучше. Еще один довод в пользу небольших страниц – это то, что если, например, представить себе программу, состоящую из восьми последовательных этапов по 4 Кбайт каждый, то при размере страниц 32 Кбайт будет пропадать впустую 28 Кбайт в каждой странице. С другой стороны, небольшой размер страницы означает, что программам будет нужно много страниц, следовательно, огромная таблица страниц. Поэтому обычно руководствуются формулой, определяющей оптимальный средний размер страниц (5.1):

$$p = \sqrt{2se}, \tag{5.1}$$

где s – средний размер процесса,

e – количество байт, требуемое для записи каждой страницы

Чаще всего используются страницы размером 4 или 8 Кбайт.

Четвертым моментом, который необходимо учитывать при разработке систем со страничной организацией памяти – это использование *единого адресного пространства для команд и данных или отдельных пространств для каждого из них*. Сейчас чаще используется единое пространство, однако это удобно только в случае достаточной его вместимости. В противном случае лучше иметь отдельные пространства.

Пятый момент – это использование *совместного доступа к страницам или их разделение*. В больших многозадачных системах часто случается, что в одно и то же время несколько пользователей работают с одной программой. Поэтому более

эффективно использовать страницы совместно, чтобы избежать одновременного присутствия в памяти двух копий одной и той же страницы. Однако не все страницы могут быть разделяемыми. Так, страницы только для чтения, такие как текст программы, можно использовать совместно, а страницы с данными – нельзя.

Шестой момент – это *политика очистки страниц*. Чтобы обеспечить достаточный запас свободных страничных блоков, во многих системах работает фоновый процесс – **страничный демон**, который большую часть времени спит, но периодически просыпается и проверяет состояние памяти. Если свободно слишком мало блоков, он начинает выбирать страницы для удаления их из памяти, используя определенный алгоритм замещения. Если эти страницы изменялись со времени загрузки, они записываются на диск.

И, наконец, седьмой момент – это *интерфейс виртуальной памяти*. Чаще всего программисты видят только виртуальное адресное пространство с небольшой физической памятью. Однако в последнее время в ОС им предоставляется определенный контроль над картой памяти, что позволяет двум и более процессам совместно использовать одну и ту же память.

5.6. Вопросы реализации

Можно выделить четыре ситуации, в которых ОС приходится выполнять работу, относящуюся к страничной подкачке: создание процесса, выполнение процесса, страничное прерывание и завершение процесса.

При создании нового процесса ОС должна определить, насколько будут велики программа и данные к ней, и создать для них таблицу страниц. Кроме того, в области подкачки на диске должно быть выделено пространство, чтобы в момент выгрузки страницы на диск было бы место, куда ее можно поместить. Наконец, информация о таблице страниц и области подкачки на диске должна быть записана в таблицу процесса.

Когда процесс планируется для исполнения, для нового процесса требуется сброс диспетчера памяти (MMU), а содержимое буфера быстрого преобразования адреса должно быть очищено, чтобы избавиться от следов предыдущего процесса.

Когда происходит страничное прерывание, ОС должна прочитать аппаратные регистры, чтобы определить, какой виртуальный адрес вызвал ошибку. Из полученной информации она должна вычислить, какая требуется страница, и определить ее местоположение на диске. После этого ОС должна считать требуемую

страницу в страничный блок. И, наконец, она должна вернуть в предыдущее состояние счетчик команд, чтобы тот указывал на вызвавшую прерывание инструкцию, и запустить эту команду заново.

Когда процесс завершается, ОС должна освободить его таблицу страниц, его страницы и дисковое пространство, которое занимают страницы, когда находятся на диске.

Наконец мы подошли к моменту, когда можно более детально описать, что же происходит при страничном прерывании. Последовательность действий следующая:

1. Аппаратное обеспечение переключает систему в режим ядра, сохраняя счетчик команд в стеке. На большинстве машин в специальных регистрах процессора сохраняется некоторая информация о состоянии текущей инструкции.
2. Запускается написанная на ассемблере программа, сохраняющая основные регистры и другую изменяющуюся информацию, защищая ее от разрушения операционной системой. Эта программа вызывает операционную систему как процедуру.
3. Операционная система обнаруживает, что произошло страничное прерывание, и пытается найти необходимую виртуальную страницу. Часто требуемую информацию содержит один из аппаратных регистров. Если нет, операционная система должна достать из стека счетчик команд, выбрать инструкцию и программно проанализировать ее, чтобы определить, что она делала в тот момент, когда случилась ошибка.
4. Как только становится известен виртуальный адрес, вызвавший прерывание, система проверяет, имеет ли силу этот адрес, и согласуется ли зашита с доступом. Если нет, то процессу посылается сигнал или процесс уничтожается. Если адрес действителен и не произошло ошибки защиты, система проверяет наличие свободных страничных блоков. Если свободных блоков нет, запускается алгоритм замещения страниц, выбирающий жертву.
5. Если выбранный страничный блок "грязный", страница заносится в график записи на диск и происходит переключение контекста, приостанавливающее вызвавший прерывание процесс и позволяющее работать другому процессу до тех пор, пока не будет выполнен перенос страницы на диск. В любом случае блок отмечается как занятый, чтобы предотвратить его использование в других целях.

6. Как только страничный блок очищается (или немедленно, или после записи на диск), операционная система ищет адрес на диске, где находится требуемая страница, и планирует дисковую операцию для ее переноса в память. Во время загрузки страницы процесс, вызвавший прерывание, все еще приостановлен и выполняется другой пользовательский процесс, если такой доступен.

7. Когда дисковое прерывание отмечает, что страница поступила в память, обновляется таблица страниц, отражая ее позицию, а блок помечается, как находящийся в нормальном состоянии.

8. Прерванная команда возвращается к тому состоянию, с которого она начиналась, и значение счетчика команд приостановленного процесса (в стеке или в системной ячейке памяти) корректируется так, чтобы указывать на эту команду.

9. Прерванный процесс вносится в график, и операционная система возвращает управление ассемблерной процедуре, вызывавшей ее.

10. Эта процедура перезагружает регистры и другую информацию о состоянии и возвращает управление в пользовательское пространство для продолжения выполнения пользовательской программы, как если бы никакого прерывания не происходило.

5.7. Сегментация

Обсуждавшаяся до сих пор виртуальная память представляет собой одномерное пространство, потому что виртуальные адреса идут один за другим от 0 до некоторого максимума. Для многих задач наличие двух и более отдельных виртуальных адресных пространств может оказаться намного лучше, чем всего одно. Так, очень часто возникают ситуации, когда одновременно имеются таблицы с излишеством ячеек и таблицы с их недостатком. В связи с этим требуется забирать излишки у одних и передавать другим. Поэтому необходим метод, освобождающий программиста от управления расширяющимися и сокращающимися таблицами рутинным способом.

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. Длина каждого сегмента может быть любой от нуля до разрешенного максимума. Различные сегменты могут быть различной длины. Более того, длины сегментов могут изменяться во время выполнения. Длина сегмента

стека может увеличиваться всякий раз, когда что-либо помещается в стек и уменьшаться при выборке данных из стека. Поскольку каждый сегмент составляет отдельное адресное пространство, разные сегменты могут расти или сокращаться независимо друг от друга.

Реализация сегментации существенно отличается от страничной организации памяти: страницы имеют фиксированный размер, а сегменты – нет. На рисунке 5.6, *а* показан пример физической памяти, изначально содержащей пять сегментов. При замене сегментов 1, 4 и 3 сегментами меньшего размера 7, 5 и 6 соответственно возникают свободные участки (рисунок 5.6, *б-г*). Это называется **поклеточной разбивкой** или **внешней фрагментацией**. С ней борются с помощью уплотнения путем перемещения свободных участков в одну область памяти (рисунок 5.6, *д*).

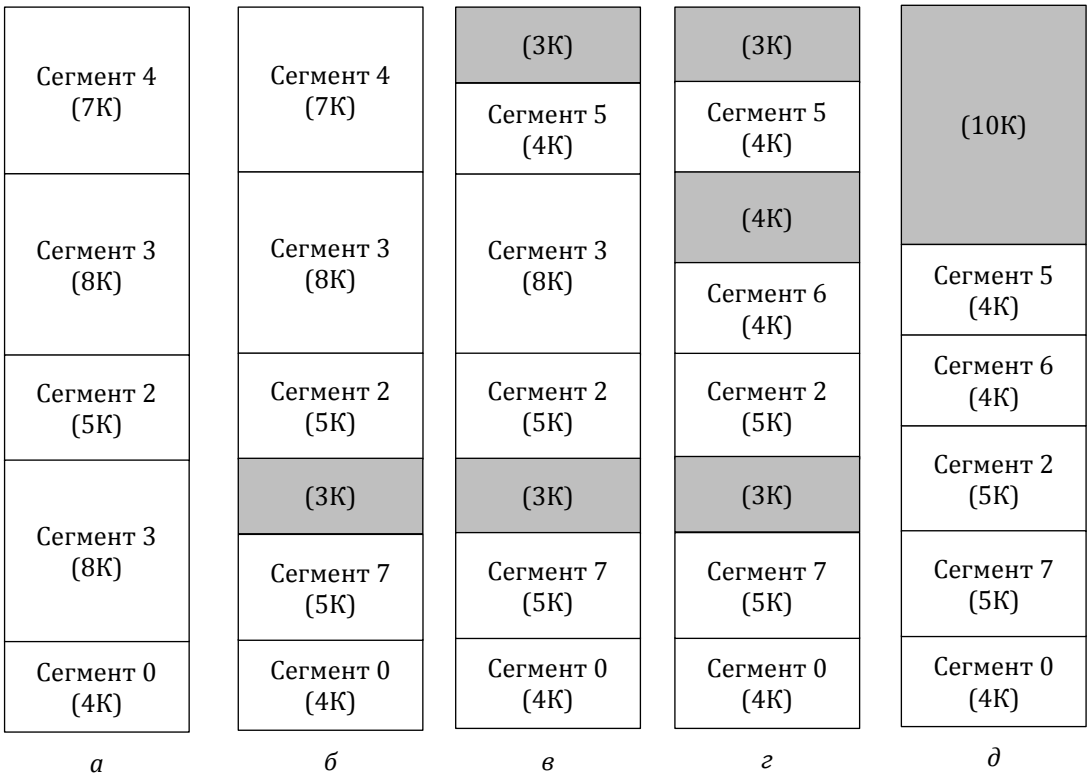


Рисунок 5.6 – Развитие внешней фрагментации (*а – г*); устранение фрагментации с помощью уплотнения (*д*)

6. ВВОД И ВЫВОД В ОПЕРАЦИОННЫХ СИСТЕМАХ

6.1. Принципы аппаратуры ввода-вывода

Одна из важнейших функций ОС состоит в управлении всеми устройствами ввода-вывода компьютера. ОС должна давать этим устройствам команды, перехватывать прерывания и обрабатывать ошибки. Она также должна обеспечить простой и удобный интерфейс между устройствами и остальной частью системы. При этом интерфейс должен быть максимально одинаковым для всех устройств.

Устройства ввода-вывода можно разделить на две категории: **блочные устройства** и **символьные устройства**. Блочные хранят информацию в виде блоков фиксированного размера, причем у каждого блока имеется свой адрес. Важное свойство блочного устройства состоит в том, что каждый его блок может быть прочитан независимо от остальных блоков. Наиболее распространенными блочными устройствами являются диски.

Символьные устройства принимают или предоставляют поток символов без какой-либо блочной структуры. Они не являются адресуемыми и не выполняют операцию поиска. К ним относятся принтеры, сетевые карты, "мыши" и т.д.

При этом устройства ввода-вывода покрывают огромный диапазон скоростей, что создает определенные трудности для программного обеспечения, которому приходится обеспечивать хорошую производительность на скоростях передачи данных, различающихся несколькими порядками.

Устройства ввода-вывода обычно состоят из механической и электронной частей. Электронный компонент устройства называется **контроллером устройства** или **адаптером**.

У каждого контролера есть несколько регистров, с помощью которых с ним может общаться центральный процессор. При помощи записи в эти регистры ОС велит устройству предоставить данные, принять данные, включиться или выключиться и т.д.

Помимо управляющих регистров, у многих устройств есть буфер данных, из которого ОС может читать данные, а также писать данные в него.

Существует два альтернативных способа реализации доступа к управляющим регистрам и буферам данных устройств ввода-вывода. Первый вариант заключается в том, что каждому управляющему регистру назначается номер **порта ввода-вывода**, 8- или 16-разрядное целое число. Второй подход состоит в отображении

всех управляющих регистров периферийных устройств на адресное пространство памяти. Такая система называется **отображаемым на адресное пространство памяти вводом-выводом**.

Независимо от того, отображаются ли регистры или буферы ввода-вывода на память или нет, центральному процессору необходимо как-то адресоваться к контроллерам устройств для обмена данными с ними. ЦП может запрашивать данные от контроллера ввода-вывода по одному байту, но подобная организация обмена данными крайне неэффективна, так как расходует огромное количество процессорного времени. Поэтому на практике часто применяется другая схема, называемая **прямым доступом к памяти** (*DMA, direct memory access*).

Чтобы понять, как работает DMA, познакомимся сначала с тем, как происходит чтение с диска при отсутствии DMA (рисунок 6.1). Сначала контроллер считывает с диска блок (один или несколько секторов) последовательно, бит за битом, пока весь блок не окажется во внутреннем буфере контроллера (*1*). Затем контроллер проверяет контрольную сумму, чтобы убедиться, что при чтении не произошло ошибки (*2*). После этого контроллер инициирует прерывание (*3*). Когда ОС начинает работу, она может прочитать блок диска побайтно или пословно, в цикле сохраняя считанное слово или байт в оперативной памяти.

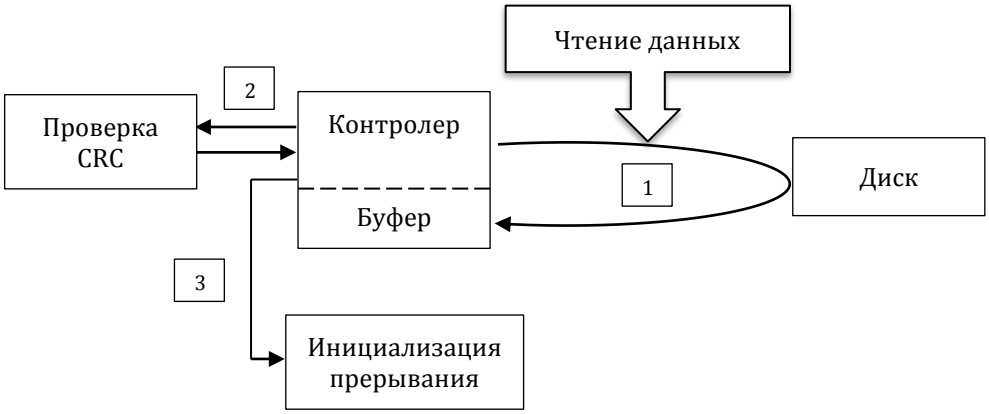


Рисунок 6.1 – Чтение данных с диска при отсутствии DMA
(объяснение в тексте)

При использовании DMA процедура совершенно другая (рисунок 6.2). Сначала ЦП программирует DMA-контроллер, устанавливая его регистры и указывая, какие данные и куда следует переместить (*1*). Затем процессор дает команду дисковому контроллеру прочитать данные во внутренний буфер и проверить контрольную сумму (*2*). Когда данные получены (3) и проверены контроллером диска (*4*), DMA может начинать свою работу.

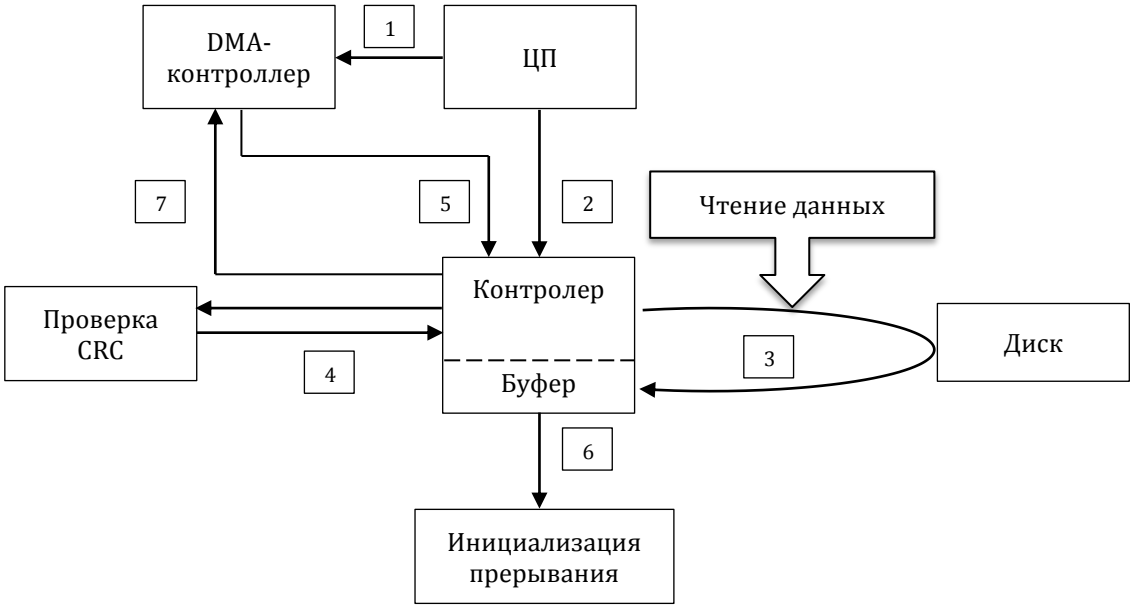


Рисунок 6.2 – Чтение данных с диска при использовании DMA

(объяснение в тексте)

DMA-контроллер начинает перенос данных, посылая дисковому контроллеру по шине запрос чтения (*5*). Контроллер диска пересылает данные из своего внутреннего буфера в оперативную память (*6*). Когда запись данных закончена, контроллер диска посылает сигнал подтверждения контроллеру DMA (*7*).

Немного остановимся на прерываниях. Когда устройство ввода-вывода заканчивает свою работу, оно инициирует прерывание. Для этого устройство выставляет сигнал на специальную линию шины. Этот сигнал распознается микросхемой контроллера прерываний, расположенной на материнской плате. Контроллер прерываний принимает решение о дальнейших действиях.

Если нет других необработанных запросов прерывания, то поступившее прерывание обрабатывается немедленно. Если есть, то новый запрос игнорируется. В этом случае устройство продолжает удерживать сигнал прерывания на шине, пока оно не будет обслужено ЦП. При этом процессор приостанавливает текущую работу и начинает выполнять обработку прерывания.

Прерывание, оставляющее машину в строго определенном состоянии, называется **точным прерыванием**. Ему присущи следующие свойства:

1. Счетчик команд сохраняется в известном месте.
2. Все команды до той, на которую указывает счетчик команд, выполнены полностью.

3. Ни одна команда после той, на которую указывает счетчик команд, не была выполнена.

4. Состояние команды, на которую указывает счетчик команд, известно.

Прерывание, не удовлетворяющее всем данным требованиям, называется **неточным прерыванием**. Оно очень осложняет работу ОС, так как в результате в стек выгружается огромное количество данных, чтобы дать возможность ОС определить, что происходило в данный момент.

6.2. Принципы программного обеспечения ввода-вывода

Ключевая концепция разработки программного обеспечения ввода-вывода известна как **независимость от устройств**. Эта концепция означает возможность написания программ, способных получать доступ к любому устройству ввода-вывода без предварительного указания конкретного устройства.

Тесно связан с концепцией независимости от устройств *принцип единообразного наименования*. Имя файла или устройства должно быть просто текстовой строкой или целым числом и никоим образом не зависеть от физического устройства.

Другим важным аспектом программного обеспечения ввода-вывода является *обработка ошибок*. Ошибки должны обрабатываться как можно ближе к аппаратуре. Если контроллер обнаружил ошибку чтения, он должен попытаться по возможности исправить эту ошибку сам. Если не может, то ошибку должен обработать драйвер устройства, возможно, попытавшись прочитать этот блок еще раз. Только если нижний уровень не может справиться с проблемой, о ней следует информировать верхний уровень.

Еще одним ключевым вопросом является *способ переноса данных*: **синхронный** (блокирующий) против **асинхронного** (управляемого прерываниями). Большинство операций ввода-вывода на физическом уровне являются асинхронными – ЦП запускает перенос данных и отправляется заниматься чем-либо другим, пока не придет прерывание. Программы пользователя значительно легче написать, использующие блокирующие операции ввода-вывода – после обращения к системному вызову *read* программа автоматически приостанавливается до тех пор, пока данные не появятся в буфере.

Еще одним аспектом программного обеспечения ввода-вывода является *буферизация*. Часто данные, поступающие с устройства, не могут быть сохранены

сразу там, куда они в конечном итоге направляются. Буферизация включает копирование данных в значительных количествах, что часто является основным фактором снижения производительности операций ввода-вывода.

И последнее понятие – это понятие выделенных устройств и устройств коллективного использования. С некоторыми устройствами ввода-вывода, такими, как диск, может одновременно работать большое количество пользователей. Другие устройства, такие, как накопители на магнитной ленте, должны оставаться в монопольное владение одному пользователю, пока он не завершит работу с этим устройством.

6.3. Программные уровни ввода-вывода

Программное обеспечение ввода-вывода обычно организуется в виде четырех уровней (рисунок 6.3). У каждого уровня есть четко очерченная функция, которую он должен выполнять, и строго определенный интерфейс с соседними уровнями.

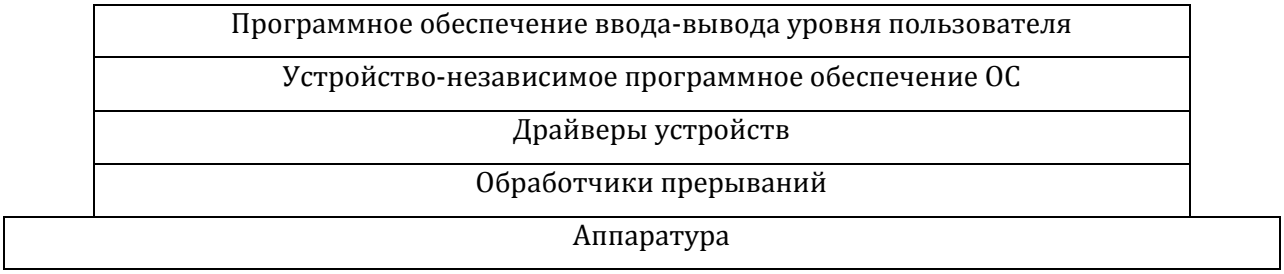


Рисунок 6.3 – Программные уровни ввода-вывода

Прерывания для большинства операций ввода-вывода являются неприятным, но необходимым фактом. Они должны быть упрятаны как можно глубже во внутренностях ОС, чтобы о них знала как можно меньшая ее часть. Лучший способ спрятать их заключается в блокировке драйвера, начавшего операцию ввода-вывода, вплоть до окончания этой операции и получения прерывания.

Когда происходит прерывание, начинает работу обработчик прерываний. По окончании необходимой работы он может разблокировать драйвер, запустивший его.

Для управления каждым устройством ввода-вывода, подключенным к компьютеру, требуется специальная программа – **драйвер устройства**. Она пишется производителем устройства и распространяется вместе с устройством. Каждый драйвер обычно поддерживает один тип устройств или, максимум, класс близких устройств.

Чтобы получить доступ к аппаратной части устройства, то есть к регистрам контроллера, драйвер устройства должен быть частью ядра ОС. Поэтому необходима определенная архитектура, позволяющая установку в ОС кусков программы, написанной другими программистами.

В большинстве ОС определен стандартный интерфейс, который должен поддерживать все блочные драйверы, и второй стандартный интерфейс, поддерживаемый всеми символьными драйверами. Эти интерфейсы включают наборы процедур, которые могут вызываться остальной ОС для обращения к драйверу.

Многие драйверы устройств обладают сходной общей структурой. Типичный драйвер начинает с проверки входных параметров (рисунок 6.4). Если они не удовлетворяют определенным критериям, драйвер возвращает ошибку. В противном случае драйвер преобразует абстрактные термины в конкретные. Например, дисковый драйвер может преобразовывать линейный номер блока в номера головки, дорожки и секторы.

Затем драйвер проверяет, не используется ли это устройство в данный момент. Если устройство занято, запрос ставится в очередь. Если свободно, то проверяется аппаратный 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 подразумевает выдачу серии команд устройству. Именно в драйвере определяется последовательность команд в зависимости от необходимости. После передачи команд 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 в ожидании прерывания от устройства. В других случаях операция завершается без задержек и драйверу блокироваться не нужно.

В любом случае по завершении выполнения операции драйвер должен проверить, завершилась ли операция без ошибок.

Хотя некоторая часть программного обеспечения ввода-вывода предназначена для работы с конкретными устройствами, другая часть является независимой от устройств. Основная задача независимого от устройств программного обеспечения состоит в выполнении функций ввода-вывода, общих

для всех устройств, и предоставлении единообразного интерфейса для программ уровня пользователя.

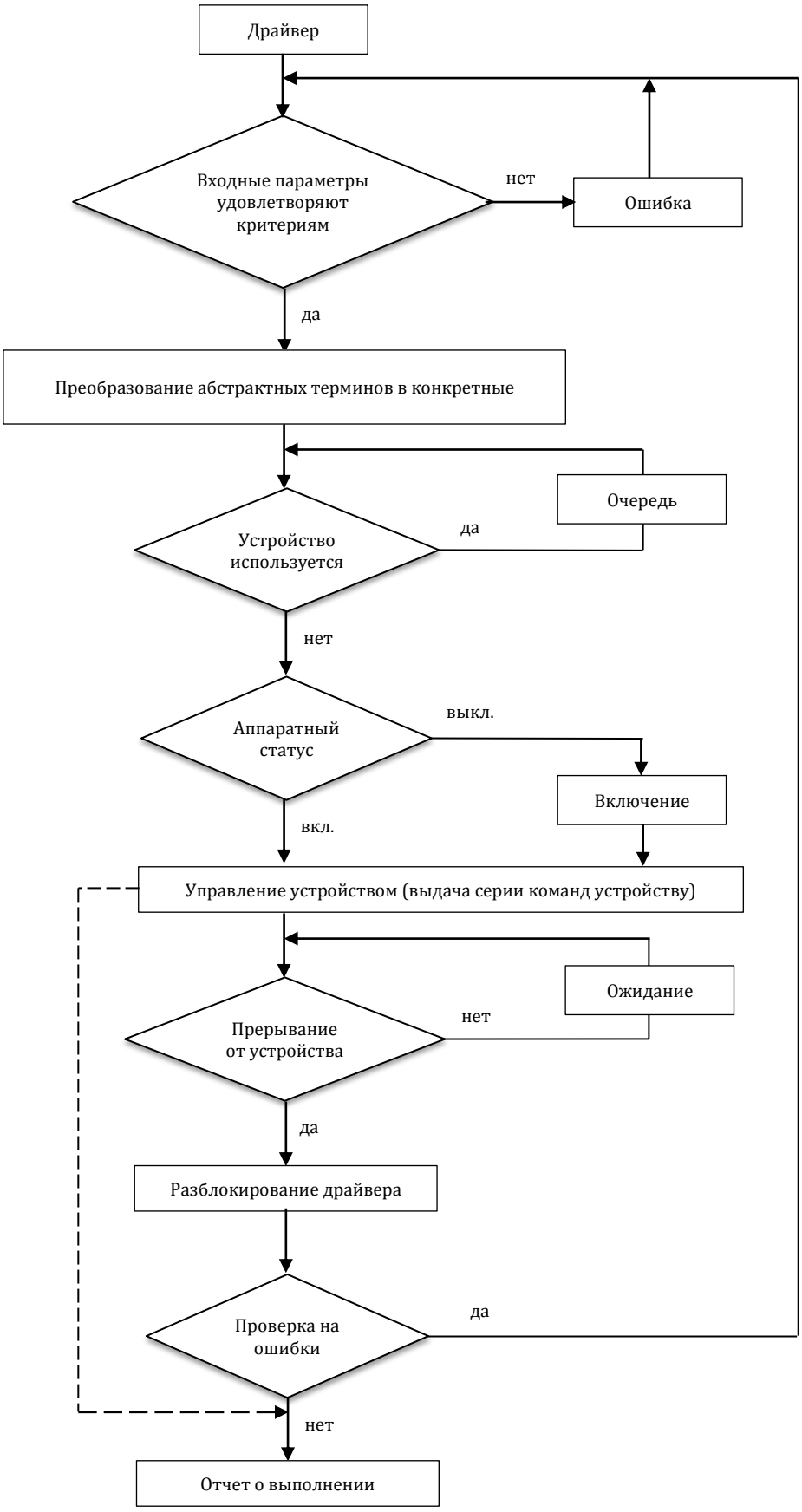


Рисунок 6.4 – Алгоритм работы драйвера

В независимом от устройств программном обеспечении обычно реализуются следующие функции:

1. Единообразный интерфейс для драйверов устройств.
2. Буферизация.
3. Сообщение об ошибках.
4. Захват и освобождение выделенных устройств.
5. Размер блока, не зависящий от устройства.

Программное обеспечение ввода-вывода пространства пользователя состоит из библиотечных процедур и спулинга (подкачки). **Спулинг** – это способ работы с выделенными устройствами в многозадачной системе. Типичное устройство, на котором используется спулинг – это принтер. Чтобы распечатать файл, процесс сначала создает специальный файл, предназначенный для печати, который помещает в *каталог спулинга*. Этот файл печатает *демон*, единственный процесс, которому разрешается пользоваться специальным файлом принтера. Спулинг также используется при передаче файлов по сети.

6.4. Диски

Существует множество типов дисков. К наиболее часто встречающимся относятся магнитные диски (гибкие и жесткие). Их особенностью является одинаковая скорость чтения и записи, что делает их идеальными в качестве дополнительной памяти. С целью создания высоконадежного устройства хранения используются наборы жестких магнитных дисков. Для распространения программ, данных и фильмов используются различные виды оптических дисков (CD-ROM, DVD-ROM, BR-D и т.д.).

Магнитные диски организованы в цилиндры, каждый из которых содержит столько дорожек, сколько есть у устройства головок. Диски разделены на зоны с большим числом секторов на внешних дорожках и меньшим на внутренних. Чтобы не усложнять жизнь ОС излишними подробностями, современные жесткие диски скрывают истинную геометрию и предоставляют в качестве интерфейса виртуальную геометрию с одинаковым числом секторов на всех цилиндрах. Встроенный микроконтроллер диска преобразует обращение к виртуальным цилиндру, головке и сектору в соответствующие физические параметры.

Оптические диски организованы совершенно по-другому. Данные на них записываются в виде спирали. Чтобы считывание данных с диска производилось с

одинаковой скоростью, по мере продвижения считывающей головки от центра диска к краю, угловая скорость вращения компакт-диска должна постоянно уменьшаться в соответствии с формулой (6.1).

$$v = r \cdot \omega, \tag{6.1}$$

где v – скорость считывания данных,

r – расстояние от центра диска до точки, в которой происходит считывание данные,

ω – угловая скорость вращения диска

Отличия между разными типами оптических дисков заключаются в следующем: для CD-ROM в качестве основы для записи используется поликарбонатная смола с углублениями (**питами**), соответствующими наличию данных и промежутками (**землей**). В CD-R запись осуществляется на слое красителя, где темные пятна являются аналогами питов. У CD-RW вместо красителя используется сплав серебра, индия, сурьмы и теллура. Этот сплав имеет два устойчивых состояния – аморфное (питы) и кристаллическое (промежутки). Кроме того, в устройствах для записи CD-RW используются лазеры с тремя уровнями мощностями. При высокой мощности осуществляется преобразование из кристаллического состояния в аморфное (создание питов), при средней – преобразование в кристаллическое (создание промежутков), при низкой – считывание. У DVD-дисков вдвое меньший размер питов (0,4 мкм против 0,8), более тугая спираль (0,74 мкм между соседними дорожками вместо 1,6) и красный лазер (с длиной волны 0,65 мкм против 0,78). В BR-дисках числовая апертура 0,85 против 0,6 у DVD-дисков, пит – 0,14 мкм, используется синий лазер с длиной волны 0,405 мкм, а защитный слой более тонкий – 100 мкм вместо 600 у DVD-дисков.

6.5. Таймеры

Таймеры очень важны для работы любой многозадачной системы по ряду причин. Среди многих других задач, они следят за временем суток и не позволяют одному процессу надолго занять центральный процессор. Программное обеспечение таймера может принимать форму драйвера устройства, несмотря на то, что таймер не является ни блочным устройством, ни символьным.

Ранее использовались таймеры, вызывающие прерывание при каждом цикле напряжения с частотой 50 или 60 Гц. Сейчас применяются таймеры, состоящие из трех компонентов: кварцевого генератора, счетчика и регистра хранения. Сигнал от

кварцевого генератора подается на вход декрементного счетчика. Когда содержимое счетчика достигает нуля, он вызывает прерывание ЦП.

Всё, что делает таймер аппаратно – он инициирует прерывания через определенный промежуток времени. Все остальное должно выполняться программно драйвером часов. Обычные *функции драйвера часов* следующие:

1. Они следят за временем суток.
2. Не позволяют процессам работать дольше, чем им разрешено.
3. Ведут учет использования центрального процессора.
4. Обрабатывают системный вызов *alarm*, инициированный процессом пользователя.
5. Поддерживают следящие таймеры для ОС.
6. Ведут наблюдение, анализ и сбор статистики.

6.6. Автономные терминалы

У каждого универсального компьютера есть, по крайней мере, одна клавиатура и один дисплей (монитор), используемые для общения с компьютером. Хотя они являются технически отдельными устройствами, они сообща образуют пользовательский интерфейс. К мэйнфреймам часто присоединяются специальные устройства, состоящие из клавиатуры и дисплея, за которыми работают удаленные пользователи. Такие устройства исторически называются **терминалами**.

Существует три типа терминалов:

1. Автономные терминалы с последовательным интерфейсом RS-232 для связи с мэйнфреймами.

2. Дисплеи персональных компьютеров с графическим интерфейсом пользователя.

3. Сетевые терминалы.

Терминалы с интерфейсом RS-232 представляют собой технические устройства, состоящие из клавиатуры и дисплея, общающиеся по последовательному интерфейсу. Чтобы послать символ по линии последовательной передачи на терминал с интерфейсом RS-232 или модем, компьютер должен передавать данные по одному биту, начиная передачу каждого символа со стартового бита и заканчивая одним или двумя стоповыми битами для разделения символов.

Терминалы с интерфейсом RS-232 являются алфавитно-цифровыми терминалами. Это означает, что экран отображает определенное количество строк текста. Обычный размер такого окна составляет 25 строк по 80 символов.

6.7. Графические интерфейсы пользователя

В настоящее время на персональных компьютерах используется **графический интерфейс пользователя** (GUI, Graphical User Interface). Графический интерфейс пользователя состоит их четырех основных элементов, из первых букв которых можно сложить слово WIMP – Windows, Icons, Menus, Pointing device – окна, иконки, меню, указывающее устройство. Программное обеспечение графического интерфейса пользователя может быть реализовано на уровне пользователя, как в семействе систем UNIX, или включено в операционную систему, как в случае Windows.

Связь с клавиатурой может осуществляться через последовательный, параллельный порт или порт USB. При нажатии любой клавиши ЦП прерывается и драйвер клавиатуры извлекает символ, читая порт ввода-вывода. Все остальное осуществляется программно, в основном в драйвере клавиатуры.

Для реализации графического интерфейса необходимо наличие манипулятора типа "мышь" или его аналогов (тач-падов, трекболов и т.д.) либо сенсорных панелей. Сообщение, посылаемое компьютеру от "мыши", содержит три параметра: изменение позиции по координатам x и y; Δx и Δy; состояние кнопок. Формат сообщения зависит от системы и числа кнопок "мыши". Обычно оно занимает 3 байт. При этом надо обратить внимание, что "мышь" сообщает только об изменениях своей позиции, а не об абсолютном значении позиции.

Рассмотрим теперь аппаратную часть дисплея. Все современные дисплеи используют растровую графику, хотя ранее в них часто применялась векторная. Область вывода дисплея представляет собой прямоугольную сетку точек, называемых **пикселами**, каждая из которых может принимать различные значения яркости и цвета.

Реализация графических дисплеев осуществляется с помощью специального устройства – **графического адаптера**. Он содержит специальную память – **видео-ОЗУ** или **видеопамять**, образующую часть адресного пространства компьютера. Это означает, что центральный процессор обращается к ней так же, как и к остальной оперативной памяти. Здесь хранится образ экрана. В последнее время графические

адаптеры стали оснащаться отдельным процессором, что позволило значительно разгрузить центральный процессор, позволив максимально освободить его от управления выводом информации на дисплей, а также значительно ускорить работу с графикой.

6.8. Сетевые терминалы

Сетевые терминалы используются для того, чтобы соединить удаленного пользователя с компьютером по локальной или глобальной сети. Существует две различные концепции, касающиеся способа работы сетевых терминалов. Одна точка зрения заключается в том, что сетевой терминал должен обладать огромной вычислительной мощностью и памятью, что должно позволить работать на нем сложным протоколам и снизить объем данных, пересылаемых по сети. **Протокол** – набор запросов и ответов, о которых договариваются отправитель и получатель, чтобы общаться по сети или другому интерфейсу. Другая точка зрения состоит в том, что терминал должен быть максимально простым и дешевым, в основном занимающимся лишь отображением пикселов на экране.

Хотя децентрализованная модель персональных компьютеров обладает определенными преимуществами, у нее есть также серьезные недостатки, которые только последнее время начинают серьезно рассматриваться. Возможно, самая большая проблема персональных компьютеров состоит в том, что у каждого ПК имеется большой жесткий диск и сложное программное обеспечение, которым требуется управлять. Соответственно, необходимо осуществлять установку и обновление версий программного обеспечения на каждом компьютере. Кроме того, сложнее осуществлять архивирование данных, которое проще выполнять централизованно.

Еще одно преимущество централизованной системы состоит в том, что в этом случае упрощается совместное использование ресурсов. Таким образом, в крупных сетях в большинстве случаев необходимы так называемые **"тонкие" клиенты**. Идеалом была бы высокопроизводительная интерактивная компьютерная система, в которой машина пользователя вообще не имела бы программного обеспечения.

6.9. Управление режимом энергопотребления

Первый универсальный электронный компьютер состоял из 18 000 электронных ламп и потреблял 140 кВт. После изобретения транзистора

потребление электроэнергии снизилось на несколько порядков. Однако сегодня по различным причинам управление режимом электропитания снова оказалось в центре внимания. Не менее остро стоит проблема электропотребления в ноутбуках и других переносных устройствах.

К снижению потребления электроэнергии существует два основных подхода. Первый из них заключается в выключении операционной системой тех частей компьютера (главным образом, устройств ввода-вывода), которые не используются в данный момент. Второй подход состоит в снижении потребления энергии прикладными программами, возможно за счет снижения качества восприятия пользователем.

Основной метод, предпринимаемый многими производителями компьютеров для продления жизни батарей и снижения энергопотребления состоит в проектировании центрального процессора, памяти и устройств ввода-вывода, способных находиться в нескольких режимах энергопотребления: включенном, режиме ожидания, гибернации и в выключенном состоянии. Переводом устройства из одного режима в другой в соответствующий момент должна управлять операционная система.

Главная проблема состоит в том, чтобы найти алгоритмы, позволяющие ОС принимать правильные решения, касающиеся энергопотребления. При этом, согласно проведенным исследованиям, основными потребителями электроэнергии являются экран, жесткий диск и центральный процессор.

Больше всех энергии потребляет дисплей, так как чтобы изображение было ярким и контрастным, экран должен иметь подсветку. Многие ОС пытаются сберегать энергию, отключая дисплей, когда в течение определенного интервала времени не наблюдается активности со стороны пользователя. Обычно этот интервал настраивается самим пользователем.

Был предложен также другой вариант – экран разбивается на несколько зон, часть из которых может светиться, если там находится активное окно, а часть остается темной.

Следующим по величине потребляемой энергии является жесткий диск. Он потребляет значительное количество энергии для поддержания высокой скорости вращения даже при отсутствии обращений к нему. Поэтому многие компьютеры останавливают жесткий диск, если к нему долго нет обращений. А когда он нужен – запускают снова. Однако при этом возникает задержка в несколько секунд,

требующаяся для его раскрутки. Кроме того, на раскрутку также требуется дополнительная энергия.

Другой способ – это поддержание большого дискового кэша в оперативной памяти. Если нужный блок находится в памяти, обращение к диску не происходит, в результате чего диск можно не раскручивать.

Еще один способ избежать лишних запусков жесткого диска состоит в предоставлении ОС информации о состоянии диска работающей программе. Некоторые программы производят периодические операции записи на диск, которые можно отложить или пропустить.

Центральный процессор также может находиться в разных режимах энергопотребления. При этом при снижении напряжения питания центрального процессора вдвое его производительность снизится также в два раза, зато энергии он будет потреблять в четыре раза меньше ввиду квадратичной зависимости. Так, если, например, программа просмотра мультимедиа должна распаковывать и отображать каждые 40 мс один кадр, но успевает это сделать за 20 мс, то оставшиеся 20 мс процессор может находиться в режиме низкого энергопотребления. Однако если снизить его напряжение на все 40 мс в 2 раза, то потреблять он будет в 4 раза меньше, а работу выполнит в течение 40 мс. Таким образом, медленная работа является более эффективной с энергетической точки зрения.

Для энергосбережения при работе с памятью возможны два подхода. Во-первых, можно периодически сохранять и выключать кэш. Его всегда можно перезагрузить из оперативной памяти без потерь информации. Более радикальный метод заключается в сохранении содержимого оперативной памяти на диске, после чего может быть выключена сама оперативная память. Когда память выключена, то и центральный процессор стоит выключить, так как обрабатывать ему будет практически нечего.

Наряду с проблемой энергосбережения остро стоит и проблема управления температурным режимом. Из-за высокой частоты современные центральные процессоры очень сильно греются. Настольные машины оснащаются внутренним электрическим вентилятором, сдувающим горячий воздух с шасси. Поскольку для настольных компьютеров снижение энергопотребления не является вопросом жизни и смерти, вентилятор обычно включен постоянно.

С мобильными устройствами ситуация иная. ОС должна постоянно следить за температурой узлов компьютера. Когда температура начинает приближаться к

максимально допустимому значению, у ОС есть выбор – включить вентилятор, который шумит и потребляет энергию, или уменьшить энергопотребление рассмотренными выше способами.

7. ФАЙЛОВАЯ СИСТЕМА

7.1. Основные понятия

Всем компьютерным приложениям нужно хранить и получать информацию. Во время работы процесс может хранить ограниченное количество данных в собственном адресном пространстве. Однако емкость такого хранилища ограничена размерами виртуального адресного пространства.

Кроме того, после завершения работы процесса информация, хранящаяся в его адресном пространстве, теряется. В то же время, для большинства приложений (например, баз данных) эта информация должна храниться неделями, месяцами или даже вечно.

Третья проблема состоит в том, что часто возникает необходимость нескольким процессам одновременно получить доступ к одним и тем же данным.

Следовательно, возникает необходимость отделения информации от процесса. Таким образом, долговременным устройствам хранения информации предъявляются следующие важные требования:

1. Устройства должны позволять хранить очень большие объемы данных.
2. Информация должна сохраняться после прекращения работы процесса, использующего ее.
3. Несколько процессов должны иметь возможность получения одновременного доступа к информации.

Обычное решение всех этих проблем состоит в хранении информации на дисках и других внешних хранителях в модулях, называемых **файлами**. Процессы по мере надобности могут читать их и создавать новые файлы. Информация, хранящаяся в файлах, должна обладать **устойчивостью** (**персистентностью**), то есть на нее не должны оказывать влияния создание или прекращение работы какого-либо процесса. Файл должен исчезать только тогда, когда его владелец дает команду удаления файла.

Файлами управляет ОС. Их структура, именование, использование, защита, реализация и доступ к ним являются важными пунктами устройства ОС. Часть ОС, работающая с файлами, называется **файловой системой**.

С точки зрения пользователя наиболее важным аспектом файловой системы является ее внешнее представление, то есть именование и защита файлов, операции с файлами и т.д.

Такие же детали внутреннего устройства, как использование связанных списков или бит-карт для слежения за свободными и занятыми блоками диска, число физических секторов в логическом блоке более важны для разработчиков.

7.2. Файлы и каталоги

Файлы относятся к абстрактному механизму. Они предоставляют способ сохранять информацию на диске и считывать ее позднее. При этом от пользователя должны скрываться такие детали, как способ и место хранения информации, а также детали работы дисков.

Наиболее важной характеристикой любого механизма абстракции является то, как именуются управляемые объекты. При создании файла процесс дает файлу имя. Когда процесс завершает работу, файл продолжает свое существование и по его имени к нему могут получить доступ другие процессы.

Точные правила именования файлов различны для разных ОС, но все современные системы поддерживают использование в качестве имен файлов 8-символьные текстовые строки. Многие файловые системы поддерживают имена файлов длиной до 255 символов.

В некоторых файловых системах, например, UNIX или Linux, различаются прописные и строчные символы, тогда как в других, таких как Windows, фактически нет.

Во многих ОС имя файла может состоять из двух частей, разделенных точкой. Часть имени файла после точки называется **расширением файла** и обычно обозначает тип файла. Однако в системах на основе UNIX размер расширения зависит от пользователя, сами расширения являются просто соглашениями, и ОС не принуждает пользователя их строго придерживаться, к тому же их может быть несколько у одного файла.

Система Windows, напротив, знает о расширениях файлов и назначает каждому расширению определенное значение. Пользователи (или процессы) могут регистрировать расширения в ОС, указывая программу, "владеющую" данным расширением.

Все файлы имеют определенную структуру (рисунок 7.1). Самый простой вариант – когда файл представляет собой *неструктурированную последовательность байтов*. В этом случае ОС не интересуется содержимым файла.

Все, что она видит – это байты. Значение этим байтам придается программами на уровне пользователя. Такой подход используется в системах UNIX и Windows.

Рассмотрение ОС файлов как просто последовательностей байтов обеспечивает максимальную гибкость. Программы пользователей могут помещать в файлы все, что угодно и именовать их любым удобным для них способом. ОС не вмешивается в этот процесс.

Вторая модель структуры представляет собой *последовательность записей фиксированной длины*, каждая со своей структурой. Для таких файлов операция чтения возвращает одну запись, а операция записи перезаписывает или дополняет одну запись. Современные системы так уже не работают.

Третий вариант файловой структуры представляет собой *дерево записей*, не обязательно одной и той же длины. Каждая запись в фиксированной позиции содержит поле **ключа**. Дерево сортировано по ключевому полю, что обеспечивает быстрый поиск заданного ключа. Основной файловой операцией здесь является не получение следующей записи, хотя это также возможно, а получение записи с указанным значением ключа. При добавлении новых записей ОС, а не пользователь должна решать, куда ее поместить. Такой тип структуры файлов применяется на мэйнфреймах.

