ТЕМА 1

ПОНЯТИЕ ОПЕРАЦИОННОЙ СИСТЕМЫ. ЭВОЛЮЦИЯ И КЛАССИФИКАЦИЯ ОПЕРАЦИОННЫХ СИСТЕМ

1.1 ПОНЯТИЕ ОПЕРАЦИОННОЙ СИСТЕМЫ

Облик всей вычислительной системы в целом в наибольшей степени определяет операционная система. Затруднения при попытке дать определение операционной системе частично связаны с тем, что *ОС выполняет две по существу мало связанные функции*:

1. *обеспечение пользователю-программисту удобств посредством предоставления для него расширенной машины;*
2. *повышение эффективности использования компьютера путем рационального управления его ресурсами.*

1.1.1 ОС как расширенная машина.

Использование большинства компьютеров на уровне машинного языка вызывает значительные затруднения, особенно это относится к вводу-выводу. Например, для организации чтения блока данных с гибкого диска программист может использовать 16 различных команд, каждая из которых требует 13 параметров, таких как номер блока на диске, номер сектора на дорожке и т. п. Когда выполнение операции с диском завершается, контроллер возвращает 23 значения, отражающих наличие и типы ошибок, которые необходимо проанализировать.

Среди программистов найдется не много желающих непосредственно заниматься программированием таких операций. При работе с диском программисту-пользователю достаточно представлять его в виде набора файлов, каждый из которых имеет имя. Работа с файлом заключается в его открытии, выполнении чтения или записи, а затем в закрытии файла. Вопросы подобные таким, как следует ли при записи использовать усовершенствованную частотную модуляцию или в каком состоянии сейчас находится двигатель механизма перемещения считывающих головок, не должны волновать пользователя.

Операционная система - программа, которая скрывает от программиста все реалии аппаратуры и предоставляет возможность простого, удобного просмотра указанных файлов, чтения или записи.

Точно так же, как ОС ограждает программистов от аппаратуры дискового накопителя и предоставляет ему простой файловый интерфейс, операционная система берет на себя все неприятные обязанности, связанные с обработкой прерываний, управлением таймерами и оперативной памятью, а также другие низкоуровневые проблемы. Абстрактная, воображаемая машина, с которой, благодаря операционной системе, может теперь иметь дело пользователь, гораздо проще и удобнее в обращении, чем реальная аппаратура, лежащая в основе этой абстрактной машины.

С этой точки зрения *функцией ОС является предоставление пользователю некоторой расширенной или виртуальной машины, которую легче программировать и с которой легче работать, чем непосредственно с аппаратурой, составляющей реальную машину.*

Идея о том, что ОС - это система, обеспечивающая удобный интерфейс пользователям, соответствует рассмотрению сверху вниз.

1.1.2 ОС как система управления ресурсами

При взгляде снизу вверх, - ОС это некоторый механизм, управляющем всеми частями сложной системы. Современные вычислительные системы состоят из процессоров, памяти, таймеров, дисков, сетевых коммуникационной аппаратуры, принтеров и других устройств. В соответствии со вторым подходом *функцией ОС является распределение процессоров, памяти, устройств и данных между процессами, конкурирующими за эти ресурсы. ОС должна управлять всеми ресурсами вычислительной машины таким образом, чтобы обеспечить максимальную эффективность ее функционирования.* Например, критерием эффективности может быть пропускная способность или реактивность системы.

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

1. *планирование ресурса* – то есть определение, кому, когда, а для делимых ресурсов и в каком количестве, необходимо выделить данный ресурс;
2. *отслеживание состояния ресурса* – то есть поддержание оперативной информации о том, занят или не занят ресурс, а для делимых ресурсов, – какое количество ресурса уже распределено, а какое свободно.

Разные ОС для решения этих общих задач управления ресурсами используют различные алгоритмы, которые и определяют их облик в целом, включая характеристики производительности, область применения, пользовательский интерфейс и т.п. Например, алгоритм управления процессором в значительной степени определяет, является ли ОС системой разделения времени, системой пакетной обработки или системой реального времени.

1.2 ЭВОЛЮЦИЯ ОС

Компьютер был изобретен английским математиком Чарльзом Бэббиджем в конце восемнадцатого века. Его "аналитическая машина" не смогла по-настоящему заработать, так как технологии того времени не удовлетворяли требованиям по изготовлению деталей точной механики, которые были необходимы для вычислительной техники. Этот компьютер не имел операционной системы.

1.2.1 Первый период развития (1945 -1955 годы)

Прогресс в создании цифровых вычислительных машин произошел после второй мировой войны. В середине 40-х годов 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 каких-либо 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.

1.2.2 Второй период (1955-1965 годы)

В середине 50-х годов появляется новая техническая база для вычислительной техники – *полупроводниковые элементы*. Компьютеры второго поколения могли уже непрерывно работать настолько долго, чтобы на них можно было возложить выполнение практически важных задач.

В этот период произошло разделение персонала на разработчиков вычислительных машин, эксплуатационников, операторов и программистов.

В эти годы появились первые алгоритмические языки, а, следовательно, и первые *системные программы – компиляторы*.

Возросла стоимость процессорного времени, что потребовало уменьшения непроизводительных затрат времени между запусками программ. Появились первые *системы пакетной обработки*, которые просто автоматизировали запуск одной программ 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.

В ходе реализации систем пакетной обработки был разработан формализованный язык управления заданиями (ЯУЗ), с помощью которого программист сообщал системе и оператору, какую работу он хочет выполнить на вычислительной машине. Совокупность нескольких заданий, как правило, в виде колоды перфокарт, получила название пакета заданий.

1.2.3 Третий период (1965-1980 годы)

Гораздо более широкие возможности третьему поколению компьютеров дал переход в середине 60-х годов в технической базе от отдельных полупроводниковых элементов типа транзисторов к *интегральным микросхемам*.

Было создано семейство программно-совместимых машин. Первым семейством программно-совместимых машин, построенных на интегральных микросхемах, явилась серия машин IBM/360, которая значительно превосходила машины второго поколения по критерию цена/производительность.

Программная совместимость потребовала и совместимости операционных систем. Такие операционные системы должны были работать на больших и малых вычислительных системах, с большим и с малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением удовлетворить всем этим противоречивым требованиям, оказались чрезвычайно сложными. Они были написаны тысячами программистов, состояли из многих миллионов ассемблерных строк кода и содержали тысячи ошибок, вызывающих нескончаемый поток исправлений.

Несмотря на огромные размеры и множество проблем, OS/360 и подобные ей операционные системы машин третьего поколения удовлетворяли большинству требований потребителей.

Важнейшим достижением ОС данного поколения явилась реализация мультипрограммирования. *Мультипрограммирование – это способ организации вычислительного процесса, при котором на одном процессоре попеременно выполняются несколько программ.* Пока одна программа выполняет операцию ввода-вывода, процессор не простаивает, как это происходило при последовательном выполнении программ (однопрограммный режим), а выполняет другую программу (многопрограммный режим). При этом каждая программа загружается в свой участок оперативной памяти, называемый разделом.

Другое нововведение – спулинг (spooling). *Спулинг* в то время *определялся 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 тип ОС – *системы разделения времени.* Вариант мультипрограммирования, применяемый в системах разделения времени, нацелен на создание для каждого отдельного пользователя иллюзии единоличного использования вычислительной машины.

1.2.4 Четвертый период (1980 год – настоящее время)

Появление больших интегральных схем (БИС) привело к очередному этапу в эволюции операционных систем. В эти годы произошло резкое возрастание степени интеграции и удешевление микросхем. Компьютер стал доступен отдельному человеку. Наступила эра персональных компьютеров.

С точки зрения архитектуры персональные компьютеры ничем не отличались от класса миникомпьютеров типа PDP-11, но вот цена у них существенно отличалась. Если миникомпьютер дал возможность иметь собственную вычислительную машину отделу предприятия или университету, то персональный компьютер сделал это возможным для отдельного человека.

Компьютеры стали широко использоваться неспециалистами, что потребовало разработки "дружественного" программного обеспечения, это положило конец кастовости программистов.

На рынке операционных систем доминировали две системы: MS-DOS и UNIX. Однопрограммная однопользовательская операционная система MS-DOS широко использовалась для компьютеров, построенных на базе микропроцессоров Intel 8088, а затем 80286, 80386 и 80486. Мультипрограммная многопользовательская операционная система UNIX доминировала в среде "не-интеловских" компьютеров, особенно построенных на базе высокопроизводительных RISC-процессоров.

В середине 80-х стали бурно развиваться сети персональных компьютеров, работающие под управлением *сетевых или распределенных ОС*.

В сетевых ОС пользователи должны быть осведомлены о наличии других компьютеров и должны делать логический вход в другой компьютер, чтобы воспользоваться его ресурсами, преимущественно файлами. Каждая машина в сети выполняет свою собственную локальную операционную систему, отличающуюся от ОС автономного компьютера наличием дополнительных средств, позволяющих компьютеру работать в сети. Сетевая ОС не имеет фундаментальных отличий от ОС однопроцессорного компьютера. Она обязательно содержит программную поддержку для сетевых интерфейсных устройств (драйвер сетевого адаптера), а также средства для удаленного входа в другие компьютеры сети и средства доступа к удаленным файлам, однако эти дополнения существенно не меняют структуру самой операционной системы.

1.3 КЛАССИФИКАЦИЯ ОС

Операционные системы могут различаться:

1. *особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера (процессорами, памятью, устройствами);*
2. *типами аппаратных платформ;*
3. *областями использования;*
4. *особенностями использованных методов проектирования;*
5. *многими другими свойствами.*

1.3.1 Особенности алгоритмов управления ресурсами

Эффективность функционирования ОС в целом зависит от алгоритмов управления локальными ресурсами компьютера. Характеризуя ОС, часто приводят важнейшие особенности реализации функций по управлению процессорами, памятью, внешними устройствами компьютера.

Например, *в зависимости от особенностей использованного алгоритма управления процессором, операционные системы делят на:*

1. *многозадачные и однозадачные;*
2. *многопользовательские и однопользовательские;*
3. *системы, поддерживающие многонитевую обработку и не поддерживающие ее;*
4. *на многопроцессорные и однопроцессорные системы.*

Поддержка многозадачности. По числу одновременно выполняемых задач операционные системы делятся на два класса:

1. *однозадачные* (например, MS-DOS, MSX) и
2. *многозадачные* (OS/2, UNIX, Windows).

Однозадачные ОС 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, внешние устройства и файлы.

Поддержка многопользовательского режима. По числу одновременно работающих пользователей ОС делятся на:

1. *однопользовательские* (MS-DOS, Windows 3.x, ранние версии OS/2);
2. *многопользовательские* (UNIX, Windows NT).

Главное отличие многопользовательских систем от однопользовательских, - наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей.

Не всякая многозадачная система является многопользовательской, и не всякая однопользовательская ОС является однозадачной.

Вытесняющая и невытесняющая многозадачность. Способ распределения важнейшего разделяемого ресурса, - 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:

1. *невытесняющая многозадачность* (NetWare, Windows 3.x);
2. *вытесняющая многозадачность* (Windows NT, OS/2, UNIX).

Основное различие между вытесняющим и невытесняющим вариантами многозадачности, - 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.

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

Многопроцессорная обработка. Другим важным свойством ОС является отсутствие или наличие в ней средств поддержки многопроцессорной обработки – *мультипроцессирование*. Мультипроцессирование приводит к усложнению всех алгоритмов управления ресурсами.

Функции поддержки многопроцессорной обработки данных имеются 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 по способу организации вычислительного процесса в системе с многопроцессорной архитектурой классифицируются:

1. асимметричные ОС;

2. *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.

Выше были рассмотрены характеристики ОС, связанные с управлением только одним типом ресурсов – процессором. Важное влияние на облик операционной системы в целом, на возможности ее использования в той или иной области оказывают особенности и других подсистем управления локальными ресурсами - подсистем управления памятью, файлами, устройствами ввода-вывода.

Специфика ОС проявляется и в том, каким образом она реализует сетевые функции: распознавание и перенаправление в сеть запросов к удаленным ресурсам, передача сообщений по сети, выполнение удаленных запросов и т.п.

1.3.2 Особенности аппаратных платформ

На свойства операционной системы непосредственное влияние оказывают аппаратные средства, на которые она ориентирована.

По типу аппаратуры различают операционные системы:

1. персональных компьютеров;
2. мини-компьютеров;
3. мейнфреймов;
4. кластеров;
5. сетей ЭВМ.

ОС большой машины является более сложной и функциональной, чем ОС персонального компьютера. В ОС больших машин функции по планированию потока выполняемых задач реализуются путем использования сложных приоритетных дисциплин и требуют большей вычислительной мощности, чем в ОС персональных компьютеров. Аналогично обстоит дело и с другими функциями.

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 приложения, которые она поддерживает, могли бы выполняться параллельно отдельными процессорами системы. В этом случае гораздо сложнее обеспечить согласованный доступ отдельных процессов к общим системным таблицам, исключить эффект гонок и прочие нежелательные последствия асинхронного выполнения работ.

К ОС кластеров предъявляются другие требования.

Кластер – слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системой.

Для функционирования кластерных систем необходима специальная аппаратура и, соответственно, программная поддержка со стороны операционной системы. Эта поддержка сводится в основном к:

1. синхронизации доступа к разделяемым ресурсам;
2. обнаружению отказов;
3. динамической реконфигурации системы.

Одной из первых разработок в области кластерных технологий были решения компании Digital Equipment на базе компьютеров VAX. Впоследствии, Digital Equipment совместно с корпорацией Microsoft

разработана кластерная технология, использующая 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, облегчающем перенос остальной части ОС, является написание ее на машинно-независимом языке, например, на С, который и был разработан для программирования операционных систем.

1.3.3 Особенности областей использования

Многозадачные ОС подразделяются на три типа в соответствии с использованными при их разработке критериями эффективности:

1. *системы пакетной обработки* (например, OS/360).
2. *системы разделения времени* (UNIX, VMS).
3. *системы реального времени* (QNX, RT/11).

Системы пакетной обработки предназначались для решения задач 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:

1. в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам;
2. из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач;
3. для одновременного выполнения выбираются задачи, предъявляющие отличающиеся требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины. 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. 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 задачи 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. 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 ведет диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго. Если квант выбран достаточно небольшим, то у всех пользователей, одновременно работающих на одной и той же машине, складывается впечатление единоличного использования машины.

Системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки (на выполнение принимается каждая запущенная пользователем задача, а не "выгодная" системе). 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* (станок, спутник, научная экспериментальная установка и т.п.) или *технологическими процессами* (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 или процессом. 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*.

1.3.4 Особенности методов построения

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. 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:

1. аккумуляцию удачных решений в форме стандартных объектов;
2. возможность создания новых объектов на базе имеющихся с помощью механизма наследования;
3. хорошую защиту данных за счет их инкапсуляции во внутренние структуры объекта, что делает данные недоступными для несанкционированного использования извне;
4. 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 операционной системе реализованы механизмы, которые дают возможность пользователю представлять и воспринимать сеть в виде традиционного однопроцессорного компьютера.

Характерные признаки распределенной организации ОС:

1. наличие единой справочной службы разделяемых ресурсов;
2. наличие единой службы времени;
3. использование механизма вызова удаленных процедур (RPC) для прозрачного распределения программных процедур по машинам;
4. использование механизма многонитевой обработки, позволяющей распараллеливать вычисления в рамках одной задачи и выполнять эту задачу сразу на нескольких компьютерах сети;
5. наличие других распределенных служб.

ТЕМА 2

СТРУКТУРА СЕТЕВОЙ ОПЕРАЦИОННОЙ СИСТЕМЫ. ОДНОРАНГОВЫЕ И ДВУХРАНГОВЫЕ ОПЕРАЦИОННЫЕ СИСТЕМЫ. ОПЕРАЦИОННЫЕ СИСТЕМЫ ДЛЯ РАБОЧИХ ГРУПП И СЕТЕЙ МАСШТАБА ПРЕДПРИЯТИЯ

2.1 СТРУКТУРА СЕТЕВОЙ ОПЕРАЦИОННОЙ СИСТЕМЫ

Сетевая операционная система составляет основу любой вычислительной сети. Каждый компьютер в сети в значительной степени автономен, поэтому *под **сетевой операционной системой** в широком смысле понимается совокупность операционных систем отдельных компьютеров, взаимодействующих с целью обмена сообщениями и разделения ресурсов по единым правилам - протоколам.*

В узком смысле сетевая ОС – это операционная система отдельного компьютера, обеспечивающая ему возможность работать в сети.

В сетевой операционной системе отдельной машины можно выделить несколько частей (рисунок 2.1):

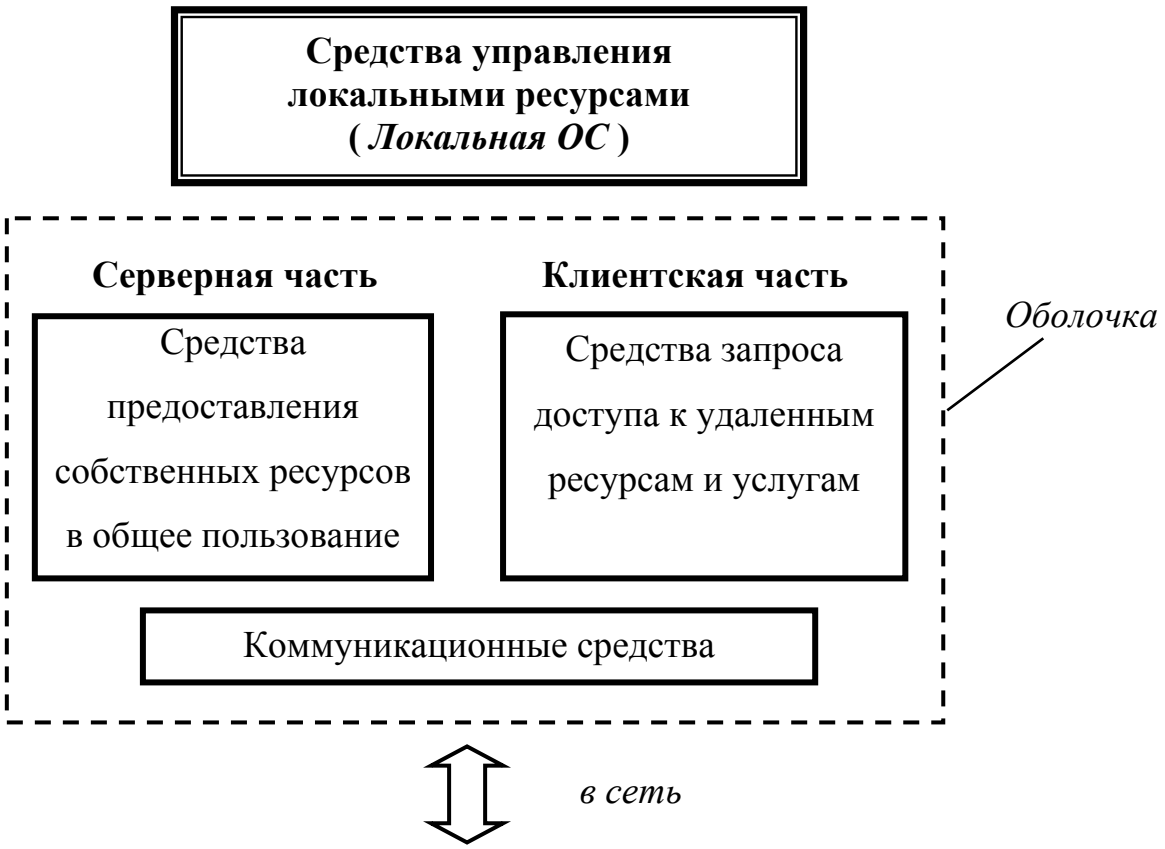


Рис. 2.1. Структура сетевой ОС

Средства управления локальными ресурсами компьютера включают функции:

1. распределения оперативной памяти между процессами;
2. планирования и диспетчеризации процессов;
3. управления процессорами в мультипроцессорных машинах;
4. управления периферийными устройствами;
5. другие функции управления ресурсами локальных ОС.

Средства предоставления собственных ресурсов и услуг в общее пользование – серверная часть ОС (сервер) обеспечивают:

1. блокировку файлов и записей, что необходимо для их совместного использования;
2. ведение справочников имен сетевых ресурсов;
3. обработку запросов удаленного доступа к собственной файловой системе и базе данных;
4. управление очередями запросов удаленных пользователей к своим периферийным устройствам.

Средства запроса доступа к удаленным ресурсам и услугам и их использования - клиентская часть ОС (редиректор) выполняют:

1. распознавание и перенаправление в сеть запросов к удаленным ресурсам от приложений и пользователей, при этом запрос поступает от приложения в локальной форме, а передается в сеть в другой форме, соответствующей требованиям сервера;
2. прием ответов от серверов и преобразование их в локальный формат, так что для приложения выполнение локальных и удаленных запросов неразличимо.

Коммуникационные средства ОС осуществляют обмен сообщениями в сети. Эта часть обеспечивает:

1. адресацию и буферизацию сообщений;
2. выбор маршрута передачи сообщения по сети;
3. надежность передачи и т.п.

Таким образом, коммуникационные средства являются средством транспортировки сообщений.

В зависимости от функций, возлагаемых на конкретный компьютер, в его операционной системе может отсутствовать либо клиентская, либо серверная части.

На рисунке 2.2 показано взаимодействие сетевых компонентов. Компьютер 1 - "чистый" клиент, компьютер 2 – "чистый" сервер. На первой машине отсутствует серверная часть, а на второй – клиентская.

Компонент клиентской части – редиректор перехватывает все запросы от приложений и анализирует их. Если выдан запрос к ресурсу данного компьютера, то он переадресовывается соответствующей подсистеме локальной ОС, если же это запрос к удаленному ресурсу, то он переправляется в сеть. При этом клиентская часть преобразует запрос из локальной формы в сетевой формат и передает его транспортной подсистеме, которая отвечает за доставку сообщений указанному серверу.

Серверная часть операционной системы компьютера 2 принимает запрос, преобразует его и передает для выполнения своей локальной ОС. После того, как результат получен, сервер обращается к транспортной подсистеме и направляет ответ клиенту, выдавшему запрос. Клиентская часть преобразует результат в соответствующий формат и адресует его тому приложению, которое выдало запрос.

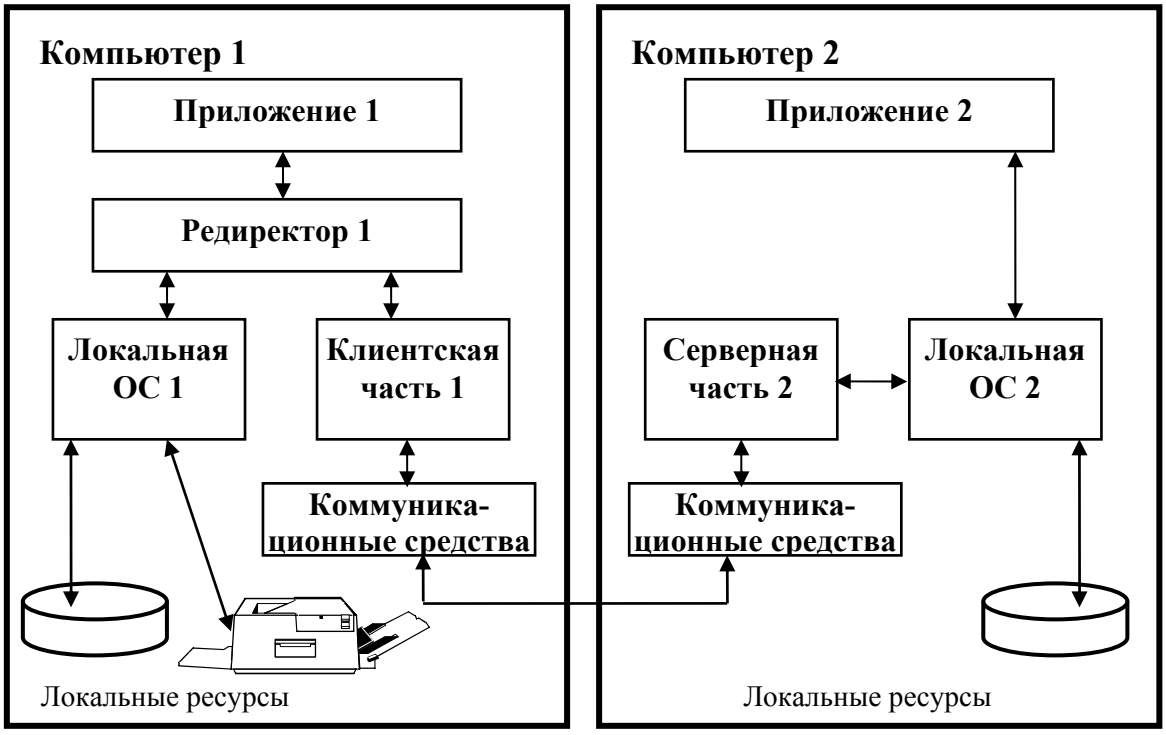


Рис. 2.2. Взаимодействие компонентов операционной системы при взаимодействии компьютеров

На практике сложилось несколько подходов к построению сетевых операционных систем (рисунок 2.3).

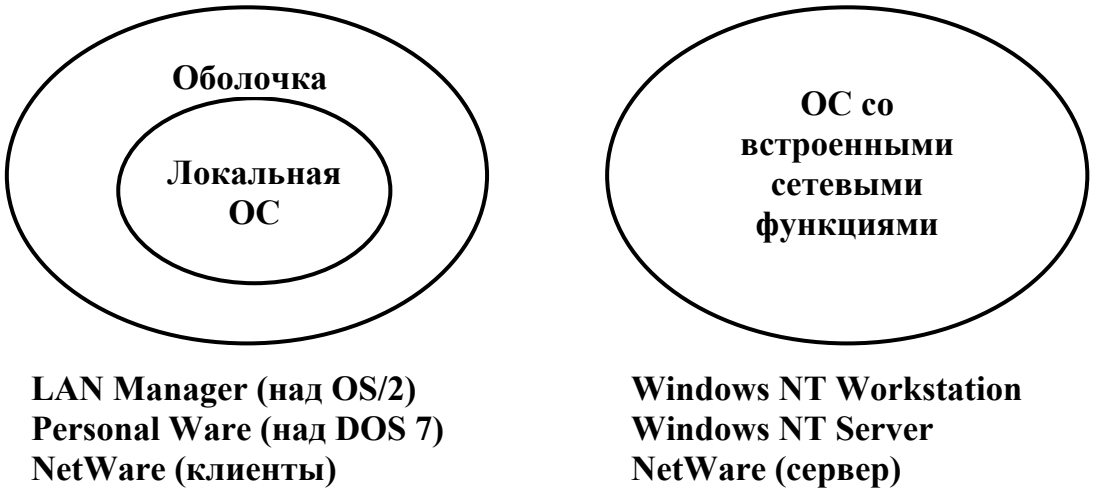


Рис. 2.3. Варианты построения сетевых ОС

Первые сетевые ОС представляли собой совокупность существующей локальной ОС и надстроенной над ней *сетевой оболочки*. В локальную ОС встраивался минимум сетевых функций, необходимых для работы сетевой оболочки, которая выполняла основные сетевые функции.

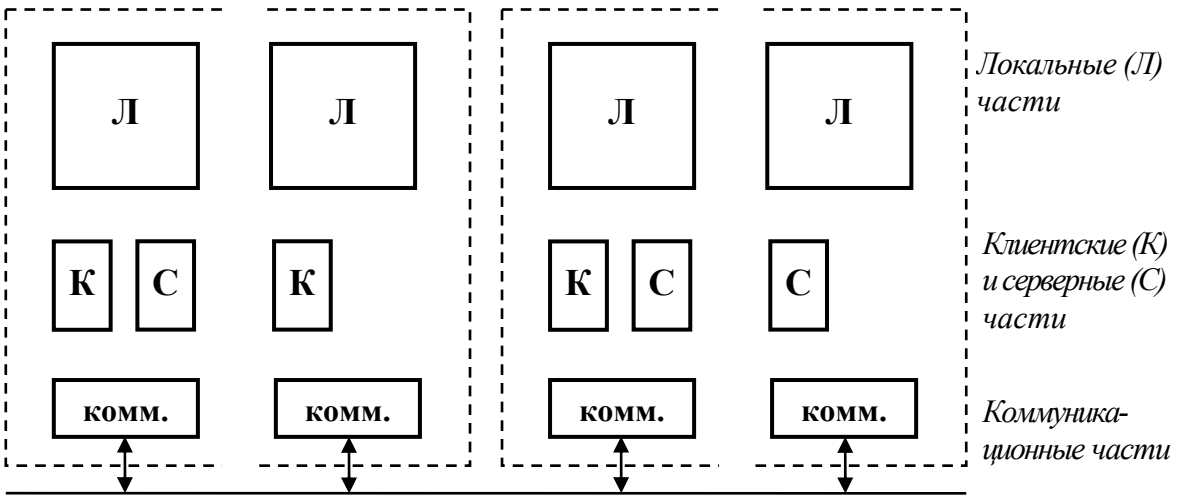
Примером является использование на каждой машине сети операционной системы MS DOS, у которой, начиная с ее третьей версии, появились такие встроенные функции, как блокировка файлов и записей, необходимые для совместного доступа к файлам. Принцип построения сетевых ОС в виде сетевой оболочки над локальной ОС используется и в таких ОС, как, например, как LANtastic или Personal Ware.

Однако более эффективными являются операционные системы, изначально предназначенные для работы в сети. Сетевые функции у ОС такого типа глубоко *встроены* в основные модули системы. Это обеспечивает их логическую стройность, простоту эксплуатации, модификации и высокую производительность.

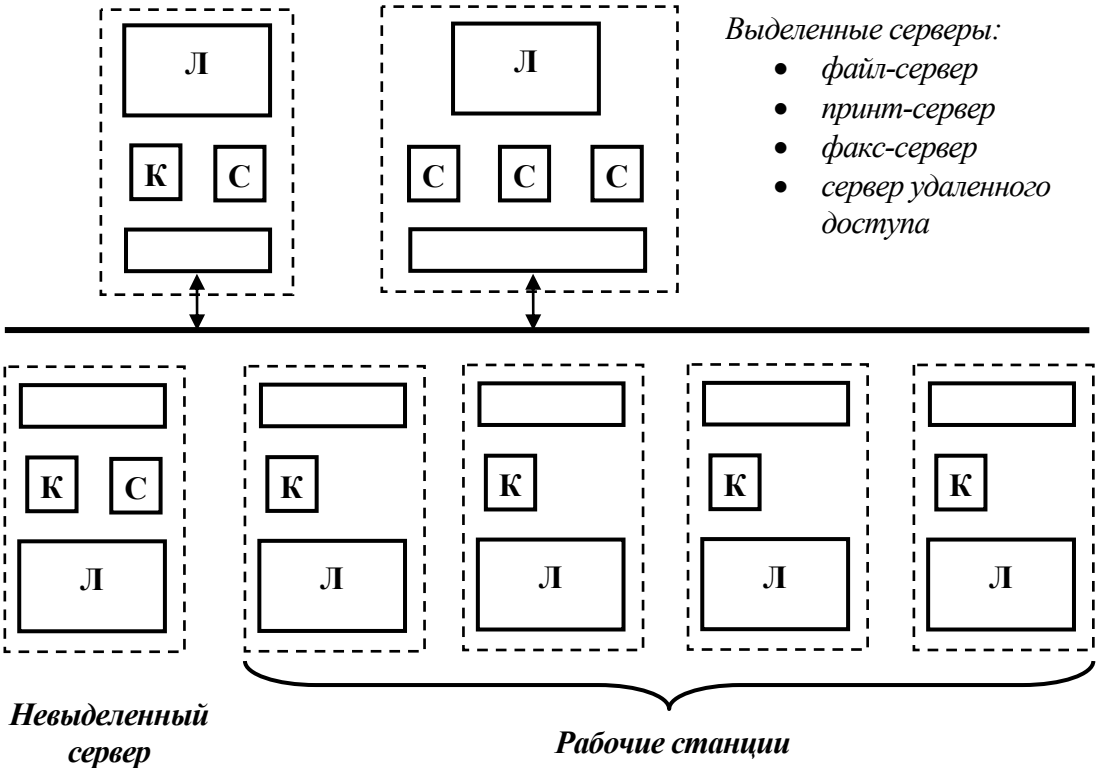
Примером такой операционной системы является Windows NT фирмы Microsoft. Эта операционная система за счет встроенных сетевых средств обеспечивает более высокие показатели производительности и защищенности информации по сравнению с сетевой ОС LAN Manager той же фирмы, разработанной совместно с IBM, являющейся надстройкой над локальной операционной системой OS/2.

2.2 ОДНОРАНГОВЫЕ СЕТЕВЫЕ ОС И ОС С ВЫДЕЛЕННЫМИ СЕРВЕРАМИ

В зависимости от того, как распределены функции между компьютерами сети, сетевые операционные системы, а следовательно, и сети делятся на два класса: одноранговые и двухранговые (рисунок 2.4). Последние чаще называют сетями с выделенными серверами.



(а)



(б)

Рис. 2.4. (а) – Одноранговая сеть (б) – Двухранговая сеть

Если компьютер предоставляет свои ресурсы другим пользователям сети, то он играет роль сервера. Компьютер, обращающийся к ресурсам другой машины, является клиентом. *Компьютер, работающий в сети, может выполнять функции либо клиента, либо сервера, либо совмещать обе эти функции.*

Если выполнение каких-либо серверных функций является основным назначением компьютера, например:

1. предоставление файлов в общее пользование всем остальным пользователям сети;
2. организация совместного использования принтера;
3. организация совместного использования факса;
4. предоставление всем пользователям сети возможности запуска на данном компьютере своих приложений.

то такой компьютер называется *выделенным сервером*. В зависимости от того, какой ресурс сервера является разделяемым, он называется:

1. файл-сервером;
2. принт-сервером;
3. факс-сервером;
4. сервером приложений и т.д.

На выделенных серверах желательно устанавливать ОС, специально оптимизированные для выполнения тех или иных серверных функций. Поэтому в сетях с выделенными серверами чаще всего используются сетевые

операционные системы, в состав которых входит *нескольких вариантов ОС*, отличающихся возможностями серверных частей.

Например, сетевая ОС Novell NetWare имеет серверный вариант, оптимизированный для работы в качестве файл-сервера, а также варианты оболочек для рабочих станций с различными локальными ОС, причем эти оболочки выполняют исключительно функции клиента.

Другим примером ОС, ориентированной на построение сети с выделенным сервером, является операционная система Windows NT. В отличие от NetWare, оба варианта данной сетевой ОС – Windows NT Server (для выделенного сервера) и Windows NT Workstation (для рабочей станции) - могут поддерживать функции и клиента и сервера. Но серверный вариант Windows NT имеет больше возможностей для предоставления ресурсов своего компьютера другим пользователям сети, так как может выполнять:

1. более широкий набор функций;
2. поддерживает большее количество одновременных соединений с клиентами;
3. реализует централизованное управление сетью;
4. имеет более развитые средства защиты.

Выделенный сервер не принято использовать в качестве компьютера для выполнения текущих задач, не связанных с его основным назначением, так как это может уменьшить производительность его работы как сервера.

В связи с такими соображениями в ОС Novell NetWare на серверной части возможность выполнения обычных прикладных программ вообще не предусмотрена, то есть сервер не содержит клиентской части, а на рабочих станциях отсутствуют серверные компоненты.

Однако в других сетевых ОС функционирование на выделенном сервере клиентской части вполне возможно. Например, под управлением Windows NT Server могут запускаться обычные программы локального пользователя, которые могут потребовать выполнения клиентских функций ОС при появлении запросов к ресурсам других компьютеров сети. При этом рабочие станции, на которых установлена ОС Windows NT Workstation, могут выполнять функции невыделенного сервера.

Итак, несмотря на то, что в сети с выделенным сервером все компьютеры в общем случае могут выполнять одновременно роли и сервера, и клиента, эта *сеть функционально не симметрична*: аппаратно и программно в ней реализованы два типа компьютеров - одни, в большей степени ориентированные на выполнение серверных функций и работающие под управлением специализированных серверных ОС, а другие - в основном выполняющие клиентские функции и работающие под управлением соответствующего этому назначению варианта ОС.

Функциональная несимметричность, как правило, вызывает и несимметричность аппаратуры – для выделенных серверов используются более мощные компьютеры с большими объемами оперативной и внешней памяти. Таким образом, функциональная несимметричность в сетях с

выделенным сервером сопровождается несимметричностью операционных систем (специализация ОС) и аппаратной несимметричностью (специализация компьютеров).

В одноранговых сетях все компьютеры равны в правах доступа к ресурсам друг друга. Каждый пользователь может по своему желанию объявить какой-либо ресурс своего компьютера разделяемым, после чего другие пользователи могут его эксплуатировать. В таких сетях на всех компьютерах устанавливается одна и та же ОС, которая предоставляет всем компьютерам в сети *потенциально* равные возможности. Одноранговые сети могут быть построены, например, на базе ОС LANtastic, Personal Ware, Windows for Workgroup, Windows NT Workstation, Windows XP Home Edition.

В одноранговых сетях также может возникнуть *функциональная несимметричность*: одни пользователи *не желают разделять свои ресурсы* с другими, и в таком случае их компьютеры выполняют роль клиента, за другими компьютерами администратор закрепил только *функции по организации совместного использования ресурсов*, а значит они являются серверами, в третьем случае, когда локальный пользователь *не возражает против использования его ресурсов* и сам *не исключает возможности обращения к другим компьютерам*, ОС, устанавливаемая на его компьютере, должна включать и серверную, и клиентскую части.

В отличие от сетей с выделенными серверами, в одноранговых сетях отсутствует специализация ОС в зависимости от преобладающей функциональной направленности – клиента или сервера. *Все вариации реализуются средствами конфигурирования одного и того же варианта ОС.*

Одноранговые сети проще в организации и эксплуатации, однако они применяются в основном для объединения небольших групп пользователей, не предъявляющих больших требований к:

1. объемам хранимой информации;
2. защищенности информации от несанкционированного доступа;
3. скорости доступа к информации.

При повышенных требованиях к этим характеристикам более подходящими являются двухранговые сети, где сервер лучше решает задачу обслуживания пользователей своими ресурсами, так как его аппаратура и сетевая операционная система специально спроектированы для этой цели.

2.3 ОС ДЛЯ РАБОЧИХ ГРУПП И ОС ДЛЯ СЕТЕЙ МАСШТАБА ПРЕДПРИЯТИЯ

Сетевые ОС имеют *разные свойства* в зависимости от того, предназначены они для сетей масштаба рабочей группы (отдела), для сетей масштаба кампуса или для сетей масштаба предприятия.

