УЧРЕЖДЕНИЕ ОБРАЗОВАНИЯ
«Брестский государственный университет имени А.С. Пушкина»

Кафедра ПМИ

ОПЕРАЦИОННЫЕ СИСТЕМЫ

Электронный учебно-методический комплекс
по учебной дисциплине для студентов специальности
1-31 03 03-01 «Прикладная математика»
1-31 03 06-01 «Экономическая кибернетика»
физико-математического факультета

Брест
БрГУ имени А.С. Пушкина
2019

Рецензенты:

доцент кафедры алгебры, геометрии и математического моделирования учреждения образования «Брестский государственный университет имени А.С. Пушкина»,
кандидат физико-математических наук, доцент

З.Н. Серая

Кафедра информатики и прикладной математики Брестского государственного технического университета

Кондратюк, А.П.

Операционные системы : электронный учебно-методический комплекс для студ. специальности 1-31 03 03-01 «Прикладная математика (научно-производственная деятельность)» и 1-31 03 06-01 «Экономическая кибернетика (математические методы и компьютерное моделирование в экономике)» физ.-мат. фак. / А.П. Кондратюк ; Брест. гос. ун-т им. А.С. Пушкина, каф. ПМИ. – Брест : электрон. издание БрГУ, 2019. – 316 с.

ЭУМК написан в соответствии с действующей базовой программой по дисциплине «Операционные системы» и ставит своей целью облегчить самостоятельную работу студентов с теоретическим материалом при подготовке к лекциям и практическим занятиям.

Предназначено для студентов специальности 1-31 03 03-01 «Прикладная математика (научно-производственная деятельность)» и 1-31 03 06-01 «Экономическая кибернетика (математические методы и компьютерное моделирование в экономике)».

СОДЕРЖАНИЕ

Предисловие

Настоящий электронный учебно-методический комплекс предназначен для студентов специальностей 1-31 03 03-01 «Прикладная математика (научно-производственная деятельность)» и 1-31 03 06-01 «Экономическая кибернетика (математические методы и компьютерное моделирование в экономике)» физико-математического факультета. Он написан в соответствии с действующей базовой программой по дисциплине «Операционные системы».

Дисциплина «Операционные системы» ориентирована на обучение студентов принципам организации и функциям основных компонентов операционной системы. Особое внимание уделяется понятию процесса, его представлению в виртуальном адресном пространстве, разделению функций пользователя и ядра системы, организации мультизадачного режима, проблемам разделения ресурсов, синхронизации взаимодействующих процессов, ключевым решениям организации файловой системы как средства абстрагирования внешних устройств хранения данных и доступа к ним, а так же средствам поддержки виртуальной памяти и динамической компоновки исполняемых программ. Изучаемые темы базируются на использовании современных технологий, математических моделей и новейшего программного и технического обеспечения компьютеров. Теоретический материал иллюстрируется примерами.

Практический раздел представлен лабораторным практикумом.

Раздел контроля знаний представлен вопросами и тестовыми заданиями.

Учебно-методический комплекс ставит своей целью облегчить самостоятельную работу студентов с теоретическим и практическим материалом при подготовке к лекциям, лабораторным занятиям и зачету.

Автор.

Учебную программу можно посмотреть по ссылке.

ГЛАВА 1
Введение

1.1 Определение операционной системы

Обеспечение пользователю-программисту удобств посредством предоставления для него расширенной машины и повышение эффективности использования компьютера путем рационального управления его ресурсами.

1.2 ОС как расширенная машина

Использование большинства компьютеров на уровне машинного языка затруднительно, особенно это касается ввода-вывода. 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, очевидно, надо анализировать. Даже если не входить в курс реальных проблем программирования ввода-вывода, ясно, что среди программистов нашлось бы не много желающих непосредственно заниматься программированием этих операций. При работе с диском программисту-

пользователю достаточно представлять его в виде некоторого 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. Программа, которая скрывает от программиста все реалии аппаратуры и предоставляет возможность простого, удобного просмотра указанных файлов, чтения или записи - это, конечно, операционная система. Точно также, как ОС ограждает программистов от аппаратуры дискового накопителя и предоставляет ему простой файловый интерфейс, операционная система берет на себя все малоприятные дела, связанные с обработкой прерываний, управлением таймерами и оперативной памятью, а также другие низкоуровневые проблемы. В каждом случае та абстрактная, воображаемая машина, с которой, благодаря операционной системе, теперь может 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.

1.3 ОС как система управления ресурсами

Идея о том, что ОС прежде всего система, обеспечивающая удобный интерфейс пользователям, соответствует рассмотрению сверху вниз. Другой взгляд, снизу вверх, дает представление об ОС как о некотором механизме, управляющем всеми частями сложной системы. Современные вычислительные системы состоят из процессоров, памяти, таймеров, дисков, накопителей на магнитных лентах, 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. Критерием эффективности может быть, например, пропускная способность или реактивность системы.

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

- планирование ресурса - то есть определение, кому, когда, а для делимых ресурсов и в каком количестве, необходимо выделить данный ресурс;
- 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 сво-

бодно.

Для решения этих общих задач управления ресурсами разные ОС используют различные алгоритмы, что в конечном счете и определяет их облик в целом, включая характеристики производительности, область применения и даже пользовательский интерфейс. Так, например, алгоритм управления процессором в значительной степени определяет, является ли ОС системой разделения времени, системой пакетной обработки или системой реального времени

1.4 Эволюция ОС

1.4.1 Первый период(1945 - 1955)

Известно, что компьютер был изобретен английским математиком Чарльзом Бэбиджем в конце восемнадцатого века. Его "аналитическая машина"так и не смогла но-настоящему заработать, потому что технологии того времени не удовлетворяли требованиям по изготовлению деталей точной механики, которые были необходимы для вычислительной техники. Известно также, что этот компьютер не имел операционной системы.

Некоторый прогресс в создании цифровых вычислительных машин произошел после второй мировой войны. В середине 40-х 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

1.4.2 Второй период(1955 - 1965)

С середины 50-х годов начался новый период в развитии вычислительной техники, связанный с появлением новой технической базы - полупроводниковых элементов. Компьютеры второго поколения стали более надежными, теперь они смогли непрерывно работать настолько долго, чтобы на них можно было возложить выполнение действительно практически важных задач. Именно в этот период произошло разделение персонала на программистов и операторов, эксплуатационщиков и разработчиков вычислительных машин.

В эти годы появились первые алгоритмические языки, а следовательно и первые системные программы - компиляторы. Стоимость процес-

сорного времени возросла, 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, 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.

1.4.3 Третий период(1965 - 1980)

Следующий важный период развития вычислительных машин относится к 1965-1980 годам. В это время в технической базе произошел переход от отдельных полупроводниковых элементов типа транзисторов к интегральным микросхемам, что дало гораздо большие возможности новому, третьему поколению компьютеров.

Для этого периода характерно также создание семейств программно-совместимых машин. Первым семейством программно-совместимых ма-

шин, построенных на интегральных микросхемах, явилась серия машин IBM/360. Построенное в начале 60-х годов это семейство значительно превосходило машины второго поколения по критерию цена/производительность. Вскоре идея программно-совместимых машин стала общепризнанной.

Программная совместимость требовала и совместимости операционных систем. Такие операционные системы должны были бы работать и на больших, и на малых вычислительных системах, с большим и с малым количеством разнообразной периферии, в коммерческой области и в области научных исследований. Операционные системы, построенные с намерением удовлетворить всем этим противоречивым требованиям, оказались чрезвычайно сложными "монстрами". Они состояли из многих миллионов ассемблерных строк, написанных тысячами программистов, и содержали тысячи ошибок, вызывающих нескончаемый поток исправлений. В каждой новой версии операционной системы исправлялись одни ошибки и вносились другие.

Однако, несмотря на необозримые размеры и множество проблем, OS/360 и другие ей подобные операционные системы машин третьего поколения действительно 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.

1.4.4 Четвертый период(1980 - настоящее время)

Следующий период в эволюции операционных систем связан с появлением больших интегральных схем (БИС). В эти годы произошло резкое возрастание степени интеграции и удешевление микросхем. Компьютер стал доступен отдельному человеку, и 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 ОС MS-DOS широко использовалась для компьютеров, построенных на базе микропроцессоров Intel 8088, а затем 80286, 80386 и 80486. Мультипрограммная многопользовательская ОС 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.

ГЛАВА 2
Классификация ОС

Операционные системы могут различаться особенностями реализации внутренних алгоритмов управления основными ресурсами компьютера (процессорами, памятью, устройствами), особенностями использованных методов проектирования, типами аппаратных платформ, областями использования и многими другими свойствами. Ниже приведена классификация ОС по нескольким наиболее основным признакам.

2.1 Особенности алгоритмов управления ресурсами

От эффективности алгоритмов управления локальными ресурсами компьютера во многом зависит эффективность всей сетевой ОС в целом. Поэтому, характеризуя сетевую ОС, часто приводят важнейшие особенности реализации функций ОС по управлению процессорами, памятью, внешними устройствами автономного компьютера. Так, например, в зависимости от особенностей использованного алгоритма управления процессором, операционные системы делят на многозадачные и однозадачные, многопользовательские и однопользовательские, на системы, поддерживающие многонитевую обработку и не поддерживающие ее, на многопроцессорные и однопроцессорные системы.

Поддержка многозадачности. По числу одновременно выполняемых задач операционные системы могут быть разделены на два класса:

- однозадачные (например, MS-DOS, MSX)
- многозадачные (ОС ЕС, OS/2, UNIX, Windows 95).

Однозадачные ОС в основном выполняют функцию предоставления пользователю виртуальной машины, делая более простым и удобным процесс взаимодействия пользователя с компьютером. Однозадачные ОС включают средства управления периферийными устройствами, средства управления файлами, средства общения с пользователем.

Многозадачные ОС, кроме вышеперечисленных функций, управляют разделением совместно используемых ресурсов, таких как процессор, оперативная память, файлы и внешние устройства.

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

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

Главным отличием многопользовательских систем от однопользовательских является наличие средств защиты информации каждого пользователя от несанкционированного доступа других пользователей. Следует заметить, что не всякая многозадачная система является много-

пользовательской, и не всякая однопользовательская ОС является однозадачной.

Вытесняющая и невытесняющая многозадачность. Важнейшим разделяемым ресурсом является процессорное время. Способ распределения процессорного времени между несколькими одновременно существующими в системе процессами (или нитями) во многом определяет специфику ОС.

Среди множества существующих вариантов реализации многозадачности можно выделить две группы алгоритмов:

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

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

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.

В наши дни становится общепринятым введение в ОС функций поддержки многопроцессорной обработки данных. Такие функции 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 могут классифицироваться по способу организации вычислительного процесса в системе с многопроцессорной архитектурой: асимметричные ОС и симметричные ОС. Асимметричная ОС целиком выполняется только на одном из процессоров системы, распределяя прикладные задачи по остальным процессорам. Симметричная ОС полностью децентрализована и использует весь пул процессоров, разделяя их между системными и прикладными задачами.

Выше были рассмотрены характеристики ОС, связанные с управлением только одним типом ресурсов - процессором. Важное влияние на облик операционной системы в целом, на возможности ее использова-

ния 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. При реализации сетевых функций возникает комплекс задач, связанных с распределенным характером хранения и обработки данных в сети: ведение справочной информации о всех доступных в сети ресурсах и серверах, адресация взаимодействующих процессов, обеспечение прозрачности доступа, тиражирование данных, согласование копий, поддержка безопасности данных.

2.2 Особенности аппаратных платформ

На свойства операционной системы непосредственное влияние оказывают аппаратные средства, на которые она ориентирована. По типу аппаратуры различают операционные системы персональных компьютеров, мини-компьютеров, мейнфреймов, кластеров и сетей ЭВМ. Среди перечисленных типов компьютеров могут встречаться как однопроцессорные варианты, так и многопроцессорные. В любом случае специфика аппаратных средств, как правило, отражается на специфике операционных систем. Очевидно, что ОС большой машины является более

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

Сетевая ОС имеет в своем составе средства передачи сообщений между компьютерами по линиям связи, которые совершенно не нужны в автономной ОС. На основе этих сообщений сетевая ОС поддерживает разделение ресурсов компьютера между удаленными пользователями, подключенными к сети. Для поддержания функций передачи сообщений сетевые ОС содержат специальные программные компоненты, реализующие популярные коммуникационные протоколы, такие как IP, IPX, Ethernet и другие.

Многопроцессорные системы требуют от операционной системы особой организации, с помощью которой сама операционная система, а также поддерживаемые ею приложения могли бы выполняться параллельно отдельными процессорами системы. Параллельная работа отдельных частей ОС создает дополнительные проблемы для разработчиков ОС, так как в этом случае гораздо сложнее обеспечить согласованный доступ отдельных процессов к общим системным таблицам, исключить эффект гонок и прочие нежелательные последствия асинхронного выполнения работ. Другие требования предъявляются к операционным си-

стемам кластеров. Кластер - слабо связанная совокупность нескольких вычислительных систем, работающих совместно для выполнения общих приложений, и представляющихся пользователю единой системой. Наряду со специальной аппаратурой для функционирования кластерных систем необходима и программная поддержка со стороны операционной системы, которая сводится в основном к синхронизации доступа к разделяемым ресурсам, обнаружению отказов и динамической реконфигурации системы. Одной из первых разработок в области кластерных технологий были решения компании Digital Equipment на базе компьютеров VAX. Недавно этой компанией заключено соглашение с корпорацией Microsoft о разработке кластерной технологии, использующей 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 C, который и был разработан для программирования операционных систем.

2.3 Особенности областей использования

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

- системы пакетной обработки (например, ОС ЕС),
- системы разделения времени (UNIX, VMS),
- системы реального времени (QNX, RT/11).

Системы пакетной обработки предназначались для решения задач в основном вычислительного характера, не требующих быстрого получения результатов. Главной целью и критерием эффективности систем пакетной обработки является максимальная пропускная способность, то есть решение максимального числа задач в единицу времени. Для достижения этой цели в системах пакетной обработки используются следующая схема функционирования: в начале работы формируется пакет заданий, каждое задание содержит требование к системным ресурсам; из этого пакета заданий формируется мультипрограммная смесь, то есть множество одновременно выполняемых задач. Для одновременного выполнения выбираются задачи, предъявляющие отличающиеся требования к ресурсам, так, чтобы обеспечивалась сбалансированная загрузка всех устройств вычислительной машины; так, например, в мультипрограммной смеси желательно одновременное присутствие вычислительных задач и задач с интенсивным вводом-выводом. Таким образом, вы-

бор нового задания из пакета заданий зависит от внутренней ситуации, складывающейся в системе, то есть выбирается "выгодное"задание. Следовательно, в таких ОС невозможно гарантировать выполнение того или иного задания в течение определенного периода времени. В системах пакетной обработки переключение процессора с выполнения одной задачи на выполнение другой происходит только в случае, если активная задача сама отказывается от процессора, например, из-за необходимости выполнить операцию ввода-вывода. Поэтому одна задача может надолго занять процессор, что делает невозможным выполнение интерактивных задач. Таким образом, взаимодействие пользователя с вычислительной машиной, на которой установлена система пакетной обработки, сводится к тому, что он приносит задание, отдает его диспетчеру-оператору, а в конце дня после выполнения всего пакета заданий получает результат. Очевидно, что такой порядок снижает эффективность работы пользователя.

Системы разделения времени призваны исправить основной недостаток систем пакетной обработки - изоляцию пользователя-программиста от процесса выполнения его задач. Каждому пользователю системы разделения времени предоставляется терминал, с которого он может вести диалог со своей программой. Так как в системах разделения времени каждой задаче выделяется только квант процессорного времени, ни одна задача не занимает процессор надолго, и время ответа оказывается приемлемым. Если квант выбран достаточно небольшим, то у всех

пользователей, одновременно работающих на одной и той же машине, складывается впечатление, что каждый из них единолично использует машину. Ясно, что системы разделения времени обладают меньшей пропускной способностью, чем системы пакетной обработки, так как на выполнение принимается каждая запущенная пользователем задача, а не та, которая "выгодна"системе, и, кроме того, имеются накладные расходы вычислительной мощности на более частое переключение процессора с задачи на задачу. Критерием эффективности систем разделения времени является не максимальная пропускная способность, а удобство и эффективность работы пользователя.

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

реакции системы, а соответствующее свойство системы - реактивностью. Для этих систем мультипрограммная смесь представляет собой фиксированный набор заранее разработанных программ, а выбор программы на выполнение осуществляется исходя из текущего состояния объекта или в соответствии с расписанием плановых работ.

Некоторые операционные системы могут совмещать в себе свойства систем разных типов, например, часть задач может выполняться в режиме пакетной обработки, а часть - в режиме реального времени или в режиме разделения времени. В таких случаях режим пакетной обработки часто называют фоновым режимом.

2.4 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 привилеги-

рованного режима в пользовательский и наоборот. Альтернативой является построение ОС на базе микроядра, работающего также в привилегированном режиме и выполняющего только минимум функций по управлению аппаратурой, в то время как функции ОС более высокого уровня выполняют специализированные компоненты ОС - серверы, работающие в пользовательском режиме. При таком построении ОС работает более медленно, так как часто выполняются переходы между привилегированным режимом и пользовательским, зато система получается более гибкой - ее функции можно наращивать, модифицировать или сужать, добавляя, модифицируя или исключая серверы пользовательского режима. Кроме того, серверы хорошо защищены друг от друга, как и любые пользовательские процессы.

- Построение ОС на базе объектно-ориентированного подхода дает возможность использовать все его достоинства, хорошо зарекомендовавшие себя на уровне приложений, внутри операционной системы, а именно: аккумуляцию удачных решений в форме стандартных объектов, возможность создания новых объектов на базе имеющихся с помощью механизма наследования, хорошую защиту данных за счет их инкапсуляции во внутренние структуры объекта, что делает данные недоступными для несанкционированного использования извне, структуризованность системы, состоящей из набора хорошо

определенных объектов.

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

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

распределенных служб.

ГЛАВА 3
Управление процессами

Важнейшей частью операционной системы, непосредственно влияющей на функционирование вычислительной машины, является подсистема управления процессами. *Процесс* (или по-другому, задача) - абстракция, описывающая выполняющуюся программу. Для операционной системы процесс представляет собой единицу работы, заявку на потребление системных ресурсов. Подсистема управления процессами планирует выполнение процессов, то есть распределяет процессорное время между несколькими одновременно существующими в системе процессами, а также занимается созданием и уничтожением процессов, обеспечивает процессы необходимыми системными ресурсами, поддерживает взаимодействие между процессами.

3.1 Состояние процессов

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

ВЫПОЛНЕНИЕ - активное состояние процесса, во время которого процесс обладает всеми необходимыми ресурсами и непосредственно выполняется процессором;

ОЖИДАНИЕ - пассивное состояние процесса, процесс заблоки-

рован, он 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;

ГОТОВНОСТЬ - также пассивное состояние процесса, но в этом случае процесс заблокирован в связи с внешними по отношению к нему обстоятельствами: процесс имеет все требуемые для него ресурсы, он 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.

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 соответственно 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 кван-

та 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.

3.2 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 состоянием регистров и программного счетчика, режимом работы процессора, указателями на открытые файлы, информацией о незавершенных операциях ввода-вывода, кодами ошибок выполняемых данным процессом системных вызовов и т.д. Эта информация называется *контекстом процесса*.

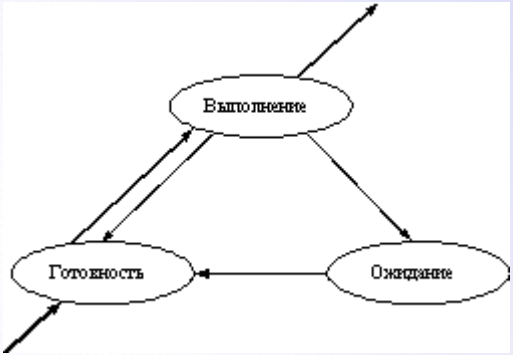


Рис. 3.1: Граф состояний процесса в многозадачной среде

Кроме этого, операционной системе для реализации планирования процессов требуется дополнительная информация: идентификатор процесса, состояние процесса, данные о степени привилегированности процесса, 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 про-

цессов, 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 только тогда начнет выполняться, 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.

3.3 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 в значительной степени аппаратно (см. раздел 2.3. *"Средства аппаратной поддержки управления памятью и многозадачной среды в микропроцессорах Intel 80386, 80486 и Pentium"*).

Существует множество различных алгоритмов планирования процессов, по разному решающих вышеперечисленные задачи, преследующих различные цели и обеспечивающих различное качество мультипрограммирования. Среди этого множества алгоритмов рассмотрим подробнее 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:

- процесс завершился и покинул систему,
- произошла ошибка,
- процесс перешел в состояние ОЖИДАНИЕ,
- 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.

Кванты, выделяемые процессам, могут быть одинаковыми для всех процессов или различными. Кванты, выделяемые одному процессу, могут быть фиксированной величины или изменяться в разные периоды жизни процесса. 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. 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 целыми или дробными, положительным или отрицательным значением. Чем выше привилегии процесса, тем меньше времени он будет проводить в очередях. Приоритет может на-

значаться директивно администратором системы в зависимости от важности работы или внесенной платы, либо 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: алгоритмы, использующие относительные приоритеты, и алгоритмы, использующие абсолютные приоритеты.

В обоих случаях выбор процесса на выполнение из очереди готовых осуществляется одинаково: 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 же произойдет ошибка, или процесс завершится). В системах с абсолютными приоритетами выполнение активного процесса прерывается еще при одном условии: 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.