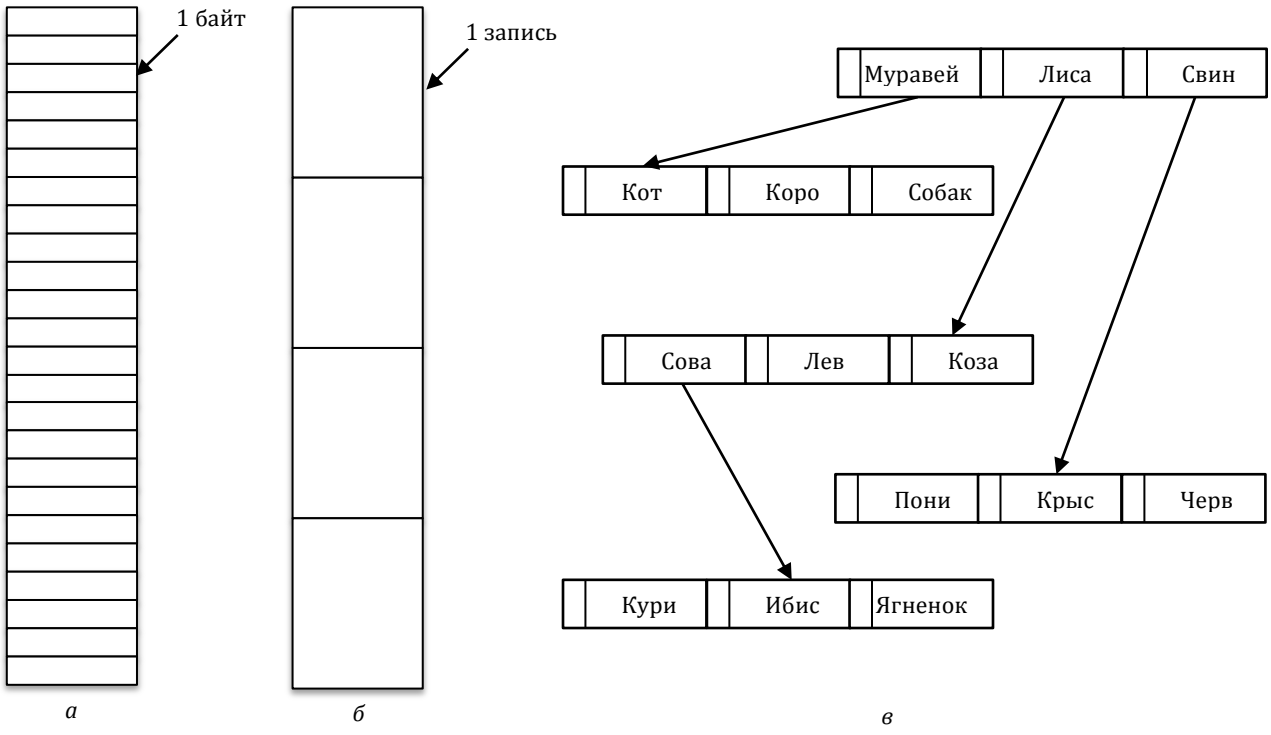


Рисунок 7.1 – Три типа файлов: последовательность байтов (*а*); последовательность записей (*б*); дерево (*в*)

Все файлы делятся на несколько типов:

1. Регулярные (обычные) – все файлы, содержащие информацию пользователя. Регулярные файлы бывают ASCII-файлами, состоящими из текстовых строк, либо двоичными файлами. Основным преимуществом ASCII-файлов является то, что они могут отображаться на экране и печататься так, как есть, без какого-либо преобразования и могут редактироваться любым текстовым редактором.

Например, на рисунке 7.2 показан простой исполняемый двоичный файл одной из версий системы UNIX. Хотя технически файл представляет собой всего лишь последовательность байтов, операционная система станет исполнять файл только в том случае, если этот файл имеет соответствующий формат. Файл состоит из пяти разделов: заголовка, текста, данных, релокационных битов и таблицы символов. Заголовок начинается с так называемого "магического числа", идентифицирующего файл как исполняемый (чтобы предотвратить случайное исполнение файла другого формата). Следом за "магическим числом" в заголовке располагаются размеры различных частей файла, адрес начала исполнения файла и некоторые флаговые биты. За заголовком следуют текст программы и данные. Они загружаются в оперативную память и настраиваются на работу по адресу загрузки при помощи битов релокации. Таблица символов используется для отладки.

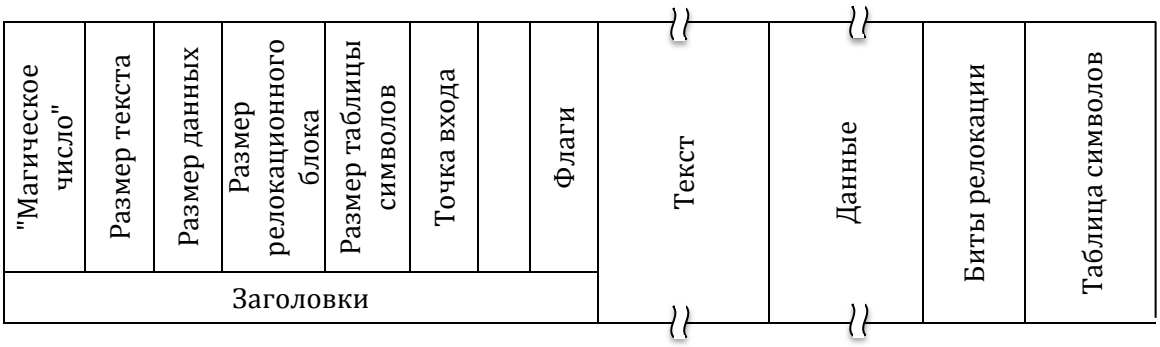


Рисунок 7.2 – Структура исполняемого файла

2. Каталоги – системные файлы, обеспечивающие поддержку структуры файловой системы. Их мы рассмотрим чуть позже.

3. Символьные специальные – применяются в UNIX и имеют отношение к вводу-выводу и используются для моделирования последовательных устройств ввода-вывода, таких как терминалы, принтеры и сети.

4. Блочные специальные – также применяются в UNIX для моделирования дисков.

Рассмотрим варианты доступа к файлам. В старых ОС предоставлялся только один тип доступа – **последовательный**. В этих системах процесс мог читать байты или записи файла только по порядку от начала к концу. Такой доступ к файлам появился, когда дисков еще не было и компьютеры оснащались магнитофонами. Поэтому даже в дисковых ОС при последовательном доступе к файлу имитировалось его чтение или запись на магнитной ленте.

С появлением дисков стало возможным читать байты или записи файла в произвольном порядке или получать доступ к записям по ключу. Файлы, байты которых могут быть прочитаны в произвольном порядке, называются **файлами произвольного доступа**.

Файлы произвольного доступа очень важны для многих приложений, например, для баз данных. Если клиент звонит в авиакомпанию с целью зарезервировать место на конкретный рейс, программа резервирования авиабилетов должна иметь возможность получить доступ к нужной записи, не читая все тысячи предшествующих записей, содержащих информацию о других рейсах.

Для указания места начала чтения используются два метода. В первом случае каждая операция *read* задает позицию в файле. При втором способе используется специальная операция *seek*, устанавливающая текущую позицию. После выполнения этой операции файл может читаться последовательно с текущей позиции.

В некоторых старых операционных системах, использовавшихся на мэйнфреймах, способ доступа к файлу (последовательный или произвольный) указывался в момент создания файла. Это позволяло операционной системе применять различные методы для хранения файлов разных классов. В современных операционных системах такого различия не проводится. Все файлы автоматически являются файлами произвольного доступа.

Рассмотрим теперь, какие атрибуты могут быть у файлов. У каждого файла есть имя и данные. Помимо этого все ОС связывают с каждым файлом также и другую информацию, например дату и время создания файла, его размер и т.д. Такие дополнительные сведения называются **атрибутами файла**. Список атрибутов различен для разных ОС. Есть атрибуты, связанные с защитой файла. Они содержат информацию о том, кто может получить доступ к файлу, а кто нет. Возможны различные схемы реализации защиты файла. В некоторых системах пользователь должен для получения доступа к файлу указать пароль. В этом случае пароль должен входить в атрибуты файла.

В качестве атрибутов также могут выступать различные флаги. Флаги представляют собой биты или короткие поля, управляющие некоторыми специфическими свойствами. Например, скрытые файлы не появляются в перечне файлов при распечатке каталога. Флаг архивации представляет собой бит, следящий затем, была ли создана для файла резервная копия. Этот флаг очищается программой архивирования и устанавливается операционной системой при изменении файла. Таким образом программа архивирования может определить, какие файлы следует архивировать. Флаг "временный" позволяет автоматически удалять помеченный так файл по окончании работы создавшего его процесса.

Атрибуты длина записи, позиция ключа и длина ключа присутствуют только у тех файлов, записи которых могут искаться по ключу. Эти атрибуты предоставляют необходимую для поиска ключа информацию.

Различные атрибуты, хранящие значения времени, позволяют следить за тем, когда файл был создан, в последний раз изменен и когда к нему в последний раз предоставлялся доступ. Эти сведения можно использовать в различных целях. Например, если исходный файл программы был модифицирован после создания соответствующего ему объектного файла, то исходный файл должен быть перекомпилирован.

Текущий размер файла содержит количество байтов в файле в настоящий момент. В некоторых старых операционных системах, использовавшихся на мэйнфреймах, при создании файла требовалось указать также максимальную длину файла, что позволяло операционной системе зарезервировать достаточно места для последующего увеличения файла. Современные операционные системы, работающие на персональных компьютерах и рабочих станциях, умеют обходиться без подобного резервирования.

Рассмотрим теперь операции с файлами. Все файлы позволяют сохранять информацию и получать ее позднее. В различных ОС имеются различные наборы файловых операций. Наиболее часто встречающимися системными вызовами, относящимися к файлам являются:

1. **Create** (создание). Файл создается без данных. Этот системный вызов объявляет о появления нового файла и позволяет установить некоторые его атрибуты.

2. **Delete** (удаление). Когда файл уже более не нужен, его удаляют, чтобы освободить пространство на диске. Этот системный вызов присутствует в каждой операционной системе.

3. **Open** (открытие). Прежде, чем использовать файл, процесс должен его открыть. Системный вызов *open* позволяет системе прочитать в оперативную память атрибуты файла и список дисковых адресов для быстрого доступа к содержимому файла при последующих вызовах.

4. **Close** (закрытие). Когда все операции с файлом закончены, атрибуты и дисковые адреса более не нужны, поэтому файл следует закрыть, чтобы освободить пространство во внутренней таблице. Многие операционные системы позволяют одновременно открыть ограниченное количество файлов. Запись на диск производится поблочно, а закрытие файла вызывает запись последнего блока файла, даже если этот блок еще не заполнен до конца.

5. **Read** (чтение). Чтение данных из файла. Обычно байты поступают с текущей позиции в файле. Вызывающий процесс должен указать количество требуемых данных и предоставить для них буфер.

6. **Write** (запись). Запись данных в файл, также в текущую позицию в файле. Если текущая позиция находится в конце файла, размер файла автоматически увеличивается. В противном случае запись производится поверх существующих данных, которые теряются навсегда.

7. **Append** (добавление). Этот системный вызов представляет собой усеченную форму вызова *write*. Он может только добавлять данные к концу файла. В операционных системах с минимальным набором системных вызовов может не быть данного системного вызова.

8. **Seek** (поиск). Для файлов произвольного доступа требуется способ указать, где располагаются данные в файле. Данный системный вызов устанавливает файловый указатель в определенную позицию в файле. После выполнения данного системного вызова данные могут читаться или записываться в этой позиции.

9. **Get attributes** (получение атрибутов). Процессам часто для выполнения их работы бывает необходимо получить атрибуты файла. Например, для сборки программ, состоящих из большого числа отдельных исходных файлов, в системе UNIX часто используется программа *make.* Эта программа исследует время изменения всех исходных и объектных файлов, благодаря чему обходится

компиляцией минимального количества файлов. Для выполнения этой работы ей требуется получить атрибуты файлов.

10. **Set attributes** (установка атрибутов). Некоторые атрибуты файла могут устанавливаться пользователем после создания файла. Этот системный вызов предоставляет такую возможность. Например, для файла может быть установлен код защиты доступа. Большинство других флагов также могут устанавливаться при помощи данного системного вызова.

11. **Rename** (переименование). Этот системный вызов позволяет изменить имя файла. Его присутствие в операционной системе не является необходимым, так как обычно файл можно скопировать с новым именем, а старый файл удалить.

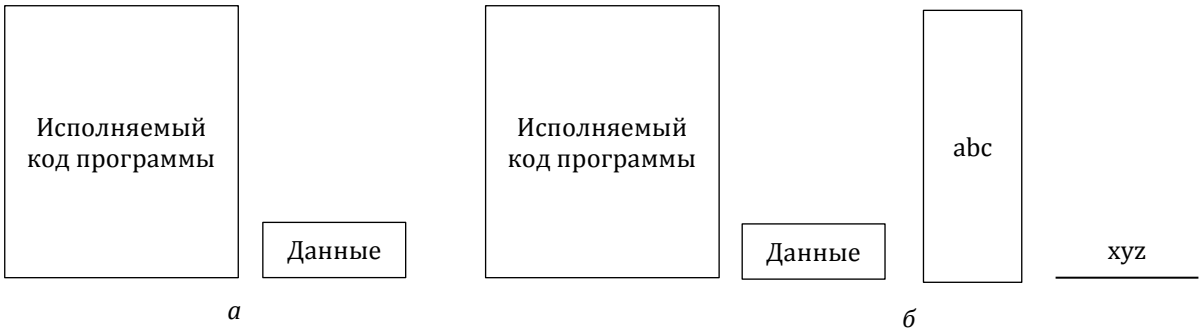
В некоторых операционных системах, начиная с системы MULTICS, был предоставлен способ отображения файлов на адресное пространство работающего процесса. Концептуально можно представить себе два новых системных вызова, *map* и *unmap*. Первый системный вызов принимает на входе два параметра: имя файла и виртуальный адрес памяти, по которому операционная система отображает указанный файл.

Предположим, например, что некий файл размером 64 Кбайт отображается на виртуальную память, начиная с адреса 512 К. После этого любая команда процессора, читающая байт по адресу 512 К, получит байт 0 этого файла и т.д. Запись по адресу 512 К + 21000 изменит байт 21000 файла. Когда процесс завершает свою работу, модифицированный файл остается на диске, как если бы он был изменен системными вызовами *seek* и *write*.

Для реализации отображения файлов на память изменяются системные внутренние таблицы. При обращении к памяти по адресу от 512 до 576 К происходит прерывание из-за отсутствия страницы, обработчик которого предоставляет считанную в память страницу 0 файла. При записи происходит приблизительно то же самое, но страница памяти, на которую отображается страница файла, помечается как модифицированная. Если потом эта страница удаляется из памяти алгоритмом замены страниц, она записывается в соответствующее место файла. После завершения процесса все модифицированные страницы сохраняются в соответствующих файлах.

Отображение файлов на память лучше всего работает в операционной системе, поддерживающей сегментацию, В такой системе каждый файл может быть отображен на свой собственный сегмент, так чтобы байт *k* файла был также байтом

k сегмента. На рисунке 7.3, *а* показан процесс с двумя сегментами, исполняемым кодом программы и данными. Предположим, что процесс копирует файл. Сначала он отображает на сегмент исходный файл, например *abc*. Затем он создает пустой сегмент и отображает его на выходной файл, *xyz*. В результате получается ситуация, показанная на рисунке 7.3, *б*.



а *б*

Рисунок 7.3 – Сегментированный процесс до отображения файла на адресное пространство (*а*); процесс после отображения существующего файла *abc* на один сегмент и создания нового сегмента для файла *xyz* (*б*)

В этот момент процесс может скопировать сегмент-источник в сегмент-приемник с помощью обычного цикла копирования памяти. При этом не требуются системные вызовы *read* и *write*. Когда копирование закончено, процесс может выполнить системный вызов *unmap*, чтобы удалить эти файлы из адресного пространства, и завершить свою работу. Выходной файл, *xyz*, теперь будет существовать на диске, как если бы он был создан обычным путем.

Хотя отображение на память устраняет необходимость обращения к системным вызовам ввода-вывода, вместе с ним появляются новые проблемы. Во-первых, в нашем примере операционной системе трудно определить длину выходного файла *xyz*. Операционная система знает номер максимальной модифицированной страницы, но она не может определить, сколько байтов было записано в эту страницу. Предположим, программа использует только страницу 0 и после выполнения цикла копирования все байты остались равны 0 (их исходному значению). Возможно, файл *xyz* состоит из 10 нулей. Может быть, он должен состоять из 100 нулей. Кто знает? Операционная система не может этого сказать. Все, что она может сделать – это создать файл, длина которого равна размеру страницы.

Вторая проблема может возникнуть при попытке одного процесса открыть файл, уже отображенный на адресное пространство другого процесса. Если первый процесс модифицирует страницу, это изменение не отразится на файле до тех пор,

пока эта страница не будет сохранена в файле. Операционная система должна прилагать особые усилия, чтобы гарантировать, что оба процесса не работают с устаревшими версиями файла.

Третья проблема, связанная с отображением файлов на память, вызвана тем, что файл может оказаться больше сегмента памяти и даже больше, чем все виртуальное адресное пространство. При этом единственный способ работы системного вызова *map* состоит в отображении на память части файла. Хотя такой метод и работает, он все же менее удобен, чем отображение всего файла целиком.

Теперь перейдем к рассмотрению каталогов. В файловых системах файлы обычно организуются в **каталоги** или **папки**, которые, в свою очередь, в большинстве ОС также являются файлами.

Простейшая форма системы каталогов состоит в том, что имеется один каталог, в котором содержатся все файлы. Иногда его называют корневым каталогом, но поскольку он в таких системах единственный, его название не имеет значение. Такая система была распространена на ранних персональных компьютерах, в частности потому, что у них было всего по одному пользователю. Первый в мире суперкомпьютер CDC 6600 (1964 г.) также имел всего один каталог для всех файлов, несмотря на то, что на нем одновременно работало много пользователей. Это решение было принято для сохранения простоты программного обеспечения.

Схематично однокаталоговая система показана на рисунке 7.4. В данном примере каталог состоит из четырех файлов. На рисунке буквами A, B и C показаны не имена файлов, а их владельцы (так как именно наличие нескольких пользователей в такой системе создает проблемы). Преимуществом такой схемы является ее простота и способность быстро находить файлы, так как они могут располагаться только в одном месте.

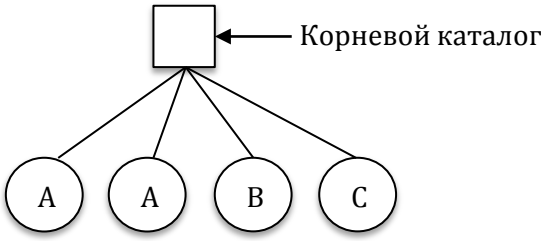


Рисунок 7.4 – Однокаталоговая система с четырьмя файлами трех владельцев

Недостаток системы с одним каталогом и несколькими пользователями состоит в том, что различные пользователи могут случайно использовать для своих

файлов одинаковые имена. Например, если пользователь A создаст файл mailbox, а затем пользователь B также создаст файл mailbox, то файл, созданный пользователем B, запишется поверх файла, созданного пользователем A. Поэтому такая схема более не используется в многопользовательских системах, но может применяться в небольших встроенных системах, например автомобильной системе, предназначенной для хранения профилей пользователей для небольшого количества водителей.

Первым этапом в деле решения проблемы одинаковых имен файлов, созданных различными пользователями, можно считать систему, в которой каждому пользователю выделяется один каталог. Она представляет собой один общий **корневой каталог**, в котором располагаются каталоги пользователей, содержащие отдельные файлы. При этом имена файлов, созданных одним пользователем, не конфликтуют с именами файлов другого пользователя. Схематично такая двухуровневая каталоговая система проиллюстрирована на рисунке 7.5. Буквы обозначают владельцев каталогов и файлов. Такая организация могла, например, использоваться на многопользовательском компьютере или в простой сети персональных компьютеров, соединенных с общим файловым сервером локальной сетью.

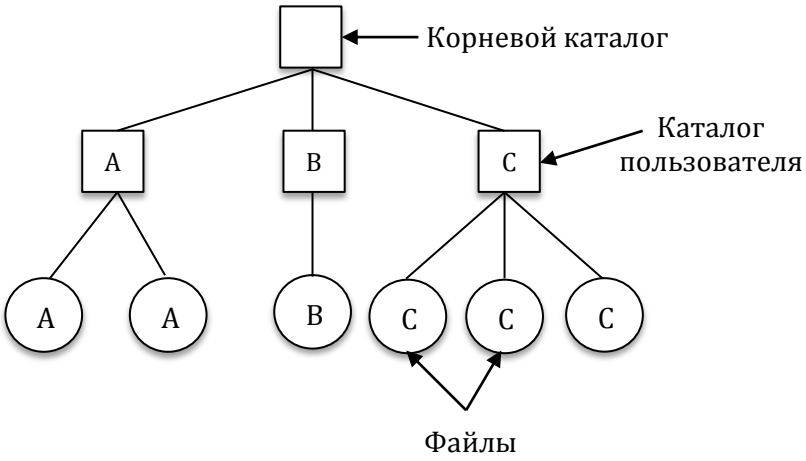


Рисунок 7.5 – Двухуровневая каталоговая система

Когда при такой схеме пользователь пытается открыть файл, система знает, что это за пользователь, и ищет файл в соответствующем каталоге. Следовательно, для работы в такой системе требуется начальная регистрация пользователя, при которой пользователь указывает свое имя или идентификатор. В одноуровневой каталоговой системе такая процедура не требовалась.

При реализации такой системы в ее базовой форме пользователи могут получать доступ только к файлам в своем собственном каталоге. Однако небольшая модификация основной схемы позволяет пользователям получать доступ к файлам других пользователей. Для этого им нужно указать идентификатор владельца файла. Например, команда

open ("x")

может быть вызовом для открытия файла *x* в каталоге пользователя, а команда

open ("nancy/x")

может быть вызовом для открытия файла *x* в каталоге другого пользователя, Нэнси.

Одна из ситуаций, в которых пользователям может понадобиться получить доступ к файлам, не находящимся в их каталогах, – это выполнение системных двоичных программ. Копирование всех системных программ во все пользовательские каталоги крайне неэффективно. Таким образом, возникает необходимость в создании, по крайней мере, одного системного каталога, содержащего все исполнимые двоичные системные файлы

Благодаря двухуровневой иерархии исчезают конфликты имен файлов между различными пользователями, но ее недостаточно для пользователей с большим числом файлов. Обычно пользователям бывает необходимо логически группировать свои файлы. Требуется некий гибкий способ, позволяющий объединить эти файлы в группы.

Следовательно, нужна некая общая иерархия (то есть дерево каталогов). При таком подходе каждый пользователь может сам создать себе столько каталогов, сколько ему нужно, группируя свои файлы естественным образом. Этот подход проиллюстрирован на рисунке 7.6. Здесь каталоги A, B и C, содержащиеся в корневом каталоге, принадлежат различным пользователям, два из которых создали подкаталоги для проектов, над которыми они работают.

Возможность создавать произвольное количество подкаталогов является мощным структурирующим инструментом, позволяющим пользователям организовать свою работу. По этой причине почти все современные файловые системы организованы подобным образом.

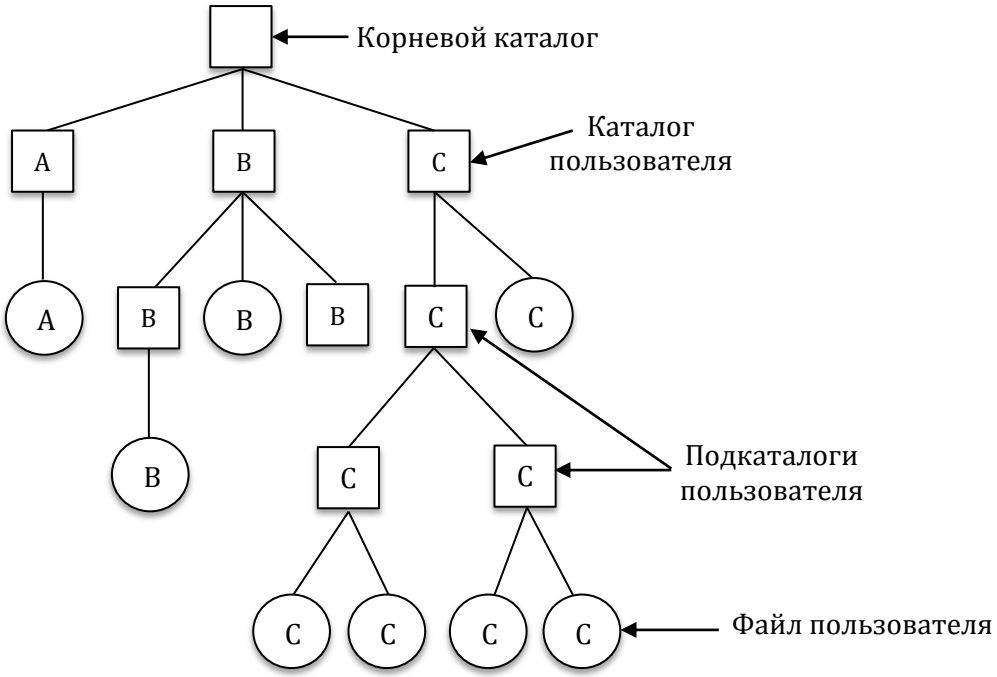


Рисунок 7.6 – Иерархическая каталоговая система

При организации файловой системы в виде дерева каталогов требуется некий способ указания файла. Для этого обычно используют два различных метода. В первом случае каждому файлу дается **абсолютное имя пути**, состоящее из имен всех каталогов от корневого до того, в котором содержится файл, и имени самого файла. Такие пути всегда начинаются от корневого каталога и являются уникальными. В UNIX компоненты пути разделяются прямой чертой (/), а в Windows – обратной (\). А в ОС MULTICS использовался символ (>). Если первый символ имени пути – разделитель, это означает, независимо от используемого в качестве разделителя символа, что путь абсолютный.

Второй вариант – это использование **относительного имени пути**. Оно используется вместе с концепцией **рабочего каталога** (**текущего каталога**). Пользователь может назначить один из каталогов текущим рабочим каталогом. В этом случае все имена путей, не начинающиеся с символа разделителя, считаются относительными и отсчитываются относительно текущего каталога. Этот вариант позволяет обращаться к подкаталогам, расположенным в текущем каталоге, используя более короткую запись.

Некоторым программам бывает нужно получить доступ к файлам независимо от того, какой каталог является в данный момент текущим. В этом случае они всегда должны использовать абсолютные имена. Например, программе проверки правописания может понадобиться для выполнения работы прочитать файл

/usr/lib/dictionary. В этом случае она должна использовать полное, абсолютное имя файла, так как она не знает, каким будет рабочий каталог при ее вызове. Абсолютное имя файла будет работать всегда, независимо от того, какой каталог является текущим в данный момент.

Если программе проверки правописания понадобится большое количество файлов из каталога /usr/lib, она может, обратившись к операционной системе, поменять рабочий каталог на /usr/lib, после чего использовать просто имя dictionary для первого параметра системного вызова *open*. Явно указав свой рабочий каталог, программа может использовать в дальнейшем относительные имена, так как точно знает, где она находится в дереве каталогов.

У каждого процесса есть свой рабочий каталог, поэтому, когда процесс меняет свой рабочий каталог и потом завершает работу, это не влияет на работу других процессов, и в файловой системе не остается никаких следов от подобных изменений рабочих каталогов. Таким образом, процесс может спокойно менять свой рабочий каталог, когда это ему удобно. С другой стороны, если библиотечная процедура поменяет свой рабочий каталог и не восстановит его при возврате управления, программа, вызвавшая ее, может оказаться не в состоянии продолжать свою работу, так как ее предположения о текущем каталоге окажутся неверными. По этой причине библиотечные процедуры редко меняют свои рабочие каталоги, а когда все-таки меняют, то обязательно восстанавливают рабочий каталог перед возвратом.

Большинство операционных систем, поддерживающих иерархические каталоги, имеют специальные элементы в каждом каталоге. Это (.) и (..) для обозначения текущего и родительского каталога.

Системные вызовы, управляющие каталогами, значительно менее схожи в различных системах, чем системные вызовы для работы с файлами. Чтобы дать представление о том, что они собой представляют и как работают, приведем следующий пример (взятый из UNIX).

1. **Create**. Создать каталог. Только что созданный каталог пуст и не содержит других элементов, кроме (.) и (..), автоматически помещаемых в каталог операционной системой.

2. **Delete**. Удалить каталог. Может быть удален только пустой каталог. Элементы (.) и (..) файлами не являются и удалены быть не могут.

3. **Opendir**. Открыть каталог. После этой операции каталог может быть прочитан. Например, для распечатки всех файлов, содержащихся в каталоге, программа, создающая листинг, открывает каталог, чтобы прочитать имена всех содержащихся в нем файлов. Прежде, чем каталог может быть прочитан, его следует открыть, подобно открытию и чтению файла.

4. **Closedir**. Закрыть каталог. Когда каталог прочитан, его следует закрыть, чтобы освободить место во внутренней таблице.

5. **Readdir**. Прочитать следующий элемент открытого каталога. В прежние времена было возможно читать каталоги с помощью обычного системного вызова *read*, но такой подход был небезопасен, так как требовал от программиста умения работать с внутренней структурой каталогов. Поэтому был создан отдельный системный вызов *readdir*, всегда возвращающий одну запись каталога стандартного формата независимо от используемой структуры каталогов.

6. **Rename**. Переименование каталога. Во многих отношениях каталоги аналогичны файлам и могут переименовываться так же, как и файлы.

7. **Link**. Связывание представляет собой технику, позволяющую файлу появляться сразу в нескольких каталогах. Этот системный вызов принимает в качестве входных параметров имя файла и имя пути и создает связь между ними. Таким образом, один и тот же файл может появляться сразу в нескольких каталогах. Подобная связь, увеличивающая на единицу счетчик i-узла файла (для учета количества каталогов со ссылками на этот файл), иногда называется **жесткой связью**.

8. **Unlink**. Удаление ссылки на файл из каталога. Если файл присутствует только в одном каталоге, то данный системный вызов удалит его из файловой системы. Если существует несколько ссылок на этот файл, то будет удалена только указанная ссылка, а остальные останутся. Этот системный вызов применяется для удаления файла в операционной системе UNIX.

Приведенный выше список содержит наиболее важные системные вызовы, но существует также множество других, например, для управления защитой информации.

7.3. Реализация файловой системы

Рассмотрим структуру файловой системы с точки зрения разработчика. Файловые системы хранятся на дисках. Большинство дисков делятся на несколько

разделов с независимой файловой системой на каждом разделе. Сектор 0 диска называется **главной загрузочной записью** (MBR, Master Boot Record) и используется для загрузки компьютера. В конце главной загрузочной записи содержится таблица разделов. В ней хранятся начальные и конечные адреса (номера блоков) каждого раздела. Один из разделов помечен 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, называемый **загрузочным**, и исполняет его. Программа, находящаяся в загрузочном блоке, загружает ОС, содержащуюся в этом разделе. Загрузочные блоки есть во всех разделах, даже в тех, где нет загружаемой ОС. Это сделано для того, чтобы в случае необходимости любой раздел мог быть сделан активным.

Наиболее важным моментом в реализации хранения файлов является учет соответствия блоков диска файлам. Для определения того, какой блок какому файлу принадлежит, в различных ОС применяются различные методы.

Простейшей схемой выделения файлам определенных блоков на диске является система, в которой файлы представляют собой непрерывные наборы соседних блоков диска. Тогда на диске, состоящем из блоков 1 Кбайт, файл размером 50 Кбайт будет занимать 50 последовательных блоков.

У такой системы есть два важных преимущества. Во-первых, ее легко реализовать. А во-вторых, производительность у нее очень высокая.

Однако у этой системы есть и серьезный недостаток – со временем диск становится фрагментированным. Поэтому чаще всего такая схема используется на CD-ROM.

Второй метод размещения файлов состоит в представлении каждого файла в виде связанного списка из блоков диска. Первое слово каждого блока используется как указатель на следующий блок. В остальной части блока хранятся данные. Такой метод позволяет использовать каждый блок диска. Нет потерь дискового пространства на фрагментацию. Кроме того, в каталоге нужно хранить только адрес первого блока файла.

С другой стороны, произвольный доступ к такому файлу будет довольно медленным. Чтобы получить доступ к блоку n, ОС должна сначала прочитать первые n – 1 блоков по очереди.

Кроме того, размер блока уменьшается на несколько байтов, требуемых для хранения указателя, что приводит к тому, что размер блока не будет являться степенью двух, что может потребовать объединять два соседних блока диска.

Оба недостатка такой схемы организации файлов могут быть устранены, если указатели на следующие блоки хранить не прямо в блоках, а в отдельной таблице. Цепочка блоков файла завершается специальным маркером (например, –1), не являющимся допустимым номером блока. Такая таблица, загружаемая в оперативную память, называется **FAT-таблицей** (File Allocation Table – таблица размещения файлов).

Эта схема позволяет использовать для данных весь блок. Кроме того, случайный доступ при этом становится намного проще. Основной недостаток этого метода в том, что вся таблица должна постоянно находиться в памяти.

Последний метод отслеживания принадлежности блока диска файлам состоит в связывании с каждым файлом структуры данных, называемой **i-узлом** (index node – индекс-узел), содержащей атрибуты файла и адреса блоков файла. Большое преимущество такой схемы заключается в том, что каждый конкретный i-узел должен находиться в памяти только тогда, когда соответствующий ему файл открыт. Чтобы избежать ситуации, когда не хватит количества адресов блоков, выделенных файлу, последний адрес блока указывает не на блок, принадлежащий файлу, а на блок диска, в котором хранится следующий i-узел, содержащий дополнительные дисковые адреса.

Часто разным пользователям бывает нужно совместно использовать одни и те же файлы. Чтобы это реализовать, совместно используемый файл одновременно должен присутствовать в каталогах, принадлежащих разным пользователям. Такое соединение между каталогом одного пользователя и совместно используемым файлом в каталоге другого называется **связью**.

Несколько слов об организации дискового пространства. Почти все файловые системы хранят файлы блоками фиксированного размера. Первая проблема связана с выбором размера блока. С одной стороны, если выбрать блоки большого размера, то в случае использования маленьких файлов будет оставаться большой незанятый объем в каждом блоке. И наоборот, если выбрать маленькие блоки, то при работе с большими файлами будет много мелких блоков, что приведет к снижению производительности. В связи с этим очень сложно выбрать золотую середину между

скоростью чтения/записи данных и эффективностью использования дискового пространства.

В операционных системах на базе UNIX выбраны блоки 1 Кбайт, а в Windows блок может быть любой степенью числа два в зависимости от размера диска.

После того, как выбран размер блоков, следует определить, как учитывать свободные и занятые блоки. Это можно осуществлять двумя способами. Первый вариант подразумевает использование списков блоков. В таком списке один блок содержит 32-битную запись номера одного свободного блока. Другой метод состоит в хранении этой информации в виде битового массива (бит-карты). При этом на каждый блок приходится по одному биту вместо 32. Свободные блоки обозначаются в массиве единицами, а занятые – нулями.

Немного остановимся на *дисковых квотах*. Чтобы не допустить захвата пользователями слишком больших участков дискового пространства, многопользовательские ОС часто содержат механизм предоставления дисковых квот. Суть в том, что администратор назначает каждому пользователю максимальную долю файлов и блоков, а ОС гарантирует, что пользователи не превышают выделенных им квот.

Когда пользователь открывает файл, ОС находит его атрибуты и дисковые адреса и помещает их в таблицу в оперативной памяти. Среди атрибутов находится элемент, сообщающий, кто является владельцем данного файла. Любые увеличения размеров файла будут учитываться в записи квот для этого пользователя.

Вторая таблица содержит запись квоты для каждого пользователя, файл которого открыт в данный момент, даже если этот файл был открыт кем-либо другим.

Когда в таблице открытых файлов создается новый элемент, в него помещается указатель на запись квоты владельца файла. При каждом добавлении нового блока к файлу общее количество блоков, числящееся за пользователем, увеличивается и сравнивается с гибким и жестким лимитом. Гибкий лимит может быть превышен, но жесткий – нет. При достижении жесткого лимита любая попытка добавить блок к файлу будет завершаться ошибкой. Аналогично проверяется количество файлов пользователя.

Когда пользователь регистрируется в системе, ОС считывает его файл квот, проверяя, не превысил ли пользователь гибкие пределы по числу блоков или числу файлов. Если один из пределов превышен, ОС выдает предупреждение, а счетчик

оставшихся предупреждений уменьшается на единицу. Когда он достигает нуля, ОС отказывает пользователю в регистрации.

Таким образом, пользователь может превысить гибкие лимиты при условии, что перед выходом из системы он удалит лишние файлы, превышающие эти пределы.

Переходя к вопросу надежности файловой системы, нельзя не сказать об *архивации данных*. Для создания резервной копии диска существует две стратегии: физическая архивация и логическая. **Физическая архивация** состоит в поблочном копировании всего диска с блока 0 по последний блок.

Логическая архивация сканирует один или несколько указанных каталогов со всеми их подкаталогами и копирует все содержащиеся в них файлы и каталоги полностью либо только изменившиеся с указанной даты.

Вторым аспектом, относящимся к проблеме надежности, является **непротиворечивость** файловой системы. Файловые системы обычно читают блоки данных, модифицируют их и записывают обратно. Если в системе произойдет сбой прежде, чем все модифицированные блоки будут записаны на диск, файловая система может оказаться в противоречивом состоянии.

Существует специальная обслуживающая программа, занимающаяся проверкой непротиворечивости. При этом есть два типа такой проверки: блоков и файлов. При проверке непротиворечивости блоков программа создает две таблицы, каждая из которых содержит счетчик для каждого блока, изначально установленный на 0. Счетчики в первой таблице учитывают, сколько раз каждый блок присутствует в файле. Счетчики во второй таблице записывают, сколько раз каждый блок учитывается в списке свободных блоков.

Проверка файлов на непротиворечивость осуществляется аналогично, только счетчики подсчитывают количество файлов.

Поговорим теперь о *производительности* файловой системы. Доступ к диску производится значительно медленнее, чем к оперативной памяти. Если требуется прочитать или записать всего одно слово, то память оказывается примерно в миллион раз быстрее жесткого диска. Поэтому во многих файловых системах применяются различные методы оптимизации, увеличивающие производительность.

Одним из таких методов является **кэширование**. Для минимизации количества обращений к диску применяется **блочный кэш** или **буферный кэш**.

Кэшем называется набор блоков, логически принадлежащих диску, но хранящихся в оперативной памяти по соображениям производительности.

Существуют различные алгоритмы управления кэшем. Обычная практика заключается в перехвате всех запросов чтения к диску и проверке наличия требующихся блоков в кэше. Если блок присутствует в кэше, то запрос чтения блока может быть удовлетворен без обращения к диску. В противном случае блок сначала считывается с диска в кэш, а оттуда копируется по нужному адресу памяти. Последующие обращения к тому же блоку могут удовлетворяться из кэша.

Второй метод увеличения производительности файловой системы называется **опережающее чтение диска**. Он состоит в попытке получить блоки диска в кэш прежде, чем они потребуются. Многие файлы считываются последовательно. Когда файловая система получает запрос на чтение блока *k* файла, она выполняет его, но после этого сразу проверяет, есть ли в кэше блок *k*+1. Если этого блока в кэше нет, файловая система читает его в надежде, что к тому моменту, когда он понадобится, этот блок уже будет считан в кэш. В крайнем случае, он уже будет на пути туда.

И, наконец, третий способ увеличения производительности файловой системы – это **снижение времени перемещения блока головок**. Достигается это помещением блоков, к которым велика вероятность доступа в течение короткого интервала времени, близко друг к другу, желательно на одном цилиндре. Когда записывается выходной файл, файловая система должна зарезервировать место для чтения таких блоков за одну операцию. Это сделать значительно проще, если свободные блоки учитываются в битовом массиве, а весь битовый массив помещается в оперативной памяти.

7.4. Примеры файловых систем

Рассмотрим основные варианты файловых систем. В качестве первого примера рассмотрим файловую систему, применяемую на CD-ROM (**CDFS**). Эти системы самые простые, так как запись на CD-ROM может осуществляться только один раз и на них не бывает свободных блоков.

Файловая система для CD-ROM описывается Международным стандартом ISO 9660, хотя имеются и дополнительные расширения этого стандарта.

Данные на CD-ROM организованы в виде спирали. Биты вдоль спирали разделены на логические блоки по 2352 байта. Часть этих байтов расходуется на

служебную информацию, в результате полезная нагрузка в каждом блоке составляет 2048 байт. У музыкальных дисков позиция блока может указываться в минутах и секундах. При этом каждая секунда содержит 75 блоков.

Каждый CD-ROM начинается с 16 блоков, чья функция не определяется стандартом. Следом располагается один блок, содержащий **основной описатель тома**, в котором хранится некоторая общая информация о CD-ROM – идентификаторы системы, тома, издателя, лица, подготовившего данные, размер логического блока (2048, 4096, 8192 и т.д.), описатель корневого каталога и др. Заполнять этот блок можно только символами верхнего регистра, цифрами и ограниченным количеством знаков препинания.

Все записи каталога, кроме первых двух, располагаются в алфавитном порядке. Первая запись представляет собой описатель самого каталога. Вторая запись является ссылкой на родительский каталог. Эти записи аналогичны каталоговым записям (.) и (..) в UNIX.

Количество каталоговых записей не ограничено. Однако глубина вложенности каталогов не может быть более восьми. В дальнейшем был предложен ряд расширений для стандарта ISO 9660. По этим расширениям длина имен файлов увеличилась до 64 символов, стало возможным использовать набор символов Unicode, глубина вложенности каталогов стала неограниченной и появилась возможность добавлять в имена каталогов расширения.

Перейдем теперь к файловым системам собственно персональных компьютеров. Одной из первых файловых систем была **CP/M** (Control Program for Microcomputers). Она была предшественницей MS-DOS и доминировала на рынке в 80-х годах XX века.

В файловой системе CP/M всего один каталог, содержащий записи фиксированного размера (32 байт). Размер каталога, фиксированный для данной реализации, может отличаться в других реализациях системы CP/M. В этом каталоге перечисляются все файлы системы. После загрузки система считывает каталог и рассчитывает битовый массив занятых и свободных блоков. Этот битовый массив, размер которого для 180-килобайтного диска составляет всего 23 байта, постоянно хранится в оперативной памяти. После завершения работы операционной системы он не сохраняется на диске. Благодаря такому подходу исчезает необходимость в проверке непротиворечивости файловой системы на диске и сохраняется на диске один блок (в процентном отношении это эквивалентно сохранению 90 Мбайт на 16-

гигабайтном диске). При этом в CP/M поддерживалась поочередная работа разных пользователей, то есть каждый пользователь видел только свои файлы.

Преемником файловой системы CP/M стала **MS-DOS**. Файловая система MS-DOS во многом напоминает файловую систему CP/M. В первой версии MS-DOS также был всего один каталог. Однако с версии MS-DOS 2.0 появилась иерархическая система каталогов. В системе MS-DOS пользователи могли видеть все файлы всех пользователей. Имена файлов в файловой системе MS-DOS могли быть только в формате 8.3, то есть 8 символов имени + 3 символа расширения.

В зависимости от количества блоков на диске в системе MS-DOS применяется три версии файловой системы FAT: **FAT-12, FAT-16** и **FAT-32**. В действительности FAT-32 является неверным названием, так как используются только 28 младших битов дискового адреса. Ее следовало бы назвать FAT-28, но степени двух звучат гораздо приятнее.

Во всех файловых системах FAT размер блока диска в байтах может быть установлен равным некоторому числу, кратному 512 (возможно, различному для каждого раздела диска), с наборами разрешенных размеров блоков (называемых корпорацией Microsoft **размерами кластеров**), различными для каждого варианта FAT. В первой версии системы MS-DOS использовалась FAT-12 с 512-байтовыми блоками, что позволяло создавать дисковые разделы размером до 2^{12} x 512 байт (на самом деле только 4086 x 512 байт, так как 10 дисковых адресов использовались как специальные маркеры – конец файла, дефектный блок и т. д.). При этом максимальный размер дискового раздела мог составлять 2 Мбайт, а в оперативной памяти FAT-таблица занимала 4096 элементов по два байта каждый. Кроме того, обработка 12-разрядных адресов была довольно медленной.

Такая система неплохо работала на гибких дисках, но с появлением жестких дисков появились проблемы. Корпорация Microsoft попыталась решить проблему, разрешив использовать дисковые блоки (кластеры) размером 1, 2 и 4 Кбайт. Это позволило сохранить структуру и размер таблицы FAT-12 и увеличить размер дискового раздела до 16 Мбайт.

Так как система MS-DOS поддерживала до четырех дисковых разделов на диске, новая файловая система FAT-12 могла работать с дисками емкостью до 64 Мбайт. Для поддержки винчестеров большего размера нужно было предпринимать что-то еще. В результате была разработана файловая система FAT-16, с 16-разрядными дисковыми указателями. Дополнительно было разрешено

использовать кластеры размеров 8, 16 и 32 Кбайт. (32 768 – это максимальное число, представляющее собой степень двух, которое может быть представлено 16 двоичными разрядами.) Теперь таблица FAT-l6 постоянно занимала 128 Кбайт оперативной памяти, но с ростом размеров памяти компьютеров она получила широкое применение и быстро вытеснила файловую систему FAT-12. Максимальный размер дискового раздела, поддерживаемый системой FAT-16, равен 2 Гбайт (64 К элементов по 32 Кбайт каждый), а максимальный размер диска – 8 Гбайт, то есть четыре раздела по 2 Гбайт каждый.

Для хранения деловой переписки такое ограничение проблем не вызывает, однако при работе с цифровым видео в стандарте DV один 2-гигабайтный файл вмещает всего лишь немногим более 9 мин видеофильма. Таким образом, на весь диск из четырех разделов может поместиться около 38 мин видео, независимо от размеров самого диска. Это ограничение также означает, что в режиме on-line редактировать можно не более 19 мин фильма, так как на диске требуется одновременно хранить и входные, и выходные файлы.

С выходом второй версии операционной системы Windows 95 была представлена файловая система FAT-32 со своими 28-разрядными адресами. При этом версия системы MS-DOS, лежащая в основе Windows 95, была адаптирована для поддержки FAT-32. Теоретически в этой системе разделы могли быть по 2^{28}x2^{15} байт, но фактически размер разделов ограничен 2 Тбайт (2048 Гбайт), так как внутренне система учитывает размеры разделов в 512-байтовых секторах с помощью 32-разрядных чисел, а 2^{32} x 2^{9} байт равно 2 Тбайт.

Помимо поддержки дисков большего размера, файловая система FAT-32 обладает двумя другими преимуществами перед системой FAT-16. Во-первых, 8-гигабайтный диск, использующий FAT-32, может состоять из всего одного раздела. При использовании FAT-16 он должен был содержать четыре раздела, что представлялось пользователям системы Windows как логические устройства C:, D:, E: и F:. Какой файл на каком устройстве располагать, решать пользователю.

Другое преимущество FAT-32 перед FAT-16 заключается в том, что для дискового раздела заданного размера могут использоваться блоки меньшего размера. Например, для 2-гигабайтного дискового раздела система FAT-16 должна пользоваться 32-килобайтными блоками, в противном случае при наличии всего 64 К доступных дисковых адресов она не смогла бы покрыть весь раздел. В то же время система FAT-32 для такого же дискового раздела может использовать,

например, блоки размером 4 Кбайт. Преимущество блоков меньшего размера заключается в том, что длина большинства файлов была менее 32 Кбайт. При размере блока в 32 Кбайт даже 10-байтовый файл будет занимать на диске 32 Кбайт. Если средний размер файлов, скажем, равен 8 Кбайт, тогда при использовании 32-килобайтных блоков около 3/4 дискового пространства будет теряться, то есть эффективность использования диска будет низкой. При 8-килобайтных файлах и 4-килобайтных блоках потерь дискового пространства не будет, но платой за это будет то, что для хранения таблицы FAT потребуется значительно больше оперативной памяти. При 4-килобайтных блоках 2-гигабайтный раздел будет состоять из 512 К блоков, поэтому таблица FAT должна состоять из 512 К элементов (занимая 2 Мбайт ОЗУ).