Сети отделов – используются небольшой группой сотрудников, решающих общие задачи. *Главной целью сети отдела является разделение локальных ресурсов*, таких как приложения, данные, лазерные принтеры и модемы. Сети отделов обычно не разделяются на подсети.

Сети кампусов – соединяют несколько сетей отделов внутри отдельного здания или внутри одной территории предприятия. Эти сети являются все еще локальными сетями, хотя и могут покрывать территорию несколько квадратных километров. Сервисы такой сети включают:

1. взаимодействие между сетями отделов;
2. доступ к базам данных предприятия;
3. доступ к факс-серверам;
4. доступ к высокоскоростным модемам;
5. доступ к высокоскоростным принтерам.

Сети предприятия (корпоративные сети) – объединяют все компьютеры всех территорий отдельного предприятия. Они могут покрывать город, регион или даже континент. В таких сетях пользователям предоставляется доступ к информации и приложениям, находящимся в других рабочих группах, других отделах, подразделениях и штаб-квартирах корпорации.

Операционные системы сетей отделов

Главной задачей операционной системы, используемой в сети масштаба отдела, *является организация разделения ресурсов, таких как приложения, данные, лазерные принтеры и, возможно, низкоскоростные модемы*.

Обычно сети отделов имеют один или два файловых сервера и не более чем 30 пользователей. Задачи управления на уровне отдела относительно просты. В задачи администратора входит:

1. добавление новых пользователей;
2. устранение простых отказов;
3. инсталляция новых узлов;
4. установка новых версий программного обеспечения.

Операционные системы сетей отделов применяются давно и достаточно отлаженные. Такая сеть обычно использует одну или максимум две сетевые ОС. Чаще всего это сеть с выделенным сервером, например, Windows NT или NetWare, или же одноранговая сеть, например, сеть Windows 98.

Операционные системы сетей кампусов

Пользователи и администраторы сетей отделов вскоре осознают, что они могут улучшить эффективность своей работы путем *получения доступа к информации других отделов своего предприятия*.

Следующим шагом в эволюции сетей является объединение локальных сетей нескольких отделов в единую сеть здания или группы зданий - *сеть кампусов*. Сети кампусов могут простираться на несколько километров. При этом глобальные соединения не требуются.

Операционная система, работающая в сети кампуса, должна обеспечивать для сотрудников одних отделов доступ к некоторым файлам и ресурсам сетей других отделов.

Сервисы, предоставляемые ОС сетей кампусов:

1. разделением файлов и принтеров;
2. предоставление доступа к серверам других типов, например, к факс-серверам и к серверам высокоскоростных модемов.
3. доступ к корпоративным базам данных, независимо от того, располагаются ли они на серверах баз данных или на миникомпьютерах.

На уровне сети кампуса начинаются *проблемы интеграции*. В общем случае, отделы уже выбрали для себя типы компьютеров, сетевого оборудования и сетевых операционных систем. Например, инженерный отдел может использовать операционную систему UNIX и сетевое оборудование Ethernet, отдел продаж может использовать операционные среды DOS/Novell и оборудование Token Ring. *Очень часто сеть кампуса соединяет разнородные компьютерные системы, в то время как сети отделов используют однотипные компьютеры.*

Операционные системы корпоративных сетей

Корпоративная сеть соединяет сети всех подразделений предприятия, в общем случае находящихся на значительных расстояниях. Корпоративные сети используют глобальные связи (WAN links) для соединения локальных сетей или отдельных компьютеров.

Наряду с базовыми сервисами, связанными с разделением файлов и принтеров, сетевая ОС, которая разрабатывается для корпораций, должна поддерживать более широкий набор сервисов, в который обычно входят:

1. почтовая служба;
2. средства коллективной работы;
3. поддержка удаленных пользователей;
4. факс-сервис;
5. обработка голосовых сообщений;
6. организация видеоконференций
7. и т.п.

Многие существующие методы и подходы к решению традиционных задач сетей меньших масштабов для корпоративной сети оказались непригодными. Например, простейшая для небольшой сети задача ведения учетной информации о пользователях выросла в сложную проблему для сети масштаба предприятия.

Особое значение приобрели *задачи преодоления гетерогенности* – в сети появились многочисленные шлюзы, обеспечивающие согласованную работу различных ОС и сетевых системных приложений.

Признаки корпоративных ОС:

1. Поддержка приложений. В корпоративных сетях выполняются сложные приложения, требующие для выполнения большой вычислительной мощности. Такие приложения разделяются на несколько частей. Например, на отдельных компьютерах выполняются:

1. часть приложения, связанная с выполнением запросов к базе данных;
2. запросы к файловому сервису;
3. часть, реализующая логику обработки данных приложения и организующая интерфейс с пользователем.

Вычислительная часть общих для корпорации программных систем может быть слишком объемной и неподъемной для рабочих станций клиентов, поэтому приложения будут выполняться более эффективно, если их наиболее сложные в вычислительном отношении части перенести на специально предназначенный для этого мощный компьютер - *сервер приложений.*

Сервер приложений должен базироваться на мощной аппаратной платформе (мультипроцессорные системы, часто на базе RISC-процессоров, специализированные кластерные архитектуры). ОС сервера приложений должна обеспечивать высокую производительность вычислений, а значит поддерживать многонитевую обработку, вытесняющую многозадачность, мультипроцессирование, виртуальную память и наиболее популярные прикладные среды (UNIX, Windows, MS-DOS, OS/2). Хорошая поддержка универсальных приложений, например, в Windows NT позволяет ей претендовать на место в мире корпоративных продуктов.

2. Справочная служба. Корпоративная ОС должна обладать способностью хранить информацию обо всех пользователях и ресурсах таким образом, чтобы обеспечивалось *управление ею из одной центральной точки.*

Корпоративная сеть должна хранить справочную информацию о самой себе:

1. данные о пользователях;
2. данные о серверах;
3. данные о рабочих станциях;
4. данные о кабельной системе и т.п.

Естественно организовать эту информацию в виде базы данных.

Данные из этой базы могут быть востребованы системами управления и администрирования. Такая база полезна при организации:

1. электронной почты;
2. систем коллективной работы;
3. службы безопасности;
4. службы инвентаризации программного и аппаратного обеспечения сети;
5. любого крупного бизнес-приложения.

Эта база данных предоставляет многообразие возможностей и порождает множество проблем. Она позволяет осуществлять различные операции поиска, сортировки, модификации и т.п., что очень сильно облегчает жизнь, как администраторам, так и пользователям. Но за эти

удобства приходится расплачиваться решением проблем распределенности, репликации и синхронизации.

В идеале сетевая справочная информация должна быть реализована в виде единой базы данных, а не представлять собой набор баз данных.

Например, в Windows NT имеется по крайней мере пять различных типов справочных баз данных:

1. *Главный справочник домена* (NT Domain Directory Service) - хранит информацию о пользователях, которая используется при организации их логического входа в сеть.
2. Данные о тех же пользователях могут содержаться и в другом справочнике, используемом электронной почтой Microsoft Mail.

Еще три базы данных поддерживают разрешение низкоуровневых адресов:

3. *WINS* – устанавливает соответствие Netbios-имен IP-адресам;
4. *справочник DNS* – сервер имен домена – оказывается полезным при подключении NT-сети к Internet, и наконец;
5. *справочник протокола DHCP* используется для автоматического назначения IP-адресов компьютерам сети.

Ближе к идеалу находятся справочные службы, поставляемые фирмой Banyan (продукт Streettalk III) и фирмой Novell (NetWare Directory Services), предлагающие единый справочник для всех сетевых приложений. *Наличие единой справочной службы для сетевой операционной системы – один из важнейших признаков ее корпоративности.*

3. Безопасность. Особую важность для ОС корпоративной сети приобретают вопросы безопасности данных. С одной стороны, в крупномасштабной сети объективно существует больше возможностей для несанкционированного доступа из-за:

1. децентрализации данных и большой распределенности "законных" точек доступа;
2. большого числа пользователей, благонадежность которых трудно установить;
3. большого числа возможных точек несанкционированного подключения к сети.

С другой стороны, корпоративные бизнес-приложения работают с данными, которые имеют большое значение для успешной работы корпорации в целом. И для защиты таких данных в корпоративных сетях наряду с различными аппаратными средствами используется весь *спектр средств защиты, предоставляемый операционной системой:*

1. избирательные или мандатные права доступа;
2. сложные процедуры аутентификации пользователей;
3. программная шифрация.

ТЕМА 3

ПРОЦЕССЫ. СОСТОЯНИЯ ПРОЦЕССОВ. КОНТЕКСТ И ДЕСКРИПТОР ПРОЦЕССА. АЛГОРИТМЫ ПЛАНИРОВАНИЯ ПРОЦЕССОВ. ВЫТЕСНЯЮЩИЕ И НЕВЫТЕСНЯЮЩИЕ АЛГОРИТМЫ ПЛАНИРОВАНИЯ ПРОЦЕССОВ.

Важнейшая функция операционной системы - организация рационального использования всех аппаратных и программных ресурсов системы. К *основным ресурсам* относятся:

1. процессоры;
2. память;
3. внешние устройства;
4. данные;
5. программы.

Вычислительная система, располагающая одними и теми же ресурсами, но управляемая различными операционными системами, может работать с разной степенью эффективности.

3.1 УПРАВЛЕНИЕ ПРОЦЕССАМИ

Подсистема управления процессами - важнейшая часть операционной системы, влияющая на функционирование вычислительной машины.

Процесс *(задача) - абстракция, описывающая выполняющуюся программу. Для операционной системы процесс представляет собой единицу работы, заявку на потребление системных ресурсов.*

Подсистема управления процессами:

1. занимается созданием и уничтожением процессов;
2. планирует выполнение процессов, то есть распределяет процессорное время между одновременно существующими в системе процессами;
3. обеспечивает процессы необходимыми системными ресурсами;
4. поддерживает взаимодействие между процессами.

3.2 СОСТОЯНИЯ ПРОЦЕССОВ

В многозадачной (многопроцессной) системе *процесс может находиться в одном из трех основных состояний:*

1. ВЫПОЛНЕНИЕ – активное состояние процесса. Процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;
2. ОЖИДАНИЕ – пассивное состояние процесса. Процесс заблокирован и не может выполняться по своим внутренним причинам. Он ждет осуществления некоторого события, например, завершения операции ввода-вывода, получения сообщения от другого процесса, освобождения какого-либо необходимого ему ресурса;
3. ГОТОВНОСТЬ – также пассивное состояние процесса. Но в этом случае процесс заблокирован внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, готов выполняться, однако процессор занят выполнением другого процесса.

В ходе жизненного цикла каждый процесс переходит из одного состояния в другое в соответствии с алгоритмом планирования процессов, реализуемым в данной операционной системе. Типичный граф состояний процесса показан на рисунке 3.1.

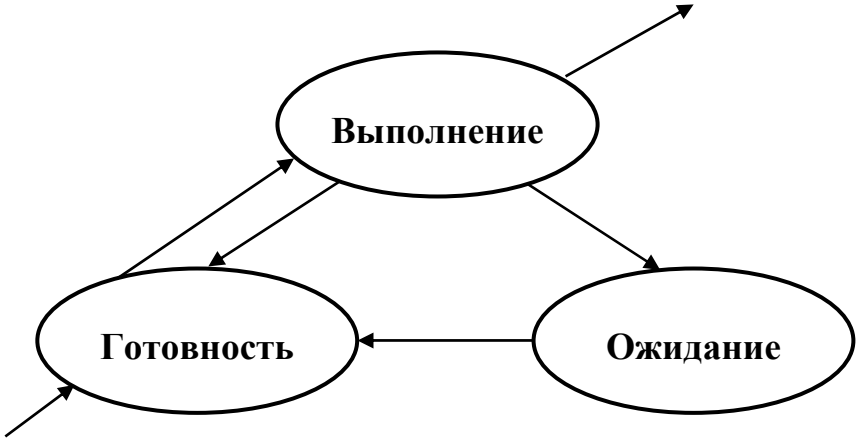


Рис. 3.1. Граф состояний процесса в многозадачной среде

В состоянии ВЫПОЛНЕНИЕ в однопроцессорной системе может находиться только один процесс, а в каждом из состояний ОЖИДАНИЕ и ГОТОВНОСТЬ – несколько процессов. Эти процессы образуют *очереди ожидающих и готовых процессов.*

Жизненный цикл процесса начинается с состояния ГОТОВНОСТЬ, когда процесс готов к выполнению и ждет своей очереди. При активизации процесс переходит в состояние ВЫПОЛНЕНИЕ и находится в нем до тех пор, пока либо он сам освободит процессор, перейдя в состояние ОЖИДАНИЯ какого-нибудь события, либо будет насильно "вытеснен" из процессора, например, вследствие исчерпания отведенного данному процессу кванта процессорного времени. В последнем случае процесс возвращается в состояние ГОТОВНОСТЬ. В это же состояние процесс переходит из состояния ОЖИДАНИЕ, после того, как ожидаемое событие произойдет.

3.3 КОНТЕКСТ И ДЕСКРИПТОР ПРОЦЕССА

На протяжении существования процесса его выполнение может быть многократно прервано и продолжено. Для того чтобы возобновить выполнение процесса, необходимо восстановить состояние его операционной среды. *Состояние операционной среды* отображается:

1. *состоянием регистров и программного счетчика;*
2. *режимом работы процессора;*
3. *указателями на открытые файлы;*
4. *информацией о незавершенных операциях ввода-вывода;*
5. *кодами ошибок выполняемых процессом системных вызовов и т.д.*

Эта информация называется ***контекстом процесса***.

Кроме этого, операционной системе *для реализации планирования* процессов требуется дополнительная информация:

1. *идентификатор процесса;*
2. *состояние процесса;*
3. *данные о степени привилегированности процесса;*
4. *место нахождения кодового сегмента и другая информация.*

В некоторых ОС (например, в ОС UNIX) информацию такого рода, используемую ОС для планирования процессов, называют ***дескриптором процесса***.

Дескриптор процесса по сравнению с контекстом содержит *более оперативную информацию*, которая должна быть легко доступна подсистеме планирования процессов. *Контекст процесса* содержит *менее актуальную информацию* и используется операционной системой только после того, как принято решение о возобновлении прерванного процесса.

Очереди процессов представляют собой дескрипторы отдельных процессов, объединенные в списки. Таким образом, каждый дескриптор, кроме всего прочего, содержит, по крайней мере, один указатель на другой дескриптор, соседствующий с ним в очереди. Такая организация очередей позволяет легко их переупорядочивать, включать и исключать процессы, переводить процессы из одного состояния в другое.

Программный код начнет выполняться только тогда, когда для него операционной системой будет создан процесс.

Создать процесс – это значит:

1. *создать информационные структуры, описывающие данный процесс, то есть его дескриптор и контекст;*
2. *включить дескриптор нового процесса в очередь готовых процессов;*
3. *загрузить кодовый сегмент процесса в оперативную память или в область свопинга.*

3.4 АЛГОРИТМЫ ПЛАНИРОВАНИЯ ПРОЦЕССОВ

Планирование процессов включает в себя решение следующих задач:

1. определение момента времени для смены выполняемого процесса;

2. выбор процесса на выполнение из очереди готовых процессов;

3. переключение контекстов "старого" и "нового" процессов.

Первые две задачи решаются программными средствами, а последняя задача в значительной степени аппаратно.

Существует множество различных алгоритмов планирования процессов, по-разному решающих вышеперечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования.

Среди этого множества алгоритмов выделяются две группы наиболее часто встречающихся алгоритмов: *алгоритмы, основанные на* ***квантовании****, и алгоритмы, основанные на* ***приоритетах***.

В соответствии с *алгоритмами, основанными на квантовании*, смена активного процесса происходит, если:

1. процесс завершился и покинул систему;
2. произошла ошибка;
3. процесс перешел в состояние ОЖИДАНИЕ;
4. исчерпан квант процессорного времени, отведенный данному процессу.

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

Таким образом, ни один процесс не занимает процессор надолго, поэтому *квантование широко используется в системах разделения времени.* Граф состояний процесса, изображенный на рисунке 3.1, соответствует алгоритму планирования, основанному на квантовании.

Алгоритмы, основанные на квантовании, могут отличаться следующими составляющими:

1. Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными;
2. Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться в разные периоды жизни процесса;
3. Процессы, которые не полностью использовали выделенный им квант (например, из-за ухода на выполнение операций ввода-вывода), могут получить или не получить компенсацию в виде привилегий при последующем обслуживании.
4. По разному может быть организована очередь готовых процессов: циклически, по правилу "первый пришел - первый обслужился" (FIFO) или по правилу "последний пришел - первый обслужился" (LIFO).

Другая группа алгоритмов использует понятие *"приоритет"* процесса.

Приоритет *– это число, характеризующее степень привилегированности процесса при использовании ресурсов вычислительной машины, в частности, процессорного времени: чем выше приоритет, тем выше привилегии.*

Приоритет может выражаться:

1. целым или дробным значением;
2. положительным или отрицательным значением.

Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может:

1. назначаться директивно администратором системы в зависимости от важности работы или внесенной платы;
2. вычисляться самой ОС по определенным правилам.

Он может оставаться фиксированным на протяжении всей жизни процесса либо изменяться во времени - в соответствии с некоторым законом. В последнем случае приоритеты называются *динамическими.*

Существует две *разновидности приоритетных алгоритмов:*

1. алгоритмы, использующие относительные приоритеты;

2. *алгоритмы, использующие абсолютные приоритеты.*

В обоих случаях из очереди готовых выбирается процесс, имеющий наивысший приоритет. По-разному решается проблема определения момента смены активного процесса.

В системах с относительными приоритетами активный процесс выполняется до тех пор, пока он сам не покинет процессор, перейдя в состояние ОЖИДАНИЕ (или произойдет ошибка, или процесс завершится).

В системах с абсолютными приоритетами выполнение активного процесса прерывается, еще и если в очереди готовых процессов появился процесс, приоритет которого выше приоритета активного процесса. В этом случае прерванный процесс переходит в состояние готовности.

На рисунке 3.2 показаны графы состояний процесса для алгоритмов с относительными (а) и абсолютными (б) приоритетами.

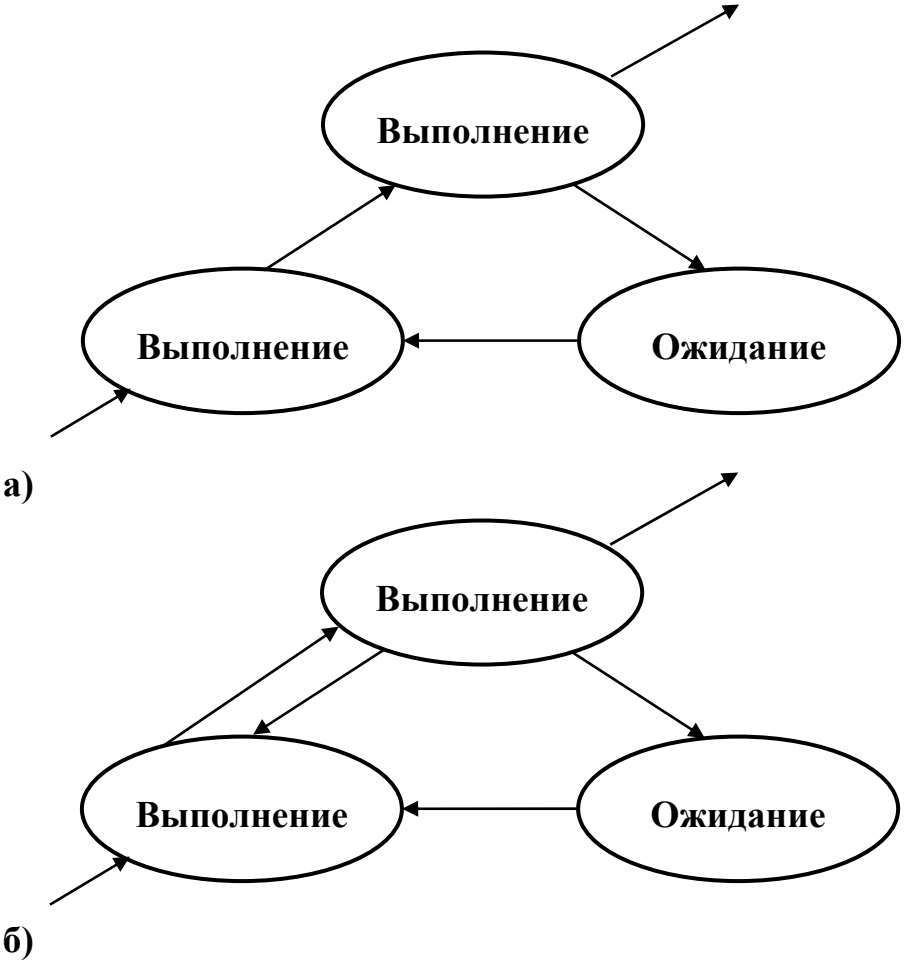


Рис. 3.2. Графы состояний процессов в системах (а) с относительными приоритетами; (б) с абсолютными приоритетами

Во многих операционных системах алгоритмы планирования построены с использованием, как квантования, так и приоритетов. Например, в основе планирования лежит квантование, а величина кванта и/или порядок выбора процесса из очереди готовых определяется приоритетами процессов.

3.4 ВЫТЕСНЯЮЩИЕ И НЕВЫТЕСНЯЮЩИЕ АЛГОРИТМЫ ПЛАНИРОВАНИЯ

Существует *два основных типа процедур планирования процессов* – ***вытесняющие (preemptive) и невытесняющие (non-preemptive).***

Non-preemptive multitasking - невытесняющая многозадачность – это способ планирования процессов, при котором *активный процесс* выполняется до тех пор, пока он *сам*, по собственной инициативе, не *отдаст управление планировщику операционной системы* для того, чтобы тот выбрал из очереди другой, готовый к выполнению процесс.

Preemptive multitasking – вытесняющая многозадачность – это такой способ, при котором решение о *переключении процессора с выполнения одного процесса на выполнение другого процесса принимается планировщиком операционной системы*, а не самой активной задачей.

Вытесняющая и невытесняющая многозадачность - это более широкие понятия, чем типы приоритетности. Приоритеты задач могут, как использоваться, так и не использоваться и при вытесняющих, и при невытесняющих способах планирования.

Основным различием между preemptive и non-preemptive вариантами многозадачности является *степень централизации механизма планирования задач.*

При *вытесняющей многозадачности механизм планирования задач целиком сосредоточен в операционной системе*, и программист пишет свое приложение, не заботясь о том, что оно будет выполняться параллельно с другими задачами. При этом операционная система выполняет следующие функции:

1. определяет момент снятия с выполнения активной задачи;
2. запоминает ее контекст;
3. выбирает из очереди готовых задач следующую и запускает ее на выполнение, загружая ее контекст.

При *невытесняющей многозадачности механизм планирования распределен между системой и прикладными программами*. Прикладная программа, получив управление от операционной системы, *сама определяет момент завершения своей очередной итерации* и передает управление ОС с помощью какого-либо системного вызова, а *ОС формирует очереди задач и выбирает в соответствии с некоторым алгоритмом (например, с учетом приоритетов) следующую задачу на выполнение.* Такой механизм создает проблемы, как для пользователей, так и для разработчиков.

Для пользователей это означает, что управление системой теряется на произвольный период времени, который определяется приложением, а не пользователем. Если приложение тратит слишком много времени на выполнение какой-либо работы, например, на форматирование диска, пользователь не может переключиться с этой задачи на другую задачу, например, на текстовый редактор, в то время как форматирование продолжалось бы в фоновом режиме.

Поэтому разработчики приложений для non-preemptive операционной среды, возлагая на себя функции планировщика, должны создавать приложения так, чтобы они выполняли свои задачи небольшими частями.

Например, программа форматирования может отформатировать одну дорожку дискеты и вернуть управление системе. После выполнения других задач система возвратит управление программе форматирования, чтобы та отформатировала следующую дорожку.

Подобный метод разделения времени между задачами предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить, чтобы его программа достаточно часто отдавала управление другим выполняемым одновременно с ней программам. Крайним проявлением "недружественности" приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации исключены, так как центральный планирующий механизм снимет зависшую задачу с выполнения.

Однако *распределение функций планировщика между системой и приложениями не всегда является недостатком*. Преимущества метода:

1. дает возможность разработчику приложений самому проектировать алгоритм планирования, наиболее подходящий для данного фиксированного набора задач. Так как разработчик сам определяет в программе момент времени отдачи управления, то при этом исключаются нерациональные прерывания программ в "неудобные" для них моменты времени;
2. легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные;
3. более высокая скорость переключения с задачи на задачу.

Примером эффективного использования невытесняющей многозадачности является файл-сервер NetWare, в котором, в значительной степени благодаря этому, достигнута высокая скорость выполнения файловых операций.

Однако, **почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений (UNIX, Windows, OS/2, VAX/VMS), реализована вытесняющая многозадачность.**

В связи с этим вытесняющую многозадачность часто называют *истинной многозадачностью.*

ТЕМА 4

СИНХРОНИЗАЦИЯ И ВЗАИМОДЕЙСТВИЕ ПРОЦЕССОВ. НЕОБХОДИМОСТЬ СИНХРОНИЗАЦИИ. ВЗАИМНОЕ ИСКЛЮЧЕНИЕ. АЛГОРИТМ ДЕЙКСТРА. ТУПИКИ. МОНИТОРЫ.

4.1 НЕОБХОДИМОСТЬ СИНХРОНИЗАЦИИ

Процессам часто нужно взаимодействовать друг с другом, например:

1. один процесс может передавать данные другому процессу;
2. несколько процессов могут обрабатывать данные из общего файла.

Во всех этих случаях возникает проблема синхронизации процессов, которая может решаться: 1) приостановкой и активизацией процессов; 2) организацией очередей; 3) блокированием и освобождением ресурсов.

Пренебрежение вопросами синхронизации процессов, выполняющихся в режиме мультипрограммирования, может привести к их неправильной работе или даже к краху системы. Рассмотрим, например (рисунок 4.1), программу печати файлов (принт-сервер).

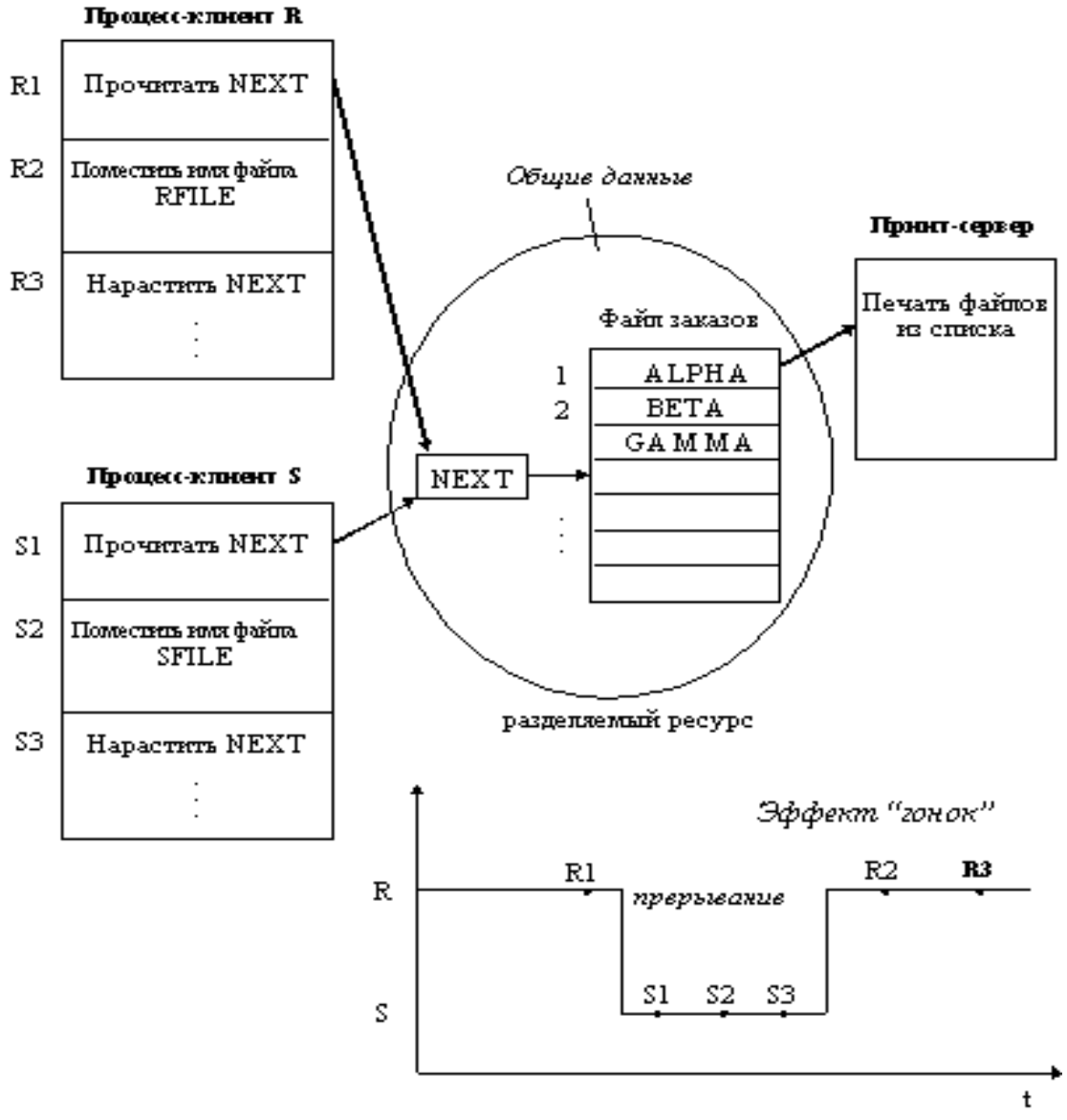


Рис. 4.1. Пример необходимости синхронизации

Эта программа печатает по очереди все файлы, имена которых последовательно в порядке поступления записывают в специальный общедоступный файл "заказов" другие программы. Особая переменная NEXT, также доступная всем процессам-клиентам, содержит *номер первой свободной для записи имени файла позиции файла "заказов"*.

Процессы-клиенты:

1. читают эту переменную;
2. записывают в соответствующую позицию файла "заказов" имя своего файла;
3. наращивают значение NEXT на единицу.

Предположим, что в некоторый момент процесс **R** решил распечатать свой файл, для этого он прочитал значение переменной NEXT, значение которой для определенности положим равным 4. Процесс запомнил это значение, но поместить имя файла не успел, так как его выполнение было прервано (например, вследствие исчерпания кванта времени).

Очередной процесс **S**, желающий распечатать файл, прочитал то же самое значение переменной NEXT, поместил в четвертую позицию имя своего файла и нарастил значение переменной на единицу. Когда в очередной раз управление будет передано процессу **R**, то он, продолжая свое выполнение, в полном соответствии со значением текущей свободной позиции, полученным во время предыдущей итерации, запишет имя файла также в позицию 4, поверх имени файла процесса **S**. Таким образом, процесс **S** никогда не увидит свой файл распечатанным.

Сложность проблемы синхронизации состоит в нерегулярности возникающих ситуаций. В предыдущем примере можно представить и другое развитие событий:

1. были потеряны файлы нескольких процессов;
2. не был потерян ни один файл.

В данном случае все определяется взаимными *скоростями процессов и моментами их прерывания*. Поэтому отладка взаимодействующих процессов является сложной задачей.

Ситуации подобные той, *когда два или более процессов обрабатывают разделяемые данные, и конечный результат зависит от соотношения скоростей процессов, называются* ***гонками***.

4.2 ВЗАИМНОЕ ИСКЛЮЧЕНИЕ

Критическая секция *- это часть программы, в которой осуществляется доступ к разделяемым данным.*

Чтобы исключить эффект гонок по отношению к некоторому ресурсу, необходимо обеспечить, чтобы в каждый момент в критической секции, связанной с этим ресурсом, находился максимум один процесс. Этот прием называют ***взаимным исключением.***

Простейший способ обеспечить взаимное исключение - *позволить процессу,* находящемуся в критической секции, *запрещать все*

прерывания. Этот способ непригоден, так как опасно доверять управление системой пользовательскому процессу: он может надолго занять процессор, а при крахе процесса в критической области крах потерпит вся система, потому что прерывания никогда не будут разрешены.

Другим способом является использование *блокирующих переменных*. С каждым разделяемым ресурсом связывается двоичная переменная, принимающая значение *1, если ресурс свободен* (ни один процесс не находится в данный момент в критической секции, связанной с данным ресурсом), и значение *0, если ресурс занят*. На рисунке 4.2 показан фрагмент алгоритма процесса, использующего для реализации взаимного исключения доступа к разделяемому ресурсу D блокирующую переменную F(D).

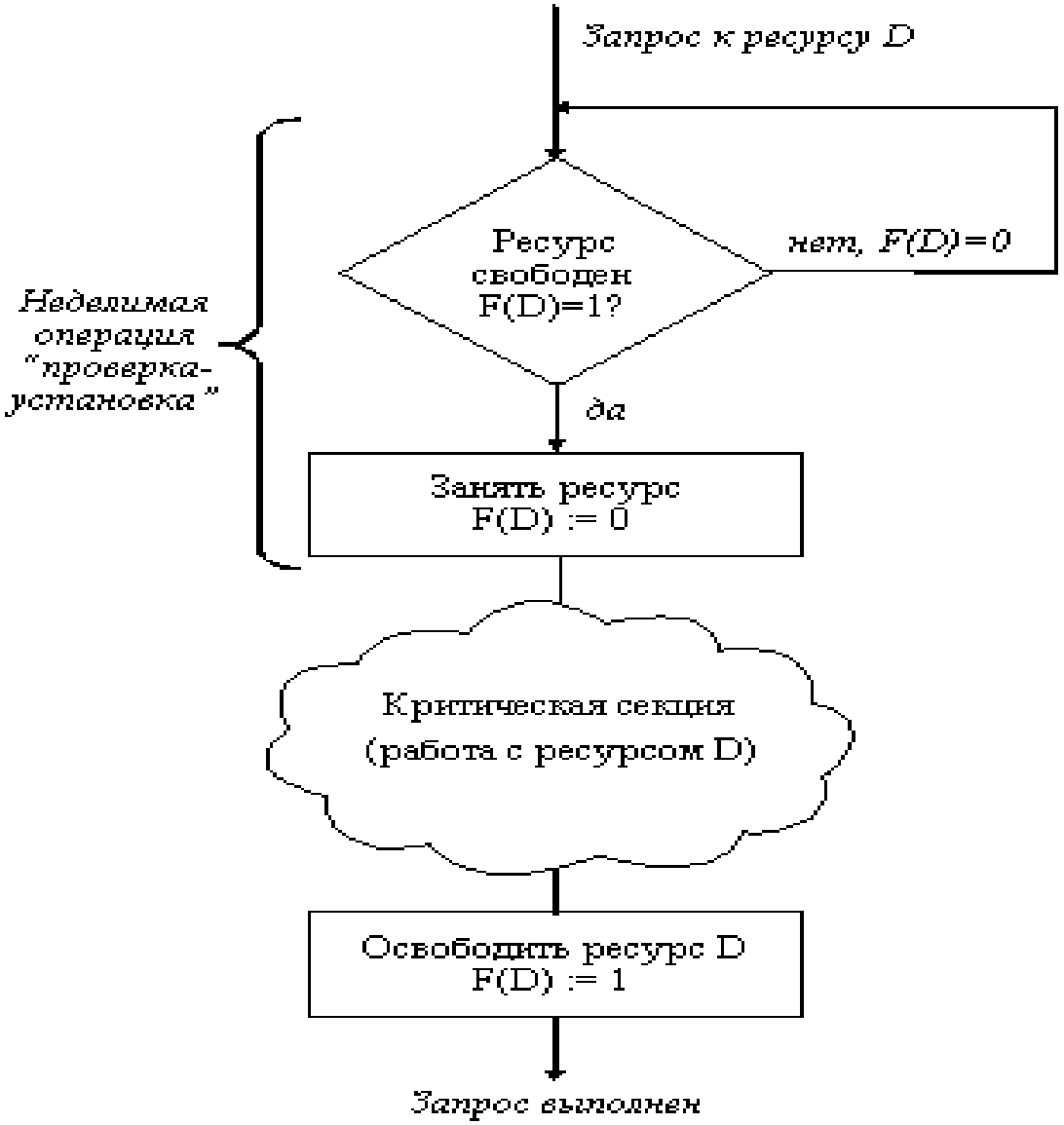


Рис. 4.2. Реализация критических секций с использованием блокирующих переменных

Перед входом в критическую секцию процесс проверяет, свободен ли ресурс D. Если он занят, то проверка циклически повторяется, если свободен, то значение переменной F(D) устанавливается в 0, и процесс входит в критическую секцию. После того, как процесс выполнит все действия с разделяемым ресурсом D, значение переменной F(D) снова устанавливается равным 1. *Если все процессы написаны с использованием вышеописанных соглашений, то взаимное исключение гарантируется.*

Операция проверки и установки блокирующей переменной должна быть неделимой. Пусть в результате проверки переменной процесс определил, что ресурс свободен, но сразу после этого, не успев установить переменную в 0, был прерван. За время его приостановки другой процесс занял ресурс, вошел в свою критическую секцию, но также был прерван, не завершив работы с разделяемым ресурсом. Когда управление было возвращено первому процессу, он, считая ресурс свободным, установил признак занятости и начал выполнять свою критическую секцию. Таким образом, был нарушен принцип взаимного исключения, что потенциально может привести к нежелательным последствиям.

Поэтому в системе команд машины желательно иметь единую команду "проверка-установка", или же реализовывать системными средствами программные примитивы, которые бы запрещали прерывания на протяжении всей операции проверки и установки.

Реализация критических секций с использованием блокирующих переменных имеет существенный *недостаток: в течение времени, когда один процесс находится в критической секции, другой процесс, которому требуется тот же ресурс, будет выполнять рутинные действия по опросу блокирующей переменной, бесполезно тратя процессорное время*

Для устранения таких ситуаций может быть использован так называемый ***аппарат событий***. С помощью этого средства могут решаться не только проблемы взаимного исключения, но и более общие задачи синхронизации процессов.

В разных операционных системах аппарат событий реализуется по-своему, но в любом случае используются системные функции аналогичного назначения, которые условно назовем WAIT(x) и POST(x), где x - идентификатор некоторого события.