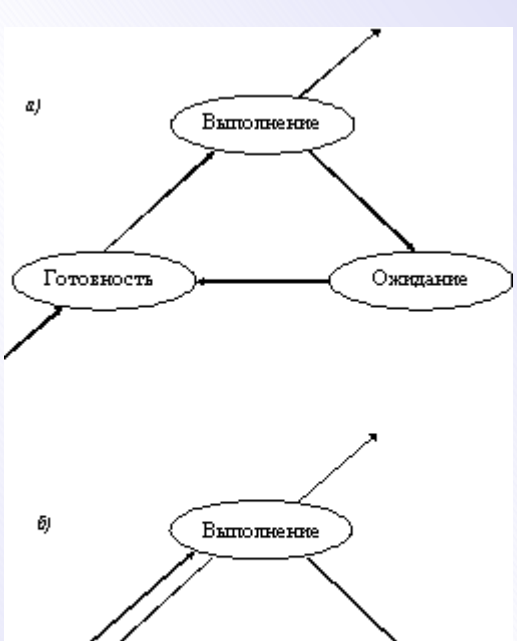


Рис. 3.2: Графы состояний процессов в системе (а) с относительными приоритетами; (б)с абсолютными приоритетами

Во многих операционных системах алгоритмы планирования построены с использованием как квантования, так и приоритетов. Например, в основе планирования лежит квантование, но 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.

Понятия *preemptive* и *non-preemptive* иногда отождествляются с понятиями приоритетных и бесприоритетных дисциплин, что совершенно неверно, а также с понятиями абсолютных и относительных приоритетов, что неверно отчасти. 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. Так в случае использования приоритетов дисциплина относительных приоритетов может быть отнесена к классу систем с невытесняющей многозадачностью, а дисциплина абсолютных приоритетов - к классу систем

с вытесняющей многозадачностью. А бесприоритетная дисциплина планирования, основанная на выделении равных квантов времени для всех задач, относится к вытесняющим алгоритмам.

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: определяет момент снятия с выполнения активной задачи, запоминает ее контекст, 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. Эта ситуация нежелательна, так как пользователи обычно не хотят долго ждать, когда машина завершит свою задачу.

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 работает, но он существенно затрудняет разработку программ и предъявляет повышенные требования к квалификации программиста. Программист должен обеспечить "дружественное"отношение своей программы к другим выполняемым одновременно с ней программам, достаточно часто отдавая им управление. Крайним проявлением "недружественности"приложения является его зависание, которое приводит к общему краху системы. В системах с вытесняющей многозадачностью такие ситуации, как правило, исключены, так как центральный планирующий механизм снимет зависшую

задачу с выполнения.

Однако распределение функций планировщика между системой и приложениями не всегда является недостатком, а при определенных условиях может быть и преимуществом, потому что дает возможность разработчику приложений самому проектировать алгоритм 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. Кроме того, легко разрешаются проблемы совместного использования данных: задача во время каждой итерации использует их монопольно и уверена, что на протяжении этого периода никто другой не изменит эти данные. Существенным преимуществом non-preemptive систем является 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. Менее удачным оказалось использование невытесняющей многозадачности в операционной среде Windows 3.x.

Однако почти во всех современных операционных системах, ориентированных на высокопроизводительное выполнение приложений (UNIX, Windows NT, OS/2, VAX/VMS), реализована вытесняющая многозадачность. В последнее время дошла очередь и до ОС класса настольных

систем, например, OS/2 Warp и Windows 95. Возможно в связи с этим вытесняющую многозадачность часто называют истинной многозадачностью.

ГЛАВА 4

Средства синхронизации и взаимодействия процессов

4.1 Проблема синхронизации

Процессам часто нужно взаимодействовать друг с другом, например, один процесс может передавать данные другому процессу, или несколько процессов могут обрабатывать данные из общего файла. Во всех этих случаях возникает проблема синхронизации процессов, которая может решаться приостановкой и активизацией процессов, организацией очередей, 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). 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 читают эту переменную, записывают в соответствующую позицию файла "заказов"имя своего файла и 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 предположим равным 4. Процесс запомнил это значение, но поместить имя файла не успел, так как его выполнение было прервано (например, в следствие исчерпания кванта). 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: были потеряны файлы нескольких процессов или, напротив, 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 Критическая секция

Важным понятием синхронизации процессов является понятие "критическая секция"программы. *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. Однако 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.

Рис. 4.1: Пример необходимости синхронизации

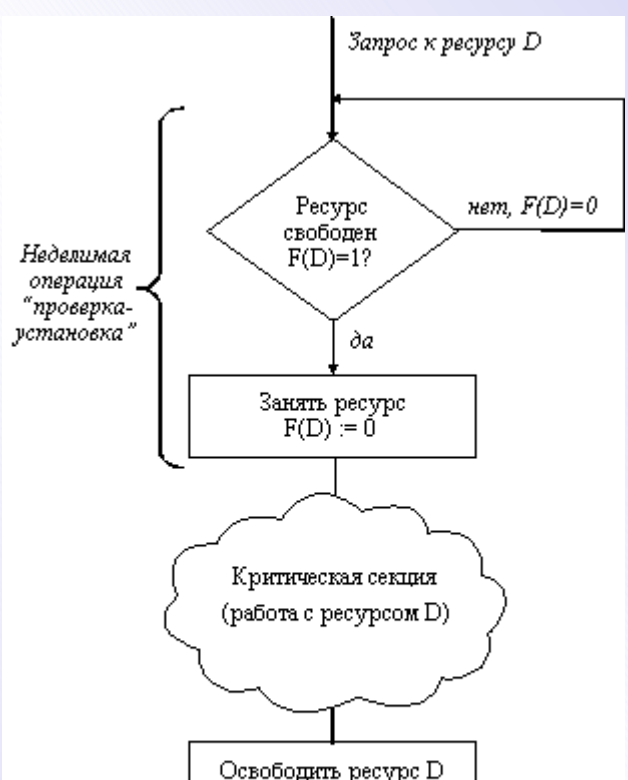


Рис. 4.2: Реализация критических секций с использованием блокирующих переменных

Другим способом является использование блокирующих переменных. С каждым разделяемым ресурсом связывается двоичная переменная, которая принимает значение 1, если ресурс свободен (то есть ни один процесс не находится в данный момент в критической секции, связанной с данным процессом), 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). 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. Следует заметить, что операция проверки и установки блокирующей переменной должна быть неделимой. Поясним это. 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 сек-

цию. Таким образом был нарушен принцип взаимного исключения, что потенциально может привести к нежелаемым последствиям. Во избежание таких ситуаций в системе команд машины желательно иметь единую команду "проверка-установка или же реализовывать системными средствами соответствующие 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 не выполняет циклический опрос, а вызывает системную функцию WAIT(D), здесь D обозначает событие, заключающееся в освобождении ресурса D. Функция WAIT(D) переводит активный процесс в состояние ОЖИДАНИЕ и делает отметку в его де-

скрипторе 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.

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:

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

P(S) : уменьшение S на 1, если это возможно. 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. 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, в то время как V-операция может при некоторых обстоятельствах активизировать другой процесс,

приостановленный операцией P (сравните эти операции с системными функциями WAIT и POST).

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.

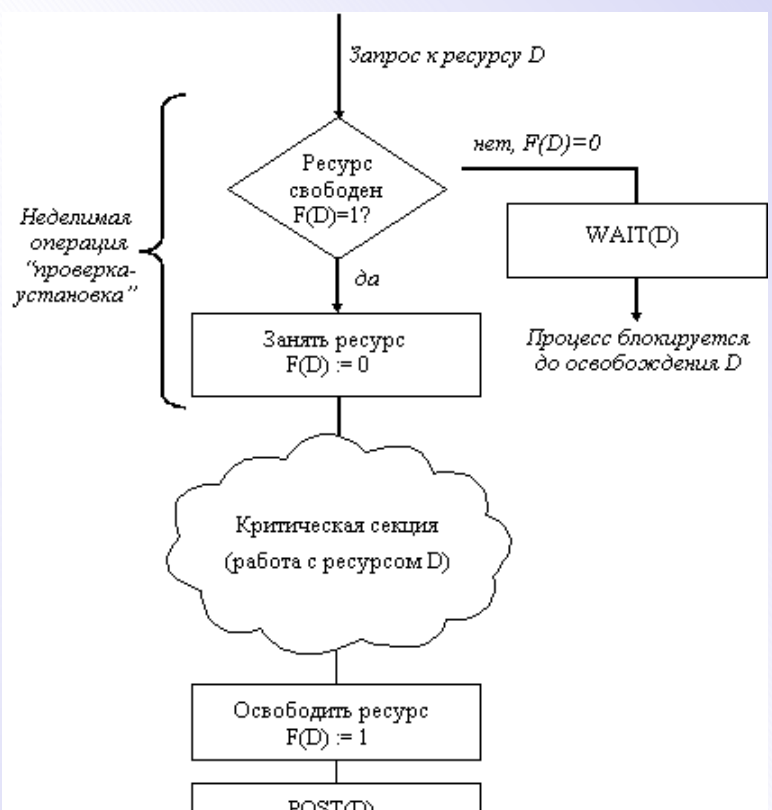


Рис. 4.3: Реализация критической секции с использованием системных функций WAIT(D) и POST(D)