Файловая система MS-DOS использует FAT для учета свободных блоков. Любой незанятый блок помечается специальным кодом. Когда системе MS-DOS требуется новый блок на диске, она ищет этот код в таблице FAT. Таким образом, битовый массив или список свободных блоков не нужны.

Операционные системы **Windows 95, 98, Me** функционировали на основе тех же самых файловых систем FAT. В Windows 95 была файловая система FAT-12 (в первой версии) и FAT-16, а в Windows 98 и Windows Me – FAT-32. Во второй версии Windows 95 были разрешены длинные имена, причем была разработана система обратной совместимости от длинных имен к формату 8.3, чтобы названия файлов, созданных в более поздних версиях ОС могли быть прочитаны в более ранних. При этом берутся первые шесть символов, преобразуются в верхний регистр, отсекаются все пробелы и лишние точки, а в конце добавляется суффикс ~1. Если такое имя уже есть, то – ~2 и т.д.

Более современная операционная система **Windows NT** и ее преемники **Windows 2000/XP/Vista/7/8** и серверные ОС Windows Server 2003/2008 работают на файловой системе **NTFS** (New Technology File System). В ней используются 64-разрядные дисковые адреса.

Файл в системе NTFS – это не просто линейная последовательность байтов как в системах FAT-32 и UNIX. Вместо этого файл состоит из множества атрибутов, каждый из которых представляется в виде потока байтов. Большинство файлов имеет несколько коротких потоков, таких как имя файла и его 64-битовый идентификатор, плюс один длинный (неименованный) поток с данными. Однако

длинных потоков с данными может быть несколько. У каждого потока своя длина. Каждый поток может блокироваться независимо от остальных потоков.

Теперь в общих чертах рассмотрим файловую систему **UNIX**. Даже в ранних версиях системы UNIX применялась довольно сложная многопользовательская файловая система, так как в основе этой системы лежала операционная система MULTICS. Мы рассмотрим файловую систему **V7**, разработанную для компьютера PDP-11, сделавшего систему UNIX знаменитой. Современные версии будут рассмотрены позже.

Файловая система представляет собой дерево, начинающееся в корневом каталоге, с добавлением связей, формирующих направленный ациклический граф. Имена файлов могут содержать до 14 символов, включающих в себя любые символы ASCII, кроме косой черты (использовавшейся в качестве разделителя компонентой пути) и символа NUL (использовавшегося для дополнения имен короче 14 символов). Символ NUL обозначается байтом 0.

Каталог UNIX содержит по одной записи для каждого файла этого каталога. Каждая каталоговая запись максимально проста, так как в системе UNIX используется схема i-узлов. Каталоговая запись состоит всего из двух полей: имени файла (14 байт) и номера i-узла для этого файла (2 байт). Эти параметры ограничивают количество файлов в файловой системе числом 64 К.

Файл в системе UNIX – это последовательность байтов произвольной длины, содержащая произвольную информацию. Не делается принципиального различия между текстовыми (ASCII) файлами, двоичными файлами и любыми другими файлами. Значение битов в файле целиком определяется владельцем файла. Системе это безразлично. Изначально размер имен файлов был ограничен 14 символами, но в системе **Berkeley** UNIX этот предел был расширен до 255 символов, что впоследствии было принято в **System V**, а также в большинстве других версий. В именах файлов разрешается использовать все ASCII-символы, кроме символа NUL, поэтому допустимы даже имена файлов, состоящие, например, из трех символов возврата каретки (хотя такое имя и не слишком удобно в использовании).

По соглашению многие программы ожидают, что имена файлов должны состоять из основного имени и расширения, отделяемого от основного имени файла точкой (которая в системе UNIX также считается символом). Так, *prog.c* – это, как правило, программа на языке C, *prog.f90* – обычно программа на языке FORTRAN 90, а *prog.o* – чаще всего объектный файл (выходные данные компилятора). Эти

соглашения никак не регулируются операционной системой, но некоторые компиляторы и другие программы ожидают файлов именно с такими расширениями. Расширения могут иметь произвольную длину, кроме того, файлы могут иметь по несколько расширений, как, например, *prog.java.Z*, что, скорее всего, представляет собой сжатую программу на языке Java.

Для удобства использования файлы могут группироваться в каталоги. Каталоги хранятся на диске в виде файлов, и до определенного предела с ними можно работать как с файлами. Каталоги могут содержать подкаталоги, что приводит к иерархической файловой системе. Корневой каталог называется / и, как правило, содержит несколько подкаталогов. Символ / также используется для разделения имен каталогов, поэтому имя */usr/ast/x* означает файл *x*, расположенный в каталоге *ast*, который, в свою очередь, находится в каталоге *usr*.

В UNIX можно выделить ряд важных каталогов, существующих в большинстве UNIX-систем (таблица 8.1). Такие каталоги всегда имеют указанное имя для совместимости с различным программным обеспечением.

Таблица 8.1 – Наиболее важные каталоги в системе UNIX.

Каталог	Содержание
bin	Двоичные (исполняемые) программы
dev	Специальные файлы для устройств ввода-вывода
etc	Разные системные файлы
lib	Библиотеки
usr	Каталоги пользователей

8. МУЛЬТИМЕДИЙНЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ

8.1. Введение в мультимедиа

Для начала необходимо определиться с терминами. **Мультимедиа** – в общеупотребительном смысле это документ, содержащий средства информации, протяженные во времени, то есть проигрываемые в течение определенного интервала времени. Другой неоднозначный термин – это **видео**, часто под ним понимается продукт, включающий как изображение, так и звук.

Основными характеристики мультимедиа являются:

1. Мультимедиа использует предельно высокие скорости передачи данных.
2. Для мультимедиа требуется воспроизведение в режиме реального времени.

Существуют различные системы передачи видеосигнала – NTSC (Северная и Южная Америка, Япония), PAL (Западная Европа), SECAM (Франция, Восточная Европа, страны бывшего СССР). Технически лучшей системой является PAL. Параметры реального времени, требуемые для приемлемого воспроизведения мультимедиа, часто называют параметрами **качества обслуживания**. К ним относятся:

1. Средняя доступная пропускная способность.
2. Максимальная пропускная способность.
3. Минимальная и максимальная задержка, которая определяет **джиттер** – неравномерность времени доставки.
4. Вероятность потери бита.

8.2. Мультимедийные файлы

В большинстве систем обычный файл состоит из линейной последовательности байтов без какой-либо структуры, о которой бы знала ОС. В мультимедиа ситуация гораздо более сложная. Во-первых, видео- и аудиоданные полностью различны. Они вводятся совершенно разными устройствами, у них различная внутренняя структура (видео передается с частотой 25 – 30 кадров в секунду, тогда как аудио обычно хранится в виде 44100 Гц), и воспроизводятся они также различными устройствами.

Во-вторых, в большинстве фильмов содержится несколько звуковых дорожек (для разных языков) и несколько видов субтитров.

В результате цифровой фильм может оказаться состоящим из большого количества файлов: видео-файла, нескольких аудио-файлов и нескольких текстовых файлов. Таким образом, файловая система должна следить за несколькими "субфайлами". При этом необходим способ синхронизации субфайлов, чтобы любая звуковая дорожка соответствовала изображению.

Рассмотрим, каким образом осуществляется кодирование звука и изображения. Аудиоволны преобразовываются в цифровую форму при помощи аналогово-цифрового преобразователя (АЦП). При этом математически доказано, что если звуковая волна не является чисто синусоидальной, а представляет собой сумму нескольких синусоидальных волн, самая высокая частота составляющих которых равна *f*, тогда для последующего восстановления сигнала достаточно измерять значения сигнала с частотой дискретизации 2*f*. Однако оцифрованные отсчеты никогда не бывают точными. Ошибка, возникающая в результате неточного соответствия квантованного сигнала к исходному, называется **шумом квантования**.

Перейдем теперь к обработке изображения. Сетчатка глаза человека обладает инерционными свойствами, то есть яркое изображение, быстро появившееся на сетчатке, остается на ней несколько миллисекунд, прежде чем угаснуть. Частота, при которой глаз перестает замечать мигание яркости источника света (изображения), составляет около 50 Гц. Все видео системы используют этот принцип для создания двигающихся изображений. Не стоит путать это с тем, что человеческому глазу необходимо 25 кадров в секунду, чтобы просматриваемое изображение воспринималось плавным. Другими словами, плавность движущегося изображения определяется количеством отличающихся изображений в секунду, тогда как мерцание зависит от частоты прорисовки экрана.

Важность этих двух параметров становится ясна, если попробовать оценить пропускную способность, необходимую для передачи цифрового видеосигнала по сети. Если передавать цифровой видеосигнал с разрешение 1024 x 768 точек при 24 битах на пиксел и 25 кадрах в секунду, потребуется поток данных со скоростью 472 Мбит/с. Удвоение же частоты, чтобы избавиться от мерцания, выглядит еще менее привлекательно.

8.3. Сжатие видеоинформации

С учетом вышесказанного, о передаче мультимедийной информации в несжатом виде не может быть и речи. К настоящему времени разработано множество различных методов сжатия. Для всех систем сжатия требуется два алгоритма: кодирования и декодирования. Эти алгоритмы обладают определенной асимметрией. Во-первых, алгоритм кодирования может быть медленным, а алгоритм декодирования должен быть быстрым и должен работать даже на дешевом оборудовании. Однако для систем реального времени, например, видеоконференций, медленное кодирование также неприемлемо.

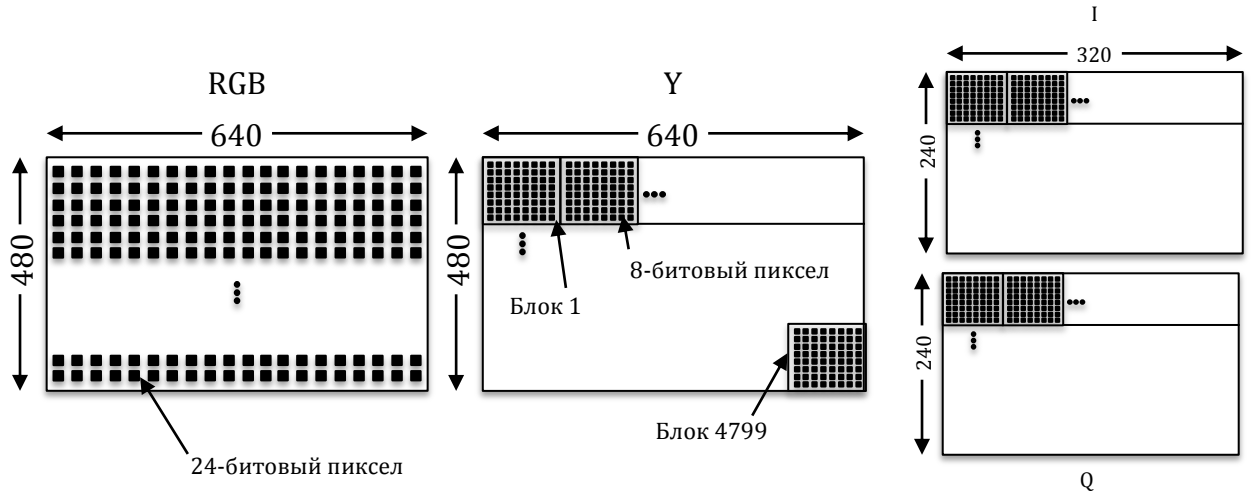
Во-вторых, процесс кодирования/декодирования не должен быть обратимым. То есть при передаче мультимедиа абсолютная точность не требуется. Система, в которой декодированный сигнал не точно соответствует кодированному оригиналу, называется **системой с потерями**. Все системы сжатия данных, применяемые в мультимедиа, являются системами с потерями, что позволяет им достичь гораздо большего коэффициента сжатия. Таким образом, понятия качества и степени сжатия обратно пропорциональны.

Для начала рассмотрим вкратце метод кодирования стандарта **JPEG**. Пусть нам необходимо сжать 24-битовое RGB-изображение размером 640x480 точек. Тогда последовательность действий будет следующая.

1. Из значений RGB вычисляются яркость и два значения цветности (параметры Y, I и Q).

2. Для значений Y, I и Q строятся отдельные матрицы с элементами в диапазоне от 0 до 255 (рисунок 8.1).

3. Путем последовательных преобразований получают 7200 матриц 8x8 коэффициентов дискретного косинусного преобразования (ДКП), причем число 0 находится в центре блока, а элементы матрицы быстро убывают с расстоянием от элемента (0, 0), который представляет собой среднее значение блока.



а *б*

Рисунок 8.1 – Исходные данные в формате RGB (*а*); после подготовки блока (*б*)

4. Следующий этап – квантование, когда каждый элемент матрицы (ДКП-коэффициент) делится на определенные табличные весовые коэффициенты из таблицы квантования, быстро увеличивающиеся по мере удаления от элемента (0, 0) (рисунок 8.2). В результате получают квантовые коэффициенты.

ДКП-коэффициенты

150	80	40	14	4	2	1	0
92	75	36	10	6	1	0	0
52	38	26	8	7	4	0	0
12	8	6	4	2	1	0	0
4	2	1	1	0	0	0	0
2	2	1	1	0	0	0	0
1	1	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Таблица квантования

1	1	2	4	8	16	32	64
1	1	2	4	8	16	32	64
2	2	2	4	8	16	32	64
4	4	4	4	8	16	32	64
8	8	8	8	8	16	32	64
16	16	16	16	16	16	32	64
32	32	32	32	32	32	32	64
64	64	64	64	64	64	64	64

Квантовые коэффициенты

160	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Рисунок 8.2 – Квантование ДКП-коэффициентов

5. Затем 64 элемента каждого блока выстраиваются в ряд таким образом, чтобы нули оказались в конце последовательности (рисунок 8.3).

150	80	20	4	1	0	0	0
92	75	18	3	1	0	0	0
26	19	13	2	1	0	0	0
3	2	2	1	0	0	0	0
1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0

Рисунок 8.3 – Порядок передачи квантованных значений

6. Количество нулей (в данном случае, 38) просто заменяется соответствующим числом (38).

Для декодирования сжатого изображения нужно выполнить все операции в обратном порядке. Стандарт JPEG почти симметричен – декодирование занимает столько же времени, сколько и кодирование. При этом он позволяет осуществлять сжатие фотографий в 20 и более раз.

При осуществлении сжатия видеоинформации можно просто кодировать каждый кадр отдельно алгоритмом JPEG. Дополнительного сжатия можно достичь, используя преимущество того факта, что последовательные кадры часто бывают почти идентичными (временная избыточность). Такой принцип используется в системе DV (Digital Video). Однако, хотя поток данных цифровой видеокамеры формата DV и ниже, чем у несжатого видео, он все же не настолько хорошо сжат, как MPEG-2.

В сценах, в которых камера и задний план неподвижны и один или два актера медленно двигаются, почти все пикселы в соседних кадрах будут идентичны. В этом случае простое вычитание каждого кадра из предыдущего и обработка разности алгоритмом JPEG даст достаточно хороший результат. Однако, в сценах, где камера поворачивается или наезжает на снимаемый объект, необходим какой-то способ компенсировать это движение камеры. В этом и состоит основное отличие MPEG-2 от JPEG.

В результате работы MPEG-2 механизма сжатия образуются кадры следующих типов:

1. I (Intracoded – автономные) – независимые неподвижные изображения, кодированные алгоритмом JPEG. Эти кадры должны появляться примерно 1-2 раза в секунду, чтобы была возможность просмотра фильма не с самого начала, чтобы была возможность декодирования других файлов при наличии ошибок при передаче какого-либо кадра, а также, чтобы упростить индикацию при перемотке вперед или назад.

2. P (Predictive – предсказывающие) – содержащие разностную информацию относительно предыдущего кадра. Они основаны на идее **макроблоков**, покрывающих 16x16 пикселов в пространстве яркости и 8x8 пикселов в пространстве цветности (рисунок 8.4). Так, макроблоки, содержащие только неподвижный задний план, будут полностью совпадать друг с другом, а макроблоки, содержащие движущегося человека, будут смещаться на некоторую величину. Именно для этих макроблоков алгоритм должен найти максимально похожие макроблоки из предыдущего кадра.

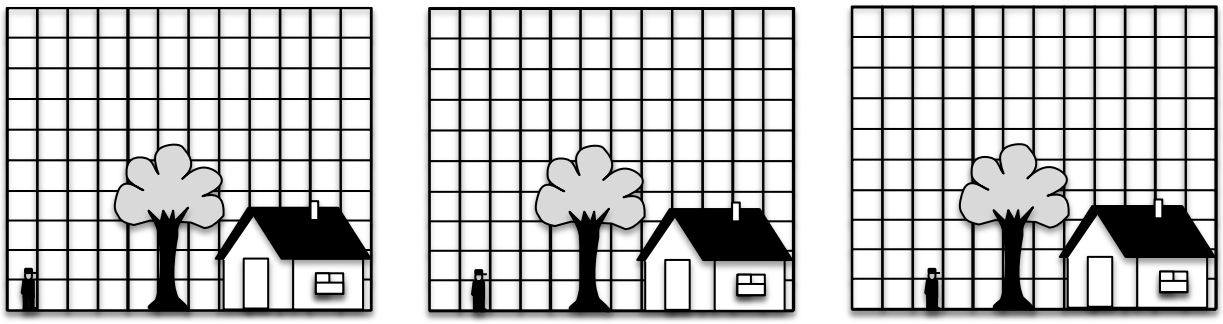


Рисунок 8.4 – Три последовательных кадра

Макроблок, для которого найден похожий на него макроблок в предыдущем кадре, кодируется в виде разности значений яркости и цветности. Затем матрицы разности кодируются аналогично стандарту JPEG. В выходном потоке макроблок представляется в виде вектора сдвига (насколько далеко сдвинулся макроблок по горизонтали и вертикали от положения в предыдущем кадре). Если же в предыдущем кадре не нашлось подходящего макроблока, текущее значение кодируется алгоритмом JPEG как в I-кадре.

3. B (Bidirectional – двунаправленные) – содержащие изменения относительно предыдущего и последующего кадров. Эти кадры подобны P-кадрам с той разницей, что позволяют привязывать макроблок либо к предыдущему кадру, либо к следующему. Это позволяет достичь лучшей компенсации движения. Для упрощения декодирования кадры в потоке MPEG должны присутствовать не в порядке их отображения, а в порядке зависимостей друг от друга. То есть на машине пользователя должна осуществляться буферизация кадров для их правильного отображения.

8.4. Планирование процессов в мультимедийных системах

Одним из отличий ОС, поддерживающих мультимедиа, является планирование процессов. Простейшая разновидность видеосервера может поддерживать отображение фиксированного числа фильмов, использующих одинаковую скорость передачи данных, видеоразрешение, частоту кадров и другие параметры. Для каждого фильма создается отдельный процесс (или поток), чья работа заключается в чтении фильма с диска по кадру и передаче этого кадра пользователю. Для управления этими процессами лучше всего использовать простой **алгоритм поочередного планирования** с добавлением механизма, следящего за временными параметрами.

Однако этот метод планирования редко может быть применен в реальной жизни. Количество пользователей, размеры кадров, разрешение могут меняться во времени. В результате может оказаться, что разным процессам потребуется работа с разной частотой для выполнения различного объема работ и с различными сроками их окончания. Планирование нескольких конкурирующих процессов, у которых есть жесткие сроки выполнения работ, называется **планированием реального времени**.

При этом необходимо учитывать, что необходимо знать, возможно ли в принципе выполнение поставленной задачи, то есть является ли данный набор процессов планируемым вообще. Если у процесса i период равен P_i и на обработку одного кадра этого процесса уходит C_i секунд работы процесса, то система будет планируемой только в том случае, если

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1,$$

где m – количество процессов.

Алгоритмы реального времени могут быть статическими или динамическими. Статические алгоритмы заранее назначают каждому процессу фиксированный приоритет, после чего выполняют приоритетное планирование с переключениями. У динамических алгоритмов нет фиксированных приоритетов.

Классическим примером статического алгоритма планирования реального времени является алгоритм **RMS** (Rate Monotonic Scheduling – планирование с приоритетом, пропорциональным частоте). Этот алгоритм может использоваться для процессов, удовлетворяющих следующим условиям:

1. Каждый периодический процесс должен быть завершен за время его периода.
2. Ни один процесс не должен зависеть от любого другого процесса.
3. Каждому процессу требуется одинаковое процессорное время на каждом интервале.
4. У непериодических процессов нет жестких сроков.
5. Прерывание процесса происходит мгновенно, без накладных расходов.

Последнее требование практически неосуществимо, однако позволяет упростить модель системы. Алгоритм RMS работает, назначая каждому процессу фиксированный приоритет, равный частоте возникновения событий процесса. То есть процесс, который должен запускаться каждые 30 мс (33 раза в секунду), получает приоритет 33; процесс, которые должен запускаться каждые 40 мс (25 раз

в секунду) – 25; процесс, который должен запускаться каждые 50 мс (20 раз в секунду) – 20. Во время работы планировщик всегда запускает готовый к работе процесс с наивысшим приоритетом, прерывая при необходимости работающий процесс (рисунок 8.5). Авторы алгоритма доказали, что RMS является оптимальным решением в классе статических алгоритмов планирования.

Проверив задачу на планируемость получаем, что процесс *A* съедает 10/30 времени центрального процессора, процесс *B* съедает 15/40 времени центрального процессора, а процесс *C* съедает 5/50 времени центрального процессора. Суммарно эти процессы потребляют 0,808 процессорного времени, что меньше единицы, и, следовательно, система является планируемой.

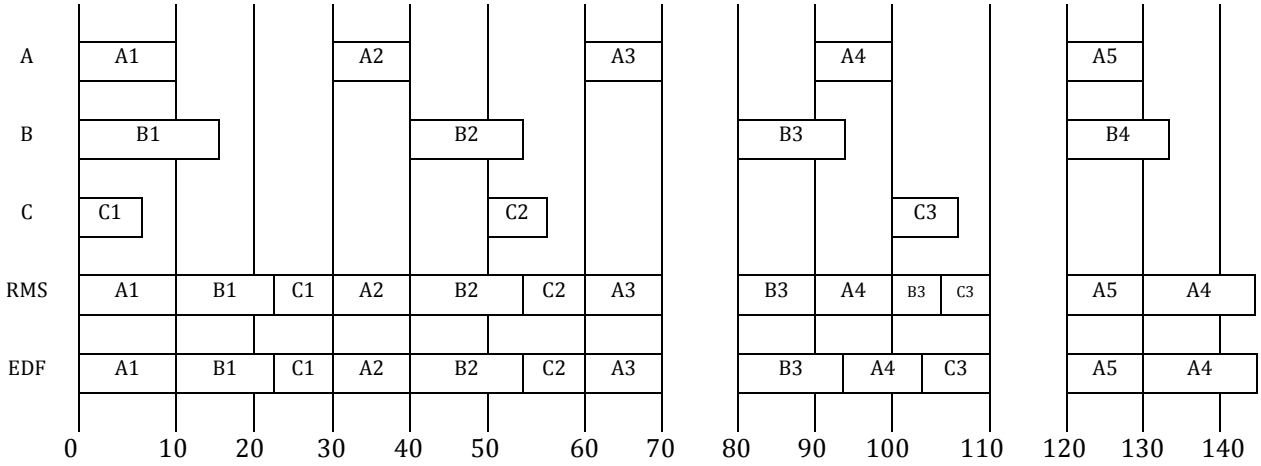


Рисунок 8.5 – Пример алгоритмов планирования реального времени RMS и EDF

Процессам *A, B и C* назначены статические приоритеты 33, 25 и 20 соответственно; это означает, что когда процесс *A* желает работать, он работает, прерывая любой другой процесс, использующий центральный процессор в данный момент. Процесс *B* может прервать работу процесса *C*, но не *A.* Процесс *C* вынужден ждать, пока центральный процессор не освободится.

Изначально все три процесса готовы к работе. Выбирается процесс с максимальным приоритетом то есть процесс *A*. Ему разрешается работать в течение 15 мс, требующихся процессу, чтобы полностью выполнить работу по передаче одного кадра (это показано в строке рисунка, помеченной RMS). Когда процесс *A* оканчивает свою работу, запускается процесс *B,* а затем процесс *C.* Вместе эти процессы потребляют 30 мс процессорного времени, поэтому, когда процесс *C* заканчивает свою работу, пора снова запускать процесс *A*. Этот цикл повторяется до тех пор, пока в момент времени $t = 70$ у системы не начинается период простоя.

В момент времени $t = 80$ процесс *B* переходит в состояние готовности и запускается. Однако в момент времени $t = 90$ процесс *A,* обладающий более высоким

приоритетом, также переходит в состояние готовности, поэтому он прерывает выполнение процесса *B* и работает, пока, не закончит свою работу к моменту времени *t* = 100. В этом месте система должна выбрать между завершением процесса *B* и запуском процесса *C.* Выбирается, естественно, процесс *B* с более высоким приоритетом.

В качестве примера динамического алгоритма планирования можно привести алгоритм **EDF** (Earliest Deadline First – процесс с ближайшим сроком завершения в первую очередь). Этот алгоритм не требует от процессов периодичности и постоянства временных интервалов использования ЦП.

Каждый раз, когда процессу требуется процессорное время, он объявляет о своем присутствии и о своем сроке выполнения задания. Планировщик хранит список процессов, сортированный по срокам выполнения заданий. Алгоритм запускает первый процесс в списке, то есть тот, у которого самый близкий по времени срок выполнения. Когда новый процесс переходит в состояние готовности, система сравнивает его срок выполнения со сроком выполнения текущего процесса. Если у нового процесса график более жесткий, он прерывает работу текущего процесса.

Пример работы алгоритма EDF показан на рисунке 8.5. Вначале все процессы находятся в состоянии готовности. Они запускаются в порядке своих крайних сроков. Процесс *A* должен быть выполнен к моменту времени *t* = 30, процесс *B* должен закончить работу к моменту времени *t* = 40, и процесс *C* должен завершить работу к моменту времени *t* = 50. Таким образом, процесс *A* запускается первым. Вплоть до момента времени *t* = 90 выбор алгоритма EDF не отличается от RMS. В момент времени *t* = 90 процесс *A* снова переходит в состоянии готовности с тем же крайним сроком завершения *t* = 120, что и у процесса *B*. Планировщик имеет право выбрать любой из процессов, но поскольку с прерыванием процесса *B* связаны накладных расходы, лучше предоставить возможность продолжать работу этому процессу.

Алгоритм EDF более надежен и работает при любой загруженности процессора, однако платой за это является большая сложность. В качестве примера ситуации, когда алгоритм EDF справляется с работой, а алгоритм RMS терпит неудачу, представлен рисунок 8.6.

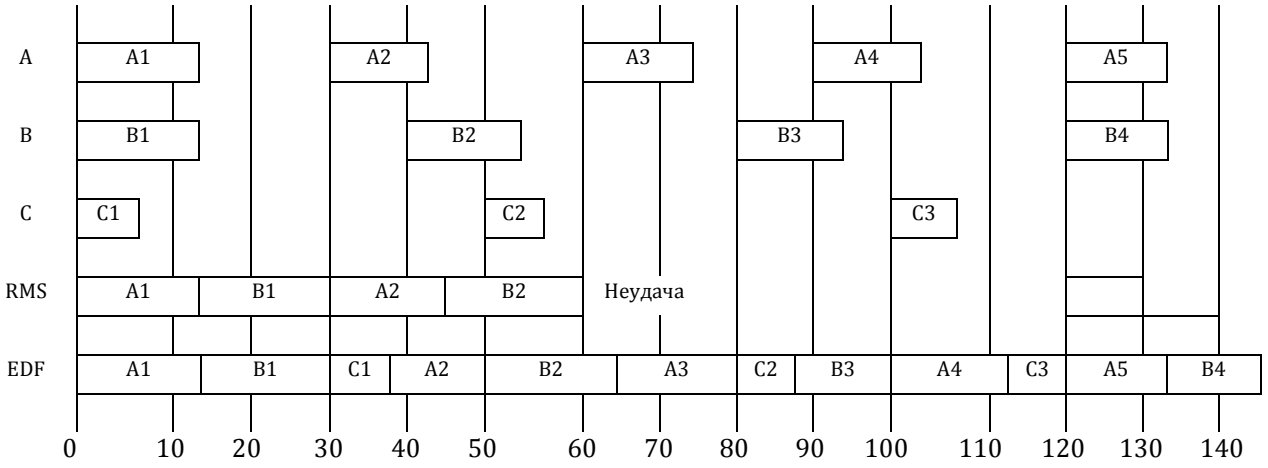


Рисунок 8.6 – Другой пример работы алгоритмов RMS и EDF

В этом примере периоды процессов *A*, *B* и *C* те же, что и прежде, но теперь процессу *A* на передачу одного кадра требуется 15 мс процессорного времени вместо 10 мс. Тест планируемости дает коэффициент использования процессора, равный 0,500 + 0,375 + 0,100 = 0,975. В запасе остается всего лишь 2,5 % времени центрального процессора, но теоретически коэффициент использования процессора не превышает допустимого предела, поэтому для данного случая должен существовать метод планирования.

В алгоритме RMS приоритеты трех процессов по-прежнему равны 33, 25 и 20, так как на них влияют только периоды, а не времена работы. На этот раз процесс *B* завершает работу к моменту времени t = 30, однако в этот момент процесс *A* снова приходит в состояние готовности. К тому моменту, когда он заканчивает свою работу (t = 45), процесс *B* также уже снова готов работать, и поскольку у него приоритет выше, чем у процесса *C,* то запускается процесс *B,* а процесс *C* пропускает свой критический срок. Таким образом, алгоритм RMS терпит фиаско.

Теперь посмотрим, как алгоритм EDF справляется с данным случаем. В момент времени t = 30 между процессами *A2* и *C1* возникает спор. Поскольку срок выполнения процесса *C1* равен 50 мс, а срок выполнения процесса *A2* равен 60 мс, планировщик выбирает процесс *C.* Этим данный алгоритм отличается от RMS, в котором побеждает процесс *A*, как обладающий более высоким приоритетом.

В момент времени t = 90 процесс *A* приходит в состояние готовности в четвертый раз. Предельный срок процесса *A* такой же, что и текущего процесса (120 мс), поэтому у планировщика появляется выбор: прервать работу процесса *B* или нет. Поскольку необходимости прерывать процесс *B* нет, то процессу *B3* разрешается продолжать работу.

В данном примере центральный процессор занят на все 100 % до момента времени t = 150. Однако в конце концов перерыв в его работе наступает, так как центральный процессор занят всего на 97,5 %. Поскольку время и начала и окончания работ во всех случаях кратно 5 мс, промежуток простоя центрального процессора будет составлять 5 мс. Чтобы относительное время простоя было равно 2,5%, 5-миллисекундный интервал бездействия должен происходить каждые 200 мс, что не поместилось на рисунок 8.6.

8.5. Размещение файлов

Размер мультимедийных файлов очень велик. Они, как правило, записываются один раз, но много раз считываются. Доступ к таким файлам чаще всего последовательный. Их воспроизведение должно удовлетворять строгим критериям качества обслуживания. Все эти свойства предполагают использование компоновки файловой системы, отличной от используемой в традиционных системах. Рассмотрим, каким образом этот вопрос решается в случае однодискового варианта хранения файла.

Самое главное требование заключается в том, чтобы данные могли передаваться в сеть или выходное устройство с требуемой скоростью и без джиттера. По этой причине выполнение нескольких операций поиска цилиндра во время считывания кадра крайне нежелательно. Одним из способов устранения излишних перемещений блока головок на сервере является использование непрерывных файлов. Однако возникает проблема при переключении с видео-файла на аудио-файл, а с него на текстовый файл. В результате был разработан другой вариант хранения, с чередованием видео, аудио и текстовой информации в одном файле. При этом сам файл остается непрерывным.

Такая стратегия хороша, если не требуется произвольный доступ к файлу, а также когда используется однопользовательский доступ к видеосерверу. Вследствие этого было создано две альтернативные стратегии организации мультимедийных файлов. Согласно первой используются дисковые блоки маленького размера (существенно меньше размеров кадра), согласно второй – большого. В первом случае используется индекс кадров с указанием на начало и размер каждого кадра. Во втором – индекс блоков с указанием номера кадра, с которого начинается блок. В случае использования маленьких блоков меньше памяти теряется на не полностью заполненных блоках, однако гораздо больший объем ОЗУ идет на хранение самого

индекса кадров. Чтобы получить компромиссное решение, используют блоки больших размеров, но в не полностью заполненные блоки помещают часть следующих кадров. Это может потребовать перемещения блока головок, снижая производительность, однако позволит сэкономить некоторую часть дискового пространства, так как внутренняя фрагментация будет устранена.

Очень часто для повышения надежности и производительности на видеосерверах используют RAID массивы. В таком случае возникает необходимость хранения одного файла на нескольких дисках. Для этого осуществляют покадровое или поблочное чередование. При этом отдельные кадры или блоки фиксированного размера, включающие несколько кадров, записываются последовательно (или в случайном порядке) на каждый диск.

9. МНОГОПРОЦЕССОРНЫЕ СИСТЕМЫ

9.1. Мультипроцессоры

Для компьютерной техники характерно постоянное стремление к росту вычислительной мощности. Сколько бы ни было доступно мощности на данный момент, ее всегда оказывается недостаточно. В прошлом решение всегда состояло в том, чтобы увеличить тактовую частоту процессора. К сожалению, сегодня мы приближаемся к некоторым фундаментальным пределам тактовой частоты. В соответствии со специальной теорией относительности Эйнштейна, никакой сигнал (в том числе электрический) не может распространяться быстрее скорости света, равной 30 см/нс в вакууме и около 20 см/нс в медном проводе или оптоволоконном кабеле. Это означает, что в компьютере с тактовой частотой 10 ГГц сигналы за один такт не могут распространяться дальше, чем на 2 см. Для 100-гигагерцового компьютера полная длина пути сигнала будет составлять максимум 2 мм. Компьютер с тактовой частотой 1 ТГц (1000 ГГц) должен иметь размеры менее 100 мкм, чтобы сигнал от одного его конца до другого успел пройти за один такт процессора.

Производство компьютеров такого размера может теоретически и возможно, но при этом мы сталкиваемся с другой фундаментальной проблемой: рассеянием тепла. Чем быстрее работает компьютер, тем больше тепловой энергии он производит, а чем меньше компьютер, тем труднее отводить эту тепловую энергию. Сейчас вентилятор для охлаждения центрального процессора больше, чем сам центральный процессор. Для перехода с частоты 1 МГц на частоту 1 ГГц требовалось всего лишь постепенное усовершенствование технологии производства микросхем. Для перехода на частоту 1 ТГц потребуются более радикальные изменения.

Другой способ увеличения вычислительной мощности системы состоит в использовании параллельных вычислений. Для этого могут использоваться многопроцессорные компьютеры, состоящие из большого количества процессоров, каждый из которых работает с "нормальной" частотой (чтобы это ни означало в данном году). Однако совместно эти процессоры обладают значительно большей мощностью, чем отдельный центральный процессор. В настоящее время серийно выпускаются и продаются системы с 1000 процессорами. В ближайшее время, вероятно, будет построена система с 1 млн. процессоров.

Высокопараллельные компьютеры часто применяются для сложных вычислений. Такие задачи, как прогноз погоды, моделирование воздушного потока вокруг крыла самолета, мировой экономики или взаимодействия лекарства с рецептором в мозге, требуют больших компьютерных мощностей. Для решения этих задач требуются долгие расчеты на большом количестве центральных процессоров одновременно. Многопроцессорные системы широко применяются для данных и подобных задач в науке и машиностроении, а также в других областях.

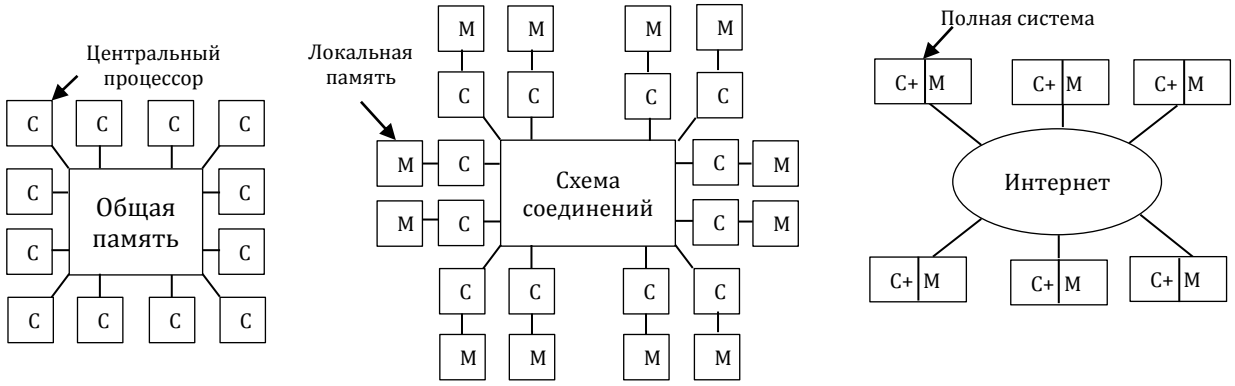
Еще один способ увеличения вычислительной мощности системы заключается в использовании параллельных расчетов на большом количестве отдельных компьютеров, соединенных в локальную или глобальную сеть. В настоящее время продолжается быстрый рост глобальной сети Интернет. Изначально эта сеть создавалась как прототип помехоустойчивой военной системы управления, затем она стала популярной среди ученых кибернетиков, а в последние годы Интернетом начали пользоваться самые широкие слои населения. Относительно недавно Интернет получил еще одно применение: поскольку он связывает по всему миру тысячи компьютеров, было решено использовать эту сеть для решения больших научных проблем. С точки зрения вычислительной мощности система, состоящая из 1000 компьютеров по всему миру, не отличается от такой же системы из 1000 компьютеров, стоящих в одном помещении, хотя характеризуется задержкой и имеет некоторые другие технические характеристики.

Поместить 1 млн. не связанных друг с другом компьютеров в одну комнату достаточно легко при условии, что у вас достаточно денег и достаточно большая комната. Разместить 1 млн. не связанных друг с другом компьютеров по всему миру еще легче, поскольку не нужно искать большого помещения. Проблемы начинаются, когда вам требуется соединить эти компьютеры друг с другом для решения одной общей задачи. Поэтому была проведена большая работа в области технологии межкомпьютерных соединений, а различные технологии привели к появлению качественно отличных типов систем и различной организации программного обеспечения.

Весь обмен информацией между электронными (или оптическими) компонентами сводится, в конечном итоге, к отправке и приему сообщений, представляющих собой строго определенные последовательности битов. Различия состоят во временных параметрах, пространственных масштабах и логической организации. Одну крайность составляют мультипроцессорные системы с общей

оперативной памятью и с числом процессоров от двух до тысячи. В этой модели каждый центральный процессор обладает равным доступом ко всей физической памяти и может читать и писать отдельные слова с помощью команд LOAD и STORE. Время доступа к памяти обычно составляет от 10 до 50 нс. Хотя такая система, показанная на рисунке 9.1, *а*, может показаться простой, ее реализация представляет собой далеко не простую задачу и обычно включает большое количество скрытно передаваемых сообщений.

Следом идут системы (рисунок 9.1, *б*), в которых пары, состоящие из центрального процессора и памяти, соединены высокоскоростной соединительной схемой. Такая разновидность системы называется мультипроцессорной системой с передачей сообщений. Каждый блок памяти является локальным для одного центрального процессора, и доступ к ней может получить только этот центральный процессор. Процессоры общаются, обмениваясь сообщениями по соединительной схеме. При хорошем соединении для передачи сообщения может потребоваться от 10 до 50 мкс, что все же значительно больше, чем время доступа к памяти в схеме на рисунке 9.1, *а*. В этой схеме нет общей глобальной памяти. Мультикомпьютеры (то есть системы с передачей сообщений) гораздо легче создать, чем мультипроцессоры (системы с общей памятью), но писать программы для них значительно труднее. Поэтому у каждого жанра есть свои поклонники.



а *б* *в*

Рисунок 9.1 – Мультипроцессорная система с общей памятью (*а*); мультипроцессорная система с передачей сообщений (*б*); глобальная распределенная система (*в*)

Третья модель, показанная на рисунке 9.1, *в*, представляет собой большое количество полноценных компьютеров, соединенных глобальной сетью, такой как Интернет, и образующих вместе распределенную систему. У каждого компьютера есть своя собственная память. Компьютеры в распределенной системе общаются,

обмениваясь друг с другом сообщениями. Основное различие схем на рисунке 9.1, *б* и рисунке 9.1, *в* заключается в том, что во втором случае используются полноценные компьютеры, а время передачи сообщений составляет от 10 до 50 мс, то есть примерно еще в 1000 раз больше. Из-за большей величины задержки применение этих слабосвязанных систем отличается от использования сильносвязанных систем, показанных на рисунке 9.1, *б*. Величина задержки у всех трех типов систем отличается друг от друга на три десятичных порядка. Это отличие примерно соответствует разнице между одним днем и тремя годами.

Рассмотрим сначала системы с общей памятью (мультипроцессоры). При правильной организации работы таких систем один процессор пишет данные в общую память, а другой их считывает. У всех мультипроцессоров каждый ЦП может адресоваться ко всей памяти. Однако по характеру доступа к памяти эти машины делятся на два класса. Мультипроцессоры, у которых каждое слово данных может быть считано с одинаковой скоростью, называются **UMA**-мультипроцессорами (Uniform Memory Access – однородный доступ к памяти). В противоположность им мультипроцессоры **NUMA** этим свойством не обладают. Такие мультипроцессоры используются, когда число процессоров превышает 100.

С точки зрения операционных систем возможны различные варианты:

1. Каждому ЦП – свою ОС.

Простейший способ организации мультипроцессорных операционных систем состоит в том, чтобы статически разделить оперативную память по числу центральных процессоров и дать каждому центральному процессору свою собственную память с собственной копией операционной системы. В результате *n* центральных процессоров будут работать как *n* независимых компьютеров. В качестве очевидного варианта оптимизации можно позволить всем центральным процессорам совместно использовать код операционной системы и хранить только индивидуальные копии данных (рисунок 9.2). Квадратики, помеченные словом Data, означают приватные данные операционной системы для каждого центрального процессора.

Следует отметить четыре аспекта данной схемы, возможно, не являющихся очевидными. Во-первых, когда процесс обращается к системному вызову, системный вызов перехватывается и обрабатывается его собственным центральным процессором при помощи структур данных в таблицах операционной системы.

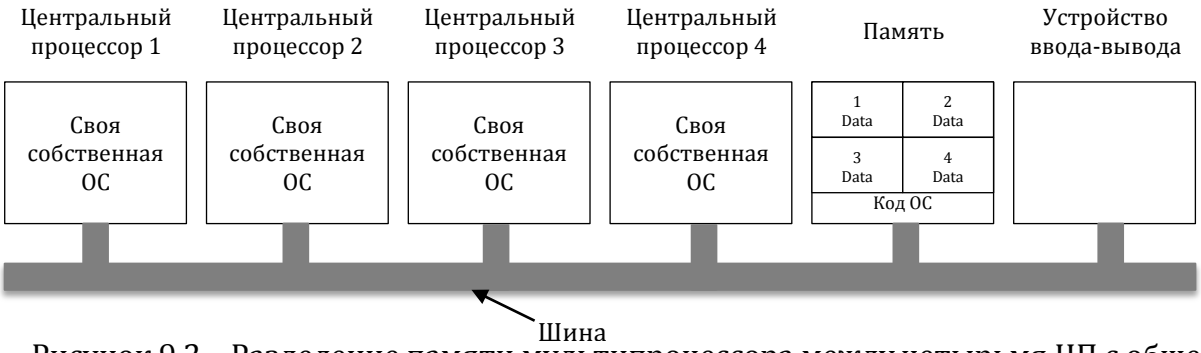


Рисунок 9.2 – Разделение памяти мультипроцессора между четырьмя ЦП с общей копией кода ОС

Во-вторых, поскольку у каждой операционной системы есть свои собственные таблицы, у нее есть также и свой набор процессов, которые она сама планирует. Совместного использования процессов нет. Если пользователь регистрируется на центральном процессоре 1, то все его процессы работают на центральном процессоре 1, в результате может случиться так, что центральный процессор 1 окажется загружен работой, тогда как центральный процессор 2 будет простаивать.

В-третьих, совместного использования страниц также нет. Может случиться так, что у центрального процессора 2 много свободных страниц, в то время как центральный процессор 1 будет постоянно заниматься свопингом. И нет никакого способа занять свободные страницы у соседнего процессора, так как выделение памяти статически фиксировано.

В-четвертых, и это хуже всего, если операционная система поддерживает буферный кэш недавно использованных дисковых блоков, то каждая операционная система будет выполнять это независимо от остальных. Таким образом, может случиться так, что некоторый блок диска будет присутствовать в нескольких буферах одновременно, причем в нескольких буферах сразу он может оказаться модифицированным, что приведет к порче данных на диске. Единственный способ избежать этого заключается в полном отказе от блочного кэша, что значительно снизит производительность системы.

2. Мультипроцессоры типа "хозяин-подчиненный".

По причине приведенных выше соображений такая модель теперь используется редко, хотя она применялась на заре эпохи мультипроцессоров, когда ставилась цель просто перенести существующие операционные системы на какой-либо новый мультипроцессор как можно быстрее. Вторая модель показана на рисунке 9.3. Здесь используется всего одна копия операционной системы, находящаяся на центральном процессоре 1 и отсутствующая на других центральных

процессорах. Все системные вызовы перенаправляются для обработки па центральный процессор 1. Центральный процессор 1 может также выполнять процессы пользователя, если у него будет оставаться для этого время. Такая схема называется **"хозяин-подчиненный"**, так как центральный процессор 1 является "хозяином", то есть ведущим, а все остальные процессоры – подчиненными, или ведомыми.

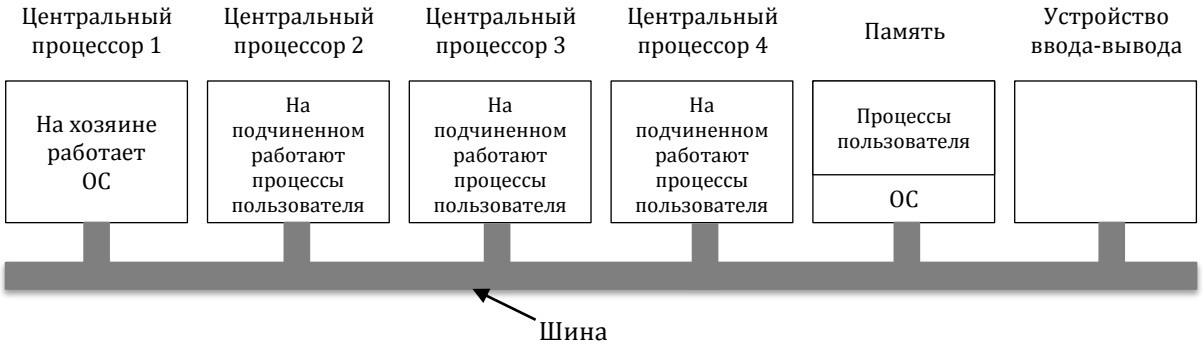


Рисунок 9.3 – Модель мультипроцессора "хозяин-подчиненный"

Модель мультипроцессора "хозяин-подчиненный" позволяет решить большинство проблем первой модели. В этой модели используется единая структура данных (например, один общий список или набор приоритетных списков), учитывающий готовые процессы. Когда центральный процессор переходит в состояние простоя, он запрашивает у операционной системы процесс, который можно обрабатывать, и при наличии готовых процессов операционная система назначает этому процессору процесс. Поэтому при такой организации никогда не может случиться так, что один центральный процессор будет простаивать, в то время как другой центральный процессор перегружен. Страницы памяти могут динамически предоставляться всем процессам. Кроме того, в такой системе есть всего один общий буферный кэш блочных устройств, поэтому дискам не грозит порча данных, как в предыдущей модели при попытке использования блочного кэша.