На рисунке 4.3 показан фрагмент алгоритма процесса, использующего эти функции. *Если ресурс занят, то процесс вызывает системную функцию WAIT(D), где D обозначает событие, заключающееся в освобождении ресурса D. Функция WAIT(D) переводит активный процесс в состояние ОЖИДАНИЕ и делает отметку в его дескрипторе о том, что процесс ожидает события D. Процесс, который в это время использует ресурс D, после выхода из критической секции выполняет системную функцию POST(D), в результате чего операционная система просматривает очередь ожидающих процессов и переводит процесс, ожидающий события D, в состояние ГОТОВНОСТЬ.*

Рис. 4.3. Реализация критической секции с использованием системных функций WAIT(D) и POST(D)

4.3 АЛГОРИТМ ДЕЙКСТРА

Обобщающее средство синхронизации процессов предложил Дейкстра, который ввел два новых примитива. В абстрактной форме эти примитивы, обозначаемые **P** и **V**, оперируют над целыми неотрицательными переменными, называемыми *семафорами*. Пусть **S** такой семафор.

Операции определяются следующим образом:

Операция V(S) :

1. переменная S увеличивается на 1 одним неделимым действием;
2. выборка, инкремент и запоминание не могут быть прерваны;
3. к S нет доступа другим процессам во время выполнения этой операции.

Операция P(S) :

1. уменьшение S на 1, если это возможно.
2. Если S=0, то невозможно уменьшить S и остаться в области целых неотрицательных значений, в этом случае процесс, вызывающий P-операцию, ждет, пока это уменьшение станет возможным.
3. Успешная проверка и уменьшение также является неделимой операцией

В частном случае, когда семафор S может принимать только значения 0 и 1, он превращается в блокирующую переменную.

Операция P заключает в себе потенциальную возможность перехода процесса, который ее выполняет, в состояние ожидания.

V-операция может при некоторых обстоятельствах активизировать другой процесс, приостановленный операцией P.

Рассмотрим использование семафоров на классическом примере взаимодействия двух процессов, выполняющихся в режиме мультипрограммирования, один из которых пишет данные в буферный пул, а другой считывает их из буферного пула.

Пусть буферный пул состоит из N буферов, каждый из которых может содержать одну запись. Процесс "писатель" должен приостанавливаться, когда все буфера оказываются занятыми, и активизироваться при освобождении хотя бы одного буфера. Напротив, процесс "читатель" приостанавливается, когда все буферы пусты, и активизируется при появлении хотя бы одной записи.

Введем два семафора: **e** - число пустых буферов и **f** - число заполненных буферов. Предположим, что запись в буфер и считывание из буфера являются критическими секциями (как в примере с принт-сервером). Введем также двоичный семафор **b**, используемый для обеспечения взаимного исключения. Тогда процессы могут быть описаны следующим образом:

```
// Глобальные переменные
#define N 256
int e = N, f = 0, b = 1;

void Writer () {
   while(1)  {
      PrepareNextRecord(); /* Подготовка новой записи   */
      P(e);                /* Уменьшить число свободных буферов, если они есть */
                           /* в противном случае - ждать, пока они освободятся */
      P(b);                /* Вход в критическую секцию  */
      AddToBuffer();       /* Добавить новую запись в буфер */
      V(b);                /* Выход из критической секции   */
      V(f);                /* Увеличить число занятых буферов */
  }
}
```

```
void Reader ()  {
   while(1)  {
      P(f);                  /* Уменьшить число занятых буферов, если они есть */
                             /* в противном случае ждать, пока они появятся  */
      P(b);                  /* Вход в критическую секцию */
     GetFromBuffer();  /* Взять запись из буфера*/
      V(b);                  /* Выход из критической секции */
      V(e);                 /* Увеличить число свободных буферов */
      ProcessRecord();  /* Обработать запись */
  }
}
```

4.4 ТУПИКИ

Приведенный выше пример может проиллюстрировать еще одну проблему синхронизации - *взаимные блокировки*, называемые также *дедлоками (deadlocks), клинчами (clinch)* или *тупиками*.

Если переставить местами операции P(e) и P(b) в программе "писателе", то при некотором стечении обстоятельств процессы могут взаимно заблокировать друг друга. Пусть "писатель" первым войдет в критическую секцию и обнаружит отсутствие свободных буферов; он начнет ждать, когда "читатель" возьмет очередную запись из буфера, но "читатель" не сможет этого сделать, так как для этого необходимо войти в критическую секцию, вход в которую заблокирован процессом "писателем".

Рассмотрим еще один пример тупика. Пусть двум процессам, выполняющимся в режиме мультипрограммирования, для выполнения их работы нужно два ресурса, например, принтер и диск. На рисунке 4.4(а) показаны фрагменты соответствующих программ. И пусть после того, как процесс **A** занял принтер (установил блокирующую переменную), он был прерван. Управление получил процесс **B**, который сначала занял диск, но при выполнении следующей команды был заблокирован, так как принтер оказался уже занятым процессом **A**. Управление снова получил процесс **A**, который в соответствии со своей программой сделал попытку занять диск и был заблокирован: диск уже распределен процессу **B**. В таком положении процессы **A** и **B** могут находиться сколь угодно долго.

В зависимости от соотношения скоростей процессов, они могут:

1. взаимно блокировать друг друга 4.4 (*б*).
2. образовывать очереди к разделяемым ресурсам 4.4 (*в*);
3. совершенно независимо использовать разделяемые ресурсы 4.4 (*г*);

Тупиковые ситуации надо отличать от простых очередей, хотя и те и другие возникают при совместном использовании ресурсов и внешне выглядят похоже: процесс приостанавливается и ждет освобождения ресурса. Однако очередь - это нормальное явление, неотъемлемый признак высокого коэффициента использования ресурсов при случайном

поступлении запросов. Она возникает тогда, когда ресурс недоступен в данный момент, но через некоторое время он освобождается, и процесс продолжает свое выполнение. *Тупик же, что видно из его названия, является в некотором роде неразрешимой ситуацией.*

В рассмотренных ниже примерах тупик был образован двумя процессами, но взаимно блокировать друг друга могут и большее число процессов.

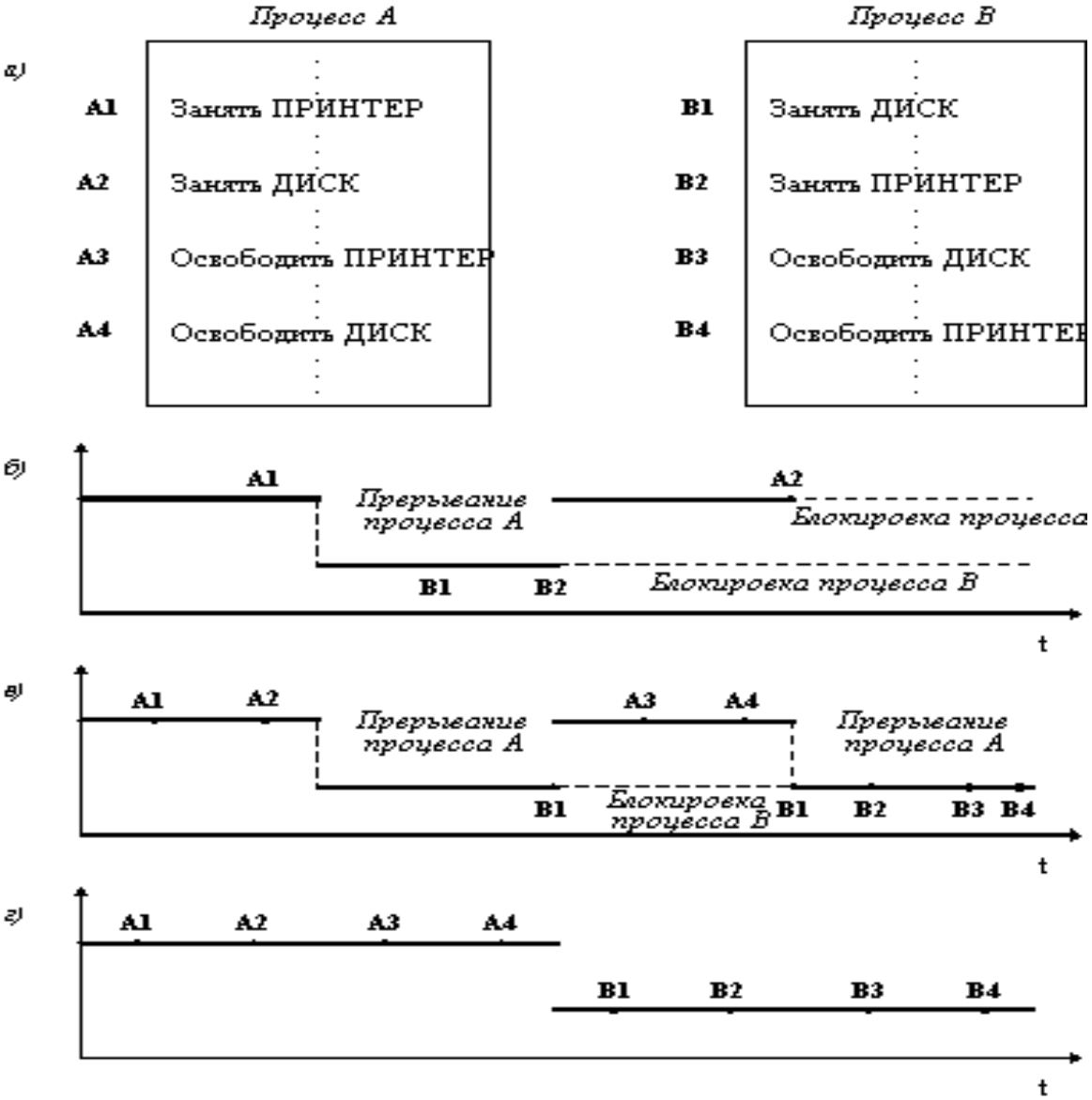


Рис. 4.4. (а) фрагменты программ А и В, разделяющих принтер и диск; (б) взаимная блокировка (клинч);(в) очередь к разделяемому диску; (г) независимое использование ресурсов

Проблема тупиков включает в себя решение следующих задач:

1. предотвращение тупиков,
2. распознавание тупиков,
3. восстановление системы после тупиков.

Предотвращение тупиков может быть осуществлено *на стадии написания программ*. Программы должны быть написаны таким образом, чтобы тупик не мог возникнуть ни при каком соотношении взаимных

скоростей процессов. Так, если бы в предыдущем примере процесс А и процесс В запрашивали ресурсы в одинаковой последовательности, то тупик был бы в принципе невозможен. *Второй подход* к предотвращению тупиков называется динамическим и заключается в *использовании определенных правил при назначении ресурсов процессам*, например, ресурсы могут выделяться в определенной последовательности, общей для всех процессов.

В некоторых случаях, когда тупиковая ситуация образована многими процессами, использующими много ресурсов, распознавание тупика является нетривиальной задачей. Существуют формальные, программно-реализованные методы распознавания тупиков, основанные на *ведении таблиц распределения ресурсов и таблиц запросов к занятым ресурсам.* Анализ этих таблиц позволяет обнаружить взаимные блокировки.

Если же тупиковая ситуация возникла, то не обязательно снимать с выполнения все заблокированные процессы.

Можно снять только часть из них, при этом освобождаются ресурсы, ожидаемые остальными процессами.

Можно *вернуть некоторые процессы в область свопинга*.

Можно совершить *"откат" некоторых процессов до так называемой контрольной точки*, в которой запоминается вся информация, необходимая для восстановления выполнения программы с данного места. Контрольные точки расставляются в программе в местах, после которых возможно возникновение тупика.

Вывод*: использовать семафоры нужно очень осторожно, так как одна незначительная ошибка может привести к останову системы.*

4.5 МОНИТОРЫ

Монитор - *высокоуровневое средство синхронизации. Это набор процедур, переменных и структур данных. Процессы могут вызывать процедуры монитора, но не имеют доступа к внутренним данным монитора.*

Мониторы имеют важное свойство, которое делает их полезными для достижения взаимного исключения: *только один процесс может быть активным по отношению к монитору.*

Компилятор обрабатывает вызовы процедур монитора особым образом. Обычно, когда процесс вызывает процедуру монитора, то первые несколько инструкций этой процедуры проверяют, не активен ли какой-либо другой процесс по отношению к этому монитору. Если да, то вызывающий процесс приостанавливается, пока другой процесс не освободит монитор. Таким образом, *исключение входа нескольких процессов в монитор реализуется не программистом, а компилятором*, что делает ошибки менее вероятными.

В распределенных системах, состоящих из нескольких процессоров, каждый из которых имеет собственную оперативную память, *семафоры и мониторы оказываются непригодными*. В таких системах синхронизация может быть реализована только с помощью *обмена сообщениями*.

ТЕМА 5
НИТИ

Многозадачность - важнейшее свойство ОС. Для поддержки этого свойства ОС определяет и оформляет для себя те внутренние единицы работы, между которыми и будет разделяться процессор и другие ресурсы компьютера. Эти *внутренние единицы работы в разных ОС носят разные названия - задача, задание, процесс, нить.* В некоторых случаях сущности, обозначаемые этими понятиями, принципиально отличаются друг от друга.

Если говорить о *процессах*, то операционная система поддерживает их обособленность: *у каждого процесса имеется свое виртуальное адресное пространство, каждому процессу назначаются свои ресурсы - файлы, окна, семафоры и т.д.* Такая обособленность нужна для того, чтобы защитить один процесс от другого, поскольку они, совместно используя все ресурсы машины, конкурируют друг с другом. В общем случае процессы принадлежат разным пользователям, разделяющим один компьютер, и ОС берет на себя роль арбитра в спорах процессов за ресурсы.

При мультипрограммировании повышается пропускная способность системы, но отдельный процесс никогда не может быть выполнен быстрее, чем если бы он выполнялся в однопрограммном режиме. Однако *задача, решаемая в рамках одного процесса, может обладать внутренним параллелизмом, который в принципе позволяет ускорить ее решение.* Например, в ходе выполнения задачи происходит обращение к внешнему устройству, и на время этой операции можно не блокировать полностью выполнение процесса, а продолжить вычисления по другой "ветви" процесса.

Для этих целей современные ОС предлагают использовать сравнительно новый механизм *многонитевой обработки (multithreading)*. При этом вводится новое понятие "нить" (thread), а понятие "процесс" в значительной степени меняет смысл.

Мультипрограммирование реализуется на уровне нитей, и задача, оформленная в виде нескольких нитей в рамках одного процесса, может быть выполнена быстрее за счет псевдопараллельного (или параллельного в мультипроцессорной системе) выполнения ее отдельных частей.

Примеры:

1. если электронная таблица была разработана с учетом возможностей многонитевой обработки, то пользователь может запросить пересчет своего рабочего листа и одновременно продолжать заполнять таблицу;
2. многонитевость особенно эффективна для выполнения распределенных приложений (многонитевый сервер может параллельно выполнять запросы сразу нескольких клиентов).

Нити, относящиеся к одному процессу, *не настолько изолированы друг от друга*, как процессы в традиционной многозадачной системе, между ними легко организовать тесное взаимодействие.

Действительно, *в отличие от процессов, которые принадлежат разным, вообще говоря, конкурирующим приложениям, все нити одного процесса всегда принадлежат одному приложению,* поэтому программист, пишущий это приложение, может заранее продумать работу множества нитей процесса таким образом, чтобы они могли взаимодействовать, а не бороться за ресурсы.

Часто бывает желательно иметь несколько нитей, разделяющих единое адресное пространство, но выполняющихся квазипараллельно, благодаря чему нити становятся подобными процессам (за исключением разделяемого адресного пространства).

Нити иногда называют *облегченными процессами или мини-процессами*.

Действительно, *нити* во многих отношениях *подобны процессам*. Каждая нить выполняется строго последовательно и имеет свой собственный программный счетчик и стек. Нити, как и процессы, могут порождать нити-потомки, могут переходить из состояния в состояние. Подобно традиционным процессам (то есть процессам, состоящим из одной нити), нити могут находится в одном из следующих состояний: ВЫПОЛНЕНИЕ, ОЖИДАНИЕ и ГОТОВНОСТЬ. Пока одна нить заблокирована, другая нить того же процесса может выполняться. Нити разделяют процессор так, как это делают процессы, в соответствии с различными вариантами планирования.

Однако *различные нити в рамках одного процесса не настолько независимы, как отдельные процессы*. ***Все такие нити имеют одно и то же адресное пространство***. Это означает, что они разделяют одни и те же глобальные переменные. Поскольку каждая нить может иметь доступ к каждому виртуальному адресу, одна нить может использовать стек другой нити.

Между нитями нет полной защиты, потому что, во-первых, это невозможно, а во-вторых, не нужно. Все нити одного процесса всегда *решают общую задачу одного пользователя*, и аппарат нитей используется здесь для более быстрого решения задачи путем ее распараллеливания.

Кроме разделения адресного пространства, все нити разделяют также набор открытых файлов, таймеров, сигналов и т.п.

Итак, ***нити имеют собственные***:

1. *программный счетчик;*
2. *регистры;*
3. *стек;*
4. *состояние;*
5. *нити-потомки.*

***Нити разделяют**:*

- *адресное пространство;*
- *глобальные переменные;*
- *открытые файлы;*
- *таймеры;*
- *семафоры;*
- *статистическую информацию.*

Многонитевая обработка повышает эффективность работы системы по сравнению с многозадачной обработкой. Например, в многозадачной среде Windows можно одновременно работать с электронной таблицей и текстовым редактором. Однако, если пользователь запрашивает пересчет своего рабочего листа, электронная таблица блокируется до тех пор, пока эта операция не завершится, что может потребовать значительного времени. В многонитевой среде в случае, если электронная таблица была разработана с учетом возможностей многонитевой обработки, предоставляемых программисту, этой проблемы не возникает, и пользователь всегда имеет доступ к электронной таблице.

Широкое применение находит многонитевая обработка в распределенных системах.

Некоторые прикладные задачи легче программировать, используя параллелизм. Например, задачи типа "писатель-читатель", в которых одна нить выполняет запись в буфер, а другая считывает записи из него. Поскольку они разделяют общий буфер, не стоит их делать отдельными процессами.

Другой пример использования нитей – это управление сигналами, такими как прерывание с клавиатуры (del или break). Вместо обработки сигнала прерывания, одна нить назначается для постоянного ожидания поступления сигналов. Таким образом, использование нитей может сократить необходимость в прерываниях пользовательского уровня. В этих примерах не столь важно параллельное выполнение, сколь важна ясность программы.

Наконец, *в мультипроцессорных системах для нитей из одного адресного пространства имеется возможность выполняться параллельно на разных процессорах.* Это действительно один из главных путей реализации разделения ресурсов в таких системах. С другой стороны, правильно сконструированные программы, которые используют нити, должны работать одинаково хорошо как на однопроцессорной машине в режиме разделения времени между нитями, так и на настоящем мультипроцессоре.

ТЕМА 6

УПРАВЛЕНИЕ ПАМЯТЬЮ. ТИПЫ АДРЕСОВ. МЕТОДЫ РАСПРЕДЕЛЕНИЯ ПАМЯТИ. РАСПРЕДЕЛЕНИЕ ПАМЯТИ ФИКСИРОВАННЫМИ, ДИНАМИЧЕСКИМИ, ПЕРЕМЕЩАЕМЫМИ РАЗДЕЛАМИ

Память является важнейшим ресурсом, требующим тщательного управления со стороны мультипрограммной операционной системы. *Распределению подлежит вся оперативная память, не занятая операционной системой.* Обычно ОС располагается в самых младших адресах, однако может занимать и самые старшие адреса.

Функциями ОС по управлению памятью являются:

1. отслеживание свободной и занятой памяти;
2. выделение памяти процессам и освобождение памяти при завершении процессов;
3. вытеснение процессов из оперативной памяти на диск, когда размеры основной памяти не достаточны для размещения в ней всех процессов, и возвращение их в оперативную память, когда в ней освобождается место;
4. настройка адресов программы на конкретную область физической памяти.

6.1 ТИПЫ АДРЕСОВ

Для идентификации переменных и команд используются ***символьные имена*** *(метки),* ***виртуальные адреса*** *и* ***физические адреса*** (рисунок 6.1).

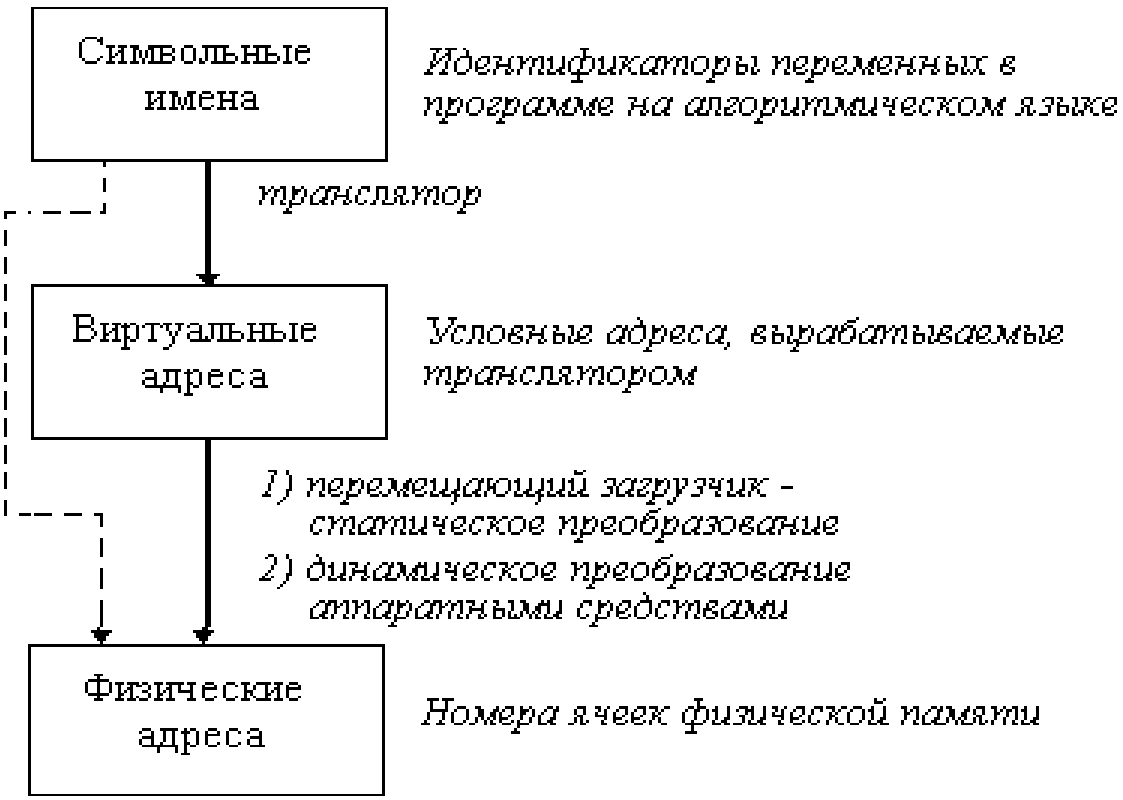


Рис. 6.1. Типы адресов

Символьные имена *присваивает пользователь* при написании программы на алгоритмическом языке или ассемблере.

Виртуальные адреса *вырабатывает транслятор*, переводящий программу на машинный язык. Так как во время трансляции, в общем случае не известно, в какое место оперативной памяти будет загружена программа, то транслятор присваивает переменным и командам виртуальные (условные) адреса, обычно считая по умолчанию, что программа будет размещена, начиная с нулевого адреса.

Совокупность виртуальных адресов процесса называется ***виртуальным адресным пространством***. Каждый процесс имеет собственное виртуальное адресное пространство. *Максимальный размер виртуального адресного пространства ограничивается разрядностью адреса, присущей данной архитектуре компьютера*, и, как правило, не совпадает с объемом физической памяти, имеющимся в компьютере.

Физические адреса *соответствуют номерам ячеек оперативной памяти*, где в действительности расположены или будут расположены переменные и команды.

Переход от виртуальных адресов к физическим может осуществляться двумя способами.

В первом случае замену виртуальных адресов на физические делает специальная системная программа - *перемещающий загрузчик*. Перемещающий загрузчик на основании имеющихся у него исходных данных о начальном адресе физической памяти, в которую предстоит загружать программу, и информации, предоставленной транслятором об адресно-зависимых константах программы, выполняет загрузку программы, совмещая ее с заменой виртуальных адресов физическими.

Второй способ заключается в том, что программа загружается в память в неизмененном виде в виртуальных адресах, при этом операционная система фиксирует смещение действительного расположения программного кода относительно виртуального адресного пространства. Во время выполнения программы при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический.

Второй способ является более гибким, он допускает перемещение программы во время ее выполнения, в то время как перемещающий загрузчик жестко привязывает программу к первоначально выделенному ей участку памяти.

Вместе с тем использование перемещающего загрузчика уменьшает накладные расходы, так как *в первом случае преобразование каждого виртуального адреса происходит только один раз во время загрузки, а во втором случае - каждый раз при обращении по данному адресу.*

В некоторых случаях (обычно в специализированных системах), когда заранее точно известно, в какой области оперативной памяти будет выполняться программа, транслятор выдает исполняемый код сразу в физических адресах.

6.2 МЕТОДЫ РАСПРЕДЕЛЕНИЯ ПАМЯТИ

Все методы управления памятью могут быть разделены на *два класса*:

1. методы, которые используют перемещение процессов между оперативной памятью и диском;
2. методы, которые не делают этого (рисунок 6.2).

Рис. 6.2 Классификация методов распределения памяти

6.3 РАСПРЕДЕЛЕНИЕ ПАМЯТИ ФИКСИРОВАННЫМИ РАЗДЕЛАМИ

Самым простым способом управления оперативной памятью является разделение ее на несколько разделов фиксированной величины.

Это может быть выполнено:

1. вручную оператором во время старта системы;
2. во время генерации системы.

Очередная задача, поступившая на выполнение, помещается либо в общую очередь (рисунок 6.3(а)), либо в очередь к некоторому разделу (рисунок 6.3(б)).

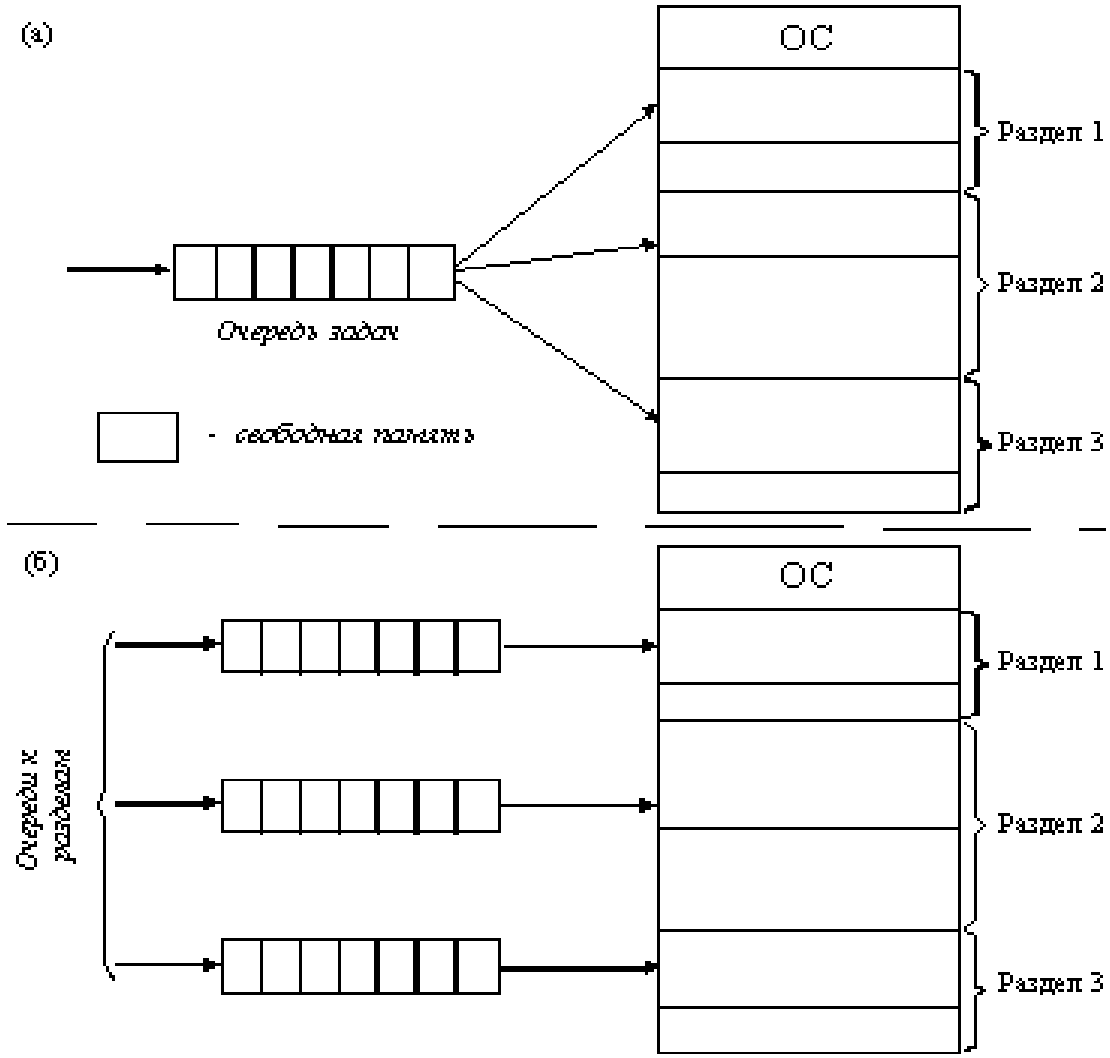


Рис. 6.3. Распределение памяти фиксированными разделами: а – с общей очередью; б – с отдельными очередями

Подсистема управления памятью в этом случае выполняет следующие задачи:

1. сравнивая размер программы, поступившей на выполнение, и свободных разделов, выбирает подходящий раздел;
2. осуществляет загрузку программы и настройку адресов.

При очевидном преимуществе – простоте реализации – данный метод имеет существенный *недостаток – жесткость.*

Первый недостаток: В каждом разделе может выполняться только одна программа, поэтому *уровень мультипрограммирования заранее ограничен числом разделов*, не зависимо от того, какой размер имеют программы. Даже если программа имеет небольшой объем, она будет занимать весь раздел, что приводит к неэффективному использованию памяти.

Второй недостаток: даже если объем оперативной памяти машины позволяет выполнить некоторую программу, разбиение памяти на разделы не позволяет сделать этого.

6.4 РАСПРЕДЕЛЕНИЕ ПАМЯТИ ДИНАМИЧЕСКИМИ РАЗДЕЛАМИ

Память машины не делится заранее на разделы.

1. Сначала вся память свободна;
2. Каждой вновь поступающей задаче выделяется необходимая ей память. Если достаточный объем памяти отсутствует, то задача не принимается на выполнение и стоит в очереди;
3. После завершения задачи память освобождается, и на это место может быть загружена другая задача.

Таким образом, в произвольный момент времени оперативная память представляет собой случайную последовательность занятых и свободных участков (разделов) произвольного размера.

На рисунке 6.4 показано состояние памяти в различные моменты времени при использовании динамического распределения:

1. В момент времени **t0** в памяти находится только ОС;
2. К моменту времени **t1** память разделена между 5 задачами, причем задача П4, завершаясь, покидает память;
3. В момент времени **t2** участок памяти, занимаемый ранее задачей П4 свободен.
4. В момент времени **t3** на освободившееся после задачи П4 место загружается задача П6.

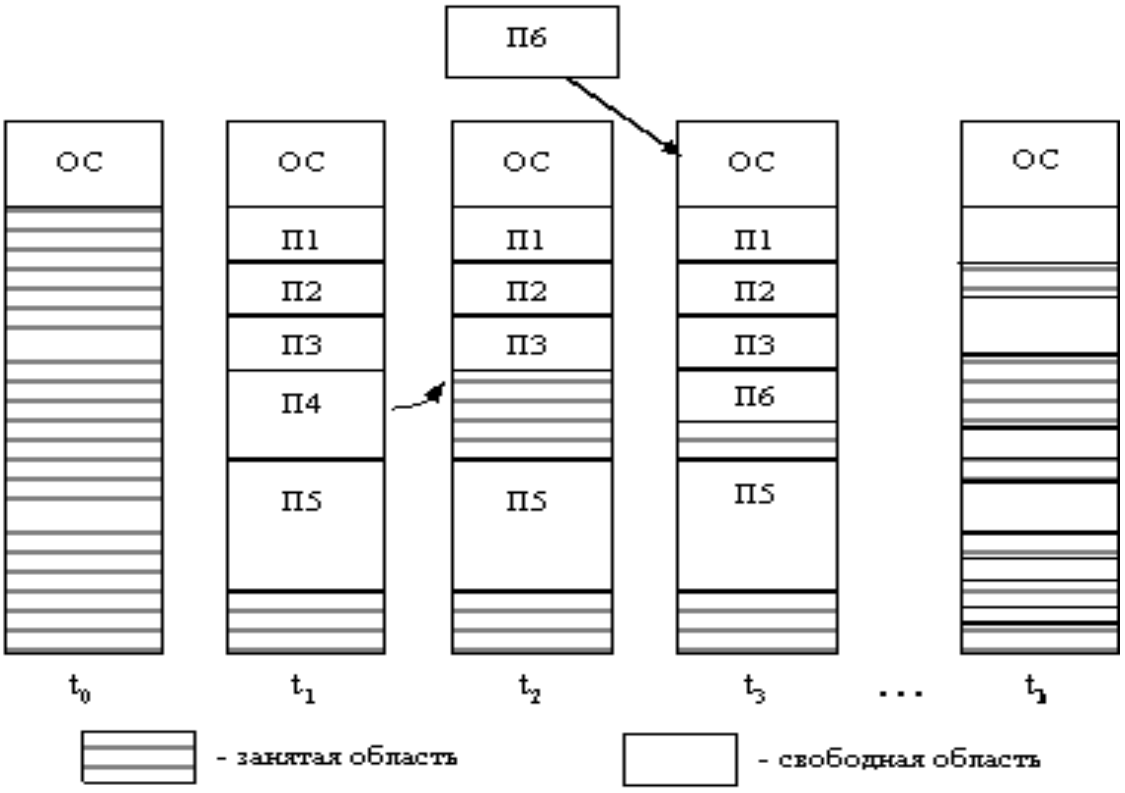


Рис. 6.4. Распределение памяти динамическими разделами

Задачами операционной системы при реализации данного метода управления памятью является:

1. ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти;
2. при поступлении новой задачи - анализ запроса, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен для размещения поступившей задачи;
3. загрузка задачи в выделенный ей раздел и корректировка таблиц свободных и занятых областей;
4. после завершения задачи корректировка таблиц свободных и занятых областей.

Программный код во время выполнения не перемещается, то есть может быть проведена единовременная настройка адресов с использованием перемещающего загрузчика.

Выбор раздела для вновь поступившей задачи может осуществляться по разным правилам, таким, например, как:

1. "первый попавшийся раздел достаточного размера";
2. "раздел, имеющий наименьший достаточный размер";
3. "раздел, имеющий наибольший достаточный размер".

Все эти правила имеют свои преимущества и недостатки.

По сравнению с методом распределения памяти фиксированными разделами данный метод обладает гораздо большей гибкостью, но ему присущ очень серьезный недостаток - *фрагментация памяти*.

Фрагментация - это наличие большого числа несмежных участков свободной памяти очень маленького размера (фрагментов). Настолько маленького, что ни одна из вновь поступающих программ не может поместиться ни в одном из участков, хотя суммарный объем фрагментов может составить значительную величину, намного превышающую требуемый объем памяти.

6.5 РАСПРЕДЕЛЕНИЕ ПАМЯТИ ПЕРЕМЕЩАЕМЫМИ РАЗДЕЛАМИ

Одним из методов борьбы с фрагментацией является перемещение всех занятых участков в сторону старших либо в сторону младших адресов, так, чтобы вся свободная память образовывала единую свободную область (рисунок 6.5). В дополнение к функциям, которые выполняет ОС при распределении памяти переменными разделами, в данном случае она должна еще *время от времени копировать содержимое разделов из одного места памяти в другое, корректируя таблицы свободных и занятых областей.* Эта процедура называется *"сжатием"*.

Сжатие может выполняться:

1. при каждом завершении задачи;
2. только тогда, когда для вновь поступившей задачи нет свободного раздела достаточного размера.

В первом случае требуется меньше вычислительной работы при корректировке таблиц, а во втором - реже выполняется процедура сжатия.

Так как программы перемещаются по оперативной памяти в ходе своего выполнения, то преобразование адресов из виртуальной формы в физическую форму должно выполняться динамическим способом.

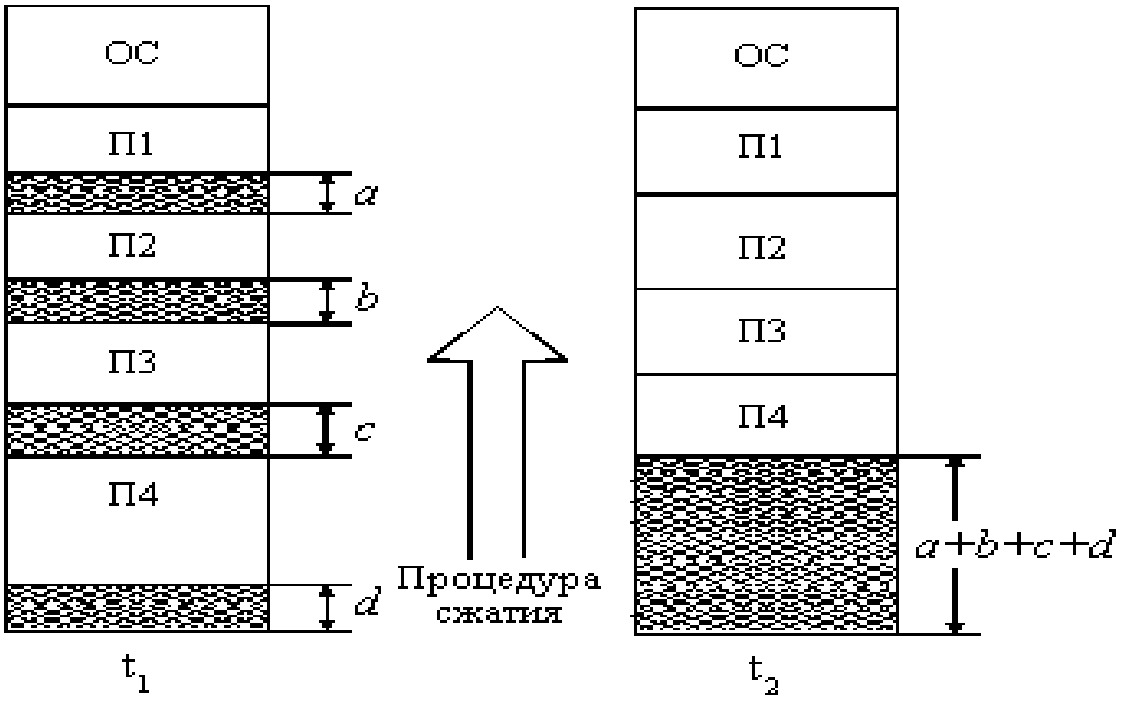


Рис. 6.5. Распределение памяти перемещаемыми разделами

Хотя процедура сжатия и приводит к более эффективному использованию памяти, она может потребовать значительного времени, что часто перевешивает преимущества данного метода.