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 в начале данного раздела). 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                   */
}
}
```

ГЛАВА 5
Тупики

Говорят, что в мультипрограммной системе процесс находится в состоянии *тупика*, *дедлока* или *клинча*, если он ожидает некоторого события, которое никогда не произойдет. Системная тупиковая ситуация, или «зависание» системы – это ситуация, когда один или более процессов оказываются в состоянии тупика. Рассмотрим пример тупика, возникающего при распределении ресурсов. Как правило, такие тупики возникают вследствие обычной конкуренции за обладание выделяемыми, или закрепляемыми ресурсами.

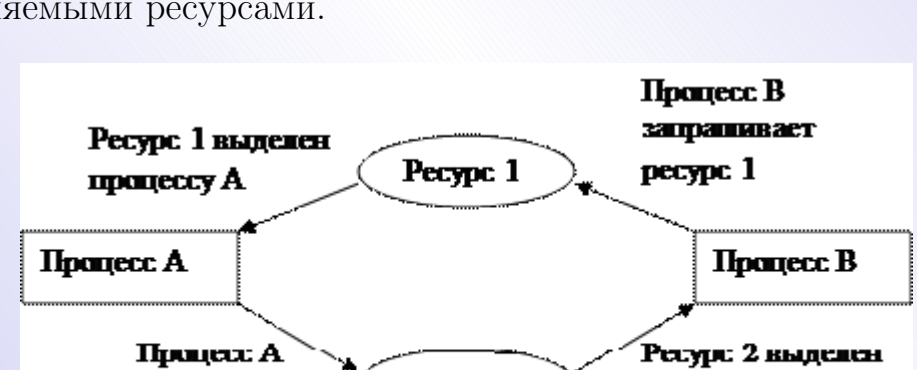


Рис. 5.1: Простая тупиковая ситуация

На рисунке 5.1 показана схема простого тупика. Каждый процесс удерживает и не желает освобождать ресурс, запрашиваемый другим процессом. В результате, оба процесса будут находиться в бесконечном ожидании освобождения запрашиваемого ресурса.

Другим примером тупика, является тупик в системе спулинга. Спулинг часто применяется для повышения производительности системы путем изолирования процессов от низкоскоростного оборудования. Если каждый процесс будет ждать распечатки каждой строки перед посылкой следующей – то такой процесс будет выполняться медленно. Поэтому строки данных, предназначенных для печати, сначала записываются на более высокоскоростное устройство, где они временно хранятся до момента распечатки. В некоторых системах спулинга, процесс должен сформировать все выходные данные – только после этого начинается реальная распечатка. В этом случае, несколько незавершенных заданий, могут оказаться в тупиковой ситуации, если предусмотренная емкость буфера будет заполнена до того, как завершиться выполнение какого-либо задания. Для восстановления может потребоваться перезапуск системы, с потерей всей работы, выполненной до этого момента. В качестве менее радикального способа, можно предложить уничтожение одного или нескольких заданий, пока у остающихся заданий не окажется достаточно свободного места в буфере. Более устойчивы в этом плане системы, позволяющие начинать печать до завершения задания, с тем чтобы буфер печати опустошался уже в процессе выполнения задания.

Третий классический пример – бесконечное откладывание. В системе, где процессам приходиться ждать, пока ОС не примет решения их запустить, может возникнуть ситуация, при которой предоставление процессора некоему процессу, будет откладываться на неопределенно долгий срок, в то время как ОС будет уделять внимание другим процессам. Бесконечное откладывание может происходить из-за распределения ресурсов по приоритетному принципу. (Процесс будет ожидать выполнения из-за постоянного прихода новых процессов с более высокими приоритетами.) В некоторых системах бесконечное откладывание предотвращается потому, что приоритет процесса увеличивается по мере того, как он ожидает выделения требуемого ему ресурса. Это называется *старением процесса*. В конце концов, ожидающий процесс получить более высокое значение приоритета, чем у всех остальных и будет исполнен.

5.1 Концепции ресурсов

ОС выполняет по преимуществу функции администратора ресурсов. Она отвечает за распределение обширных наборов ресурсов различного типа. Рассмотрим ресурсы, которые являются «оперативно перераспределяемыми» (*preemtible*), такие как центральный процессор и основная память.

Программа, занимающая в данный момент некоторую область основной памяти, может быть вытеснена. Программа, выдавшая запрос

ввода-вывода, практически не может работать с основной памятью в процессе длительного периода ожидания завершения данной операции ввода-вывода. Центральный процессор является самым динамичным ресурсом.

Если конкретный процесс достигает точки, когда он не может эффективно использовать центральный процессор, право управления центральным процессором отбирается у этого процесса и передается другому.

Организация динамического переключения ресурсов является исключительно важным фактором для обеспечения эффективной работы мультипрограммных вычислительных систем.

Определенные виды ресурсов являются оперативно неперераспределяемыми (nonpreemtible), в том смысле что их нельзя произвольно отбирать у процессов, за которыми они закреплены. В качестве примера можно провести лентопротяжные устройства.

Некоторые виды ресурсов допускают разделение между несколькими процессами, а некоторые предусматривают только монопольное пользование индивидуальными процессами.

Данные и программы – это тоже ресурсы, требующие соответственной организации и распределения. Так, в мультипрограммных системах, многим пользователям требуется одновременная работа с редакторами. Однако, иметь собственную копию редактора в основной памяти для каждого пользователя было бы неэкономично. Поэтому в па-

мять записывается одно копия программы и несколько копий соответствующих данных для каждого пользователя. Код программы, которые не может изменяться во время выполнения, называется *реентерабильным* или *многократно - входимым*. Код, который может изменяться, но предусматривает повторную инициализацию при каждом выполнении, называется кодом *последовательного (многократного) использования*. С реентерабельным кодом могут коллективно работать несколько процессов одновременно, а с кодом последовательного исполнения – только один процесс в каждый конкретный момент времени.

5.2 Четыре необходимых условия возникновения тупика

Коффман, Элфик и Шошани сформулировали следующие четыре необходимых условия наличия тупика:

1. Процессы требуют предоставления им права монопольного управления ресурсами, которые им выделяются (*условие взаимоисключения*).
2. Процессы удерживают за собой ресурсы, уже выделенные им, ожидая в то же время выделения дополнительных ресурсов (*условие ожидания ресурсов*).
3. Ресурсы нельзя отобрать у процессов, удерживающих их, пока эти ресурсы не будут использованы для завершения работы (*условие неперераспределяемости*).
4. Существует кольцевая цепь процессов, в которой каждый процесс удерживает за собой один или более ресурсов, требующихся следующему процессу цепи (*условие кругового ожидания*).

На сегодняшний день существуют следующие четыре направления по исключению тупиков:

- Предотвращение тупиков;
- Обход тупиков;

- Обнаружение тупиков;

- Восстановление после тупиков.

5.3 Предотвращение тупиков

Хавендер показал, что возникновение тупика невозможно, если нарушено хотя бы одно из указанных выше четырех необходимых условий. Он предложил следующую стратегию:

- Каждый процесс должен запрашивать все требуемые ему ресурсы сразу, причем не может начинать выполняться до тех пор, пока все они не будут ему предоставлены.

- Если процесс, удерживающий определенные ресурсы, получает отказ в удовлетворении запроса на дополнительные ресурсы, этот процесс должен освободить свои первоначальные ресурсы и при необходимости запросить их снова вместе с дополнительными.

- Введение линейной упорядоченности по типам ресурсов для всех процессов; другими словами, если процессу выделены ресурсы данного типа, в дальнейшем он может запросить только ресурсы более далеких по порядку типов.

5.4 Предотвращение тупиков и алгоритм банкира

Дейкстра предложил алгоритм банкира, который получил свое название потому, что он как бы имитирует действия банкира, который располагая неким источником капитала, выдает ссуды и принимает платежи.

Когда при описании алгоритма банкира мы будем говорить о ресурсах, мы будем подразумевать ресурсы одного и того же типа, однако этот алгоритм можно легко распространить на пулы ресурсов нескольких различных типов.

Текущее состояние вычислительной машины называется *надежным*, если ОС может обеспечить всем текущим пользователям завершение их заданий в течении конечного времени. В противном случае текущее состояние системы называется *ненадежным*.

Алгоритм банкира говорит о том, что выделять ресурсы процессам можно только в том случае, когда после очередного выделения устройств состояние системы остается надежным.

Если известно, что текущее состояние является надежным, это вовсе не значит, что все последующие состояния также будут надежными.

Предположим, что в системе имеется 12 одинаковых ресурсов, распределеемых между тремя процессами, как показано в **Таблица 1**.

Это состояние «надежное», поскольку оно дает возможность всем трем процессам закончить работу. В распоряжении ОС имеется 12 ресурсов, из которых в данный момент 10 в работе, а 2 в резерве. Если

этих 2 резервных ресурса выделить процессу 2, удовлетворяя тем самым его требованиям, что даст ему возможность завершиться и освободить 6 своих ресурсов, что позволит системе выполнить процессы 1 и 3. Таким образом, ***основной критерий надежного состояния - это существование последовательности действий, позволяющей всем пользователям закончить работу***.

Таблица 1 Надежное состояние

	Текущее количество выделенных ресурсов		Максимальная потребность
Процесс 1	1		4
Процесс 2	4		6
Процесс 3	5		8
Резерв		2	

Таблица 2 Ненадежное состояние

	Текущее количество выделенных ресурсов		Максимальная потребность
Процесс 1	8		10
Процесс 2	2		5
Процесс 3	1		3
Резерв		1	

Таблица 2 иллюстрирует другое состояние системы. Здесь из 12 ресурсов, занято 11 и 1 ресурс находится в резерве. В подобной ситуации, независимо от того, какой процесс запросит этот резервный ресурс, мы не можем гарантировать, что все 3 пользователя закончат работу. Здесь важно отметить, что термин ***«Ненадежное состояние» не предполагает, что в данный момент существует или в какое-то время обязательно возникнет тупиковая ситуация. Он просто говорит о том, что в случае некоторой неблагоприятной последовательности событий система может зайти в тупик.***

Если известно, что данное состояние надежно, это вовсе не означает, что все последующие состояния также будут надежными. Рассмотрим например случай, который показывает **Таблица 3**

Таблица 3 Состояние 1

	Текущее количество выделенных ресурсов		Максимальная потребность
Процесс 1	1		4
Процесс 2	4		6
Процесс 3	5		8
Резерв		2	

Предположим теперь, что процесс 3 запрашивает дополнительный ресурс. Если бы система удовлетворила бы его, она бы перешло в состояние, которое показывает **Таблица 4**.

Таблица 4 Состояние 2

	Текущее количество выделенных ресурсов		Максимальная потребность
Процесс 1	1		4
Процесс 2	4		6
Процесс 3	6		8
Резерв		1	

Конечно, это состояние не обязательно приведет к тупиковой ситуации. Однако, если состояние 1 было надежным, то состояние 2 стало ненадежным. Состояние 2 характеризует систему, в которой нельзя гарантировать успешное завершение всех процессов. В резерве здесь остается только один ресурс, в то время как в наличии должно быть как минимум два, что бы либо процесс 1, либо процесс 3 могли завершить свою работу и вернуть занимаемые ими ресурсы ОС, тем самым позволив выполнится процессу 2.

Сейчас уже должно быть ясно, каким образом осуществляется распределение ресурсов по алгоритму банкира. Условия «взаимоисключения», «ожидания дополнительных ресурсов» и «неперераспределяемости» выполняются. Процессы могут претендовать на монопольное ис-

пользование ресурсов, которые им потребуются. Процессам реально разрешено удерживать за собой ресурсы, причем ресурсы нельзя отбирать у процесса, которому они выделены. Пользователи не ставят перед системой особенно сложных задач, запрашивая в каждый момент времени только один ресурс. Система может либо удовлетворить, либо отклонить каждый запрос. Если запрос отклоняется, процесс удерживает за собой уже выделенные ему ресурсы и ждет определенный, конечный период времени, пока этот запрос в конце концов не будет удовлетворен. *Система удовлетворяет только те запросы, при которых ее состояние остается надежным.* Запрос пользователя, приводящий к переходу системы в ненадежное состояние, откладывается до тех пор, пока его все же можно будет выполнить. Таким образом, поскольку система всегда поддерживается в надежном состоянии, рано или поздно (но в течении конечного времени) все запросы будут удовлетворены и все процессы смогут завершить свою работу. Тем не менее, алгоритм банкира имеет следующие недостатки:

- Алгоритм банкира исходит из фиксированного количества распределяемых ресурсов. Поскольку устройства, предоставляющие ресурсы, нуждаются в профилактике, либо ремонте, мы не можем считать, что количество ресурсов всегда остается фиксированным.
- Алгоритм требует, что бы число работающих процессов оставалось постоянным. Подобное требование является нереалистичным, так

как в современных ОС число процессов может меняться все время.

- Алгоритм требует, чтобы банкир –распределитель ресурсов гарантированно удовлетворил все запросы за некий конечный период времени. Очевидно, что для реальных систем необходимы более конкретные гарантии.

- Алгоритм требует, чтобы процессы гарантированно «платили долги». (т.е. возвращали выделенные им ресурсы) в течении некоторого конечного времени.

- Алгоритм требует, что бы процессы заранее указывали свои максимальные потребности в ресурсах. По мере того, как распределение ресурсов становиться все более динамичным, все труднее оценивать максимальные потребности процесса.

5.5 Обнаружение тупиков

Обнаружение тупика – это установление факта, что возникла тупиковая ситуация, и определение процессов и ресурсов, вовлеченных в тупиковую ситуацию. При рассмотрении этой задачи, применяется весьма распространенная нотация , согласно которой распределение ресурсов и запросы изображаются в виде направленного графа. Квадраты означают процессы, а большие круги – классы идентичных ресурсов.

Малые кружки, находящиеся внутри больших, обозначают количество идентичных ресурсов каждого класса. На рис. 5.2 показаны отношения, которые могут представлены на графе запросов и распределения ресурсов. На рис. 5.2а) показано, что процесс P1 запрашивает ресурс типа R1. Стрелка от квадрата P1 доходит только до большого круга. Это говорит о том, что в настоящий момент, запрос на выделение данного ресурса находиться в рассмотрении. На рис. 5.2б) показано, что процессу P2 выделен ресурс типа R2 (один из двух идентичных ресурсов). Рис. 5.2в) демонстрирует ситуацию, приближающуюся к тупику: процесс P3 запрашивает ресурс R3 выделенный процессу R4. И наконец рис. 5.2г) демонстрирует тупиковую ситуацию, когда P5 запрашивает R4, который занят процессом P6, который запрашивает ресурс R5, выделенный процессу P5. Это пример «кругового ожидания».

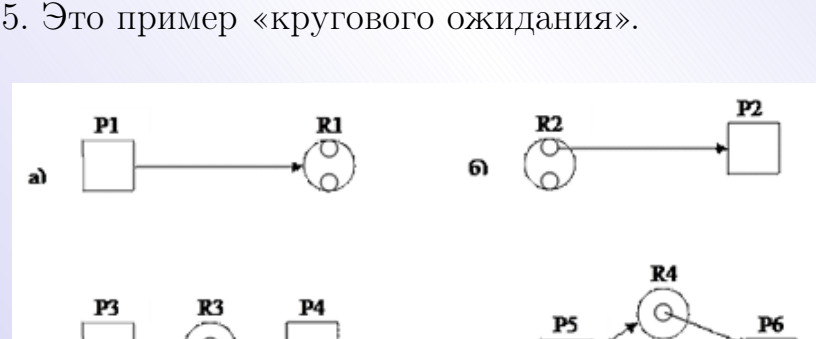


Рис. 5.2: Граф запросов и распределения ресурсов

Графы запросов и распределения ресурсов динамично меняются по мере того, как процессы запрашивают ресурсы, получают их в свое распоряжение, а через какое-то время возвращают ОС.

Задача механизма обнаружения тупиков – определить, не возникла ли в системе тупиковая ситуация. Одним из таких способов является приведение, или редукция графа, - это позволяет определить процессы, которые могут завершиться, и процессы, которые будут оставаться в тупиковой ситуации.

Если запросы ресурсов для некоторого процесса могут быть удовлетворены, то мы говорим, что *граф можно редуцировать на этот процесс*. Такая редукция эквивалентна изображению графа в том виде, который он будет иметь в случае, если данный процесс завершить свою работу и возвратит ресурсы системе. Редукция графа на конкретный процесс изображается исключением стрелок, идущих к этому процессу от ресурсов и стрелок к ресурсам от этого процесса. **Если граф можно редуцировать на все процессы, значит, тупиковой ситуации нет, а если этого сделать нельзя, то все «нередуцированные» процессы образуют набор процессов, вовлеченных в тупиковую ситуацию.**

Важно отметить, что порядок, в котором осуществляется редукция графа, не имеет значения: окончательный результат в любом случае будет тем же самым.

5.6 Восстановление после тупиков

Систему, оказавшуюся в тупике, необходимо вывести из него, нарушив одно или несколько необходимых условий его существования. При этом, как правило, несколько процессов потеряют частично или уже полностью проделанную ими работу. Однако это обойдется гораздо дешевле, чем оставлять систему в тупиковой ситуации. Сложность восстановления системы обуславливается рядом факторов:

- Прежде всего в первый момент вообще может быть неочевидно, что система попала в тупиковую ситуацию.
- В большинстве систем нет достаточно эффективных средств, позволяющих приостановить процесс на неопределенно долгое время, вывести его из системы и возобновить его выполнение впоследствии. В действительности некоторые процессы, например процессы реального времени, должны работать непрерывно и не допускают приостоновки и последующего возобновления.
- Если даже в системе существуют эффективные средства остановки/возобновления процессов, то их использование требует, как правило, значительных затрат машинного времени, а также внимания высококвалифицированного оператора. А подобного оператора может и не быть.

- Восстановление после тупика небольших масштабов требует обычно и небольшого количества работы; крупный же тупик (с участием десятков или даже сотен процессов) может потребовать громадной работы.

ГЛАВА 6
Управление памятью

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: отслеживание свободной и занятой памяти, выделение памяти процессам и освобождение памяти при завершении процессов, 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, а также 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 (Рис. 6.1).

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

Виртуальные адреса вырабатывает транслятор, переводящий про-

грамму 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.

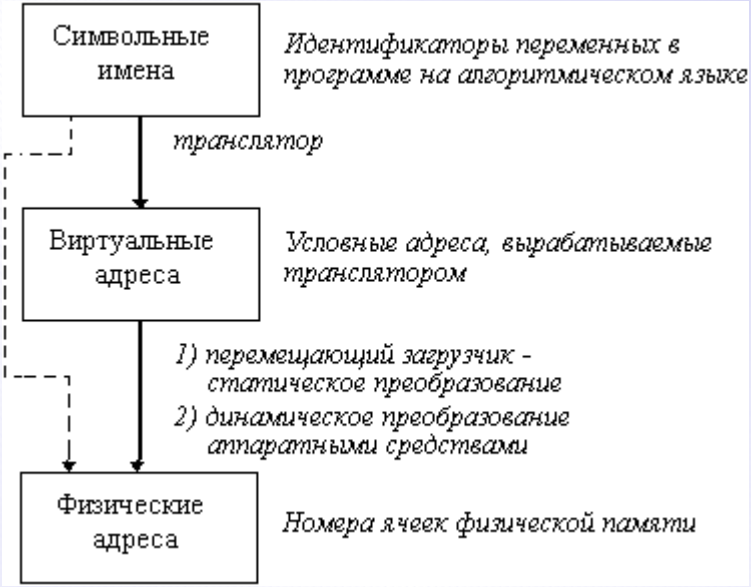


Рис. 6.1: Типы адресов

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, предо-

ставленной 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 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 без использования дискового пространства

Все методы управления памятью могут быть разделены на два класса: методы, которые используют перемещение процессов между оперативной памятью и диском, и методы, которые не делают этого (Рис. 6.2). Начнем с последнего, более простого класса методов.

Рис. 6.2: Классификация методов распределения памяти

6.2.1 Распределение памяти фиксированными разделами

Самым простым способом управления оперативной памятью является разделение ее на несколько разделов фиксированной величины. Это может быть выполнено вручную оператором во время старта системы или во время ее генерации. Очередная задача, поступившая на выполнение, помещается либо в общую очередь (Рис. 6.3а), либо в очередь к некоторому разделу (Рис. 6.3б).

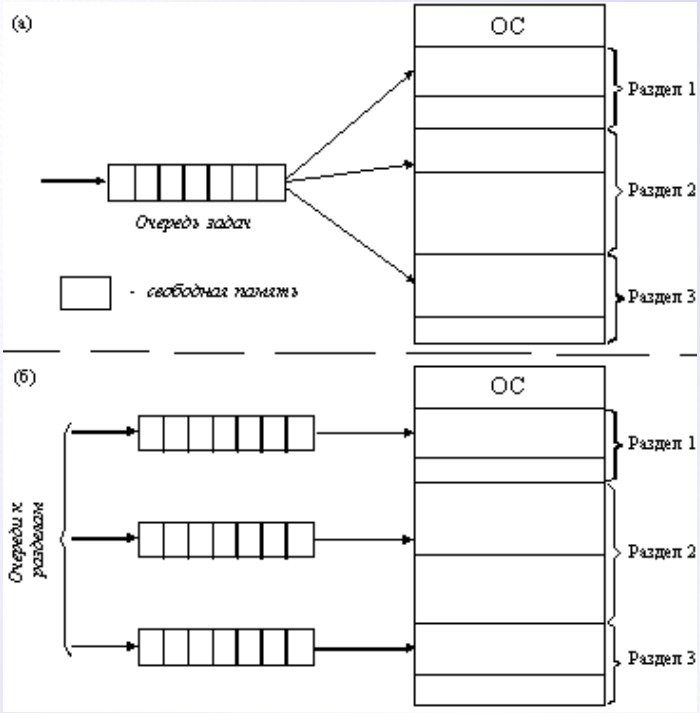


Рис. 6.3: Распределение памяти фиксированными разделами: а - с общей очередью; б - с отдельными очередями

Подсистема управления памятью в этом случае выполняет следующие задачи:

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

При очевидном преимуществе - простоте реализации - данный метод имеет существенный недостаток - жесткость. Так как в каждом разделе может выполняться только одна программа, то 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. С другой стороны, даже если объем оперативной памяти машины позволяет выполнить некоторую программу, разбиение памяти на разделы не позволяет сделать этого.

6.2.2 Распределение памяти разделами переменной величины

В этом случае память машины не делится заранее на разделы. Сначала вся память свободна. Каждой вновь поступающей задаче выделяется необходимая ей память. Если достаточный объем памяти отсутствует, то задача не принимается на выполнение и стоит в очереди. После завер-

шения 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 рис. 6.4 показано состояние памяти в различные моменты времени при использовании динамического распределения. Так в момент t0 в памяти находится только ОС, а к моменту t1 память разделена между 5 задачами, причем задача П4, завершаясь, покидает память. На освободившееся после задачи П4 место загружается задача П6, поступившая в момент t3.

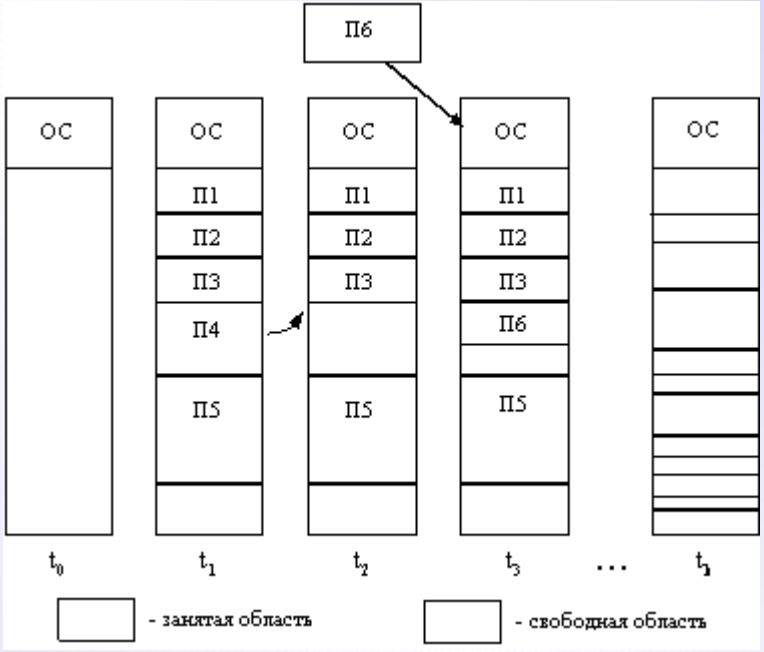


Рис. 6.4: Распределение памяти фиксированными разделами

Задачами операционной системы при реализации данного метода управления памятью является:

- ведение таблиц свободных и занятых областей, в которых указываются начальные адреса и размеры участков памяти
- при поступлении новой задачи - анализ запроса, просмотр таблицы свободных областей и выбор раздела, размер которого достаточен

для размещения поступившей задачи

- загрузка задачи в выделенный ей раздел и корректировка таблиц свободных и занятых областей,
- после завершения задачи корректировка таблиц свободных и занятых областей.

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