Недостаток этой модели состоит в том, что при большом количестве центральных процессоров хозяин может стать узким местом системы. Ведь ему приходится обрабатывать все системные вызовы от всех центральных процессоров. Например, если обработка системных вызовов занимает 10% времени, тогда 10 центральных процессоров завалят хозяина работой, а при 20 центральных процессорах хозяин уже не будет успевать их обрабатывать, и система начнет

простаивать. Следовательно, такая модель проста и работоспособна для небольших мультипроцессоров, но на больших она работать не может.

3. Симметричные мультипроцессоры.

Третья модель, представляющая собой **симметричные мультипроцессоры** (SMP, Symmetric Multiprocessor), позволяет устранить перекос предыдущей модели. Как и в предыдущей схеме, в памяти находится всего одна копия операционной системы, но выполнять ее может любой процессор. При системном вызове на центральном процессоре, обратившемся к системе с системным вызовом, происходит прерывание с переходом в режим ядра и обработкой системного вызова. Модель симметричного мультипроцессора показана на рисунке 9.4.

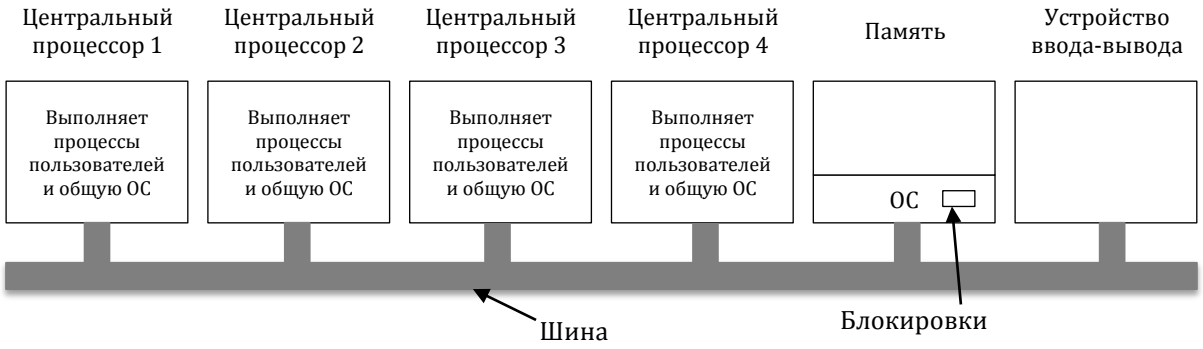


Рисунок 9.4 – Модель симметричного мультипроцессора

Эта модель обеспечивает динамический баланс процессов и памяти, поскольку в ней имеется всего один набор таблиц операционной системы. Она также позволяет избежать простоя системы, связанного с перегрузкой ведущего центрального процессора ("хозяина"), так как в ней нет ведущего центрального процессора. И все же данная модель имеет собственные проблемы. В частности, если код операционной системы будет выполняться одновременно на двух или более нейтральных процессорах, произойдет катастрофа. Представьте себе два центральных процессора, одновременно берущих один и тот же процесс для запуска или запрашивающих одну и ту же свободную страницу памяти. Простейший способ разрешения подобных проблем заключается в связывании мьютекса (то есть блокировки) с операционной системой, в результате чего вся система превращается в одну большую критическую область. **Мьютекс** – это бит, который изменяет свое значение в зависимости от того, занят ресурс или свободен. Когда центральный процессор хочет выполнять код операционной системы, он должен сначала получить мьютекс. Если мьютекс заблокирован, процессор вынужден ждать. Таким

образом, любой центральный процессор может выполнить код операционной системы, но в каждый момент времени только один из них будет делать это.

9.2. Мультикомпьютеры

Привлекательность и популярность мультипроцессоров связана с тем, что они предлагают простую коммуникативную модель. Однако большие мультипроцессоры сложны в построении и поэтому дороги. Для решения этой проблемы были разработаны **мультикомпьютеры (многомашинные системы)**, представляющие собой тесно связанные ЦП, у которых нет общей памяти. У каждого из них своя отдельная оперативная память.

Создание таких систем не представляет сложности, так как их основными компонентами являются усеченные варианты обычных персональных компьютеров.

Базовый узел многомашинной системы состоит из ЦП, памяти, сетевого интерфейса и иногда жесткого диска. При этом каждый узел может включать мультипроцессорную систему. Для создания многомашинных систем могут объединяться сотни и даже тысячи узлов. Существуют различные топологии соединения узлов.

В небольшой системе может существовать один коммутатор, к которому присоединяются все узлы, образуя топологию "звезда" (рисунок 9.5, *а*). Подобная топология применяется в современных коммутируемых сетях Ethernet, Fast Ethernet и Gigabit Ethernet.

Альтернативная топология может представлять собой кольцо, в котором каждый узел соединяется с двумя соседними узлами без помощи коммутаторов (рисунок 9.5, *б*).

Третий вариант топологии представляет собой решетку или сеть (рисунок 9.5, *в*). Это уже двумерная топология, применяемая во многих коммерческих системах. Она обладает высокой степенью регулярности и легко масштабируется до больших размеров. Топология решетки имеет диаметр, которым называют самый длинный путь между двумя узлами. Диаметр решетки увеличивается пропорционально квадратному корню от общего числа узлов решетки. Один из вариантов топологии решетки, у которой крайние узлы соединены друг с другом, называется двойным тором (рисунок 9.5, *г*). Такая схема обладает большей устойчивостью к

повреждениям и сбоям, чем простая решетка, к тому же дополнительные линии связи снижают ее диаметр.

Топология куб (рисунок 9.5, *д*) представляет собой регулярную трехмерную топологию, На рисунке изображен куб размером 2x2x2, но на практике применяются кубы значительно больших размеров. На рисунке 9.5, *е* показан четырехмерный куб, созданный из двух трехмерных кубов с помощью соединений соответствующих узлов. Эту идею можно развивать, создавая пяти- и шестимерные кубы и т. д. Созданный таким образом *n*-мерный куб называется гиперкубом. Подобная топология используется во многих параллельных компьютерах, потому что диаметр у них растет линейно в зависимости от размерности. Другими словами, диаметр представляет собой логарифм по основанию 2 от числа узлов, например, у 10-мерного гиперкуба с 1024 узлами диаметр будет равен всего 10, в результате чего такая схема обладает прекрасными временными характеристиками (низкой задержкой). Обратите внимание, что если организовать эти 1024 узла в виде квадратной решетки 32x32, то диаметр в этом случае будет равен 62, что более чем в шесть раз хуже, чем для гиперкуба. Платой за небольшой диаметр гиперкуба является большое число ответвлений на каждом узле, пропорциональное размерности гиперкуба, и, таким образом, большее количество (и большая стоимость) связей между узлами.

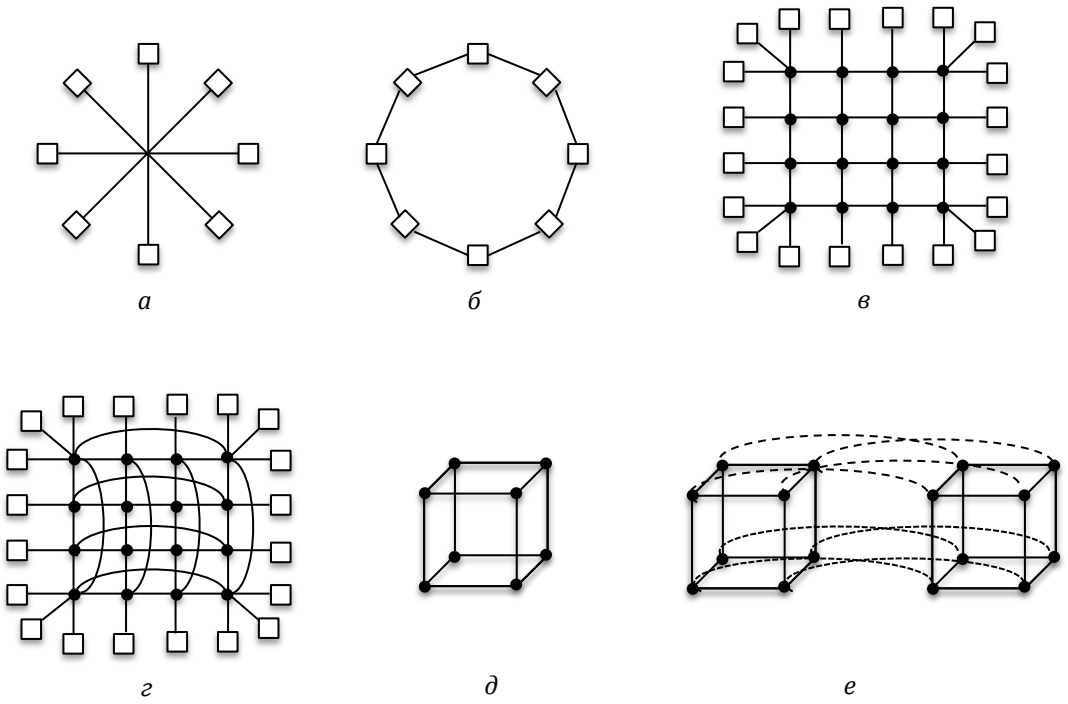


Рисунок 9.5 – Различные топологии соединения узлов: один коммутатор (*а*); кольцо (*б*); решетка (*в*); двойной тор (*г*); куб (*д*); четырехмерный гиперкуб (*е*)

В многомашинных системах применяются две коммутационные схемы. В первой из них – **коммутации пакетов с промежуточным хранением** каждое сообщение сначала разбивается на отдельные фрагменты, называемые **пакетами**. Пакет передается побитно, и когда прибывает весь пакет, он копируется на следующий коммутатор. Когда пакет прибывает на коммутатор, соединенный с пунктом назначения, пакет копируется на сетевую карту узла-получателя и, наконец, попадает в оперативную память этого узла. Минусом такой схемы является увеличение задержки передачи сообщения по сети.

Второй режим коммутации, **коммутация каналов**, заключается в том, что первый коммутатор сначала устанавливает путь через все коммутаторы до коммутатора-получателя. Как только путь установлен, биты от источника к приемнику передаются без остановок. При этом промежуточного хранения не производится. Для коммутации каналов требуется фаза начальной установки, занимающая некоторое время, зато по завершении этапа установки весь процесс передачи данных идет быстрее. Когда сообщение отправлено, путь должен быть снова разорван. Существует вариант коммутации каналов, называемый **червячной маршрутизацией**, при которой каждый пакет разбивается на подпакеты, и первый подпакет начинает передаваться еще до того, как был установлен полный путь

9.3. Распределенные системы

Теперь немного остановимся на распределенных системах. Эти системы сходны с многомашинными в том, что в такой системе нет общей физической памяти, а у каждого узла есть своя память. Но имеется и ряд важных отличий от многомашинных:

1. В отличие от узлов многомашинных систем узлы распределенных систем представляют собой полноценные компьютеры.
2. Узлы многомашинных систем располагаются обычно в одном помещении, что позволяет соединить их высокоскоростной сетью, тогда как узлы распределенных систем могут быть распределены по всему миру.
3. Все узлы многомашинной системы работают под управлением одной ОС, совместно используют единую файловую систему и находятся под общим административным управлением.

Наиболее ярким примером распределенных систем является сеть Интернет.

Компьютеры могу объединяться при помощи различных вариантов топологии: общая шина, кольцо (Token Ring), звезда и др. Самым широко используемым вариантом в настоящее время является звезда. Звезда состоит из узла и хостов. В качестве хостов выступают компьютеры или периферийные устройства. В узлах могут использоваться концентраторы (hub), коммутаторы (switch) или маршрутизаторы (router). Самыми простыми устройствами являются концентраторы, которые сейчас практически не используются. В них сигнал, поступивший на один из портов, уходит на все остальные порты, не зависимо от того, какой порт связан с хостом-получателем. Коммутаторы уже более сложные устройства, они строят таблицу коммутации на основании ответов, приходящих с разных портов от хостов-получателей. Поэтому сигналы по коммутаторам передаются из одного входного порта на один выходной. Маршрутизаторы предназначены для установки в тех узлах, которые являются пограничными между различными подсетями. Их функция – в прокладке маршрутов от хостов одной сети к хостам другой. Они строят таблицу маршрутизации, в которой указываются маршрут с наименьшей стоимостью и запасные маршруты для всех известных маршрутизатору хостов. Поэтому таблица маршрутизации намного больше, чем таблица коммутации. Кроме того, работа маршрутизатора требует больше времени, что вызывает более длительные задержки сигнала, чем при прохождении коммутаторов.

В качестве каналов передачи данных может быть использован телефонный кабель (до 56 Кбит/с), коаксиальный кабель (RG-58, скорость до 10 Мбит/с), витая пара (UTP-3 – до 10 Мбит/с, UTP-5 – до 100 Мбит/с, UTP-5e – до 1 Гбит/с, UTP-6, UTP-6a, UTP-7 – до 10 Гбит/с, UTP-7a – до 40 Гбит/с), волоконно-оптический кабель (до 20 Тбит/с и более), спутниковые каналы (до 5-6 Мбит/с), Wi-Fi (Wireless Fidelity – беспроводная точность воспроизведения, 10-50 Мбит/с, теоретический предел – 600 Мбит/с), BlueTooth (до 1 Мбит/с, теоретически – до 24 Мбит/с).

В таблице 9.1 представлено сравнение трех типов многопроцессорных систем.

Таблица 9.1 – Сравнение типов многопроцессорных систем

Аспект	**Мультипроцессоры**	**Мультикомпьютеры**	**Распределенная система**
Конфигурация узла	ЦП	ЦП, ОЗУ, сетевой интерфейс	Полный компьютер
Периферия узла	Все общее	Общая, кроме, может быть, дисков	Полный набор для каждого узла
Расположение	В одном блоке	В одном помещении	Возможно, по всему миру
Связь между узлами	Общая память	Выделенные линии	Традиционная сеть

Таблица 9.1 (продолжение)

Операционные системы	Одна, общая	Несколько, одинаковые	Могут быть различными
Файловые системы	Одна, общая	Одна, общая	У каждого узла своя
Администрирование	Одна организация	Одна организация	Много организаций

10. БЕЗОПАСНОСТЬ

10.1. Понятие безопасности

Многие компании обладают ценной информацией, которую они тщательно охраняют. Эта информация может быть технической, коммерческой, финансовой, юридической и т.д. По мере того, как возрастают объемы информации, хранящейся в компьютерных системах, необходимость в защите информации становится все важнее.

У проблемы безопасности существует три наиболее важных аспекта. Первым аспектом является **природа угроз**.

В компьютерных системах существует три основных вида угроз:

1. Демонстрация данных.
2. Подделка или порча данных.
3. Отказ обслуживания

В соответствии с наличествующими угрозами с позиций безопасности у компьютерной системы есть три главные задачи:

1. Конфиденциальность данных – секретные данные должны оставаться секретными.
2. Целостность данных – неавторизованные пользователи не должны иметь возможность модифицировать данные без разрешения владельца.
3. Доступность системы – никто не может вывести систему из строя.

Еще одним немаловажным аспектом проблемы безопасности является **право пользователя на конфиденциальность личной информации**.

В литературе по безопасности человека, пытающегося завладеть чужой информацией, называют **злоумышленником**. Все злоумышленники делятся на два вида: **пассивные** – пытаются прочитать данные, которые им читать не разрешено, и **активные** – пытаются незаконно изменить данные.

Наиболее распространенными категориями злоумышленников являются:

1. Случайные любопытные пользователи, не применяющие специальных технических средств. У многих людей есть компьютеры, соединенные с общим файловым сервером. И если не установить специальной защиты, благодаря естественному любопытству многие люди станут читать чужую электронную почту и другие файлы. Например, во многих системах UNIX новые, только что созданные файлы по умолчанию доступны для чтения всем желающим.

2. Члены организации, занимающиеся шпионажем. Студенты, системные программисты, операторы и другой технический персонал часто считают взлом системы безопасности локальной компьютерной системы личным вызовом. Как правило, они имеют высокую квалификацию и готовы посвящать достижению поставленной перед собой цели значительное количество времени.

3. Те, кто совершают решительные попытки личного обогащения. Некоторые программисты, работающие в банках, предпринимали попытки украсть деньги у банка, в котором они работали. Использовавшиеся схемы варьировались от изменения способов округления сумм в программах, для сбора, таким образом, с миру по нитке, до шантажа ("Заплатите мне, или я уничтожу всю банковскую информацию").

4. Занимающиеся коммерческим и военным шпионажем. Шпионаж представляет собой серьезную и хорошо финансируемую попытку конкурента или другой страны украсть программы, коммерческие тайны, ценные идеи и технологии, схемы микросхем, бизнес-планы и т. д. Часто такие попытки включают подключение к линиям связи или установку антенн, направленных на компьютер для улавливания его электромагнитного излучения.

Очевидно, что попытка предотвратить кражу военных секретов враждебным иностранным государством отличается от противостояния попыткам студентов установить забавные сообщения в систему. Необходимые для поддержания секретности и защиты усилия зависят от предполагаемого противника.

Еще одной разновидностью угрозы безопасности является **вирус**. В общем случае вирус представляет собой программу, реплицирующую саму себя и (как правило) причиняющую тот или иной ущерб. В определенном смысле автор вируса также является злоумышленником, часто обладающим высокой квалификацией. Основное различие между обычным злоумышленником и вирусом состоит в том, что первый из них представляет собой человека, лично пытающегося взломать систему с целью причинения ущерба, тогда как вирус является программой, написанной таким человеком и выпущенной в свет с надеждой на причинение ущерба. Злоумышленники пытаются взломать определенные системы (например, сеть какого-либо банка или Пентагона), чтобы украсть или уничтожить определенные данные, тогда как вирус обычно действует не столь направленно. Таким образом, злоумышленника можно уподобить наемному убийце, пытающемуся уничтожить

конкретного человека, в то время как автор вируса больше напоминает террориста, пытающегося убить большое количество людей, а не кого-либо конкретно.

Помимо различных угроз со стороны злоумышленников, существует третий аспект – **опасность потери данных в результате несчастного случая**. Наиболее частыми причинами случайной потери данных являются:

1. Форс-мажор – пожары, наводнения, крысы и т.д.
2. Аппаратные и программные ошибки – сбои ЦП, нечитаемые диски, ошибки в программах и при передаче данных и т.д.
3. Человеческий фактор – неправильный ввод данных, запуск не той программы и т.д.

Большая часть этих проблем может быть разрешена при помощи своевременного создания соответствующих резервных копий, хранимых на всякий случай вдали от оригинальных данных. Хотя проблема защиты информации от случайных потерь кажется пустяковой по сравнению с задачей противостояния умным злоумышленникам, на практике больше ущерба наносят именно несчастные случаи.

10.2. Основы криптографии

Задача криптографии заключается в том, чтобы взять сообщение или файл, называемый **открытым текстом**, и преобразовать его в **зашифрованный текст** таким образом, чтобы только посвященные могли преобразовать его обратно в открытый текст. При этом алгоритмы шифрации и дешифрации всегда должны быть открытыми, каким бы странным это ни казалось.

Секретность зависит от параметров алгоритмов, называемых **ключами**. Мы будем использовать формулу $C = E(P, K_E)$, обозначающую, что при зашифровке открытого текста P с помощью ключа K получается зашифрованный текст C. Аналогично, формула $P = D(C, K_E)$ означает расшифровку зашифрованного текста C для восстановления открытого текста. Схематично процессы шифрования и дешифрования показаны на рисунке 10.1.

Существует шифрование двух видов – шифрование с секретным ключом и шифрование с открытым ключом. Многие криптографические системы, обладают тем свойством, что по ключу шифрования легко найти ключ дешифрации, и наоборот. Такие системы называются системами **шифрования с секретным ключом** или системами **шифрования с симметричным ключом**.

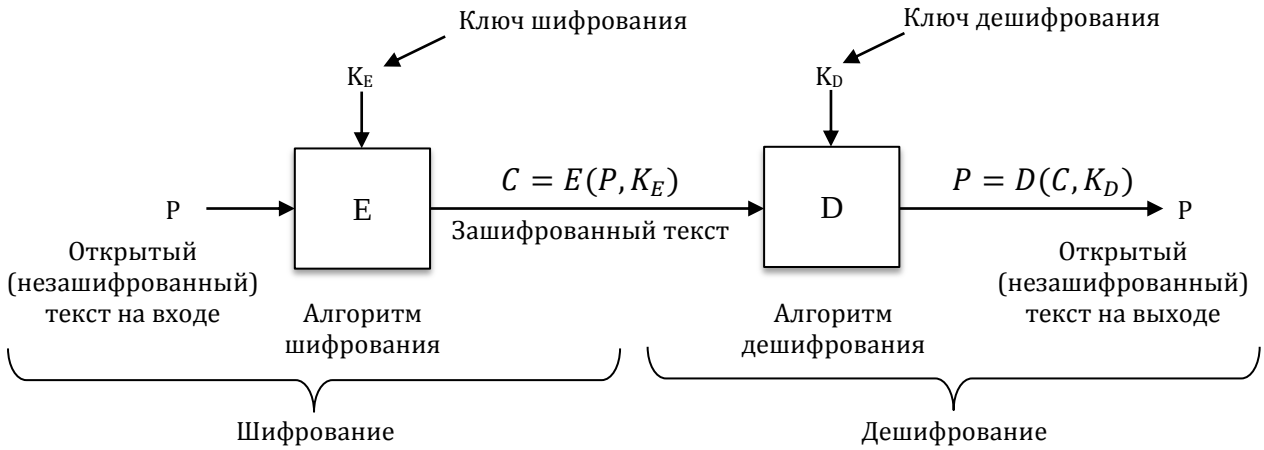


Рисунок 10.1 – Открытый текст и зашифрованный текст

Самый простой пример алгоритма шифрования с секретным ключом – **моноалфавитная подстановка**, в которой каждая буква заменяется другой буквой.

На первый взгляд такая система может показаться надежной, так как существует для английского языка 26! = 4×10^{26} вариантов применения ключа, а для русского – 33! = 8×10^{36} вариантов. Однако если известно, на каком языке написан оригинал, то можно ориентироваться по статистическим характеристикам языка. То есть по частоте встречаемости определенных букв в словах языка оригинала. При этом с той же частотой будут встречаться заменяющие их буквы. Хотя можно использовать различные варианты, увеличивающие сложность дешифровки человеком, не имеющим ключа. Для серьезного уровня безопасности, вероятно, следует использовать ключи длиной в 1024 бит. При такой длине ключа пространство ключей составит $2^{1024} = 2 \times 10^{308}$ ключей. Более короткие ключи могут остановить любителей, но не специальные службы развитых государств.

Системы с секретным ключом эффективны благодаря своей простоте, однако у них есть серьезный недостаток – и отправитель, и получатель должны обладать общим секретным ключом.

Для решения этой проблемы применяется шифрование с открытым ключом. Главное свойство этой системы заключается в том, что для шифрования и дешифрования используются различные ключи и по заданному ключу шифрования определить ключ дешифрации практически невозможно. Например, для шифрования используется простая операция, а для дешифрации без ключа придется выполнить огромный объем сложных вычислений.

В качестве примера можно привести две задачи – одна – это возвести в квадрат число 1234567890, а другая – извлечь квадратный корень из числа 1524157875019052100. Это взаимно противоположные операции, однако с первой

справится любой шестиклассник, а вот вторую без калькулятора решит далеко не каждый взрослый. Такой вид асимметрии и формирует основу криптографии с открытым ключом.

Система шифрования с открытым ключом **RSA** использует тот факт, что перемножить два больших числа значительно легче, чем разложить большое число на множители, особенно когда в числах сотни цифр. Главный недостаток систем шифрования с открытым ключом заключается в том, что они в тысячи раз медленнее.

Шифрование с открытым ключом используется следующим образом. Все участники выбирают пару ключей (открытый ключ, закрытый ключ) и публикуют открытый ключ. Открытый ключ используется для шифрования, а закрытый – для дешифрации. Как правило, формирование ключей автоматизировано, иногда в качестве начального числа используется пароль, выбираемый пользователем. Чтобы отправить пользователю секретное сообщение, корреспондент зашифровывает его открытым ключом получателя. Поскольку закрытый ключ есть только у получателя, только он один сможет расшифровать сообщение.

Есть множество ситуаций, когда требуется, некая функция f, обладающая тем свойством, что при заданной функции f и параметре x вычисление $y = f(x)$ легко выполнимо, но по заданному $f(x)$ найти значение x невозможно по вычислениям. Такая функция, как правило, перемешивает биты сложным образом. Вначале она может присвоить числу y значение x. Затем в ней может располагаться цикл, выполняющийся столько раз, сколько в числе x содержится единичных битов. На каждом цикле биты числа y перемешиваются способом, зависящим от номера цикла, плюс к числу y прибавляются определенные константы. Такие функции называются необратимыми.

Теперь немного поговорим о цифровых подписях. Они используются тогда, когда необходимо, с одной стороны, обеспечить сохранность информации, передаваемой по открытым каналам, а с другой – подтвердить то, что отправителем является именно тот, кто заявлен. Например, предположим, клиент банка посылает банку по электронной почте сообщение с поручением купить для него определенные акции. Через час после того, как это сообщение было отправлено и поручение исполнено, биржа рушится. Теперь клиент отрицает, что он отправлял сообщение банку. Банк, естественно, воспроизводит сообщение, но клиент заявляет, что банк его подделал. Как судье определить, кто говорит правду? С помощью

цифровой подписи можно подписывать сообщения, посылаемые по электронной почте, и другие цифровые документы таким образом, чтобы отправитель не смог потом отрицать, что посылал их.

Один из основных способов получения цифровой подписи заключается в том, что документ пропускается через необратимый алгоритм хэширования, который очень трудно инвертировать (рисунок 10.2, *а*). Хэш-функция формирует результат фиксированной длины, независящей от изначальной длины документа.

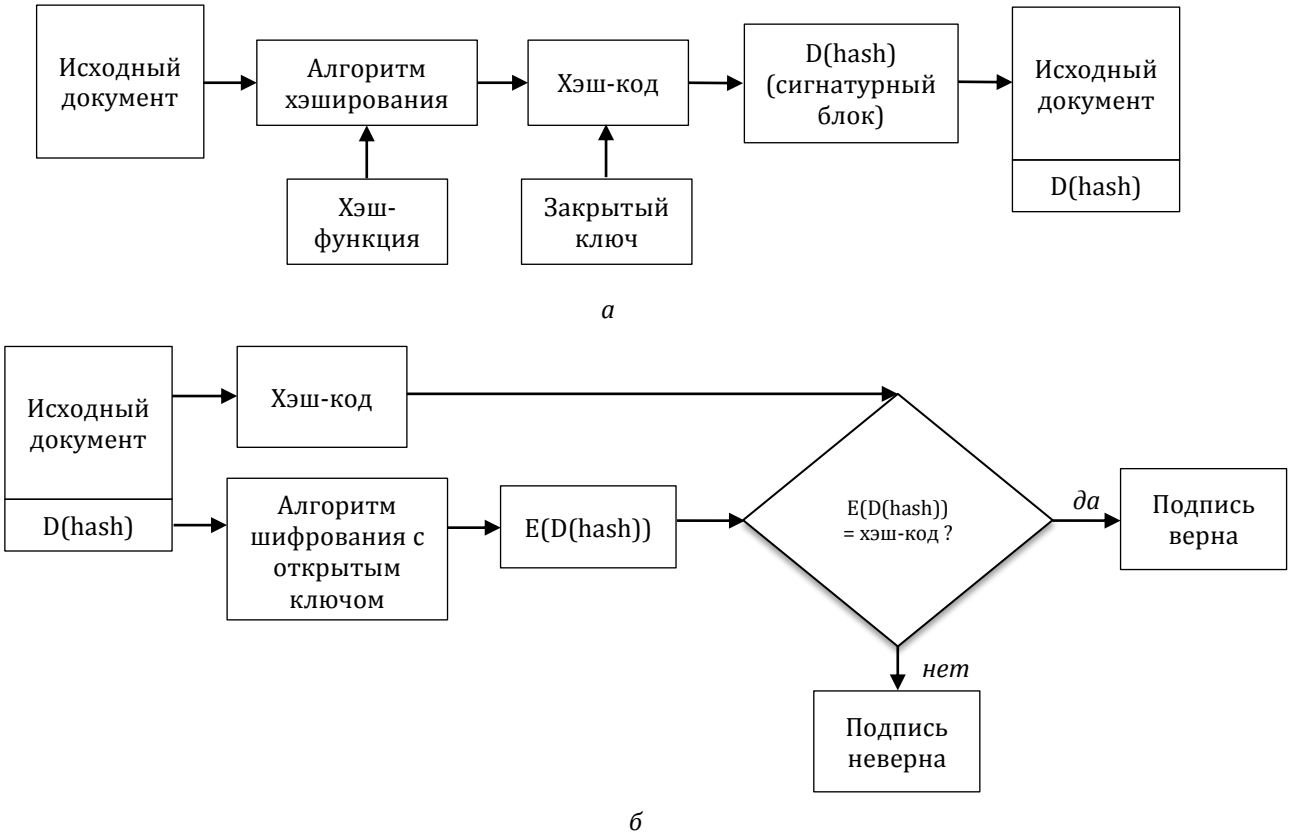


Рисунок 10.2 – Использование электронной цифровой подписи: создание (*а*); проверка (*б*)

На следующем шаге предполагается использовать криптографию с открытым ключом. Владелец документа с помощью своего закрытого ключа из хэша получает *D(hash)*. Это значение, называемое **сигнатурным блоком**, добавляется к документу и посылается получателю.

Когда документ и хэш-код прибывают, получатель сначала с помощью соответствующего алгоритма вычисляет хэш-код документа (рисунок 10.2, *б*). Затем получатель применяет к сигнатурному блоку алгоритм шифрования с открытым ключом, получая *E(D(hash))*. В результате он снова получает оригинальное значение хэш-кода. Если вычисленный заново хэш-код не совпадает с расшифрованным сигнатурным блоком, это значит, что либо сообщение, либо сигнатурный блок были

повреждены – или случайно, или преднамеренно. Смысл этой схемы в том, что медленное шифрование с открытым ключом применяется только для небольшого по размерам хэш-кода. Обратите внимание, что данный метод работает только в том случае, если для всех *x*

$$E\big(D(x)\big) = x.$$

Такое свойство не гарантировано априори для всех функций шифрования, так как все, что изначально требовалось, – это чтобы

$$D\big(E(x)\big) = x,$$

то есть E представляет собой функцию шифрования, а D – функцию дешифрования. Для возможности применения этих функций в цифровых подписях требуется, чтобы порядок применения функций не имел значения, то есть функции D и E должны обладать свойством коммутативности.

Чтобы использовать схему электронной подписи, получатель должен знать открытый ключ отправителя. Некоторые пользователи публикуют свойства открытых ключей на своих web-страницах. Другие этого не делают, так как опасаются, что злоумышленник может взломать страницу и незаметно подменить ключ. Для защиты от подобных действий требуется специальный механизм распределения открытых ключей. Один из широко применяемых методов заключается в том, что отправитель прикрепляет к сообщению **сертификат**, содержащий имя пользователя, и открытый ключ, подписанные ключом доверенной третьей стороны. Как только пользователь получит открытый ключ третьей стороны, он может получать сертификаты ото всех отправителей, использующих эту доверенную третью сторону для создания своих сертификатов.

10.3. Аутентификация пользователей

Когда пользователь регистрируется на компьютере, ОС, как правило, желает определить, кем является данный пользователь, и запускает процесс, называемый **аутентификацией**.

Злоумышленнику, чтобы причинить ущерб какой-либо системе, необходимо сначала зарегистрироваться в ней. Это означает, что он должен преодолеть используемую в данной системе процедуру аутентификации. В популярной прессе таких людей называют **хакерами**. Однако в компьютерном мире слово "хакер" стало чем-то вроде почетного титула, закрепляемого за великими программистами. Таким образом, термин "хакер" стал двузначным. В прессе хакерами обычно называют

жуликов, однако далеко не все высококвалифицированные программисты являются мошенниками. Поэтому для обозначения людей, вламывающихся в чужие компьютерные системы, используется слово **взломщик** (cracker).

Большинство методов аутентификации пользователей основано на распознавании:

– чего-то, известного пользователю;
– чего-то, имеющегося у пользователя;
– чего-то, чем является пользователь.

На этих трех принципах построены три различные схемы аутентификации, обладающие различными сложностями и характеристиками безопасности.

Рассмотрим несколько вариантов аутентификации. Наиболее широко применяется аутентификация на основе **пароля и имени пользователя**. При этом чаще всего в системе хранится список пар "имя пользователя-пароль", с которым и осуществляется сверка вводимых параметров. При этом сам пароль при введении либо не отображается (в UNIX), либо отображается в виде точек или звездочек (в Windows). При этом правильный алгоритм должен выдавать ошибку после введения всей пары, чтобы злоумышленник не знал, где именно он ошибся.

Рассмотрим, каким образом в ОС может осуществляться защита паролей. В некоторых (старых) операционных системах файл паролей хранился на диске в незашифрованном виде, но защищался стандартными системными механизмами защиты. Хранить все пароли на диске в незашифрованном виде означает самому искать неприятности, так как многие люди слишком часто будут иметь доступ к нему. Это системные администраторы, операторы, обслуживающий персонал, программисты, руководители и, возможно, даже секретарши.

Лучшее решение выглядит следующим образом. Программа регистрации просит пользователя ввести свое имя и пароль. Пароль немедленно используется в качестве ключа для шифрации определенного блока данных. То есть выполняется необратимая функция, на вход которой подается пароль. Затем программа регистрации считывает файл паролей, представляющий собой последовательность ASCII-строк, по одной на пользователя, и находит в нем строку, содержащую имя пользователя. Если (зашифрованный) пароль, содержащийся в этой строке, совпадает с только что вычисленным зашифрованным паролем, регистрация разрешается, в противном случае в регистрации пользователю отказывается. Преимущество этой схемы состоит в том, что никто, даже суперпользователь, не

может просмотреть пароли пользователей, поскольку нигде в системе они не хранятся в отрытом виде.

Однако такая схема может быть атакована следующим образом. Взломщик сначала создаст словарь, состоящий, из вероятных паролей. Затем они зашифровываются с помощью известного алгоритма. Не важно, сколько времени занимает данный процесс, так как все это выполняется заблаговременно. Затем, вооружившись списком пар (пароль, зашифрованный пароль), взломщик наносит удар. Он считывает (доступный для всех) файл паролей и сравнивает хранящиеся в нем зашифрованные пароли с содержимым своего списка. Каждое совпадение означает, что взломщику стали известны имя регистрации и соответствующий ему пароль. Этот процесс может быть автоматизирован простым сценарием оболочки. При запуске такого сценария, как правило, можно определить сразу несколько десятков паролей.

Поэтому был разработан метод, делающий данный способ взлома практически бесполезным. Идея заключается в том, что с каждым паролем связывается псевдослучайное число, состоящее из n битов, названное "*солью*". При каждой смене пароля это число также меняется. В файле паролей хранится не зашифрованный пароль, а зашифрованная вместе пара пароля и случайного числа.

Посмотрим теперь, как такой вариант хранения паролей повлияет на описанный выше метод взлома, при котором злоумышленник составлял список вероятных паролей, зашифровывал их и сохранял в отсортированном файле, чтобы ускорить поиск пароля. Теперь, если взломщик предполагает, что, например, *Dog* может быть паролем, ему уже недостаточно зашифровать *Dog* и поместить результат в файл. Теперь ему придется зашифровать уже 2^n строк, таких как *Dog0000*, *Dog0001*, *Dog0002* и т.д., и поместить все эти варианты в файл. В системе UNIX данный метод применяется с $n = 12$, поэтому размер файла увеличивается в 2^{12} раз.

Для повышения надежности в некоторых современных версиях системы UNIX доступ чтения к самому файлу паролей напрямую запрещен, а для регистрации предоставляется специальная программа, просматривающая содержимое этого файла по запросу и устанавливающая задержку между запросами, чтобы существенно снизить скорость подбора паролей при любом методе. Такая комбинация добавления "соли" к паролям, запрета непосредственного чтения файла

паролей и замедления доступа к файлу через специальную процедуру может противостоять многим вариантам методов взлома системы.

Добавление случайных чисел к файлу паролей защищает систему от взломщиков, пытающихся заранее составить большой список зашифрованных паролей и, таким образом, взломать несколько паролей сразу. Однако данный метод бессилен помочь в том случае, когда пароль легко отгадать, например, если пользователь *David* использует пароль *David*. Взломщик может просто попытаться отгадать пароли один за другим. Обучение пользователей в данной области может помочь, но оно редко проводится. Помимо обучения пользователей может использоваться помощь компьютера. На некоторых системах устанавливается программа, формирующая случайные, легко произносимые бессмысленные слова, которые могут использоваться в качестве паролей (желательно с использованием прописных символов и специальных символов, добавленных внутрь). Программа, вызываемая пользователем для установки или смены пароля, может также выдать предупреждение при выборе слабого пароля. Среди требований программы к паролю могут быть, например, следующие:

1. Пароль должен содержать минимум 7 символов.
2. Пароль должен содержать как строчные, так и прописные символы.
3. Пароль должен содержать как минимум одну цифру или специальный символ.
4. Пароль не должен представлять собой слово, содержащееся в словаре, имя собственное и т.д.

Некоторые операционные системы требуют от пользователей регулярной смены паролей, чтобы ограничить ущерб от утечки пароля. Недостаток такой схемы в том, что если пользователи должны слишком часто менять свои пароли, они достаточно быстро устают придумывать и запоминать хорошие пароли и переходят к простым паролям. Если система запрещает выбирать простые пароли, пользователи забывают сложные пароли и начинают записывать их на листках бумаги, приклеиваемых к мониторам, что становится главной дырой в защите.

Предельный вариант частой смены паролей представляет собой использование одноразовых паролей. При использовании одноразовых паролей пользователь получает блокнот, содержащий список паролей. Для каждого входа в систему используется следующий пароль в списке. Если взломщику и удастся узнать уже использованный пароль, он ему не пригодится, так как каждый раз

используется новый пароль. Предполагается, что пользователь постарается не терять блокнот с паролями.

В действительности без такого блокнота можно обойтись, если применить элегантную схему, разработанную Лесли Лампортом, гарантирующую пользователю безопасную регистрацию в небезопасной сети при использовании одноразовых паролей. Метод Лампорта может применяться для того, чтобы позволить пользователю с домашнего персонального компьютера регистрироваться на сервере по Интернету, даже в том случае, если злоумышленники смогут просматривать и копировать весь его поток в обоих направлениях. Более того, никаких секретов не нужно хранить ни на домашнем персональном компьютере, ни на сервере.

Алгоритм основан на необратимой функции, то есть функции $y = f(x)$, обладающей тем свойством, что по заданному *x* легко найти *y*, но по заданному *y* подобрать *x* невозможно по вычислениям. Вход и выход должны иметь одинаковую длину, например 128 битов.

Пользователь выбирает секретный пароль, который он запоминает. Затем он также выбирает целое число *n*, означающее количество одноразовых паролей, формируемое алгоритмом. Для примера рассмотрим $n = 4$, хотя на практике используются намного большие значения. Пусть секретный пароль равен *s*. Тогда первый пароль получается в результате выполнения необратимой функции $f(s)$ *n* раз (то есть четыре раза):

$$P_1 = f(f\left(f\left(f(s)\right)\right)).$$

Второй пароль получается, если применить необратимую функцию $f(s)$ *n*-1 раз (то есть три раза) и т.д. Основной момент, на который следует обратить здесь внимание, заключается в том, что при использовании данного метода легко вычислить предыдущий пароль, но почти невозможно определить следующий. Например, по данному P_2 легко найти P_1, но невозможно определить P_3.

Еще один вариант – это использование схемы аутентификации **"оклик-отзыв"**, когда пользователю отправляется вопрос, на который он должен ответить соответствующим образом. Это может быть и заранее определенная функция, например x^2, тогда пользователь, получая каждый раз разные числа, должен в ответ указать квадрат полученного числа. Также на компьютер пользователя при регистрации может посылаться произвольное число *r*, являющееся параметром для вычисляемой функции $f(r, k)$, где f – не являющаяся секретной функция, а k – заранее выбранный секретный ключ, который вручную заносится на сервер.

Получив результат, сервер сверяет его с результатом своих аналогичных вычислений. Если результаты одинаковы, то доступ разрешается. Преимущество такой схемы перед обычным паролем заключается в том, что если злоумышленник даже запишет весь трафик в обоих направлениях, он не сможет получить информацию, которая поможет ему в следующий раз. Конечно, вычисляемая функция должна быть достаточно сложной, чтобы даже при большом количестве наблюдений злоумышленник не смог вычислить значение *k*.

Второй метод аутентификации пользователей заключается в проверке некоторого физического объекта, который есть у пользователя, а не информации, которую он знает. Например, в течение столетий применялись металлические дверные ключи. Сегодня этим физическим объектом часто является **пластиковая карта**, вставляемая в специальное устройство чтения, подключенное к терминалу или компьютеру. Как правило, пользователь должен не только вставить карту, но также ввести пароль, чтобы предотвратить использование потерянной или украденной карты. С этой точки зрения использование банкомата (ATM, Automatic Teller Machine) начинается с того, что пользователь регистрируется на компьютере банка с удаленного терминала (банкомата) при помощи пластиковой карты и пароля. Сегодня в большинстве стран применяется PIN-код (PIN, Personal Identification Number – личный идентификационный номер), состоящий всего из 4 цифр, что позволяет избежать необходимости установки полной клавиатуры на банкоматы.

Существует две разновидности пластиковых карт, хранящих информацию: магнитные карты и карты с микросхемами. **Магнитные карты** содержат около 140 байт информации, записанной на магнитной ленте, приклеенной к пластику. Эта информация может быть считана терминалом и передана на центральный компьютер. Часто эти данные содержат пароль пользователя (например, его PIN-код), так что терминал может сам проверить подлинность пользователя без помощи головного компьютера. Как правило, пароль шифруется ключом, известным только банку. Применять магнитные карты для идентификации рискованно, так как устройства чтения и записи этих карт дешевы и широко распространены.

Карты, содержащие в себе микросхемы, в свою очередь, подразделяются на две категории: карты, хранящие информацию, и смарт-карты (smart card – "умная" карта). **Карты, хранящие информацию**, содержат небольшое количество памяти (как правило, менее 1 Кбайт), использующей технологию EEPROM (Electrically

Erasable Programmable Read-Only Memory – электрически стираемое программируемое ПЗУ). На такой карте нет центрального процессора, поэтому сохраняемое значение должно изменяться внешним центральным процессором (в считывающем устройстве). Такие карты применяются, например, в качестве телефонных карт, деньги за которые заплачены заранее. При звонке телефон просто уменьшает на единицу значение в карте, но деньги при этом из рук в руки не переходят. По этой причине такие карты в основном выпускаются одной компанией для использования только на их машинах (скажем, телефонах или торговых автоматах). Их можно использовать для аутентификации пользователя при регистрации и хранить на них пароль размером в 1 Кбайт, который посылается на удаленный компьютер, но это редко делается.

Однако сегодня большой объем работ в сфере безопасности проводится со **смарт-картами**. На данный момент они обладают, как правило, 8-разрядным центральным процессором, работающим с тактовой частотой 13,56 МГц, 64 Кбайт ПЗУ, 32 – 256 Кбайт EEPROM и каналом связи со скоростью до 212 кбит/с для обмена данными с устройством чтения. Со временем эти "умные" карты становятся все "умнее", но они ограничены по многим параметрам, включая толщину микросхемы (так как она должна быть встроена в карту), ширину микросхемы (чтобы избежать ее поломки, когда пользователь сгибает карту) и стоимость.

Смарт-карты могут использоваться для хранения денег, как и карты, хранящие данные, но по сравнению с этим видом карт смарт-карты обладают значительно лучшими характеристиками в плане безопасности и универсальности. Эти карты можно зарядить деньгами в банкомате или дома по телефону с помощью специального устройства, поставляемого банком. Позднее, например, в магазине, пользователь может разрешить снять с карты определенную денежную сумму (напечатав YES), в результате чего карта посылает небольшое шифрованное сообщение продавцу. Затем продавец может передать это сообщение в банк, чтобы получить указанную в нем сумму.

Основная концепция безопасности смарт-карт проста: она представляет собой маленький, защищенный от подделок компьютер, способный вступить в диалог (называемый протоколом) с центральным компьютером для аутентификации пользователя. Так, у смарт-карт есть много других возможных применений (хранение в надежно закодированном виде сведений о состоянии здоровья владельца, например данных об аллергии на определенные лекарства и т. п.), но нас

сейчас интересует другое. Вопрос в том, как можно использовать смарт-карты для безопасной регистрации. Так, пользователь, желающий приобрести товары на коммерческом web-сайте, может вставить смарт-карту в домашний адаптер, соединенный с персональным компьютером. Это обеспечит не только более надежную аутентификацию пользователя, но также позволит web-сайту сразу же вычесть нужную сумму из смарт-карты, что позволит избежать основных накладных расходов (и риска), связанных с использованием кредитной карты при покупках в режиме on-line.

Со смарт-картами могут применяться различные схемы аутентификации. Простой протокол "оклик-отзыв" работает следующим образом. Сервер посылает 512-разрядное случайное число смарт-карте, которая добавляет к нему 512-разрядный пароль, хранящийся в электрически стираемом программируемом ПЗУ. Затем сумма возводится в квадрат, и средние 512 бит посылаются обратно на сервер, которому известен пароль пользователя, поэтому сервер может произвести те же операции и проверить правильность результата. Если даже злоумышленник видит оба сообщения, он не может определить по ним пароль. Сохранять эти сообщения также нет смысла для взломщика, так как в следующий раз сервер пошлет пользователю другое 512-разрядное случайное число. Конечно, вместо возведения в квадрат может применяться (и, как правило, применяется) более хитрый алгоритм.

И, наконец, третий метод аутентификации основан на измерении физических характеристик пользователя, которые трудно подделать. Они называются **биометрическими** параметрами (отпечатки пальцев, тембр голоса, рисунок радужной оболочки глаза и т.д.).

Работа типичной биометрической системы состоит из двух этапов: внесение пользователя в список и идентификация. Во время первого этапа характеристики пользователя измеряются и оцифровываются. Затем извлекаются существенные особенности, которые сохраняются в записи, ассоциированной с пользователем. Эта запись может храниться на компьютере в централизованной базе данных (например, для регистрации в системе с удаленного компьютера) или в смарт-карте, которую пользователь носит с собой и вставляет в устройство чтения смарт-карт (например, банкомата).

Второй этап процесса представляет собой идентификацию. Пользователь вводит регистрационное имя. Затем система снова производит замеры. Если новые

значения совпадают с хранящимися в записи, регистрация разрешается, в противном случае в регистрации пользователю отказывается. Ввод имени при регистрации нужен, так как измерения не точны, поэтому их сложно использовать в качестве индекса для поиска. Кроме того, два человека могут обладать очень близкими характеристиками, поэтому требование соответствия этих параметров для одного определенного пользователя является значительно более строгим требованием, чем простое совпадение с характеристиками любого пользователя.