ТЕМА 7

ПОНЯТИЕ ВИРТУАЛЬНОЙ ПАМЯТИ. РЕАЛИЗАЦИИ ВИРТУАЛЬНОЙ ПАМЯТИ: СТРАНИЧНОЕ, СЕГМЕНТНОЕ, СТРАНИЧНО-СЕГМЕНТНОЕ РАСПРЕДЕЛЕНИЕ, СВОПИНГ.

7.1 ПОНЯТИЕ ВИРТУАЛЬНОЙ ПАМЯТИ

Пользователи уже достаточно давно столкнулись с проблемой размещения в памяти программ, размер которых превышал имеющуюся в наличии свободную память.

Самым первым решением было разбиение программы на части, называемые *оверлеями*. 0-ой оверлей начинал выполняться первым. Когда он заканчивал свое выполнение, он вызывал другой оверлей. Все *оверлеи хранились на диске и перемещались между памятью и диском средствами операционной системы*. Однако разбиение программы на части и *планирование их загрузки в оперативную память должен был осуществлять программист.*

Развитие методов организации вычислительного процесса в этом направлении привело к появлению метода, известного под названием *виртуальная память*.

Виртуальным *называется ресурс, который пользователю или пользовательской программе представляется обладающим свойствами, которыми он в действительности не обладает.*

Например, пользователю может быть предоставлена *виртуальная оперативная память*, размер которой превосходит всю имеющуюся в системе реальную оперативную память. Пользователь пишет программы так, как будто в его распоряжении имеется однородная оперативная память большого объема, но в действительности все данные, используемые программой, хранятся на одном или нескольких разнородных запоминающих устройствах, обычно на дисках, и при необходимости частями отображаются в реальную память.

Виртуальная память *- это совокупность программно-аппаратных средств, позволяющих пользователям писать программы, размер которых превосходит имеющуюся оперативную память.*

Для этого *виртуальная память решает следующие задачи:*

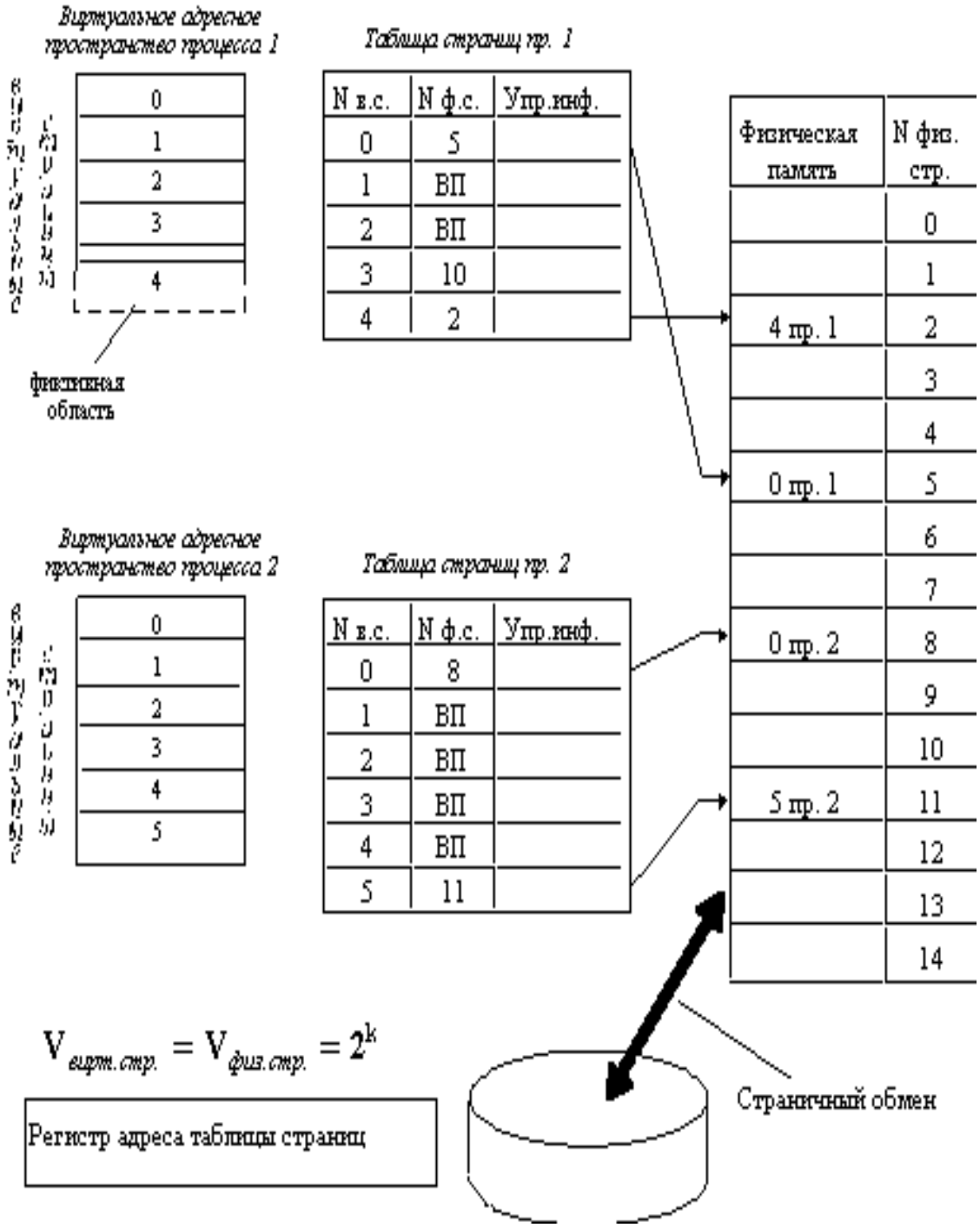
1. размещает данные в запоминающих устройствах разного типа, например, часть программы в оперативной памяти, а часть на диске;
2. перемещает по мере необходимости данные между запоминающими устройствами разного типа, например, подгружает нужную часть программы с диска в оперативную память;
3. преобразует виртуальные адреса в физические.

Эти действия выполняются *автоматически*, без участия программиста, то есть *механизм виртуальной памяти является прозрачным по отношению к пользователю.*

Наиболее распространенными реализациями виртуальной памяти являются страничное, сегментное, странично-сегментное распределение памяти и свопинг.

7.2. СТРАНИЧНОЕ РАСПРЕДЕЛЕНИЕ

На рисунке 7.1 показана схема страничного распределения памяти.



$$V_{вирт.стр.} = V_{физ.стр.} = 2^k$$

Рис. 7.1. Страничное распределение памяти

1. Виртуальное адресное пространство каждого процесса делится на *части одинакового, фиксированного для данной системы размера, называемые виртуальными страницами*.
2. В общем случае размер виртуального адресного пространства не является кратным размеру страницы, поэтому последняя страница каждого процесса дополняется *фиктивной областью*.
3. Вся оперативная память машины также делится на части такого же размера, называемые *физическими страницами* (или блоками). Размер страницы обычно выбирается равным степени двойки: 512, 1024 и т.д., это позволяет упростить механизм преобразования адресов.

Система с сегментной организацией функционирует следующим образом:

1. При загрузке процесса *часть его виртуальных страниц помещается в оперативную память, а остальные – на диск*. Смежные виртуальные *страницы не обязательно располагаются в смежных физических страницах.*
2. При загрузке операционная система создает для каждого процесса *информационную структуру – таблицу страниц*, в которой устанавливается *соответствие между номерами виртуальных и физических страниц для страниц, загруженных в оперативную память, или делается отметка о том, что виртуальная страница выгружена на диск*. Кроме того, в таблице страниц содержится *управляющая информация*, такая как:
   2.1. признак модификации страницы;
   2.2. признак невыгружаемости (выгрузка некоторых страниц может быть запрещена);
   2.3. признак обращения к странице (используется для подсчета числа обращений за определенный период времени)
   2.4. другие данные, формируемые и используемые механизмом виртуальной памяти.
3. При активизации очередного процесса *в специальный регистр процессора загружается адрес таблицы страниц данного процесса*.
4. При каждом обращении к памяти происходит чтение из таблицы страниц информации о виртуальной странице, к которой произошло обращение. Если данная виртуальная страница находится в оперативной памяти, то *выполняется преобразование виртуального адреса в физический адрес*.
5. Если нужная виртуальная страница в данный момент выгружена на диск, то происходит *страничное прерывание.* Выполняющийся процесс переводится в состояние ожидания, и активизируется другой процесс из очереди готовых. Параллельно программа обработки страничного прерывания находит на диске требуемую виртуальную страницу и пытается загрузить ее в оперативную память. Если в памяти имеется свободная физическая страница, то загрузка выполняется немедленно, если же таких страниц нет, то решается вопрос, какую страницу следует выгрузить из оперативной памяти.

В данной ситуации может быть использовано много разных *критериев выбора*. Наиболее популярные из них следующие:

1. дольше всего не использовавшаяся страница;
2. первая попавшаяся страница;
3. страница, к которой в последнее время было меньше всего обращений.

В некоторых системах используется понятие рабочего множества страниц. *Рабочее множество определяется для каждого процесса и представляет собой перечень наиболее часто используемых страниц, которые должны постоянно находиться в оперативной памяти и поэтому не подлежат выгрузке.*

После того, как выбрана страница, которая должна покинуть оперативную память, анализируется ее *признак модификации*. Если выталкиваемая страница с момента загрузки была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то соответствующая физическая страница просто объявляется свободной.

Механизм преобразования виртуального адреса в физический адрес при страничной организации памяти показан на рисунке 7.2.

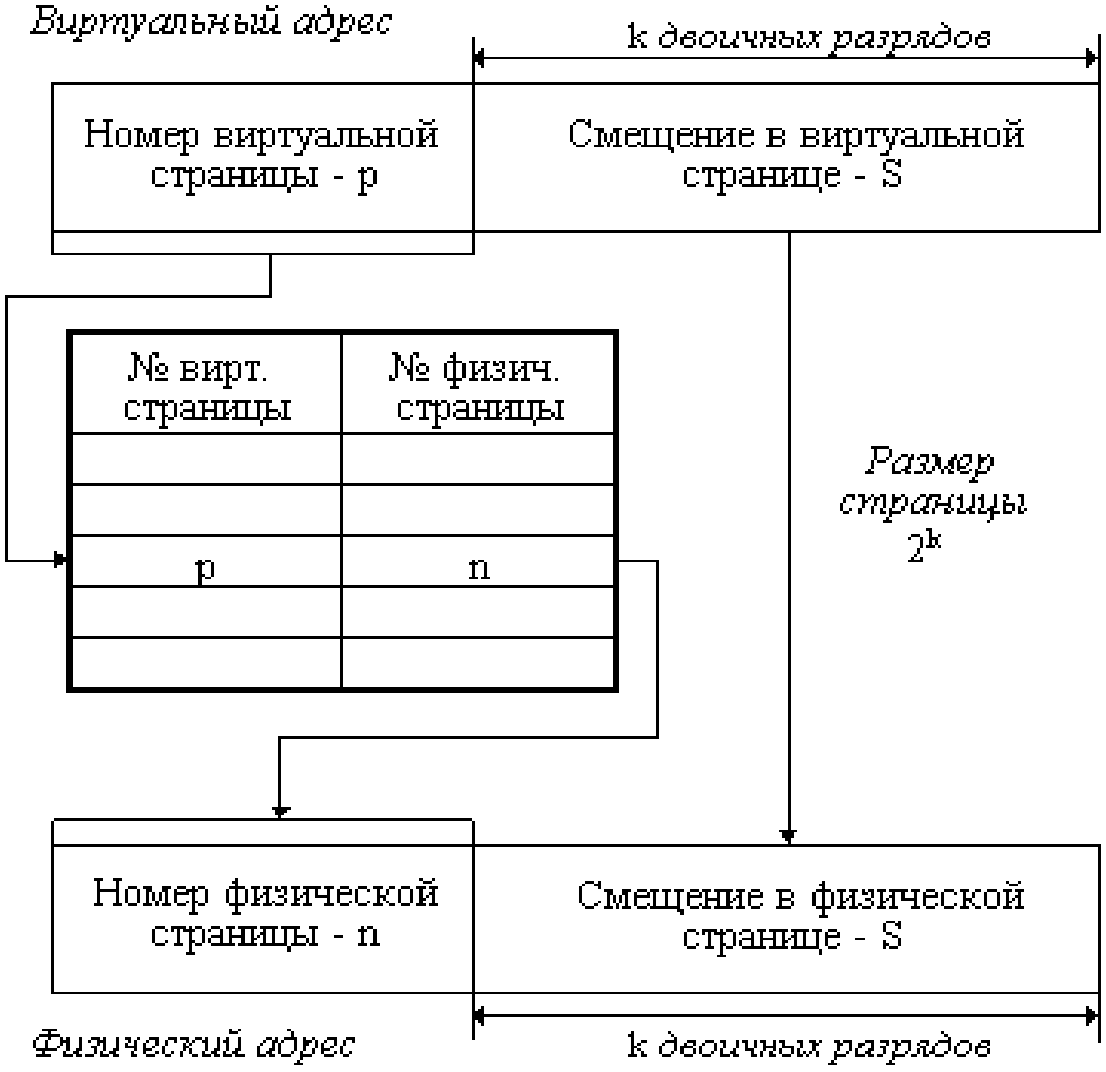


Рис. 7.2. Механизм преобразования виртуального адреса в физический при страничной организации памяти

Виртуальный адрес при страничном распределении может быть представлен в виде пары **(p, s)**, где **p** – номер виртуальной страницы процесса (нумерация страниц начинается с 0), а **s** – смещение в пределах виртуальной страницы. Так как размер страницы равен 2 в степени **k**, то смещение **s** может быть получено простым отделением **k** младших разрядов в двоичной записи виртуального адреса. Оставшиеся старшие разряды представляют собой двоичную запись номера страницы **p**.

При каждом обращении к оперативной памяти аппаратными средствами выполняются следующие действия:

1. на основании *начального адреса таблицы страниц* (содержимое регистра адреса таблицы страниц), *номера виртуальной страницы* (старшие разряды виртуального адреса) и *длины записи в таблице страниц* (системная константа) определяется адрес нужной записи в таблице,
2. из этой записи извлекается *номер физической страницы*,
3. к номеру физической страницы *присоединяется смещение* (младшие разряды виртуального адреса).

Использование в пункте 3 того факта, что размер страницы равен степени 2, позволяет применить операцию конкатенации (присоединения) вместо более длительной операции сложения, что уменьшает время получения физического адреса, а значит, повышает производительность компьютера.

На производительность системы со страничной организацией памяти влияют:

1. *Временные затраты, связанные с обработкой страничных прерываний*. При часто возникающих страничных прерываниях система может тратить большую часть времени впустую, на свопинг страниц. Чтобы уменьшить частоту страничных прерываний, следовало бы увеличивать размер страницы. Кроме того, увеличение размера страницы уменьшает размер таблицы страниц, а значит, уменьшает затраты памяти. С другой стороны, если страница велика, значит, велика и фиктивная область в последней виртуальной странице каждой программы. В среднем на каждой программе теряется половина объема страницы, что в сумме при большой странице может составить существенную величину.
2. *Временные затраты, связанные преобразованием виртуального адреса в физический адрес.* Время преобразования виртуального адреса в физический определяется временем доступа к таблице страниц. В связи с этим ее стремятся размещать в "быстрых" запоминающих устройствах (набор специальных регистров или память, использующая для уменьшения времени доступа ассоциативный поиск и кэширование данных).

Страничное распределение памяти может быть реализовано в упрощенном варианте, без выгрузки страниц на диск. В этом случае все виртуальные страницы всех процессов постоянно находятся в оперативной памяти. Такой вариант страничной организации хотя и не предоставляет пользователю виртуальной памяти, но почти исключает фрагментацию за счет того, что программа может загружаться в несмежные области, а также того, что при загрузке виртуальных страниц никогда не образуется остатков.

7.3 СЕГМЕНТНОЕ РАСПРЕДЕЛЕНИЕ

При страничной организации виртуальное адресное пространство процесса делится механически на равные части.

Это *не позволяет дифференцировать способы доступа к разным частям программы* (сегментам), а это свойство часто является очень полезным. Например, можно запретить обращаться с операциями записи и чтения в кодовый сегмент программы, а для сегмента данных разрешить только чтение. Кроме того, разбиение программы на "осмысленные" части делает принципиально возможным разделение одного сегмента несколькими процессами. Например, если два процесса используют одну и ту же математическую подпрограмму, то в оперативную память может быть загружена только одна копия этой подпрограммы.

Схема сегментного распределения памяти, реализующего эти возможности, приведена на рисунке 7.3.

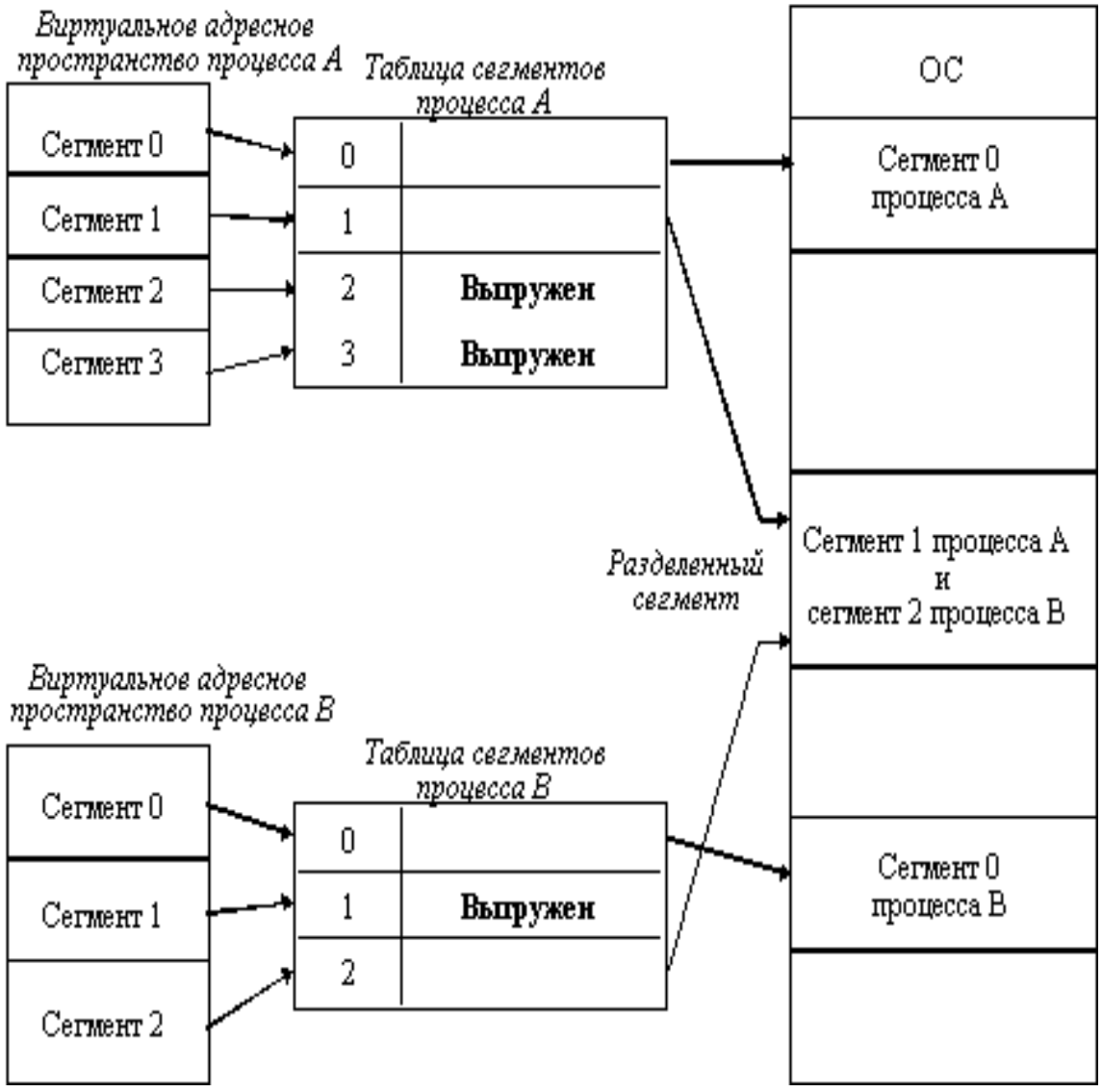


Рис. 7.3. Распределение памяти сегментами

1. *Виртуальное адресное пространство процесса делится на сегменты*, размер которых определяется программистом с учетом смыслового значения содержащейся в них информации. Отдельный сегмент может представлять собой подпрограмму, массив данных и т.п. Иногда сегментация программы выполняется по умолчанию компилятором.
2. *При загрузке процесса часть сегментов помещается в оперативную память* (при этом для каждого из этих сегментов операционная система подыскивает подходящий участок свободной памяти), а *часть сегментов размещается в дисковой памяти*. Сегменты одной программы могут занимать в оперативной памяти несмежные участки.
3. Во время загрузки система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается:
   3.1. начальный физический адрес сегмента в оперативной памяти;
   3.2. размер сегмента;
   3.3. правила доступа;
   3.4. признак модификации;
   3.5. признак обращения к данному сегменту за последний интервал времени;
   3.6. некоторая другая информация.
4. Если виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

Система с сегментной организацией функционирует аналогично системе со страничной организацией:

1. время от времени происходят прерывания, связанные с отсутствием нужных сегментов в памяти;
2. при необходимости освобождения памяти некоторые сегменты выгружаются;
3. при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический адрес;
4. кроме того, при обращении к памяти проверяется, разрешен ли доступ требуемого типа к данному сегменту.

Виртуальный адрес при сегментной организации памяти может быть представлен парой **(g, s)**, где **g** – номер сегмента, а **s** – смещение в сегменте. *Физический адрес получается путем сложения начального физического адреса сегмента, найденного в таблице сегментов по номеру g, и смещения s.*

Недостатками данного метода распределения памяти являются:

1. фрагментация на уровне сегментов;
2. более медленное по сравнению со страничной организацией преобразование адреса.

7.4 СТРАНИЧНО-СЕГМЕНТНОЕ РАСПРЕДЕЛЕНИЕ

Данный метод представляет собой комбинацию страничного и сегментного и поэтому сочетает в себе достоинства обоих подходов.

1. *Виртуальное пространство процесса делится на сегменты*;
2. Каждый *сегмент в свою очередь делится на виртуальные страницы*, которые нумеруются в пределах сегмента.
3. *Оперативная память делится на физические страницы*.

Функционирование системы со странично-сегментной организацией:

1. Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске.
2. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении.
3. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса.
4. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс.

Схема преобразования виртуального адреса в физический показана на рис. 7.4

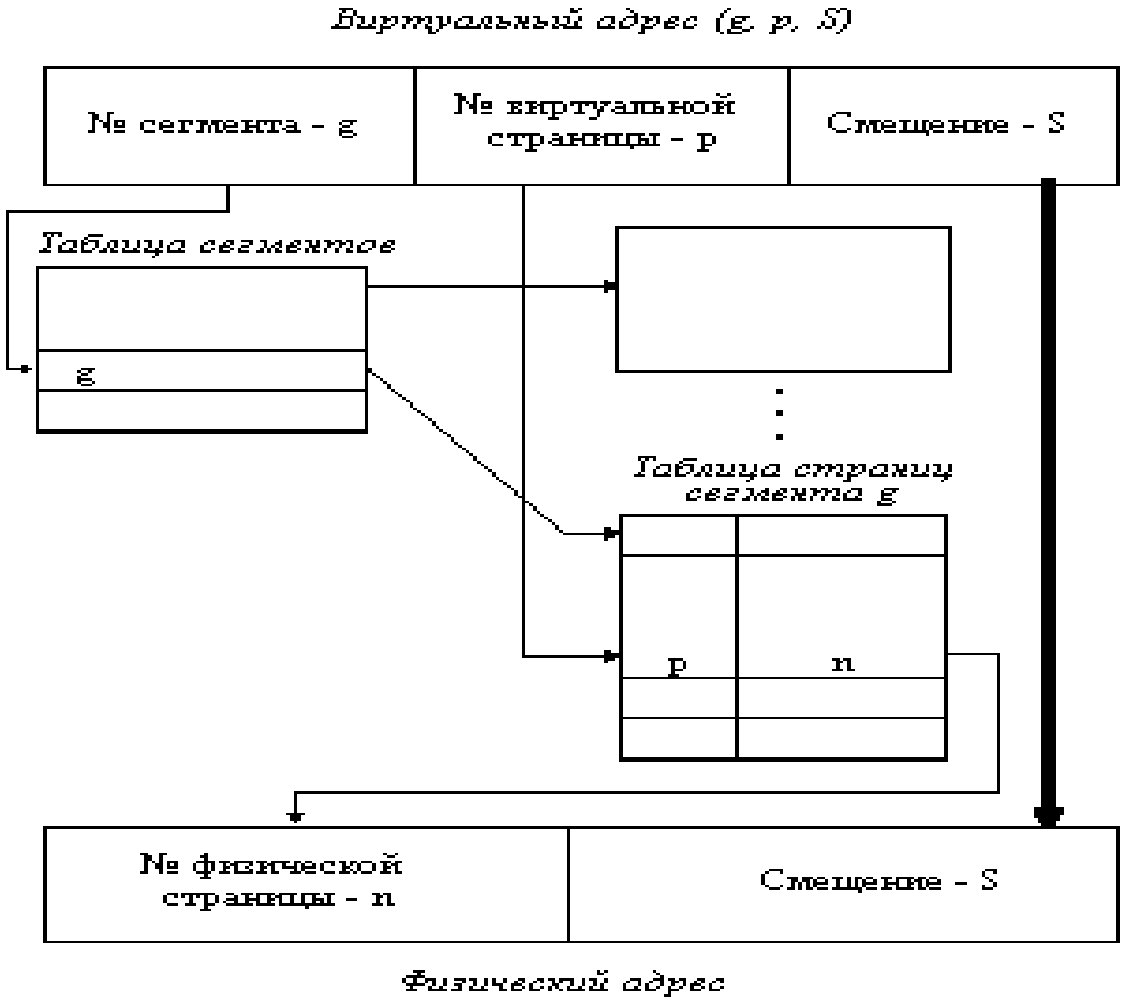


Рис. 7.4. Схема преобразования виртуального адреса в физический для сегментно-страничной организации памяти

7.5 СВОПИНГ

Разновидностью виртуальной памяти является *свопинг*. На рисунке 7.5 показан график зависимости коэффициента загрузки процессора в зависимости от числа одновременно выполняемых процессов и доли времени, проводимого этими процессами в состоянии ожидания ввода-вывода.

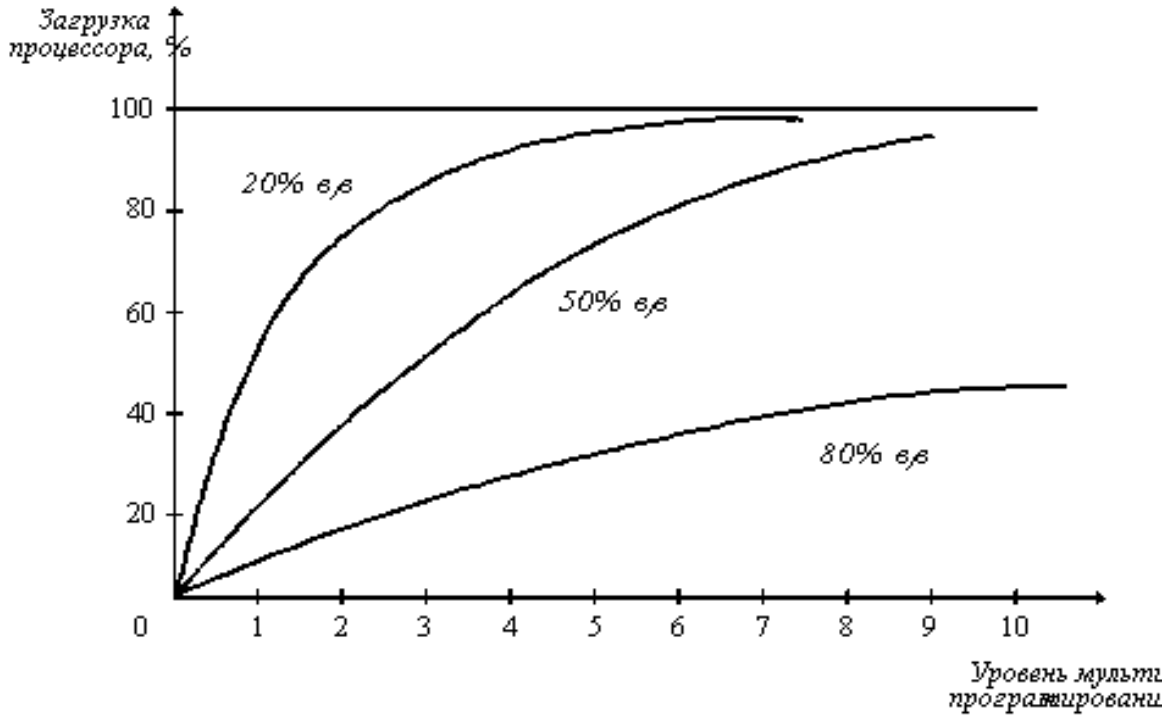


Рис. 7.5. Зависимость загрузки процессора от числа задач и интенсивности ввода-вывода

Видно, что для загрузки процессора на 90% достаточно трех счетных задач. Для того же, чтобы обеспечить такую же загрузку интерактивными задачами (ведут интенсивный ввод-вывод), потребуются десятки таких задач.

Необходимым условием для выполнения задачи является загрузка ее в оперативную память, объем которой ограничен. В этих условиях был предложен *метод организации вычислительного процесса, называемый* ***свопингом***. В соответствии с этим методом *некоторые процессы (обычно находящиеся в состоянии ожидания) временно выгружаются на диск. Планировщик операционной системы не исключает их из своего рассмотрения, и при наступлении условий активизации некоторого процесса, находящегося в области свопинга на диске, этот процесс перемещается в оперативную память. Если свободного места в оперативной памяти не хватает, то выгружается другой процесс.*

При свопинге, в отличие от рассмотренных ранее методов реализации виртуальной памяти, *процесс перемещается между памятью и диском целиком, то есть в течение некоторого времени процесс может полностью отсутствовать в оперативной памяти.* Существуют различные алгоритмы выбора процессов на загрузку и выгрузку, а также различные способы выделения оперативной и дисковой памяти загружаемому процессу.

ТЕМА 8

ИЕРАРХИЯ ЗАПОМИНАЮЩИХ УСТРОЙСТВ. ПРИНЦИП КЭШИРОВАНИЯ ДАННЫХ

8.1 ИЕРАРХИЯ ЗАПОМИНАЮЩИХ УСТРОЙСТВ

Память вычислительной машины представляет собой иерархию запоминающих устройств (ЗУ):

1. внутренние регистры процессора;
2. различные типы сверхоперативной и оперативной памяти;
3. диски;
4. флэш-память;
5. и т.п.

Запоминающие устройства *отличаются средним временем доступа и стоимостью хранения данных в расчете на один бит* (рисунок 8.1). Пользователь хотел бы иметь и недорогую и быструю память. *Кэш-память представляет некоторое компромиссное решение этой проблемы.*

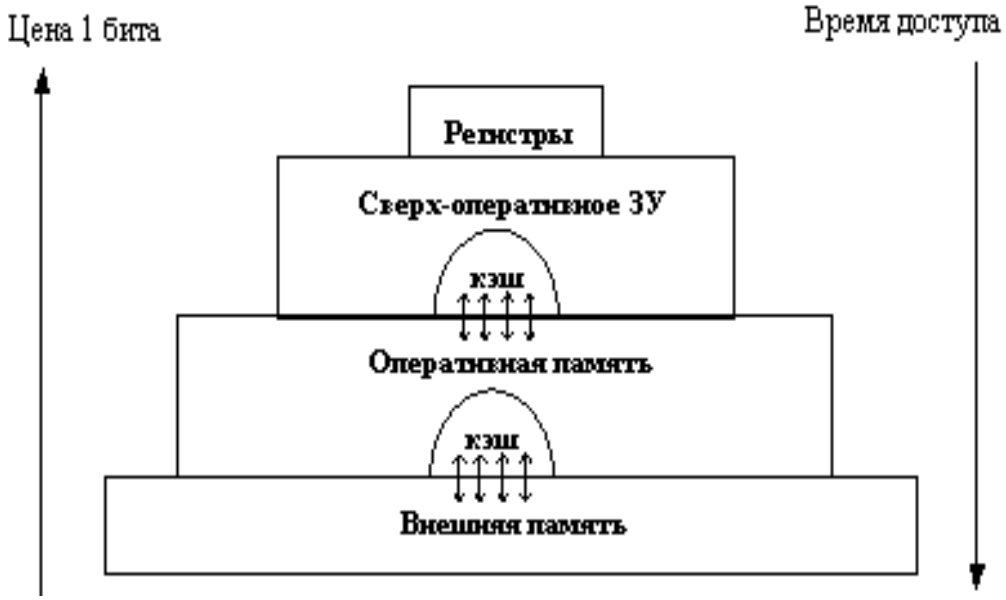


Рис. 8.1 Иерархия запоминающих устройств

8.2 ПРИНЦИП КЭШИРОВАНИЯ ДАННЫХ

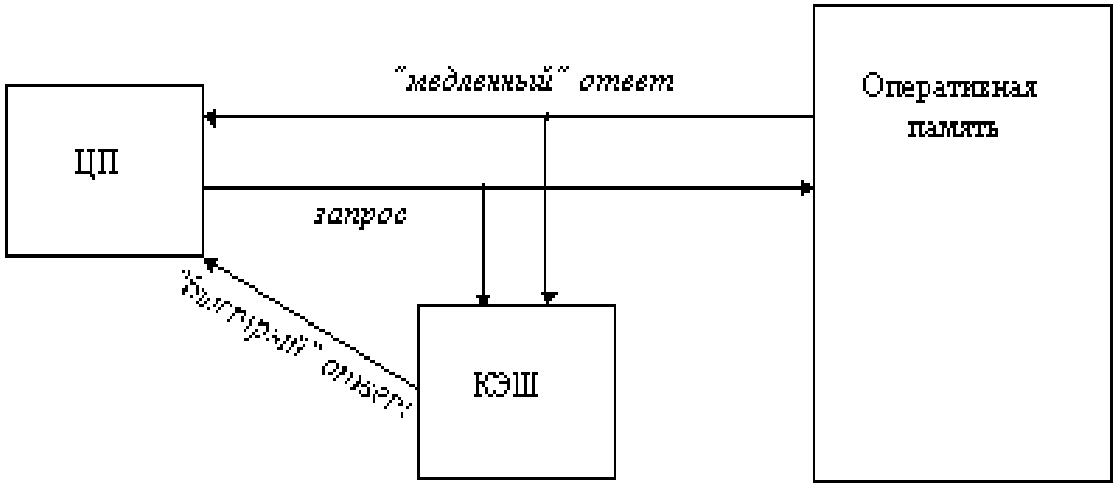
Кэш-память *– это способ организации совместного функционирования двух типов запоминающих устройств, отличающихся временем доступа и стоимостью хранения данных, который позволяет уменьшить среднее время доступа к данным за счет динамического копирования в "быстрое" ЗУ наиболее часто используемой информации из "медленного" ЗУ.*

Кэш-памятью часто называют не только способ организации работы двух типов запоминающих устройств, но и одно из устройств - "быстрое" ЗУ. Оно стоит дороже и, как правило, имеет сравнительно небольшой объем. Важно, что *механизм кэш-памяти является прозрачным для*

пользователя, который не должен сообщать никакой информации об интенсивности использования данных и не должен никак участвовать в перемещении данных из ЗУ одного типа в ЗУ другого типа. Все это делается автоматически системными средствами.

Рассмотрим частный случай *использования кэш-памяти для уменьшения среднего времени доступа к данным, хранящимся в оперативной памяти*.

Для этого между процессором и оперативной памятью помещается быстрое ЗУ, называемое просто кэш-памятью (рисунок 8.2). В качестве такового может быть использована, например, ассоциативная память.



Структура кэш-памяти

Адрес данных в ОП	Данные	Управл. информация	
		бит модиф.	бит обращ.

Рис. 8.2 Кэш-память

Содержимое кэш-памяти представляет собой совокупность записей обо всех загруженных в нее элементах данных. Каждая запись об элементе данных включает в себя:

1. адрес, который этот элемент данных имеет в оперативной памяти;
2. управляющую информацию: признак модификации и признак обращения к данным за некоторый последний период времени.

В системах, оснащенных кэш-памятью, каждый запрос к оперативной памяти выполняется в соответствии со следующим алгоритмом:

1. Просматривается содержимое кэш-памяти с целью определения, не находятся ли нужные данные в кэш-памяти (кэш-память не является адресуемой, поэтому поиск нужных данных осуществляется по содержимому - значению поля "адрес в оперативной памяти", взятому из запроса);
2. Если данные обнаруживаются в кэш-памяти, то они считываются из нее, и результат передается в процессор.
3. Если нужных данных нет, то они вместе со своим адресом копируются из оперативной памяти в кэш-память, и результат выполнения запроса передается в процессор.
4. При копировании данных может оказаться, что в кэш-памяти нет свободного места, тогда выбираются данные, к которым в последний период было меньше всего обращений, для вытеснения из кэш-памяти.
5. Если вытесняемые данные были модифицированы за время нахождения в кэш-памяти, то они переписываются в оперативную память. Если нет, то их место в кэш-памяти объявляется свободным.

На практике в кэш-память считывается не один элемент данных, к которому произошло обращение, а целый блок данных. Это увеличивает вероятность так называемого "попадания в кэш" (нахождение нужных данных в кэш-памяти).

Покажем, как среднее время доступа к данным зависит от вероятности попадания в кэш. Пусть имеется основное запоминающие устройство со средним временем доступа к данным t1 и кэш-память, имеющая время доступа t2, очевидно, что t2<t1. Обозначим через t среднее время доступа к данным в системе с кэш-памятью, а через p -вероятность попадания в кэш. По формуле полной вероятности имеем:

```
t = t1*(1 - p) + t2*p
```

Из нее видно, что среднее время доступа к данным в системе с кэш-памятью линейно зависит от вероятности попадания в кэш и изменяется от среднего времени доступа в основное ЗУ (при p = 0) до среднего времени доступа непосредственно в кэш-память (при p = 1).