Выбор раздела для вновь поступившей задачи может осуществляться по разным правилам, таким, например, как "первый попавшийся раздел достаточного размера или "раздел, имеющий наименьший достаточный размер или "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.

6.2.3 Перемещаемые разделы

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 (Рис. 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 либо при каждом завершении задачи, либо 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 должно выполняться динамическим способом.

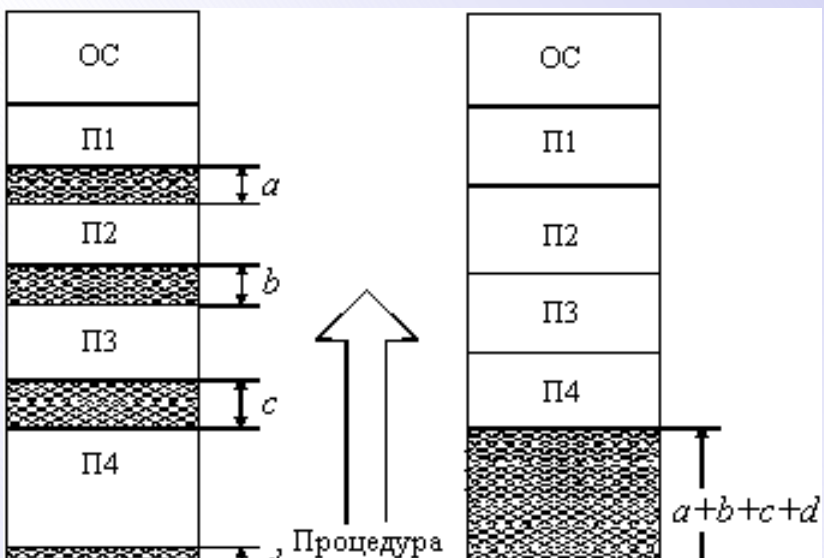


Рис. 6.5: Распределение памяти перемещаемыми разделами

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

6.3 Методы распределения памяти с использованием дискового пространства

6.3.1 Понятие виртуальной памяти

Уже достаточно давно пользователи столкнулись с проблемой размещения в памяти программ, размер которых превышал имеющуюся в наличии свободную память. 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. Так, 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.

Таким образом, виртуальная память - это совокупность программно-аппаратных средств, позволяющих пользователям писать программы, размер которых превосходит имеющуюся оперативную память; для этого виртуальная память решает следующие задачи:

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

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

Наиболее распространенными реализациями виртуальной памяти является страничное, сегментное и странично-сегментное распределение памяти, а также свопинг.

6.3.2 Страничное распределение

На рис. 6.6 показана схема страничного распределения памяти. Виртуальное адресное пространство каждого процесса делится на части одинакового, фиксированного для данной системы размера, называемые виртуальными страницами. В общем случае размер виртуального адресного пространства не является кратным размеру страницы, поэтому последняя страница каждого процесса дополняется фиктивной областью.

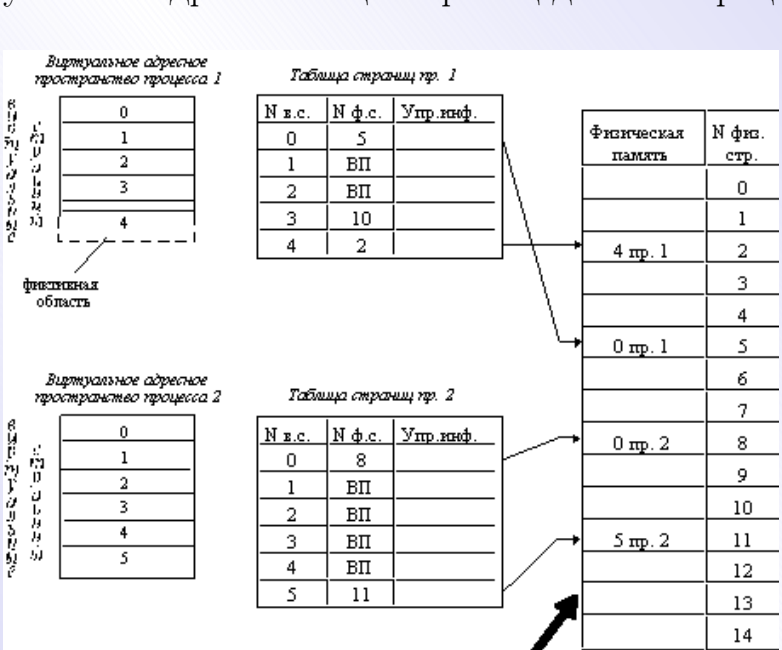
Вся оперативная память машины также делится на части такого же размера, называемые физическими страницами (или блоками).

Размер страницы обычно выбирается равным степени двойки: 512, 1024 и т.д., это позволяет упростить механизм преобразования адресов.

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. 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 признак модификации страницы, признак невыгружаемости (выгрузка некоторых страниц может быть запрещена), признак обращения к странице

(используется для подсчета числа обращений за определенный период времени) и другие данные, формируемые и используемые механизмом виртуальной памяти.

При активизации очередного процесса в специальный регистр процессора загружается адрес таблицы страниц данного процесса.



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

Рис. 6.6: Страничное распределение памяти

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. Если же нужная виртуальная страница в данный момент выгружена на диск, то происходит так называемое 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 свободных 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:

- дольше всего не использовавшаяся страница,
- первая попавшаяся страница,
- 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 пред-

ставляет 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 (из таблицы страниц). Если выталкиваемая страница с момента загрузки была модифицирована, то ее новая версия должна быть переписана на диск. Если нет, то она может быть просто уничтожена, то есть соответствующая физическая страница объявляется свободной.

Рассмотрим механизм преобразования виртуального адреса в физический при страничной организации памяти (Рис. 6.7).

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. Учитывая, что размер страницы равен 2 в степени k, 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.

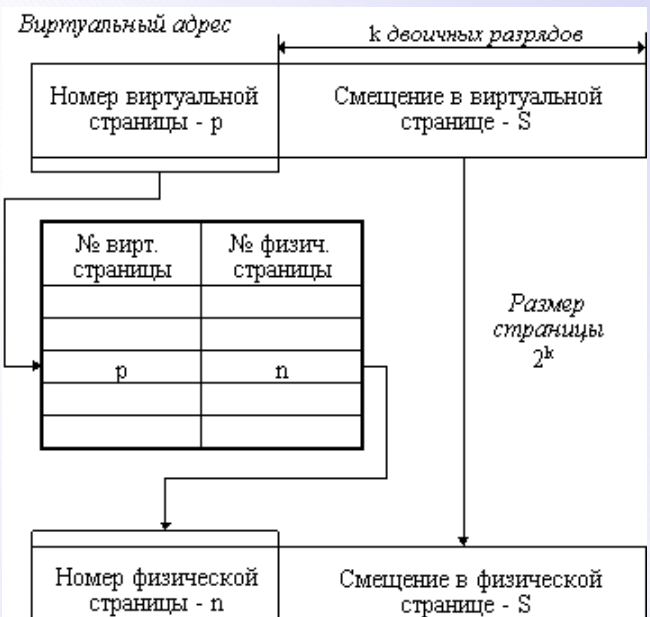


Рис. 6.7: Механизм преобразования виртуального адреса в физический при страничной организации памяти

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 временные затраты, связанные с обработкой страничных прерываний и преобразованием виртуального адреса в физический. 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. Время преобразования виртуального адреса в физический в значительной степени определяется временем доступа к таблице страниц. В связи с этим таблицу страниц стремятся размещать в "быстрых"запоминающих устройствах. Это может быть, например, 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.

6.3.3 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 бывает очень полезным. Например, можно запретить обращаться с операциями записи и чтения в кодовый сегмент про-

граммы, 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.

Рассмотрим, каким образом сегментное распределение памяти реализует эти возможности (Рис. 6.8). 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.

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. Во время загрузки система создает таблицу сегментов процесса (аналогичную таблице страниц), в которой для каждого сегмента указывается начальный физический адрес сегмента в оперативной памяти, размер сегмента, правила доступа, признак модификации, признак обращения к данному сегменту за последний интервал времени и некоторая другая информация. Если

виртуальные адресные пространства нескольких процессов включают один и тот же сегмент, то в таблицах сегментов этих процессов делаются ссылки на один и тот же участок оперативной памяти, в который данный сегмент загружается в единственном экземпляре.

Система с сегментной организацией функционирует аналогично системе со страничной организацией: время от времени происходят прерывания, связанные с отсутствием нужных сегментов в памяти, при необходимости освобождения памяти некоторые сегменты выгружаются, при каждом обращении к оперативной памяти выполняется преобразование виртуального адреса в физический. 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.

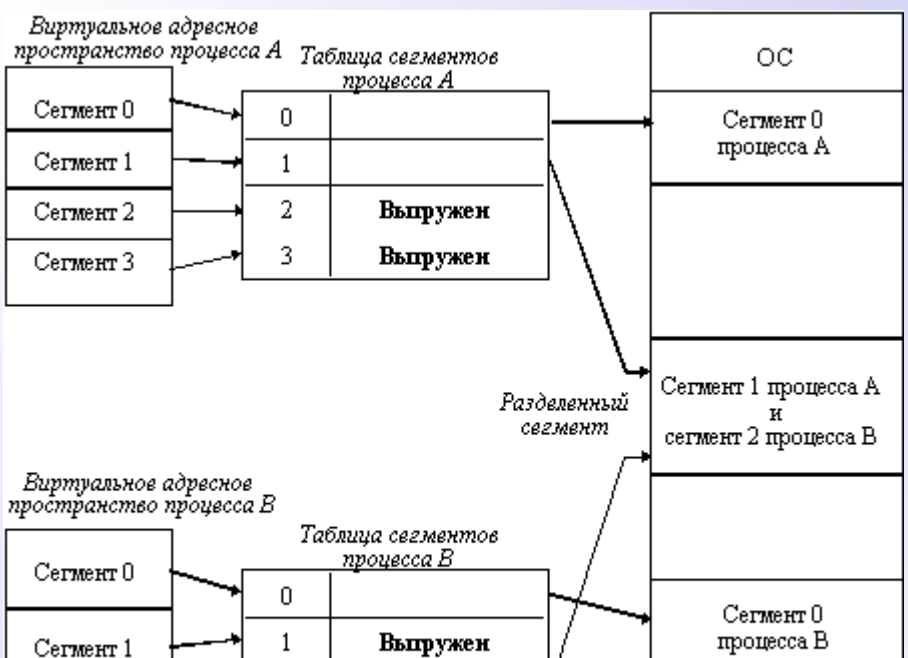


Рис. 6.8: Распределение памяти сегментами

Недостатком данного метода распределения памяти является фрагментация на уровне сегментов и более медленное по сравнению со страничной организацией преобразование адреса.

6.3.4 Странично-сегментное распределение

Как видно из названия, данный метод представляет собой комбинацию страничного и сегментного распределения памяти и, вследствие этого, сочетает в себе достоинства обоих подходов. Виртуальное пространство процесса делится на сегменты, а каждый сегмент в свою очередь делится на виртуальные страницы, которые нумеруются в пределах сегмента. Оперативная память делится на физические страницы. Загрузка процесса выполняется операционной системой постранично, при этом часть страниц размещается в оперативной памяти, а часть на диске. Для каждого сегмента создается своя таблица страниц, структура которой полностью совпадает со структурой таблицы страниц, используемой при страничном распределении. Для каждого процесса создается таблица сегментов, в которой указываются адреса таблиц страниц для всех сегментов данного процесса. Адрес таблицы сегментов загружается в специальный регистр процессора, когда активизируется соответствующий процесс. На рис. 6.9 показана схема преобразования виртуального адреса в физический для данного метода.

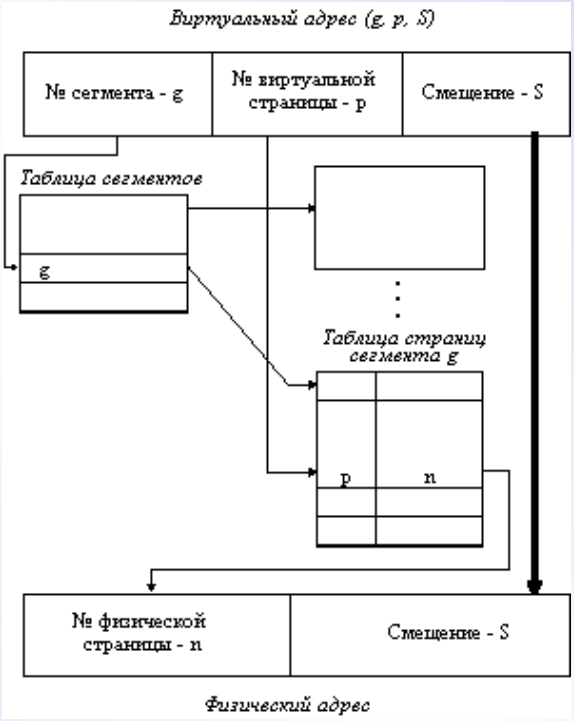


Рис. 6.9: Схема преобразования виртуального адреса в физический для сегментно-страничной организации памяти

6.3.5 Свопинг

Разновидностью виртуальной памяти является свопинг.

На рис. 6.10 показан график зависимости коэффициента загрузки процессора в зависимости от числа одновременно выполняемых процессов и доли времени проводимого этими процессами в состоянии ожидания ввода-вывода.

Из рисунка видно, что для загрузки процессора на 90% достаточно всего трех счетных задач. Однако для того, чтобы обеспечить такую же загрузку интерактивными задачами, выполняющими 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.

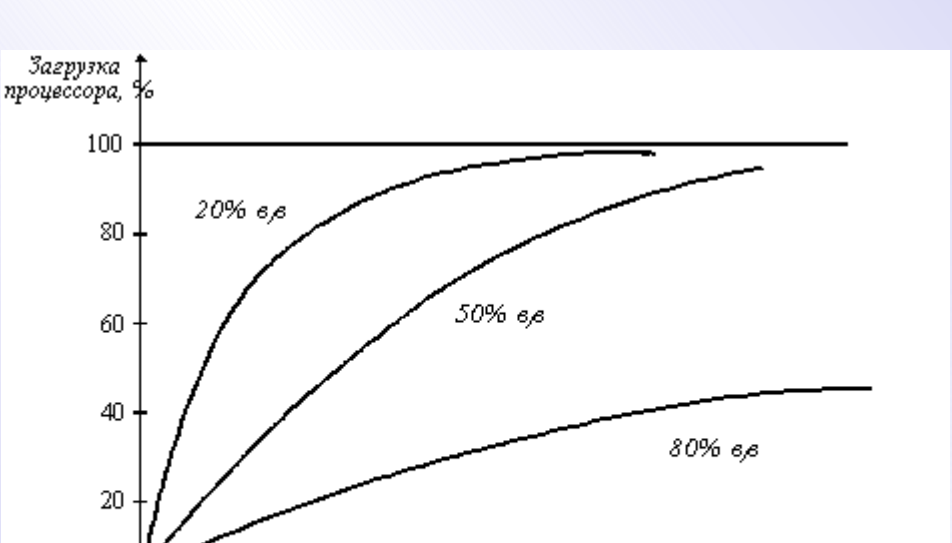


Рис. 6.10: Зависимость загрузки процессора от числа задач и интенсивности ввода-вывода

6.4 Иерархия запоминающих устройств. Принцип кэширования данных

Память вычислительной машины представляет собой иерархию запоминающих устройств (внутренние регистры процессора, различные типы сверхоперативной и оперативной памяти, диски, ленты), отличающихся средним временем доступа и стоимостью хранения данных в расчете на один бит (Рис. 6.11). Пользователю хотелось бы иметь и недорогую и быструю память. Кэш-память представляет некоторое компромиссное решение этой проблемы.

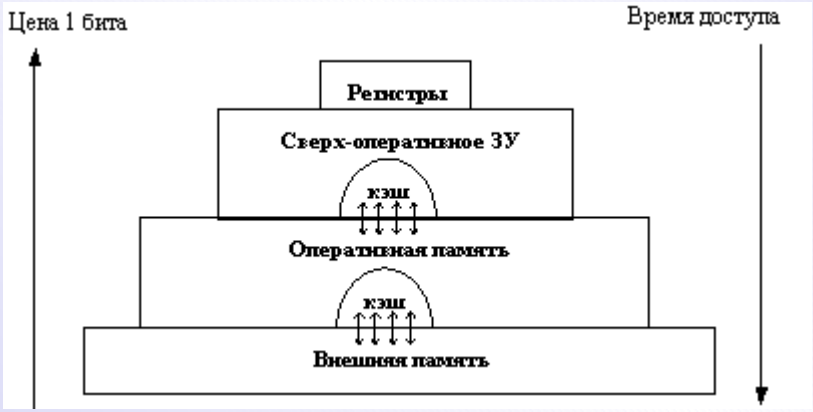


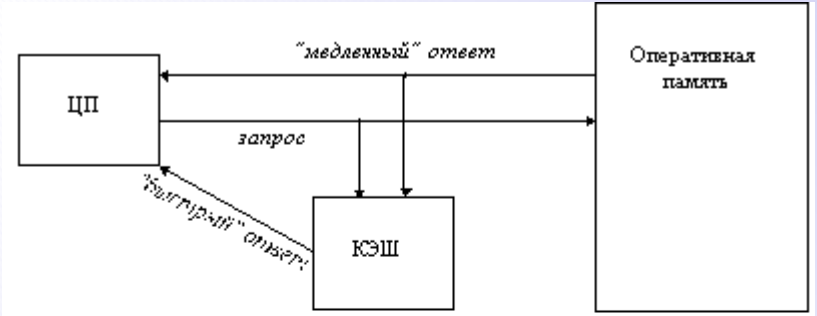
Рис. 6.11: Иерархия ЗУ

Кэш-память - это способ организации совместного функционирования двух типов запоминающих устройств, отличающихся временем до-

ступа 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 (Рис. 6.12). В качестве такового может быть использована, например, ассоциативная память. Содержимое кэш-памяти представляет собой совокупность записей обо всех загруженных в нее элементах данных. Каждая запись об элементе данных включает в себя адрес, который этот элемент данных имеет в оперативной памяти, и управляющую информацию: признак модификации и признак обращения к данным за некоторый последний период времени. В системах, оснащенных кэш-памятью, каждый запрос к опе-



Структура кэш-памяти

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

Рис. 6.12: Кэш-память

ративной 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 взято-

му 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. 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. Если вытесняемые данные были модифицированы за время нахождения в кэш-памяти, то они переписываются в оперативную память. Если же эти данные не были модифицированы, 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 то есть нахождения 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 t_1 и кэш-память, имеющая время доступа t_2, очевидно, что $t_2 < t_1$. 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:

$$t = t_1(1 - p) + t_2(p)$$

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.

Пространственная локальность. Если произошло обращение по некоторому адресу, то с высокой степенью вероятности в ближайшее время произойдет обращение к соседним адресам.

Временная локальность. Если произошло обращение по некоторому адресу, то следующее обращение по этому же адресу с большой вероятностью произойдет в ближайшее время.

Все предыдущие рассуждения справедливы и для других пар запоминающих устройств, например, для оперативной памяти и внешней памяти. В этом случае уменьшается среднее время доступа к данным, расположенным на диске, и роль кэш-памяти выполняет буфер в оперативной памяти.

ГЛАВА 7
Файловая система

Файловая система - это часть операционной системы, назначение которой состоит в том, чтобы обеспечить пользователю удобный интерфейс при работе с данными, хранящимися на диске, и обеспечить совместное использование файлов несколькими пользователями и процессами.

В широком смысле понятие "файловая система"включает:

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

7.1 Имена файлов

Файлы идентифицируются именами. Пользователи дают файлам символьные имена, при этом учитываются ограничения ОС как на используемые символы, так и на длину имени. До недавнего времени эти гра-

ницы были весьма узкими. Так в популярной файловой системе FAT длина имен ограничивается известной схемой 8.3 (8 символов - собственно имя, 3 символа - расширение имени), а в ОС UNIX System V имя не может содержать более 14 символов. Однако 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, как правило, поддерживают длинные символьные имена файлов. Например, Windows NT в своей новой 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. Таким образом, одной из важных задач становится проблема генерации соответствующих коротких имен. Длинные имена поддерживаются не только новыми файловыми системами, но и новыми версиями хорошо известных файловых систем. Например, в ОС Windows 95 используется файловая система VFAT, представляющая собой существенно измененный вариант FAT. Среди многих других усовершенство-

ваний одним из главных достоинств VFAT является поддержка длинных имен. Кроме проблемы генерации эквивалентных коротких имен, при реализации нового варианта 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 взаимно-однозначное 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.

7.2 Типы файлов

Файлы бывают разных типов: обычные файлы, специальные файлы, файлы-каталоги.

Обычные файлы в свою очередь подразделяются на текстовые и дво-

ичные. Текстовые файлы состоят из строк символов, представленных в ASCII-коде. Это могут быть документы, исходные тексты программ и т.п. Текстовые файлы можно прочитать на экране и распечатать на принтере. Двоичные файлы не используют ASCII-коды, они 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 исходя из некоторых соображений (например, файлы, содержащие программы игр, или файлы, составляющие один программный пакет), а 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 атри-

бутов разные характеристики, например:

- информация о разрешенном доступе
- пароль для доступа к файлу
- владелец файла
- создатель файла
- признак "только для чтения"
- признак "скрытый файл"
- признак "системный файл"
- признак "архивный файл"
- признак "двоичный/символьный"
- признак "временный"(удалить после завершения процесса)
- признак блокировки
- длина записи
- указатель на ключевое поле в записи
- длина ключа

- времена создания, последнего доступа и последнего изменения
- текущий размер файла
- 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 (Рис. 7.1). Каталоги могут образовывать иерархическую структуру за счет того, что каталог более низкого уровня может входить в каталог более высокого уровня (Рис. 7.2).

Рис. 7.1: Структура каталогов: а - структура записи каталога MS-DOS (32 байта); б - структура записи каталога ОС UNIX

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'е - 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.

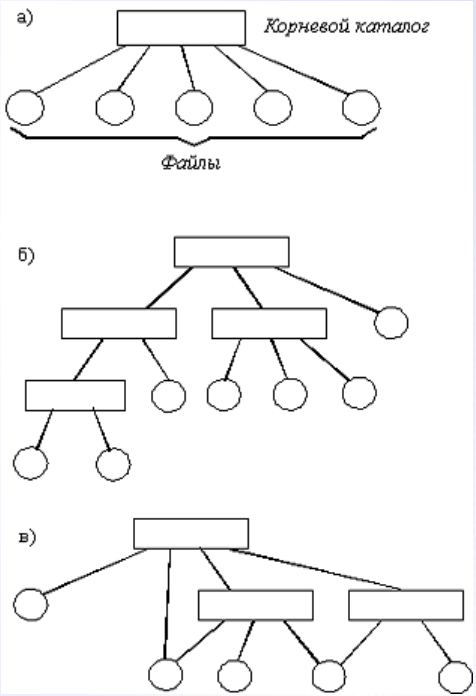


Рис. 7.2: Логическая организация файловой системы а - одноуровневая; б - иерархическая (дерево); в - иерархическая (сеть)

7.3 Логическая организация файла

Программист имеет дело с логической организацией файла, представляя файл в виде определенным образом организованных логических за-

писей. 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 Рис. 7.3 показаны несколько схем логической организации файла. Записи могут быть фиксированной длины или переменной длины. Записи могут быть расположены в файле последовательно (последовательная организация) или 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.

Рис. 7.3: Способы логической организации файлов

7.4 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 дан-

ных, которой внешнее устройство обменивается с оперативной памятью. Непрерывное размещение – простейший вариант физической организации (Рис. 7.4а), при котором файлу предоставляется последовательность блоков диска, образующих единый сплошной участок дисковой памяти. Для задания адреса файла в этом случае достаточно указать только номер начального блока. Другое достоинство этого метода – простота. Но имеются и два существенных недостатка. Во-первых, во время создания файла заранее не известна его длина, а значит не известно, сколько памяти надо зарезервировать для этого файла, во-вторых, 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 используемыми.

Следующий способ физической организации - размещение в виде связанного списка блоков дисковой памяти (Рис. 7.4б). При таком способе в начале каждого блока содержится указатель на следующий блок. В этом случае адрес файла также может быть задан одним числом - номером первого блока. В отличие от предыдущего способа, каждый блок может быть присоединен в цепочку какого-либо файла, следовательно фрагментация отсутствует. Файл может изменяться во время своего существования, наращивая число блоков. Недостатком является сложность реализации доступа к произвольно заданному месту файла: для того, чтобы прочитать пятый по порядку блок файла, необходимо последова-

тельно прочитать четыре первых блока, прослеживая цепочку номеров блоков. Кроме того, при этом способе 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. Популярным способом,

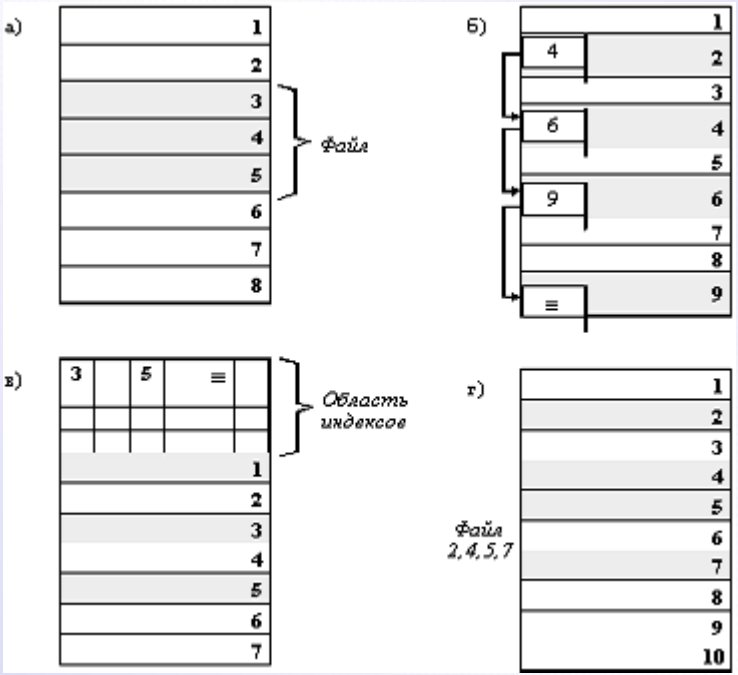


Рис. 7.4: Физическая организация файла а - непрерывное размещение; б - связанный список блоков; в - связанный список индексов; г - перечень номеров блоков

используемым, например, в файловой системе FAT операционной системы MS-DOS, является использование связанного списка индексов. 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. При такой физической организации сохраняются все достоинства предыдущего способа, но снимаются оба отмеченных недостатка: во-первых, для доступа к произвольному месту файла достаточно прочитать только блок индексов, отсчитать нужное количество блоков файла по цепочке и определить номер нужного блока, и, во-вторых, данные файла занимают блок целиком, а значит имеют объем, равный степени двойки.

В заключение рассмотрим задание физического расположения файла путем простого перечисления номеров блоков, занимаемых этим файлом. ОС UNIX использует вариант данного способа, позволяющий обеспечить фиксированную длину адреса, независимо от размера файла. Для хранения адреса файла выделено 13 полей. Если размер файла меньше или равен 10 блокам, то номера этих блоков непосредственно перечислены в первых десяти полях адреса. Если размер файла больше 10 блоков, то следующее 11-е поле содержит адрес блока, в котором могут быть расположены еще 128 номеров следующих блоков файла. Если файл больше, чем 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. И, наконец, если файл больше $10 + 128 + 128^{128}$, то используется последнее 13-е поле для тройной косвенной адресации, что позволяет задать адрес файла, имеющего размер максимум $10 + 128 + 128^{128} + 128^{128^{128}}$.

7.5 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:

- создание файла,
- уничтожение файла,
- открытие файла,
- закрытие файла,
- чтение файла,
- запись в файл,
- дополнение файла,

- поиск в файле,
- получение атрибутов файла,
- установление новых значений атрибутов,
- переименование,
- выполнение файла,
- чтение каталога,
- и 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 (Рис. 7.5). 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.

Рис. 7.5: Матрица прав доступа

Различают два основных подхода к определению прав доступа:

- избирательный доступ, когда для каждого файла и каждого пользователя сам владелец может определить допустимые операции;
- 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.

7.6 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 буферный пул, располагающийся в оперативной памяти, и комплекс программ, управляющих этим пулом. Каждый буфер пула имеет размер, равный одному блоку. 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 доступа к файлу. Если же нужный блок в буферном пуле отсутствует, то он считывается с устройства и одновременно с передачей запрашивающему процессу копируется в один из буферов подсистемы буферизации. При отсутствии свободного буфера на диск вытесняется наименее используемая информация. Таким образом, подсистема буферизации работает по принципу кэш-памяти

7.7 Общая модель файловой системы

Функционирование любой файловой системы можно представить многоуровневой моделью (Рис. 7.6), в которой 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.

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).

На следующем, базовом уровне по уникальному имени файла определяются его характеристики: права доступа, адрес, размер и другие. Как уже было сказано, характеристики файла могут входить в состав каталога или храниться в отдельных таблицах. При открытии файла его характеристики перемещаются с диска в оперативную память, что-

бы уменьшить среднее время доступа к файлу. В некоторых файловых системах (например, HPFS) при открытии файла вместе с его характеристикам в оперативную память перемещаются несколько первых блоков файла, содержащих данные. Следующим этапом реализации запроса к

Рис. 7.6: Общая модель файловой системы

файлу является 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.

На логическом уровне определяются координаты запрашиваемой логической записи в файле, то есть требуется определить, 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.

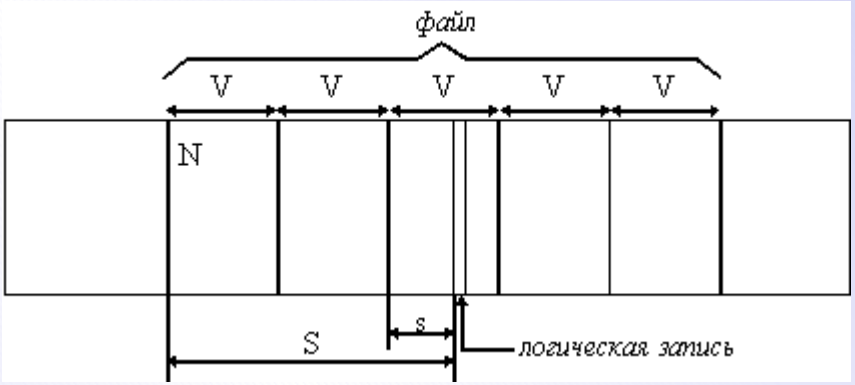


Рис. 7.7: Функции физического уровня файловой системы

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

$n = N + \left[\frac{S}{V}\right]$, где $\left[\frac{S}{V}\right]$ - целая часть числа $\frac{S}{V}$