Измеряемые характеристики должны отличаться у различных пользователей в достаточно широких пределах, чтобы система могла безошибочно различать разных людей. Например, цвет волос не является хорошим индикатором, так как у очень многих людей волосы одного и того же цвета. Кроме того, эти характеристики не должны сильно изменяться со временем. Например, голос человека может изменяться вследствие простуды, а лицо может выглядеть по-другому благодаря наличию или отсутствию бороды или макияжа во время начальных замеров. Поскольку последующие измерения никогда точно не совпадут с первоначальными, разработчики такой системы должны решить, насколько точным должно быть сходство. В частности, им придется решить, что лучше – отказать пару раз законному пользователю или время от времени разрешать вход в систему жулику. Коммерческий сайт может решить, что лучше терпеть небольшие убытки от мошенников, чем отказывать в регистрации постоянным клиентам, тогда как сервер лаборатории ядерного оружия может решить, что лучше иногда отказать в регистрации настоящему сотруднику, чем позволить пару раз в год войти в систему случайному чужаку.

11. ОПЕРАЦИОННАЯ СИСТЕМА UNIX

11.1. История

Одна из проблем, с которой можно столкнуться при изучении UNIX – это существование множества клонов и версий, включая AIX, BSD, 1BSD, HP-UX, Linux, MINIX, OSF/1, SCO UNIX, System V, Solaris, XENIX и многие другие, причем каждая из них распадается на свои подверсии. Однако фундаментальные принципы и системные вызовы практически для всех этих систем во многом совпадают.

Такое разнообразие версий связано, в том числе, с богатой историей UNIX. Рассмотрим ее поподробнее.

Первой сколько-нибудь успешной операционной системой стала Дартмутская система, разработанная в 60-е годы XX века в Дартмутском колледже и Массачусетском технологическом институте. В этой системе функционировал только BASIC, в связи с чем она достаточно быстро уступила место системе CTSS, разработанной в Массачусетском технологическом институте. Это была универсальная система, получившая огромный успех в научных кругах. Вскоре ее разработчики объединили усилия с лабораторией Bell Labs и корпорацией General Electric и начали разработку системы второго поколения MULTICS (Multiplexed Information and Computing Service – мультиплексная информационная и вычислительная служба). Позднее лаборатория Bell Labs вышла из проекта и разработала более упрощенную систему UNICS (Uniplexed Information and Computing Service – униплексная информационная и вычислительная служба), название которой в дальнейшем трансформировалось в UNIX.

В 70-е годы появилась версия UNIX, позволившая перейти с компьютеров PDP-7 на PDP-11. Для этого пришлось переписать ее на специально разработанный язык B, представляющий собой упрощенную версию языка BCPL, который, в свою очередь, являлся упрощением языка PL/1. Однако язык B оказался не очень удачным и был разработан его преемник – язык C. Этот язык оказался как раз тем языком, который был нужен, и он сохраняет лидирующие позиции в области системного программирования до сих пор. Это способствовало появлению UNIX version 7, которая стала первой переносимой операционной системой, позволявшей работать как на PDP-11, так и на Interdata 8/32. К середине 80-х ОС UNIX широко применялась на компьютерах и инженерных рабочих станциях самых различных производителей. Многие компании приобрели лицензии на исходные тексты, чтобы производить

свои версии системы UNIX. Одной из таких компаний была небольшая начинающая фирма Microsoft, в течение нескольких лет продававшая UNIX version 7 под именем XENIX.

Одним из самых успешных вариантов UNIX стала System V, достаточно долго продержавшаяся на рынке. Она сменила System III, которая была первой коммерческой версией UNIX.

В середине 90-х годов основной собственник ОС UNIX – компания AT&T, являвшаяся учредителем Bell Labs, решила стать телефонной компанией и продала свой бизнес корпорации Novell, а та, в свою очередь – компании Santa Cruz Operation (SCO).

В последствии Калифорнийский университет в Беркли разработал свои версии UNIX – 1BSD, 2BSD, 3BSD и 4BSD. В последней из них появилась поддержка сетей и протокол TCP/IP, ставший стандартом для UNIX, а затем и для всех систем.

Главной проблемой UNIX являлось то, что не существовало единого стандарта для производителей этой ОС. К концу 80-х широкое распространение получили две различные и плохо совместимые версии UNIX: 4.3BSD и System V Release 3. Чтобы как-то объединить эти две системы был организован комитет POSIX (Portable operating system), который смог разработать общий стандарт, известный как **1003.1**. Однако через некоторое время снова произошел раскол на два лагеря, OSF, включающий IBM, DEC и HP, а также UI, возглавляемый AT&T. Правда постепенно рынок сделал выбор в пользу UI, основанной на System V.

За эти годы система UNIX стала противоречить основной идее – она должна быть простой и небольшой. Чтобы вернуться к этому требованию, в 1987 году была разработана ОС MINIX, основанная на UNIX Version 7. На ее основе финским студентом Линусом Торвальдсом в 1991 году была разработана собственная версия, названная Linux.

Linux постепенно стала завоевывать популярность и становилась более мощной. Далеко не последнюю роль в этом сыграло то, что она распространялась бесплатно. Сейчас в Linux более 80% из 150 системных вызовов представляют точные копии системных вызовов в POSIX, BSD или System V.

11.2. Обзор системы

Операционная система UNIX ориентирована на опытных программистов, а не на простых пользователей, что накладывает на нее серьезный отпечаток. Поэтому ей присущ ряд свойств:

1. Так, она должна удовлетворять *принципу наименьшей неожиданности* – если команда ls A* осуществляет поиск всех файлов с именем, начинающимся на букву A, то и команда rm A* должна удалять все файлы с именем на A, а не с именем A*.

2. *Мощь и гибкость* – в системе должно быть небольшое количество базовых элементов, которые можно комбинировать бесконечным числом способов, чтобы приспособить их для конкретного приложения. Так, одно из основных правил UNIX заключается в том, что каждая программа должна выполнять одну функцию, но делать это хорошо.

3. *Отсутствие бесполезной избыточности*. Например, вместо того, чтобы писать *copy*, достаточно написать *cp*. Чтобы получить список всех строк, содержащих строку "ard" из файла *f*, надо написать команду *grep ard f*, а не вводить отдельно команду *grep*, которая выдаст сообщение "я ищу символьные строки, введите, пожалуйста, искомую строку", потом вводить *ard*, получив сообщение от программы "а теперь введите имя файла".

Операционную систему UNIX можно рассматривать в виде пирамиды (рисунок 11.1). У основания пирамиды располагается аппаратное обеспечение, состоящее из центрального процессора, памяти, дисков, терминалов и других устройств. На голом "железе" работает ОС UNIX. Ее функция заключается в управлении аппаратным обеспечением и предоставлении всем программам интерфейса системных вызовов. Эти системные вызовы позволяют программам создавать процессы, файлы и прочие ресурсы, а также управлять ими.

Программы обращаются к системным вызовам, помещая аргументы в регистры центрального процессора (или иногда в стек) и выполняя команду эмулированного прерывания для переключения из пользовательского режима в режим ядра и передачи управления операционной системе UNIX. Поскольку на языке C невозможно написать команду эмулированного прерывания, этим занимаются библиотечные функции, по одной на системный вызов. Эти процедуры написаны на ассемблере, но они могут вызываться из программ, написанных на C. Каждая такая

процедура помещает аргументы в нужное место и выполняет команду эмулированного прерывания TRAP. Таким образом, чтобы обратиться к системному вызову *read*, программа на C должна вызвать библиотечную процедуру *read*. Кстати, в стандарте POSIX определен именно интерфейс библиотечных функций, а не интерфейс системных вызовов. Другими словами, стандарт POSIX определяет библиотечные процедуры, соответствующие системным вызовам, их параметры, что они должны делать и какой результат возвращать. В стандарте даже не упоминаются фактические системные вызовы.

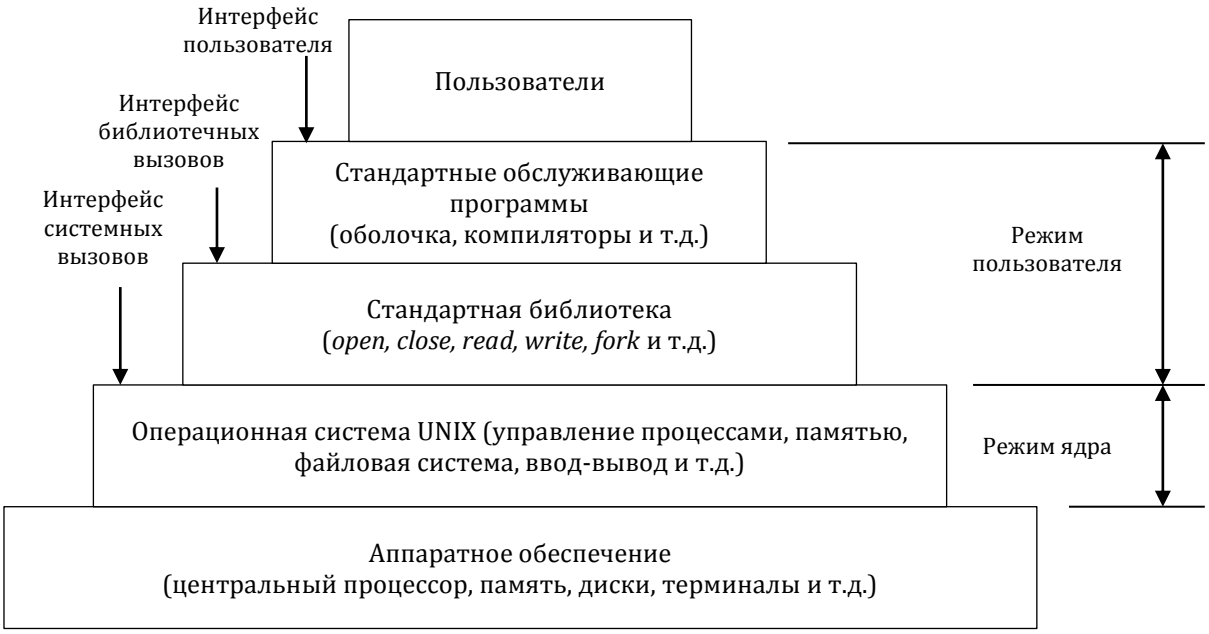


Рисунок 11.1 – Уровни операционной системы UNIX

Помимо ОС и библиотеки системных вызовов, все версии UNIX содержат большое количество стандартных программ, некоторые из них описываются стандартом POSIX 1003.2, тогда как другие могут различаться в разных версиях системы UNIX. К этим программам относятся командный процессор (оболочка), компиляторы, редакторы, программы обработки текста и утилиты для работы с файлами. Именно эти программы и запускаются пользователем с терминала.

Таким образом, можно говорить о трех интерфейсах в ОС UNIX: интерфейсе системных вызовов, интерфейсе библиотечных функций и интерфейсе, образованном набором стандартных обслуживающих программ. Хотя именно последний большинство пользователей считает системой UNIX, в действительности он не имеет практически никакого отношения к самой ОС и легко может быть заменен.

В некоторых версиях системы UNIX, например, этот ориентированный на ввод с клавиатуры интерфейс пользователя был заменен графическим интерфейсом пользователя, ориентированным на использование мыши, для чего не потребовалось никаких изменений в самой системе. Именно эта гибкость сделала систему UNIX столь популярной и позволила ей пережить многочисленные изменения технологии, лежащей в ее основе.

У многих версий системы UNIX имеется графический интерфейс пользователя, схожий с популярными интерфейсами, примененными на компьютере Macintosh и, впоследствии, в системе Windows. Однако истинные программисты предпочитают интерфейс командной строки, называемый **оболочкой** (shell). Он быстрее, мощнее, проще расширяется и не требует работы с мышью.

Когда оболочка запускается, она инициализируется, а затем печатает на экране символ приглашения к вводу (обычно это знак доллара или процента) и ждет, когда пользователь введет командную строку.

После того как пользователь введет командную строку, оболочка извлекает из нее первое слово и ищет файл с таким именем. Если такой файл удается найти, оболочка запускает его. При этом работа оболочки приостанавливается на время работы запущенной программы. По завершении работы программы оболочка снова печатает приглашение и ждет ввода следующей строки. Здесь важно подчеркнуть, что оболочка представляет собой обычную пользовательскую программу. Все, что ей нужно, – это способность ввода с терминала и вывода на терминал, а также возможность запускать другие программы.

В оболочке можно писать команды с использованием (или без) дополнительных параметров, которые называются **флагами** или **ключами** и обозначаются знаком тире. Так, команда head 20 file напечатает первые 10 строк файла с именем 20, а затем – файла с именем file. Команда head -20 file напечатает первые 20 строк файла с именем file.

Кроме того, можно использовать так называемые **волшебные символы** или **джокеры**. К ним относятся, например, * и ?. Кроме того, в квадратных скобках можно указать множество символов, из которых программа должна будет выбрать один.

Программа вроде оболочки не должна открывать терминал, чтобы прочитать с него или вывести на него строку. Вместо этого запускаемые программы автоматически получают доступ к файлу, называемому стандартным устройством ввода (standard input), и к файлу, называемому стандартным устройством вывода

(standard output), а также к файлу, называемому standard error (стандартное устройство для вывода сообщений об ошибках). По умолчанию всем трем устройствам соответствует терминал, то есть клавиатура для ввода и экран для вывода. Многие программы в системе UNIX читают данные со стандартного устройства ввода и пишут на стандартное устройство вывода. Например, команда *sort* вызывает программу *sort*, читающую строки с терминала (пока пользователь не нажмет комбинацию клавиш CTRL+D, чтобы обозначить конец файла), а затем сортирует их в алфавитном порядке и выводит результат на экран.

Стандартные ввод и вывод можно перенаправить с помощью символов < и >. Разрешается их одновременное использование в одной командной строке. Например, команда sort <in >out заставляет программу sort взять в качестве входного файл in и направить вывод в файл out. Поскольку стандартный вывод сообщений об ошибках не был перенаправлен, все сообщения об ошибках будут печататься на экране. Программа, считывающая данные со стандартного устройства ввода, выполняющая определенную обработку этих данных и записывающая результат в поток стандартного вывода, называется **фильтром**.

В системе UNIX возможно использовать несколько команд в одной строке, разделяя их знаком ";". Кроме того, возможно объединение команд в **конвейер**. Это последовательность команд, соединенных символом канала "|". Рассмотрим это на примере команды *sort*. Без использования конвейера ее можно использовать следующим образом:

sort <in >temp; head -30 <temp; rm temp

Эта последовательность команд означает следующее. Сначала запускается программа *sort*, которая принимает данные из файла *in* и записывает результат в файл *temp*. Затем запускается программа *head*, которая выводит первые 30 строк из файла *temp* на экран монитора. И, наконец, файл *temp* удаляется. Ту же самую последовательность в виде конвейера можно записать так:

sort <in | head -30

При этом конвейеры могут быть и более длинными. Например:

grep ter *.t | sort | head -20 | tail -5 >foo

Здесь на монитор выводятся все строки, содержащие строку "ter" во всех файлах, оканчивающихся на ".t", после чего они сортируются. Затем первые 20 строк выбираются командой *head*, которая передает их программе *tail*, записывающей последние 5 строк (с 16 по 20 в отсортированном списке) в файл *foo*.

UNIX является универсальной многозадачной системой. Один пользователь может одновременно запустить несколько программ, каждую в виде отдельного процесса. Синтаксис оболочки для запуска фонового процесса состоит в использовании символа амперсанда в конце строки. Таким образом, строка

wc -l <a >b &

запустит программу подсчета количества слов *wc*, которая сосчитает число строк (флаг -*l*) во входном файле *a* и запишет результат в файл *b*, но будет делать это в фоновом режиме. Как только команда введена пользователем, оболочка напечатает символ приглашения к вводу и перейдет в режим ожидания следующей команды. Конвейеры также могут выполняться в фоновом режиме. Можно одновременно запустить несколько фоновых конвейеров.

Список команд оболочки может быть помещен в файл, а затем этот файл с командами может быть выполнен, для чего нужно запустить оболочку с этим файлом в качестве входного аргумента. Вторая программа оболочки просто выполнит перечисленные в этом файле команды одну за другой, точно так же, как если бы эти команды вводились с клавиатуры. Файлы, содержащие команды оболочки, называются **сценариями оболочки**. Сценарии оболочки могут присваивать значения переменным оболочки и затем считывать их. Они также могут запускаться с параметрами и, кроме того, использовать конструкции *if*, *for*, *while* и *case*. Таким образом, сценарии оболочки представляют собой настоящие программы, написанные на языке оболочки. Существует альтернативная оболочка Berkley C, разработанная таким образом, чтобы сценарии оболочки (и команды языка вообще) выглядели во многих аспектах подобно программам на C. Поскольку оболочка представляет собой всего лишь еще одну пользовательскую программу, было написано много различных ее версий.

Пользовательский интерфейс UNIX состоит не только из оболочки, но также из большого числа стандартных обслуживающих программ, называемых также утилитами. Эти утилиты можно разделить на категории:

1. Команды управления файлами и каталогами.
2. Фильтры.
3. Средства разработки программ (текстовые редакторы и компиляторы).
4. Текстовые процессоры.
5. Системное администрирование.
6. Разное.

Стандарт POSIX определяет синтаксис и семантику менее 100 из этих программ, в основном относящихся к первым трем категориям. Идея стандартизации данных программ заключается в том, чтобы можно было писать сценарии оболочки, которые работали бы на всех системах UNIX. Помимо этих стандартных утилит, разумеется, существует еще масса прикладных программ, таких как web-браузеры, программы просмотра изображений и т.д. В таблице 11.1 перечислены некоторые стандартные утилиты.

Таблица 11.1 – Некоторые утилиты UNIX, требуемые стандартом POSIX

Программа	Функция	Программа	Функция
cat	Конкатенация нескольких файлов в стандартный выходной поток	od	Шестнадцатеричный дамп файла
chmod	Изменение режима защиты файла	paste	Вставка колонок текста в файл
cp	Копирование файлов	pr	Форматирование файла для печати
cut	Вырезание колонок текста из файла	rm	Удаление файлов
grep	Поиск определенной последовательности символов в файле	rmdir	Удаление каталогов
head	Извлечение из файла первых строк	sort	Сортировка строк файла по алфавиту
ls	Распечатка каталога	tail	Извлечение из файла последних строк
make	Компиляция файлов для создания двоичного файла	tr	Преобразование символов из одного набора в другой
mkdir	Создание каталога	sed	Строковый редактор

Теперь поподробнее рассмотрим ядро системы. Обзор структуры ядра системы UNIX представляет собой довольно непростое дело, так как существует множество различных ее версий. Мы рассмотрим структуру ядра на примере UNIX 4.4BSD (рисунок 11.2), хотя она также применима ко многим другим версиям, возможно, с небольшими изменениями в тех или иных местах.

Нижний уровень ядра состоит из драйверов устройств и процедуры диспетчеризации процессов. Все драйверы системы UNIX делятся на два класса: драйверы символьных устройств и драйверы блочных устройств Основное различие между этими двумя классами устройств заключается в том, что на блочных устройствах разрешается операция поиска, а на символьных нет. Технически сетевые устройства представляют собой символьные устройства, но они обрабатываются по-иному, поэтому их, вероятно, правильнее выделить в отдельный класс, как это и было сделано на схеме. Диспетчеризация процессов производится при возникновении прерывания. При этом низкоуровневая программа

останавливает выполнение работающего процесса, сохраняет его состояние в таблице процессов ядра и запускает соответствующий драйвер. Кроме того, диспетчеризация процессов производится также, когда ядро завершает свою работу и пора снова запустить процесс пользователя. Программа диспетчеризации процессов написана на ассемблере и представляет собой отдельную от процедуры планирования программу.

<table>
<tr><td colspan="4">Системные вызовы</td><td colspan="3">Аппаратные и эмулированные прерывания</td></tr>
<tr><td colspan="2">Управление терминалом</td><td>Сокеты</td><td>Именование файла</td><td>Отображение адресов</td><td>Страничные прерывания</td><td rowspan="2">Обработка сигналов</td><td rowspan="2">Создание и завершение процессов</td></tr>
<tr><td rowspan="2">Необработанный телетайп</td><td>Обработанный телетайп</td><td>Сетевые протоколы</td><td>Файловые системы</td><td colspan="2">Виртуальная память</td></tr>
<tr><td>Дисциплины линии связи</td><td>Маршрути-зация</td><td>Буферный кэш</td><td colspan="2">Страничный кэш</td><td colspan="2">Планирование процесса</td></tr>
<tr><td colspan="2">Символьные устройства</td><td>Драйверы сетевых устройств</td><td colspan="3">Драйверы дисковых устройств</td><td>Диспетчеризация процессов</td></tr>
<tr><td colspan="7">Аппаратура</td></tr>
</table>

Рисунок 11.2 – Структура ядра операционной системы UNIX 4.4BSD

В более высоких уровнях программы отличаются в каждом из четырех "столбцов" диаграммы. Слева располагаются символьные устройства. Они могут использоваться двумя способами. Некоторым программам, таким как текстовые редакторы *vi* и *emacs* требуется каждая нажатая клавиша без какой-либо обработки. Для этого служит ввод-вывод с необработанного терминала (телетайпа). Другое программное обеспечение, например оболочка (*sh*), принимает на входе уже готовую текстовую строку, позволяя пользователю редактировать ее, пока не будет нажата клавиша ENTER. Такое программное обеспечение пользуется вводом с терминала в обработанном виде и дисциплинами линии связи.

Сетевое программное обеспечение часто бывает модульным, с поддержкой множества различных устройств и протоколов. Уровень выше сетевых драйверов выполняет своего рода функции маршрутизации, гарантируя, что правильный пакет направляется правильному устройству или блоку управления протоколами. Большинство систем UNIX содержит в своем ядре полноценный маршрутизатор

Интернета, и, хотя его производительность ниже, чем у аппаратного маршрутизатора, эта программа появилась раньше современных аппаратных маршрутизаторов. Над уровнем маршрутизации располагается стек протоколов, обязательно включая протоколы IP и TCP, но также иногда и некоторые дополнительные протоколы. Над сетевыми протоколами располагается интерфейс сокетов, позволяющий программам создавать сокеты для отдельных сетей и протоколов. Для использования сокетов пользовательские программы получают дескрипторы файлов.

Над дисковыми драйверами располагаются буферный кэш и страничный кэш файловой системы. В ранних системах UNIX буферный кэш представлял собой фиксированную область памяти, а остальная память использовалась для страниц пользователя. Во многих современных системах UNIX этой фиксированной границы уже не существует, и любая страница памяти может быть схвачена для выполнения любой задачи, в зависимости от того, что требуется в данный момент.

Над буферным кэшем располагаются файловые системы. Большинством систем UNIX поддерживаются несколько файловых систем, включая быструю файловую систему Беркли, журнальную файловую систему, а также различные виды файловых систем System V. Все эти файловые системы совместно используют общий буферный кэш. Выше файловых систем помещается именование файлов, управление каталогами, управление жесткими и символьными связями, а также другие свойства файловой системы, одинаковые для всех файловых систем.

Над страничным кэшем располагается система виртуальной памяти. В ней вся логика работы со страницами, например алгоритм замещения страниц. Поверх нее находится программа отображения файлов на виртуальную память и высокоуровневая программа управления страничными прерываниями. Эта программа решает, что нужно делать при возникновении страничного прерывания. Сначала она проверяет допустимость обращения к памяти и, если все в порядке, определяет местонахождение требуемой страницы и то, как она может быть получена.

Последний столбец имеет отношение к управлению процессами. Над диспетчером располагается планировщик процессов, выбирающий процесс, который должен быть запущен следующим. Если потоками управляет ядро, то управление потоками также помещается здесь, хотя в некоторых системах UNIX управление потоками вынесено в пространство пользователя. Над планировщиком расположена

программа для обработки сигналов и отправки их в требуемом направлении, а также программа, занимающаяся созданием и завершением процессов.

Верхний уровень представляет собой интерфейс системы. Слева располагается интерфейс системных вызовов. Все системные вызовы поступают сюда и направляются одному из модулей низших уровней в зависимости от природы системного вызова. Правая часть верхнего уровня представляет собой вход для аппаратных и эмулированных прерываний, включая сигналы, страничные прерывания, разнообразные исключительные ситуации процессора и прерывания ввода-вывода.

11.3. Процессы в системе UNIX

Единственными активными сущностями в системе UNIX являются процессы. Каждый процесс запускает одну программу и изначально получает один поток управления. То есть у процесса есть один счетчик команд, указывающий на следующую исполняемую команду процессора. Большинство версий UNIX позволяют процессу после того, как он запущен, создавать дополнительные потоки.

UNIX представляет собой многозадачную систему, так что несколько независимых процессов могут работать одновременно. У каждого пользователя может быть одновременно несколько активных процессов, так что в большой системе могут одновременно работать сотни и даже тысячи процессов. Часть из них работает в фоновом режиме (демоны). Типичным демоном является *cron daemon*. Он каждую минуту просыпается и проверяет, есть ли какие-то задачи. Если задачи есть, он их выполняет, а затем снова засыпает. Этот демон используется, в частности, для выполнения периодических заданий. Кроме того, он позволяет планировать в системе UNIX активность на минуты, часы, дни и даже месяцы вперед, а также управлять входящей и исходящей электронной почтой, очередями на принтер, проверять, достаточно ли еще осталось свободных страниц памяти и т. д.

Процессы в UNIX создаются при помощи системного вызова *fork*, который создает точную копию исходного (**родительского**) процесса – **дочерний процесс**. У родительского и дочернего процесса есть свои собственные образы памяти. Если родительский процесс впоследствии изменяет какие-либо свои переменные, изменения остаются невидимыми для дочернего процесса, и наоборот.

Файлы, открытые родительским процессом до выполнения системного вызова *fork*, будут видны и дочернему процессу. При этом все изменения,

произведенные с этим файлом, будут видны каждому процессу. Чтобы процессы различали, какой из них родительский, а какой – дочерний, системный вызов *fork* устанавливает каждому процессу PID (Process identifier – идентификатор процесса). Если процесс дочерний, то PID будет равен нулю, а если родительский, то – отличен от нуля.

Процессы распознаются по своим PID-идентификаторам. При создании процесса его PID выдается родителю нового процесса. Если дочерний процесс желает узнать свой PID, он может воспользоваться системным вызовом *getpid*. Идентификаторы процессов используются различным образом. Например, когда дочерний процесс завершается, его PID также выдается его родителю. Это может быть важно, так как у родительского процесса может быть много дочерних процессов. Поскольку у дочерних процессов также могут быть дочерние процессы, исходный процесс может создать целое дерево детей, внуков, правнуков и т. д.

В системе UNIX процессы могут общаться друг с другом с помощью разновидности обмена сообщениями. Можно создать канал между двумя процессами, в который один процесс может писать поток байтов, а другой процесс может его читать. Эти каналы иногда называют **трубами**. Синхронизация процессов достигается путем блокирования процесса при попытке прочитать данные из пустого канала. Когда данные появляются в канале, процесс разблокируется.

При помощи каналов организуются конвейеры оболочки. Когда оболочка видит строку вроде

sort <f | head

она создает два процесса, *sort* и *head*, а также устанавливает между ними канал таким образом, что стандартный поток вывода программы *sort* соединяется со стандартным потоком ввода программы *head*. При этом все данные, формируемые программой *sort*, попадают напрямую программе *head*, для чего не требуется временного файла. Если канал переполняется, система приостанавливает работу программы *sort*, пока программа *head* не удалит из него хоть сколько-нибудь данных.

Процессы также могут общаться другим способом: при помощи программных прерываний. Один процесс может послать другому так называемый **сигнал**. Процессы могут сообщить системе, какие действия следует предпринимать, когда придет сигнал. У процесса есть выбор: проигнорировать сигнал, перехватить его или позволить сигналу убить процесс (действие по умолчанию для большинства

сигналов). Если процесс выбрал перехват посылаемых ему сигналов, он должен указать процедуры обработки сигналов. Когда сигнал прибывает, управление внезапно передается обработчику. Когда процедура обработки сигнала завершает свою работу, управление снова возвращается в то место процесса, в котором оно находилось, когда пришел сигнал. Обработка сигналов аналогична обработке аппаратных прерываний ввода-вывода. Процесс может посылать сигналы только членам его **группы процессов**, состоящей из его прямого родителя, всех прародителей, братьев и сестер, а также детей (внуков и правнуков). Процесс может также послать сигнал сразу всей своей группе за один системный вызов.

Стандартом POSIX регламентируется всего 12 сигналов. В большинстве систем UNIX также имеются дополнительные сигналы, но программы, использующие их, могут оказаться непереносимыми на другие версии UNIX.

В первой версии системы UNIX не было потоков. Это свойство было добавлено много лет спустя. Изначально применялось множество различных пакетов поддержки потоков, однако распространение этих различных пакетов привело к тому, что написать переносимую программу стало очень сложно. В конце концов, системные вызовы, используемые для управления потоков, были стандартизированы в виде части стандарта POSIX (P1003.1c).

В стандарте POSIX не указывается, должны ли потоки реализовываться в пространстве ядра или в пространстве пользователя. Системные вызовы, определенные в стандарте P1003.1c, были тщательно отобраны так, чтобы потоки могли быть реализованы любым способом. До тех пор, пока пользовательские программы четко придерживаются семантики стандарта P1003.1c, оба способа реализации должны работать корректно. Когда используется системная реализация потоков, они являются настоящими системными вызовами. При использовании потоков на уровне пользователя они полностью реализуются в динамической библиотеке в пространстве пользователя.

Для синхронизации потоков одного процесса используются **мьютексы** (mutual exclusion – взаимное исключение), которые охраняют какой-либо ресурс, например буфер, совместно используемый двумя потоками. Чтобы гарантировать, что только один поток в каждый момент времени имеет доступ к общему ресурсу, потоки блокируют (захватывают) мьютекс перед обращением к ресурсу и разблокируют (отпускают) его, когда ресурс им больше не нужен. Это позволяет избежать состояния состязания.

Если необходима долговременная блокировка, то вместо мьютексов используются **переменные состояния**. В этом случае один поток ждет, когда второй поток изменит значение переменной на определенное, соответствующее тому, что совместно используемый ресурс свободен. Если ни один поток в этот момент не ждет, когда освободится ресурс, этот сигнал просто теряется. Другими словами, переменные состояния не считаются семафорами.

Рассмотрим, каким образом осуществляется реализация процессов в UNIX. У каждого процесса есть пользовательская часть, в которой работает программа пользователя. Однако когда один из потоков обращается к системному вызову, происходит эмулированное прерывание с переключением в режим ядра. После этого поток начинает работу в контексте ядра, с отличной картой памяти и полным доступом к ресурсам машины. Это все еще тот же самый поток, но более мощный, со своим стеком ядра и счетчиком команд в режиме ядра. Это важно, так как системный вызов может блокироваться на полпути, например, ожидая завершения дисковой операции. При этом счетчик команд и регистры будут сохранены таким образом, чтобы позднее поток можно было восстановить в режиме ядра.

Ядро поддерживает две ключевые структуры данных, относящихся к процессам: **таблицу процессов** и **структуру пользователя**. Таблица процессов является резидентной. В ней содержится информация, необходимая для всех процессов, даже для тех процессов, которых в данный момент нет в памяти. Структура пользователя выгружается на диск, освобождая место в памяти, когда относящийся к ней процесс отсутствует в памяти, чтобы не тратить память на ненужную в данный момент информацию. В таблице процессов содержится информация следующих категорий:

1. **Параметры планирования** – приоритеты процессов, процессорное время, потребленное за последний учитываемый период, количество времени, проведенное процессом в режиме ожидания.
2. **Образ памяти** – указатели на сегменты программы, данных и стека, или, если используется страничная организация памяти, то – указатели на соответствующие им таблицы страниц.
3. **Сигналы** – маски, указывающие, какие сигналы игнорируются, какие перехватываются, какие временно заблокированы, а какие находятся в процессе доставки.

4. **Разное** – текущее состояние процесса, события, ожидаемые процессом, время до истечения интервала будильника, PID процесса и родительского процесса, идентификаторы пользователя и группы.

В структуре пользователя содержится информация, которая не требуется, когда процесса физически нет в памяти, и он не выполняется. Например, хотя процессу, выгруженному на диск, можно послать сигнал, выгруженный процесс не может прочитать файл. По этой причине информация о сигналах должна храниться в таблице процессов, постоянно находящейся в памяти, даже когда процесс не присутствует в памяти. С другой стороны, сведения об описателях файлов могут храниться в структуре пользователя и загружаться в память вместе с процессом.

В структуре пользователя хранятся следующие данные:

1. **Машинные регистры** – когда происходит прерывание с переключением в режим ядра.

2. **Состояние системного вызова** – информация о текущем системном вызове, включая параметры и результаты.

3. **Таблица дескрипторов файлов** – когда происходит обращение к системному вызову, работающему с файлом, дескриптор файла используется в качестве индекса в данной таблице, что позволяет найти структуру данных (i-узел), соответствующую данному файлу.

4. **Учетная информация** – указатель на таблицу, учитывающую процессорное время, использованное процессом в пользовательском и системном режиме. В некоторых системах здесь также ограничивается процессорное время, которое может использовать процесс, максимальный размер стека, количество страниц памяти и т.д.

5. **Стек ядра** – фиксированный стек для использования процессом в режиме ядра.

Немного остановимся на алгоритме планировании в системе UNIX. Этот алгоритм имеет два уровня. Низкоуровневый алгоритм выбирает следующий процесс из набора процессов в памяти, готовых к работе. Высокоуровневый алгоритм перемещает процессы из памяти на диск и обратно, что предоставляет всем процессам возможность попасть в память и быть запущенными.

В низкоуровневом алгоритме используется несколько очередей. С каждой очередью связан диапазон непересекающихся значений приоритетов. Процессы, выполняющиеся в режиме пользователя, имеют положительные значения

приоритетов. У процессов, выполняющихся в режиме ядра (обращающихся к системным вызовам), значения приоритетов отрицательные. Отрицательные значения приоритетов считаются наивысшими, а положительные – минимальными. В очередях располагаются только процессы, находящиеся в памяти и готовые к работе.

Когда запускается низкоуровневый планировщик, он ищет очередь, начиная с самого высокого приоритета (то есть с наименьшего отрицательного значения), пока не находит очередь, в которой есть хотя бы один процесс. После этого в этой очереди выбирается и запускается первый процесс. Ему разрешается работать в течение максимального кванта времени, как правило, 100 мс, или пока он не заблокируется. Если процесс использует весь свой квант времени, он помещается обратно, в конец очереди, а алгоритм планирования запускается снова. Таким образом, процессы, входящие в одну группу приоритетов, совместно используют центральный процессор в порядке циклической очереди. Раз в секунду приоритет каждого процесса пересчитывается по новой.

11.4. Управление памятью в UNIX

Модель памяти, используемая в системе UNIX, довольно проста, что должно обеспечить переносимость программ, а также реализацию ОС UNIX на машинах с сильно отличающимися модулями памяти – от элементарных до сложного оборудования со страничной организацией.

У каждого процесса в системе UNIX есть адресное пространство, состоящее из трех сегментов: текста (программы), данных и стека. **Текстовый (программный) сегмент** содержит машинные команды, образующие исполняемый код программы. Как правило, он доступен только для чтения. Следовательно, он не изменяется ни в размерах, ни по своему содержанию.

Сегмент данных содержит переменные, строки, массивы и другие данные программы. Он состоит из двух частей: инициализированных данных и неинициализированных данных (BSS). Инициализированная часть сегмента данных содержит переменные и константы компилятора, значения которых должны быть заданы при запуске программы. Неинициализированные данные необходимы лишь с точки зрения оптимизации. Когда начальное значение глобальной переменной явно не указано, то, согласно семантике языка С, ее значение устанавливается равным 0. На практике большинство глобальных переменных не

инициализируются, и, таким образом, их начальное значение равно 0. Это можно реализовать следующим образом: создать целый сегмент исполняемого двоичного файла, точно равного по размеру числу байтов данных, и проинициализировать весь этот сегмент нулями.

Однако из экономии места на диске этого не делается. Файл содержит только те переменные, начальные значения которых явно заданы. Вместо неинициализированных переменных компилятор помещает в исполняемый файл просто одно слово, содержащее размер области неинициализированных данных в байтах. При запуске программы операционная система считывает это слово, выделяет нужное число байтов и обнуляет их.

В отличие от текстового сегмента, который не может изменяться, сегмент данных может модифицироваться. Программы изменяют свои переменные постоянно. Более того, многим программам требуется выделение дополнительной памяти динамически, во время выполнения. Чтобы реализовать это, операционная система UNIX разрешает сегменту данных расти при динамическом выделении памяти программам и уменьшаться при освобождении памяти программами. Программа может установить размер своего сегмента данных с помощью системного вызова *brk*. Таким образом, чтобы получить больше памяти, программа может увеличить размер своего сегмента данных. Этим системным вызовом пользуется библиотечная процедура *malloc*, используемая для выделения памяти.

Сегмент стека на большинстве компьютеров начинается около старших адресов виртуального адресного пространства и растет вниз к нулю. Если указатель стека оказывается ниже нижней границы сегмента стека, то происходит аппаратное прерывание, при котором ОС понижает границу сегмента стека на одну страницу памяти. Программы не управляют явно размером сегмента стека.

Когда программа запускается, ее стек не пуст. Напротив, он содержит все переменные окружения (оболочки), а также командную строку, введенную в оболочке при вызове этой программы. Таким образом, программа может узнать параметры, с которыми она была запущена.

Когда два пользователя запускают одну и ту же программу, в памяти можно хранить две копии программы, но это неэффективно. Вместо этого большинством систем UNIX поддерживаются **текстовые сегменты совместного использования**. Отображение осуществляется аппаратным обеспечением виртуальной памяти.

Сегменты данных и стека никогда не бывают общими. Если размер любого из сегментов должен быть увеличен, то отсутствие свободного места в соседних страницах памяти не является проблемой, так как соседние виртуальные страницы памяти не обязаны отображаться па соседние физические страницы.

Если на компьютере поддерживаются раздельные адресные пространства для команд и для данных, то система UNIX может этим воспользоваться. Это позволяет удвоить доступное адресное пространство.

Многими версиями UNIX поддерживается **отображение файлов на адресное пространство памяти**. Это свойство позволяет отображать файл на часть адресного пространства процесса, так чтобы можно было читать из файла и писать в файл, как если бы это был массив, хранящийся в памяти. Отображение файла на адресное пространство памяти делает произвольный доступ к нему существенно более легким, чем при использовании системных вызовов, таких как *read* и *write*. Совместный доступ к библиотекам предоставляется именно при помощи этого механизма. На рисунке 11.3 показан файл, одновременно отображенный на адресные пространства двух процессов по различным виртуальным адресам.

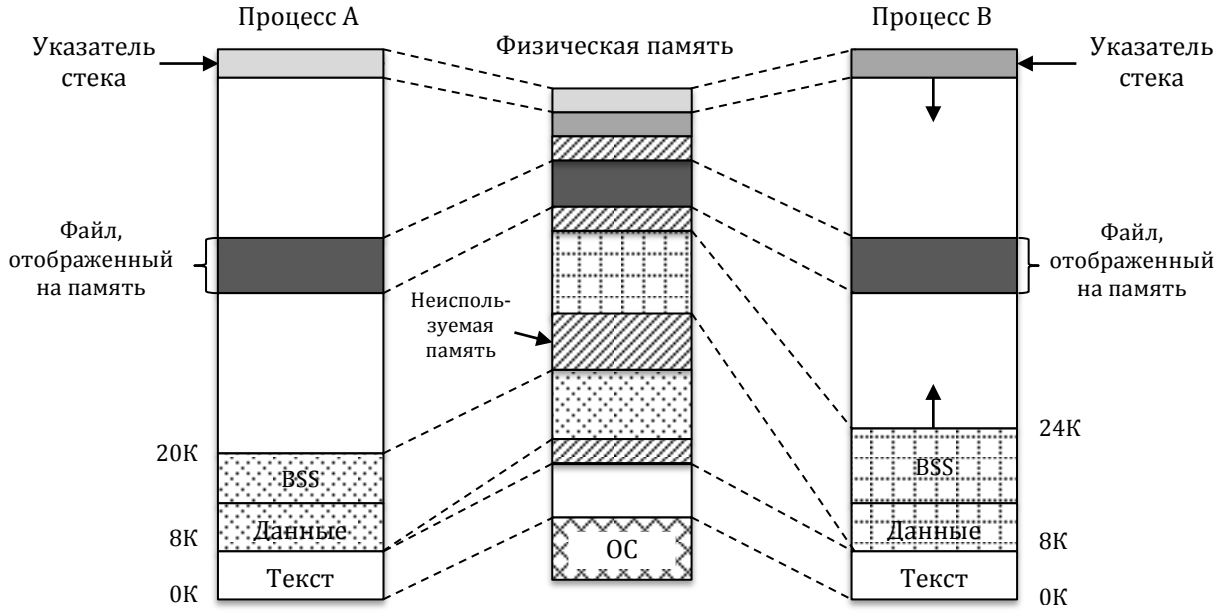


Рисунок 11.3 – Два процесса совместно используют один отображенный на память файл

Дополнительное преимущество отображения файла на память заключается в том, что два или более процессов могут отобразить на свое адресное пространство один и тот же файл. При этом запись в этот файл одним из процессов мгновенно становится видимой всем остальным. Таким образом, отображение на адресное

пространство памяти временного файла (который будет удален после завершения работы процессов) представляет собой механизм реализации общей памяти для нескольких процессов, причем у такого механизма будет высокая пропускная способность.

Немного остановимся на реализации управления памятью в UNIX. До версии 3BSD большинство систем UNIX основывалось на свопинге (подкачке). Когда загружалось больше процессов, чем могло поместиться в памяти, некоторые из них выгружались на диск. Выгружаемый процесс всегда выгружался на диск целиком (исключение составляли только совместно используемые текстовые сегменты). Таким образом, процесс мог быть либо в памяти, либо на диске. Перемещением данных между памятью и диском управлял верхний уровень двухуровневого планировщика, называвшийся **свопером** (swapper). Выгрузка данных из памяти на диск инициировалась, когда у ядра, кончалась свободная память.

Кроме того, когда наступало время запустить процесс, уже достаточно долго находящийся на диске, часто бывало необходимо удалить из памяти другой процесс, чтобы освободить место для запускаемого процесса.

Выбирая жертву, свопер сначала рассматривал блокированные (например, ожиданием ввода с терминала) процессы. Лучше удалить из памяти процесс, который не может работать, чем работоспособный процесс. Если такие процессы находились, из них выбирался процесс с наивысшим значением суммы приоритета и времени пребывания в памяти. Таким образом, хорошими кандидатами на выгрузку были процессы, потребившие большое количество процессорного времени, либо находящиеся в памяти уже достаточно долгое время, даже если большую его часть они занимались вводом выводом. Если блокированных процессов не было, тогда на основе тех же критериев выбирался готовый процесс.

Каждые несколько секунд свопер исследовал список выгруженных процессов, проверяя, не готов ли какой-либо из этих процессов к работе. Если процессы в состоянии готовности обнаруживались, из них выбирался процесс, дольше всех находящийся на диске. Затем свопер проверял, будет ли это легкий свопинг или тяжелый. **Легким свопингом** считался тот, для которого не требовалось дополнительное высвобождение памяти. При этом нужно было всего лишь загрузить выгруженный на диск процесс. **Тяжелым свопингом** назывался свопинг, при котором для загрузки в память выгруженного на диск процесса из нее требовалось удалить один или несколько других процессов.

Начиная с версии 3BSD, стало возможным использовать страничную подкачку. Идея проста: чтобы работать, процессу не нужно целиком находиться в памяти. Все, что в действительности требуется, – это структура пользователя и таблицы страниц. Если они загружены, то процесс считается находящимся в памяти и может быть запущен планировщиком. Страницы с сегментами текста, данных и стека загружаются в память динамически, по мере обращения к ним. Если пользовательской структуры и таблицы страниц нет в памяти, то процесс не может быть запущен, пока свопер не загрузит их.

Алгоритм замещения страниц выполняется страничным демоном. Раз в 250 мс он просыпается, чтобы сравнить количество свободных страничных блоков с системным параметром *lotsfree* (равным, как правило, 1/4 объема памяти). Если число свободных страничных блоков меньше, чем значение этого параметра, страничный демон начинает переносить страницы из памяти на диск, пока количество свободных страничных блоков не станет равно *lotsfree*. Если же количество свободных страничных блоков больше или равно *lotsfree*, тогда страничный демон, ничего не предпринимая, отправляется спать дальше. Если у компьютера много памяти и мало активных процессов, страничный демон спит практически все время.

Страничный демон использует модифицированную версию алгоритма часов. Это глобальный алгоритм, то есть при удалении страницы он не учитывает, чья это страница. Таким образом, количество страниц, выделяемых каждому процессу, меняется со временем.

Основной алгоритм часов работает, сканируя в цикле страничные блоки (как если бы они лежали на окружности циферблата часов). На первом проходе, когда стрелка часов указывает на страничный блок, сбрасывается его бит использования. На втором проходе у каждого страничного блока, к которому не было доступа с момента первого прохода, бит использования останется сброшенным, и этот страничный блок будет помещен в список свободных страниц (после записи его на диск, если он "грязный"). Страничный блок в списке свободных страниц сохраняет свое содержание, что позволяет восстановить страницу, если она потребуется прежде, чем будет перезаписана.

Изначально в UNIX использовался основной алгоритм часов, но затем было обнаружено, что при больших объемах оперативной памяти проходы занимают слишком много времени. Тогда алгоритм был заменен алгоритмом часов с двумя

стрелками. В этом алгоритме страничный демон поддерживает два указателя на карту памяти. При работе он сначала очищает бит использования передней стрелкой, а затем проверяет этот бит задней стрелкой, после чего перемещает обе стрелки. Если две стрелки находятся близко друг от друга, то только у очень активно используемых страниц появляется шанс, что к ним будет обращение между проходами двух стрелок.

При каждом запуске страничного демона стрелки проходят не полный оборот, а столько, сколько необходимо, чтобы количество страниц в списке свободных страниц было не менее *lotsfree*.

Если операционная система обнаруживает, что частота подкачки страниц слишком высока, а количество свободных страниц все время ниже *lotsfree*, свопер начинает удалять из памяти один или несколько процессов, чтобы остановить состязание за свободные страничные блоки.

11.5. Управление памятью в Linux

Каждый процесс системы Linux на 32-разрядной машине получает 3 Гбайт виртуального адресного пространства для себя с оставшимся 1 Гбайт памяти для страничных таблиц и других данных ядра. Один гигабайт ядра не виден в пользовательском режиме, но становится доступным, когда процесс переключается в режим ядра. Адресное пространство создается при создании процесса и перезаписывается системным вызовом *exec*.

Виртуальное адресное пространство делится на однородные непрерывные области, выровненные по границам страниц. Таким образом, каждая область состоит из набора соседних страниц с одинаковым режимом защиты и одинаковыми свойствами подкачки. Примерами областей являются текстовый сегмент и файлы, отображенные на память. Между областями в виртуальном адресном пространстве могут быть свободные участки. Любое обращение процесса к памяти в этих свободных участках приводит к фатальному страничному прерыванию. Размер страницы фиксирован на уровне 4 Кбайт.

Каждая область описывается в ядре записью *vm_area_struct*. Все структуры *vm_area_struct* одного процесса связаны вместе в список, отсортированный по виртуальным адресам, что позволяет быстро находить все страницы. Когда список становится слишком длинным (более 32 записей), создается дерево для ускорения поиска. Запись *vm_area_struct* перечисляет свойства области. К ним относятся режим

защиты (например, только чтение или чтение/запись), является ли данная область фиксированной в памяти (невыгружаемой), и направление, в котором область может расти (вверх для сегментов данных, вниз для сегмента стека).