В реальных системах вероятность попадания в кэш составляет примерно 0,9. Высокое значение вероятности нахождения данных в кэш-памяти связано с наличием у данных объективных свойств: *пространственной и временной локальности.*

1. *Пространственная локальность*. Если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.
2. *Временная локальность*. Если произошло обращение по некоторому адресу, то следующее обращение по этому же адресу с большой вероятностью произойдет в ближайшее время.

Если рассмотреть оперативную память и внешнюю память, то в этом случае уменьшается среднее время доступа к данным, расположенным на диске, и роль кэш-памяти выполняет буфер в оперативной памяти.

ТЕМА 9

УПРАВЛЕНИЕ ВВОДОМ-ВЫВОДОМ. ФИЗИЧЕСКАЯ ОРГАНИЗАЦИЯ УСТРОЙСТВ ВВОДА-ВЫВОДА. ОРГАНИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ .

9.1 УПРАВЛЕНИЕ ВВОДОМ-ВЫВОДОМ

Одной из главных функций ОС является управление всеми устройствами ввода-вывода компьютера.

Операционная система должна:

1. передавать устройствам команды;
2. перехватывать прерывания;
3. обрабатывать ошибки;
4. обеспечивать интерфейс между устройствами и остальной частью системы.

В целях развития интерфейс должен быть одинаковым для всех типов устройств (независимость от устройств).

9.2 ФИЗИЧЕСКАЯ ОРГАНИЗАЦИЯ УСТРОЙСТВ ВВОДА-ВЫВОДА

Устройства ввода-вывода делятся на два типа:

1. блок-ориентированные устройства;
2. байт-ориентированные устройства.

Блок-ориентированные устройства хранят информацию в блоках фиксированного размера, каждый из которых имеет свой собственный адрес. Самое распространенное блок-ориентированное устройство – диск.

Байт-ориентированные устройства не адресуемы и не позволяют производить операцию поиска, они генерируют или потребляют последовательность байтов. Примерами являются терминалы, строчные принтеры, сетевые адаптеры.

Однако некоторые внешние устройства не относятся ни к одному классу, например, часы, которые, с одной стороны, не адресуемы, а с другой стороны, не порождают потока байтов. Это устройство только выдает сигнал прерывания в некоторые моменты времени.

Внешнее устройство обычно состоит из:

1. механического компонента;
2. электронного компонента.

Электронный компонент называется контроллером устройства или адаптером.

Механический компонент представляет собственно устройство.

Некоторые контроллеры могут управлять несколькими устройствами. Если интерфейс между контроллером и устройством стандартизован, то независимые производители могут выпускать совместимые как контроллеры, так и устройства.

Операционная система обычно имеет дело не с устройством, а с контроллером.

Контроллер, как правило, выполняет простые функции, например, преобразует поток бит в блоки, состоящие из байт, и осуществляют контроль и исправление ошибок.

Каждый контроллер имеет *несколько регистров, которые используются для взаимодействия с центральным процессором.* В некоторых компьютерах эти регистры являются частью физического адресного пространства. В таких компьютерах нет специальных операций ввода-вывода. В других компьютерах адреса регистров ввода-вывода, называемых часто портами, образуют собственное адресное пространство за счет введения специальных операций ввода-вывода (например, команд IN и OUT в процессорах i86).

ОС выполняет ввод-вывод, записывая команды в регистры контроллера. Например, контроллер гибкого диска IBM PC принимает 15 команд, таких как READ, WRITE, SEEK, FORMAT и т.д. *Когда команда принята, процессор оставляет контроллер и занимается другой работой. При завершении команды контроллер организует прерывание для того, чтобы передать управление процессором операционной системе, которая должна проверить результаты операции. Процессор получает результаты и статус устройства, читая информацию из регистров контроллера.*

9.3 ОРГАНИЗАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ВВОДА-ВЫВОДА

Основная идея организации программного обеспечения ввода-вывода состоит в *разбиении его на несколько уровней*.

Нижние уровни обеспечивают экранирование особенностей аппаратуры от верхних уровней.

Верхние уровни, в свою очередь, обеспечивают удобный интерфейс для пользователей.

Ключевые принципы и проблемы организации программного обеспечения устройств ввода-вывода:

1. *Независимость от устройств*. Вид программы не должен зависеть от того, читает ли она данные с гибкого диска или с жесткого диска.
2. *Единообразное именование*. Для именования устройств должны быть приняты единые правила.
3. *Обработка ошибок*. Ошибки следует обрабатывать как можно ближе к аппаратуре. Если контроллер обнаруживает ошибку чтения, то он должен попытаться ее скорректировать. Если же это ему не удается, то исправлением ошибок должен заняться драйвер устройства. Многие ошибки могут исчезать при повторных попытках выполнения операций ввода-вывода, например, ошибки, вызванные наличием пылинок на головках чтения или на диске. И только если нижний уровень не может справиться с ошибкой, он сообщает об ошибке верхнему уровню.
4. *Использование блокирующих (синхронных) и неблокирующих (асинхронных) передач.* Большинство операций физического ввода-вывода выполняется асинхронно – процессор начинает передачу и переходит на другую работу, пока не наступает прерывание.

Пользовательские программы намного легче писать, если операции ввода-вывода блокирующие – после команды READ программа приостанавливается до тех пор, пока данные не попадут в буфер программы. ОС выполняет операции ввода-вывода асинхронно, но представляет их для пользовательских программ в синхронной форме.

5. *Работа с разделяемыми и выделенными устройствами*. Диски – это разделяемые устройства, так как одновременный доступ нескольких пользователей к диску не представляет собой проблему. Принтеры – это выделенные устройства, потому что нельзя смешивать строчки, печатаемые различными пользователями. Наличие выделенных устройств создает для операционной системы некоторые проблемы.

Для решения поставленных проблем *программное обеспечение ввода-вывода делится на четыре слоя* (рисунок 9.1):

1. Обработка прерываний;
2. Драйверы устройств;
3. Независимый от устройств слой операционной системы;
4. Пользовательский слой программного обеспечения.

Рис. 9.1 Многоуровневая организация подсистемы ввода-вывода

9.3.1 Обработка прерываний

Прерывания должны быть скрыты как можно глубже в недрах операционной системы, чтобы как можно меньшая часть ОС имела с ними дело. Наилучший способ состоит в разрешении процессу, инициировавшему операцию ввода-вывода, блокировать себя до завершения операции и наступления прерывания.

Процесс может блокировать себя, используя:

1. вызов DOWN для семафора;
2. вызов WAIT для переменной условия;
3. вызов RECEIVE для ожидания сообщения.

При наступлении прерывания *процедура обработки прерывания выполняет разблокирование процесса*, инициировавшего операцию ввода-вывода, *используя вызовы UP, SIGNAL или посылая процессу сообщение*.

В любом случае эффект от прерывания будет состоять в том, что ранее заблокированный процесс теперь продолжит свое выполнение.

9.3.2. Драйверы устройств

Весь зависимый от устройства код помещается в драйвер устройства. Каждый драйвер управляет устройствами одного типа или, может быть, одного класса.

В операционной системе *только драйвер устройства знает о конкретных особенностях какого-либо устройства*. Например, только драйвер диска имеет дело с дорожками, секторами, цилиндрами, временем установления головки и другими факторами, обеспечивающими правильную работу диска.

Драйвер устройства принимает запрос от устройств программного слоя и решает, как его выполнить. Типичным запросом является чтение n блоков данных. Если драйвер был свободен во время поступления запроса, то он начинает выполнять запрос немедленно. Если же он был занят обслуживанием другого запроса, то вновь поступивший запрос присоединяется к очереди уже имеющихся запросов, и он будет выполнен, когда наступит его очередь.

Первый шаг в реализации запроса ввода-вывода, например, для диска, состоит в *преобразовании его из абстрактной формы в конкретную*. Для дискового драйвера это означает преобразование номеров блоков в номера цилиндров, головок, секторов, проверку, работает ли мотор, находится ли головка над нужным цилиндром. Другими словами, *он должен решить, какие операции контроллера нужно выполнить и в какой последовательности.*

После передачи команды контроллеру драйвер должен решить, блокировать ли себя до окончания заданной операции или нет. Если операция занимает значительное время, как при печати некоторого блока данных, то драйвер блокируется до тех пор, пока операция не завершится, и обработчик прерывания не разблокирует его. Если команда ввода-вывода выполняется быстро (например, прокрутка экрана), то драйвер ожидает ее завершения без блокирования.

9.3.3 Независимый от устройств слой операционной системы

Большая часть программного обеспечения ввода-вывода является независимой от устройств. Точная граница между драйверами и независимыми от устройств программами определяется системой, так как некоторые функции, которые могли бы быть реализованы независимым способом, в действительности выполнены в виде драйверов для повышения эффективности или по другим причинам.

Типичными функциями для независимого от устройств слоя являются:

1. обеспечение общего интерфейса к драйверам устройств;
2. именование устройств;
3. защита устройств;
4. обеспечение независимого размера блока;
5. буферизация;
6. распределение памяти на блок-ориентированных устройствах;
7. распределение и освобождение выделенных устройств;
8. уведомление об ошибках.

Рассмотрим некоторые функции данного перечня. Верхним слоям программного обеспечения не удобно работать с блоками разной величины, поэтому *данный слой обеспечивает единый размер блока,* например, за счет объединения нескольких различных блоков в единый логический блок. В связи с этим *верхние уровни имеют дело с абстрактными устройствами, которые используют единый размер логического блока независимо от размера физического сектора.*

При создании файла или заполнении его новыми данными необходимо выделить ему новые блоки. Для этого ОС должна вести список или битовую карту свободных блоков диска. На основании информации о наличии свободного места на диске может быть разработан алгоритм поиска свободного блока, независимый от устройства и реализуемый программным слоем, находящимся выше слоя драйверов.

9.3.4 Пользовательский слой программного обеспечения

Большая часть программного обеспечения ввода-вывода находится внутри ОС. Однако *некоторая его часть содержится в библиотеках, связываемых с пользовательскими программами.*

Системные вызовы, включающие вызовы ввода-вывода, обычно делаются библиотечными процедурами. Если программа, написанная на языке С, содержит вызов

```
count = write (fd, buffer, nbytes),
```

то библиотечная процедура **write** будет связана с программой.

Набор подобных процедур является частью системы ввода-вывода. В частности, форматирование ввода или вывода выполняется библиотечными процедурами. Примером может служить функция **printf** языка С, которая принимает строку формата и некоторые переменные в качестве входной информации, затем строит строку символов ASCII и делает вызов **write** для вывода этой строки. Стандартная библиотека ввода-

вывода содержит большое число процедур, которые выполняют ввод-вывод и работают как часть пользовательской программы.

Другой категорией программного обеспечения ввода-вывода является подсистема спулинга (spooling).

Спулинг *– это способ работы с выделенными устройствами в мультипрограммной системе.*

Типичное устройство, требующее спулинга, - строчный принтер. Технически легко позволить каждому пользовательскому процессу открыть специальный файл, связанный с принтером. Однако такой способ опасен из-за того, что пользовательский процесс может монополизировать принтер на произвольное время.

Вместо этого *создается специальный процесс - монитор, который получает исключительные права на использование этого устройства.* Также создается специальный каталог, называемый каталогом спулинга.

Для того, чтобы напечатать файл, пользовательский процесс помещает выводимую информацию в этот файл и помещает его в каталог спулинга.

Процесс-монитор по очереди распечатывает все файлы, содержащиеся в каталоге спулинга.

ТЕМА 10

ФАЙЛОВАЯ СИСТЕМА. ИМЕНА ФАЙЛОВ. ТИПЫ ФАЙЛОВ. ЛОГИЧЕСКАЯ ОРГАНИЗАЦИЯ ФАЙЛА. ФИЗИЧЕСКАЯ ОРГАНИЗАЦИЯ И АДРЕС ФАЙЛА. ПРАВА ДОСТУПА К ФАЙЛУ. КЭШИРОВАНИЕ ДИСКА

10.1 ФАЙЛОВАЯ СИСТЕМА

Файловая система *– это часть операционной системы, назначение которой состоит в том, чтобы обеспечить пользователю:*

1. *удобный интерфейс при работе с данными, хранящимися на диске;*
2. *обеспечить совместное использование файлов несколькими пользователями и процессами.*

В широком смысле понятие "файловая система" включает:

1. совокупность всех файлов на диске;
2. наборы структур данных, используемых для управления файлами, такие, например, как каталоги файлов, дескрипторы файлов, таблицы распределения свободного и занятого пространства на диске;
3. комплекс системных программных средств, реализующих управление файлами, в частности: создание, уничтожение, чтение, запись, именование, поиск и другие операции над файлами.

10.2 ИМЕНА ФАЙЛОВ

Файлы идентифицируются именами. Пользователи дают файлам символьные имена, при этом учитываются ограничения ОС как на используемые символы, так и на длину имени.

Ранее эти границы были весьма узкими. Так в файловой системе **FAT** длина имен ограничивается схемой 8.3 (8 символов – собственно имя, 3 символа – расширение имени), а в ОС UNIX System V имя не может содержать более 14 символов.

Пользователю гораздо удобнее работать с длинными именами, поскольку они позволяют дать файлу действительно мнемоническое название, по которому даже через достаточно большой промежуток времени можно будет вспомнить, что содержит этот файл. Поэтому современные файловые системы поддерживают длинные символьные имена файлов. Например, **Windows** в своей файловой системе **NTFS** устанавливает, что имя файла может содержать до 255 символов, не считая завершающего нулевого символа.

При переходе к длинным именам возникает *проблема совместимости* с ранее созданными приложениями, использующими короткие имена. Чтобы приложения могли обращаться к файлам в соответствии с принятыми ранее соглашениями, *файловая система должна уметь предоставлять эквивалентные короткие имена (псевдонимы) файлам, имеющим длинные имена*.

Длинные имена поддерживаются также файловыми системами **VFAT и FAT32.** Кроме *проблемы генерации эквивалентных коротких имен*, при реализации новых вариантов FAT важной задачей была *задача хранения длинных имен при условии, что принципиально метод хранения и структура данных на диске не должны были измениться.*

Обычно разные файлы могут иметь одинаковые символьные имена. В этом случае *файл однозначно идентифицируется так называемым составным именем, представляющем собой последовательность символьных имен каталогов.*

В некоторых системах одному и тому же файлу не может быть дано несколько разных имен, а в других такое ограничение отсутствует. В последнем случае операционная система присваивает файлу дополнительно *уникальное имя*, так, чтобы можно было установить взаимнооднозначное соответствие между файлом и его уникальным именем. *Уникальное имя представляет собой числовой идентификатор* и используется программами операционной системы. Примером такого уникального имени файла является *номер индексного дескриптора* в системе UNIX.

10.3 ТИПЫ ФАЙЛОВ

Файлы бывают следующих типов:

1. обычные файлы;
2. специальные файлы;
3. файлы-каталоги.

Обычные файлы подразделяются на:

1. текстовые;
2. двоичные.

Текстовые файлы — компьютерные файлы, содержащие текстовые данные, как правило, организованные в виде строк. Это могут быть документы, исходные тексты программ и т.п. Текстовые файлы можно прочитать на экране и распечатать на принтере.

Двоичные файлы часто имеют сложную внутреннюю структуру, например, объектный код программы или архивный файл. Все операционные системы должны уметь распознавать хотя бы один тип файлов – их собственные исполняемые файлы.

Специальные файлы – это файлы, ассоциированные с устройствами ввода-вывода, которые позволяют пользователю выполнять операции ввода-вывода, используя обычные команды записи в файл или чтения из файла. Эти команды обрабатываются вначале программами файловой системы, а затем на некотором этапе выполнения запроса преобразуются ОС в команды управления соответствующим устройством. *Специальные файлы, так же как и устройства ввода-вывода, делятся на блок-ориентированные и байт-ориентированные.*

Каталог – это, с одной стороны, *группа файлов, объединенных пользователем по некоторым соображениям*. Например, файлы,

содержащие программы игр, или файлы, составляющие один программный пакет. С другой стороны – это *файл, содержащий системную информацию о группе файлов, его составляющих*. В каталоге содержится список файлов, входящих в него, и устанавливается соответствие между файлами и их характеристиками (атрибутами).

В разных файловых системах могут использоваться в качестве атрибутов разные *характеристики*, например:

1. информация о разрешенном доступе;
2. пароль для доступа к файлу;
3. владелец файла;
4. создатель файла;
5. признак "только для чтения";
6. признак "скрытый файл";
7. признак "системный файл";
8. признак "архивный файл";
9. признак "двоичный/символьный";
10. признак "временный" (удалить после завершения процесса);
11. признак блокировки;
12. длина записи;
13. указатель на ключевое поле в записи;
14. длина ключа;
15. времена создания, последнего доступа и последнего изменения;
16. текущий размер файла;
17. максимальный размер файла.

Каталоги могут непосредственно содержать значения характеристик файлов, как это сделано в файловой системе MS-DOS, или ссылаться на таблицы, содержащие эти характеристики, как это реализовано в ОС UNIX (рисунок 10.1).

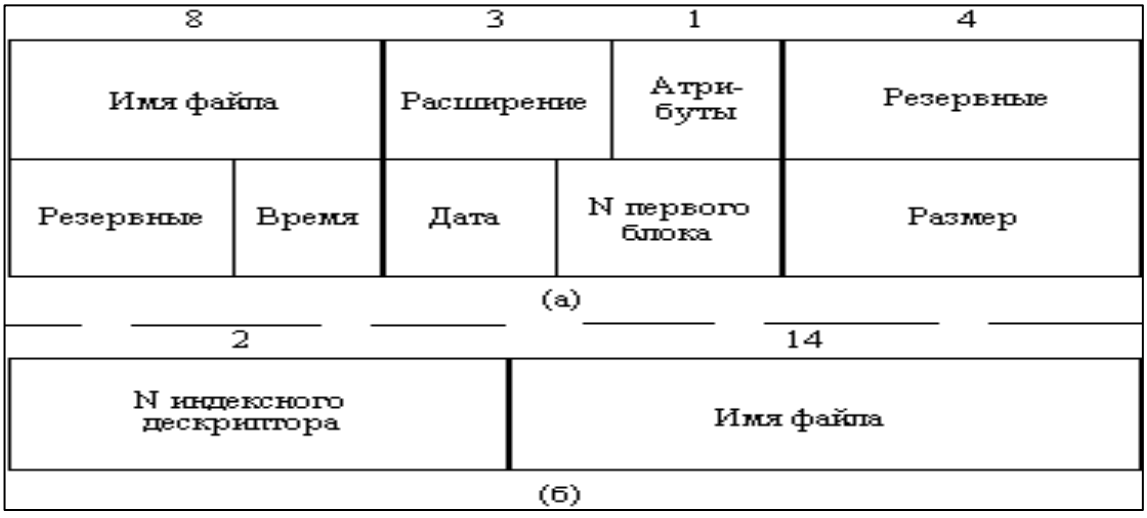


Рис. 10.1 Структура каталогов:
а – структура записи каталога MS-DOS (32 байта);
б – структура записи каталога ОС UNIX

Каталоги могут образовывать иерархическую структуру за счет того, что каталог более низкого уровня может входить в каталог более высокого уровня (рисунок 10.2).

Иерархия каталогов может быть деревом или сетью.

Каталоги образуют *дерево, если файлу разрешено входить только в один каталог*, и *сеть – если файл может входить сразу в несколько каталогов.* В MS-DOS каталоги образуют древовидную структуру, а в UNIX-сетевую. Как и любой другой файл, каталог имеет символьное имя и однозначно идентифицируется составным именем, содержащим цепочку символьных имен всех каталогов, через которые проходит путь от корня до данного каталога.

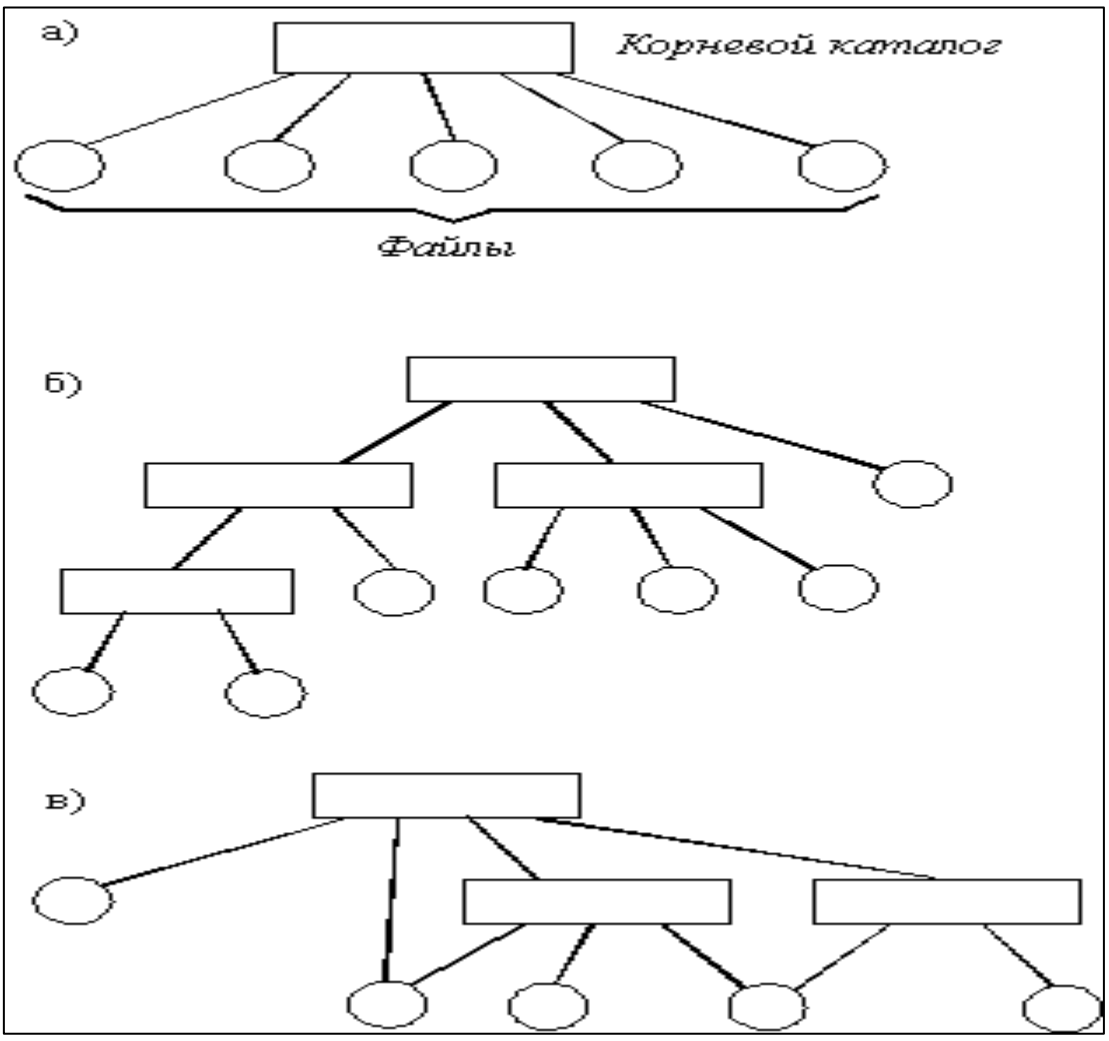


Рис. 10.2 Логическая организация файловой системы
а - одноуровневая; б - иерархическая (дерево); в - иерархическая (сеть)

10.4 ЛОГИЧЕСКАЯ ОРГАНИЗАЦИЯ ФАЙЛА

Программист имеет дело с логической организацией файла, *представляя файл в виде определенным образом организованных логических записей.*

Логическая запись - это наименьший элемент данных, которым может оперировать программист при обмене с внешним устройством.

Даже если физический обмен с устройством осуществляется большими единицами, операционная система обеспечивает программисту доступ к отдельной логической записи.

На рисунке 10.3 показаны несколько схем логической организации файла:

Рис. 10.3 Способы логической организации файлов

Записи могут быть:

1. фиксированной длины;
2. переменной длины;

Записи могут быть расположены в файле:

1. последовательно (последовательная организация);
2. в более сложном порядке, с использованием так называемых индексных таблиц, позволяющих обеспечить быстрый доступ к отдельной логической записи (индексно-последовательная организация). Для идентификации записи может быть использовано специальное поле записи, называемое ключом.

В файловых системах ОС UNIX и MS-DOS файл имеет простейшую логическую структуру - последовательность однобайтовых записей.

10.5 ФИЗИЧЕСКАЯ ОРГАНИЗАЦИЯ И АДРЕС ФАЙЛА

Физическая организация файла описывает правила расположения файла на устройстве внешней памяти, в частности на диске. Файл состоит из *физических записей – блоков*.

Блок – наименьшая единица данных, которой внешнее устройство обменивается с оперативной памятью.

Варианты физической организации файла показаны на рисунке 10.4.

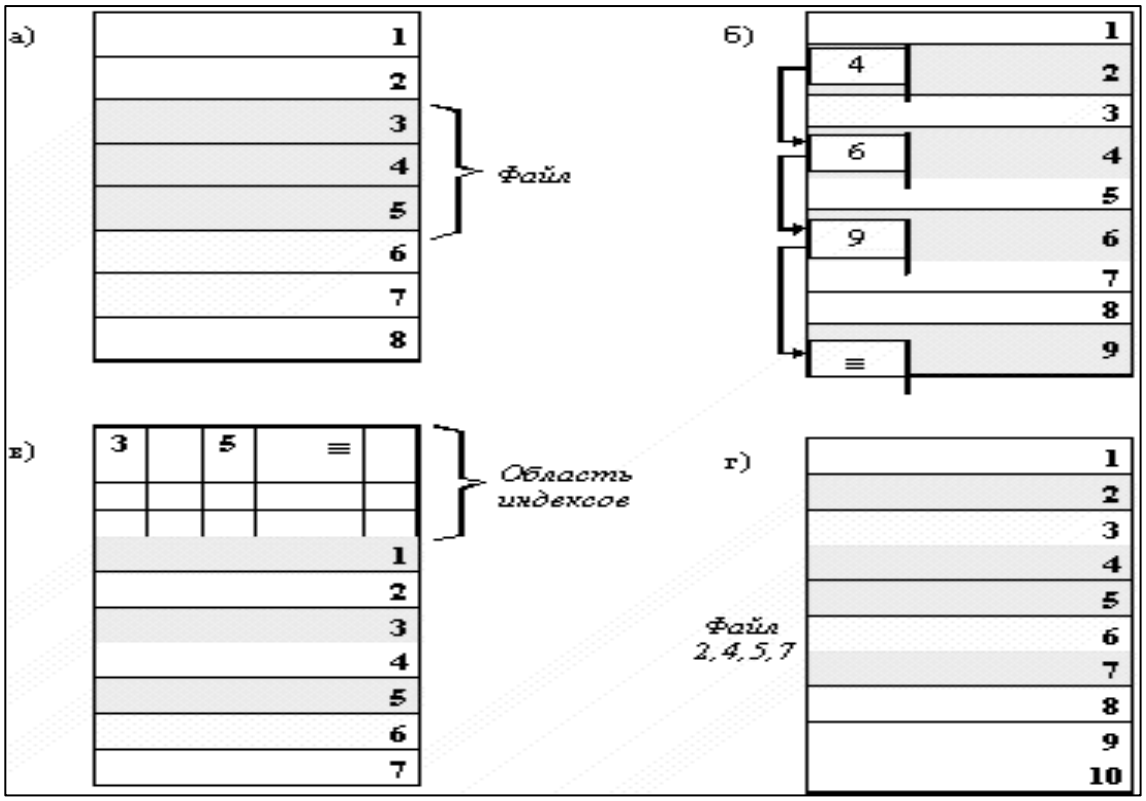


Рис. 10.4 . Физическая организация файла
а – непрерывное размещение; б – связанный список блоков;
в – связанный список индексов; г – перечень номеров блоков

Непрерывное размещение – простейший вариант физической организации (рисунок 10.4(а)), при котором *файлу предоставляется последовательность блоков диска, образующих единый сплошной участок дисковой памяти.*

Достоинства метода:

1. Для задания адреса файла достаточно указать только номер начального блока;
2. Простота.

Недостатки метода:

1. Во время создания файла заранее не известна его длина, а значит не известно, сколько памяти надо зарезервировать для этого файла;
2. при таком порядке размещения неизбежно возникает фрагментация, и пространство на диске используется не эффективно, так как отдельные участки маленького размера (минимально 1 блок) могут остаться не использованными.

Размещение в виде связанного списка блоков дисковой памяти показано на рисунке 10.4(б). При таком способе *в начале каждого блока содержится указатель на следующий блок*.

Достоинства метода:

1. Адрес файла также может быть задан одним числом - номером первого блока;
2. Каждый блок может быть присоединен в цепочку какого-либо файла, следовательно, фрагментация отсутствует;
3. Файл может изменяться во время своего существования, наращивая число блоков.

Недостатки метода:

1. Сложность реализации доступа к произвольно заданному месту файла (для того, чтобы прочитать пятый по порядку блок файла, необходимо последовательно прочитать четыре первых блока, прослеживая цепочку номеров блоков);
2. Количество данных файла, содержащихся в одном блоке, не равно степени двойки (одно слово израсходовано на номер следующего блока), а многие программы читают данные блоками, размер которых равен степени двойки.

Использование связанного списка индексов при физической организации файла является популярным способом, используемым, например, в файловой системе FAT операционной системы MS-DOS.

С каждым блоком связывается некоторый элемент – индекс. Индексы располагаются в отдельной области диска (в MS-DOS это таблица FAT). Если некоторый блок распределен некоторому файлу, то индекс этого блока содержит номер следующего блока данного файла.

Достоинства такой физической организации:

1. сохраняются все преимущества предыдущего способа;
2. для доступа к произвольному месту файла достаточно прочитать только блок индексов, отсчитать нужное количество блоков файла по цепочке и определить номер нужного блока;
3. данные файла занимают блок целиком, а значит, имеют объем, равный степени двойки.

Задание физического расположения файла путем простого перечисления номеров блоков, занимаемых этим файлом, используется, например, в ОС UNIX. Это обеспечивает фиксированную длину адреса, независимо от размера файла. Для хранения адреса файла выделено 13 полей. Если размер файла <= 10 блокам, то номера этих блоков непосредственно перечислены в первых десяти полях адреса. Если размер файла > 10 блоков, то 11-е поле содержит адрес блока, в котором могут быть расположены еще 128 номеров следующих блоков файла. Если файл > 10+128 блоков, то используется 12-е поле, в котором находится номер блока, содержащего 128 номеров блоков, которые содержат по 128 номеров блоков данного файла. Если файл > 10+128+128*128, то используется последнее 13-е поле для тройной косвенной адресации, что позволяет задать адрес файла, имеющего размер максимум 10+ 128 + 128*128 + 128*128*128.

10.6 ПРАВА ДОСТУПА К ФАЙЛУ

Определить права доступа к файлу – значит *определить для каждого пользователя набор операций, которые он может применить к данному файлу*. В разных файловых системах может быть определен свой список дифференцируемых операций доступа. Этот список может включать следующие *операции*:

1. создание файла;
2. уничтожение файла;
3. открытие файла;
4. закрытие файла;
5. чтение файла;
6. запись в файл;
7. дополнение файла;
8. поиск в файле;
9. получение атрибутов файла;
10. установка новых значений атрибутов;
11. переименование файла;
12. выполнение файла;
13. чтение каталога;
14. другие операции с файлами и каталогами.

В самом общем случае права доступа могут быть описаны матрицей прав доступа, в которой *столбцы соответствуют всем файлам системы, строки – всем пользователям*, а *на пересечении строк и столбцов указываются разрешенные операции* (рисунок 10.5).

Имена пользователей	Имена файлов			
	modern.txt	win.exe	class.dbf	unix.ppt
kira	читать	выполнять	–	выполнять
genya	читать	выполнять	–	выполнять читать
nataly	читать	–	–	выполнять читать
victor	читать писать	–	создать	–

Рис. 10.5 Матрица прав доступа

В некоторых системах пользователи могут быть разделены на *отдельные категории*. Для всех пользователей одной категории определяются единые права доступа.

Например, в системе UNIX все пользователи подразделяются на три категории: владельца файла, членов его группы и всех остальных.

Различают два основных подхода к определению прав доступа:

1. *избирательный доступ*, когда для каждого файла и каждого пользователя сам владелец может определить допустимые операции;
2. *мандатный подход*, когда система наделяет пользователя определенными правами по отношению к каждому разделяемому ресурсу (в данном случае файлу) в зависимости от того, к какой группе пользователь отнесен.

10.7 КЭШИРОВАНИЕ ДИСКА

В некоторых файловых системах запросы к внешним устройствам, в которых адресация осуществляется блоками (диски, ленты), перехватываются *промежуточным программным слоем - подсистемой буферизации.*

Подсистема буферизации представляет собой:

1. буферный пул, располагающийся в оперативной памяти;
2. комплекс программ, управляющих этим пулом.

Каждый буфер пула имеет размер, равный одному блоку.

Подсистема буферизации функционирует следующим образом:

1. При поступлении запроса на чтение некоторого блока подсистема буферизации просматривает свой буферный пул и, если находит, требуемый блок, то копирует его в буфер запрашивающего процесса (операция ввода-вывода считается выполненной, хотя физического обмена с устройством не происходило, очевиден выигрыш во времени при доступе к файлу);
2. Если нужный блок в буферном пуле отсутствует, то он считывается с устройства и одновременно с передачей запрашивающему процессу копируется в один из буферов подсистемы буферизации;
3. При отсутствии свободного буфера на диск вытесняется наименее используемая информация.

Таким образом, подсистема буферизации работает по принципу кэш-памяти.

ТЕМА 11

ОБЩАЯ МОДЕЛЬ ФАЙЛОВОЙ СИСТЕМЫ. ОТОБРАЖАЕМЫЕ В ПАМЯТЬ ФАЙЛЫ. СОВРЕМЕННЫЕ АРХИТЕКТУРЫ ФАЙЛОВЫХ СИСТЕМ

11.1 ОБЩАЯ МОДЕЛЬ ФАЙЛОВОЙ СИСТЕМЫ

Функционирование любой файловой системы можно представить многоуровневой моделью (рисунок 11.1).

Каждый уровень предоставляет некоторый интерфейс (набор функций) вышележащему уровню, а сам, в свою очередь, для выполнения своей работы использует интерфейс (обращается с набором запросов) нижележащего уровня.

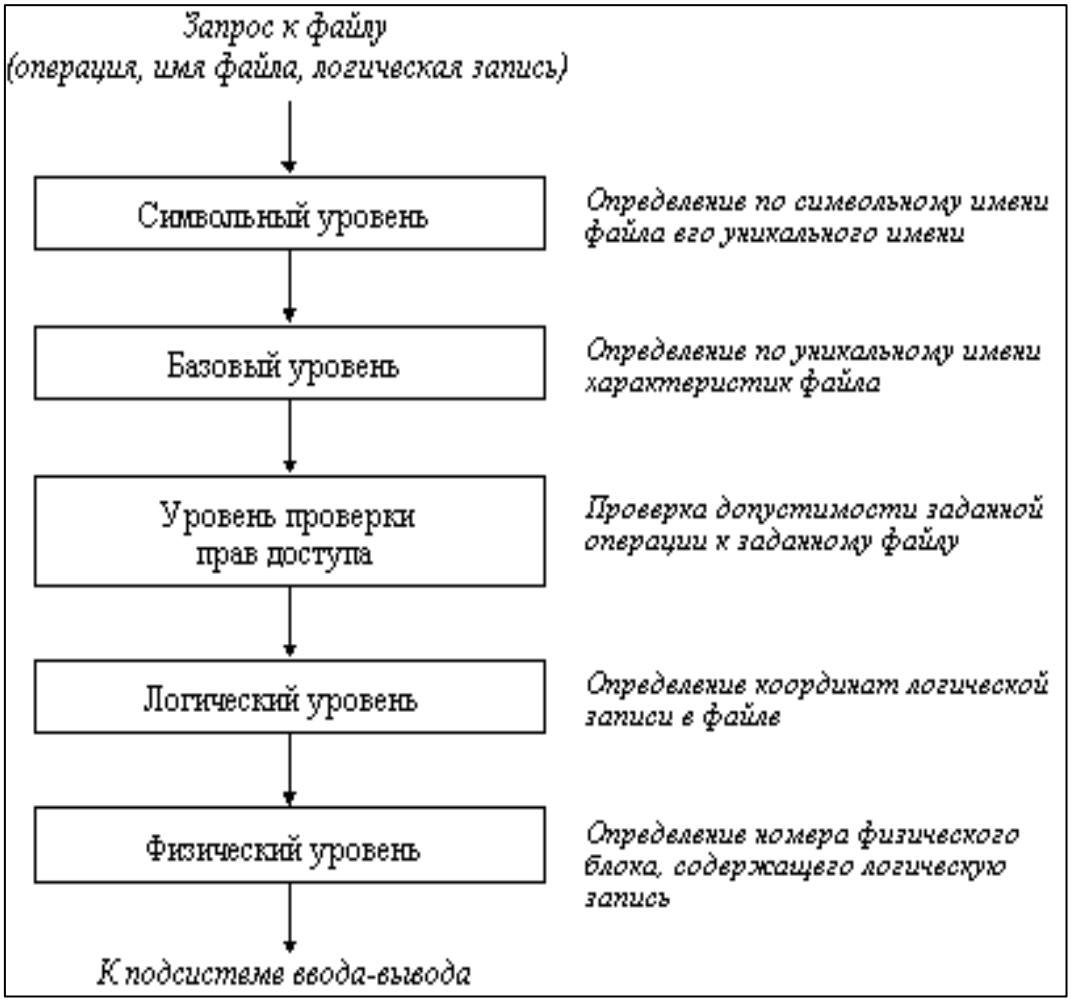


Рис. 11.1 Общая модель файловой системы

1. Задачей *символьного уровня является определение по символьному имени файла его уникального имени*.

В файловых системах, в которых каждый файл может иметь только одно символьное имя (например, MS-DOS), этот уровень отсутствует, так как символьное имя, присвоенное файлу пользователем, является одновременно уникальным. и может быть использовано операционной системой.

В других файловых системах, в которых один и тот же файл может иметь несколько символьных имен, на данном уровне просматривается цепочка каталогов для определения уникального имени файла. В файловой системе UNIX, например, уникальным именем является номер индексного дескриптора файла (i-node).

2. На *базовом уровне по уникальному имени файла определяются его характеристики: права доступа, адрес, размер и другие*.