$s = R \cdot \left[\frac{S}{V}\right]$ - дробная часть числа $\frac{S}{V}$

На физическом уровне файловая система определяет номер физического блока, который содержит требуемую логическую запись, и смеще-

ние логической записи в физическом блоке. Для решения этой задачи используются результаты работы логического уровня - смещение логической записи в файле, адрес файла на внешнем устройстве, а также сведения о физической организации файла, включая размер блока. Рис. 7.7 иллюстрирует работу физического уровня для простейшей физической организации файла в виде непрерывной последовательности блоков. Подчеркнем, что задача физического уровня решается независимо от того, как был логически организован файл.

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.

7.8 Отображаемые в память файлы

По сравнению с доступом к памяти, традиционный доступ к файлам выглядит запутанным и неудобным. 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.

Предположим, например, что файл f имеет длину 64 K и отображается на область виртуального адресного пространства с начальным адресом 512 K. После этого любая машинная команда, которая читает содержимое байта по адресу 512 K, получает 0-ой байт этого файла и т.д. Очевидно, что запись по адресу $512 + 1100$ изменяет 1100 байт файла. При завершении процесса на диске остается модифицированная версия файла, как если бы он был изменен комбинацией вызовов SEEK и WRITE.

В действительности при отображении файла внутренние системные таблицы изменяются так, чтобы данный файл служил хранилищем страниц виртуальной памяти на диске. Таким образом, чтение по адресу 512 K вызывает страничный отказ, в результате чего страница 0 переносится в физическую память. Аналогично, запись по адресу $512+1100$ вызывает страничный отказ, 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 под-

держивает 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 рис. 7.8а изображен процесс, который имеет два сегмента-кода и данных. Предположим, что 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. 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.

Хотя отображение файлов исключает потребность в выполнении ввода-вывода и тем самым облегчает программирование, этот способ порождает и некоторые новые проблемы. Во-первых, 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. Как

это определить? Операционная система не может это сообщить. Все, что она может сделать, так это создать файл, длина которого равна размеру страницы. Вторая проблема проявляется (потенциально), если один

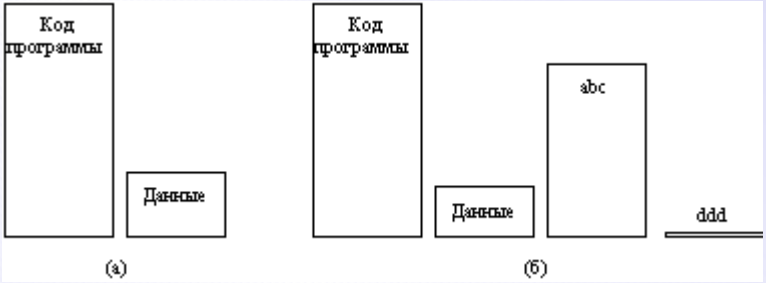


Рис. 7.8: (а) Сегменты процесса перед отображением файлов в адресное пространство; (б) Процесс после отображения существующего файла abc в один сегмент и создания нового сегмента для файла ddd

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.

Третья проблема состоит в том, что 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. Хотя такая работа, очевидно, менее удобна, чем отображение целого файла.

7.9 Современные архитектуры файловых систем

Разработчики новых операционных систем стремятся обеспечить пользователя 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 (Рис. 7.9), на верхнем уровне которой располагается так называемый переключатель файловых систем (в Windows 95, 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.

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.

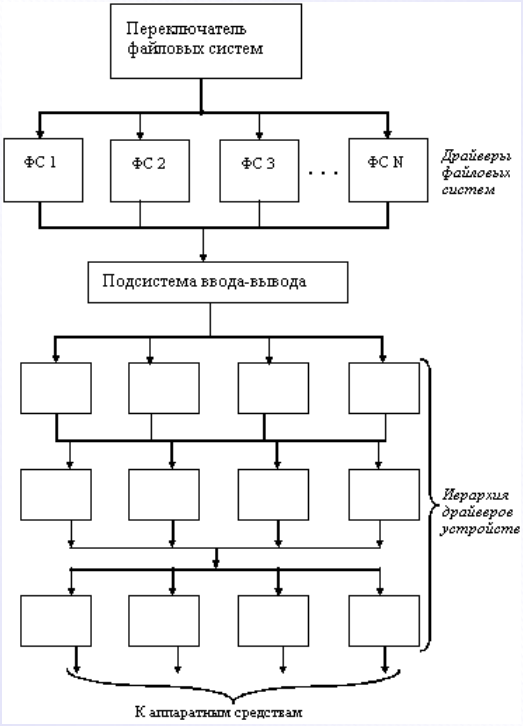


Рис. 7.9: Архитектура современной файловой системы

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 драйверы устройств определенного типа (драйверы жестких дисков или накопителей на лентах), драйверы, поддерживаемые поставщиками (такие драйверы перехватывают запросы к блочным устройствам и могут частично изменить поведение существующего драйвера этого устройства, например, зашифровать данные), 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 устрой-

ства 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.

ГЛАВА 8

Управление вводом-выводом

Одной из главных функций ОС является управление всеми устройствами ввода-вывода компьютера. ОС должна передавать устройствам команды, перехватывать прерывания и обрабатывать ошибки; она также должна обеспечивать интерфейс между устройствами и остальной частью системы. В целях развития интерфейс должен быть одинаковым для всех типов устройств (независимость от устройств).

8.1 Физическая организация устройств ввода-вывода

Устройства ввода-вывода делятся на два типа: *блок-ориентированные* устройства и *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 пре-

рывания в некоторые моменты времени.

Внешнее устройство обычно состоит из механического и 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.

8.2 Организация программного обеспечения ввода-вывода

Основная идея организации программного обеспечения ввода-вывода состоит в разбиении его на несколько уровней, причем нижние уровни обеспечивают экранирование особенностей аппаратуры от верхних, а те, в свою очередь, обеспечивают удобный интерфейс для пользователей.

Ключевым принципом является независимость от устройств. Вид программы не должен зависеть от того, читает ли она данные с гибкого диска или с жесткого диска.

Очень близкой к идее независимости от устройств является идея единообразного именования, то есть для именования устройств должны быть приняты единые правила.

Другим важным вопросом для программного обеспечения ввода-вывода является обработка ошибок. Вообще говоря, 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.

Еще один ключевой вопрос – это 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 автоматически 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.

Последняя проблема состоит в том, что одни устройства являются разделяемыми, а другие - выделенными. 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 целесообразно разделить программное обеспечение ввода-вывода на четыре слоя (Рис. 8.1):

- Обработка прерываний,
- Драйверы устройств,
- Независимый от устройств слой операционной системы,
- Пользовательский слой программного обеспечения.

8.3 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, например, вызов DOWN для семафора, или вызов WAIT для переменной условия, или 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.

8.4 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.

Рис. 8.1: Многоуровневая организация подсистемы ввода-вывода

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. Короче говоря, 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.

8.5 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:

- обеспечение общего интерфейса к драйверам устройств,
- именование устройств,
- защита устройств,
- обеспечение независимого размера блока,
- буферизация,
- распределение памяти на блок-ориентированных устройствах,
- распределение и освобождение выделенных устройств,
- уведомление об ошибках.

Остановимся на некоторых функциях 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.

8.6 Пользовательский слой программного обеспечения

Хотя большая часть программного обеспечения ввода-вывода находится внутри ОС, 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 C, которая принимает строку формата и, возможно, 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. Рассмотрим типичное устройство, требующее спулинга - строчный принтер. Хотя технически легко позволить каждому пользовательскому процессу открыть специальный файл, связанный с принтером, 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.

ГЛАВА 9
Что такое ЛВС?

Локальная сеть (ЛВС) представляет собой коммуникационную систему, позволяющую совместно использовать ресурсы компьютеров, подключенных к сети, таких как принтеры, плоттеры, диски, модемы, приводы CD-ROM и другие периферийные устройства. Локальная сеть обычно ограничена территориально одним или несколькими близко расположенными зданиями. Каждый компьютер в составе ЛВС должен иметь следующие компоненты:

1. сетевой адаптер;
2. кабель;
3. сетевая операционная система (сетевые программы).

9.1 Сетевые адаптеры

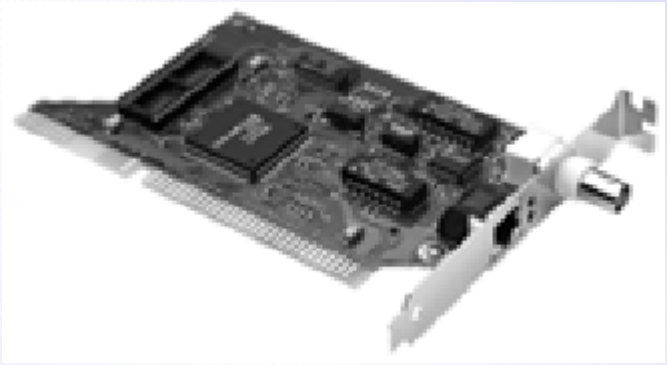


Рис. 9.1: Сетевой адаптер

Сетевые адаптеры (Рис. 9.1) и кабели являются аппаратной основой организации компьютерных сетей, их нормальная работа жизненно важна для сети. С кабелями и адаптерами связано обычно 80% неполадок в сети.

В каждом компьютере должен быть установлен сетевой адаптер, обеспечивающий подключение к выбранному типу кабеля.

Функцией сетевого адаптера является передача и прием сетевых сигналов из кабеля. Адаптер воспринимает команды и данные от сетевой операционной системы (ОС), преобразует эту информацию в один из стандартных форматов и передает ее в сеть через подключенный к адаптеру кабель.

9.2 Конфигурация адаптера

Каждый адаптер, устанавливаемый в компьютер, должен нормально работать с остальной частью ПК. Нужно всегда обращать внимание на два важнейших параметра каждого устройства, устанавливаемого в компьютер.

I/Obase

Базовый адрес ввода-вывода является "каналом по которому адаптер взаимодействует с другими компонентами компьютера. Каждое устройство должно использовать уникальный диапазон адресов ввода-вывода.

IRQ

Номер линии запроса прерывания. Запрос прерывания является сигналом, передаваемым устройством для того, чтобы привлечь внимание процессора (прервать его текущую деятельность). Такой сигнал обычно подается при появлении новых данных или завершении той или иной операции. Каждое устройство должно использовать уникальное значение IRQ.

Таблица 5 содержит адреса ввода-вывода и номера IRQ, используемые различными устройствами.

Устройство	IRQ	IOBase	Адресс памяти	Канал ПДП (DMA)
Системный таймер	0			
Клавиатура	1			
Коммуникационный порт COM1	3	3F8-3FF		
Коммуникационный порт COM2	7	2F8-2FF		
Параллельный (принтерный) порт LPT1	5	378-37F		
Параллельный (принтерный)порт LPT2 [На компьютерах за исключением XT]	5	320-32F	C800-CBFF	3
Дисковый контроллер XT	5			
Дисковый контроллер AT	14	1F0-1F8		
Контроллер SCSI	Различные			1,3
Адаптер VGA	2,9 или без IRQ	3C0-3DA	A0000-BFFFF	0
VGA BIOS			C0000-C7FFF	
Адаптер EGA	2,9 или без IRQ	3C0-3CF	A0000-AFFFF B0000-BFFFF	0
EGA BIOS			C0000-C7FFF	
Адаптер MDA		3B0-3BF	B0000-B3FFF	0
Адаптер Hercules		3B4-3BF	B0000-BfFFF	
Адаптер CGA		3D0-3DF	B8000-B3FFF	0
Буфер символов CGA, EGA			B8000-BBFFF	
Игровой порт(джойстик)		200-20F		

Таблица 5 Адреса ввода-вывода

9.3 Сетевые кабели

Рис. 9.2: Сетевой кабель

Кабель (Рис. 9.2) обеспечивает канал связи компьютера с остальными машинами сети. При установке кабелей нужно точно следовать спецификациям. Пренебрежение этим правилом может принести очень много неприятностей. Отметим разницу между кабелем и кабельным сегментом говоря о кабеле, будем всегда иметь в виду отрезок провода, соединяющего два узла сети; сегментом же будем называть весь комплект кабелей от одного конца сети до другого (между терминаторами). Терминаторы представляют собой резисторы, устанавливаемые на обоих концах сегмента для согласования волнового сопротивления кабеля. Сигнал, дошедший до конца сегмента, поглощается терминатором - это позволяет избавиться от паразитных отраженных сигналов в сети. Если терминаторы не устанавливать, отраженный от конца кабеля сигнал снова попадает в кабель - этот отраженный сигнал будет являться в данном случае помехой и может породить множество проблем вплоть до полной неработоспособности сети.

9.4 Кабель на основе скрученных пар (витая пара, TP)

Кабель содержит две или более пары проводов, скрученных один с другим по всей длине кабеля. Скручивание позволяет повысить помехоустойчивость кабеля и снизить влияние каждой пары на все остальные.

9.5 Коаксиальный кабель

Рис. 9.3: Коаксиальный кабель

Состоит из центрального проводника (сплошного или многожильного), покрытого слоем полимерного изолятора, поверх которого расположен другой проводник (экран) (Рис. 9.3). Экран представляет собой оплетку из медного провода вокруг изолятора или обернутую вокруг изолятора фольгу. В высококачественных кабелях присутствуют и оплетка и фольга. Коаксиальный кабель обеспечивает более высокую помехоустойчивость по сравнению с витой парой, но он дороже. Существуют различные виды коаксиальных кабелей. При установке сети следует выбирать кабель в точном соответствии со спецификацией.

9.6 Оптический кабель

Состоит из одного или нескольких кварцевых волокон (иногда полимерных), покрытых защитной оболочкой. Оболочка как правило состоит из нескольких слоев для обеспечения лучшей защиты волокон.

9.7 Архитектура сети

Сетевая архитектура сродни архитектуре строений. Архитектура здания отражает стиль конструкций и материалы, используемые для постройки. Архитектура сети описывает не только физическое расположение сетевых устройств, но и тип используемых адаптеров и кабелей. Кроме того, сетевая архитектура определяет методы передачи данных по кабелю.

9.7.1 Топология сети

Топология сети описывает схему физического соединения компьютеров. Существуют 3 основных типа сетевой топологии:

Общая шина.

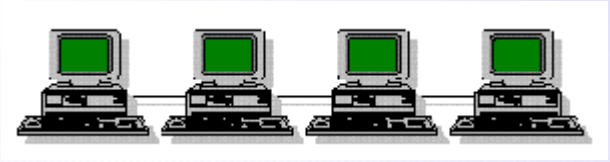


Рис. 9.4: Топология “Общая шина”

При использовании шинной топологии (Рис. 9.4) компьютеры соединяются в одну линию, по концам которой устанавливают терминаторы. Преимущества шинной топологии заключаются в простоте организации сети и низкой стоимости. Недостатком является низкая устойчивость к повреждениям - при любом обрыве кабеля вся сеть перестает работать, а поиск повреждения весьма затруднителен.

Звезда.

При использовании топологии "звезда"(Рис. 9.5), каждый компьютер подключается к специальному концентратору (хабу). Преимуществом этой топологии является ее устойчивость к повреждениям кабеля - при обрыве перестает работать только один из узлов сети и поиск повреждения значительно упрощается. Недостатком является более высокая стоимость.

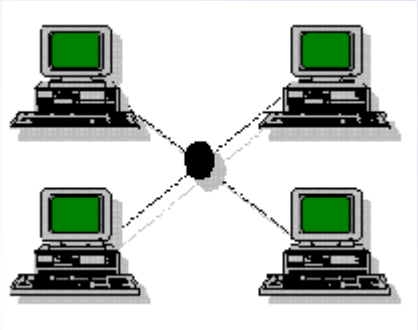


Рис. 9.5: Топология “Звезда”

Кольцо.

При такой топологии узлы сети образуют виртуальное кольцо (концы кабеля соединены друг с другом, Рис. 9.6). Каждый узел сети соединен с двумя соседними. Эту топологию активно продвигает фирма IBM (сети Token Ring). Преимуществом кольцевой топологии является ее высокая надежность (за счет избыточности), однако стоимость такой сети достаточно высока за счет расходов на адаптеры, кабели и дополнительные приспособления.

Спецификации IEEE

Каждая сеть должна следовать определенным правилам (протоколам) при передаче данных от одного компьютера к другому. Протокол определяет способ доступа узла к передающей среде (кабелю) и способ передачи информации от одного узла к другому.

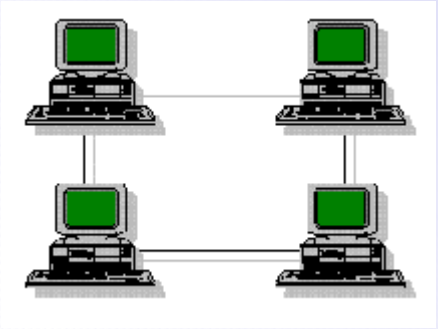


Рис. 9.6: Топология “Кольцо”

В настоящее время используется два типа протоколов доступа к среде:

- **передача маркера** (token) используется в сетях IBM Token Ring и FDDI;
- **множественный доступ с детектированием несущей** (CSMA) используется в сетях Ethernet.

Протокол Ethernet был разработан в 1973 году компанией Xerox и развит впоследствии ею совместно с Intel и Digital Equipment Corp. С тех пор этот протокол стал международным стандартом организации компьютерных сетей. Стандарт был документирован и развит институтом IEEE и получил известность как спецификация IEEE 802.3. IEEE представляет собой организацию Международного Комитета по Стандартам

(ISO), ответственной за выработку сетевых стандартов.

9.8 Серверы и рабочие станции

Сеть представляет собой не просто компьютеры, соединенные вместе кабелем. Сеть - это набор компьютеров, осуществляющих обмен данными между собой с определенными целями. Если компьютер Ивана хочет "разговаривать"с машиной Марии, это обычно означает, что Иван что-то хочет получить от Марии. Для удовлетворения таких потребностей, каждый компьютер должен осуществлять определенные функции. Независимо от используемых на каждом компьютере приложений все машины сети делятся на два класса - серверы и рабочие станции.

9.8.1 Что такое сервер?

Сервером будем называть компьютер, предоставляющий свои ресурсы (например, диски) другим компьютерам сети. Серверы предоставляют свои ресурсы рабочим станциям.

9.8.2 Что такое рабочая станция?

Рабочая станция или клиент использует ресурсы сервера. Рабочие станции имеют доступ к сетевым ресурсам, но своих ресурсов в общее пользование не предоставляют. С сетевыми ресурсами обычно связыва-

ют локальные имена (A-Z для дисков; LPTx, COMx - для портов)

9.8.3 Сети с выделенными серверами и одноранговые сети

- Сети с архитектурой клиент-сервер используют центральный сервер для обслуживания запросов клиентов Рис. 9.7а), тогда как одноранговые сети позволяют любой рабочей станции функционировать одновременно в качестве сервера, если этого требуют задачи Рис. 9.7б).

- По сравнению с универсально одноранговой архитектурой сеть клиент-сервер более специализирована.

Рис. 9.7: Сети с выделенными серверами а) и одноранговые сети б)

9.8.4 Сети с архитектурой клиент-сервер

- Специализированный компьютер (выделенный сервер) используется для установки всех разделяемых ресурсов. Такое решение ускоряет доступ пользователей к централизованным ресурсам сети.
- Сетевое администрирование проще за счет незначительного числа серверов в сети и их узкой специализации.
- Высокие требования к выделенному серверу обеспечение высокой производительности требует установки на сервере большого количества оперативной памяти, диска большого размера и использования в сервере производительного процессора.
- При нарушении работы сервера сеть становится практически неработоспособной.

9.8.5 Одноранговые сети

- Сетевые приложения могут быть распределены по многочисленным серверам для повышения производительности сети и снижения расходов.
- Гибкое разделение ресурсов любого узла сети.
- Администрирование одноранговой сети может быть сложнее за счет

большего числа серверов и более развитых возможностей каждого сервера.

- Невыделенные серверы медленнее специализированных.
- В сети LANtastic могут использоваться как выделенные серверы, так и невыделенные серверы/рабочие станции.

9.8.6 Распределенные (глобальные) сети

В общем случае сети, расположенные в одном (или нескольких близко расположенных) здании называется локальной (ЛВС). Сети, объединяющие компьютеры в разных зданиях, городах и странах, она называется распределенной (WAN -Wide Area Network). Распределенные сети очень большого масштаба (например, Internet, EUNET, Relcom, FIDO) часто называют глобальными. Распределенные сети состоят из трех основных компонент:

1. Локальные сети, как узлы распределенной сети
2. Каналы, соединяющие ЛВС.
3. Оборудование и программы, обеспечивающие локальным сетям доступ к каналам связи.

9.8.7 Оборудование распределенных сетей

Для объединения локальных сетей требуется специальное оборудование независимо от того, находятся ли эти ЛВС в одном здании или связаны через распределенную сеть.

- Повторители (Repeater) - усиливают полученный из кабельного сегмента сигнал и передают его в другой сегмент.
  - объединяют идентичные ЛВС;
  - простое усиление сигналов.
- Мосты (Bridge) передают сообщения на основе записей в таблице пересылки.
  - Возможность фильтрации сетевого трафика;
  - сохраняет информацию о всех узлах;
  - соединяет идентичные или разные сети (например, Ethernet и Token Ring).
- Маршрутизаторы (Router) обеспечивают выбор маршрута обмена данными между узлами сети.
  - Принимает решение о выборе "лучшего пути";

– Дистанция обычно оценивается в интервалах (hop) - промежутках между двумя соседними маршрутизаторами на пути от отправителя к получателю.

ГЛАВА 10
Сетевые операционные системы

10.1 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.

Рис. 10.1: Структура сетевой ОС

В сетевой операционной системе отдельной машины можно выделить несколько частей (Рис. 10.1):

- Средства управления локальными ресурсами компьютера: функции распределения оперативной памяти между процессами, планирования и диспетчеризации процессов, управления процессорами в мультипроцессорных машинах, управления периферийными устройствами и другие функции управления ресурсами локальных ОС.

- Средства предоставления собственных ресурсов и услуг в общее пользование - серверная часть ОС (сервер). Эти средства обеспечивают, например, блокировку файлов и записей, что необходимо для их совместного использования; ведение справочников имен сетевых ресурсов; обработку запросов удаленного доступа к собственной файловой системе и базе данных; управление очередями запросов удаленных пользователей к своим периферийным устройствам.

- Средства запроса доступа к удаленным ресурсам и услугам и их использования - клиентская часть ОС (редиректор). Эта часть выполняет 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. Клиентская часть также осуществляет прием ответов от серверов и преобразование их в локальный формат, так что для приложения выполнение локальных и удаленных запросов неразличимо.