Структура *vm_area_struct* также содержит данные о том, является ли данная область приватной областью процесса или ее совместно используют несколько процессов. После системного вызова *fork* система Linux создает копию списка областей для дочернего процесса, но у дочернего и родительского процессов оказываются указатели на одни и те же таблицы страниц. Области помечаются как доступные для чтения/записи, но страницы доступны только для чтения. Если любой из процессов пытается записать данные в такую страницу, происходит прерывание, ядро видит, что область логически доступна для записи, а страница недоступна, поэтому оно дает процессу копию страницы, которую помечает как доступную для чтения/записи. Таким образом реализован механизм копирования при записи.

Кроме того, в структуре *vm_area_struct* записано, есть ли у этой области памяти место хранения на диске, и если да, то где оно расположено. Текстовые сегменты в качестве резервного хранения используют двоичные файлы, а отображаемые на адресное пространство памяти файлы выгружаются на диск в соответствующие им файлы. Всем остальным областям, таким как область стека, не назначаются области резервного хранения, пока не потребуется их выгрузка на диск.

В системе Linux используется трехуровневая схема страничной подкачки. Каждый виртуальный адрес разбивается на четыре поля, как показано на рисунке 11.4.

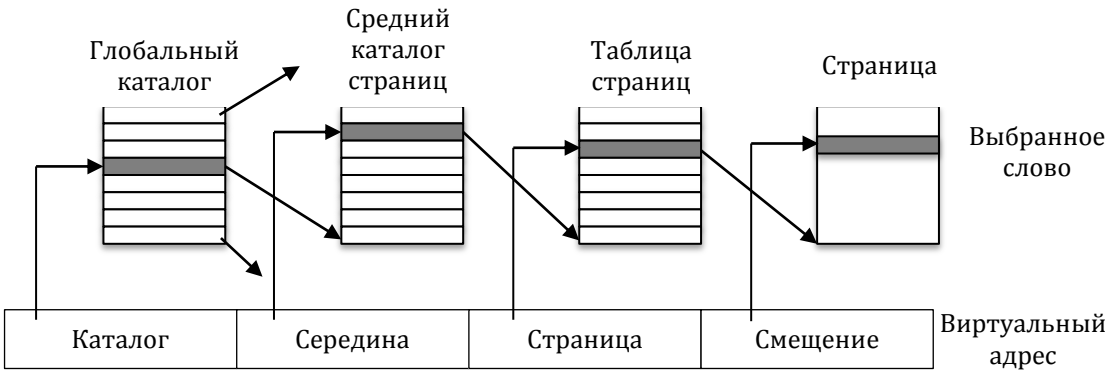


Рисунок 11.4 – Трехуровневые таблицы страниц Linux

Каталоговое поле используется как индекс в глобальном каталоге, в котором есть личный каталог для каждого процесса. Содержание элемента каталога является указателем на одну из средних страничных таблиц, которые тоже проиндексированы полем виртуального адреса. Наконец, элемент средней таблицы указывает на таблицу страниц, также проиндексированную полем страницы виртуального адреса. Элемент в последней таблице содержит указатель на нужную страницу.

Физическая память используется для различных целей. Само ядро жестко фиксировано. Ни одна его часть не выгружается на диск. Остальная часть памяти доступна для страниц пользователей, буферного кэша, используемого файловой системой, страничного кэша и других задач. Буферный кэш содержит блоки файлов, которые были недавно считаны или были считаны заранее в надежде на то, что они скоро могут понадобиться. Его размер динамически меняется. Буферный кэш состязается за место в памяти со страницами пользователей. Страничный кэш в действительности не является настоящим отдельным кэшем, а представляет собой просто набор страниц пользователя, которые более не нужны и ожидают выгрузки на диск. Если страница, находящаяся в страничном кэше, потребуется снова, прежде чем она будет удалена из памяти, ее можно быстро объявить находящейся в памяти.

Кроме этого, операционная система Linux поддерживает динамически загружаемые модули, в основном драйверы устройств. Они могут быть произвольного размера и каждому из них должен быть выделен непрерывный участок в памяти ядра. Для выполнения всех этих требований система Linux управляет памятью таким образом, что она может получить по желанию участок памяти произвольного размера. Для этого используется алгоритм, известный как **"дружественный" алгоритм**.

Основная идея управления блоками памяти заключается в следующем. Изначально память состоит из единого непрерывного участка. В нашем примере на рисунке 11.5, *а* размер этого участка равен 64 страницам. Когда поступает запрос на выделение памяти, он сначала округляется до степени двух, например до 8 страниц. Затем весь блок памяти делится пополам (рисунок 11.5, *б*). Так как получившиеся в результате этого деления надвое участки памяти все еще слишком велики, нижняя половинка делится пополам еще (рисунок 11.5, *в*) и еще (рисунок 11.5, *г*). Теперь мы получили участок памяти нужного размера. Этот участок предоставляется обратившемуся процессу (затененный на рисунке 11.5, *г*).

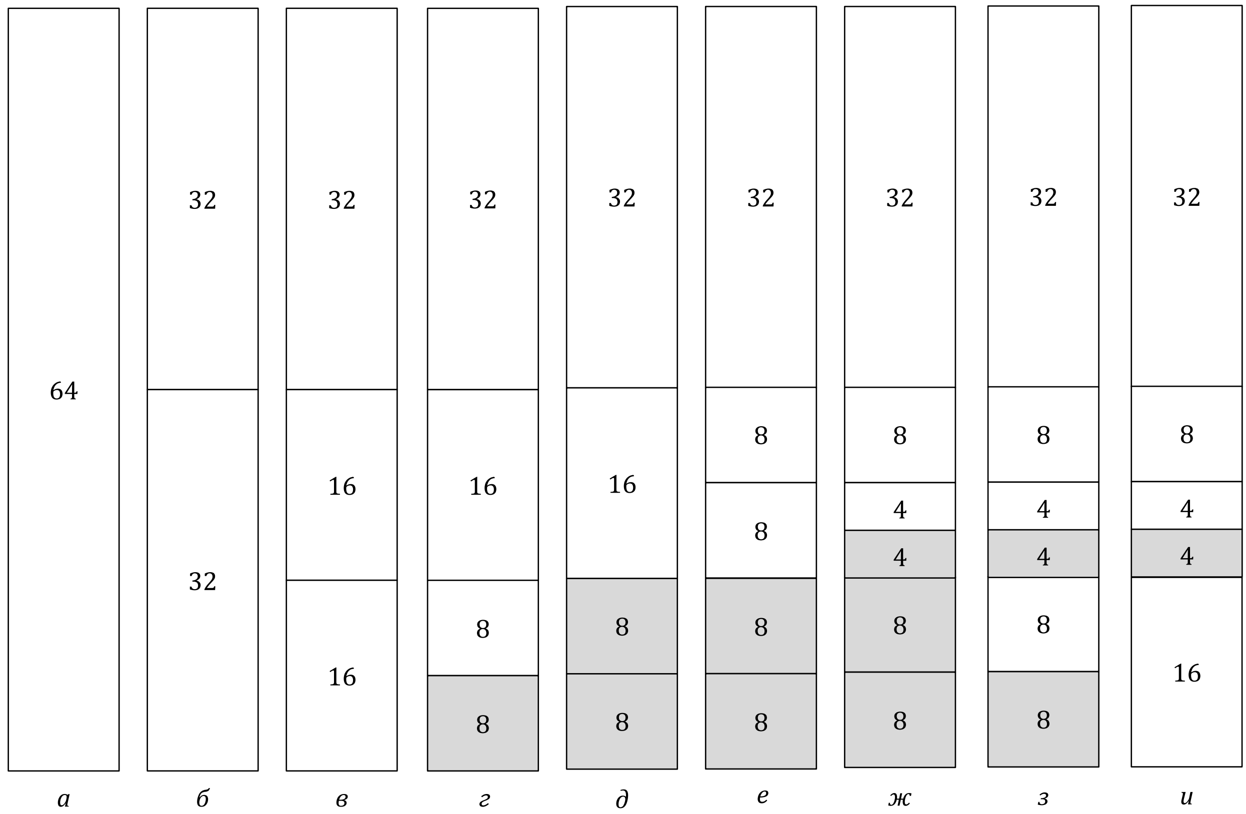


Рисунок 11.5 – Этапы работы дружественного алгоритма

Теперь предположим, что приходит второй запрос на 8 страниц. Он может быть удовлетворен немедленно (рисунок 11.5, *д*). Следом поступает запрос на 4 страницы. При этом делится надвое наименьший участок (рисунок 11.5, *е*) и выделяется половина от половины (рисунок 11.5, *ж*). Затем освобождается второй 8-страничный участок (рисунок 11.5, *з*). Наконец освобождается оставшийся 8-страничный участок. Поскольку эти два участка были "приятелями", то есть они вышли из одного 16-страничного блока, они снова объединяются в 16-страничный блок (рисунок 11.5, *и*).

Операционная система Linux управляет памятью при помощи данного алгоритма. К нему добавляется массив, в котором первый элемент представляет собой начало списка блоков размером в 1 единицу, второй элемент является началом списка блоков размером в 2 единицы, третий элемент – началом списка блоков размером в 4 единицы и т.д. Таким образом, можно быстро найти любой блок с размером, кратным степени 2.

Этот алгоритм приводит к существенной внутренней фрагментации, так как, если вам нужен 65-страничный участок, вы получите 128-страничный блок.

Чтобы как-то решить эту проблему, в системе Linux есть второй алгоритм выделения памяти, выбирающий блоки памяти при помощи "приятельского" алгоритма, а затем нарезающий из этих блоков более мелкие фрагменты и управляющий этими фрагментами отдельно. Кроме того, существует третий алгоритм выделения памяти, использующийся, когда выделяемая память должна быть непрерывна только в виртуальном адресном пространстве, но не в физической памяти. Все эти алгоритмы выделения памяти были взяты из системы UNIX System V.

Операционная система Linux является системой, предоставляющей страницы по требованию, без предварительной загрузки страниц и без концепции рабочего набора (хотя в ней есть системный вызов для указания пользователем страницы, которая ему может скоро понадобиться). Текстовые сегменты и отображаемые на адресное пространство памяти файлы подгружаются из соответствующих им файлов на диске. Все остальное выгружается либо в область подкачки, если она присутствует, либо в файлы подкачки фиксированной длины, которых может быть от одного до восьми. Файлы подкачки могут динамически добавляться и удаляться, и у каждого есть свой приоритет. Выгрузка страниц в отдельный раздел диска, доступ к которому осуществляется как к отдельному устройству, не содержащему файловой системы, более эффективна, чем выгрузка в файл, по нескольким причинам. Во-первых, не требуется преобразование блоков файла в блоки диска. Во-вторых, физическая запись может быть любого размера, а не только размера блока файла. В-третьих, страница всегда пишется прямо на устройство в виде единого непрерывного участка, а при записи в файл подкачки это может быть и не всегда так.

Страницы на устройстве подкачки или дисковом разделе подкачки не выделяются, пока они не потребуются. Каждое устройство или файл подкачки начинается с битового массива, в котором сообщается, какие страницы свободны. Когда страница, у которой нет места хранения на диске, должна быть удалена из памяти, из файлов (или разделов) подкачки, в которых еще есть свободное место, выбирается файл с наивысшим приоритетом, и в нем выделяется место для страницы. Как правило, у раздела подкачки (если таковой имеется) более высокий приоритет, чем у любого файла подкачки. Координата страницы на диске записывается в таблицу страниц.

Алгоритм замещения страниц работает следующим образом. Система Linux пытается поддерживать некоторые страницы свободными, чтобы их можно было предоставить при необходимости. Конечно, этот пул страниц должен постоянно пополняться, поэтому реальный алгоритм страничной подкачки заключается в том, как это происходит. Во время загрузки процесс *init* запускает страничный демон *kswapd*, который работает раз в секунду. Он проверяет, есть ли достаточное количество свободных страниц. Если да, он отправляется спать еще секунду, хотя он может быть разбужен и раньше, если внезапно понадобятся дополнительные страницы. Страничный демон состоит из цикла, который выполняется до шести раз с возрастающей срочностью. Почему шесть? Вероятно, автор программы думал, что четырех будет недостаточно, а восемь будет слишком много. В отдельных местах операционная система Linux реализована именно так.

Тело цикла выполняет обращения к трем процедурам, каждая из которых пытается получить различные типы страниц. Значение срочности передается в виде параметра, сообщающего процедуре, сколько усилий требуется предпринять, чтобы получить некоторые страницы. Как правило, это означает, сколько страниц нужно проверить, прежде чем опустить руки. В результате этот алгоритм сначала выбирает легко доступные страницы каждой категории, после чего переходит к труднодоступным. Когда получено достаточное количество страниц, страничный демон снова отправляется спать.

Первая процедура пытается получить те страницы из страничного кэша и буферного кэша файловой системы, к которым в последнее время не было обращений, для чего используется алгоритм часов. Вторая процедура ищет совместно используемые страницы, которыми никто из пользователей, похоже, не пользуется активно. Третья процедура, пытающаяся получить страницы, используемые одиночными пользователями, является наиболее интересной, поэтому рассмотрим ее подробнее.

Сначала выполняется цикл по всем процессам, в котором определяется, у какого процесса больше всего страниц на данный момент находится в памяти. Как только такой процесс найден, сканируются все его структуры *vm_area_struct* и изучаются все страницы в порядке виртуальных адресов, начиная с того места, на котором этот процесс был отложен в прошлый раз. Если страница недействительна, отсутствует в памяти, используется совместно, фиксирована в памяти или используется для DMA, то она пропускается. Если у страницы установлен бит

обращения к ней, этот бит сбрасывается и страница отправляется в резерв. Если же бит сброшен, эта страница отнимается у процесса. В результате данный алгоритм подобен алгоритму часов (с той разницей, что страницы не сканируются в порядке FIFO).

Если страница, выбранная для удаления из памяти, чистая, она удаляется немедленно. Если страница "грязная" и у нее есть место резервного хранения на диске, она устанавливается в очередь записи на диск. Наконец, если у "грязной" страницы нет места резервного хранения на диске, она отправляется в страничный кэш, из которого она может быть снова получена позднее, если обращение к ней поступит прежде, чем она будет фактически выгружена на диск. В основе идеи сканирования страниц в порядке виртуальных адресов лежит надежда на то, что страницы, расположенные близко друг к другу в виртуальном адресном пространстве, скорее всего, будут использоваться или не использоваться вместе, как единая группа, поэтому их следует записывать на диск как группу, а затем вместе считывать в память.

В управлении памятью принимает участие еще один демон, *bdflush*. Он периодически просыпается (а в некоторых случаях его явно будят), чтобы проверить, не превысило ли количество "грязных" страниц определенного предельного уровня. Если превысило, демон начинает сохранять их на диске.

11.6. Ввод и вывод в системе UNIX

Система ввода-вывода в UNIX основана на том, что все устройства ввода-вывода в ней выглядят как файлы. Доступ к ним осуществляется с помощью тех же системных вызовов *read* и *write*, которые используются для доступа к обычным файлам. Если есть необходимость задать параметры устройства, то для этого используется специальный системный вызов.

В UNIX каждому устройству ввода-вывода назначается имя пути в каталоге */dev*. Например, диск может иметь путь */dev/hd1*, у принтера может быть путь */dev/lp*, а у сети – */dev/net*. Чтобы вывести файл на принтер может быть использована команда

```
cp file /dev/lp
```

Команда *cp* даже не знает, что она работает с принтером. Таким образом, для выполнения ввода-вывода не требуется специального механизма.

Для взаимодействия с драйверами устройств используются специальные файлы, которые могут быть двух категорий – блочные и символьные. Блочные позволяют осуществлять произвольный доступ к нужному блоку, а символьные – только последовательно.

У каждого драйвера есть **номер старшего устройства**, который идентифицирует, каким устройством он может управлять. Если таких устройств несколько (например, несколько жестких дисков), то каждому из них присваивается **номер младшего устройства**. Вместе номера младшего и старшего устройств однозначно идентифицируют устройство ввода-вывода. В некоторых случаях один драйвер может управлять двумя связанными устройствами. Так, драйвер, соответствующий символьному специальному файлу */dev/tty*, управляет и клавиатурой, и экраном, которые воспринимаются как терминал.

Рассмотрим, как в UNIX осуществляется работа с сетью. Ключевым в этом процессе является понятие **сокета**. Они образуют пользовательский интерфейс с сетью, как почтовые ящики образуют интерфейс с почтовой системой, а телефонные розетки позволяют подключить телефон к телефонной системе.

Сокеты могут динамически создаваться и разрушаться. Сокеты могут быть трех основных типов:

1. Надежный, ориентированный на соединение байтовый поток. Он позволяет двум процессам на разных машинах установить между собой прямой канал. В этот канал байты подаются в определенном порядке и так же выходят из него с другой стороны. Эта система гарантирует, что все посланные байты прибудут на другой конец канала и именно в том порядке, как были отправлены.

2. Надежный, ориентированный на соединение поток пакетов. Он сохраняет границы между пакетами. Например, если отправитель посылает пять раз системный вызов *write*, то в первом варианте получатель получит сразу все пять вызовов, а во втором он должен пять раз сгенерировать системный вызов *read*.

3. Ненадежная передача пакетов. Этот тип полезен для приложений реального времени и для обработки ошибок. При этой схеме передача никак не гарантируется. Пакеты могут теряться, приходить в неверном порядке. Но в случае, когда больше ценится производительность, чем надежность (например, для доставки мультимедиа), эта схема имеет преимущества.

Рассмотрим, каким образом происходит реализация ввода-вывода в операционной системе UNIX.

Каждый драйвер разделен на несколько частей. Верхняя часть драйвера работает в режиме вызывающего процесса и служит интерфейсом с остальной системой UNIX. Нижняя часть работает в контексте ядра и взаимодействует с устройством. Драйверам разрешается обращаться к процедурам ядра для выделения памяти, управления таймером, управления DMA и т.д.

Цель той части системы, которая занимается операциями ввода-вывода с блочными специальными файлами, заключается в минимизации количества операций переноса данных. Для достижения данной цели в системах UNIX между дисковыми драйверами и файловой системой помещается **буферный кэш**. Он представляет собой таблицу в ядре, в которой хранятся тысячи недавно использованных блоков. Если нужный блок есть в кэше, то он получается оттуда, при этом обращения к диску удается избежать.

Так как символьные специальные файлы имеют дело с символьными потоками, а не перемещают блоки данных между памятью и диском, они не пользуются буферным кэшем. Вместо этого в первых версиях UNIX каждый драйвер символьного устройства выполнял всю работу, требуемую для данного устройства. Однако позднее этот механизм был изменен, так как выяснилось, что многие драйверы дублировали друг друга.

В системе BSD было применено решение, основанное на структурах данных, называемых С-списками. Каждый С-список представляет собой блок размером до 64 символов плюс счетчик и указатель на следующий блок. Символы, поступающие с терминала или любого другого символьного устройства, буферизуются в цепочках таких блоков.

При этом данные не передаются напрямую между драйвером и процессом. Между ними существуют процедуры, расположенные в ядре и называемые **дисциплинами линии связи**. Они работают как фильтр, обрабатывая поток данных и формируя **обработанный символьный поток**.

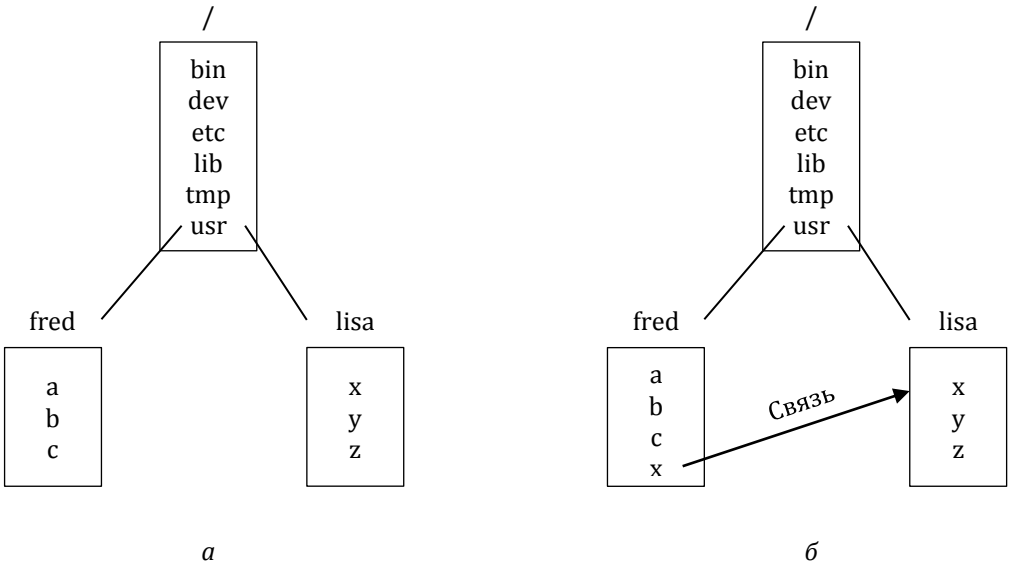
В системе System V было применено другое решение – **потоки данных**. В потоке всегда есть голова и соединение с драйвером. В промежутке – произвольно изменяемое число модулей, которые обрабатывают данные как буфер потока и передают их дальше.

11.7. Файловая система в UNIX

Файлы в системе UNIX представляют собой последовательность байтов произвольной длины, содержащую произвольную информацию. Значение битов в файле целиком определяется владельцем файла. Изначально в UNIX было ограничение длины имен файлов в 14 символов, в последующем этот предел был расширен до 255 символов.

Пользователям часто бывает необходимо обратиться к файлам, принадлежащим другим пользователям, или к своим файлам, расположенным в другом месте дерева файлов. Например, если два пользователя совместно используют один файл, он будет находиться в каталоге, принадлежащем одному из них, поэтому другому пользователю понадобится для обращения к этому файлу использовать абсолютное имя пути (или менять свой рабочий каталог). Если абсолютный путь достаточно длинен, то необходимость вводить его каждый раз может весьма сильно раздражать. В системе UNIX эта проблема решается при помощи так называемых **связей**, представляющих собой записи каталога, указывающие на другие файлы.

В качестве примера рассмотрим ситуацию на рисунке 11.6, *а*. Фред и Лиза вместе работают над одним проектом, и каждому из них нужен доступ к файлам другого. Если рабочий каталог Фреда */usr/fred*, он может обращаться к файлу *x* в каталоге Лизы как */usr/lisa/x*. Однако Фред может также создать новую запись в своем каталоге (рисунок 11.6, *б*), после чего он сможет обращаться к этому файлу просто как к *x*.



а *б*

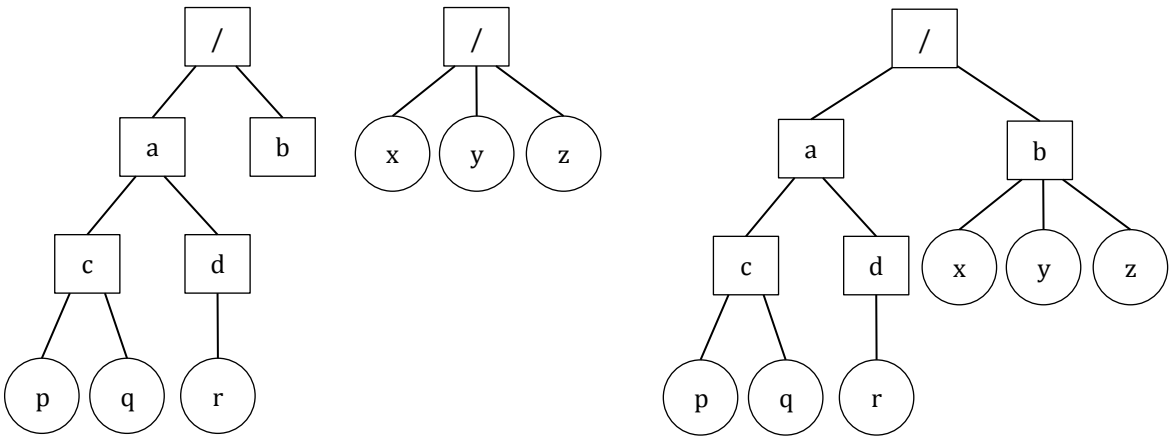
Рисунок 11.6 – До создания связи (*а*); после создания связи (*б*)

Одной из особенностей файловой системы в UNIX является то, что в ней существует возможность монтировать один диск в дерево каталогов другого диска.

На многих компьютерах установлено по два и более дисков. Например, на мэйнфреймах в банках часто бывает необходимо подключать по 100 и более дисков к одной машине, чтобы хранить большие базы данных. Даже у персональных компьютеров часто есть, по меньшей мере, два диска. При наличии у компьютера нескольких дисков возникает вопрос, как ими управлять.

Одно из решений заключается в том, чтобы установить самостоятельную файловую систему на каждый отдельный диск и управлять ими как отдельными файловыми системами. Рассмотрим, например, ситуацию, изображенную на рисунке 11.7, *а*. Здесь показаны 2 жестких диска, которые мы будем называть *C*: и *E*:, У каждого есть собственный корневой каталог и файлы. При таком решении пользователь должен помимо каталогов указывать также и устройство, если оно отличается от используемого по умолчанию. Например, чтобы скопировать файл *x* в каталог *d* (предполагая, что по умолчанию выбирается диск *C*:), следует ввести команду

```
cp E:/x /a/d/x
```



а *б*

Рисунок 11.7 – Раздельные файловые системы (*а*); после монтирования (*б*)

Такой подход применяется в операционных системах MS-DOS, и Windows. Решение, применяемое в операционной системе UNIX, заключается в том, чтобы позволить монтировать один диск в дерево файлов другого диска. В нашем примере мы можем смонтировать диск *E*: в каталог */b*, получая в результате файловую систему, показанную на рисунке 11.7, *б*. Теперь пользователь видит единое дерево

файлов и уже не должен думать о том, какой файл на каком устройстве хранится. В результате приведенная выше команда примет вид

cp /b/x /a/d/x

То есть все будет выглядеть так, как если бы файл копировался из одного каталога жесткого диска в другой каталог того же диска.

Другое интересное свойство – это **блокировка**. В некоторых приложениях несколько процессов могут одновременно использовать один и тот же файл, что может приводить к конфликту. Одно из решений – это создание в приложении критической области. Однако если эти процессы принадлежат независимым пользователям, то в результате будет недоступен весь файл или каталог. Это не всегда удобно. Блокировка позволяет более гибко подходить к решению этой проблемы. Она позволяет за одну неделимую операцию блокировать от целого файла до одного единственного байта. Механизм блокировки требует от вызывающего его процесса указать блокируемый файл, начальный байт и количество байтов. Если операция завершается успешно, система создает запись в таблице, в которой указывается, что определенные байты файла заблокированы.

Существует два типа блокировки: **с монополизацией** и **без монополизации**. Если часть файла уже содержит блокировку без монополизации, то повторная установка блокировки без монополизации на это место файла разрешается, но попытка установки блокировку с монополизацией будет отвергнута. Если же какая-либо область файла содержит блокировку с монополизацией, то любые попытки заблокировать любую часть этой области файла будут отвергаться, пока не будет снята монопольная блокировка. Для успешной установки блокировки необходимо, чтобы каждый байт в данной области был доступен.

При установке блокировки процесс должен указать, хочет ли он сразу получить управление или будет ждать, пока не будет снята предыдущая блокировка. Если он хочет воспользоваться содержимым файла без ожидания, то он немедленно получает ответ об успехе или неудаче операции. В противном случае – он будет ожидать снятия блокировки.

Рассмотрим реализацию файловой системы в UNIX. В традиционной файловой системе (то есть V7) раздел диска начинался с блока 0, который не используется системой и часто содержит программу загрузки компьютера. Затем следовал блок 1, так называемый **супер-блок** (рисунок 11.8). В нем хранилась критическая информация о размещении файловой системы – количество i-узлов, дисковых

блоков, а также начало списка свободных блоков диска. Если будет поврежден супер-блок, то файловая система окажется нечитаемой. Далее следовали i-узлы, за ними – блоки с данными. Здесь хранились все файлы и каталоги. Если файл состоял более, чем из одного блока, то блоки могли быть расположены не подряд. В действительности блоки большого файла, как правило, оказывались разбросанными по всему диску. Именно эту проблему должны были решить усовершенствования версии Berkeley.

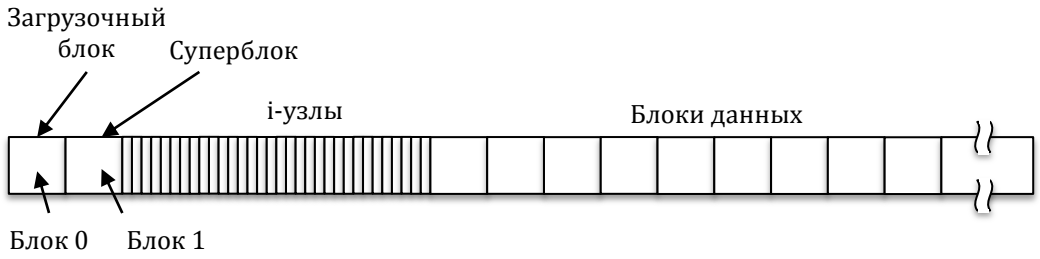


Рисунок 11.8 – Расположение классической файловой системы UNIX на диске

Каталог в традиционной файловой системе представлял собой несортированный набор 16-байтовых записей. Каждая запись состояла из 14-байтного имени файла и номера i-узла. Чтобы открыть файл в рабочем каталоге, система просто считывала каталог, сравнивая имя искомого файла с каждой записью, пока не найдет нужную запись или пока не закончится каталог.

Следующее поколение файловой системы UNIX – Berkeley Fast File System – позволило реорганизовать каталоги и реализовать увеличение имени файлов с 14 до 255 символов. Конечно, изменение структуры всех каталогов означало, что старые программы более не работали. Для обеспечения совместимости двух систем в системе Berkeley были разработаны системные вызовы *opendir*, *closedir*, *readdir* и *rewinddir*, чтобы программы могли читать каталоги, не зная их внутренней структуры. Позднее длинные имена файлов и эти системные вызовы были добавлены ко всем другим версиям UNIX и к стандарту POSIX.

Второе нововведение – это разбиение диска на **группы цилиндров**, у каждой из которых были собственные супер-блок, i-узлы и блоки данных. Идея такой организации диска заключается в том, чтобы хранить i-узел и блоки данных файла ближе друг к другу. Это позволило снизить время, затрачиваемое жестким диском на перемещение головок. По мере возможности блоки для файла выделяются в группе цилиндров, в которой содержится i-узел.

Третье изменение – это использование блоков двух размеров. Для хранения больших файлов значительно эффективнее использовать небольшое количество

крупных блоков, чем много маленьких блоков. С другой стороны, размер многих файлов в системе UNIX невелик, поэтому при использовании только блоков большого размера расходовалось бы слишком много дискового пространства. В ней применяются восьмикилобайтные блоки, которые в случае необходимости могут разбиваться на килобайтные фрагменты. Наличие блоков двух размеров обеспечивает эффективное чтение/запись для больших файлов и эффективное использование дискового пространства для небольших. Платой за эффективность является значительная дополнительная сложность программы.

Изначально в операционной системе Linux использовалась файловая система операционной системы MINIX. Однако в системе MINIX длина имен файлов ограничивалась 14 символами (для совместимости с UNIX Version 7), а максимальный размер файла был равен 64 Мбайт. Поэтому у разработчиков операционной системы Linux практически сразу появился интерес к усовершенствованию файловой системы. Первым шагом вперед стала файловая система Ext, в которой длина имен файлов была увеличена до 255 символов, а размер файлов – до 2 Гбайт. Однако эта система была медленнее файловой системы MINIX, поэтому исследования некоторое время продолжались. Наконец, была разработана файловая система Ext2 с длинными именами файлов, длинными файлами и высокой производительностью. Эта файловая система и стала основной файловой системой Linux. Однако операционная система Linux также поддерживает еще более десятка файловых систем, используя для этого файловую систему NFS (описанную далее). При компоновке операционной системы Linux предлагается сделать выбор файловой системы, которая будет встроена в ядро. Другие файловые системы при необходимости могут динамически подгружаться во время исполнения в виде модулей.

Файловая система Ext2 очень похожа на файловую систему Berkeley Fast File system с небольшими изменениями. Вместо того чтобы использовать группы цилиндров, что практически ничего не значит при современных дисках с виртуальной геометрией, она делит диск на группы блоков, независимо от того, где располагаются границы между цилиндрами. Каждая группа блоков начинается с суперблока, в котором хранится информация о том, сколько блоков и i-узлов находится в данной группе, о размере группы блоков и т.д. (рисунок 11.9). Затем следует описатель группы, содержащий информацию о расположении битовых массивов, количестве свободных блоков и i-узлов в группе, а также количестве

каталогов в группе. Эта информация важна, так как файловая система Ext2 пытается распространить каталоги равномерно по всему диску.

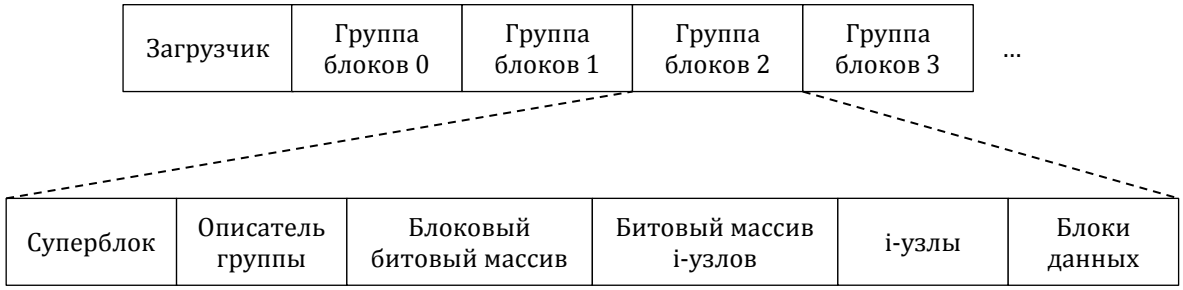


Рисунок 11.9 – Размещение файловой системы Ext2 на диске

В двух битовых массивах ведется учет свободных блоков и свободных i-узлов. Размер каждого битового массива равен одному блоку. При размере блоков в 1 Кбайт такая схема ограничивает размер группы блоков 8192 блоками и 8192 i-узлами. На практике ограничение числа i-узлов никогда не встречается, так как блоки заканчиваются раньше. Затем располагаются сами i-узлы. Размер каждого i-узла – 128 байт, что в два раза больше размера стандартных i-узлов в UNIX.

Работа файловой системы похожа на функционирование быстрой файловой системы Berkeley. Однако в отличие от BSD, в системе Linux используются дисковые блоки только одного размера, 1 Кбайт. Быстрая файловая система Berkeley использует 8-килобайтные блоки, которые затем разбиваются при необходимости на килобайтные фрагменты. Файловая система Ext2 делает примерно то же самое, но более простым способом. Как и система Berkeley, когда файл увеличивается в размерах, файловая система Ext2 пытается поместить новый блок файла в ту же группу блоков, что и остальные блоки, желательно сразу после предыдущих блоков. Кроме того, при создании нового файла в каталоге файловая система Ext2 старается выделить ему блоки в той же группе блоков, в которой располагается каталог. Новые каталоги, наоборот, равномерно распределяются по всему диску.

Другой файловой системой Linux является файловая система /proc (process – процесс). Идея этой файловой системы изначально была реализована в 8-й редакции операционной системы UNIX, созданной лабораторией Bell Labs, а позднее скопированной в 4.4BSD и System V. Однако в операционной системе Linux данная идея получила дальнейшее развитие. Основная концепция этой файловой системы заключается в том, что для каждого процесса системы создается подкаталог в каталоге /proc. Имя подкаталога формируется из PID процесса в десятичном

формате. Например, /proc619 – это каталог, соответствующий процессу с PID 619. В этом каталоге располагаются файлы, которые хранят информацию о процессе – его командную строку, строки окружения и маски сигналов. В действительности этих файлов на диске нет. Когда они считываются, система получает информацию от фактического процесса и возвращает ее в стандартном формате.

Многие расширения, реализованные в операционной системе Linux, относятся к файлам и каталогам, расположенным в каталоге /proc. Они содержат информацию о центральном процессоре, дисковых разделах, векторах прерывания, счетчиках ядра, файловых системах, подгружаемых модулях и о многом другом. Непривилегированные программы пользователя могут читать большую часть этой информации, что позволяет им узнать о поведении системы безопасным способом. Некоторые из этих файлов могут записываться в каталог /proc, чтобы изменить параметры системы.

Еще одной интересной файловой системой в мире UNIX является NFS (Network File System – сетевая файловая система). Она была разработана корпорацией Sun Microsystems и используется на всех современных UNIX системах. Эта файловая система позволяет на логическом уровне объединять файловые системы отдельных компьютеров в единое целое. В результате разные файловые системы представлены как отдельные каталоги единой системы.

11.8. Безопасность в UNIX

Операционная система UNIX с самого момента своего создания была многопользовательской системой. То есть системы обеспечения безопасности были встроены в нее с самого начала.

Каждому пользователю в ОС UNIX выдается свой уникальный **UID** (User ID – идентификатор пользователя), представляющий целое число от 0 до 65 535. Идентификатором владельца помечаются файлы, процессы и другие ресурсы. По умолчанию владельцем файла является пользователь, создавший этот файл, хотя владельца можно сменить.

Пользователи могут объединяться в группы, которые нумеруются при помощи GID (Group ID – идентификатор группы), имеющего вид 16-разрядного целого числа. Включение пользователя в группу выполняет администратор системы вручную. Изначально пользователь мог входить в одну группу, теперь имеется возможность включения его в несколько групп одновременно.

Механизм безопасности в UNIX заключается в следующем. Каждый процесс несет UID и GID владельца. Когда создается файл, он получает те же идентификаторы, что и создавший его процесс. Кроме того, файл получает набор разрешений доступа, определяемых создающим процессом. Эти разрешения определяют права доступа для владельца файла, для других членов группы владельца файла и для прочих пользователей. Эти права могут быть трех видов – чтение, запись и исполнение файла – *r*, *w* и *x* (*read*, *write* и *execute*). Возможность исполнять файл, конечно, имеет смысл только в том случае, если этот файл является исполняемой двоичной' программой. Попытка запустить файл, у которого есть разрешение на исполнение, но который не является исполняемым (то есть не начинается с соответствующего заголовка), закончится ошибкой. Поскольку существует три категории пользователей и три вида доступа для каждой категории, все режимы доступа к файлу можно закодировать 9 битами. Некоторые примеры этих 9-разрядных чисел и их значения показаны в таблице 11.2.

Таблица 11.2 – Некоторые примеры режимов защиты файлов

Двоичное	**Символьное**	**Разрешенный доступ**
111000000	rwx------	Владелец может читать, писать и исполнять
111111000	rwxrwx---	Владелец и группа могут читать, писать и исполнять
110100000	rw-r-----	Владелец может читать и писать, группа может читать
110100100	rw-r—r--	Владелец может читать и писать, все остальные – читать
111101101	rwxr-xr-x	Владелец имеет все права, остальные могут читать и исполнять
000000000	---------	Ни у кого нет доступа
000000111	------rwx	Только у посторонних есть доступ (странно, но законно)

Первые два примера в таблице понятны. В них к файлу предоставляется полный доступ для владельца файла и для его группы соответственно. В третьем примере группе владельца разрешается читать файл, но не разрешается его изменять, а всем посторонним запрещается всякий доступ. Вариант из четвертого примера часто применяется в тех случаях, когда владелец файла желает сделать файл с данными публичным. Пятый пример показывает режим защиты файла, представляющего собой опубликованную программу. В шестом примере доступ запрещен всем. Такой режим иногда используется для файлов-пустышек, применяемых для реализации взаимных исключений, так как любая попытка создания такого файла приведет к ошибке, если такой файл уже существует. Если несколько программ одновременно попытаются создать такой файл в качестве блокировки, только первой из них это удастся. Режим, показанный в последнем

примере, довольно странный, так как он предоставляет всем посторонним пользователям больше привилегий, чем владельцу файла. Тем не менее, такой режим допустим. К счастью, у владельца файла всегда есть способ изменить в дальнейшем режим доступа к файлу, даже если ему будет запрещен всякий доступ к самому файлу.

В системе UNIX существует **суперпользователь** (superuser или **root**). Он имеет право полного доступа ко всем файлам системы, независимо от того, кто является их владельцем и как они защищены. Его UID равен 0. Процессы с таким идентификатором также обладают возможностью обращаться к тем системным вызовам, доступ к которым запрещен всем остальным пользователям. Обычно пароль суперпользователя известен только системному администратору.

Каталоги представляют собой файлы и обладают теми же самыми режимами защиты, что и обычные файлы. Отличие состоит в том, что бит *x* интерпретируется в отношении каталогов как разрешение не исполнения, а поиска в каталоге. Таким образом, каталог с режимом *rwxr-xr-x* позволяет своему владельцу читать, изменять каталог, а также искать в нем файлы, а всем остальным пользователям разрешает только читать каталоги и искать файлы в них, но не создавать в них новые файлы или удалять файлы из этого каталога.

У специальных файлов, соответствующих устройствам ввода-вывода, есть те же самые биты защиты. Благодаря этому может использоваться тот же самый механизм для ограничения доступа к устройствам ввода-вывода. Например, владельцем специального файла принтера */dev/lp* может быть суперпользователь (root) или специальный пользователь, демон принтера. При этом режим доступа к файлу может быть установлен равным *rw*-------, чтобы все остальные пользователи не могли напрямую обращаться к принтеру. В противном случае при одновременной печати на принтере нескольких процессов получится полный хаос.

Конечно, тот факт, что файлом */dev/lp* владеет демон и этот файл имеет режим доступа *rw*-------, означает, что более никто не может выводить данные на принтер. Хотя такой способ и позволяет избежать множества неприятностей, однако время от времени пользователям бывает необходимо напечатать что-нибудь. В действительности существует более общая проблема регулируемого доступа ко всем устройствам ввода-вывода и другим системным ресурсам.

Эта проблема была решена с помощью добавления к перечисленным выше 9 бит нового бита защиты, **бита SETUID**. Когда выполняется программа с

установленным битом SETUID, то запускаемому процессу присваивается не UID вызвавшего его пользователя или процесса, а UID владельца файла. Когда процесс пытается открыть файл, то проверяется не UID запустившего его пользователя, а рабочий UID. Таким образом, если программой, обращающейся к принтеру, будет владеть демон с установленным битом SETUID, то любой пользователь сможет запустить ее, и запущенный процесс будет обладать полномочиями демона (например, правами доступа к */dev/lp*), но только для запуска этой программы (которая может устанавливать задания в очередь на принтер).

В операционной системе UNIX есть множество программ, владельцем которых является системный администратор, но у них установлен бит SETUID. Например, программе *passwd*, позволяющей пользователям менять свои пароли, требуется доступ записи к файлу паролей. Если разрешить изменять этот файл кому угодно, то ничего хорошего из этой затеи не получится. Вместо этого есть программа, владельцем которой выступает root, и у файла этой программы установлен бит SETUID. Хотя у этой программы есть полный доступ к файлу паролей, она изменит только пароль вызывавшего ее пользователя и не затронет остального содержимого файла.

Помимо бита SETUID, есть также еще и бит SETGID, работающий аналогично и временно предоставляющий пользователю рабочий GID программы. Однако на практике этот бит почти не используется.

Чаще всего из системных вызовов безопасности в UNIX используется системный вызов *chmod*. С его помощью можно изменить режим защиты файла. Например, оператор

```
s = chmod("/usr/ast/newgame", 0755)
```

устанавливает для файла *newgame* режим доступа *rwxr-xr-x*, что позволяет запускать эту программу всем пользователям (обратите внимание, что 0755 представляет собой восьмеричную константу, что удобно в данном случае, так как биты защиты группируются тройками). Изменять биты защиты могут только владелец файла и суперпользователь.

Системный вызов *access* проверяет, будет ли разрешен определенный тип доступа при заданных UID и GID. Этот системный вызов нужен, чтобы избежать появления брешей в системе безопасности. Он используется в программах с установленным битом SETUID, владельцем которых является root. Такие программы могут выполнять любые действия, поэтому им иногда бывает необходимо

определить, уполномочен ли вызвавший их пользователь на выполнение определенных действий.

Рассмотрим, каким образом реализуется безопасность в UNIX. Когда пользователь входит в систему, программа регистрации *login* (которая является SETUID root) запрашивает у пользователя его имя и пароль. Затем она хэширует пароль и ищет его в файле паролей */etc/passwd*, чтобы определить, соответствует ли хэш-код содержащимся в нем значениям. Хэширование применяется, чтобы избежать хранения пароля в незашифрованном виде где-либо в системе. Если пароль введен верно, то программа регистрации запускает оболочку, которую предпочитает пользователь. Затем программа регистрации использует системные вызовы *setuid* и *setgid*, чтобы установить для себя UID и GID (как мы помним, она была запущена как SETUID root). После этого программа регистрации открывает клавиатуру для стандартного ввода (файл с дескриптором 0) и экран для стандартного вывода (файл с дескриптором 1), а также экран для вывода стандартного потока сообщений об ошибках (файл с дескриптором 2). Наконец, она выполняет оболочку, которую указал пользователь, таким образом, завершая свою работу.

С этого момента начинает работу оболочка с установленными UID и GID, а также стандартными потоками ввода, вывода и ошибок, настроенными на устройства ввода-вывода по умолчанию. Все процессы, которые она запускает при помощи системного вызова *fork* (то есть команды, вводимые пользователем с клавиатуры), автоматически наследуют UID и GID оболочки, поэтому у них будет верное значение владельца и группы. Все файлы, создаваемые этими процессами, также будут иметь эти значения.

Когда любой процесс пытается открыть файл, система сначала проверяет биты защиты в i-узле файла для заданных значений рабочих UID и GID, чтобы определить, разрешен ли доступ для данного процесса. Если доступ разрешен, файл открывается и процессу возвращается дескриптор файла. В противном случае файл не открывается, а процессу возвращается значение "–1". При последующих обращениях к системным вызовам *read* и *write* проверка не выполняется. В результате, если режим защиты файла изменяется уже после того, как файл открыт, новый режим не повлияет на процессы, которые уже успели открыть этот файл.

В операционной системе Linux защита файлов и ресурсов осуществляется так же: как и в UNIX.

12. ОПЕРАЦИОННАЯ СИСТЕМА WINDOWS

12.1. История Windows

Операционные системы корпорации Microsoft можно разделить на три семейства: MS-DOS, Consumer Windows (Windows 95/98/Me) и Windows NT. Рассмотрим вкратце все эти семейства.

В 1981 году корпорация IBM создала персональный компьютер IBM PC, основанный на процессоре Intel 8088. Этот компьютер оснащался 16-разрядной операционной системой MS-DOS 1.0 с резидентной программой размером 8 Кбайт. Разработчиком системы была начинающая фирма Microsoft, более известная как создатель BASIC для систем на базе Intel 8080 и Zilog Z80. Через два года появилась вторая версия MS-DOS с 24 Кбайт резидентного кода. Она содержала программу обработки командной строки (оболочку) с большим количеством функций, позаимствованных у UNIX.

В 1986 году появились новые процессоры Intel 286, на базе которых IBM выпустила новый компьютер IBM PC/AT. AT означало Advanced Technology (передовая технология), так как этот процессор работал на впечатляющей тогда частоте в 8 МГц и мог адресоваться к 16 Мбайт памяти. Правда, в большинстве компьютеров было только по 1-2 Мбайт памяти из-за большой ее стоимости. Для этого компьютера была выпущена версия MS-DOS 3.0, занимавшая в памяти 36 Кбайт. Со временем операционная система MS-DOS менялась, количество команд в ней увеличивалось, но она так и оставалась ориентированной на командную строку.