Характеристики файла могут входить в состав каталога или храниться в отдельных таблицах. При открытии файла его характеристики перемещаются с диска в оперативную память, чтобы уменьшить среднее время доступа к файлу. В некоторых файловых системах (например, HPFS) при открытии файла вместе с его характеристиками в оперативную память перемещаются несколько первых блоков файла, содержащих данные.

3. Следующий этап реализации запроса к файлу - *проверка прав доступа* к нему.

Для этого сравниваются полномочия пользователя или процесса, выдавших запрос, со списком разрешенных видов доступа к данному файлу. Если запрашиваемый вид доступа разрешен, то выполнение запроса продолжается, если нет, то выдается сообщение о нарушении прав доступа.

4. Задача *логического уровня - определение координат запрашиваемой логической записи в файле*.

На этом этапе определяется, на каком расстоянии (в байтах) от начала файла находится требуемая логическая запись. При этом абстрагируются от физического расположения файла, он представляется в виде непрерывной последовательности байт. Алгоритм работы данного уровня зависит от логической организации файла. Например, если файл организован как последовательность логических записей фиксированной длины ***l***, то **n-ая** логическая запись имеет смещение ***l**(n-1)** байт. Для определения координат логической записи в файле с индексно-последовательной организацией выполняется чтение таблицы индексов (ключей), в которой непосредственно указывается адрес логической записи.

5. *На физическом уровне файловая система определяет номер физического блока, который содержит требуемую логическую запись, и смещение логической записи в физическом блоке.*

Для решения задачи используются результаты работы логического уровня:

1. смещение логической записи в файле;
2. адрес файла на внешнем устройстве,
3. сведения о физической организации файла, включая размер блока.

Рисунок 11.2 иллюстрирует работу физического уровня для простейшей физической организации файла в виде непрерывной последовательности блоков. Замечание: задача физического уровня решается независимо от того, как был логически организован файл.

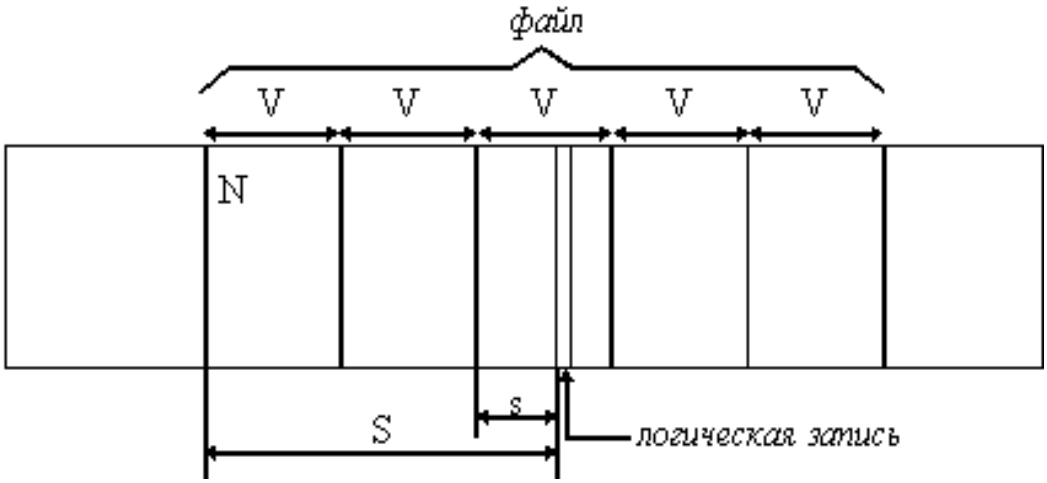


Рис. 11.2 Функции физического уровня файловой системы

Исходные данные:

V – размер блока

N – номер первого блока файла

S – смещение логической записи в файле

Требуется определить на физическом уровне:

n – номер блока, содержащего требуемую логическую запись

s – смещение логической записи в пределах блока

n = N + [S/V], где [S/V] - целая часть числа S/V

s = R [S/V] - дробная часть числа S/V

После определения номера физического блока, файловая система обращается к системе ввода-вывода для выполнения операции обмена с внешним устройством. В ответ на этот запрос в буфер файловой системы будет передан нужный блок, в котором на основании полученного при работе физического уровня смещения выбирается требуемая логическая запись.

11.2 ОТОБРАЖАЕМЫЕ В ПАМЯТЬ ФАЙЛЫ

Традиционный доступ к файлам выглядит запутанным и неудобным по сравнению с доступом к памяти, По этой причине некоторые ОС, начиная с MULTICS, обеспечивают *отображение файлов в адресное пространство выполняемого процесса.*

Это выражается в появлении двух новых системных вызовов: ***MAP (отобразить) и UNMAP (отменить отображение).*** Первый вызов передает операционной системе в качестве параметров имя файла и виртуальный адрес, и операционная система отображает указанный файл в виртуальное адресное пространство по указанному адресу.

Пусть файл ***f*** имеет длину 64 Кб и отображается на область виртуального адресного пространства с начальным адресом 512 К. После этого любая машинная команда, которая читает содержимое байта по адресу 512 К, получает 0-ой байт файла ***f*** и т.д. Очевидно, что запись по адресу 512 К + 1100 изменяет 1100 байт файла. При завершении процесса

на диске остается модифицированная версия файла, как если бы он был изменен комбинацией вызовов SEEK и WRITE.

При отображении файла внутренние системные таблицы изменяются так, чтобы *данный файл служил хранилищем страниц виртуальной памяти на диске.*

Таким образом, чтение по адресу 512 Кб вызывает страничное прерывание, в результате чего страница 0 переносится в физическую память. Аналогично, запись по адресу 512 К + 1100 вызывает страничное прерывание, в результате которого страница, содержащая этот адрес, перемещается в память, после чего осуществляется запись в память по требуемому адресу. Если эта страница вытесняется из памяти алгоритмом замены страниц, то она записывается обратно в файл в соответствующее его место.

При завершении процесса все отображенные и модифицированные страницы переписываются из памяти в файл.

Отображение файлов лучше всего работает в системе, которая поддерживает сегментацию. В такой системе каждый файл может быть отображен в свой собственный сегмент, так что k-ый байт в файле является k-ым байтом сегмента. На рисунке 11.3(а) изображен процесс, который имеет два сегмента: кода и данных. Пусть этот процесс копирует файлы. Для этого он сначала отображает файл-источник, например, abc. Затем он создает пустой сегмент и отображает на него файл назначения, например, файл ddd.

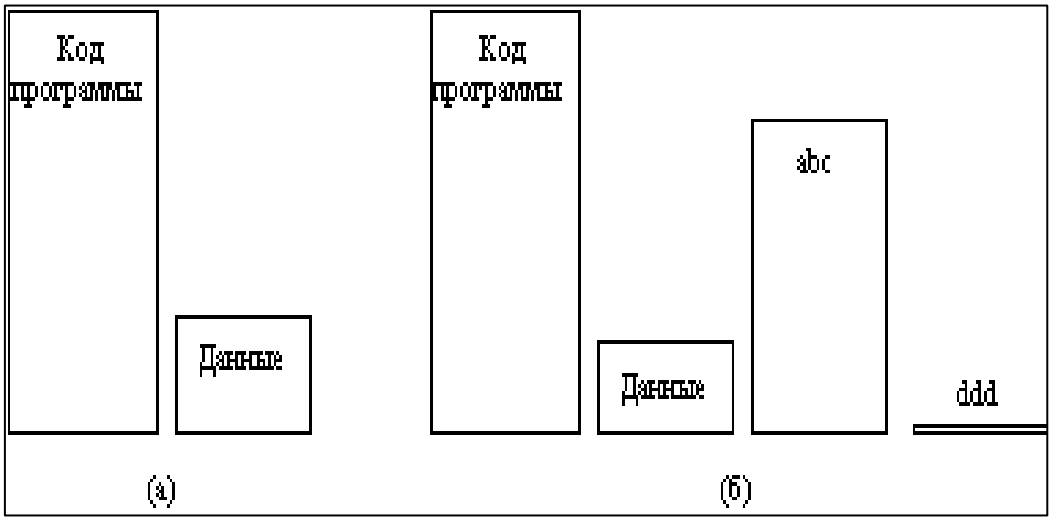


Рис. 11.3 Отображение файлов

(а) Сегменты процесса перед отображением файлов в адресное пространство;
(б) Процесс после отображения существующего файла abc в один сегмент и создания нового сегмента для файла ddd

С этого момента *процесс может копировать сегмент-источник в сегмент-приемник с помощью обычного программного цикла, использующего команды пересылки в памяти типа* ***mov***. Никакие вызовы READ или WRITE не нужны.

После выполнения копирования процесс может выполнить вызов UNMAP для удаления файла из адресного пространства, а затем завершиться. Выходной файл ddd будет существовать на диске, как если бы он был создан обычным способом.

Достоинство метода: исключает потребность в выполнении ввода-вывода и тем самым облегчает программирование.

Проблемы, порождаемые методом:

1. Для системы сложно узнать точную длину выходного файла (в данном примере ddd). Проще указать наибольший номер записанной страницы, но нет способа узнать, сколько байт в этой странице было записано. Предположим, что программа использует только страницу номер 0, и после выполнения все байты все еще установлены в значение 0 (их начальное значение). Быть может, файл состоит из 10 нулей. А может быть, он состоит из 100 нулей. Операционная система не может это сообщить. Все, что она может сделать, так это создать файл, длина которого равна размеру страницы.
2. Вторая проблема проявляется (потенциально), если один процесс отображает файл, а другой процесс открывает его для обычного файлового доступа. Если первый процесс изменяет страницу, то это изменение не будет отражено в файле на диске до тех пор, пока страница не будет вытеснена на диск. Поддержание согласованности данных файла для этих двух процессов требует от системы больших забот.
3. Файл может быть больше, чем сегмент, и даже больше, чем все виртуальное адресное пространство. Единственный способ ее решения состоит в реализации вызова MAP таким образом, чтобы он мог отображать не весь файл, а его часть. Такая работа, очевидно, менее удобна, чем отображение целого файла.

11.3 СОВРЕМЕННЫЕ АРХИТЕКТУРЫ ФАЙЛОВЫХ СИСТЕМ

Разработчики новых ОС стремятся дать пользователю возможностью работать сразу *с несколькими файловыми системами*. В новом понимании файловая система состоит из многих составляющих, в число которых входят и *файловые системы в традиционном понимании*.

Новая файловая система имеет многоуровневую структуру (рисунок 11.4), на верхнем уровне которой располагается *переключатель файловых систем* (в Windows, например, такой переключатель называется устанавливаемым диспетчером файловой системы – installable filesystem manager, IFS). Он обеспечивает интерфейс между запросами приложения и конкретной файловой системой, к которой обращается это приложение. Переключатель файловых систем преобразует запросы в формат, воспринимаемый следующим уровнем – *уровнем файловых систем.*

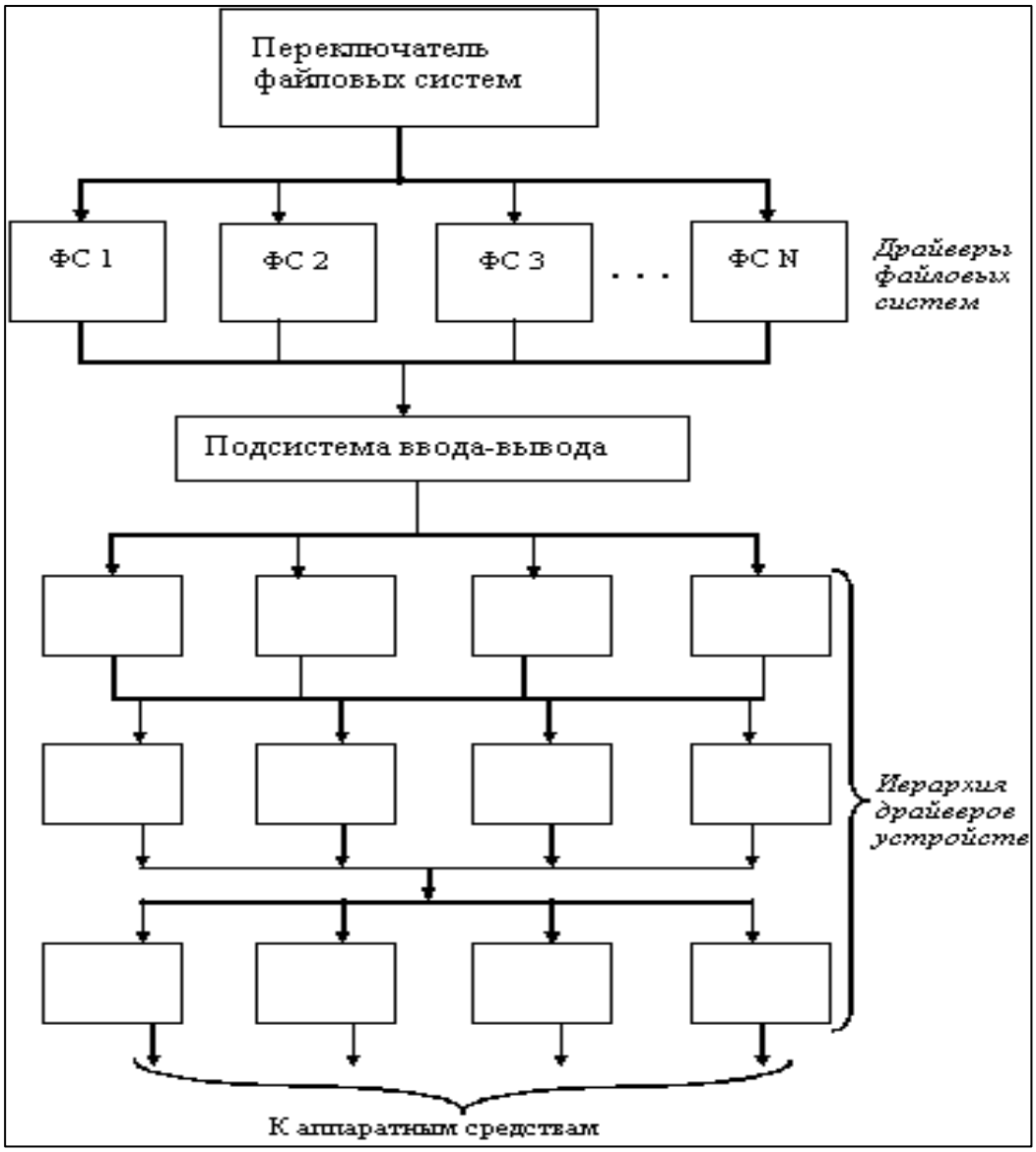


Рис. 11.4. Архитектура современной файловой системы

Каждый компонент уровня файловых систем выполнен в виде драйвера соответствующей файловой системы и поддерживает определенную организацию файловой системы.

Переключатель является единственным модулем, который может обращаться к драйверу файловой системы. Приложение не может обращаться к нему напрямую. Драйвер файловой системы может быть написан в виде реентерабельного кода, что позволяет сразу нескольким приложениям выполнять операции с файлами. *Каждый драйвер файловой системы в процессе собственной инициализации регистрируется у переключателя*, передавая ему таблицу точек входа, которые будут использоваться при последующих обращениях к файловой системе.

Для выполнения своих функций *драйверы файловых систем обращаются к подсистеме ввода-вывода*, образующей следующий слой файловой системы новой архитектуры.

Подсистема ввода вывода – это составная часть файловой системы, которая отвечает за загрузку, инициализацию и управление всеми модулями низших уровней файловой системы. *Обычно эти модули представляют собой драйверы портов, которые непосредственно занимаются работой с аппаратными средствами.*

Кроме этого *подсистема ввода-вывода обеспечивает некоторый сервис драйверам файловой системы*, что позволяет им осуществлять запросы к конкретным устройствам.

Подсистема ввода-вывода должна постоянно присутствовать в памяти и *организовывать совместную работу иерархии драйверов устройств*. В эту иерархию могут входить:

1. драйверы устройств определенного типа (драйверы жестких дисков или флэш-памяти);
2. драйверы, поддерживаемые поставщиками (такие драйверы перехватывают запросы к блочным устройствам и могут частично изменить поведение существующего драйвера этого устройства, например, зашифровать данные);
3. драйверы портов, которые управляют конкретными адаптерами.

Большое число уровней архитектуры файловой системы обеспечивает авторам драйверов устройств большую гибкость – *драйвер может получить управление на любом этапе выполнения запроса* – от вызова приложением функции, которая занимается работой с файлами, до того момента, когда работающий на самом низком уровне драйвер устройства начинает просматривать регистры контроллера.

Многоуровневый механизм работы файловой системы реализован посредством цепочек вызова. В ходе инициализации драйвер устройства может добавить себя к цепочке вызова некоторого устройства, определив при этом уровень последующего обращения.

Подсистема ввода-вывода помещает адрес целевой функции в цепочку вызова устройства, используя заданный уровень для того, чтобы должным образом упорядочить цепочку. По мере выполнения запроса, подсистема ввода-вывода последовательно вызывает все функции, ранее помещенные в цепочку вызова.

Внесенная в цепочку вызова *процедура драйвера может решить передать запрос дальше* – в измененном или в неизмененном виде – на следующий уровень, или, если это возможно, процедура *может удовлетворить запрос*, не передавая его дальше по цепочке.

ТЕМА 12

ПОНЯТИЕ ПРОЦЕССА В UNIX. ОБРАЗ, ДЕСКРИПТОР И КОНТЕКСТ ПРОЦЕССА. ПОРОЖДЕНИЕ ПРОЦЕССОВ. ПЛАНИРОВАНИЕ ПРОЦЕССОВ

12.1 УПРАВЛЕНИЕ ПРОЦЕССАМИ В UNIX

В основе UNIX лежит концепция *процесса – единицы управления и единицы потребления ресурсов*.

Процесс представляет собой программу в состоянии выполнения, причем в UNIX в рамках одного процесса не могут выполняться никакие параллельные действия.

12.2 ОБРАЗ, ДЕСКРИПТОР И КОНТЕКСТ ПРОЦЕССА

Каждый процесс работает в своем виртуальном адресном пространстве. *Образом процесса называется совокупность участков физической памяти, отображаемых на виртуальные адреса процесса,*

При *управлении процессами* операционная система использует два основных типа информационных структур:

1. *дескриптор процесса (структура **proc**);*
2. *контекст процесса (структура **user**).*

Дескриптор процесса содержит информацию о процессе, которая необходима ядру в течение всего жизненного цикла процесса, независимо от того, находится ли он в активном или пассивном состоянии, находится ли образ процесса в оперативной памяти или выгружен на диск.

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 ним структуры) *содержится информация:*

1. о состоянии процесса;
2. о расположении образа процесса в оперативной памяти и на диске;
3. о значении отдельных составляющих приоритета, а также его итоговое значение – глобальный приоритет;
4. идентификатор пользователя, создавшего процесс;
5. информация о родственных процессах;
6. информация о событиях, осуществления которых ожидает данный процесс;
7. некоторая другая информация.

Контекст процесса содержит менее оперативную, но более объемную часть информации о процессе, необходимую для возобновления выполнения процесса с прерванного места:

1. содержимое регистров процессора;
2. коды ошибок выполняемых процессором системных вызовов;
3. информацию обо всех открытых данным процессом файлов;
4. информацию о незавершенных операциях ввода-вывода (указатели на структуры file);
5. другие данные, характеризующие состояние вычислительной среды в момент прерывания.

Контекст, как и дескриптор процесса, доступен только программам ядра, то есть находится в виртуальном адресном пространстве операционной системы. Однако он хранится не в области ядра, а непосредственно примыкает к образу процесса и перемещается вместе с ним, если это необходимо, из оперативной памяти на диск.

В UNIX для процессов предусмотрены *два режима выполнения:*

1. привилегированный режим – "система";
2. обычный режим – "пользователь".

В режиме "пользователь" запрещено выполнение *действий, связанных с управлением ресурсами системы*, в частности:

1. корректировка системных таблиц;
2. управление внешними устройствами;
3. маскирование прерываний;
4. обработка прерываний.

В режиме "система" выполняются программы ядра, а в режиме "пользователь" – оболочка и прикладные программы.

Для выполнения привилегированных действий *пользовательский процесс обращается с запросом к ядру в форме системного вызова.* В результате вызова управление передается соответствующей программе ядра. С момента начала выполнения системного вызова процесс считается системным. Таким образом, один и тот же *процесс может находиться в пользовательской и системной фазах*. Эти фазы никогда не выполняются одновременно.

В ранних версиях UNIX процесс, работающий в режиме системы, не мог быть вытеснен другим процессом. Из-за этого организация ядра упрощалась, т.к. все функции ядра не были реентерабельными. Однако, при этом реактивность системы страдала - любой процесс, даже низкоприоритетный, войдя в системную фазу, мог оставаться в ней сколь угодно долго. Из-за этого свойства UNIX не мог использоваться в качестве ОС реального времени.

В более поздних версиях, и в SVR4 в том числе, организация ядра усложнилась и процесс можно вытеснить и в системной фазе, но не в произвольный момент времени, а только в определенные периоды его работы, когда процесс сам разрешает это сделать установкой специального сигнала.

В SVR4 имеется несколько процессов, которые не имеют пользовательской фазы, например, процесс pageout, организующий выталкивание страниц на диск.

12.3 ПОРОЖДЕНИЕ ПРОЦЕССОВ

Порождение процессов в системе UNIX происходит следующим образом:

1. При создании процесса строится *образ порожденного процесса, являющийся точной копией образа породившего процесса;*
2. *Сегмент данных и сегмент стека отца действительно копируются на новое место*, образуя сегменты данных и стека сына;
3. Процедурный сегмент копируется только тогда, когда он не является разделяемым. В противном случае сын становится еще одним процессом, разделяющим данный процедурный сегмент;
4. После выполнения системного вызова fork оба процесса продолжают выполнение с одной и той же точки;
5. Чтобы процесс мог опознать, является ли он отцом или сыном, системный вызов fork возвращает в качестве своего значения в породивший процесс идентификатор порожденного процесса, а в порожденный процесс NULL.

Типичное разветвление на языке С записывается так:

```
if( fork() )  {  действия отца }
else          { действия сына }
```

Идентификатор сына может быть присвоен переменной, входящей в контекст процесса-отца. Так как контекст процесса наследуется его потомками, то дети могут узнать идентификаторы своих старших братьев.

Таким образом, сумма знаний наследуется при порождении и может быть распространена между родственными процессами. Наследуются все характеристики процесса, содержащиеся в контексте.

На независимости идентификатора процесса от выполняемой процессом программы построен механизм, позволяющий процессу прийти к выполнению другой программы с помощью системного вызова exec.

Итак, в UNIX *порождение нового процесса происходит в два этапа*:

1. Создается копия процесса-родителя, то есть *дублируется дескриптор, контекст и образ процесса;*
2. У нового процесса производится замена кодового сегмента на заданный сегмент.

Вновь созданному процессу операционная система присваивает целочисленный идентификатор, *уникальный* за весь период функционирования системы.

12.4 ПЛАНИРОВАНИЕ ПРОЦЕССОВ

В системе UNIX System V Release 4 реализована *вытесняющая многозадачность, основанная на использовании приоритетов и квантования.*

Все процессы разбиты на *несколько групп, называемых* ***классами приоритетов***. Каждая *группа имеет свои характеристики планирования процессов*.

Созданный процесс наследует характеристики планирования процесса-родителя, которые включают:

1. класс приоритета;
2. величину приоритета в этом классе.

Процесс остается в данном классе до тех пор, пока не будет выполнен системный вызов, изменяющий его класс. В UNIX System V Release 4 возможно включение новых классов приоритетов при инсталляции системы.

Имеется *три приоритетных класса:*

1. класс реального времени;
2. класс системных процессов;
3. класс процессов разделения времени.

Приоритетность (привилегии) процесса тем выше, чем больше число, выражающее приоритет. В таблице 12.1 показаны диапазоны изменения приоритетов для разных классов.

Таблица 12.1 Приоритетные классы процессов

Приоритетный класс	**Выбор планировщика**	**Глобальное значение приоритета**
Реальное время (real time)	Первый . .	159 ……. 100
Системные процессы (system)	. . .	99 . 60
Процессы разделения времени (time-shared)	. . . последний	59 . . 0
Возможно добавление новых классов		

Значения приоритетов определяются для разных классов по-разному.

Процессы системного класса *используют стратегию фиксированных приоритетов.* Системный класс зарезервирован для процессов ядра. *Уровень приоритета процессу назначается ядром и никогда не изменяется.* Замечание: пользовательский процесс, перешедший в системную фазу, не переходит при этом в системный класс приоритетов.

Процессы реального времени *также используют стратегию фиксированных приоритетов, но пользователь может их изменять.* Так как при наличии готовых к выполнению процессов реального времени другие процессы не рассматриваются, то процессы реального времени надо тщательно проектировать, чтобы они не захватывали процессор на слишком долгое время.

Характеристики планирования процессов реального времени включают две величины:

1. уровень глобального приоритета;
2. квант времени.

Для каждого уровня приоритета имеется по умолчанию своя величина кванта времени. Процессу разрешается захватывать процессор на указанный квант времени, а по его истечении планировщик снимает процесс с выполнения.

По умолчанию UNIX System V Release 4 назначает новому процессу ***класс разделения времени***.

Состав класса процессов разделения времени наиболее неопределенный и часто меняющийся, в отличие от системных процессов и процессов реального времени.

Для справедливого распределения времени процессора между процессами используется *стратегия динамических приоритетов*, которая адаптируется к операционным характеристикам процесса.

Величина приоритета, назначаемого процессам разделения времени, вычисляется пропорционально значениям *двух составляющих:*

1. *пользовательской части;*
2. *системной части.*

Пользовательская часть приоритета может быть изменена:

1. суперпользователем;
2. владельцем процесса, но в последнем случае только в сторону его снижения.

Системная составляющая позволяет планировщику управлять процессами в зависимости от того, *как долго они используют процессор.*

Тем процессам, которые потребляют большие периоды времени без ухода в состояние ожидания, приоритет снижается.

Процессам, которые часто уходят в состояние ожидания после короткого периода использования процессора, приоритет повышается.

Таким образом, процессы, ведущие себя не по-джентльменски, реже выбираются на выполнение.

Но процессам с низким приоритетом даются большие кванты времени, чем процессам с высокими приоритетами.

Поэтому, хотя низкоприоритетный процесс и не работает так часто, как высокоприоритетный, но, когда он наконец выбирается на выполнение, ему отводится больше времени.

Характеристики, используемые планировщиком, для процессов разделения времени:

Ts_globpri	содержит величину глобального приоритета
Ts_quantum	определяет количество тиков системных часов, которые отводятся процессу до его вытеснения
ts_tqexp	системная часть приоритета, назначаемая процессу при истечении его кванта времени
ts_slpret	Системная составляющая приоритета, назначаемая процессу после выхода его из состояния ожидания; ожидающим процессам дается высокий приоритет, так что они быстро получают доступ к процессору после освобождения ресурса
ts_maxwaite	Максимальное число секунд, которое разрешается потреблять процессу; если квант времени истекает до кванта ts_quantum, то, следовательно, считается, что процесс ведет себя по-джентльменски, и ему назначается более высокий приоритет
ts_lwait	величина системной части приоритета, назначаемая процессу, если истекает ts_maxwait секунд

Для процессов разделения времени в дескрипторе процесса proc имеется указатель на структуру, специфическую для данного класса процесса. Эта *структура состоит из полей, используемых для вычисления глобального приоритета*:

ts_timeleft	число тиков, остающихся в кванте процесса
ts_cpupri	системная часть приоритета процесса
ts_uprilim, ts_upri	верхний предел и текущее значение пользовательской части приоритета. Эти две переменные могут модифицироваться пользователем
ts_nice	используется для обратной совместимости с системным вызовом nice. Она содержит текущее значение величины nice, которая влияет на результирующую величину приоритета. Чем выше эта величина, тем меньше приоритет

В версии SVR4 нет поддержки многонитевой (multithreading) организации процессов на уровне ядра, хотя и есть два системных вызова для организации нитей в пользовательском режиме.

Во многих коммерческих реализациях UNIX, базирующихся на кодах SVR4, в ядро включена поддержка нитей за счет собственной модификации исходных текстов SVR4.

ТЕМА 13

УПРАВЛЕНИЕ ПАМЯТЬЮ В ОС UNIX. СТРУКТУРА ФИЗИЧЕСКОЙ ПАМЯТИ. СВОПИНГ

13.1 УПРАВЛЕНИЕ ПАМЯТЬЮ В ОС UNIX

В UNIX System V Release 4 реализована ***сегментно-страничная модель памяти*** в ее традиционном виде.

Используются механизмы:

1. управления страницами;
2. свопинга, когда на диск выталкиваются все страницы какого-либо процесса. Свопинг применяется в "предаварийных" ситуациях, когда размер свободной оперативной памяти уменьшается до некоторого заданного порога, так что работа всей системы очень затрудняется.

На рисунке 13.1 показаны *основные структуры, описывающие виртуальное адресное пространство отдельного процесса*.

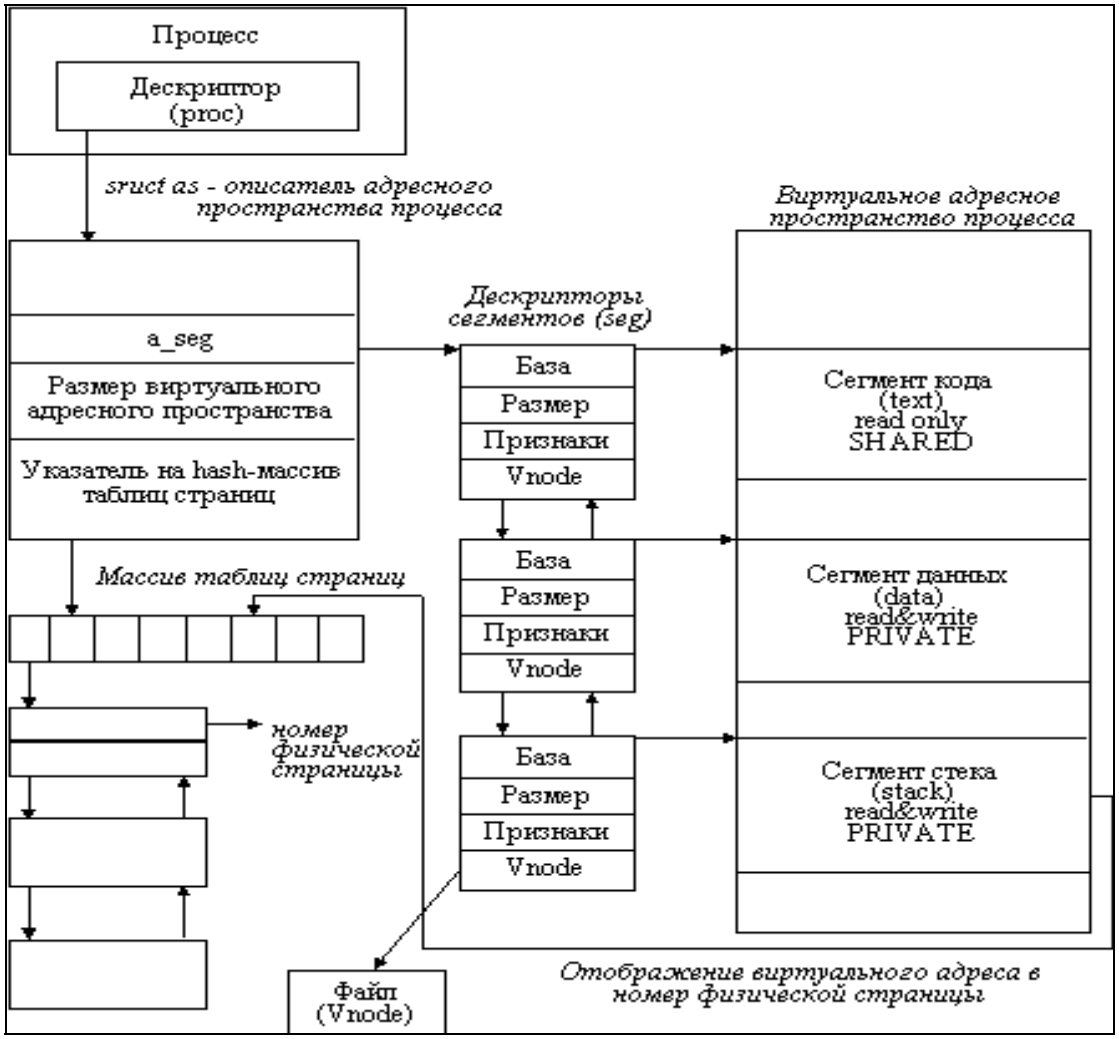


Рис. 13.1 Сегментно-страничная модель виртуальной памяти UNIX

В ***дескрипторе процесса*** **proc** содержится указатель на структуру **as**.

Структура **as** ***описывает все виртуальные сегменты***, которыми обладает данный процесс.

Элемент **a_seg** в структуре as ***указывает на первый дескриптор сегмента процесса***.

Каждый ***дескриптор сегмента (структура*** **seg*****) описывает один виртуальный сегмент процесса***. Дескрипторы сегментов процесса связаны в двунаправленный список. Дескриптор сегмента содержит:

1. базовый адрес начала сегмента в виртуальном адресном пространстве процесса;
2. размер сегмента;
3. указатели на операции, которые допускаются над этим сегментом (дублирование, освобождение, отображение и т.д.).

Имеются следующие *типы виртуальных сегментов:*

1. *Текст (text) - содержит коды команд исполняемого модуля процесса*. Он обычно обозначается "только для чтения", так чтобы ни сам процесс, ни другие процессы не могли изменить его кодовую часть. *Текстовый сегмент может разделяться многими процессами*, например, всеми пользователями, работающими с одним редактором.
2. *Данные (data) - содержит данные, используемые и модифицируемые процессом во время выполнения.* К сегменту данных обычно разрешается иметь доступ для чтения и записи. В отличие от текстового сегмента, *сегмент данных никогда не разделяется другими процессами.*
3. *Стек (stack) - содержит стек процесса*. Он помечается доступным для чтения и записи и, подобно сегменту данных, *не может разделяться другими процессами.*

Есть еще два типа сегментов:

1. *Разделяемая память (shared memory) - область памяти, доступная для чтения и записи нескольким процессам*
2. *Отображенный файл (mapped file) - сегменты отображенного файла используются для того, чтобы отобразить части файлов в адресное пространство процесса*, и использовать стандартные механизмы ОС управления виртуальной памятью для ускорения доступа к файлам.

Поле **s_data** дескриптора сегмента указывает на структуру данных **segvn_dat**, в которой *содержится специфическая для сегмента информация*:

1. *type:* признак, является ли сегмент разделяемым или личным;
2. *vp и offset:* указатель на vnode файла и смещение в этом файле, которые задают адрес, начиная с которого расположены на диске данные этого сегмента;
3. *amp:* указатель на карту анонимных страниц сегмента.

Каждый сегмент имеет связь с дисковым пространством, на котором хранятся данные, отображаемые в данный сегмент виртуального адресного пространства.

Это может быть *файл или часть файла* на диске, или же это может быть о*бласть свопинга*, которая файлом не является.

Сегмент кода или сегмент инициализированных данных обычно связан с файлом, в котором хранится исполняемая программа. Под связью с файлом понимается отображение виртуального сегмента и его страниц на определенную область диска, из которой загружаются данные виртуальных страниц сегмента при их перемещении в оперативную память, а также, куда помещаются данные при вытеснении виртуальных страниц на диск.

Виртуальные страницы, которые были изначально взяты из определенного файла, который описывается на уровне ядра структурой **vnode**, называются *vnode-страницами*.

Страницы, которые появились только при развертывании процесса, а это обычно страницы стека или неинициализированных сегментов данных, *называются анонимными страницами*.

Однако анонимные страницы также имеют связь с файлом, в который они выталкиваются при их вытеснении из физической памяти (так называемое свопинг-устройство). На свопинг-устройство также указывает vnode, поэтому в этом качестве может выступать любой файл, а перемещение страниц из свопинг-устройства в память осуществляется теми же функциями, что используются для vnode-страниц.

Отображение виртуальных страниц сегмента на физические задается с помощью таблицы HAT (Hardware Address Translation), указатель на которую имеется в структуре адресного пространства процесса **as**.

Структура таблицы HAT зависит от аппаратной платформы, но в любом случае *с ее помощью можно найти таблицу или таблицы страниц, содержащих дескрипторы страниц (структуры типа* **pte***)*.

Дескриптор страницы содержит:

1. признак наличия данной виртуальной страницы в физической памяти;
2. номер соответствующей физической страницы;
3. признак "модификация";
4. признак "была ссылка" и т.п.

Признаки 3,4 и т.п. помогают операционной системе планировать процесс вытеснения виртуальных страниц на диск.

В UNIX System V Release 4 используется *алгоритм перемещения виртуальных страниц процесса в физическую память по запросу*.

Обычно *при запуске процесса в физическую память помещается только небольшая часть страниц, необходимая для старта процесса*, а остальные страницы загружаются при страничных сбоях. Очевидно, что начальный период работы любого процесса порождает повышенную

нагрузку на систему. Если при поиске виртуального адреса в соответствующем дескрипторе обнаруживается признак отсутствия этой страницы в физической памяти, то происходит страничное прерывание, и ядро перемещает эту страницу с диска в физическую память.

Для поиска страницы на диске используется либо:

1. информация из структуры s_data сегмента;
2. vnode и offset, если страница типа vnode;
3. информация о расположении анонимной страницы в области свопинга с помощью информации о ее расположении там по карте amp.

Если в физической памяти недостаточно места для размещения затребованной процессом страницы, то ОС выгружает некоторые страницы на диск. Этот процесс осуществляется специальным процессом ядра, "выталкивателем страниц", имеющем в UNIX System V Release 4 имя **pageout.** Для принятия решения о том, какую виртуальную страницу нужно переместить на диск, процессу **pageout** нужно иметь информацию о текущем состоянии физической памяти.

13.2 СТРУКТУРА ФИЗИЧЕСКОЙ ПАМЯТИ

На рисунке 13.2 показан пример распределения физической памяти.