- Коммуникационные средства ОС, с помощью которых происходит обмен сообщениями в сети. Эта часть обеспечивает адресацию и буферизацию сообщений, выбор маршрута передачи сообщения по сети, надежность передачи и т.п., то есть является средством транспортировки сообщений.

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

На Рис. 10.2 показано взаимодействие сетевых компонентов. Здесь компьютер 1 выполняет роль "чистого"клиента, а компьютер 2 - роль "чистого"сервера, соответственно на первой машине отсутствует серверная часть, а на второй - клиентская. На рисунке отдельно показан компонент клиентской части - редиректор. Именно редиректор перехватывает все запросы, поступающие 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.

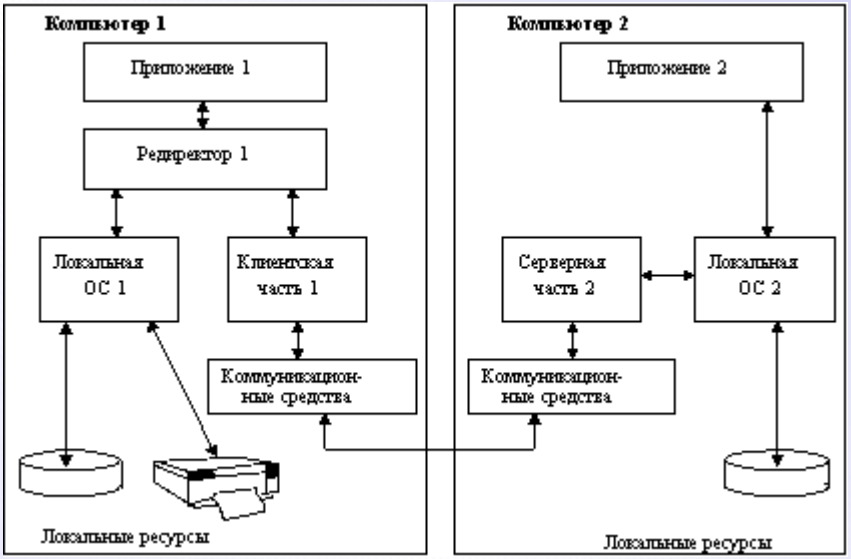


Рис. 10.2: Взаимодействие компонентов операционной системы при взаимодействии компьютеров

На практике сложилось несколько подходов к построению сетевых операционных систем (Рис. 10.3).

Первые сетевые ОС представляли собой совокупность существующей локальной ОС и надстроенной над ней *сетевой оболочки*. При этом в локальную ОС встраивался минимум сетевых функций, необходимых для работы сетевой оболочки, которая выполняла основные сетевые функции. Примером такого подхода является использование на каждой машине сети операционной системы MS DOS (у которой начиная с ее тре-

тьей 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 современных ОС, таких, например, как LANtastic или Personal Ware. Однако более эффективным представляется путь разра-

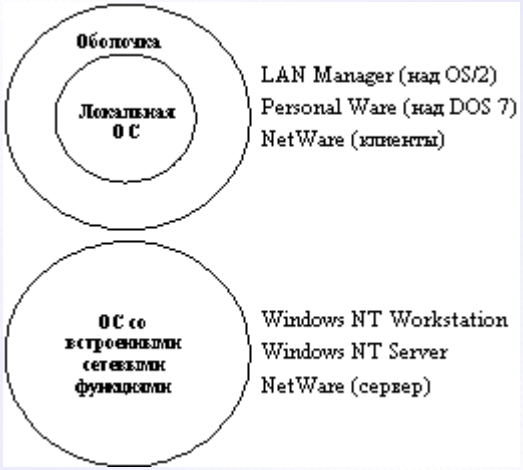


Рис. 10.3: Варианты построения сетевых ОС

ботки операционных систем, изначально предназначенных для работы в сети. Сетевые функции у ОС такого типа глубоко *встроены* в основные модули системы, что обеспечивает их логическую стройность, простоту эксплуатации и модификации, а также высокую производительность. Примером такой ОС является система Windows NT фирмы Microsoft,

которая за счет встроенности сетевых средств обеспечивает более высокие показатели производительности и защищенности информации по сравнению с сетевой ОС LAN Manager той же фирмы (совместная разработка с IBM), являющейся надстройкой над локальной операционной системой OS/2.

10.2 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 (Рис. 10.4). Последние чаще называют сетями с выделенными серверами.

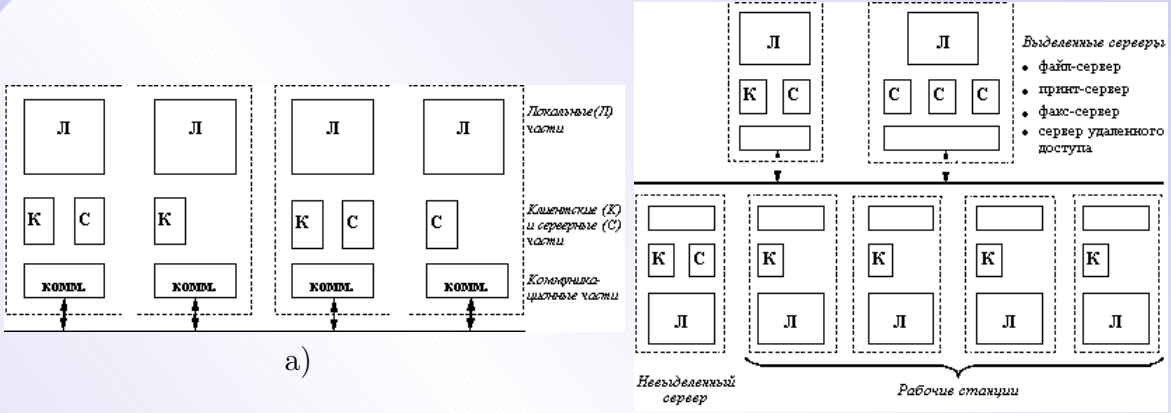
Если компьютер предоставляет свои ресурсы другим пользователям сети, то он играет роль сервера. При этом компьютер, обращающийся к ресурсам другой машины, является клиентом. Как уже было сказано, компьютер, работающий в сети, может выполнять функции либо клиента, либо сервера, либо совмещать обе эти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

Если выполнение каких-либо серверных функций является основным назначением компьютера (например, предоставление файлов в общее пользование всем остальным пользователям сети или организация совместного использования факса, или 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 файл-сервером, факс-сервером, принт-сервером, сервером приложений и т.д.

Очевидно, что 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 более широкий набор функций, поддерживает большее количество одновременных соединений с клиентами, реализует централизованное управление сетью, имеет более развитые средства защиты.

Выделенный сервер не принято использовать в качестве компьютера



а) б)

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.

Важно понять, что 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 *потенциальноравные* возможности. Одноранговые сети могут быть построены, например, на базе ОС LANtastic,

Personal Ware, Windows for Workgroup, Windows NT Workstation.

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 объемам хранимой информации, ее защищенности от несанкционированного доступа и к скорости доступа. 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.

10.3 ОС для рабочих групп и ОС для сетей масштаба предприятия

Сетевые операционные системы 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 в несколько квадратных километров. Сервисы такой сети включают взаимодействие между сетями отделов, доступ к базам данных предприятия, доступ к факс-серверам, высокоскоростным модемам и высокоскоростным принтерам.

- *Сети предприятия (корпоративные сети)* – объединяют все компьютеры всех территорий отдельного предприятия. Они могут покрывать город, регион или даже континент. В таких сетях поль-

зователям предоставляется доступ к информации и приложениям, находящимся в других рабочих группах, других отделах, подразделениях и штаб-квартирах корпорации.

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 добавление новых пользователей, устранение простых отказов, инсталляция новых узлов и установка новых версий программного обеспечения. Операционные системы сетей отделов хорошо отработаны и разнообразны, также, как и сами сети отделов, уже давно применяющиеся и достаточно отлаженные. Такая сеть обычно использует одну или максимум две сетевые ОС. Чаще всего это сеть с выделенным сервером NetWare 3.x или Windows NT, или же одноранговая сеть, например сеть Windows for Workgroups.

Пользователи и администраторы сетей отделов вскоре осознают, что они могут улучшить эффективность своей работы путем получения доступа к информации других отделов своего предприятия. Если сотрудник, занимающийся продажами, может получить доступ к характеристикам конкретного продукта и включить их в презентацию, то эта информация будет более свежей и будет оказывать большее влияние на

покупателей. Если отдел маркетинга может получить доступ к характеристикам продукта, который еще только разрабатывается инженерным отделом, то он может быстро подготовить маркетинговые материалы сразу же после окончания разработки.

Итак, следующим шагом в эволюции сетей является объединение локальных сетей нескольких отделов в единую сеть здания или группы зданий. Такие сети называют сетями кампусов. Сети кампусов могут простираться на несколько километров, но 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. Услуги, предоставляемые ОС сетей кампусов, не ограничиваются простым разделением файлов и принтеров, а часто предоставляют доступ и к серверам других типов, например, к факс-серверам и к серверам высокоскоростных модемов. Важным сервисом, предоставляемым операционными системами данного класса, является доступ к корпоративным базам данных, независимо от того, располагаются ли они на серверах баз данных или на миникомпьютерах.

Именно 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 операцион-

ные среды DOS/Novell и оборудование Token Ring. Очень часто сеть кампуса соединяет разнородные компьютерные системы, в то время как сети отделов используют однотипные компьютеры.

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.

Пользователям корпоративных сетей требуются все те приложения и услуги, которые имеются в сетях отделов и кампусов, плюс некоторые дополнительные приложения и услуги, например, доступ к приложениям мейнфреймов и миникомпьютеров и к глобальным связям. Когда ОС разрабатывается для локальной сети или рабочей группы, то ее главной обязанностью является разделение файлов и других сетевых ресурсов (обычно принтеров) между локально подключенными пользователями. Такой подход не применим для уровня предприятия. 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 почтовая служба, средства коллективной работы, поддержка удаленных пользователей, факс-сервис, обработка голосовых сообщений, организация видеоконференций и др.

Кроме того, многие существующие методы и подходы к решению традиционных задач сетей меньших масштабов для корпоративной сети

оказались непригодными. На первый план вышли такие задачи и проблемы, которые в сетях рабочих групп, отделов и даже кампусов либо имели второстепенное значение, либо вообще не проявлялись. Например, простейшая для небольшой сети задача ведения учетной информации о пользователях выросла в сложную проблему для сети масштаба предприятия. А использование глобальных связей требует от корпоративных ОС поддержки протоколов, хорошо работающих на низкоскоростных линиях, и отказа от некоторых традиционно используемых протоколов (например, тех, которые активно используют широковещательные сообщения). Особое значение приобрели задачи преодоления гетерогенности - в сети появились многочисленные шлюзы, обеспечивающие согласованную работу различных ОС и сетевых системных приложений.

К признакам корпоративных ОС могут быть отнесены также следующие особенности.

Поддержка приложений. В корпоративных сетях выполняются сложные приложения, требующие для выполнения большой вычислительной мощности. Такие приложения разделяются на несколько частей, например, на одном компьютере выполняется часть приложения, связанная с выполнением запросов к базе данных, на другом - запросов к файловому сервису, а на клиентских машинах - 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 кли-

ентов, 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). В этом отношении сетевую ОС NetWare трудно отнести к корпоративным продуктам, так как в ней отсутствуют почти все требования, предъявляемые к серверу приложений. В то же время хорошая поддержка универсальных приложений в Windows NT собственно и позволяет ей претендовать на место в мире корпоративных продуктов.

Справочная служба. Корпоративная ОС должна обладать способностью хранить информацию обо всех пользователях и ресурсах таким образом, чтобы обеспечивалось управление ею из одной центральной точки. Подобно большой организации, корпоративная сеть нуждается в централизованном хранении как можно более полной справочной информации о самой себе (начиная с данных о пользователях, серверах, рабочих станциях и кончая данными о кабельной системе). Естествен-

но организовать эту информацию в виде базы данных. Данные из этой базы могут быть востребованы многими сетевыми системными приложениями, в первую очередь системами управления и администрирования. Кроме этого, такая база полезна при организации электронной почты, систем коллективной работы, службы безопасности, службы инвентаризации программного и аппаратного обеспечения сети, да и для практически любого крупного бизнес-приложения.

База данных, хранящая справочную информацию, предоставляет все то же многообразие возможностей и порождает все то же множество проблем, что и любая другая крупная база данных. 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, специализирующихся на хранении информации того или иного вида, как это часто бывает в реальных операционных системах. Например, в Windows NT имеется по крайней мере пять различных типов справочных баз данных. Главный справочник домена (NT Domain Directory Service) хранит информацию о пользователях, которая используется при организации их логического входа в сеть. Данные о тех же пользователях могут содержаться и в другом справочнике, используемом электрон-

ной почтой Microsoft Mail. Еще три базы данных поддерживают разрешение низкоуровневых адресов: WINS - устанавливает соответствие Netbios-имен IP-адресам, справочник DNS - сервер имен домена - оказывается полезным при подключении NT-сети к Internet, и наконец, 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.

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 децентрализации данных и большой распределенности "законных"точек доступа, из-за большого числа пользователей, благонадежность которых трудно установить, а также из-за большого числа возможных точек несанкционированного подключения к сети. С другой стороны, корпоративные бизнес-приложения работают с данными, которые имеют жизненно важное 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: из-

бирательные или мандатные права доступа, сложные процедуры аутентификации пользователей, программная шифрация.

10.4 Управление распределенными ресурсами

Базовые примитивы передачи сообщений в распределенных системах

Единственным по-настоящему важным отличием распределенных систем от централизованных является межпроцессная взаимосвязь. В централизованных системах связь между процессами, как правило, предполагает наличие разделяемой памяти. Типичный пример - проблема "поставщик-потребитель в этом случае один процесс пишет в разделяемый буфер, а другой - читает из него. Даже наиболее простая форма синхронизации - семафор - требует, чтобы хотя бы одно слово (переменная самого семафора) было разделяемым. В распределенных системах нет какой бы то ни было разделяемой памяти, таким образом вся природа межпроцессных коммуникаций должна быть продумана заново.

Основой этого взаимодействия может служить только передача по сети сообщений. В самом простом случае системные средства обеспечения связи могут быть сведены к двум основным системным вызовам (примитивам), один - для посылки сообщения, другой - для получения сообщения. В дальнейшем на их базе могут быть построены более мощные средства сетевых коммуникаций, такие как распределенная файловая система или вызов удаленных процедур, которые, в свою очередь, также могут служить основой для построения других сетевых сервисов.

Несмотря на концептуальную простоту этих системных вызовов - ПО-

СЛАТЬ и ПОЛУЧИТЬ - существуют различные варианты их реализации, от правильного выбора которых зависит эффективность работы сети. В частности, эффективность коммуникаций в сети зависит от способа задания адреса, от того, является ли системный вызов блокирующим или неблокирующим, какие выбраны способы буферизации сообщений и насколько надежным является протокол обмена сообщениями.

10.4.1 Способы адресации

Для того, чтобы послать сообщение, необходимо указать адрес получателя. В очень простой сети адрес может задаваться в виде константы, но в более сложных сетях нужен и более изощренный способ адресации.

Одним из вариантов адресации на верхнем уровне является использование физических адресов сетевых адаптеров. Если в получающем компьютере выполняется только один процесс, то ядро будет знать, что делать с поступившим сообщением - передать его этому процессу. Однако, если на машине выполняется несколько процессов, то ядру не известно, какому из них предназначено сообщение, поэтому использование сетевого адреса адаптера в качестве адреса получателя приводит к очень серьезному ограничению - на каждой машине должен выполняться только один процесс.

Альтернативная адресная система использует имена назначения, состоящие из двух частей, определяющие номер машины и номер процесса.

Однако адресация типа "машина-процесс"далека от идеала, в частности, она не гибка и не прозрачна, так как пользователь должен явно задавать адрес машины-получателя. В этом случае, если в один прекрасный день машина, на которой работает сервер, отказывает, то программа, в которой жестко используется адрес сервера, не сможет работать с другим сервером, установленном на другой машине.

Другим вариантом могло бы быть назначение каждому процессу уникального адреса, который никак не связан с адресом машины. Одним из способов достижения этой цели является использование централизованного механизма распределения адресов процессов, который работает просто, как счетчик. При получении запроса на выделение адреса он просто возвращает текущее значение счетчика, а затем наращивает его на единицу. Недостатком этой схемы является то, что централизованные компоненты, подобные этому, не обеспечивают в достаточной степени расширяемость систем. Еще один метод назначения процессам уникальных идентификаторов заключается в разрешении каждому процессу выбора своего собственного идентификатора из очень большого адресного пространства, такого как пространство 64-х битных целых чисел. Вероятность выбора одного и того же числа двумя процессами является ничтожной, а система хорошо расширяется. Однако здесь имеется одна проблема: как процесс-отправитель может узнать номер машины процесса-получателя. В сети, которая поддерживает широковещательный режим (то есть в ней предусмотрен такой адрес, который прини-

мают все сетевые адаптеры), отправитель может широковещательно передать специальный пакет, который содержит идентификатор процесса назначения. Все ядра получат эти сообщения, проверят адрес процесса и, если он совпадает с идентификатором одного из процессов этой машины, пошлют ответное сообщение "Я здесь содержащее сетевой адрес машины.

Хотя эта схема и прозрачна, но широковещательные сообщения перегружают систему. Такой перегрузки можно избежать, выделив в сети специальную машину для отображения высокоуровневых символьных имен. При применении такой системы процессы адресуются с помощью символьных строк, и в программы вставляются эти строки, а не номера машин или процессов. Каждый раз перед первой попыткой связаться, процесс должен послать запрос специальному отображающему процессу, обычно называемому сервером имен, запрашивая номер машины, на которой работает процесс-получатель.

Совершенно иной подход - это использование специальной аппаратуры. Пусть процессы выбирают свои адреса случайно, а конструкция сетевых адаптеров позволяет хранить эти адреса. Теперь адреса процессов не обнаруживаются путем широковещательной передачи, а непосредственно указываются в кадрах, заменяя там адреса сетевых адаптеров.

10.4.2 Блокирующие и неблокирующие примитивы

Примитивы бывают блокирующими и неблокирующими, иногда они называются соответственно синхронными и асинхронными. При использовании блокирующего примитива, процесс, выдавший запрос на его выполнение, приостанавливается до полного завершения примитива. Например, вызов примитива ПОЛУЧИТЬ приостанавливает вызывающий процесс до получения сообщения.

При использовании неблокирующего примитива управление возвращается вызывающему процессу немедленно, еще до того, как требуемая работа будет выполнена. Преимуществом этой схемы является параллельное выполнение вызывающего процесса и процесса передачи сообщения. Обычно в ОС имеется один из двух видов примитивов и очень редко - оба. Однако выигрыш в производительности при использовании неблокирующих примитивов компенсируется серьезным недостатком: отправитель не может модифицировать буфер сообщения, пока сообщение не отправлено, а узнать, отправлено ли сообщение, отправитель не может. Отсюда сложности в построении программ, которые передают последовательность сообщений с помощью неблокирующих примитивов.

Имеется два возможных выхода. Первое решение - это заставить ядро копировать сообщение в свой внутренний буфер, а затем разрешить процессу продолжить выполнение. С точки зрения процесса эта схема ничем не отличается от схемы блокирующего вызова: как только процесс

снова получает управление, он может повторно использовать буфер.

Второе решение заключается в прерывании процесса-отправителя после отправки сообщения, чтобы проинформировать его, что буфер снова доступен. Здесь не требуется копирование, что экономит время, но прерывание пользовательского уровня делает программирование запутанным, сложным, может привести к возникновению гонок.

Вопросом, тесно связанным с блокирующими и неблокирующими вызовами, является вопрос тайм-аутов. В системе с блокирующим вызовом ПОСЛАТЬ при отсутствии ответа вызывающий процесс может заблокироваться навсегда. Для предотвращения такой ситуации в некоторых системах вызывающий процесс может задать временной интервал, в течение которого он ждет ответ. Если за это время сообщение не поступает, вызов ПОСЛАТЬ завершается с кодом ошибки.

10.4.3 Буферизуемые и небуферизуемые примитивы

Примитивы, которые были описаны, являются небуферизуемыми примитивами. Это означает, что вызов ПОЛУЧИТЬ сообщает ядру машины, на которой он выполняется, адрес буфера, в который следует поместить пребывающее для него сообщение.

Эта схема работает прекрасно при условии, что получатель выполняет вызов ПОЛУЧИТЬ раньше, чем отправитель выполняет вызов ПОСЛАТЬ. Вызов ПОЛУЧИТЬ сообщает ядру машины, на которой вы-

полняется, по какому адресу должно поступить ожидаемое сообщение, и в какую область памяти необходимо его поместить. Проблема возникает тогда, когда вызов ПОСЛАТЬ сделан раньше вызова ПОЛУЧИТЬ. Каким образом сможет узнать ядро на машине получателя, какому процессу адресовано вновь поступившее сообщение, если их несколько? И как оно узнает, куда его скопировать?

Один из вариантов - просто отказаться от сообщения, позволить отправителю взять тайм-аут и надеяться, что получатель все-таки выполнит вызов ПОЛУЧИТЬ перед повторной передачей сообщения. Этот подход не сложен в реализации, но, к сожалению, отправитель (или скорее ядро его машины) может сделать несколько таких безуспешных попыток. Еще хуже то, что после достаточно большого числа безуспешных попыток ядро отправителя может сделать неправильный вывод об аварии на машине получателя или о неправильности использованного адреса.