Обратив внимание на графический пользовательский интерфейс компьютера Apple Lisa, предшественника Apple Macintosh, Microsoft добавила к MS-DOS графическую оболочку, которую назвала Windows. В 1985 г. была выпущена версия Windows 1.0, в 1987 г. – Windows 2.0. Обе эти системы не были особенно удачными. Только версия Windows 3.0 для процессора Intel 386, выпущенная в 1990 г. и особенно версии Windows 3.1 и 3.11 for Workgroups добились большого коммерческого успеха. Все эти версии не были как таковыми операционными системами. Это были просто графические оболочки поверх MS-DOS. Все программы работали в одном и том же адресном пространстве, и ошибка в одной из них могла повесить всю систему.

В августе 1995 г. появилась Windows 95, в которой почти все функции были переданы ей от MS-DOS, имевшей в тот момент версию 7.0. Windows 95 была частично 32-разрядной. Она содержала большие куски 16-разрядного кода и продолжала использовать MS-DOS со всеми ее ограничениями. Единственное значительное отличие в Windows 95 – это возможность использования длинных имен файлов вместо формата 8.3, поддерживаемого MS-DOS.

В июне 1998 г. вышла новая версия – Windows 98, в которой также продолжала являться основой операционная система MS-DOS (версия 7.1). Она не сильно отличалась от Windows 95, хотя еще больше функций было передано в нее от MS-DOS, и она стала поддерживать большие жесткие диски (больше 2 Гб). Кроме того, в интерфейс пользователя была очень тесно интегрирована поддержка Интернет, что вызвало судебные преследования за нарушение антимонопольного законодательства.

Главными недостатками Windows 98 были большой 16-разрядный кусок программного кода, неполная многозадачность, а также особенности использования имеющихся 4 Гб адресного пространства. Оно разделялось таким образом, что существовала возможность помещения структуры данных ядра в пространство пользователя для совместимости со старыми программами для MS-DOS. В результате могло происходить повреждение критических областей данных и падение всей системы.

В 2000 году была выпущена слегка модернизированная версия – Windows Me (Millennium Edition). В ней были исправлены некоторые недостатки, добавлен ряд новых функций, но фактически это была та же Windows 98. Она позволяла лучше организовать работу с Интернет и мультимедиа, а также в ней появилась возможность возвращения к последней работающей конфигурации в случае неверной установки каких-либо параметров.

Уже в конце 80-х годов Microsoft поняла, что необходима разработка полностью 32-разрядной операционной системы, никак не связанной с MS-DOS. В результате в 1993 году была выпущена Windows NT 3.1. Этот номер версии был выбран в расчете на то, что пользователи перейдут с популярной тогда Windows 3.1 на Windows NT 3.1. Однако этого не произошло ввиду того, что на тот момент не существовало 32-разрядных программ, кроме того, новая ОС требовала значительно больших объемов памяти. При этом при выпуске версий Windows 95, 98 и Me

Microsoft постоянно говорила, что уж эта версия будет окончательно последней системой, основанной на MS-DOS.

Тем не менее, несмотря на отсутствие популярности для персональных компьютеров, Windows NT 3.1 стала пользоваться спросом для серверов, так как значительно превосходила другие версии Windows по характеристикам. Были выпущены несколько версий Windows NT 3.x, но они также не были особенно популярны.

Первая серьезно усовершенствованная версия системы NT появилась в 1996 г. и получила название Windows NT 4.0. Эта система уже обладала хорошей мощностью, безопасностью и надежностью. При этом она использовала графический интерфейс популярной тогда Windows 95. Это позволило привлечь большее число поклонников.

Windows NT с самых первых версий была рассчитана на переносимость с платформы на платформу, поэтому была почти полностью написана на языке C, однако сохраняя небольшие включения на ассемблере для низкоуровневых функций. Программная основа у первой версии состояла из 3,1 млн. строк, тогда как у Windows NT 4.0 она выросла до 16 млн. строк. В ней кроме языка C использовался также и язык C++ для написания пользовательского интерфейса.

Следом за Windows NT 4.0 должна была появиться версия 5.0, но вместо нее появилась Windows 2000. Она уже стала первой операционной системой, сделавшей попытку объединения поклонников как ветви Windows 98, так и ветви Windows NT. В ней полностью использовалась 32-разрядная, а впоследствии и 64-разрядная технология одновременно с интерфейсом популярной тогда Windows 98. Кроме того, она была хорошо защищенной, безопасной, многозадачной, многопроцессорной (до 32 процессоров) и многопользовательской системой.

От Windows 98 Windows 2000 получила поддержку технологии plug-and-play, шины USB, стандарта IEEE 1394 (FireWire), инфракрасного порта, а также управление питанием. Кроме того, появились и новые функции, которых ранее нигде не было – Active Directory, система безопасности Kerberos, поддержка смарт-карт, мониторинг системы, инфраструктура системного администрирования и другие. Windows 2000 получила и новую версию файловой системы – NTFS, которая позволяла получать одновременный доступ различным пользователям к одному связанному файлу. Как только пользователь обращался для записи к этому файлу, то одновременно создавалась его копия.

Операционная система Windows 2000 имела несколько версий – Professional для персональных компьютеров и минимальных сетей до 10 клиентов, а также максимум для 2 процессоров. Для больших серверов были версии Server, Advanced Server и Datacenter Server. Они отличались количеством поддерживаемых процессоров, клиентов и объема ОЗУ.

Следующее поколение операционной системы Windows, появившееся в 2001 году, снова разделилось на системы для ПК и системы для серверов. Операционная система для ПК получила название Windows XP (NT 5.1), а для серверов – Windows 2003 (NT 5.2), которая также делилась на версии в зависимости от возможностей.

Windows XP была, фактически, самой удачной и самой популярной из всех операционных систем корпорации Microsoft – ее распространенность составляла 42-76% от всех ОС, используемых на рабочих станциях.

В Windows XP появился новый интерфейс, сглаживание текста ClearType, возможность быстрого переключения пользователей, удаленный помощник, программа восстановления системы, запись на CD-R/RW и многие другие функции. Несмотря на большую любовь пользователей к Windows XP, связанную с устойчивостью и быстродействием, у нее есть очень серьезный недостаток в плане безопасности – при установке этой ОС на компьютер по умолчанию новый пользователь получает права администратора и в дальнейшем большинство пользователей так и продолжают работать с этими правами, что небезопасно.

Следующей операционной системой от корпорации Microsoft стала Windows Vista (NT 6.0), выпущенная в ноябре 2006 года. Ее серверным аналогом является Windows Server 2008. Эта операционная система получила новый интерфейс, улучшенную безопасность и меньшую скорость работы. Она может использовать внешние USB накопители для свопинга, что ускоряет ее работу. Также в ней появилась система контроля учетных записей пользователей (UAC – User Access Control), которая требует явного разрешения пользователя при выполнении любого действия, требующего административных полномочий, вне зависимости от прав текущего аккаунта пользователя. Кроме того, Vista использует новые технологии шифрования дисков и файлов, а также предотвращает заражение вирусами с извлекаемых носителей благодаря отключенному автозапуску программ с флэш-карт и устройств USB.

Windows Vista не пользовалась большим спросом ввиду завышенных требований к оборудованию, медленной работе, надоедающих пользователям

запросам UAC, а также долгим автоматическим обновлениям, которые, тем не менее, требуют ручного вмешательства. В 2007 году Windows Vista получила титул "Провал года".

Относительно недавно (в 2009 году), то есть всего через 2 года после начала продаж Windows Vista, ей на смену пришла новая операционная система – Windows 7 (NT 6.1) (серверный аналог – Windows Server 2008 R2). Это, фактически, вторая по популярности операционная система от Microsoft после Windows XP (34 % на данный момент). Возможно, она даже сможет выйти на первое место, учитывая темпы роста популярности (в 2 раза за 1 год).

Новшествами Windows 7 являются поддержка сенсорного управления, более тесная интеграция с производителями драйверов, возможность кэширования содержимого сетевого трафика. Кроме того, в ней улучшены совместимость с Windows XP, работа с мультимедиа, функционирование UAC, изменены технологии шифрования, улучшена защита от вирусов, а также улучшен интерфейс пользователя.

К недостаткам Windows 7 относят недостаточно быструю производительность по сравнению с Windows XP (однако, улучшенную по сравнению с Windows Vista), большую нагрузку на батареи нетбуков (особенно, выпущенных ранее выхода этой версии ОС), а также использование технологий, позволяющих Microsoft вторгаться в личную жизнь пользователя.

В октябре 2012 года была представлена новая версия ОС от компании Microsoft – Windows 8 (NT 6.2). Ее серверным вариантом является Windows Server 2012. Windows 8 позиционируется как единое решение для настольных и планшетных компьютеров, то есть она рассчитана одновременно на сенсорный и традиционный интерфейс.

Windows 8, в отличие от своих предшественников – Windows 7 и Windows XP, – использует новый интерфейс под названием *Metro*. Этот интерфейс появляется первым после запуска системы; он схож по функциональности с рабочим столом – стартовый экран имеет плитки приложений (сродни ярлыкам и иконкам), по нажатию на которые запускается приложение, открывается сайт или папка (в зависимости от того, к какому элементу или приложению привязана плитка).

Также в системе присутствует и "классический" рабочий стол в виде отдельного приложения. Вместо меню "Пуск" в интерфейсе первоначально использовался "активный угол", нажатие на который открывало стартовый экран.

После одного из обновлений кнопка "Пуск" вернулась на свое исконное место, однако пользователю предоставляется возможность от нее отказаться. Прокрутка в Metro-интерфейсе идет горизонтально. Также, если сделать жест уменьшения (или нажать на минус внизу экрана), будет виден весь стартовый экран. Плитки на стартовом экране можно перемещать и группировать, давать группам имена и изменять размер плиток (доступно только для плиток, которые были изначально большими). В зависимости от разрешения экрана система автоматически определяет количество строк для плиток – на стандартных планшетных компьютерах три ряда плиток. Цвет стартового экрана меняется в новой панели управления, также меняется и орнамент на заднем фоне.

Основные нововведения Windows 8:

– Учетная запись Microsoft и синхронизация параметров: Возможность войти в Windows с помощью Live ID. Это позволит войти в профиль пользователя и загрузить настройки через интернет, а также добавляет интеграцию со SkyDrive.

– Магазин приложений Windows Store: единственный способ покупки и загрузки Metro-приложений.

– Два новых метода для аутентификации пользователя: картинка-пароль, позволяющая пользователю войти в систему при помощи трех касаний, и четырёхзначный PIN-код, а также встроенная поддержка биометрических устройств. Пароль нелокальной учетной записи пользователя соответствует паролю учетной записи Microsoft.

– Internet Explorer 10. IE 10 в Windows 8 включен в настольном и сенсорном вариантах. Последний не поддерживает плагины или ActiveX, но включает в себя версию проигрывателя Adobe Flash Player, который оптимизирован для сенсорного управления.

– Проводник. Проводник включает в себя Ribbon-ленту (наподобие ленты в Microsoft Office и Windows Essentials) и улучшения в способах разрешения конфликтов при переносе или копировании файлов.

– Восстановление системы. Добавлено две новые функции: Обновление (*Refresh*) и Сброс (*Reset*). Обновление для Windows восстанавливает все системные файлы в исходное состояние, сохраняя при этом все настройки, пользовательские файлы и приложения. Сброс же возвращает компьютер к заводским настройкам.

– Новый диспетчер задач. В Windows 8 диспетчер задач был полностью изменен. Добавлены новые графики производительности, оптимизировано

управление выполняющимися приложениями, фоновыми процессорами и службами на единой вкладке "Производительность". Также в диспетчер задач было перенесено управление автозагрузками из "Конфигурации системы".

– Функция "Семейная безопасность" была встроена в Windows, управление семейной безопасностью осуществляется в панели управления.

– Добавлена поддержка USB 3.0, Bluetooth 4.0, DirectX 11.1 и NET.Framework 4.5.

– Персонализация: после запуска на экране появляется картинка с текущим временем и датой. Для начала работы нужно нажать любую кнопку, открыв экран приветствия. Саму картинку можно сменить в настройках. Добавлено автоопределение цвета в темах для рабочего стола.

– Новая панель управления в стиле Metro UI, которая позволяет быстро изменить некоторые настройки системы.

– Усовершенствованный поиск: На начальном экране нужно лишь нажать любую клавишу для начала поиска по приложениям, параметрам и т.п.

– Переключение раскладки клавиатуры: менять раскладку клавиатуры можно также с помощью сочетаний клавиш Windows + Space.

Windows 8 получает отрицательные отзывы со стороны владельцев компьютеров без сенсорного дисплея из-за преобладания интерфейса Metro. Пользователи критикуют изменённый интерфейс, заставляющий тратить дополнительное время на обучение работе с новой операционной системой. Хотя большинство новшеств описано в справочной системе, которая вызывается нажатием клавиши F1 при открытом рабочем столе.

Например, новичку сложно найти кнопку для перезагрузки или установленные приложения.

Нет встроенной в ОС возможности отключить при загрузке Windows показ экрана "Пуск" и сразу перейти на рабочий стол. После установки Windows на данном экране присутствуют только плитки интерфейса Metro, к тому же в основном с малопопулярными в России сервисами Microsoft. В то же время для пользователей настольных компьютеров привычным является рабочий стол с ярлыками приложений и панелью задач. При этом приложения, запускаемые с рабочего стола, как правило, обладают большей функциональностью, в том числе по настройке под конкретного пользователя, по сравнению с Metro-приложениями, так как в последних упор сделан более на визуальную составляющую и на качественное,

соответствующее стилю Metro, отображение содержимого. Частичным решением вышеописанной проблемы является закрепление на экране "Пуск" ярлыков нужных приложений, но это не изменяет существенно опыт взаимодействия.

Вместе с отрицательными отзывами от первых пользователей всплыл недоработанный сервис активации, способный предоставить бесплатный код активации любому пользователю. В декабре 2012 года эта уязвимость была устранена.

Также наблюдается проблема, связанная со снижением скорости беспроводного интернет-соединения у владельцев устройств со встроенными Wi-Fi чипами от Broadcom при использовании старой версии драйвера от Windows 7.

Главный маркетинговый директор Microsoft, Тами Реллер, в одном из интервью сказал, что некоторые ключевые элементы Windows 8 будут изменены при выпуске обновленной версии системы Windows 8.1. Это было воспринято некоторыми СМИ как фактическое признание неудачи компании с выпуском Windows 8.

12.2. Структура системы

Операционные системы Windows семейства NT состоят из двух основных частей: самой операционной системы, работающей в режиме ядра, и подсистем окружения, работающих в режиме пользователя. Ядро управляет процессами, памятью, файловой системой и т.д. Подсистемы окружения – это отдельные процессы, помогающие пользователю выполнять определенные системные функции.

Одно из многих усовершенствований системы NT по сравнению с Windows 3.1 заключалось в ее модульной структуре. Она состояла из относительно небольшого ядра, работавшего в режиме ядра, плюс нескольких серверных процессов, работавших в режиме пользователя. Процессы пользователя взаимодействовали с серверными процессами с помощью модели клиент-сервер: клиент посылал серверу сообщение, а сервер выполнял определенную работу и возвращал клиенту результат в ответном сообщении. Такая модульная структура упрощала перенос системы на другие компьютеры. В результате операционная система Windows NT была успешно перенесена на платформы с процессорами, отличными от процессоров Intel, а именно: Alpha корпорации DEC, Power PC корпорации IBM и MIPS фирмы SGI. Кроме того, такая структура защищала ядро от ошибок в коде серверов.

Однако для увеличения производительности, начиная с версии NT 4.0, довольно большая часть операционной системы (например, управление системными вызовами и вся экранная графика) были возвращены ядро. Такая схема сохранилась и в последующих версиях Windows.

Операционная система разделена на несколько уровней, каждый из которых пользуется службами лежащего ниже уровня. Эта структура проиллюстрировала на рисунке 12.1. Затененная область обозначает исполняющую систему. Квадратики, помеченные символом "D" обозначают драйверы устройств. Сервисные процессы являются системными демонами. Один из уровней разделен горизонтально на множество модулей. У каждого модуля есть определенная функция, а также четко определенный интерфейс для взаимодействия с другими модулями.

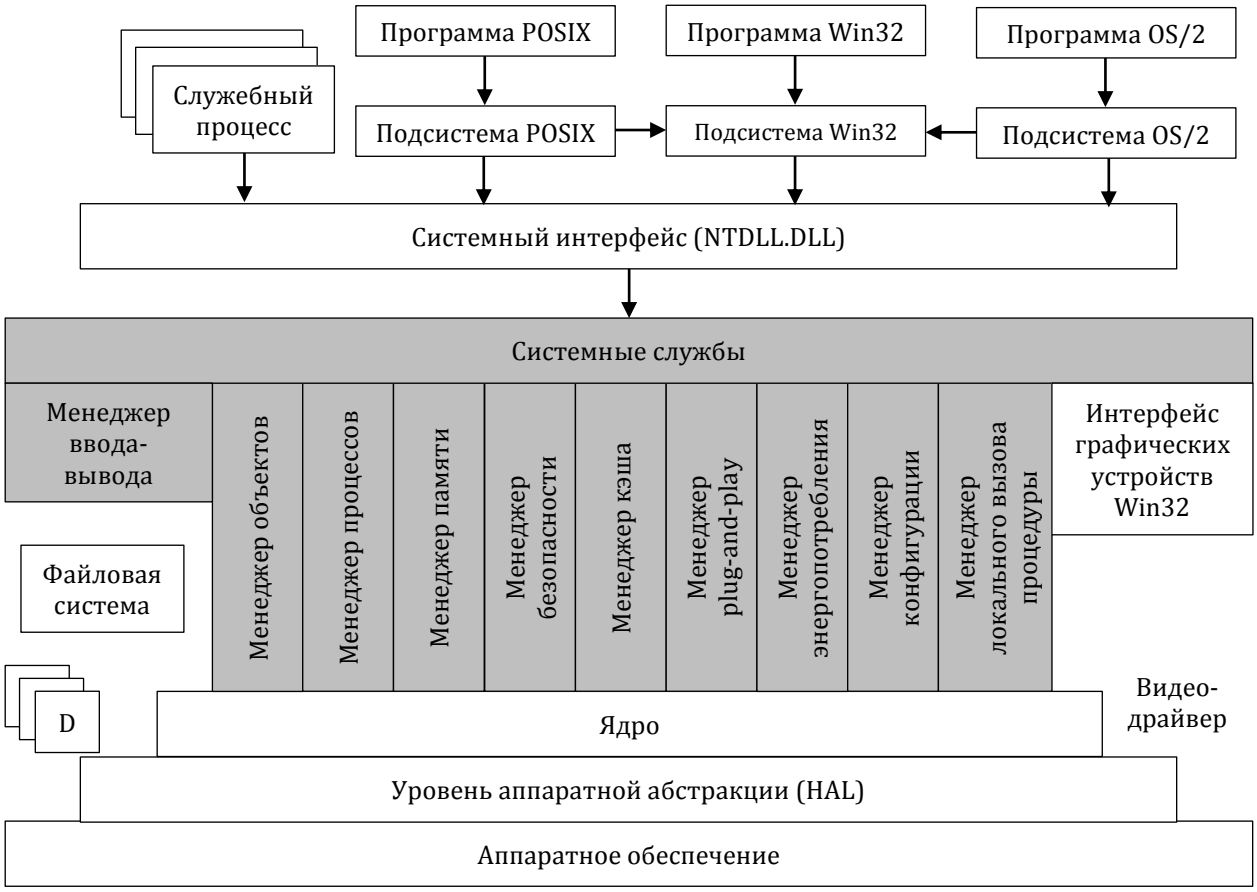


Рисунок 12.1 – Структура Windows семейства NT

Два нижних уровня программного обеспечения, уровень аппаратных абстракций (HAL, Hardware Abstraction Layer) и ядро написаны на языке C и ассемблере и являются частично машинно-зависимыми. Верхние уровни написаны исключительно на C и почти полностью машинно-независимы. Драйверы написаны на C или, в некоторых случаях, на C++.

Работа уровня аппаратной абстракции (HAL) заключается в том, чтобы предоставлять всей остальной системе абстрактные аппаратные устройства, свободные от отличительных особенностей. Эти устройства представляются в виде машинно-независимых служб (процедурных вызовов и макросов), которые могут использоваться остальной ОС и драйверами. Это позволяет легче осуществлять переносимость ОС с одной аппаратной платформы на другую. То есть этот уровень позволяет скрывать различия между материнскими платами различных производителей. Уровень HAL управляет регистрами и адресами устройств, прерываниями, DMA, таймерами, спин-блокировками, а также предоставляет интерфейс взаимодействия с BIOS.

Хотя эффективность уровня HAL, является довольно высокой, для мультимедийных приложений ее может быть недостаточно. По этой причине корпорация Microsoft также производит пакет программного обеспечения, называемый DirectX, расширяющий функциональность уровня HAL дополнительными процедурами и предоставляющий пользовательским процессам прямой доступ к аппаратному обеспечению.

Выше уровня HAL находятся ядро ОС и драйверы устройств. Часть ядра и большая часть уровня HAL постоянно находятся в оперативной памяти. Большая часть ядра написана на языке С кроме тех мест, в которых производительность считается важнее всех остальных задач.

Назначение ядра – сделать всю остальную часть ОС независимой от аппаратуры и легко переносимой на другие платформы. Ядро получает доступ к аппаратуре через уровень HAL. Оно построено на чрезвычайно низкоуровневых службах уровня HAL, формируя из них абстракции более высоких уровней. Например, у уровня HAL есть вызовы для связывания процедур обработки прерываний с прерываниями и установки их приоритетов, но больше практически ничего уровень HAL в этой области не делает. Ядро, напротив, предоставляет полный механизм для переключения контекста. Оно должным образом сохраняет все регистры центрального процессора, изменяет таблицы страниц, сохраняет кэш центрального процессора и т. д. Когда все эти действия выполнены, работавший ранее поток оказывается полностью сохраненным в таблицах, расположенных в памяти. Затем ядро настраивает карту памяти нового потока и загружает его регистры. После чего новый поток готов к работе.

Программа планирования потоков также располагается в ядре. Когда наступает пора проверить, не готов ли к работе новый поток, например, после того, как истечет выделенный потоку квант времени или по завершении процедуры обработки прерываний ввода-вывода, ядро выбирает поток и выполняет переключение контекста, необходимое, чтобы запустить этот поток. С точки зрения остальной операционной системы, переключение потоков автоматически осуществляется более низкими уровнями, так что для более высоких уровней не остается никакой работы.

Еще одна функция ядра – это предоставление низкоуровневой поддержки двум классам объектов – управляющим объектам и объектам диспетчеризации. Они представляют собой внутренние объекты, на основе которых исполняющая система строит объекты пользователя.

Над ядром и драйверами устройств располагается верхняя часть ОС, называемая **исполняющей системой**. Она написана на языке С, не зависит от архитектуры и может быть перенесена на новые машины с относительно небольшими усилиями. Она состоит из 10 компонентов, представляющих собой набор процедур, работающих вместе для выполнения некоторой задачи. К этим компонентам относятся:

1. **Менеджер объектов** управляет всеми объектами, известными операционной системе. К ним относятся процессы, потоки, файлы, каталоги, семафоры, устройства ввода-вывода, таймеры и многое другое. При создании объекта менеджер объектов получает в адресном пространстве ядра блок виртуальной памяти и возвращает этот блок в список свободных блоков, когда объект уничтожается. Его работа заключается в том, чтобы следить за всеми объектами.

2. **Менеджер ввода-вывода** формирует каркас для управления устройствами ввода-вывода и предоставляет общие службы ввода-вывода. Он предоставляет остальной части системы независимый от устройств ввод-вывод, вызывая для выполнения физического ввода-вывода соответствующий драйвер. Здесь также располагаются все драйверы устройств (обозначены символом "D" на рисунке 12.1). Файловые системы формально являются драйверами устройств под управлением менеджера ввода-вывода. Существует два драйвера для файловых систем FAT и NTFS, независимые друг от друга и управляющие различными разделами диска. Все файловые системы FAT управляются одним драйвером.

3. **Менеджер процессов** управляет процессами и потоками, включая их создание и завершение. Он занимается не стратегиями, применяемыми по отношению к процессам, а механизмом, используемым для управления ими. Менеджер процессов основывается на объектах потоков и процессов ядра и добавляет к ним дополнительные функции. Это ключевой элемент многозадачности в Windows.

4. **Менеджер памяти** реализует архитектуру виртуальной памяти со страничной подкачкой по требованию операционной системы. Он управляет преобразованием виртуальных страниц в физические страничные блоки. Таким образом, он реализует правила защиты, ограничивающие доступ каждого процесса только теми страницами, которые принадлежат его адресному пространству, а не адресным пространствам других процессов (кроме специальных случаев). Он также контролирует определенные системные вызовы, относящиеся к виртуальной памяти.

5. **Менеджер безопасности** приводит в исполнение сложный механизм безопасности Windows, удовлетворяющий требованиям класса C2 Оранжевой книги Министерства обороны США. В Оранжевой книге перечислено множество правил, которые должна соблюдать система, начиная с аутентификации при регистрации и заканчивая управлением доступом, а также обнулением страниц перед их повторным использованием.

6. **Менеджер кэша** хранит в памяти блоки диска, которые использовались в последнее время, чтобы ускорить доступ к ним в случае, если они понадобятся вновь. Его работа состоит в том, чтобы определить, какие блоки понадобятся снова, а какие нет. Операционная система может одновременно использовать несколько файловых систем. В этом случае менеджер кэша обслуживает все файловые системы, таким образом, каждой файловой системе не нужно заниматься управлением собственного кэша. Когда требуется блок, он запрашивается у менеджера кэша. Если у менеджера кэша нет блока, он обращается за блоком к соответствующей файловой системе. Поскольку файлы могут отображаться в адресное пространство процессов, менеджер кэша должен взаимодействовать с менеджером виртуальной памяти, чтобы обеспечить требуемую непротиворечивость. Количество памяти, выделенной для кэша, динамически изменяется и может увеличиваться или уменьшаться при необходимости.

7. **Менеджер plug-and-play** получает все уведомления об установленных новых устройствах. Для некоторых устройств проверка производится при загрузке системы, но не после нее. Другие устройства, например устройства USB, могут подключаться в любое время, и их подключение запускает пересылку сообщения менеджеру plug-and-play, который затем находит и загружает соответствующий драйвер.

8. **Менеджер энергопотребления** управляет потреблением электроэнергии. Он выключает монитор и диски, если к ним не было обращений в течение определенного интервала времени. На переносных компьютерах менеджер энергопотребления следит за состоянием батарей и, когда заряд батарей подходит к концу, предпринимает соответствующие действия. Эти действия, как правило, заключаются в том, что он сообщает работающим программам о состоянии батарей. В результате программы могут сохранить свои файлы и приготовиться к корректному завершению работы.

9. **Менеджер конфигурации** отвечает за состояние реестра. Он добавляет новые записи и ищет запрашиваемые ключи.

10. **Менеджер вызова локальной процедуры** обеспечивает высокоэффективное взаимодействие между процессами и их подсистемами. Поскольку этот путь нужен для выполнения некоторых системных вызовов, эффективность оказывается критичной, вот почему для этого не используются стандартные механизмы межпроцессного взаимодействия.

Также в пространстве ядра располагается исполняющий модуль Win32 GDI, который обрабатывает определенные системные вызовы (но не все). Изначально он располагался в пространстве пользователя, но в версии NT 4.0 для увеличения производительности был перенесен в пространство ядра. **Интерфейс графических устройств GDI** (Graphic Device Interface) занимается управлением графическими изображениями для монитора и принтеров. Он предоставляет системные вызовы, позволяющие пользовательским программам выводить данные на монитор и принтеры независящим от устройств способом. Он также содержит оконный менеджер и драйвер дисплея. До версии NT 4.0 интерфейс графических устройств также находился в пространстве пользователя, но производительность при этом оставляла желать лучшего, поэтому корпорация Microsoft переместила его в ядро.

На верху исполняющей системы размещается уровень системных служб. Его функции заключаются в предоставлении интерфейса к исполняющей системе. Он

принимает настоящие системные вызовы Windows и вызывает другие части исполняющей системы для их выполнения.

При загрузке операционная система Windows загружается в память как набор файлов. Основная часть операционной системы, состоящая из ядра и исполняющей системы, хранится в файле *ntoskrnl.exe*. Уровень HAL представляет собой библиотеку общего доступа, расположенную в отдельном файле *hal.dll*. Интерфейс Win32 и интерфейс графических устройств хранятся вместе в третьем файле, *win32k.sys*. Наконец, загружается множество драйверов устройств. У большинства из них расширение *.sys*.

Драйверы устройств не являются частью двоичного файла *ntoskrnl.exe*. Преимущество такого подхода заключается в том, что как только драйвер устанавливается в систему, он добавляется в реестр и затем динамически загружается при каждой загрузке системы. Таким образом, файл *ntoskrnl.exe* остается одинаковым для всех конфигураций системы, но каждая система точно настраивается на конфигурацию аппаратуры. Каждый драйвер может управлять одним или несколькими устройствами ввода-вывода, но драйвер устройства может также выполнять действия, не относящиеся к какому-либо специфическому устройству – шифровать поток данных или даже просто предоставлять доступ к структурам данных ядра.

Существуют драйверы для реально видимых и осязаемых устройств ввода-вывода, таких как диски и принтеры, но также есть драйверы для многих внутренних устройств и микросхем, о которых практически никто ничего не слышал. Кроме того, как уже было сказано, файловые системы также представлены в виде драйверов устройств. Самый большой драйвер устройства для интерфейса Win32 (GDI и видеодрайвер) показан на правой стороне рисунка 12.1. Он обрабатывает множество системных вызовов и управляет большей частью графики. Поскольку пользователи могут устанавливать новые драйверы, у них есть возможность изменить содержимое ядра и повредить систему. По этой причине драйверы следует писать с большой осторожностью.

Немного остановимся на компонентах, работающих в режиме пользователя. Таких компонентов три: динамические библиотеки DLL, подсистемы окружения и служебные процессы. Эти компоненты работают вместе, предоставляя каждому пользовательскому процессу интерфейс, отличный от интерфейса системных вызовов текущей версии Windows.

ОС Windows поддерживает три различных документированных интерфейса прикладного программирования API: Win32, POSIX и OS/2. Официальным интерфейсом для ОС Windows является Win32. Программа, написанная при использовании библиотеки DLL и подсистемы окружения Win32 будет работать без каких-либо изменений на всех последних версиях Windows, несмотря на то, что сами системные вызовы в различных системах различны.

В операционной системе Windows средой POSIX предоставляется минимальная поддержка для приложений UNIX. Эта поддержка была введена, так как некоторые министерства правительства США требовали, чтобы ОС для правительственных компьютеров были совместимы со стандартом 1003.1. Перенос же любой реальной программы из UNIX в Windows практически невозможен.

Функциональность подсистемы OS/2 ограничена практически так же, как и POSIX. На практике она полностью бесполезна.

Рассмотрим способ реализации этих интерфейсов на примере Win32. Программа, пользующаяся интерфейсом Win32, как правило, состоит из большого количества обращений к функциям Win32 API, например *CreateWindow*, *DrawMenuBar* и *OpenSemaphore*. Существуют тысячи подобных вызовов, и большинство программ использует значительное их количество. Один из возможных способов реализации заключается в статическом связывании каждой программы, использующей интерфейс Win32, со всеми библиотечными процедурами, которыми она пользуется. При таком подходе каждая двоичная программа будет содержать копию всех используемых ею процедур в своем исполняемом двоичном файле.

Недостаток такого подхода заключается в том, что при этом расходуется много памяти, если пользователь одновременно откроет несколько программ, использующих одни и те же библиотечные процедуры. Например, программы *Word, Excel* и *PowerPoint* используют абсолютно одинаковые процедуры для открытия диалоговых окон, рисования окон, отображения меню, работы с буфером обмена и т.д. Поэтому, если одновременно открыть все эти программы, при такой реализации программ в памяти будут находиться три идентичные копии каждой библиотечной процедуры.

Чтобы избежать подобной проблемы, все версии Windows поддерживают динамические библиотеки, называемые **DLL** (Dynamic-Link Library – динамически подсоединяемая библиотека). Каждая динамическая библиотека содержит набор

тесно связанных библиотечных процедур и все их структуры данных в одном файле, как правило (но не всегда), с расширением *.dll*. Когда приложение компонуется, компоновщик видит, что некоторые библиотечные процедуры принадлежат к динамическим библиотекам, и записывает эту информацию в заголовок исполняемого файла. Обращения к процедурам динамических библиотек производятся не напрямую, а при помощи вектора передачи в адресном пространстве вызывающего процесса. Изначально этот вектор заполнен нулями, так как адреса вызываемых процедур еще неизвестны.

При запуске прикладного процесса все требуемые динамические библиотеки обнаруживаются (на диске или в памяти) и отображаются на виртуальное адресное пространство процесса. Затем вектор передачи заполняется верными адресами, что позволяет вызывать библиотечные процедуры через этот вектор с незначительной потерей производительности. Выигрыш такой схемы заключается в том, что при запуске нескольких приложений, использующих одну и ту же динамическую библиотеку, в физической памяти требуется только одна копия текста DLL (но каждый процесс получает свою собственную копию приватных статических данных в DLL). В операционной системе Windows динамические библиотеки используются очень активно для всех аспектов системы.

Самым важным понятием в операционной системе Windows является понятие **объектов**. Они предоставляют однородный и непротиворечивый интерфейс ко всем системным ресурсам и структурам данных, таким как процессы, потоки, семафоры и т.д. Эта однородность достигается несколькими путями.

1. Все объекты именуются по одной и той же схеме. Доступ к ним также предоставляется одинаково, при помощи дескриптора объектов.
2. Так как доступ к объектам всегда осуществляется через менеджер объектов, то все проверки, связанные с защитой, располагаются в одном месте.
3. Возможно совместное использование объектов по одной и той же схеме.
4. Так как все объекты открываются и закрываются через менеджер объектов, несложно отследить, какие объекты все еще используются, а какие можно безопасно удалить.
5. Эта однородная модель для управления объектов позволяет легко регулировать квоты ресурсов.

Объекты представляют собой структуры данных в виртуальном адресном пространстве. Поэтому при перезагрузке они теряются.

Каждый объект содержит заголовок с определенной информацией, общей для всех объектов всех типов. Поля заголовка включают имя объекта, каталог, в котором объект живет в пространстве объектов, информацию защиты, а также список процессов, у которых есть дескрипторы к данному объекту. Каждый заголовок объекта также содержит поле цены квоты, представляющей собой плату, взимаемую с процесса за открытие объекта. Если файловый объект стоит один пункт, а процесс принадлежит к заданию, у которого есть 10 пунктов квоты, то суммарно все процессы этого задания могут открыть не более 10 файлов. Таким образом, для объектов каждого типа могут реализовываться ограничения на ресурсы.

Чтобы вовремя освободить занимаемое виртуальное адресное пространство, в заголовке объекта имеется счетчик ссылок на объект. При открытии объекта он увеличивается на единицу, при закрытии – уменьшается на единицу. Если его значение становится равным нулю, то объект удаляется.

Объекты подразделяются на типы. Тип объекта определяется указателем на объект типа. Фиксированного списка типов объектов не существует, но можно перечислить наиболее употребительные:

1. Процесс.
2. Поток.
3. Семафор.
4. Мьютекс.
5. Событие.
6. Порт.
7. Таймер.
8. Очередь.
9. Открытый файл.
10. Маркер доступа.
11. Профиль.
12. Секция.
13. Ключ.
14. Каталог объектов.
15. Символьная ссылка.
16. Устройство.
17. Драйвер устройства.

Для того чтобы менеджер объектов мог следить за созданием и удалением объектов, он поддерживает пространство имен, в котором располагаются все объекты системы. Пространство имен объектов является одним из трех пространств, поддерживаемых в Windows. Остальные два – это пространство имен файловой системы и пространство имен реестра. Все три пространства имеют вид иерархических каталогов с множеством вложенных подкаталогов.

12.3. Процессы и потоки в Windows

В операционной системе Windows поддерживаются традиционные процессы, способные общаться и синхронизироваться друг с другом так же, как это делают процессы в UNIX. Каждый процесс содержит, по крайней мере, один поток, содержащий, в свою очередь, как минимум одно волокно (облегченный поток). Более того, для управления определенными ресурсами процессы могут объединяться в задания. Все вместе – задания, процессы, потоки и волокна – образует общий набор инструментов для управления ресурсами и реализации параллелизма как на однопроцессорных, так и на многопроцессорных машинах. Краткое описание этих четырех понятий приведено в таблице 12.1.

Таблица 12.1. Основные понятия, используемые для управления центральным процессором и ресурсами

Название	Описание
Задание	Набор процессов с общими квотами и лимитами
Процесс	Контейнер для ресурсов
Поток	Сущность, планируемая ядром
Волокно	Облегченный поток, управляемый полностью в пространстве пользователя

Задание в Windows представляет собой набор, состоящий 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 (не позволяющее 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 для всего задания. Задания также могут ограничивать свои процессы в вопросах безопасности,

например, запрещать им получать права администратора (суперпользователя) даже при наличии правильного пароля.

Процессы являются более интересными, чем задания, а также более важными. Как и в системе UNIX, процессы представляют собой контейнеры для ресурсов. У каждого процесса есть 4-гигабайтное адресное пространство, в котором пользователь занимает нижние 2 Гбайт (в серверных версиях этот размер может быть по желанию увеличен до 3 Гбайт), а операционная система занимает остальную его часть. Таким образом, операционная система присутствует в адресном пространстве каждого процесса, хотя она и защищена от изменений с помощью аппаратного блока управления памятью MMU. У процесса есть идентификатор процесса, один или несколько потоков, список дескрипторов (управляемых в режиме ядра) и маркер доступа, хранящий информацию защиты. Процессы создаются с помощью вызова Win32, который принимает на входе имя исполняемого файла, определяющего начальное содержимое адресного пространства, и создает первый поток.

Каждый процесс начинается с одного потока, но новые потоки могут создаваться динамически. Потоки формируют основу планирования центрального процессора, так как операционная система всегда для запуска выбирает поток, а не процесс. Соответственно, у каждого потока есть состояние (готовый, работающий, блокированный и т.д.), тогда как у процессов состояний нет. Потоки могут динамически создаваться вызовом Win32, которому в адресном пространстве процесса задается адрес начала исполнения. У каждого потока есть идентификатор потока, выбираемый из того же пространства, что и идентификаторы процессов, поэтому один и тот же идентификатор никогда не будет использован одновременно для процесса и для потока.

Как правило, поток работает в пользовательском режиме, но когда он обращается к системному вызову, то переключается в режим ядра, после чего продолжает выполнять тот же поток, с теми же свойствами и ограничениями, которые были у него в режиме пользователя. У каждого потока есть два стека, один используется в режиме ядра, а другой в режиме пользователя. Помимо состояния, идентификатора и двух стеков, у каждого потока есть контекст (в котором сохраняются его регистры, когда он не работает), приватная область для локальных переменных, а также может быть свой собственный маркер доступа. Если у потока есть свой маркер доступа, то он перекрывает маркер доступа процесса, чтобы

клиентские потоки могли передать свои права доступа серверным потокам, выполняющим работу для них. Когда поток завершает свою работу, он может прекратить свое существование. Когда прекращает существование последний активный поток, процесс завершается.

Важно понимать, что потоки представляют собой концепцию планирования, а не концепцию владения ресурсами. Любой поток может получить доступ ко всем объектам его процесса. Все, что ему для этого нужно сделать, – это заполучить дескриптор и обратиться к соответствующему вызову Win32. Для потока нет никаких ограничений доступа к объекту, связанных с тем, что этот объект создан или открыт другим потоком. Система даже не следит за тем, какой объект каким потоком создан. Как только дескриптор объекта помещен в таблицу дескрипторов процесса, любой поток процесса может его использовать.

Помимо нормальных потоков, работающих в процессах пользователя, в операционной системе Windows есть множество процессов-демонов, не связанных ни с каким пользовательским процессом (они ассоциированы со специальной системой или простаивающими процессами). Некоторые демоны выполняют административные задачи, тогда как другие формируют пул, и ими могут пользоваться компоненты исполняющей системы или драйверы, которым нужно выполнить какие-либо асинхронные задачи в фоновом режиме.

Переключение потоков в операционной системе Windows занимает довольно много времени, так как для этого необходимо переключение в режим ядра, а затем возврат в режим пользователя. Для предоставления сильно облегченного псевдопараллелизма в Windows используются **волокна**, подобные потокам, но планируемые в пространстве пользователя создавшей их программой (или ее системой поддержки исполнения). У каждого потока может быть несколько волокон, так же как у процесса может быть несколько потоков. Когда волокно логически блокируется, оно помещается в очередь блокированных волокон, после чего для работы выбирается другое волокно в контексте того же потока.

Операционная система не знает о смене волокон, так как все тот же поток продолжает работу. Так как операционная система ничего не знает о волокнах, то с ними, в отличие от заданий, процессов и потоков, не связаны объекты исполняющей системы. Для управления волокнами нет и настоящих системных вызовов. Однако для этого есть вызовы Win32 API. Они относятся к тем вызовам Win32 API, которые не обращаются к системным вызовам.

В операционной системе Windows не поддерживается какой-либо иерархии процессов, например "родительский – дочерний". Все созданные процессы равны. Однако существует негласная иерархия, заключающаяся в том, кто чьим дескриптором владеет. Хотя эти дескрипторы не могут напрямую передаваться другим процессам, у процесса есть способ создать дубликат дескриптора. Дубликат дескриптора может быть передан другому процессу и использоваться им, поэтому неявная иерархия процессов может просуществовать недолго.

12.4. Межпроцессное взаимодействие в Windows

Для общения друг с другом потоки могут использовать широкий спектр возможностей, включая каналы, именованные каналы, почтовые ящики, вызов удаленной процедуры и совместно используемые файлы. Каналы могут работать в одном из двух режимов, выбираемом при создании канала: байтовом и режиме сообщений. Байтовые каналы работают так же, как и в системе UNIX. Каналы сообщений в чем-то похожи на байтовые каналы, но сохраняют границы между сообщениями, так что четыре записи по 128 байт будут читаться с другой стороны канала как четыре сообщения по 128 байт, а не как одно 512-байтовое сообщение, как это может случиться с байтовыми каналами. Также имеются именованные каналы, для которых существуют те же два режима. Именованные каналы, в отличие от обычных каналов, могут использоваться по сети.

Почтовые ящики представляют собой особенность системы Windows, которой нет в UNIX. В некоторых аспектах они подобны каналам, но не во всем. Во-первых, почтовые ящики являются однонаправленными, тогда как каналы могут работать в обоих направлениях. Они также могут использоваться по сети, но не предоставляют гарантированной доставки. Наконец, они позволяют отправляющему процессу использовать широковещание для рассылки сообщения не одному, а сразу многим получателям.

Сокеты подобны каналам с тем отличием, что они при нормальном использовании соединяют процессы на разных машинах. Например, один процесс пишет в сокет, а другой процесс на удаленной машине читает из него. Сокеты также могут использоваться для соединения процессов на одной машине, но поскольку их использование влечет за собой бо́льшие накладные расходы, чем использование каналов, то, как правило, они применяются в контексте сети.

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 можно легко реализовать общий буфер, применяемый в задаче производителя и потребителя.

12.5. Планирование

В операционной системе Windows нет центрального потока планирования. Вместо этого, когда какой-либо поток не может более выполняться, этот поток сам переходит в режим ядра и запускает планировщика, чтобы определить, на какой поток переключиться. Текущий поток выполняет программу планировщика при одном из следующих условий:

1) Поток блокируется на семафоре, мьютексе, событии, операции ввода-вывода и т. д.;

2) Поток сигнализирует каким-либо объектом (например, выполняет операцию *up* на семафоре);

3) Истекает квант времени работающего потока.

В первом случае поток уже работает в режиме ядра, чтобы выполнить операцию с объектом диспетчера или ввода-вывода. Возможно, он не может продолжать работу, поэтому он должен сохранить свой контекст, запустить программу планировщика, чтобы выбрать своего преемника, и загрузить контекст этого потока, чтобы запустить его.

Во втором случае поток также находится в ядре. Однако после сигнализирования объектом он, определенно, может продолжать работу, так как эта операция никогда не приводит к блокированию. Тем не менее, поток должен запустить процедуру планировщика, чтобы посмотреть, нет ли среди готовых к работе потока с более высоким приоритетом. Если такой поток есть, происходит переключение на этот поток, так как операционная система Windows является системой с приоритетным прерыванием (то есть переключение потока может произойти в любой момент, а не только тогда, когда у текущего потока закончится выделенный ему квант времени).

В третьем случае происходит эмулированное прерывание с передачей управления в ядро. При этом поток также запускает процедуру планировщика, чтобы определить, какой поток следует запустить после текущего потока. Если все

остальные потоки в данный момент окажутся заблокированными, планировщик может продолжить выполнение текущего потока, выделив ему новый квант времени. В противном случае происходит переключение потока.

Планировщик также вызывается при еще двух условиях:

1. Завершается операция ввода-вывода.
2. Истекает ожидание таймера.

В первом случае какой-нибудь поток, возможно, ожидал окончания этой операции ввода-вывода и теперь может продолжить свою работу. Необходимо определить, должен ли этот поток прервать выполнение текущего потока, так как потокам не гарантируется минимальный рабочий интервал времени. Планировщик не запускается во время работы самой процедуры обработки прерываний (так как при этом прерывания могут оказаться запрещенными на слишком долгий срок). Вместо этого отложенный вызов устанавливается в очередь и выполняется немного позднее, после того как процедура обработки прерываний закончит свою работу. Во втором случае поток выполнил операцию *down* на семафоре или блокировался на каком-либо другом объекте, но установленное время ожидания истекло. И в этом случае обработчик прерываний должен установить вызов в очередь, чтобы он не был запущен во время работы обработчика прерываний. Если в результате тайм-аута поток оказался готовым к работе, будет запущен планировщик, и если ничего более важного в данный момент нет, будет выполнен отложенный вызов процедуры.

Система планирования включает два вызова Win32 API, предоставляющих процессам возможность влиять на планирование потоками. Алгоритм планирования в большой степени определяется этими вызовами. Во-первых, есть вызов *SetPriorityClass*, устанавливающий класс приоритета всех потоков вызывающего процесса. К допустимым значениям приоритета относятся: реального времени, высокий, выше нормы, нормальный, ниже нормы и неработающий.

Во-вторых, имеется вызов *SetThreadPriority*, устанавливающий относительный приоритет некоторого потока по сравнению с другими потоками данного процесса. Приоритет может иметь следующие значения: критичный ко времени, самый высокий, выше нормы, нормальный, ниже нормы, самый низкий и неработающий. Таким образом, шесть классов процессов и семь классов потоков могут образовать 42 комбинации. Эта информация поступает на вход алгоритма планирования.

В системе существует 32 уровня приоритета, пронумерованные от 0 до 31. 42 комбинации отображаются на эти 32 приоритета по определенной схеме. Базовый алгоритм планирования состоит из процедуры сканирования массива от приоритета 31 до приоритета 0. Как только найден непустой элемент, выбирается поток в начале очереди и запускается на один квант времени. Когда квант истекает, поток направляется в конец очереди своего приоритета, а следующим выбирается поток в начале очереди. Другими словами, когда есть несколько готовых потоков с наивысшим уровнем приоритета, они запускаются поочередно, получая каждый по одному кванту времени. Если готовых потоков нет, запускается бездействующий поток.

Следует отметить, что при планировании не учитывается, какому процессу принадлежит тот или иной поток. То есть планировщик не выбирает сначала процесс, а затем поток в этом процессе. Он смотрит только на потоки. Он даже не знает, какой процесс владеет конкретным потоком. На многопроцессорной системе каждый центральный процессор сам занимается планированием своих потоков при помощи массива приоритетов. Чтобы гарантировать, что в каждый момент времени лишь один центральный процессор работает с массивом, используется спин-блокировка.