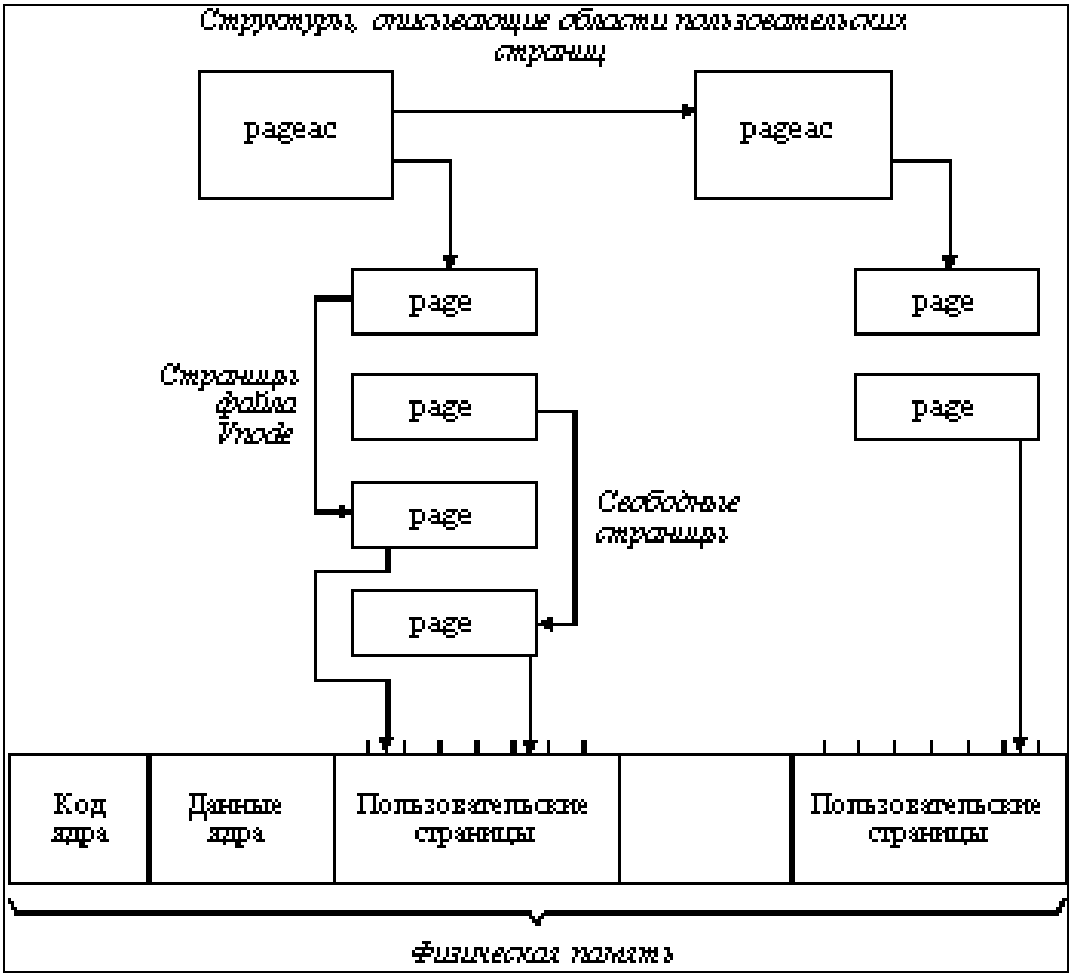


Рис. 13.2 Структуры, описывающие физическую память

Начиная с самых нижних адресов, сегменты располагаются в нижеприведенной последовательности:

1. *текстовый сегмент ядра*;
2. *сегмент данных ядра*;
3. *динамический сегмент данных ядра*, в котором отводится место для структур ядра, например, для дескрипторов процессов.

В оставшейся физической памяти могут располагаться в общем случае несколько *областей для хранения страниц пользовательских процессов.* Эти области описываются таблицей **pageac_table[...]**, каждый элемент которой содержит:

1. номера начальной и конечной страницы области;
2. указатели на дескрипторы первой и последней страниц, размещенных в этой области.

Для каждой физической страницы имеется *дескриптор страницы - структура* **page***,* в котором содержится информация о том:

1. свободна или занята страница;
2. загружена ли в нее в данный момент виртуальная страница;
3. модифицировано ли ее содержимое;
4. сколько процессов хотят сохранить эту страницу в памяти;
5. другая информация.

В каждый момент времени *дескриптор физической страницы может состоять в одном из следующих списков:*

1. *Список хешированных виртуальных дескрипторов файла.* Каждый отображаемый или выполняемый файл описывается виртуальным дескриптором (vnode). Страницы, относящиеся к отдельному vnode, связываются в список. При этом используется хеширование для быстрого поиска в случае, когда произойдет страничное прерывание.
2. *Список свободных страниц.*
3. *Список страниц, образующих кэш страниц.* Этот список подобен списку свободных станиц, но данные в этих страницах остаются действительными.

Страницы обычно помещаются в список кэша процессом pageout. Если происходит поиск какой-либо страницы, и эта страница находится в списке страниц кэша, то она повторно используется. Например, страница может быть изъята процессом pageout и помещена в кэш-список. Процесс может затем повторно использовать эту страницу перед тем, как операционная система назначит ее другому процессу. Таким образом, *кэшированная страница может быть повторно назначена процессу без перемещения данных с диска.*

Для вытеснения виртуальных страниц используется несколько констант, описывающих размер свободной памяти. Эти константы используются как пороговые значения для действий по освобождению физической памяти (рисунок 13.3).

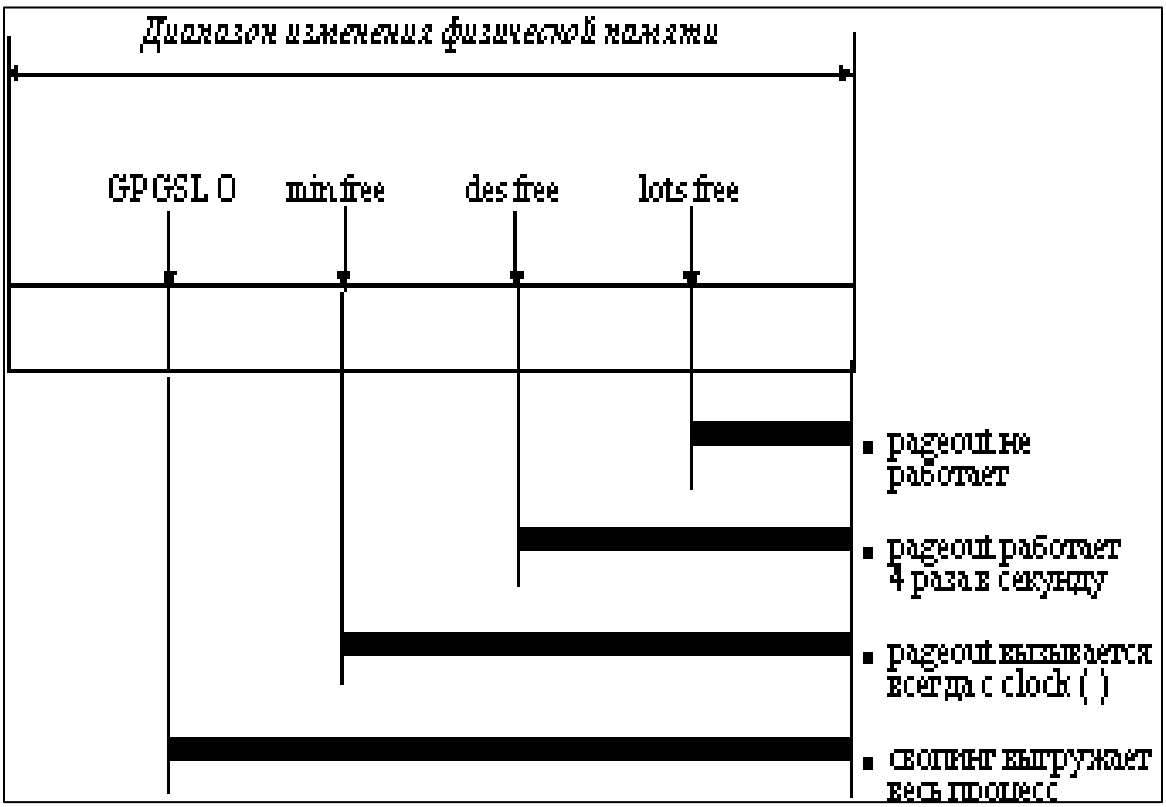


Рис. 13.3 Пороговые значения для действий по освобождению физической памяти

Если свободная память в системе превышает порог lotsfree, то процесс pageout не вызывается вовсе.

Если размер свободной физической памяти находится в пределах от desfree до lotsfree, то pagefree вызывается 4 раза в секунду.

Если ее размер становится меньше порога desfree, то pagefree вызывается при каждом цикле работы функции clock().

Если же свободная память становится меньше порога GPGSLO, то в действие вступает *процесс свопинга*, который в UNIX System V Release 4 называется **shed.**

Этот процесс выбирает определенный процесс и выгружает все его страницы на диск, освобождая тем самым сразу значительное место в памяти.

Таким образом, UNIX System V Release 4 *использует механизм свопинга процессов* для освобождения физической памяти для других процессов. Процесс, выгруженный на диск, исключается из претендентов на выполнение.

Через некоторое время процесс **shed** вызывается снова. Если количество свободной памяти превысило GPGSLO, то процесс загружается с диска в память и включается в очередь готовых к выполнению процессов.

Процесс вытеснения страниц pageout использует при поиске страниц для вытеснения *алгоритм NRU* (Not Recently Used), выбирающий для вытеснения не используемые в последнее время страницы. Этот алгоритм использует *признаки модификации и доступа страниц.*

Процесс **pageout** периодически очищает эти признаки у тех страниц, которые не свободны. Если при следующем вызове процесс pageout видит, что эти признаки равны нулю, то значит доступа к этим страницам с момента предыдущего вызова процесса pageout не было, поэтому эти страницы вытесняются на диск.

Процесс pageout циклически проверяет все страницы физической памяти, поэтому он называется *часовым алгоритмом*, что отражает просмотр страниц как бы по часовой стрелке.

Было замечено, что обход всех страниц при их большом количестве занимает слишком много времени, поэтому в UNIX System V Release 4 применяется *модифицированный часовой алгоритм*.

Механизм хорошо иллюстрируется часами с двумя стрелками, которые движутся синхронно, то есть угол между ними сохраняется постоянным. Первая стрелка указывает на *виртуальные страницы, признаки которых обнуляются,* а вторая - на *страницы, признаки которых проверяются* и, в случае их равенства нулю, страница вытесняется их физической памяти.

При каждом вызове процесс **pageout** делает лишь часть полного оборота, поэтому при небольшом зазоре между стрелками в памяти остаются только страницы, к которым идет интенсивное обращение.

ТЕМА 14

СИСТЕМА ВВОДА-ВЫВОДА В ОС UNIX. ПОДСИСТЕМА БУФЕРИЗАЦИИ. ДРАЙВЕРЫ.

Основу системы ввода-вывода ОС UNIX составляют:

1. драйверы внешних устройств;
2. средства буферизации данных.

ОС UNIX использует *два различных интерфейса с внешними устройствами*:

1. байт-ориентированный;
2. блок-ориентированный.

14.1 ПОДСИСТЕМА БУФЕРИЗАЦИИ

Любой запрос на ввод-вывод к блок-ориентированному устройству преобразуется в запрос к *подсистеме буферизации*.

Подсистема буферизации представляет собой:

1. буферный пул;
2. комплекс программ управления этим пулом.

Буферный пул состоит из *буферов, находящихся в области ядра*. Размер отдельного буфера равен размеру блока данных на диске.

С каждым буфером связана специальная структура - *заголовок буфера*.

Заголовок буфера содержит следующую информацию:

1. *Данные о состоянии буфера:*
   1.1. занят/свободен;
   1.2. чтение/запись;
   1.3. признак отложенной записи;
   1.4. ошибка ввода-вывода.
2. *Данные об устройстве - источнике информации, находящейся в этом буфере:*
   2.1. тип устройства;
   2.2. номер устройства;
   2.3. номер блока на устройстве;
3. *Адрес буфера.*
4. *Ссылка на следующий буфер в очереди свободных буферов, назначенных для ввода-вывода какому-либо устройству.*

Упрощенный *алгоритм выполнения запроса к подсистеме буферизации* приведен на рисунке 14.1. *Данный алгоритм реализуется набором функций*, наиболее важные из которых приведены ниже.

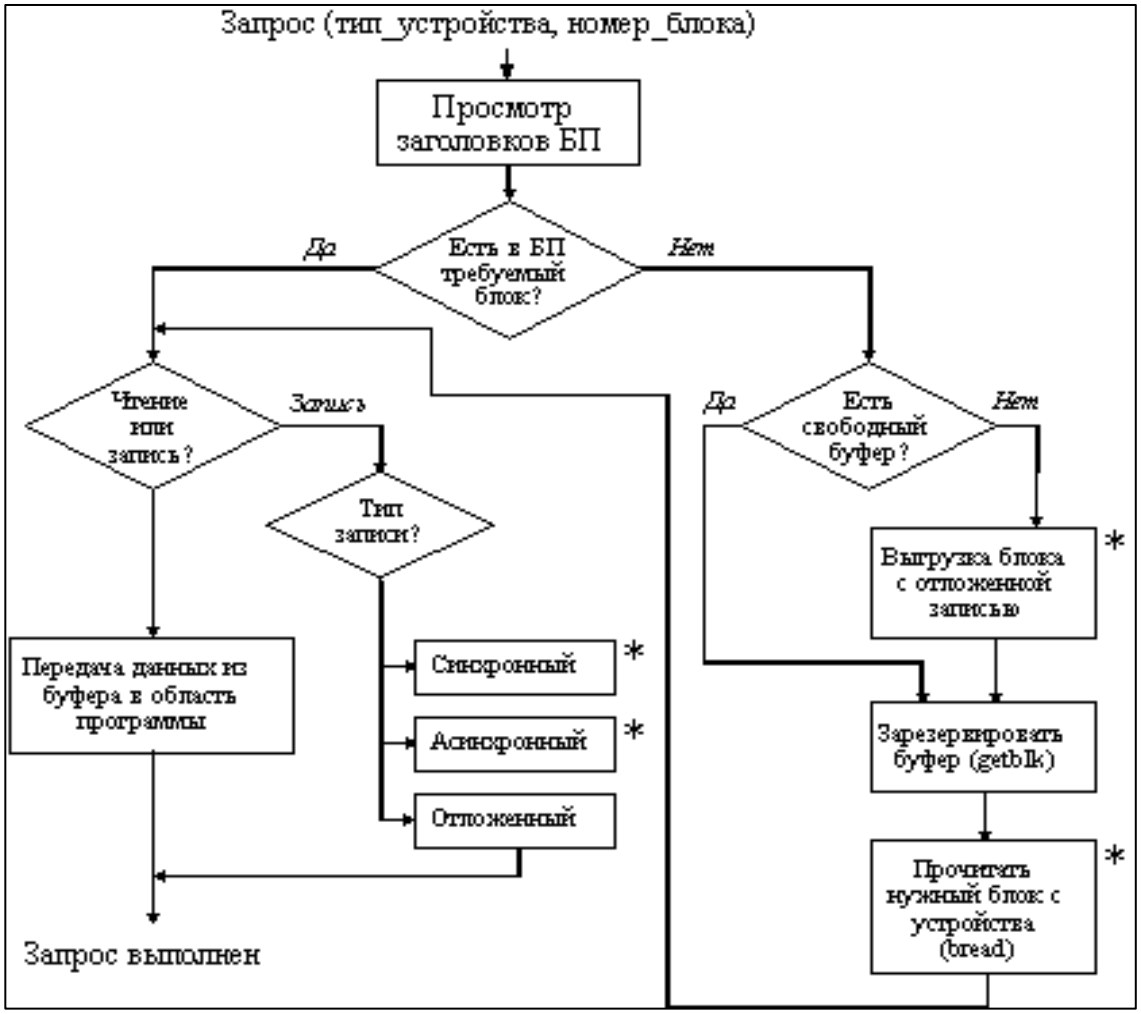


Рис. 14.1 Упрощенная схема выполнения запросов подсистемой буферизации

Функция bwrite - синхронная запись. В результате выполнения данной функции:

1. немедленно инициируется физический обмен с внешним устройством;
2. *процесс, выдавший запрос, ожидает результат выполнения операции ввода-вывода;*
3. в данном случае в процессе может быть предусмотрена собственная реакция на ошибочную ситуацию.

Такой тип записи используется тогда, когда необходима гарантия правильного завершения операции ввода-вывода.

Функция bawrite - асинхронная запись. При таком типе записи:

1. немедленно инициируется физический обмен с устройством;
2. *завершения операции ввода-вывода процесс не дожидается;*
3. возможные ошибки ввода-вывода не могут быть переданы в процесс, выдавший запрос.

Такая операция записи целесообразна при поточной обработке файлов, когда ожидание завершения операции ввода-вывода не обязательно, но есть уверенность в повторении этой операции.

Функция bdwrite - отложенная запись. При этом:

1. передача данных из системного буфера не производится;
2. в заголовке буфера делается отметка о том, что буфер заполнен и может быть выгружен, если потребуется освободить буфер.

Функции getblk - получить блок

1. ищет в буферном пуле буфер, содержащий указанный блок данных;
2. если такого блока в буферном пуле нет, то осуществляется поиск любого свободного буфера, при этом возможна выгрузка на диск буфера, содержащего в заголовке признак отложенной записи.

Функции bread - получить блок.

1. ищет в буферном пуле буфер, содержащий указанный блок данных;
2. при отсутствии заданного блока в буферном пуле организуется его загрузка в какой-нибудь свободный буфер. Если свободных буферов нет, то производится выгрузка буфера с отложенной записью.

Функция getblk используется тогда, *когда содержимое зарезервированного блока не существенно*, например, при записи на устройство данных, объем которых равен одному блоку.

Таким образом, за счет отложенной записи в системном буферном пуле задерживается некоторое число блоков данных. *При возникновении запроса к внешней памяти просматривается содержимое буферного пула.*

При этом вероятность обнаружения данных в системном пуле достаточно велика. Это обусловлено объективными свойствами пространственной и временной локальности данных. В соответствии с описанным алгоритмом буферизации, в системном буферном пуле оседает наиболее часто используемая информация. Таким образом, *система буферизации выполняет роль кэш-памяти по отношению к диску.*

Достоинство метода: кэширование диска уменьшает среднее время доступа к данным на диске.

Недостаток метода: снижается надежность файловой системы, так как в случае внезапной потери питания или отказа диска может произойти потеря блоков, содержащихся в системном буфере.

Недостаток частично *компенсируется регулярной* (каждую секунду) *принудительной записью всех блоков из системной области на диск.*

Вышеописанный механизм – это механизм старого буферного кэша, использовавшегося в предыдущих версиях UNIX System V в качестве основного дискового кэша.

В UNIX System V Release 4 используется *новый механизм, основанный на отображении файлов в физическую память.* Однако старый механизм кэширования также сохранен, так как *новый кэш используется только для блоков данных файлов*, но *непригоден для кэширования административной информации диска*, такой как inode, каталог и т.д.

Новый буферный кэш организован следующим образом. Расположение данных в файле характеризуется их *смещением от начала файла.* Так как *ядро ссылается на любой файл с помощью структуры* **vnode**, то расположение данных в файле определяется парой vnode/offset.

Доступ к файлу по адресу vnode/offset достигается с помощью сегмента виртуальной памяти типа **segmap**, который подобен сегменту segvn, используемому системой виртуальной памяти.

Метод доступа к файлам, основанный на сегментах **segmap***, называется новым буферным кэшем.* Этот способ кэширования *использует модель страничного доступа к памяти* для ссылок на блоки файлов. Размер страницы, используемой новым буферным кэшем, машинно-зависим.

Для отображения блоков файлов используется адресное пространство ядра, которое также как и пользовательское виртуальное пространство описывается структурой **as.**

Одним из сегментов в адресном пространстве ядра является сегмент **segkmap**, который используется для страничного доступа к области файлов.

14.2 ДРАЙВЕРЫ

Драйвер *- это совокупность программ (секций), предназначенная для управления передачей данных между внешним устройством и оперативной памятью.* Связь ядра системы с драйверами показана на рисунке 14.2.

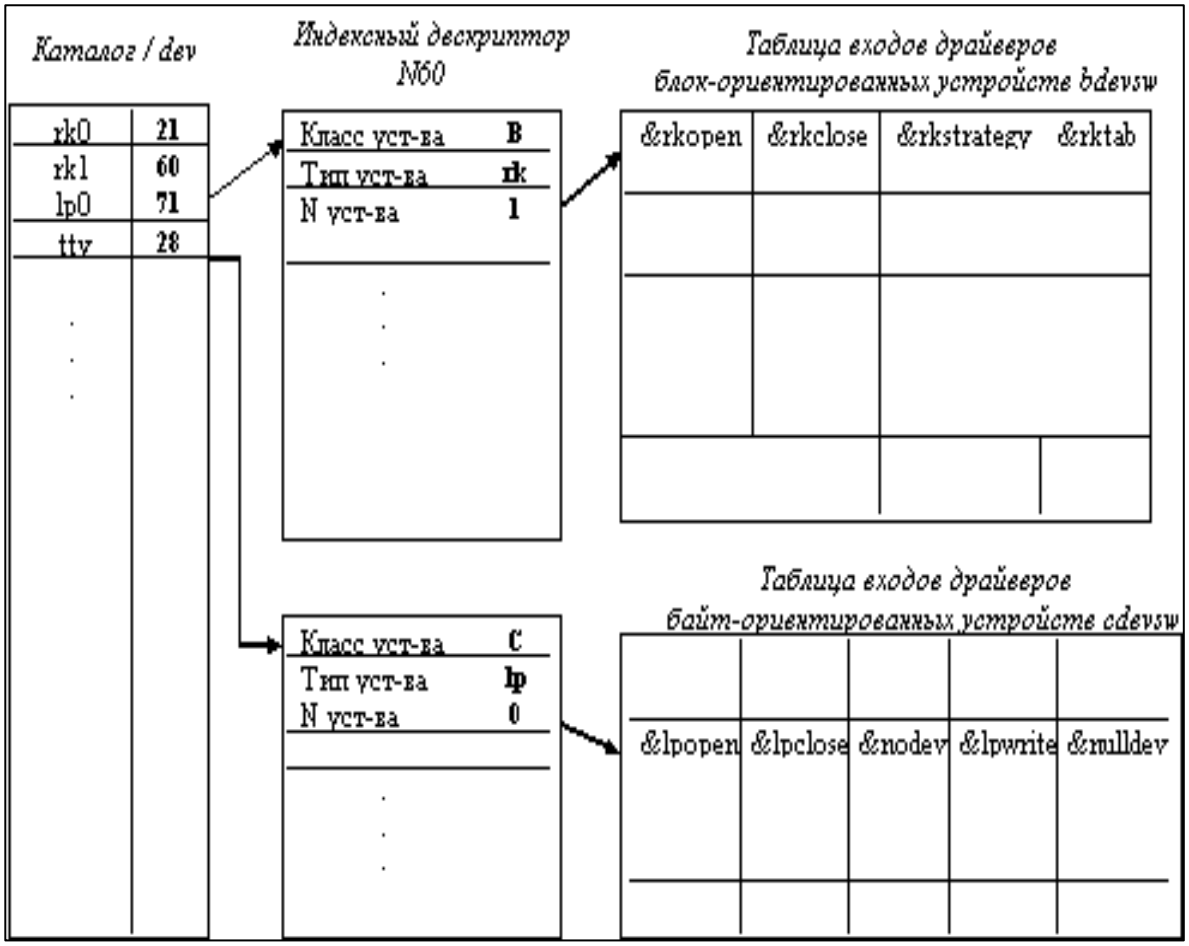


Рис. 14.2 Организация связи ядра с драйверами

Эта *связь обеспечивается с помощью двух системных таблиц:*

1. **bdevsw** - таблица блок-ориентированных устройств;
2. **cdevsw** - таблица байт-ориентированных устройств.

Для связи используется следующая *информация из индексных дескрипторов специальных файлов*:

1. класс устройства (байт-ориентированное или блок-ориентированное);
2. тип устройства (жесткий диск, гибкий диск, устройство печати, дисплей, канал связи и т.д.);
3. номер устройства.

Класс устройства определяет выбор таблицы bdevsw или cdevsw. Эти *таблицы содержат адреса программных секций драйверов, причем одна строка таблицы соответствует одному драйверу*.

Тип устройства определяет выбор драйвера. Типы устройств пронумерованы, т.е. *тип определяет номер строки выбранной таблицы*.

Номер устройства передается драйверу в качестве параметра, так как в ОС UNIX драйверы спроектированы в расчете на обслуживание нескольких устройств одного типа.

Такая организация логической связи между ядром и драйверами позволяет легко настраивать систему на новую конфигурацию внешних устройств путем модификации таблиц bdevsw и cdevsw.

Драйвер байт-ориентированного устройства в общем случае состоит из:

1. секции открытия файлов;
2. секции чтения файлов;
3. секции записи файлов;
4. секции управления режимом работы устройства.

В зависимости от типа устройства некоторые секции могут отсутствовать. Это определенным образом отражено в таблице cdevsw.

Секции записи и чтения обычно используются совместно с модулями обработки прерываний ввода-вывода от соответствующих устройств.

На рисунке 14.3 показано *взаимодействие секции записи драйвера байт-ориентированного устройства с модулем обработки прерываний.*

Секция записи осуществляет передачу байтов из рабочей области программы, выдавшей запрос на обмен, в системный буфер, организованный в виде очереди байтов. Передача байтов идет до тех пор, пока системный буфер не заполнится до некоторого, заранее определенного в драйвере, уровня. В результате секция записи драйвера приостанавливается, выполнив системную функцию **sleep** (аналог функций типа **wait**).

Модуль обработки прерываний работает асинхронно секции записи. Он вызывается в моменты времени, определяемые готовностью устройства принять следующий байт. Если при очередном прерывании оказывается, что очередь байтов уменьшилась до определенной нижней границы, то модуль обработки прерываний активизирует секцию записи драйвера путем обращения к системной функции **wakeup.**

Аналогично организована работа при чтении данных с устройства.

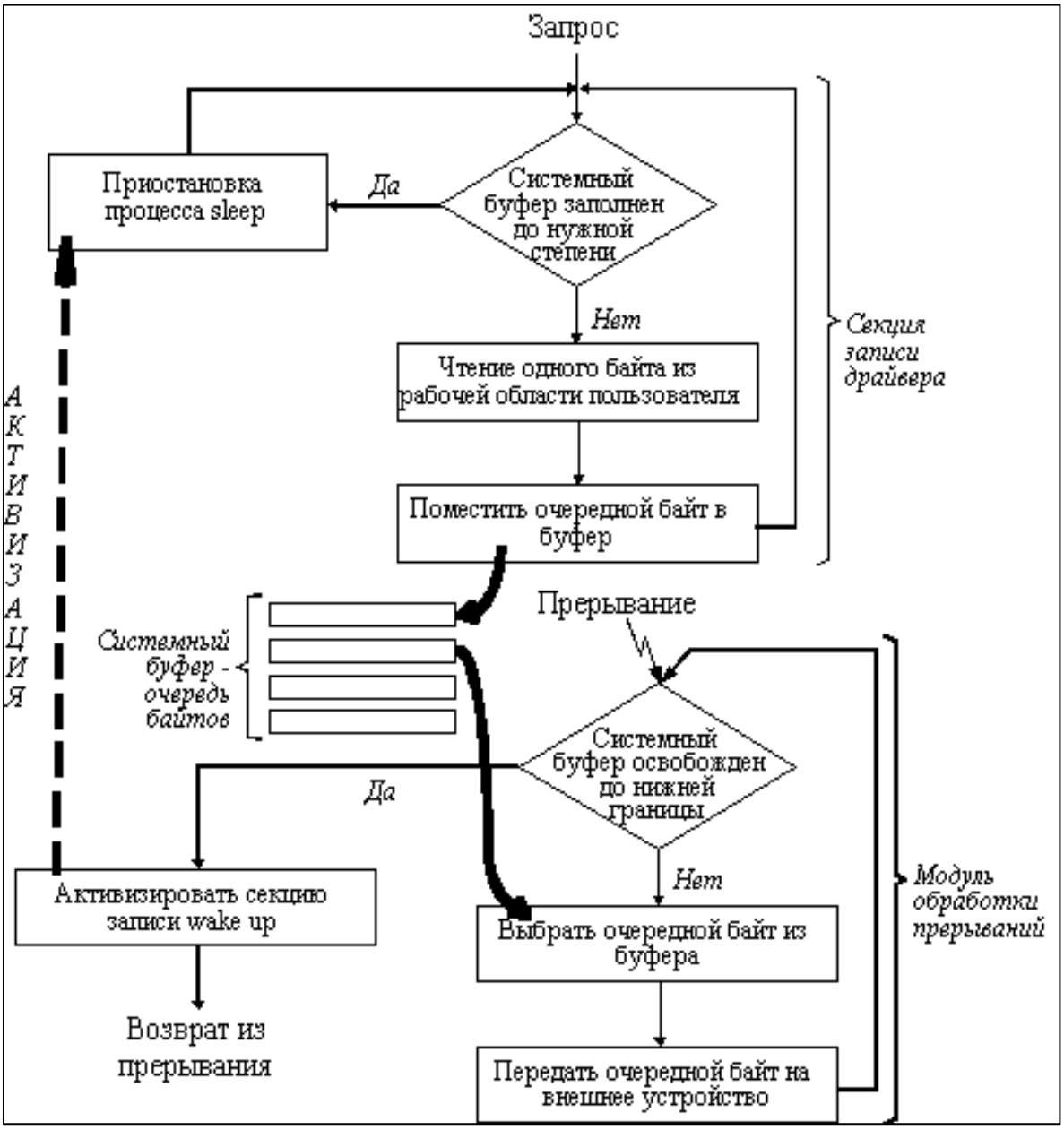


Рис. 14.3 Взаимодействие секции записи драйвера с модулем обработки прерывания

Драйвер блок-ориентированного устройства состоит из секций:

1. открытия файлов;
2. закрытия файлов;
3. секции стратегии.

Кроме адресов этих секций, в таблице bdevsw указаны адреса так называемых таблиц устройств (rktab). Эти таблицы содержат *информацию о состоянии устройства*:

1. занято или свободно;
2. указатели на буфера, для которых активизированы операции обмена с данным устройством;
3. указатели на цепочку буферов, в которых находятся блоки данных, предназначенные для обмена с данным устройством.

На рисунке 14.4 приведена упрощенная схема драйвера жесткого диска.

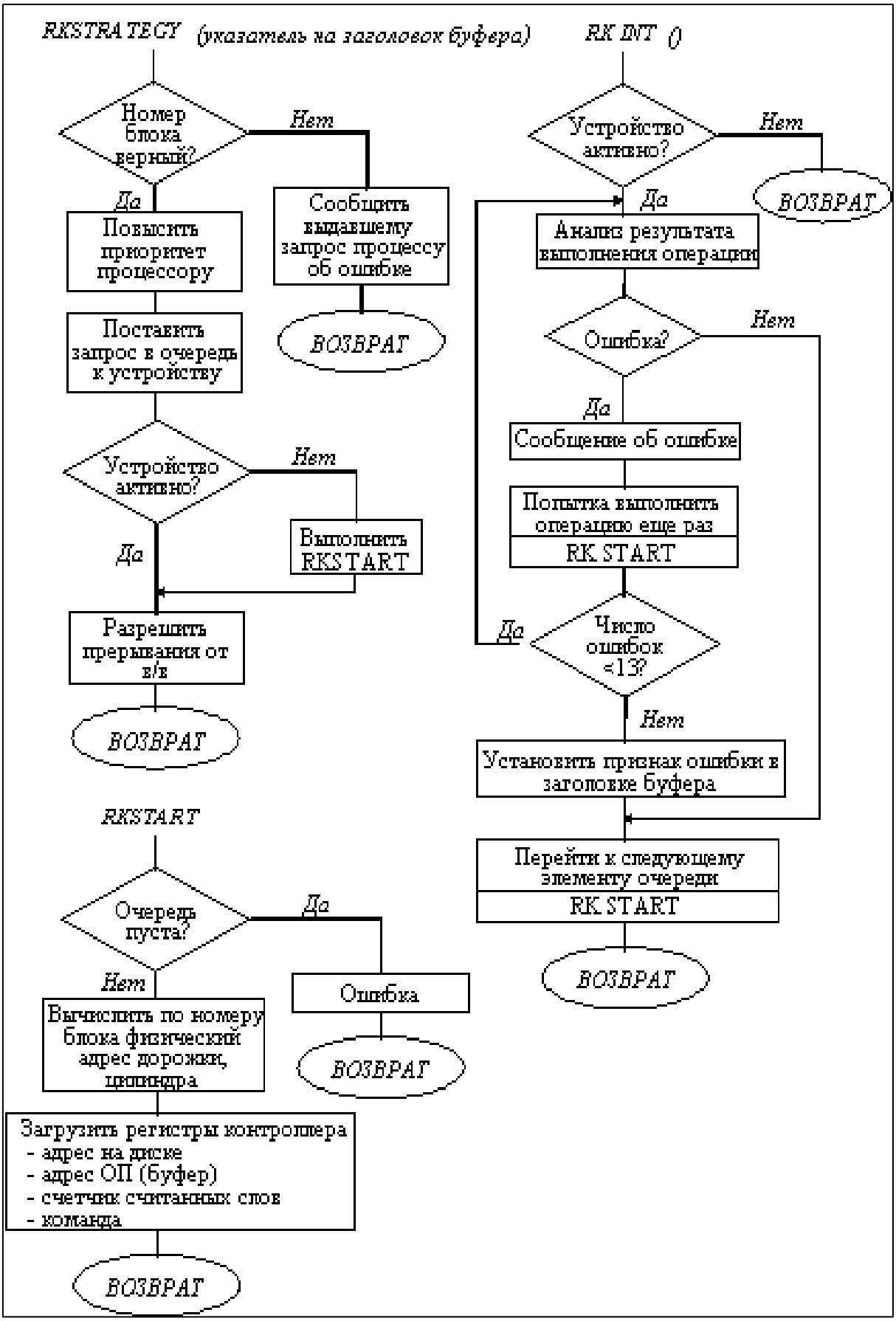


Рис 14.4 Структурная схема драйвера диска типа RK

Секция стратегии – **rkstrategy**:

1. выполняет постановку запроса на ввод-вывод в очередь к устройству путем присоединения указанного буфера к цепочке буферов, уже предназначенных для обмена с данным устройством;
2. в случае необходимости секция стратегии запускает устройство (программа **rkstart**) для выполнения чтения или записи блока с устройства;
3. вся информация о требуемой операции может быть получена из заголовка буфера, указатель на который передается секции стратегии в качестве аргумента.

После запуска устройства управление возвращается процессу, выдавшему запрос к драйверу.

Об окончании ввода-вывода каждого блока устройство оповещает операционную систему сигналом прерывания.

Первое слово вектора прерываний данного устройства содержит адрес секции драйвера - модуля обработки прерываний **rkintr.** Модуль обработки прерываний проводит анализ правильности выполнения ввода-вывода. Если зафиксирована ошибка, то несколько раз повторяется запуск этой же операции, после чего драйвер переходит к вводу-выводу следующего блока данных из очереди к устройству.

ТЕМА 15

ФАЙЛОВЫЕ СИСТЕМЫ UNIX. ТРАДИЦИОННАЯ ФАЙЛОВАЯ СИСТЕМА S5. ВИРТУАЛЬНАЯ ФАЙЛОВАЯ СИСТЕМА VFS.

15.1 ФАЙЛОВЫЕ СИСТЕМЫ UNIX SYSTEM V RELEASE 4

В UNIX System V Release 4 реализован *механизм виртуальной файловой системы VFS (Virtual File System)*, *который позволяет ядру системы одновременно поддерживать несколько различных типов файловых систем.* Механизм VFS поддерживает для ядра некоторое абстрактное представление о файловой системе, скрывая от него конкретные особенности каждой файловой системы.

Типы файловых систем, поддерживаемых в UNIX System V Release 4:

1. **s5** - традиционная файловая система UNIX System V, поддерживаемая в ранних версиях UNIX System V от AT&T;
2. **ufs** - файловая система, используемая по умолчанию в UNIX System V Release 4, которая ведет происхождение от файловой системы SunOS, которая в свою очередь, происходит от файловой системы Berkeley Fast File System (FFS);
3. **nfs** - адаптация известной файловой системы NFS фирмы Sun Microsystems, которая позволяет разделять файлы и каталоги в гетерогенных сетях;
4. **rfs** - файловая система Remote File Sharing из UNIX System V Release 3. По функциональным возможностям близка к NFS, но требует на каждом компьютере установки UNIX System V Release 3 или более поздних версий этой ОС;
5. **veritas** - отказоустойчивая файловая система с транзакционным механизмом операций;
6. **specfs** - новый тип файловой системы обеспечивает единый интерфейс ко всем специальным файлам, описываемым в каталоге /dev;
7. **fifofs** - эта новая файловая система использует механизм VFS для реализации файлов FIFO, также известных как конвейеры (pipes), в среде STREAMS;
8. **bfs** - загрузочная файловая система. Предназначена для быстрой и простой загрузки и поэтому представляет собой очень простую плоскую файловую систему, состоящую из одного каталога;
9. **/proc** - файловая система этого типа обеспечивает доступ к образу адресного пространства каждого активного процесса системы, обычно используется для отладки и трассировки;
10. **/dev/fd** - этот тип файловой системы обеспечивает удобный метод ссылок на дескрипторы открытых файлов.

Не во всех коммерческих реализациях поддерживаются все эти файловые системы, отдельные производители могут предоставлять только некоторые из них.

15.2 ТРАДИЦИОННАЯ ФАЙЛОВАЯ СИСТЕМА S5

15.2.1 Типы файлов файловой системы S5

Файловая система UNIX S5 поддерживает *логическую организацию файла в виде последовательности байтов.* По функциональному назначению различаются:

1. обычные файлы;
2. каталоги;
3. специальные файлы.

Обычные файлы содержат ту информацию, которую заносит в них пользователь или которая образуется в результате работы системных и пользовательских программ, то есть *ОС не накладывает никаких ограничений на структуру и характер информации, хранимой в обычных файлах.*

Каталог - файл, содержащий служебную информацию файловой системы о группе файлов, входящих в данный каталог. В каталог могут входить обычные, специальные файлы и каталоги более низкого уровня.

Специальный файл - фиктивный файл, ассоциируемый с каким-либо устройством ввода-вывода. Специальный файл используется для унификации механизма доступа к файлам и внешним устройствам.

15.2.2 Структура файловой системы S5

Файловая система S5 *имеет иерархическую структуру*, в которой уровни создаются за счет каталогов, содержащих информацию о файлах более низкого уровня.

Каталог самого верхнего уровня называется корневым и имеет имя **root.** Иерархическая структура удобна для многопользовательской работы: *каждый пользователь локализуется в своем каталоге или поддереве каталогов*, и вместе с тем все файлы в системе логически связаны. *Корневой каталог файловой системы всегда располагается на системном устройстве* (диск, имеющий такой признак). Это не означает, что все остальные файлы могут содержаться только на этом устройстве.