Второй подход к этой проблеме заключается в том, чтобы хранить хотя бы некоторое время, поступающие сообщения в ядре получателя на тот случай, что вскоре будет выполнен соответствующий вызов ПОЛУЧИТЬ. Каждый раз, когда поступает такое "неожидаемое"сообщение, включается таймер. Если заданный временной интервал истекает раньше, чем происходит соответствующий вызов ПОЛУЧИТЬ, то сообщение теряется.

Хотя этот метод и уменьшает вероятность потери сообщений, он по-

рождает проблему хранения и управления преждевременно поступившими сообщениями. Необходимы буферы, которые следует где-то размещать, освобождать, в общем, которыми нужно управлять. Концептуально простым способом управления буферами является определение новой структуры данных, называемой почтовым ящиком.

Процесс, который заинтересован в получении сообщений, обращается к ядру с запросом о создании для него почтового ящика и сообщает адрес, по которому ему могут поступать сетевые пакеты, после чего все сообщения с данным адресом будут помещены в его почтовый ящик. Такой способ часто называют буферизуемым примитивом.

10.4.4 Надежные и ненадежные примитивы

Ранее подразумевалось, что когда отправитель посылает сообщение, адресат его обязательно получает. Но реально сообщения могут теряться. Предположим, что используются блокирующие примитивы. Когда отправитель посылает сообщение, то он приостанавливает свою работу до тех пор, пока сообщение не будет послано. Однако нет никаких гарантий, что после того, как он возобновит свою работу, сообщение будет доставлено адресату.

Для решения этой проблемы существует три подхода. Первый заключается в том, что система не берет на себя никаких обязательств по поводу доставки сообщений. Реализация надежного взаимодействия

становится целиком заботой пользователя.

Второй подход заключается в том, что ядро принимающей машины посылает квитанцию-подтверждение ядру отправляющей машины на каждое сообщение. Посылающее ядро разблокирует пользовательский процесс только после получения этого подтверждения. Подтверждение передается от ядра к ядру. Ни отправитель, ни получатель его не видят.

Третий подход заключается в использовании ответа в качестве подтверждения в тех системах, в которых запрос всегда сопровождается ответом. Отправитель остается заблокированным до получения ответа. Если ответа нет слишком долго, то посылающее ядро может переслать запрос специальной службе предотвращения потери сообщений.

ГЛАВА 11
Заключение

11.1 Современные концепции и технологии проектирования операционных систем

11.1.1 Требования, предъявляемые к ОС 90-х годов

Операционная система является сердцевиной сетевого программного обеспечения, она создает среду для выполнения приложений и во многом определяет, какими полезными для пользователя свойствами эти приложения будут обладать. В связи с этим рассмотрим требования, которым должна удовлетворять современная ОС.

Очевидно, что главным требованием, предъявляемым к операционной системе, является способность выполнения основных функций: эффективного управления ресурсами и обеспечения удобного интерфейса для пользователя и прикладных программ. Современная ОС, как правило, должна реализовывать мультипрограммную обработку, виртуальную память, свопинг, поддерживать многооконный интерфейс, а также выполнять многие другие, совершенно необходимые функции. Кроме этих функциональных требований к операционным системам предъявляются не менее важные рыночные требования. К этим требованиям относятся:

- *Расширяемость*. Код должен быть написан таким образом, чтобы

можно было легко внести дополнения и изменения, если это потребуется, и не нарушить целостность системы.

- *Переносимость*. Код должен легко переноситься с процессора одного типа на процессор другого типа и с аппаратной платформы (которая включает наряду с типом процессора и способ организации всей аппаратуры компьютера) одного типа на аппаратную платформу другого типа.

- *Надежность и отказоустойчивость*. Система должна быть защищена как от внутренних, так и от внешних ошибок, сбоев и отказов. Ее действия должны быть всегда предсказуемыми, а приложения не должны быть в состоянии наносить вред ОС.

- *Совместимость*. ОС должна иметь средства для выполнения прикладных программ, написанных для других операционных систем. Кроме того, пользовательский интерфейс должен быть совместим с существующими системами и стандартами.

- *Безопасность*. ОС должна обладать средствами защиты ресурсов одних пользователей от других.

- *Производительность*. Система должна обладать настолько хорошим быстродействием и временем реакции, насколько это позволяет аппаратная платформа.

Рассмотрим более подробно некоторые из этих требований.

Расширяемость

В то время, как аппаратная часть компьютера устаревает за несколько лет, полезная жизнь операционных систем может измеряться десятилетиями. Примером может служить ОС UNIX. Поэтому операционные системы всегда эволюционно изменяются со временем, и эти изменения более значимы, чем изменения аппаратных средств. Изменения ОС обычно представляют собой приобретение ею новых свойств. Например, поддержка новых устройств, таких как CD-ROM, возможность связи с сетями нового типа, поддержка многообещающих технологий, таких как графический интерфейс пользователя или объектно-ориентированное программное окружение, использование более чем одного процессора. Сохранение целостности кода, какие бы изменения не вносились в операционную систему, является главной целью разработки.

Расширяемость может достигаться за счет модульной структуры ОС, при которой программы строятся из набора отдельных модулей, взаимодействующих только через функциональный интерфейс. Новые компоненты могут быть добавлены в операционную систему модульным путем, они выполняют свою работу, используя интерфейсы, поддерживаемые существующими компонентами.

Использование объектов для представления системных ресурсов также улучшает расширяемость системы. Объекты - это абстрактные типы

данных, над которыми можно производить только те действия, которые предусмотрены специальным набором объектных функций. Объекты позволяют единообразно управлять системными ресурсами. Добавление новых объектов не разрушает существующие объекты и не требует изменений существующего кода.

Прекрасные возможности для расширения предоставляет подход к структурированию ОС по типу клиент-сервер с использованием микроядерной технологии. В соответствии с этим подходом ОС строится как совокупность привилегированной управляющей программы и набора непривилегированных услуг-серверов. Основная часть ОС может оставаться неизменной в то время, как могут быть добавлены новые серверы или улучшены старые.

Средства вызова удаленных процедур (RPC) также дают возможность расширить функциональные возможности ОС. Новые программные процедуры могут быть добавлены в любую машину сети и немедленно поступить в распоряжение прикладных программ на других машинах сети.

Некоторые ОС для улучшения расширяемости поддерживают загружаемые драйверы, которые могут быть добавлены в систему во время ее работы. Новые файловые системы, устройства и сети могут поддерживаться путем написания драйвера устройства, драйвера файловой системы или транспортного драйвера и загрузки его в систему.

Требование переносимости кода тесно связано с расширяемостью. Расширяемость позволяет улучшать операционную систему, в то время как переносимость дает возможность перемещать всю систему на машину, базирующуюся на другом процессоре или аппаратной платформе, делая при этом по возможности небольшие изменения в коде. Хотя ОС часто описываются либо как переносимые, либо как непереносимые, переносимость - это не бинарное состояние. Вопрос не в том, может ли быть система перенесена, а в том, насколько легко можно это сделать. Написание переносимой ОС аналогично написанию любого переносимого кода - нужно следовать некоторым правилам.

Во-первых, большая часть кода должна быть написана на языке, который имеется на всех машинах, куда вы хотите переносить систему. Обычно это означает, что код должен быть написан на языке высокого уровня, предпочтительно стандартизованном, например, на языке C. Программа, написанная на ассемблере, не является переносимой, если только вы не собираетесь переносить ее на машину, обладающую командной совместимостью с вашей.

Во-вторых, следует учесть, в какое физическое окружение программа должна быть перенесена. Различная аппаратура требует различных решений при создании ОС. Например, ОС, построенная на 32-битовых адресах, не может быть перенесена на машину с 16-битовыми адресами

(разве что с огромными трудностями).

В-третьих, важно минимизировать или, если возможно, исключить те части кода, которые непосредственно взаимодействуют с аппаратными средствами. Зависимость от аппаратуры может иметь много форм. Некоторые очевидные формы зависимости включают прямое манипулирование регистрами и другими аппаратными средствами.

В-четвертых, если аппаратно зависимый код не может быть полностью исключен, то он должен быть изолирован в нескольких хорошо локализуемых модулях. Аппаратно-зависимый код не должен быть распределен по всей системе. Например, можно спрятать аппаратно-зависимую структуру в программно-задаваемые данные абстрактного типа. Другие модули системы будут работать с этими данными, а не с аппаратурой, используя набор некоторых функций. Когда ОС переносится, то изменяются только эти данные и функции, которые ими манипулируют.

Для легкого переноса ОС при ее разработке должны быть соблюдены следующие требования:

- *Переносимый язык высокого уровня.* Большинство переносимых ОС написано на языке C (стандарт ANSI X3.159-1989). Разработчики выбирают C потому, что он стандартизован, и потому, что C-компиляторы широко доступны. Ассемблер используется только для тех частей системы, которые должны непосредственно взаимодействовать с аппаратурой (например, обработчик прерываний) или

для частей, которые требуют максимальной скорости (например, целочисленная арифметика повышенной точности). Однако непереносимый код должен быть тщательно изолирован внутри тех компонентов, где он используется.

- *Изоляция процессора.* Некоторые низкоуровневые части ОС должны иметь доступ к процессорно-зависимым структурам данных и регистрам. Однако код, который делает это, должен содержаться в небольших модулях, которые могут быть заменены аналогичными модулями для других процессоров.

- *Изоляция платформы.* Зависимость от платформы заключается в различиях между рабочими станциями разных производителей, построенными на одном и том же процессоре (например, MIPS R4000). Должен быть введен программный уровень, абстрагирующий аппаратуру (кэши, контроллеры прерываний ввода-вывода и т. п.) вместе со слоем низкоуровневых программ таким образом, чтобы высокоуровневый код не нуждался в изменении при переносе с одной платформы на другую.

Совместимость

Одним из аспектов совместимости является способность ОС выполнять программы, написанные для других ОС или для более ранних версий

данной операционной системы, а также для другой аппаратной платформы.

Необходимо разделять вопросы двоичной совместимости и совместимости на уровне исходных текстов приложений. Двоичная совместимость достигается в том случае, когда можно взять исполняемую программу и запустить ее на выполнение на другой ОС. Для этого необходимы: совместимость на уровне команд процессора, совместимость на уровне системных вызовов и даже на уровне библиотечных вызовов, если они являются динамически связываемыми.

Совместимость на уровне исходных текстов требует наличия соответствующего компилятора в составе программного обеспечения, а также совместимости на уровне библиотек и системных вызовов. При этом необходима перекомпиляция имеющихся исходных текстов в новый выполняемый модуль.

Совместимость на уровне исходных текстов важна в основном для разработчиков приложений, в распоряжении которых эти исходные тексты всегда имеются. Но для конечных пользователей практическое значение имеет только двоичная совместимость, так как только в этом случае они могут использовать один и тот же коммерческий продукт, поставляемый в виде двоичного исполняемого кода, в различных операционных средах и на различных машинах.

Обладает ли новая ОС двоичной совместимостью или совместимостью исходных текстов с существующими системами, зависит от многих

факторов. Самый главный из них - архитектура процессора, на котором работает новая ОС. Если процессор, на который переносится ОС, использует тот же набор команд (возможно с некоторыми добавлениями) и тот же диапазон адресов, тогда двоичная совместимость может быть достигнута достаточно просто.

Гораздо сложнее достичь двоичной совместимости между процессорами, основанными на разных архитектурах. Для того, чтобы один компьютер выполнял программы другого (например, DOS-программу на Mac), этот компьютер должен работать с машинными командами, которые ему изначально непонятны. Например, процессор типа 680x0 на Mac должен исполнять двоичный код, предназначенный для процессора 80x86 в PC. Процессор 80x86 имеет свои собственные дешифратор команд, регистры и внутреннюю архитектуру. Процессор 680x0 не понимает двоичный код 80x86, поэтому он должен выбрать каждую команду, декодировать ее, чтобы определить, для чего она предназначена, а затем выполнить эквивалентную подпрограмму, написанную для 680x0. Так как к тому же у 680x0 нет в точности таких же регистров, флагов и внутреннего арифметико-логического устройства, как в 80x86, он должен имитировать все эти элементы с использованием своих регистров или памяти. И он должен тщательно воспроизводить результаты каждой команды, что требует специально написанных подпрограмм для 680x0, гарантирующих, что состояние эмулируемых регистров и флагов после выполнения каждой команды будет в точности таким же, как и

на реальном 80x86.

Это простая, но очень медленная работа, так как микрокод внутри процессора 80x86 исполняется на значительно более быстродействующем уровне, чем эмулирующие его внешние команды 680x0. За время выполнения одной команды 80x86 на 680x0, реальный 80x86 может выполнить десятки команд. Следовательно, если процессор, производящий эмуляцию, не настолько быстр, чтобы компенсировать все потери при эмуляции, то программы, исполняющиеся под эмуляцией, будут очень медленными.

Выходом в таких случаях является использование так называемых прикладных сред. Учитывая, что основную часть программы, как правило, составляют вызовы библиотечных функций, прикладная среда имитирует библиотечные функции целиком, используя заранее написанную библиотеку функций аналогичного назначения, а остальные команды эмулирует каждую по отдельности.

Соответствие стандартам POSIX также является средством обеспечения совместимости программных и пользовательских интерфейсов. Во второй половине 80-х правительственные агентства США начали разрабатывать POSIX как стандарты на поставляемое оборудование при заключении правительственных контрактов в компьютерной области. POSIX - это "интерфейс переносимой ОС, базирующейся на UNIX". POSIX - собрание международных стандартов интерфейсов ОС в стиле UNIX. Использование стандарта POSIX (IEEE стандарт 1003.1 - 1988)

позволяет создавать программы стиле UNIX, которые могут легко переноситься из одной системы в другую.

Безопасность

В дополнение к стандарту POSIX правительство США также определило требования компьютерной безопасности для приложений, используемых правительством. Многие из этих требований являются желаемыми свойствами для любой многопользовательской системы. Правила безопасности определяют такие свойства, как защита ресурсов одного пользователя от других и установление квот по ресурсам для предотвращения захвата одним пользователем всех системных ресурсов (таких как память).

Обеспечение защиты информации от несанкционированного доступа является обязательной функцией сетевых операционных систем. В большинстве популярных систем гарантируется степень безопасности данных, соответствующая уровню C2 в системе стандартов США.

Основы стандартов в области безопасности были заложены "*Критериями оценки надежных компьютерных систем*". Этот документ, изданный в США в 1983 году национальным центром компьютерной безопасности (NCSC - National Computer Security Center), часто называют Оранжевой Книгой.

В соответствии с требованиями Оранжевой книги безопасной считается такая система, которая "посредством специальных механизмов за-

щиты контролирует доступ к информации таким образом, что только имеющие соответствующие полномочия лица или процессы, выполняющиеся от их имени, могут получить доступ на чтение, запись, создание или удаление информации".

Иерархия уровней безопасности, приведенная в Оранжевой Книге, помечает низший уровень безопасности как D, а высший - как A.

- В класс D попадают системы, оценка которых выявила их несоответствие требованиям всех других классов.

- Основными свойствами, характерными для C-систем, являются: наличие подсистемы учета событий, связанных с безопасностью, и избирательный контроль доступа. Уровень C делится на 2 подуровня: уровень C1, обеспечивающий защиту данных от ошибок пользователей, но не от действий злоумышленников, и более строгий уровень C2. На уровне C2 должны присутствовать *средства секретного входа*, обеспечивающие идентификацию пользователей путем ввода уникального имени и пароля перед тем, как им будет разрешен доступ к системе. *Избирательный контроль доступа*, требуемый на этом уровне позволяет владельцу ресурса определить, кто имеет доступ к ресурсу и что он может с ним делать. Владелец делает это путем предоставляемых прав доступа пользователю или группе пользователей. *Средства учета и наблюдения (auditing)* - обеспечивают возможность обнаружить и зафиксировать важные

события, связанные с безопасностью, или любые попытки создать, получить доступ или удалить системные ресурсы. *Защита памяти* – заключается в том, что память инициализируется перед тем, как повторно используется. На этом уровне система не защищена от ошибок пользователя, но поведение его может быть проконтролировано по записям в журнале, оставленным средствами наблюдения и аудитинга.

- Системы уровня B основаны на помеченных данных и распределении пользователей по категориям, то есть реализуют *мандатный контроль доступа*. Каждому пользователю присваивается рейтинг защиты, и он может получать доступ к данным только в соответствии с этим рейтингом. Этот уровень в отличие от уровня C защищает систему от ошибочного поведения пользователя.

- Уровень A является самым высоким уровнем безопасности, он требует в дополнение ко всем требованиям уровня B выполнения формального, математически обоснованного доказательства соответствия системы требованиям безопасности.

Различные коммерческие структуры (например, банки) особо выделяют необходимость учетной службы, аналогичной той, что предлагают государственные рекомендации C2. Любая деятельность, связанная с безопасностью, может быть отслежена и тем самым учтена. Это как раз то,

что требует C2 и то, что обычно нужно банкам. Однако, коммерческие пользователи, как правило, не хотят расплачиваться производительностью за повышенный уровень безопасности. A-уровень безопасности занимает своими управляющими механизмами до 90% процессорного времени. Более безопасные системы не только снижают эффективность, но и существенно ограничивают число доступных прикладных пакетов, которые соответствующим образом могут выполняться в подобной системе. Например для ОС Solaris (версия UNIX) есть несколько тысяч приложений, а для ее аналога B-уровня - только сотня.

11.2 Тенденции в структурном построении ОС

Как уже отмечалось выше, для удовлетворения требований, предъявляемых к современной ОС, большое значение имеет ее структурное построение. Операционные системы прошли длительный путь развития от монолитных систем к хорошо структурированным модульным системам, способным к развитию, расширению и легкому переносу на новые платформы.

11.2.1 Монолитные системы

В общем случае "структура"монолитной системы представляет собой отсутствие структуры (Рис. 11.1). ОС написана как набор процедур,

каждая из которых может вызывать другие, когда ей это нужно. При использовании этой техники каждая процедура системы имеет хорошо определенный интерфейс в терминах параметров и результатов, и каждая вольна вызвать любую другую для выполнения некоторой нужной для нее полезной работы.

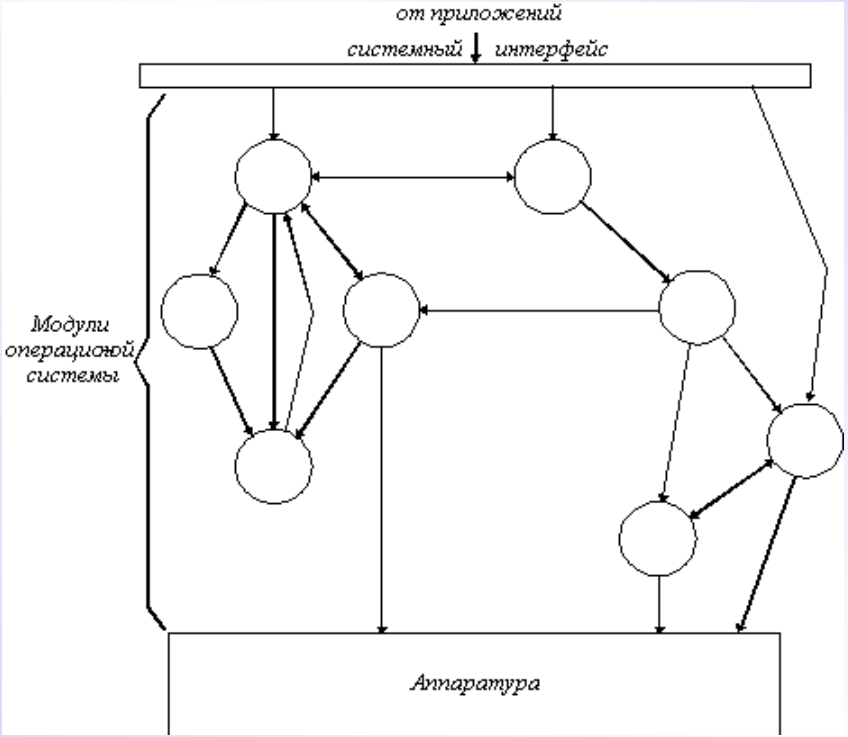


Рис. 11.1: Монолитная структура ОС

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

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

1. Главная программа, которая вызывает требуемые сервисные процедуры.

2. Набор сервисных процедур, реализующих системные вызовы.

3. Набор утилит, обслуживающих сервисные процедуры.

В этой модели для каждого системного вызова имеется одна сервисная процедура. Утилиты выполняют функции, которые нужны нескольким сервисным процедурам. Это деление процедур на три слоя показано на Рис. 11.2.