Приоритеты с 16 по 31 зарезервированы для самой системы и для потоков, которым такой высокий приоритет явно задаст системный администратор. Обычные пользователи не могут запускать потоки со столь высокими приоритетами, и существует веская причина для этого. Если бы пользовательский процесс мог работать с приоритетом более высоким, чем, скажем, поток клавиатуры или мыши, то длительная работа такого высокоприоритетного потока без операций ввода-вывода (например, в цикле) повесила бы всю систему.

Пользовательские потоки работают с приоритетами от 1 до 15. Устанавливая приоритеты процесса и потока, пользователь может отдавать преимущество тому или иному потоку. Нулевой поток работает в фоновом режиме и съедает все процессорное время, на которое больше никто не претендует. Его работа заключается в обнулении страниц для менеджера памяти. Если и у этого потока нет работы, работает пустой поток. Однако он не является полноценным потоком. При определенных условиях текущий приоритет пользовательского потока может быть поднят операционной системой выше базового приоритета, но никогда не может быть установлен выше приоритета 15.

Когда же увеличивается приоритет потока? Во-первых, когда завершается операция ввода-вывода и освобождается ожидающий ее поток, приоритет потока увеличивается, чтобы дать шанс этому потоку быстрее запуститься и снова запустить операцию ввода-вывода. Суть в том, чтобы поддерживать 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 прибавляется две единицы, если это поток переднего плана (то есть процесс, управляющий окном, которому в данный момент направляется ввод с клавиатуры), и одна единица в противном случае. Таким образом, интерактивный процесс получает преимущество перед большой толпой других процессов. Наконец, если поток графического интерфейса пользователя просыпается, потому что стал доступен оконный ввод, он также получает прибавку приоритета по той же самой причине.

12.6. Управление памятью в Windows

В операционной системе Windows у каждого пользовательского процесса есть собственное виртуальное адресное пространство. Виртуальные адреса 32-разрядные, поэтому у каждого процесса 4 Гбайт виртуального адресного пространства. Нижние 2 Гбайт за вычетом около 256 Мбайт доступны для программы и данных процесса; верхние 2 Гбайт защищенным образом отображаются на память ядра. Страницы виртуального адресного пространства имеют фиксированный размер (4 Кбайт) и подгружаются по требованию.

Конфигурация виртуального адресного пространства для трех пользовательских процессов в слегка упрощенном виде показана на рисунок 12.2. Белым цветом на рисунке изображена область приватных данных процесса. Затененные области представляют собой память, совместно используемую всеми процессами. Нижние и верхние 64 Кбайт каждого виртуального адресного пространства в обычном состоянии не отображаются на физическую память. Это делается преднамеренно, чтобы облегчить перехват программных ошибок. Недействительные указатели часто имеют значение 0 или –1, и попытки их использования в системе Windows вызовут немедленное прерывание вместо чтения или, что еще хуже, записи слова по неверному адресу. Однако когда запускаются

старые программы MS-DOS в режиме эмуляции, нижние 64 Кбайт могут отображаться на физическую память.

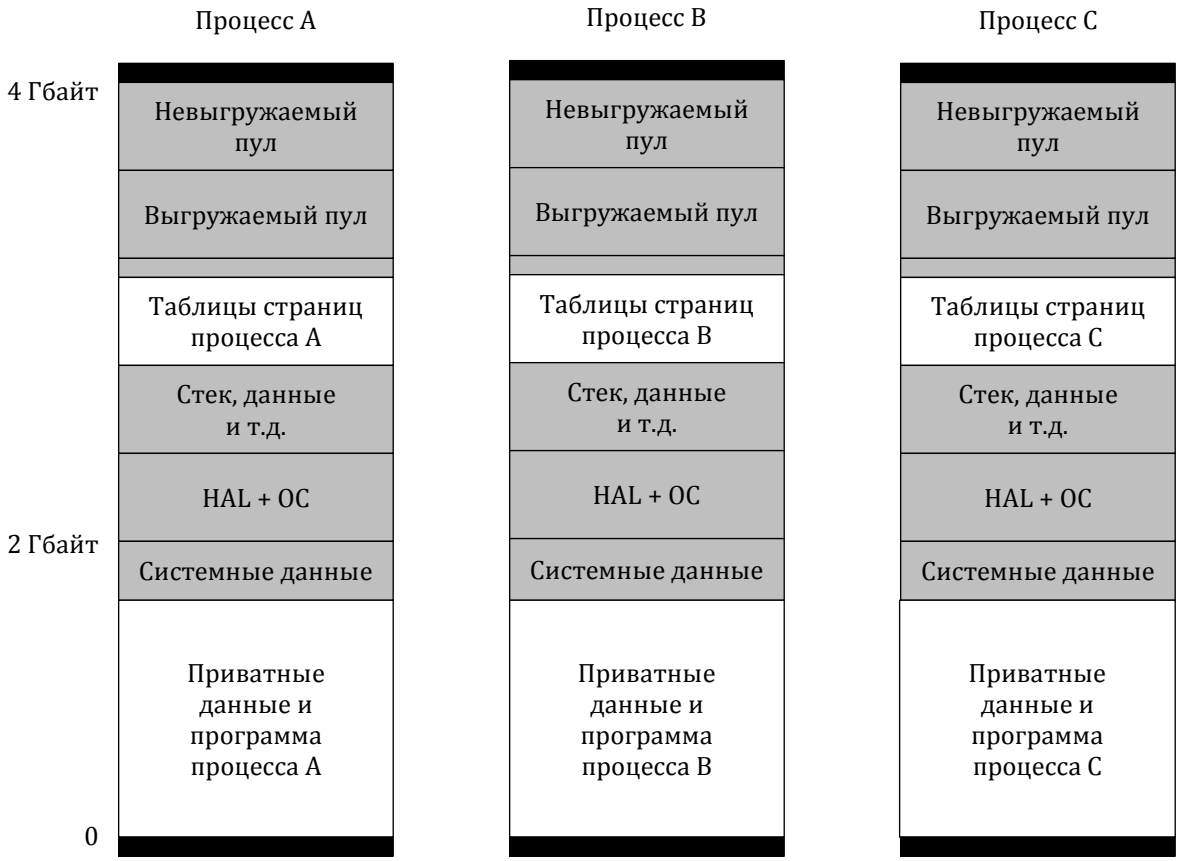


Рисунок 12.2 – Конфигурация виртуального адресного пространства для трех пользовательских процессов

Начиная с адреса 64 К, могут располагаться приватные данные и программа пользователя. Они могут занимать почти 2 Гбайт. Последний фрагмент этих 2 Гбайт памяти содержит некоторые системные указатели и таймеры, используемые совместно всеми пользователями в режиме доступа "только чтение". Отображение данных в эту область памяти позволяет всем процессам получать к ним доступ без лишних системных вызовов.

Верхние 2 Гбайт виртуального адресного пространства содержат операционную систему, включая код, данные и выгружаемый и невыгружаемый пулы (используемые для объектов и т.д.). Верхние 2 Гбайт используются совместно всеми процессами, кроме таблиц страниц, которые являются индивидуальными для каждого процесса. Верхние 2 Гбайт процессам в режиме пользователя запрещены для записи, а, по большей части, также запрещены и для чтения. Причина, по которой они размещаются здесь, заключается в том, что когда поток обращается к

системному вызову, он переключается в режим ядра, но остается все тем же потоком. Если сделать всю операционную систему и все ее структуры данных (как и весь пользовательский процесс) видимыми в адресном пространстве потока, когда он переключается в режим ядра, то отпадает необходимость в изменении карты памяти или выгрузке кэша при входе в ядро. Все, что нужно сделать, – это переключиться на стек режима ядра. Платой за более быстрые системные вызовы при данном подходе является уменьшение приватного адресного пространства для каждого процесса. Большим базам данных тесно в таких рамках, вот почему в серверных версиях Windows есть возможность использования 3 Гбайт для адресного пространства пользовательских процессов.

Каждая виртуальная страница может находиться в одном из трех состояний – свободном, зарезервированном и фиксированном. **Свободная страница** не используется в настоящий момент, и ссылка на нее вызывает страничное прерывание. Когда процесс запускается, все его страницы находятся в свободном состоянии, пока программа и исходные данные не будут отображены на их адресное пространство. Как только данные или программа отображаются на страницу, страница называется **фиксированной**. Обращение к фиксированной странице преобразуется при помощи аппаратного обеспечения виртуальной памяти и завершается успехом, если эта страница находится в оперативной памяти. В противном случае происходит страничное прерывание, операционная система находит требуемую страницу на диске и считывает ее в оперативную память.

Виртуальная страница может также находиться в **зарезервированном** состоянии, в таком случае эта страница не может отображаться, пока резервирование не будет явно удалено. Например, когда создается новый поток, в виртуальном адресном пространстве резервируется 1 Мбайт пространства для стека, но фиксируется только одна страница. Такая техника означает, что стек может вырасти до 1 Мбайт без опасения, что какой-либо другой поток захватит часть необходимого непрерывного виртуального адресного пространства. Помимо состояния (свободная, зарезервированная или фиксированная), у страниц есть также и другие атрибуты, например, страница может быть доступной для чтения, записи или исполнения.

При выделении фиксированным страницам места резервного хранения используется интересный компромисс. Простая стратегия в данном случае состояла бы в отведении для каждой фиксированной страницы одной страницы в файле

подкачки во время фиксации страницы. Это означало бы, что всегда есть место, куда записать каждую фиксированную страницу, если потребуется удалить ее из памяти. Недостаток такой стратегии заключается в том, что при этом может потребоваться файл подкачки размером со всю виртуальную память всех процессов.

Чтобы не тратить пространство на диске понапрасну, в Windows фиксированным страницам, у которых нет естественного места хранения на диске (например, страницам стека), не выделяются страницы на диске до тех пор, пока не настанет необходимость их выгрузки на диск. Такая схема усложняет систему, так как во время обработки страничного прерывания может понадобиться обращение к файлам, в которых хранится информация о соответствии страниц, а чтение этих файлов может вызвать дополнительные страничные прерывания. С другой стороны, для страниц, которые никогда не выгружаются, пространства на диске не требуется.

В Windows, как и во многих версиях UNIX, файлы могут отображаться напрямую на области виртуального адресного пространства (то есть занимать множество соседних страниц). После того как файл отображен на адресное пространство, он может читаться и писаться при помощи обычных команд обращения к памяти. При отображении файла на память версия файла, находящаяся в памяти, может отличаться от дисковой версии (вследствие записи в виртуальное адресное пространство). Однако когда отображение файла прекращается или файл принудительно выгружается на диск, дисковая версия снова приводится в соответствие с последними изменениями файла в памяти.

Два и более процессов могут одновременно отображать на свои виртуальные адресные пространства одну и ту же часть одного и того же файла. Читая и записывая слова памяти, процессы могут общаться друг с другом и передавать друг другу информацию с очень большой скоростью, так как копирование при этом не требуется. У различных процессов могут быть различные права доступа. Поскольку все процессы, использующие отображаемый на память файл, совместно используют одни и те же страницы, изменения, произведенные одним процессом, немедленно становятся видимыми для всех остальных процессов, даже если файл на диске еще не был обновлен. Также предпринимаются меры, благодаря которым процесс, открывающий файл для нормального чтения, видит текущие страницы в ОЗУ, а не устаревшие страницы с диска.

Следует отметить, что при совместном использовании двумя программами одного файла DLL может возникнуть проблема, если одна из программ изменит

статические данные файла. Если не предпринять специальных действий, то другой процесс увидит измененные данные, что, скорее всего, не соответствует намерениям этого процесса. Эта проблема решается таким способом: все отображаемые страницы помечаются как доступные только для чтения, хотя в то же время некоторые из них тайно помечаются как в действительности доступные и для записи. Когда к такой странице происходит обращение операции записи, создается приватная копия этой страницы и отображается на память. Теперь в эту страницу можно писать, не опасаясь задеть других пользователей или оригинальную копию диске. Такая техника называется **копированием при записи**.

Рассмотрим теперь **алгоритм замещения страниц** в Windows. Замена страниц происходит следующим образом. Система пытается поддерживать определенное количество свободных страниц в памяти, чтобы, когда произойдет страничное прерывание, свободная страница могла быть найдена немедленно, без необходимости сначала записать несколько других страниц на диск. В результате применения такой стратегии большинство страничных прерываний удовлетворяются при помощи всего одной дисковой операции (чтения страницы с диска), хотя иногда приходится выполнять две операции (запись на диск "грязной" страницы, после чего с диска читается требуемая страница).

Конечно, страницы, пополняющие список свободных страниц, должны откуда-то поступать. Поэтому настоящая работа алгоритма замещения страниц характеризуется тем, как эти страницы забираются у процессов и помещаются в список свободных страниц (в действительности существует четыре списка свободных страниц, но в данный момент для простоты будем считать, что это один список). Посмотрим теперь, как операционная система Windows освобождает страницы. Начнем с того, что в системе подкачки активно используется понятие рабочего набора. У каждого процесса (не у каждого потока) есть рабочий набор. Этот набор состоит из отображенных страниц, находящихся в памяти, при обращении к которым, следовательно, не происходит страничных прерываний. Размер и состав рабочего набора, естественно, меняются по мере работы процесса.

Рабочий набор каждого процесса описывается двумя параметрами: минимальным и максимальным размерами. Эти размеры не являются жесткими границами.

Процесс может иметь в памяти меньше страниц, чем значение нижней границы, или (при определенных обстоятельствах) больше установленного

максимума. Вначале эти границы одинаковы для каждого процесса, но они могут меняться со временем. Начальное значение минимума по умолчанию находится в диапазоне от 20 до 50 страниц, а начальное значение максимума по умолчанию находится в диапазоне от 45 до 345 страниц, в зависимости от общего объема оперативной памяти. Значения по умолчанию могут быть изменены системным администратором.

Если происходит страничное прерывание, а размер рабочего набора меньше минимального значения, то к рабочему набору добавляется страница, С другой стороны, если происходит страничное прерывание, а размер рабочего набора больше максимального значения, то из рабочего набора (но не из памяти) изымается страница, чтобы выделить место для новой страницы. Этот алгоритм означает, что в операционной системе Windows используется локальный алгоритм, не позволяющий процессу получить слишком много памяти, что предотвращает причинение процессами ущерба друг другу. Однако система пытается настроить эти параметры. Например, если она замечает, что один процесс слишком активно занимается подкачкой (а остальные процессы нет), система может увеличить значение максимального предела для рабочего набора; таким образом, алгоритм представляет собой смесь локальных и глобальных решений, Тем не менее, существует абсолютный предел размера рабочего набора: даже если в системе работает всего один процесс, он не может занять последние 512 страниц, чтобы оставить немного оперативной памяти для новых процессов.

Однако история на этом не заканчивается. Раз в секунду выделенный демон-поток ядра**,** называемый **менеджером балансового множества,** проверяет, достаточно ли в системе свободных страниц. Если свободных страниц меньше, чем нужно, он запускает **менеджер рабочих наборов,** который исследует рабочие наборы и освобождает дополнительные страницы. Менеджер рабочих наборов сначала определяет порядок, в котором нужно наследовать процессы. В первую очередь страницы отнимаются у больших процессов, которые бездействовали в течение долгого времени. В последнюю очередь рассматривается процесс переднего плана.

Затем менеджер рабочих наборов начинает исследование процессов в выбранном порядке. Если рабочий набор процесса в настоящий момент оказывается меньше своего нижнего предела или с момента последней инспекции число страничных прерываний у этого процесса было выше определенного уровня, то

страницы у него не отнимаются. В противном случае менеджер рабочих наборов отнимает у процесса одну или несколько страниц. Количество забираемых у процесса страниц довольно сложным образом зависит от общего объема ОЗУ, от того, насколько много требуется памяти текущим процессам, от того, как размер текущего рабочего набора соотносится с верхним и нижним пределами, а также от других параметров. Все страницы рассматриваются по очереди.

На однопроцессорной машине если бит обращений к странице сброшен, то счетчик, связанный со страницей, увеличивается на единицу. Если этот бит установлен в единицу, счетчик обнуляется. После сканирования из рабочего набора удаляются страницы с наибольшими значениями счетчика. Поток продолжает изучать процессы, пока он не высвободит достаточного количества страниц, после чего он останавливается. Если полный перебор всех процессов не привел к освобождению достаточного числа страниц, менеджер рабочих наборов начинает второй проход, на котором он уже "состригает" с процессов страницы более агрессивно, даже отнимая их при необходимости у процессов, размер рабочего набора которых меньше минимального.

На многопроцессорной системе алгоритм, основанный на проверке бита обращений, уже не работает, так как, хотя текущий центральный процессор не обращался в последнее время к данной странице, к ней могли обращаться другие центральные процессоры. Исследование же битов обращений всех центральных процессоров представляет собой слишком дорогое удовольствие. Поэтому бит обращений вообще не учитывается, а удаляются самые старые страницы.

Следует отметить, что с точки зрения процедуры замены страниц операционная система сама рассматривается как процесс. Она владеет страницами, и у нее также есть рабочий набор. Этот рабочий набор тоже может быть уменьшен. Однако некоторые части системы и невыгружаемый пул фиксированы в памяти и не могут выгружаться ни при каких обстоятельствах.

12.7. Ввод-вывод в Windows

Менеджер ввода-вывода родственен менеджеру plug-and-play. Основная идея механизма plug-and-play заключается в настраиваемой шине. Существует большое количество различных шин, поэтому менеджер plug-and-play может послать каждому разъему запрос и попросить устройство назвать себя. Определив, что за устройство подключено к шине, менеджер plug-and-play выделяет для него

аппаратные ресурсы, такие как уровни прерываний, находит необходимые драйверы и загружает их в память. При загрузке каждого драйвера для него создается **объект драйвера**. Для некоторых шин, например SCSI, настройка происходит только во время загрузки операционной системы. Для других шин, таких как USB и IEEE 1394, она может производиться в любой момент, для чего требуется тесный контакт между менеджером plug-and-play, драйвером шины (который и выполняет настройку) и менеджером ввода-вывода.

Менеджер ввода-вывода также тесно связан с менеджером энергопотребления. Менеджер энергопотребления может перевести компьютер в одно из шести состояний. Устройства ввода-вывода также могут находиться в различных состояниях. Включением и выключением этих устройств занимаются вместе менеджер энергопотребления и менеджер ввода-вывода.

Формально все файловые системы представляют собой драйверы ввода-вывода. Обращения к блокам диска от пользовательских процессов сначала посылаются менеджеру кэша. Если менеджер кэша не может удовлетворить запрос из кэша, он просит менеджер ввода-вывода вызвать драйвер соответствующей файловой системы, чтобы тот получил требуемый блок с диска.

Интересная особенность операционной системы Windows заключается в поддержке **динамических дисков**. Эти диски могут распространяться на несколько дисковых разделов или даже физических дисков и их можно переконфигурировать на лету, без перезагрузки. Таким образом, логические тома более не привязаны к отдельному разделу или даже одному жесткому диску, что позволяет получить файловую систему, располагающуюся на нескольких дисках прозрачным для пользователя способом.

Другой интересный аспект Windows заключается в поддержке асинхронного ввода-вывода. Поток может начать операцию ввода-вывода, а затем продолжить выполнение параллельно с вводом-выводом. Такая возможность особенно важна для серверов. Существует множество способов, с помощью которых поток может определить, что операция ввода-вывода завершена. Один из способов состоит в создании в момент обращения к вызову ввода-вывода объекта события, а потом ожидания этого события. Другой способ заключается в указании очереди, в которую будет послано сообщение о завершении операции ввода-вывода. Третий способ заключается в предоставлении процедуры обратного вызова, к которой обращается система, когда операция ввода-вывода завершена.

Чтобы гарантировать, что драйверы устройств хорошо работают с остальной частью системы Windows, корпорация Microsoft определила для драйверов модель **Windows Driver Model**, которой драйверы устройств должны соответствовать. Более того, корпорация Microsoft также предоставляет набор инструментов, который должен помочь разработчикам в создании драйверов, соответствующих модели Windows Driver Model. Согласующиеся с WDM драйверы должны удовлетворять всем следующим требованиям (а также некоторым другим):

1. Обрабатывать входящие запросы ввода-вывода, поступающие в стандартном формате.

Запросы ввода-вывода передаются драйверам в виде стандартизированных пакетов, называемых **IRP** (Input/output Request Packet – пакет запроса ввода-вывода). Драйверы, согласующиеся с моделью WDM, должны уметь обрабатывать пакеты IRP.

2. Основываться на объектах, как и остальная часть системы Windows.

Драйвер должен поддерживать работу с объектами, то есть поддерживать определенный список методов, к которым может обращаться остальная система. Он также должен корректно работать с другими объектами операционной системы Windows, доступ к которым осуществляется при помощи дескрипторов объектов.

3. Позволять динамическое добавление или удаление устройств plug-and-play.

Драйверы, согласующиеся с моделью WDM, должны полностью поддерживать устройства plug-and-play. Это означает, что если устройство, управляемое драйвером, внезапно добавляется в систему или удаляется из системы, драйвер должен быть готов к получению данной информации и корректной реакции на эту информацию, даже в том случае, если устройство удаляется в момент обращения к нему.

4. Допускать, когда это возможно, управление энергопотреблением.

Также драйверы должны поддерживать управление энергопотреблением для тех устройств, для которых это возможно. Например, если система решает, что теперь пора перейти в режим низкого энергопотребления, все драйверы должны поддерживать этот режим, чтобы сберегать энергию. Они также должны поддерживать обратный переход в режим нормального функционирования.

5. Допускать реконфигурацию в терминах использования ресурсов.

Драйвер должен быть настраиваемым, что означает отсутствие каких бы то ни было встроенных предположений о линиях прерываний или портах ввода-вывода, используемых определенным устройством. Например, на компьютерах IBM PC и сменивших их моделях порт принтера более 20 лет имел адрес 0x378 и вряд ли будет изменен теперь. Но драйвер принтера, в который этот адрес жестко зашит, не является согласующимся с моделью WDM.

6. Быть реентерабельными для возможности их использования на мультипроцессорах, то есть два процесса могут использовать одни и те же инструкции.

Драйверы устройств также должны работать на мультипроцессорах, так как поддержка мультипроцессоров была заложена в операционную систему Windows при разработке. Это требование означает, что во время обработки драйвером запроса от одного центрального процессора может прийти запрос от другого центрального процессора. Второй центральный процессор может начать выполнение программы драйвера одновременно с первым центральным процессором. Драйвер должен функционировать корректно, даже когда он вызывается одновременно двумя и более центральными процессорами. Это означает, что доступ ко всем чувствительным структурам данных должен предоставляться только внутри критических областей. Простое предположение, что других обращений к драйверу не будет, пока не завершится обработка текущего обращения к нему, недопустимо.

7. Обладать переносимостью между различными версиями операционных систем Windows.

Каждый драйвер должен поставлять набор процедур, которые могут быть вызваны для получения требуемого обслуживания. Первая процедура, называемая *DriverEntry*, инициализирует драйвер. Она вызывается сразу после загрузки драйвера. Процедура может создавать таблицы и структуры данных, но не должна обращаться к самому устройству. Она также заполняет некоторые поля объекта драйвера, созданного менеджером ввода-вывода при загрузке драйвера. Поля в объекте драйвера включают указатели на все остальные процедуры, предоставляемые драйвером. Кроме того, для каждого устройства, управляемого драйвером (например, для каждого диска IDE, управляемого драйвером диска IDE), создается **объект устройства** и инициализируется так, чтобы он указывал на объект драйвера. Эти объекты драйверов помещаются в специальный каталог \??.

При наличии объекта устройства можно легко найти объект драйвера и, таким образом, обращаться к его методам.

Вторая процедура драйвера называется *AddDevice*. Она вызывается (менеджером plug-and-play) всего один раз для каждого добавляемого устройства. После этого драйвер вызывается первым пакетом IRP, который устанавливает вектор прерываний и инициализирует аппаратуру. Кроме того, драйвер должен содержать процедуру обработки прерываний, различные процедуры, управляющие таймерами, путь быстрого ввода-вывода, управление DMA, позволять прервать исполняющийся текущий запрос и многое другое.

В операционной системе Windows драйвер должен сам выполнять всю работу, как, например, выполняет ее драйвер принтера на рисунке 12.3.

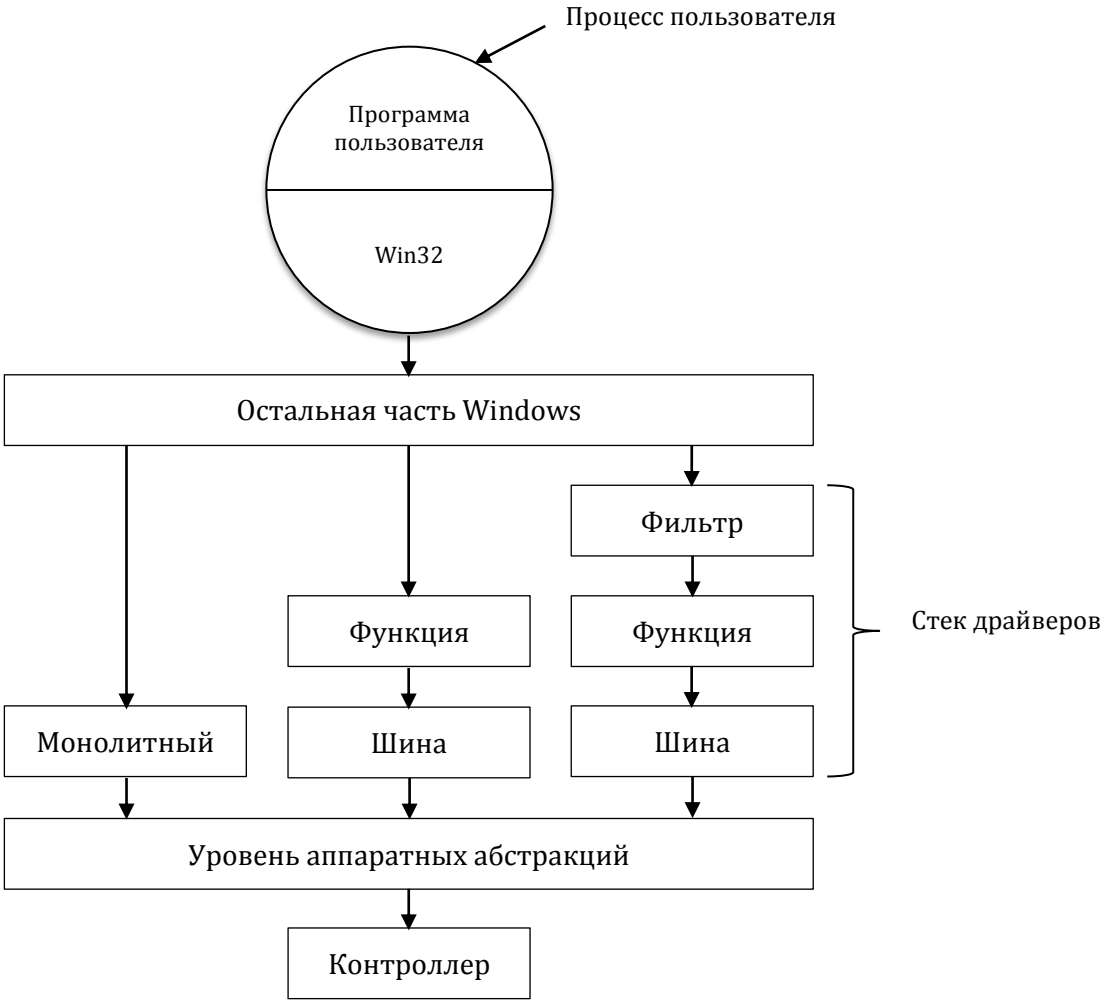


Рисунок 12.3 – Работа драйвера в системе Windows

Это означает, что запрос может проходить через целую последовательность драйверов, каждый из которых выполняет свою часть работы. Два таких драйвера также показаны на этом рисунке.

Стеки драйверов позволяют отделить управление шиной от управления собственно устройством. Управление шиной PCI отличается большой сложностью, что вызвано большим количеством режимов и транзакций шины. Таким образом, отделение управления шиной от управления устройством, подключенным к данной шине, облегчает работу по созданию драйвера. Программисту, пишущему драйвер устройства, более не нужно изучать вопрос управления шиной. Он может просто использовать стандартный драйвер шины, находящийся в стеке драйверов. У драйверов USB и SCSI есть части, специфичные для конкретных устройств, и общая часть, для которой используются отдельные драйверы.

Кроме того, стеки драйверов позволяют добавлять в стек **драйверы-фильтры**. Фильтрующий драйвер выполняет некоторые преобразования проходящих через них данных. Например, фильтрующий драйвер может сжать данные по пути к диску или зашифровать их по дороге в сеть. Помещение драйверного фильтра в стек драйверов означает, что ни прикладная программа, ни настоящий драйвер устройства не должны знать о присутствии фильтрующего драйвера и что фильтрующий драйвер работает автоматически для всех данных, поступающих с устройства или на устройство.

12.8. Файловая система Windows

Операционная система Windows поддерживает несколько файловых систем, самыми важными из которых являются FAT-16, FAT-32 и NTFS (New Technology File System – файловая система новой технологии). Файловая система FAT-16 – это старая файловая система MS-DOS. В ней используются 16-разрядные дисковые адреса, что ограничивает размер дискового раздела двумя гигабайтами. В файловой системе FAT-32 используются 32-разрядные дисковые адреса и поддерживаются дисковые разделы размером до 2 Тбайт. Система NTFS представляет собой новую файловую систему, разработанную специально для Windows NT и перенесенную в Windows 2000 и последующие версии. В ней используются 64-разрядные дисковые адреса, таким образом, теоретически эта файловая система может поддерживать дисковые разделы размером до 2^{64} байт, хотя по другим техническим причинам их размер ограничен меньшими размерами. Операционной системой Windows также поддерживаются файловые системы для CD-ROM, DVD и BR-D , в которых разрешено только чтение. Одна и та же работающая операционная система может одновременно иметь доступ к нескольким файловым системам различного типа.

Длина имени файла в системе NTFS ограничена 255 символами; полная длина пути ограничивается 32 767 символами. Для имен файлов используется кодировка Unicode, что позволяет пользователям в странах, в которых не используется латинский алфавит (например, в Греции, Японии, Индии, России и Израиле), писать имена файлов на своем родном языке.

Файловая система NTFS полностью поддерживает имена, чувствительные к регистру, однако интерфейс Win32 API поддерживает их не полностью, поэтому это преимущество теряется при обращении к программам, обязанным использовать интерфейс Win32 (например, для совместимости с Windows 98).

Файл в системе NTFS – это не просто линейная последовательность байтов, как файлы в системах FAT-32 и UNIX. Вместо этого файл состоит из множества атрибутов, каждый из которых представляется в виде потока байтов. Большинство файлов имеет несколько коротких потоков, таких как имя файла и его 64-битовый идентификатор, плюс один длинный (неименованный) поток с данными. Однако у файла может быть и несколько длинных потоков данных. При обращении к каждому потоку после имени файла через двоеточие указывается имя потока, например *foo:stream1*. У каждого потока своя длина. Каждый поток может блокироваться независимо от остальных потоков. Идея нескольких потоков позаимствована у системы Apple Macintosh, в которой файлы имеют по два потока, ветвь данных и ветвь ресурса. Эта концепция была добавлена в файловую систему NTFS, чтобы сервер с системой NTFS мог обслуживать клиенты Macintosh.

Файловые потоки могут использоваться не только для совместимости с Macintosh. Например, программа редактирования фотографий может использовать неименованный поток для основного изображения, а именованный поток – для небольшой пиктограммы. Эта схема проще, чем традиционный способ, при котором изображения помещаются в один и тот же файл, одно за другим. Другой пример использования потоков данных – электронная обработка текста. Эти программы часто создают две версии документа, временную для использования во время редактирования и окончательную версию, когда пользователь закончил работу. Если поместить временную версию в именованный поток, а окончательную версию в неименованный поток, обе версии автоматически оказываются в одном файле и без какой-либо дополнительной обработки пользуются одинаковыми правами доступа, временными штампами и т.д.

Рассмотрим структуру файловой системы. Каждый том NTFS (то есть дисковый раздел) содержит файлы, каталоги, битовые массивы и другие структуры данных. Каждый том организован как линейная последовательность блоков (кластеров по терминологии Microsoft). Размер блока фиксирован для каждого тома и варьируется в пределах от 512 байт до 64 Кбайт, в зависимости от размера тома. Для большинства дисков NTFS используются блоки размером в 4 Кбайт, как компромисс между большими блоками (для эффективности операций чтения/записи) и маленькими блоками (для уменьшения потерь дискового пространства на внутреннюю фрагментацию). Обращение к блокам осуществляется по их смещению от начала тома, для которого используются 64-разрядные числа.

Главной структурой данных в каждом томе является **главная файловая таблица MFT** (Master File Table), представляющая собой линейную последовательность записей фиксированного (1 Кбайт) размера. Каждая запись MFT описывает один файл или один каталог. В ней содержатся атрибуты файла, такие как его имя и временные штампы, а также список дисковых адресов, указывающих на расположение блоков файла. Если файл очень большой, то иногда бывает необходимо использовать две и более записи главной файловой таблицы, чтобы вместить список всех блоков файла. В этом случае первая запись MFT, называемая **базовой записью**, указывает на другие записи MFT.

Сама главная файловая таблица представляет собой файл и, как и любой файл, может располагаться в любом месте тома, тем самым устраняется проблема дефектных секторов на первой дорожке дискового раздела. Кроме того, этот файл может при необходимости расти до максимального размера в 2^{48} записей.

Каждая запись MFT состоит из последовательности пар (заголовок атрибута, значение). Каждый атрибут начинается с заголовка, идентифицирующего этот атрибут и сообщающего длину значения, так как некоторые атрибуты, например имя файла или данные, могут иметь переменную длину. Если значение атрибута достаточно короткое, чтобы поместиться в запись MFT, оно помещается туда. Если же это значение слишком длинное, оно располагается в другом месте диска, а в запись MFT помещается указатель на него.

Первые 16 записей MFT зарезервированы для файлов метаданных NTFS. Каждая запись описывает нормальный файл, у которого есть атрибуты и блоки данных, как у любого файла. У каждого такого файла есть имя, начинающееся с символа доллара, указывающего на то, что это файл метаданных. Первая запись

описывает сам файл MFT. В частности, она содержит информацию о расположении блоков файла MFT, что позволяет системе найти файл MFT. Очевидно, чтобы найти всю остальную информацию о файловой системе, у операционной системы Windows должен быть некий способ нахождения первого блока файла MFT. Номер первого блока файла MFT содержится в загрузочном блоке, куда он помещается при установке системы.

Следующая запись представляет собой дубликат первой части файла MFT. Эта информация является настолько ценной, что наличие второй копии может быть необходимо на случай, если один из первых блоков главной файловой таблицы вдруг станет дефектным. Далее следует журнал. Когда в файловой системе производятся структурные изменения, такие как добавление нового каталога или удаление существующего каталога, информация о предстоящей операции регистрируется в журнале. Таким образом, увеличивается вероятность корректного восстановления файловой системы в случае сбоя во время выполнения операции. Изменения атрибутов файлов также регистрируются здесь. В этом журнале не регистрируются только изменения данных пользователя. Далее записывается информация о томе, например его размер, метка и версия. Далее следуют определения атрибутов, сведения о корневом каталоге и другие метаданные, после которых идут пользовательские файлы.

Записи MFT для каталогов разного размера различны. В записи для небольших каталогов содержится несколько каталоговых записей, каждая из которых описывает файл или каталог. Поиск файла в каталоге по имени состоит в последовательном переборе всех имен файлов.

Для больших каталогов используется другой формат. Вместо того чтобы линейно перечислять файлы, используется дерево, обеспечивающее поиск в алфавитном порядке и упрощающее добавление в каталог новых имен в соответствующие места.

Одной из особенностей файловой системы NTFS является поддержка прозрачного сжатия файлов. Файл может быть создан в сжатом режиме. Это означает, что файловая система NTFS будет автоматически пытаться сжать блоки этого файла при записи их на диск и автоматически распаковывать их при чтении. Процессы, читающие этот файл или пишущие в него, не будут даже догадываться о том, что при этом происходит компрессия или декомпрессия данных.

Сжатие данных файла происходит следующим образом. Когда файловая система NTFS записывает на диск файл, помеченный для сжатия, она изучает первые 16 (логических) блоков файла, независимо от того, сколько сегментов на диске они занимают. Затем к этим блокам применяется алгоритм сжатия. Если полученные на выходе блоки могут поместиться в 15 или менее блоков, то сжатые данные записываются на диск, предпочтительно в виде одного сегмента. Если получить выигрыш хотя бы в один блок не удается, то данные 16 блоков так и записываются в несжатом виде. Затем весь алгоритм повторяется для следующих 16 блоков и т.д.

Произвольный доступ к сжатому файлу возможен, но не совсем прост. Как файловой системе NTFS найти определенный блок в сжатом файле? Ответ состоит в том, что для этого сначала потребуется прочитать и распаковать весь сегмент файла. После этого система может определить, где находится нужный блок, и передать его читающему процессу. Сжатие файла частями именно по 16 блоков явилось компромиссом. Если бы файл сжимался меньшими порциями, эффективность сжатия снизилась бы. Выбор больших размеров сжимаемых фрагментов привел бы к замедлению произвольного доступа к блокам.

Еще одной особенностью файловой системы NTFS является шифрование файлов. Сегодня компьютеры используются для хранения самых разнообразных конфиденциальных данных, включая планы слияния корпораций, налоговую информацию и любовную переписку. Владельцы подобных данных, как правило, не желают, чтобы она попала в посторонние руки. Информация может оказаться утрачена, например, при потере или краже переносного компьютера. Настольный компьютер можно загрузить с гибкого диска с системой MS-DOS, чтобы обойти систему безопасности Windows. Наконец, жесткий диск можно просто вынуть из одного компьютера и установить на другой компьютер.

В операционной системе Windows эти проблемы решаются при помощи возможности шифрования файлов. В результате применения шифрования, даже если компьютер будет украден или перезагружен в системе MS-DOS, файлы останутся нечитаемыми. Чтобы использовать шифрование в операционной системе Windows, нужно пометить каталог как зашифрованный, в результате чего будут зашифрованы все файлы в этом каталоге, а все новые файлы, перемещенные в этот каталог или созданные в нем, также будут зашифрованы. Само шифрование и дешифрование выполняется не файловой системой NTFS, а специальным драйвером EFS (Encrypting File System — шифрующая файловая система), размещающимся

между NTFS и пользовательским процессом. Таким образом, прикладная программа не знает о шифровании, а сама файловая система NTFS только частично вовлечена в этот процесс.

Познакомимся теперь с тем, как шифруются файлы в операционной системе Windows. Когда пользователь сообщает системе, что хочет зашифровать определенный файл, формируется случайный 128-разрядный ключ. Ключ используется для поблочного шифрования файла с помощью симметричного алгоритма, параметром в котором используется этот ключ. Каждый новый шифруемый файл получает новый случайный 128-разрядный ключ, так что никакие два файла не используют один и тот же ключ шифрования, что увеличивает защиту данных в случае, если какой-либо из ключей окажется скомпрометированным. Независимое шифрование каждого блока файла необходимо для сохранения возможности произвольного доступа к блокам файла.

Чтобы файл мог быть впоследствии расшифрован, ключ файла должен где-то храниться. Если бы ключ хранился на диске в открытом виде, тогда злоумышленник, укравший файлы, мог бы легко найти его и воспользоваться им для расшифровки украденных файлов. В этом случае сама идея шифрования файлов оказалась бы бессмысленной. Поэтому ключи файлов сами должны храниться на диске в зашифрованном виде. Для этого используется шифрование с открытым ключом.

После того как файл зашифрован, система с помощью информации в системном реестре ищет расположение открытого ключа пользователя. Открытый ключ можно без каких-либо опасений хранить прямо в реестре, так как по открытому ключу невозможно определить закрытый ключ, необходимый для расшифровки файлов. Затем случайный 128-разрядный ключ файла шифруется открытым ключом, а результат сохраняется на диске.

Чтобы расшифровать файл, с диска считывается зашифрованный случайный 128-разрядный ключ файла. Однако для его расшифровки необходим закрытый ключ. В идеале этот ключ должен храниться на смарт-карте, вне компьютера, и вставляться в считывающее устройство только тогда, когда требуется расшифровать файл. Хотя операционная система Windows поддерживает смарт-карты, она не позволяет хранить на них закрытые ключи.

Вместо этого, когда пользователь в первый раз зашифровывает файл с помощью системы EFS, операционная система Windows формирует пару ключей (закрытый ключ, открытый ключ) и сохраняет закрытый ключ, зашифрованный при

помощи симметричного алгоритма шифрования, на диске. Ключ для этого симметричного алгоритма формируется либо из пароля пользователя для регистрации в системе, либо из ключа, хранящегося на смарт-карте, если регистрация при помощи смарт-карты разрешена. Таким образом, система EFS может расшифровать закрытый ключ во время регистрации пользователя в системе и хранить его в своем виртуальном адресном пространстве во время работы, чтобы иметь возможность расшифровывать 128-разрядные ключи файлов без дополнительного обращения к диску. Когда компьютер выключается, закрытый ключ стирается из виртуального адресного пространства системы EFS, так что никто, даже украв компьютер, не получит доступа к закрытому ключу.

12.9. Безопасность в Windows

Рассмотрим теперь систему безопасности в Windows в целом. Операционная система Windows NT была разработана так, чтобы соответствовать уровню C2 требований безопасности Министерства обороны США. Этот стандарт требует наличия у операционных систем определенных свойств, позволяющих относить данные системы к достаточно надежным для выполнения военных задач определенного рода. Хотя при разработке новых версий операционной системы Windows не ставилось особой цели соответствия требованиям уровня C2, она унаследовала множество свойств безопасности от NT, включая следующие:

1. Безопасная регистрация в системе с мерами предосторожности против попыток применения фальшивой программы регистрации.

Безопасная регистрация означает, что системный администратор может потребовать ото всех пользователей наличия пароля для входа в систему. Программа, имитирующая регистрацию в системе, использовалась ранее на некоторых системах злоумышленниками с целью выведать пароль пользователя. Такая программа запускалась в надежде, что пользователь сядет за компьютер и введет свое имя и пароль. Имя и пароль записывались на диск, после чего пользователю сообщалось, что в регистрации ему отказано. В операционной системе Windows подобный обман пользователя невозможен, так как пользователь для входа в систему должен нажать комбинацию клавиш CTRL+ALT+DEL. Эта комбинация клавиш всегда перехватывается драйвером клавиатуры, который вызывает при этом настоящую программу регистрации. Пользовательский процесс

не может сам перехватить эту комбинацию клавиш или отменить ее обработку драйвером.

2. Дискреционное управление доступом.

Дискреционное управление доступом позволяет владельцу файла или другого объекта указать, кто может пользоваться объектом и каким образом.

3. Управление привилегированным доступом.

Средства управления привилегированным доступом позволяют системному администратору (суперпользователю) получать доступ к объекту, несмотря на установленные его владельцем разрешения доступа.

4. Защита адресного пространства для каждого процесса.

Под защитой адресного пространства имеется в виду лишь то, что у каждого процесса есть собственное защищенное виртуальное адресное пространство, недоступное для любого неавторизованного процесса.

5. Обнуление страниц перед выделением их процессу.

Это означает, что при увеличении стека выделяемые для него страницы заранее обнуляются, так что процесс не может обнаружить в них информации, помещенной предыдущим владельцем страницы памяти.

6. Аудит безопасности.

Его следует понимать как регистрацию системой в журнале определенных событий, относящихся к безопасности. Впоследствии этот журнал может просматривать системный администратор.

Рассмотрим, как функционирует система безопасности в Windows.

У каждого пользователя (и группы) операционной системы Windows есть идентификатор безопасности SID (Security IDentifier), по которому операционная система отличает его от других пользователей. Идентификаторы безопасности представляют собой двоичные числа с коротким заголовком, за которым следует длинный случайный компонент. Каждый SID должен быть уникален в пределах всей планеты. Когда пользователь запускает процесс, этот процесс и его потоки работают под идентификатором пользователя. Большая часть системы безопасности спроектирована так, чтобы гарантировать предоставление доступа к каждому объекту только потокам с авторизованными идентификаторами безопасности.

У каждого процесса есть маркер доступа, в котором указывается SID и другие свойства. Когда пользователь регистрируется в системе, процесс *winlogon* назначает маркер доступа начальному процессу. Последующие процессы, как правило,

наследуют этот маркер. Маркер доступа процесса изначально применяется ко всем потокам процесса. Однако поток во время исполнения может получить другой маркер доступа. В этом случае маркер доступа потока перекрывает маркер доступа процесса. В частности, клиентский поток может передать свой маркер доступа серверному потоку, чтобы сервер мог получить доступ к защищенным файлам и другим объектам клиента. Такой механизм называется **перевоплощением**.

Другим основным понятием является **дескриптор защиты**. У каждого объекта есть ассоциированный с ним дескриптор защиты, содержащий список пользователей и групп, имеющих доступ к данному объекту.

Как только пользователь регистрируется в системе, выполняется операция защиты при открытии объекта. Во время обработки процедуры открытия объекта менеджер безопасности проверяет наличие у вызывающего процесса соответствующих прав доступа. Для этого он просматривает все маркеры доступа вызывающего процесса, а также список DACL (разграничительного контроля доступа), ассоциированный с объектом. Он просматривает по очереди элементы списка. Как только он находит запись, соответствующую идентификатору SID вызывающего процесса, поиск прав доступа считается законченным. Если вызывающий процесс обладает необходимыми правами, объект открывается, в противном случае в открытии объекта отказывается.

Помимо разрешающих записей списки DACL могут также содержать запрещающие записи. Поскольку менеджер безопасности прекращает поиск, наткнувшись на первую запись с указанным идентификатором, запрещающие записи помещаются в начало списка DACL, чтобы пользователь, которому строго запрещен доступ к какому-либо объекту, не смог получить его как член какой-либо группы, которой этот доступ предоставлен.

После того как объект открыт, дескриптор объекта возвращается вызывающему процессу. При последующих обращениях проверяется только, входит ли данная операция в число операций, разрешенных в момент открытия объекта, чтобы, например, не допустить записи в файл, открытый для чтения.

ЗАКЛЮЧЕНИЕ

Учебное пособие "Операционные системы ЭВМ" позволяет изучить основные принципы устройства и организации операционных систем для персональных компьютеров. В пособии рассмотрены как общие моменты, так и частные особенности таких широко распространенных операционных систем, как UNIX и Windows семейства NT. В пособии рассмотрены следующие темы:

1. Основные понятия.
2. Процессы и потоки.
3. Межпроцессное взаимодействие.
4. Взаимоблокировки.
5. Управление памятью.
6. Ввод и вывод.
7. Файловые системы.
8. Мультимедийные операционные системы.
9. Многопроцессорные системы.
10. Безопасность.
11. Операционные системы семейства UNIX.
12. Операционные системы Windows семейства NT.

В результате изучения данного учебного пособия студенты получают теоретические знания, необходимые для активного использования и администрирования операционных систем, а также для создания системного программного обеспечения. Практическая часть курса будет рассмотрена в соответствующем учебно-методических пособиях по выполнению практических и лабораторных работ.

ЛИТЕРАТУРА

1. Танненбаум Э. Современные операционные системы. 3-е изд. – СПб.:Питер, 2010. – 1120 с.: ил. – (Серия "Классика Computer Science").

2. Олифер В.Г., Олифер Н.А. Сетевые операционные системы. – СПб.: Питер, 2002. – 544 с.: ил.

3. Руссинович М., Соломон Д. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс. / Пер. с англ. – 4-е изд. – М.: Издательско-торговый дом "Русская Редакция"; СПб.: Питер, 2005. – 992 с.: ил.