Для связи иерархий файлов, расположенных на разных носителях, применяется монтирование файловой системы, выполняемое системным вызовом **mount**.

Пусть имеются две файловые системы, расположенные на разных дисках (рисунок 15.1). Операция монтирования заключается в следующем:

1. в корневой файловой системе выбирается некоторый существующий каталог, содержащий один пустой файл (в данном примере каталог **man**, содержащий файл **dummy**).
2. После выполнения монтирования выбранный *каталог* **man** *становится корневым каталогом второй файловой системы*. Через этот каталог смонтированная файловая система подсоединяется как поддерево к общему дереву (рисунок 15.2). При этом нет логической разницы между основной и монтированными файловыми системами.

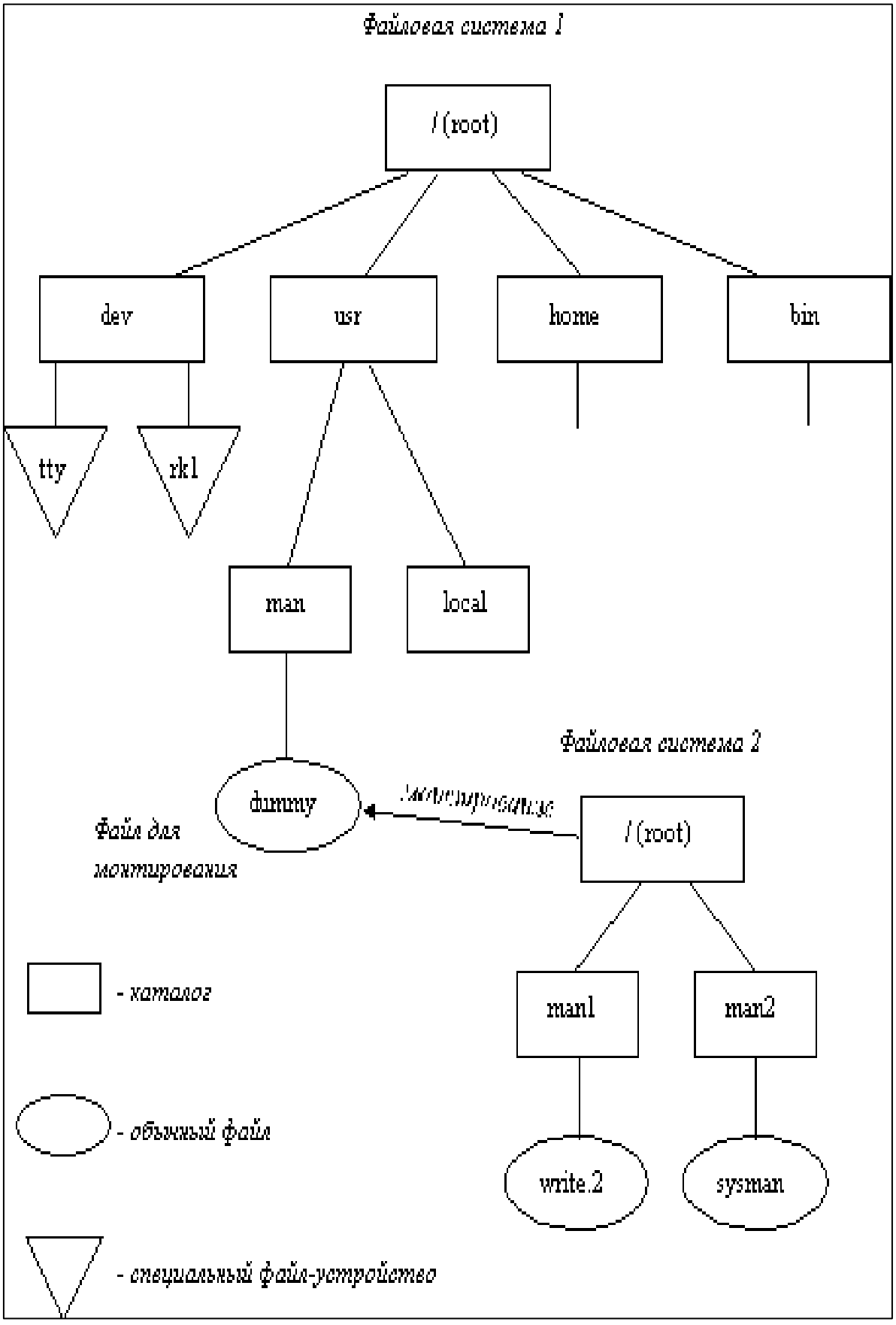


Рис. 15.1 Традиционная файловая система S5

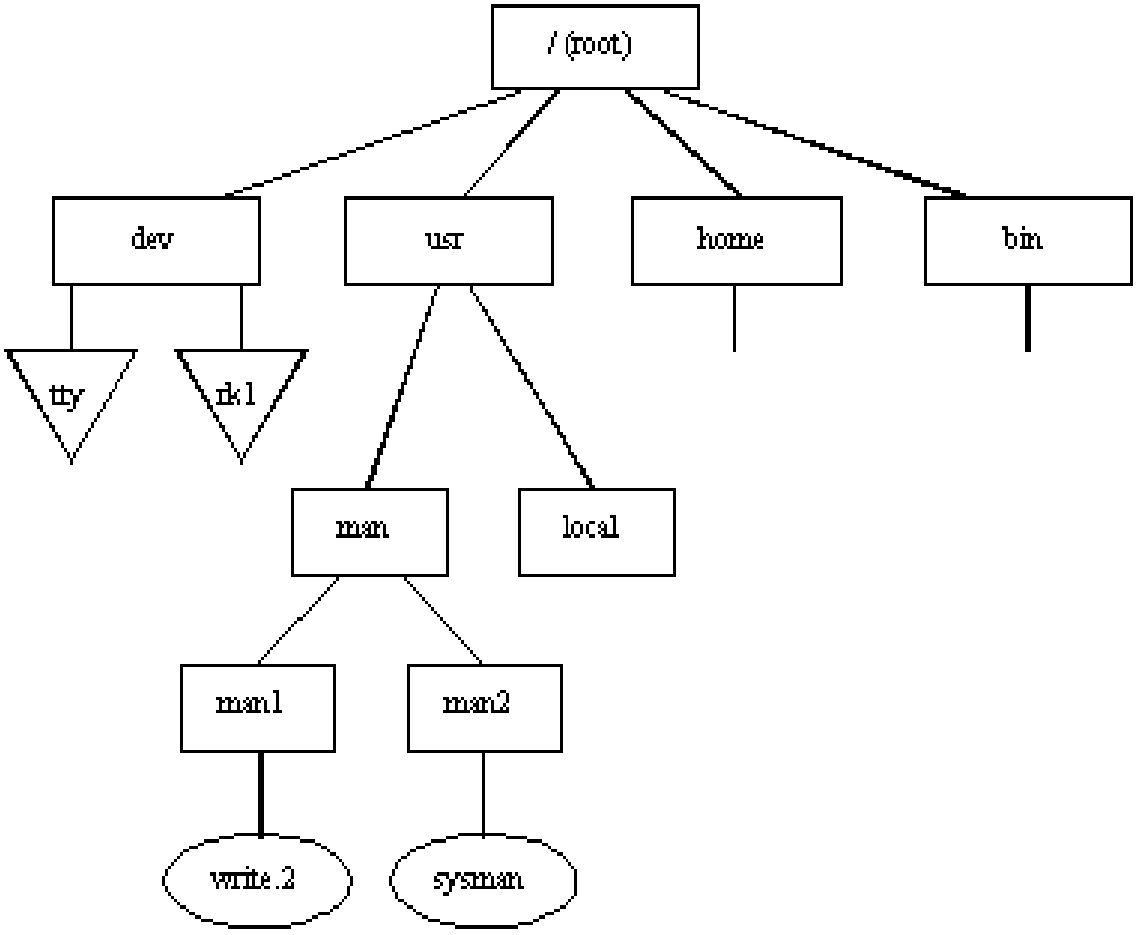


Рис. 15.2 Общая файловая система (после монтирования)

15.2.3 Имена файлов в файловой системе S5

В UNIX для файла существует *три типа имени*:

1. краткое;
2. полное;
3. относительное.

Краткое имя идентифицирует файл в пределах одного каталога. Оно может состоять не более чем из 14 символов и содержать так называемый суффикс, отделяемый точкой.

Полное имя однозначно определяет файл. Оно состоит из цепочки имен каталогов, через которые проходит маршрут от корневого каталога до данного файла. Имена каталогов разделяются символами "/", при этом имя корневого каталога не указывается, например, **/mnt/rk2/test.c** , где **mnt** и **rk2** - имена каталогов, а **test.c** - имя файла. Каждому полному имени в ОС соответствует только один файл, однако *файл может иметь несколько различных имен, так как ссылки на один и тот же файл могут содержаться в разных каталогах* (жесткие связи).

Относительное имя файла связано с понятием "текущий каталог", то есть каталог, имя которого задавать не нужно, так как оно подразумевается

по умолчанию. Имя файла относительно текущего каталога называется относительным. Оно представляет собой цепочку имен каталогов, через которые проходит маршрут от текущего каталога до данного файла. *Относительное имя в отличие от полного не начинается с символа "/".* Если в предыдущем примере принять за текущий каталог **/mnt**, то относительное имя файла **test.c** будет **rk2/test.c**.

15.2.4 Привилегии доступа в файловой системе S5

В UNIX S5 все *пользователи по отношению к данному файлу делятся на три категории*:

1. владелец;
2. член группы владельца;
3. все остальные.

Группа - это пользователи, которые объединены по какому-либо признаку, например, по принадлежности к одной разработке.

Кроме этого, в системе существует *суперпользователь, обладающий абсолютными правами по доступу ко всем файлам системы.*

Определены три вида доступа к файлу:

1. чтение;
2. запись;
3. выполнение.

Привилегии доступа к каждому файлу определены для каждой из трех категорий пользователей и для каждой из трех операций доступа. Начальные значения прав доступа к файлу устанавливаются при его создании операционной системой и могут изменяться его владельцем или суперпользователем.

15.2.5 Физическая организация файла в файловой системе S5

В общем случае *файл может располагаться в несмежных блоках дисковой памяти.*Логическая последовательность блоков в файле задается набором из 13 элементов.

Первые 10 элементов предназначаются для непосредственного указания номеров первых 10 блоков файла.

Если размер файла превышает 10 блоков, то *в 11 элементе* указывается номер блока, в котором содержится список следующих 128 блоков файла.

Если файл имеет размер более, чем 10+128 блоков, то используется 12-й элемент, содержащий номер блока, в котором указываются номера 128 блоков, каждый из которых может содержать еще по 128 номеров блоков файла. Таким образом, *12-й элемент* используется для двухуровневой косвенной адресации.

В случае, если файл больше, чем $10+128+128^2$ блоков, то используется 13 элемент для трехуровневой косвенной адресации. При таком способе адресации предельный размер файла составляет 2 113 674 блока.

Традиционная файловая система S5 поддерживает размеры блоков 512, 1024 или 2048 байт.

15.2.6 Структуры индексных дескрипторов и каталогов в системе S5

Вся необходимая операционной системе *информация о файле, кроме его символьного имени, хранится в специальной системной таблице, называемой индексным дескриптором* (**inode**) *файла.*

Индексные дескрипторы всех файлов имеют одинаковый размер - 64 байта и *содержат данные:*

1. о типе файла;
2. о физическом расположении файла на диске (13 элементов);
3. о размере в байтах;
4. о дате создания, последней модификации, последнего обращения к файлу;
5. о привилегиях доступа и некоторую другую информацию.

Индексные дескрипторы пронумерованы и хранятся в специальной области файловой системы. *Номер индексного дескриптора является уникальным именем файла.* Соответствие между полными символьными именами файлов и их уникальными именами устанавливается с помощью иерархии каталогов.

Каталог представляет собой совокупность записей обо всех файлах и каталогах, входящих в него. Каждая запись состоит из 16 байтов:

1. 14 байтов отводится под короткое символьное имя файла или каталога;
2. 2 байта - под номер индексного дескриптора этого файла.

В каталоге файловой системы S5 непосредственно не указываются характеристики файлов. Такая организация файловой системы позволяет с меньшими затратами перестраивать систему каталогов.

Например, при включении или исключении файла из каталога идет манипулирование меньшими объемами информации. Кроме того, при включении одного и того же файла в разные каталоги не нужно иметь несколько копий, как характеристик, так и самих файлов. С этой целью *в индексном дескрипторе ведется учет ссылок на этот файл из всех каталогов.* Как только число ссылок становится равным нулю, индексный дескриптор данного файла уничтожается.

Расположение файловой системы S5 на диске показано на рисунке 15.3.

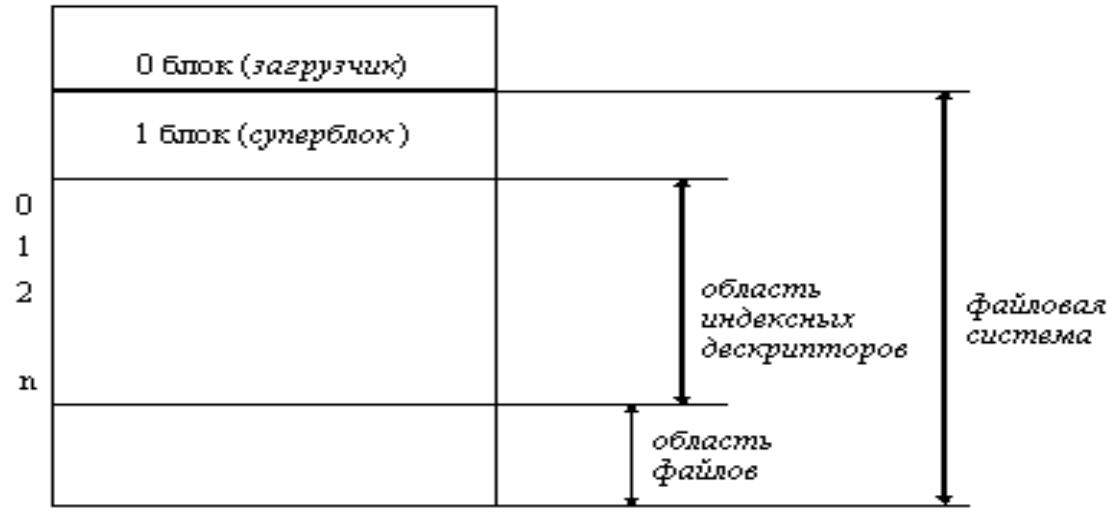


Рис. 15.3 Расположение файловой системы S5 на диске

Все *дисковое пространство, отведенное под файловую систему, делится на четыре области:*

1. *загрузочный блок (boot),* в котором хранится загрузчик операционной системы;
2. *суперблок (superblock)* - содержит самую общую информацию о файловой системе:
   2.1. размер файловой системы;
   2.2. размер области индексных дескрипторов;
   2.3. число индексных дескрипторов;
   2.4. список свободных блоков;
   2.5. список свободных индексных дескрипторов;
   2.6. другую административную информацию;
3. *область индексных дескрипторов*, порядок расположения индексных дескрипторов в которой соответствует их номерам;
4. *область данных*, в которой расположены обычные файлы и файлы-каталоги. Специальные файлы представлены в файловой системе только записями в соответствующих каталогах и индексными дескрипторами специального формата, но места в области данных не занимают.

Доступ к файлу осуществляется путем последовательного просмотра всей цепочки каталогов, входящих в полное имя файла, и соответствующих им индексных дескрипторов. *Поиск файла завершается после получения всех характеристик из индексного дескриптора заданного файла.*

Эта процедура требует в общем случае нескольких обращений к диску, пропорционально числу составляющих в полном имени файла. *Для уменьшения среднего времени доступа к файлу его дескриптор копируется в специальную системную область памяти.* Копирование индексного дескриптора входит в процедуру открытия файла.

При открытии файла ядро выполняет следующие действия:

1. Проверяет, существует ли файл; если не существует, то можно ли его создать. Если существует, то разрешен ли к нему доступ требуемого вида.
2. Копирует индексный дескриптор с диска в оперативную память; если с указанным файлом уже ведется работа, то новая копия индексного дескриптора не создается.
3. Создает в области ядра структуру, предназначенную для отображения текущего состояния операции обмена данными с указанным файлом. Эта структура, называемая **file**, содержит данные о типе операции (чтение, запись или чтение и запись), о числе считанных или записанных байтов, указатель на байт файла, с которым проводится операция.
4. Делает отметку в контексте процесса, выдавшего системный вызов на операцию с данным файлом.

15.3 ВИРТУАЛЬНАЯ ФАЙЛОВАЯ СИСТЕМА VFS

В UNIX System V Release 3 был реализован *механизм переключения файловых систем* (File System Switch, FSS), *позволяющий операционной системе поддерживать различные типы файловых систем*.

В соответствии с этим подходом информация о файловой системе и файлах разбивается на две части - зависимую от типа файловой системы и не зависимую. *FSS обеспечивает интерфейс между ядром и файловой системой, транслируя запросы ядра в операции, зависящие от типа файловой системы.* При этом ядро имеет представление только о независимой части файловой системы.

В UNIX System V Release 4 используется схема, реализованная фирмой Sun Microsystems с аналогичным подходом. Эта схема называется переключателем виртуальной файловой системы - **Virtual File System (VFS)**.

VFS не ориентируется на какую-либо конкретную файловую систему, механизмы реализации файловой системы полностью скрыты как от пользователя, так и от приложений. В ОС нет системных вызовов, предназначенных для работы со специфическими типами файловой системы, а имеются абстрактные вызовы типа open, read, write и другие, которые имеют содержательное описание, обобщающее некоторым образом содержание этих операций в наиболее популярных типах файловых систем (например, **s5, ufs, nfs** и т.п.). VFS также предоставляет ядру возможность оперирования файловой системой, как с единым целым:

1. операции монтирования и демонтирования;
2. операции получения общих характеристик конкретной файловой системы (размера блока, количества свободных и занятых блоков и т.п.) в единой форме.

Если конкретный тип файловой системы не поддерживает какую-то абстрактную операцию VFS, то файловая система должна вернуть ядру код возврата, извещающий об этом факте.

В VFS вся информация о файлах разделена на две части - не зависящую от типа файловой системы, которая хранится в специальной структуре ядра - структуре vnode, и зависящую от типа файловой системы - структура inode, формат которой на уровне VFS не определен. Используется только ссылка на нее в структуре vnode. Имя inode не означает, что эта структура совпадает со структурой индексного дескриптора inode файловой системы s5.

Виртуальная файловая система VFS поддерживает следующие типы файлов:

1. обычные файлы;
2. каталоги;
3. специальные файлы;
4. именованные конвейеры;
5. символьные связи.

Содержательное описание обычных файлов, каталогов и специальных файлов и связей не отличается от их описания в файловой системе s5.

15.3.1 Символьные связи

Версия UNIX System V Release 4 вводит новый тип связи - *мягкая связь*, называемая символьной связью и реализуемая с помощью системного вызова **symlink**.

Символьная связь - это файл данных, содержащий имя файла, с которым предполагается установить связь. Слово "предполагается" использовано потому, что символьная связь может быть создана даже с несуществующим файлом. При создании символьной связи образуется как новый вход в каталоге, так и новый индексный дескриптор inode. Кроме этого, резервируется отдельный блок данных для хранения полного имени файла, на который он ссылается.

Многие системные вызовы пользуются файлом символьных связей для поиска реального файла. Связанные файлы не обязательно располагаются в той же файловой системе.

Имеются *три системных вызова, которые имеют отношение к символьным связям*:

1. **readlink** - чтение полного имени файла или каталога, на который ссылается символьная связь. Эта информация хранится в блоке, связанном с символьной связью.
2. **lstat** - аналогичен системному вызову **stat**, но используется для получения информации о самой связи.
3. **lchown** - аналогичен системному вызову **chown**, но используется для изменения владельца самой символьной связи.

15.3.2 Именованные конвейеры

Конвейер - это средство обмена данными между процессами. Конвейер буферизует данные, поступающие на его вход, таким образом, что процесс, читающий данные на его выходе, получает их в порядке "первый пришел - первый вышел" (FIFO). В ранних версиях UNIX для обмена данными между процессами использовались неименованные конвейеры - **pipes**, которые представляли собой очереди байт в оперативной памяти.

Из-за отсутствия имен, такие конвейеры могли использоваться только для передачи данных между родственными процессами, получившими указатель на конвейер в результате копирования сегмента данных из адресного пространства процесса-прародителя. *Именованные конвейеры позволяют обмениваться данными произвольной паре процессов*, т.к. каждому такому конвейеру соответствует файл на диске. Никакие данные не связываются с файлом-конвейером, но все равно в каталоге содержится запись о нем, и он имеет индексный дескриптор. В UNIX System V Release 4 конвейеры реализуются с использованием коммуникационных модулей **streams**.

15.3.3 Файлы, отображенные в памяти

Новая архитектура виртуальной памяти UNIX System V Release 4 позволяет отображать содержимое файла (или устройства) в виде последовательности байтов в виртуальное адресное пространство процесса. Это упрощает процедуру доступа процесса к данным.

15.3.4 Реализация файловой системы VFS

UNIX System V Release 4 имеет массив структур **vfssw[…]**, каждая из которых описывает файловую систему конкретного типа, которая может быть установлена в системе. Структура **vfssw** состоит из четырех полей:

1. символьного имени файловой системы;
2. указателя на функцию инициализации файловой системы;
3. указателя на структуру, описывающую функции, реализующие абстрактные операции VFS в данной конкретной файловой системе;
4. флаги, которые не используются в описываемой версии UNIX.

Пример инициализированного массива структур vfssw:

```
struct vfssw vfssw[] = {
{0, 0 , 0 ,0 }, - нулевой элемент не используется
{"spec",  specint,  &spec_vfsops,    0},  - SPEC
{"vxfs",  vx_init,  &vx_vfsops,      0},  - Veritas
{"cdfs",  cdfsinit, &cdfs_vfsops,    0},  - CD ROM
{"ufs",   ufsinit,  &ufs_vfsops,     0},  - UFS
{"s5",    vx_init,  &vx_vfsops,      0},  - S5
{"fifo",  fifoinit, &fifovfsops,     0},  - FIFO
{"dos",   dosinit,  &dos_vfsops,     0},  - MS-DOS
```

Функции инициализации файловых систем вызываются во время инициализации операционной системы. Эти функции ответственны за создание внутренней среды файловой системы каждого типа.

Структура **vfsops**, описывающая операции, которые выполняются над файловой системой, состоит из 7 полей, так как в UNIX System V Release 4 предусмотрено 7 абстрактных операций над файловой системой:

VFS_MOUNT	монтирование файловой системы
VFS_UNMOUNT	размонтирование файловой системы
VFS_ROOT	получение vnode для корня файловой системы
VFS_STATVFS	получение статистики файловой системы
VFS_SYNC	выталкивание буферов файловой системы на диск
VFS_VGET	получение vnode по номеру дескриптора файла
VFS_MOUNTROOT	монтирование корневой файловой системы

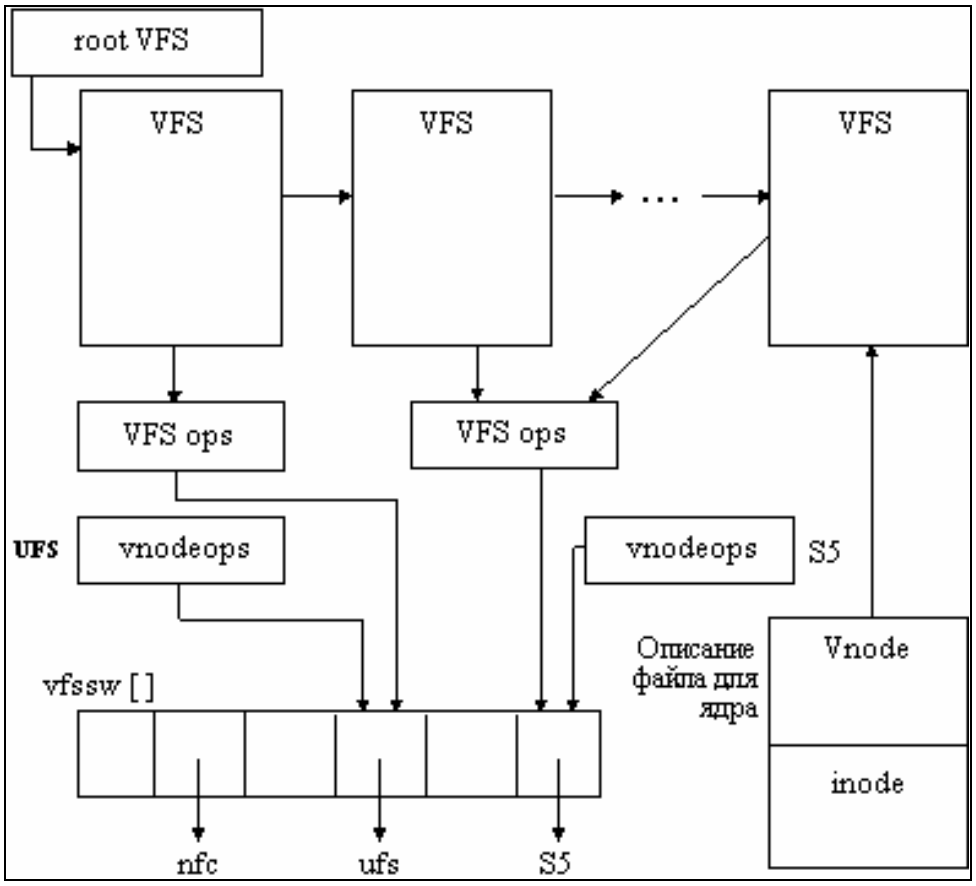


Рис. 15.4 Монтирование файловых систем в VFS

Операция **vfs_mount** выполняет традиционное для UNIX монтирование файловой системы на указанный каталог уже смонтированной файловой системы для образования общего дерева, а операция **vfs_unmount** отменяет монтирование.

Операция **vfs_root** используется при разборе полного имени файла, когда встречается дескриптор vnode, который связан со смонтированной на него файловой системой. Операция **vfs_root** помогает найти vnode, который является корнем смонтированной файловой системы.

Операция **vfs_statvfs** позволяет получить независимую от типа файловой системы информацию *о размере блока файловой системы, о количестве блоков и количестве свободных блоков в единицах этого размера, о максимальной длине имени файла и т.п*.

Операция **vfs_sync** выталкивает содержимое буферов диска из оперативной памяти на диск.

Операция **vfs_mountroot** позволяет смонтировать корневую файловую систему, то есть систему, содержащую корневой каталог / общего дерева. Для указания того, какая файловая система будет монтироваться как корневая, в UNIX System V Release 4 используется переменная **rootfstype**, содержащая символьное имя корневой файловой системы, например **"ufs"**.

Таким образом, в UNIX System V Release 4 одновременно в единое дерево могут быть смонтированы несколько файловых систем различных типов, поддерживающих операцию монтирования (рисунок 15.4).

VOP_OPEN	открыть файл
VOP_CLOSE	закрыть файл
VOP_READ	читать из файла
VOP_WRITE	записать в файл
VOP_IOCTL	управление в/в
VOP_SETFL	установить флаги статуса
VOP_GETATTR	получить атрибуты файла
VOP_SETATTR	установить атрибуты файла
VOP_LOOKUP	найти vnode по имени файла
VOP_CREATE	создать файл
VOP_REMOVE	удалить файл
VOP_LINK	связать файл
VOP_MAP	отобразить файл в память

Рис. 15.5 Абстрактные операции над файлами

Кроме операций над файловой системой в целом, для каждого типа файловой системы (**s5, ufs** и т.д.), установленной в ОС, необходимо описать способ реализации абстрактных операций над файлами, которые допускаются в VFS. Этот способ описывается для каждого типа файловой системы в структуре vnodeops, состав которой приведен на рисунке 15.5.

Как видно из состава списка абстрактных операций, они образованы объединением операций, характерных для наиболее популярных файловых систем UNIX. Для того, чтобы обращение к специфическим функциям не зависело от типа файловой системы, для каждой операции **в vnodeops** определен макрос с общим для всех типов файловых систем именем, например, **vop_open, vop_close, vop_read** и т.п. Эти макросы определены в файле <sys/vnode.h> и соответствуют системным вызовам.

Таким образом, в структуре **vnodeops** скрыты зависящие от типа файловой системы реализации стандартного набора операций над файлами. Даже если файловая система какого-либо конкретного типа не поддерживает определенную операцию над своими файлами, она должна создать соответствующую функцию, которая выполняет некоторый минимум действий: или сразу возвращает успешный код завершения, или возвращает код ошибки. Для анализа и обработки полного имени файла в VFS используется операция **vop_lookup**, позволяющая по имени файла найти ссылку на его структуру vnode.

Работа ядра с файлами во многом основана на использовании структуры **vnode**, поля которой представлены на рисунке 15.6.

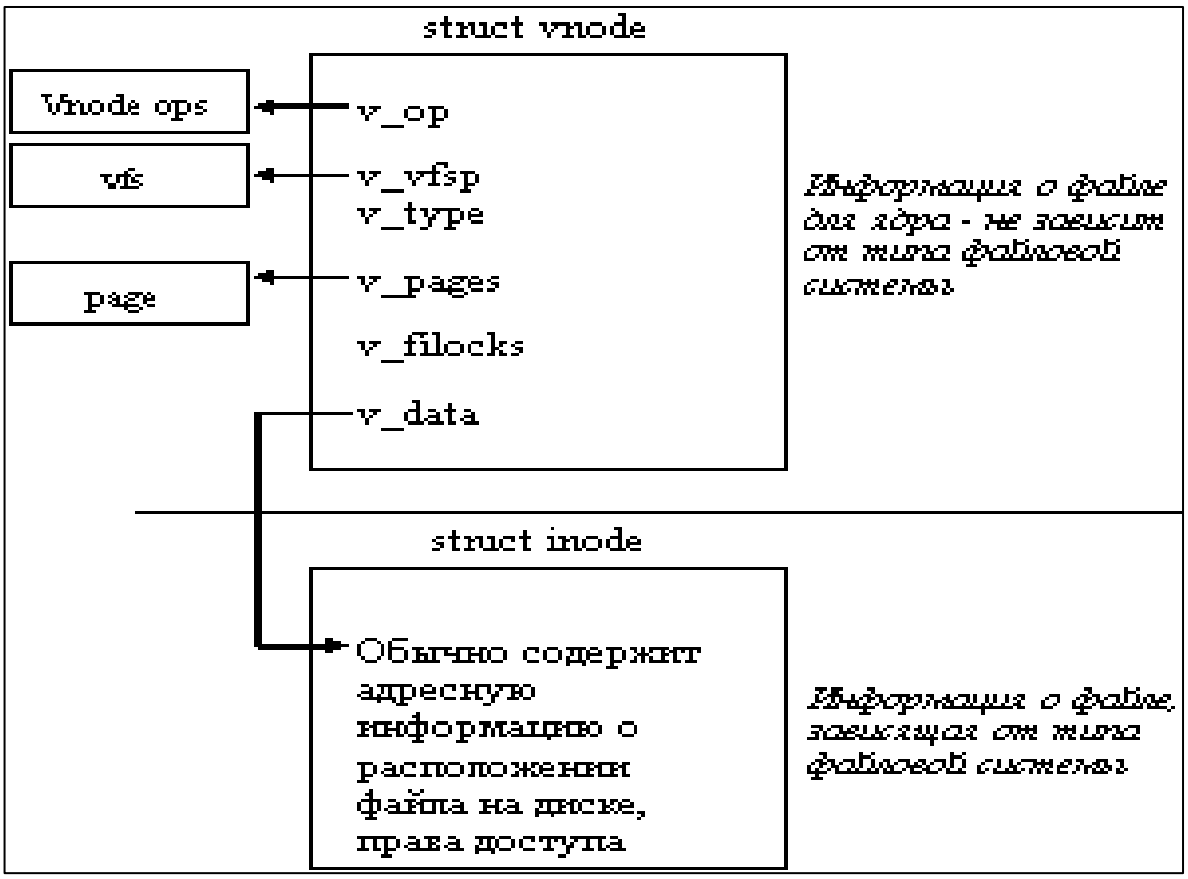


Рис. 15.6 Описатель файла - vnode

Структура **vnode** используется ядром для связи файла с определенным типом файловой системы через поле **v_vfsp** и конкретными реализациями файловых операций через поле **v_op.**

Поле **v_pages** используется для указания на таблицу физических страниц памяти в случае, когда файл отображается в физическую память.

В **vnode** также содержится тип файла и указатель на зависимую от типа файловой системы часть описания характеристик файла - структуру **inode,** обычно содержащую адресную информацию о расположении файла на носителе и о правах доступа к файлу. Кроме этого, **vnode** используется ядром для хранения информации о блокировках (locks), примененных процессами к отдельным областям файла.

Ядро в своих операциях с файлами оперирует для описания области файла **парой vnode, offset**, которая *однозначно определяет файл и смещение в байтах внутри файла.*

При каждом открытии процессом файла ядро создает в системной области памяти новую структуру типа **file**, которая, как и в случае традиционной файловой системы s5, описывает как открытый файл, так и операции, которые процесс собирается производить с файлом (например, чтение).

Структура **file** *содержит такие поля, как:*

1. **flag** - определение режима открытия (только для чтения, для чтения и записи и т.п.);
2. **struct vnode * f_vnode** - указатель на структуру vnode (заменивший по сравнению с S5 указатель на inode);
3. **offset** - смещение в файле при операциях чтения/записи;
4. **struct cred * f_cred** - указатель на структуру, содержащую права процесса, открывшего файл (структура находится в дескрипторе процесса);
5. указатели на предыдущую и последующую структуру типа file, связывающие все такие структуры в список.

Связь структур процесса с системным списком структур **file** показана на рисунке 15.7

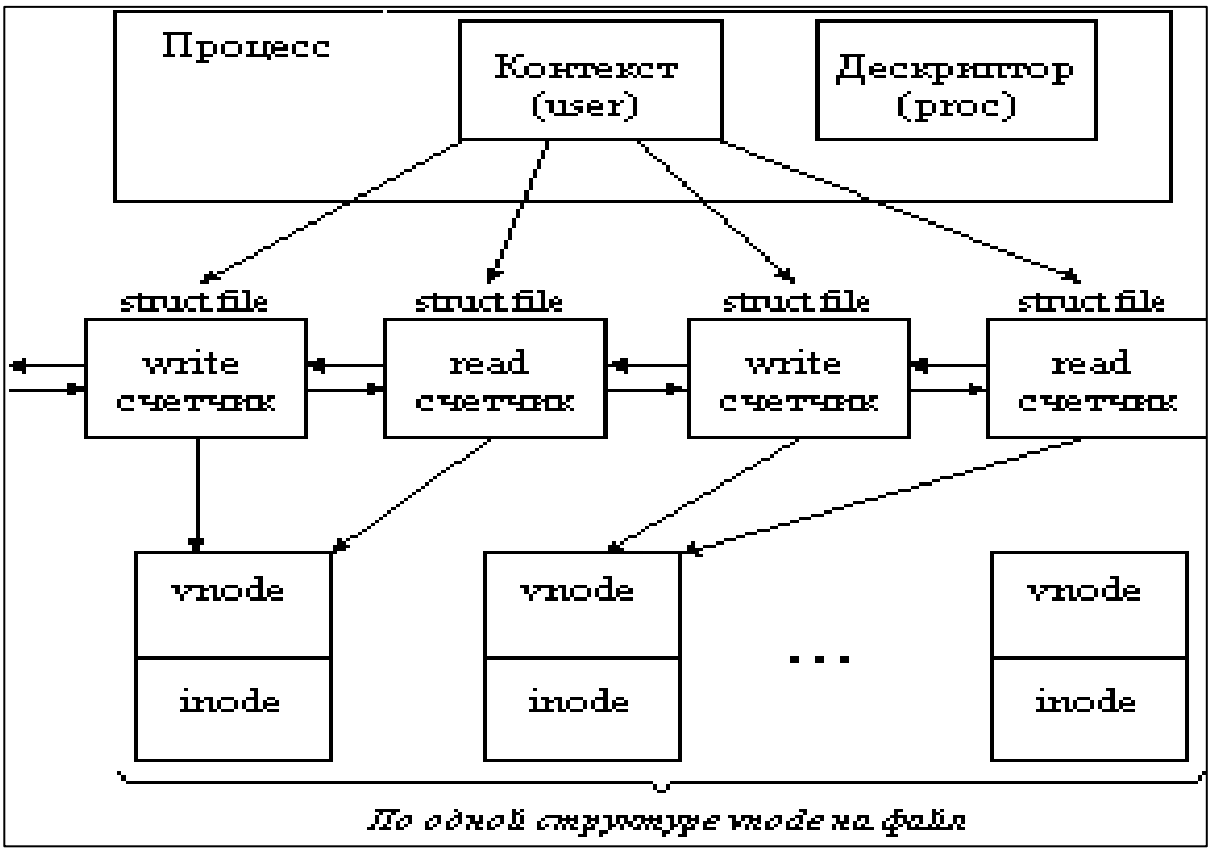


Рис. 15.7 Связь процесса с его файлами

В отличие от структур типа **file** структуры типа **vnode** заводятся операционной системой для каждого активного (открытого) файла в единственном экземпляре, поэтому структуры file могут ссылаться на одну и ту же структуру vnode.

Структуры vnode не связаны в какой-либо список. Они появляются по требованию в системном пуле памяти и присоединяются к структуре данных, которая инициировала появление этого vnode, с помощью соответствующего указателя. Например, в случае структуры file в ней

используется указатель **f_vnode** на соответствующую структуру vnode, описывающую нужный файл. Аналогично, если файл связан с образом процесса (то есть это файл, содержащий выполняемый модуль), то отображение сегмента памяти, содержащего части этого файла, осуществляется посредством указателя **vp** (в структуре **segvn_data**) на vnode этого файла.

Все операции с файлами в UNIX System V Release 4 производятся с помощью связанной с файлом структуры vnode. Когда процесс запрашивает операцию с файлом (например, операцию **open**), то независимая от типа файловой системы часть ОС передает управление зависимой от типа файловой системы части ОС для выполнения операции. *Если зависимая часть обнаруживает, что структуры vnode, описывающей нужный файл, нет в оперативной памяти, то зависимая часть заводит для него новую структуру vnode.*

Для ускорения доступа к файлам в UNIX System V Release 4 используется механизм быстрой трансляции имен файлов в соответствующие им ссылки на структуры vnode. Этот механизм основан на наличии кэша, *хранящего максимально 800 записей* об именах файлов и указателях vnode.