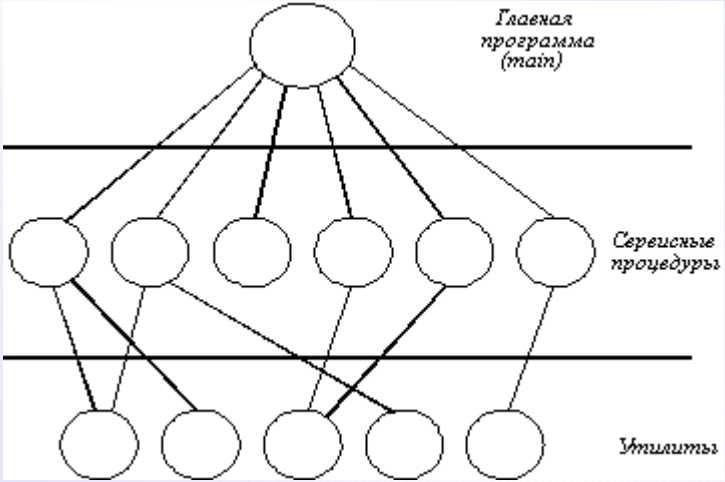


Рис. 11.2: Простая структуризация монолитной ОС

11.2.2 Многоуровневые системы

Обобщением предыдущего подхода является организация ОС как

иерархии уровней. Уровни образуются группами функций операционной системы - файловая система, управление процессами и устройствами и т.п. Каждый уровень может взаимодействовать только со своим непосредственным соседом - выше- или нижележащим уровнем. Прикладные программы или модули самой операционной системы передают запросы вверх и вниз по этим уровням.

Первой системой, построенной таким образом была простая пакетная система THE, которую построил Дейкстра и его студенты в 1968 году.

Система имела 6 уровней. Уровень 0 занимался распределением времени процессора, переключая процессы по прерыванию или по истечении времени. Уровень 1 управлял памятью - распределял оперативную память и пространство на магнитном барабане для тех частей процессов (страниц), для которых не было места в ОП, то есть слой 1 выполнял функции виртуальной памяти. Слой 2 управлял связью между консолью оператора и процессами. С помощью этого уровня каждый процесс имел свою собственную консоль оператора. Уровень 3 управлял устройствами ввода-вывода и буферизовал потоки информации к ним и от них. С помощью уровня 3 каждый процесс вместо того, чтобы работать с конкретными устройствами, с их разнообразными особенностями, обращался к абстрактным устройствам ввода-вывода, обладающим удобными для пользователя характеристиками. На уровне 4 работали пользовательские программы, которым не надо было заботиться ни о процессах, ни о памяти, ни о консоли, ни об управлении устройствами

ввода-вывода. Процесс системного оператора размещался на уровне 5.

В системе THE многоуровневая схема служила, в основном, целям разработки, так как все части системы компоновались затем в общий объектный модуль . Дальнейшее обобщение многоуровневой концепции было сделано в ОС MULTICS. В системе MULTICS каждый уровень (называемый кольцом) является более привилегированным, чем вышележащий. Когда процедура верхнего уровня хочет вызвать процедуру нижележащего, она должна выполнить соответствующий системный вызов, то есть команду TRAP (прерывание), параметры которой тщательно проверяются перед тем, как выполняется вызов. Хотя ОС в MULTICS является частью адресного пространства каждого пользовательского процесса, аппаратура обеспечивает защиту данных на уровне сегментов памяти, разрешая, например, доступ к одним сегментам только для записи, а к другим - для чтения или выполнения. Преимущество подхода MULTICS заключается в том, что он может быть расширен и на структуру пользовательских подсистем. Например, профессор может написать программу для тестирования и оценки студенческих программ и запустить эту программу на уровне n, в то время как студенческие программы будут работать на уровне n+1, так что они не смогут изменить свои оценки.

Многоуровневый подход был также использован при реализации различных вариантов ОС UNIX.

Хотя такой структурный подход на практике обычно работал непло-

хо, сегодня он все больше воспринимается монолитным. В системах, имеющих многоуровневую структуру было нелегко удалить один слой и заменить его другим в силу множественности и размытости интерфейсов между слоями. Добавление новых функций и изменение существующих требовало хорошего знания операционной системы и массы времени. Когда стало ясно, что операционные системы живут долго и должны иметь возможности развития и расширения, монолитный подход стал давать трещину, и на смену ему пришла модель клиент-сервер и тесно связанная с ней концепция микроядра.

11.2.3 Модель клиент-сервер и микроядра

Модель клиент-сервер - это еще один подход к структурированию ОС. В широком смысле модель клиент-сервер предполагает наличие программного компонента - потребителя какого-либо сервиса - клиента, и программного компонента - поставщика этого сервиса - сервера. Взаимодействие между клиентом и сервером стандартизуется, так что сервер может обслуживать клиентов, реализованных различными способами и, может быть, разными производителями. При этом главным требованием является то, чтобы они запрашивали услуги сервера понятным ему способом. Инициатором обмена обычно является клиент, который посылает запрос на обслуживание серверу, находящемуся в состоянии ожидания запроса. Один и тот же программный компонент мо-

жет быть клиентом по отношению к одному виду услуг, и сервером для другого вида услуг. Модель клиент-сервер является скорее удобным концептуальным средством ясного представления функций того или иного программного элемента в той или иной ситуации, нежели технологией. Эта модель успешно применяется не только при построении ОС, но и на всех уровнях программного обеспечения, и имеет в некоторых случаях более узкий, специфический смысл, сохраняя, естественно, при этом все свои общие черты.

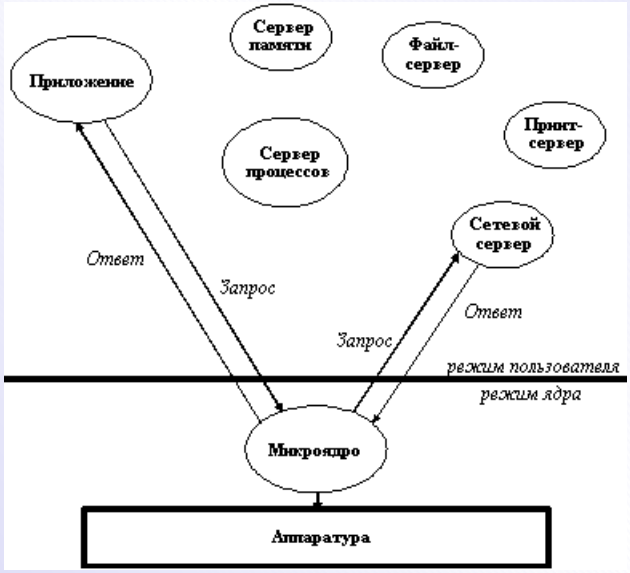


Рис. 11.3: Структура ОС клиент-сервер

Применительно к структурированию ОС идея состоит в разбиении ее на несколько процессов - серверов, каждый из которых выполняет отдельный набор сервисных функций - например, управление памятью, создание или планирование процессов. Каждый сервер выполняется в пользовательском режиме. Клиент, которым может быть либо другой компонент ОС, либо прикладная программа, запрашивает сервис, посылая сообщение на сервер. Ядро ОС (называемое здесь микроядром), работая в привилегированном режиме, доставляет сообщение нужному серверу, сервер выполняет операцию, после чего ядро возвращает результаты клиенту с помощью другого сообщения (Рис. 11.3).

Подход с использованием микроядра заменил вертикальное распределение функций операционной системы на горизонтальное. Компоненты, лежащие выше микроядра, хотя и используют сообщения, пересылаемые через микроядро, взаимодействуют друг с другом непосредственно. Микроядро играет роль регулировщика. Оно проверяет сообщения, пересылает их между серверами и клиентами, и предоставляет доступ к аппаратуре.

Данная теоретическая модель является идеализированным описанием системы клиент-сервер, в которой ядро состоит только из средств передачи сообщений. В действительности различные варианты реализации модели клиент-сервер в структуре ОС могут существенно различаться по объему работ, выполняемых в режиме ядра.

На одном краю этого спектра находится разрабатываемая фирмой

IBM на основе микроядра Mach операционная система Workplace OS, придерживающаяся чистой микроядерной доктрины, состоящей в том, что все несущественные функции ОС должны выполняться не в режиме ядра, а в непривилегированном (пользовательском) режиме. На другом - Windows NT, в составе которой имеется исполняющая система (NT executive), работающая в режиме ядра и выполняющая функции обеспечения безопасности, ввода-вывода и другие.

Микроядро реализует жизненно важные функции, лежащие в основе операционной системы. Это базис для менее существенных системных служб и приложений. Именно вопрос о том, какие из системных функций считать несущественными, и, соответственно, не включать их в состав ядра, является предметом спора среди соперничающих сторонников идеи микроядра. В общем случае, подсистемы, бывшие традиционно неотъемлемыми частями операционной системы - файловые системы, управление окнами и обеспечение безопасности - становятся периферийными модулями, взаимодействующими с ядром и друг с другом.

Главный принцип разделения работы между микроядром и окружающими его модулями - включать в микроядро только те функции, которым абсолютно необходимо исполняться в режиме супервизора и в привилегированном пространстве. Под этим обычно подразумеваются машинозависимые программы (включая поддержку нескольких процессоров), некоторые функции управления процессами, обработка прерываний, поддержка пересылки сообщений, некоторые функции управ-

ления устройствами ввода-вывода, связанные с загрузкой команд в регистры устройств. Эти функции операционной системы трудно, если не невозможно, выполнить программам, работающим в пространстве пользователя.

Есть два пути решения этой проблемы. Один путь - разместить несколько таких, чувствительных к режиму работы процессора, серверов, в пространстве ядра, что обеспечит им полный доступ к аппаратуре и, в то же время, связь с другими процессами с помощью обычного механизма сообщений. Такой подход был использован, например, при разработке Windows NT: кроме микроядра, в привилегированном режиме работает часть Windows NT, называемая executive управляющей программой. Она включает ряд компонентов, которые управляют виртуальной памятью, объектами, вводом-выводом и файловой системой (включая сетевые драйверы), взаимодействием процессов, и частично системой безопасности.

Другой путь, заключается в том, чтобы оставить в ядре только небольшую часть сервера, представляющую собой механизм реализации решения, а часть, отвечающую за принятие решения, переместить в пользовательскую область. В соответствии с этим подходом, например, в микроядре Mach, на базе которого разработана Workplace OS, размещается только часть системы управления процессами (и нитями), реализующая диспетчеризацию (то есть непосредственно переключение с процесса на процесс), а все функции, связанные с анализом приоритетов, выбором

очередного процесса для активизации, принятием решения о переключении на новый процесс и другие аналогичные функции выполняются вне микроядра. Этот подход требует тесного взаимодействия между внешним планировщиком и резидентным диспетчером.

Здесь важно сделать различие - запуск процесса или нити требует доступа к аппаратуре, так что по логике - это функция ядра. Но ядру все равно, какую из нитей запускать, поэтому решения о приоритетах нитей и дисциплине постановки в очередь может принимать работающий вне ядра планировщик. Кроме уже представленных соображений, перемещение планировщика на пользовательский уровень может понадобиться для чисто коммерческих целей. Некоторые производители ОС (например, IBM и OSF со своими вариантами микроядра Mach) планируют лицензировать свое микроядро другим поставщикам, которым может потребоваться заменить исходный планировщик на другой, поддерживающий, например, планирование в задачах реального времени или реализующий какой-то специальный алгоритм планирования. А вот другая ОС - Windows NT, также использующая микроядерную концепцию - воплотила понятие приоритетов реального времени в своем планировщике, резидентно расположенном в ядре, и это не дает возможности заменить ее планировщик на другой.

Как и управление процессами, управление памятью может распределяться между микроядром и сервером, работающим в пользовательском режиме. Например, в Workplace OS микроядро управляет аппаратурой

страничной памяти. Пейджер (система управления страничной памятью), работающий вне ядра, определяет стратегию замещения страниц (т.е. решает, какие страницы следует удалить из памяти для размещения страниц, выбранных с диска в ответ на прерывание по отсутствию необходимой страницы), а микроядро выполняет перемещение выбранных пейджером страниц. Как и планировщик процессов, пейджер является заменяемой составной частью.

Драйверы устройств также могут располагаться как внутри ядра, так и вне его. При размещении драйверов устройств вне микроядра для обеспечения возможности разрешения и запрещения прерываний, часть программы драйвера должна исполняться в пространстве ядра. Отделение драйверов устройств от ядра делает возможной динамическую конфигурацию ОС. Кроме динамической конфигурации, есть и другие причины рассматривать драйверы устройств в качестве процессов пользовательского режима. СУБД, например, может иметь свой драйвер, оптимизированный под конкретный вид доступа к диску, но его нельзя будет подключить, если драйверы будут расположены в ядре. Этот подход также способствует переносимости системы, так как функции драйверов устройств могут быть во многих случаях абстрагированы от аппаратной части.

В настоящее время именно операционные системы, построенные с использованием модели клиент-сервер и концепции микроядра, в наибольшей степени удовлетворяют требованиям, предъявляемым к современ-

ным ОС.

Высокая степень переносимости обусловлена тем, что весь машинно-зависимый код изолирован в микроядре, поэтому для переноса системы на новый процессор требуется меньше изменений и все они логически сгруппированы вместе.

Технология микроядер является основой построения множественных прикладных сред, которые обеспечивают совместимость программ, написанных для разных ОС. Абстрагируя интерфейсы прикладных программ от расположенных ниже операционных систем, микроядра позволяют гарантировать, что вложения в прикладные программы не пропадут в течение нескольких лет, даже если будут сменяться операционные системы и процессоры.

Расширяемость также является одним из важных требований к современным операционным системам. Является ли операционная система маленькой, как DOS, или большой, как UNIX, для нее неизбежно настанет необходимость приобрести свойства, не заложенные в ее конструкцию. Увеличивающаяся сложность монолитных операционных систем делала трудным, если вообще возможным, внесение изменений в ОС с гарантией надежности ее последующей работы. Ограниченный набор четко определенных интерфейсов микроядра открывает путь к упорядоченному росту и эволюции ОС.

Обычно операционная система выполняется только в режиме ядра, а прикладные программы - только в режиме пользователя, за исключени-

ем тех случаев, когда они обращаются к ядру за выполнением системных функций. В отличие от обычных систем, система построенная на микроядре, выполняет свои серверные подсистемы в режиме пользователя, как обычные прикладные программы. Такая структура позволяет изменять и добавлять серверы, не влияя на целостность микроядра.

Иногда имеется потребность и в сокращении возможностей ОС. На Windows NT или UNIX перешло бы большее число пользователей, если бы для этих операционных систем не требовалось 16 Мб оперативной памяти и 70 Мб и более пространства на жестком диске. Микроядро не обязательно подразумевает небольшую систему. Надстроенные службы, типа файловой системы и системы управления окнами, добавят к ней немало. Конечно же, не всем нужна безопасность класса C2 или распределенные вычисления. Если важные, но предназначенные для определенных потребителей свойства можно исключать из состава системы, то базовый продукт подойдет более широкому кругу пользователей.

Использование модели клиент-сервер повышает надежность. Каждый сервер выполняется в виде отдельного процесса в своей собственной области памяти, и таким образом защищен от других процессов. Более того, поскольку серверы выполняются в пространстве пользователя, они не имеют непосредственного доступа к аппаратуре и не могут модифицировать память, в которой хранится управляющая программа. И если отдельный сервер может потерпеть крах, то он может быть перезапущен без останова или повреждения остальной части ОС.

Эта модель хорошо подходит для распределенных вычислений, так как отдельные серверы могут работать на разных процессорах мультипроцессорного компьютера или даже на разных компьютерах. При получении от процесса сообщения микроядро может обработать его самостоятельно или переслать другому процессу. Так как микроядру все равно, пришло ли сообщение от локального или удаленного процесса, подобная схема передачи сообщений является элегантным базисом для RPC. Однако такая гибкость не дается даром. Пересылка сообщений не так быстра, как обычные вызовы функций, и ее оптимизация является критическим фактором успеха операционной системы на основе микроядра. Windows NT, например, в некоторых случаях заменяет перенос сообщений на коммуникационные каналы с общей памятью, имеющие более высокую пропускную способность. Хотя это и стоит дороже в смысле потребления фиксированной памяти микроядра, данная альтернатива может помочь сделать модель пересылки сообщений более практичной.

11.2.4 Коммерческие версии микроядер

Одной из первых представила понятие микроядра фирма Next, которая использовала в своих компьютерах систему Mach, прошедшую большой путь развития в университете Карнеги-Меллона при помощи агентства Министерства обороны США DARPA. Теоретически, ее небольшое привилегированное ядро, окруженное службами пользовательско-

го режима, должно было обеспечить беспрецедентную гибкость и модульность. Но на практике это преимущество было несколько уменьшено наличием монолитного сервера операционной системы BSD 4.3, который выполнялся в пользовательском пространстве над микроядром Mach. Однако Mach дал Next возможность предоставить службу передачи сообщений и объектно-ориентированные средства, которые предстали перед конечными пользователями в качестве элегантного интерфейса пользователя с графической поддержкой конфигурирования сети, системного администрирования и разработки программного обеспечения.

Затем пришла Microsoft Windows NT, рекламировавшая в качестве ключевых преимуществ применения микроядра не только модульность, но и переносимость. Конструкция NT позволяет ей работать на системах на основе процессоров Intel, MIPS и Alpha (и последующих), и поддерживать симметричную многопроцессорность. Из-за того, что NT должна была выполнять программы, написанные для DOS, Windows, OS/2 и использующих соглашения POSIX, Microsoft использовала модульность, присущую микроядерному подходу для того, чтобы сделать структуру NT не повторяющей ни одну из существующих операционных систем. Вместо этого NT поддерживает каждую надстроенную операционную систему в виде отдельного модуля или подсистемы.

Более современные архитектуры микроядра были предложены Novell, USL, Open Software Foundation, IBM, Apple и другими. Одним из основных соперников NT на арене микроядер является микроядро Mach 3.0,

которое и IBM и OSF взялись привести к коммерческому виду. (Next в качестве основы для NextStep пока использует Mach 2.5, но при этом внимательно присматривается к Mach 3.0.). Основной соперник Mach - микроядро Chorus 3.0 фирмы Chorus Systems, выбранный USL за основу для своих предложений. Это же микроядро будет использоваться в SpringOS фирмы Sun, объектно-ориентированном преемнике Solaris.

Сегодня стало ясно, что имеется тенденция движения от монолитных систем в сторону подхода с использованием небольших ядер. Именно такой подход уже использовался компаниями QNX Software и Unisys, в течение нескольких лет поставляющих пользующиеся успехом операционные системы на основе микроядра. QNX фирмы QNX Software обслуживает рынок систем реального времени, а CTOS фирмы Unisys популярна в области банковского дела.

11.2.5 Объектно-ориентированный подход

Хотя технология микроядер и заложила основы модульных систем, способных развиваться регулярным образом, она не смогла в полной мере обеспечить возможности расширения систем. В настоящее время этой цели в наибольшей степени соответствует объектно-ориентированный подход, при котором каждый программный компонент является функционально изолированным от других.

Основным понятием этого подхода является "объект". *Объект* – это

единица программ и данных, взаимодействующая с другими объектам посредством приема и передачи сообщений. Объект может быть представлением как некоторых конкретных вещей - прикладной программы или документа, так и некоторых абстракций – процесса, события. Программы (функции) объекта определяют перечень действий, которые могут быть выполнены над данными этого объекта. Объект-клиент может обратиться к другому объекту, послав сообщение с запросом на выполнение какой-либо функции объекта-сервера.

Объекты могут описывать сущности, которые они представляют, с разной степенью детализации. Для обеспечения преемственности при переходе к более детальному описанию разработчикам предлагается механизм наследования свойств уже существующих объектов, то есть механизм, позволяющий порождать более конкретные объекты из более общих. Например, при наличии объекта "текстовый документ"разработчик может легко создать объект "текстовый документ в формате Word 6.0 добавив соответствующее свойство к базовому объекту. Механизм наследования позволяет создать иерархию объектов, в которой каждый объект более низкого уровня приобретает все свойства своего предка.

Внутренняя структура данных объекта скрыта от наблюдения. Нельзя произвольно изменять данные объекта. Для того, чтобы получить данные из объекта или поместить данные в объект, необходимо вызывать соответствующие объектные функции. Это изолирует объект от того кода, который использует его. Разработчик может обращаться к

функциям других объектов, или строить новые объекты путем наследования свойств других объектов, ничего не зная о том, как они сконструированы. Это свойство называется инкапсуляцией.

Таким образом, объект предстает для внешнего мира в виде "черного ящика"с хорошо определенным интерфейсом. С точки зрения разработчика, использующего объект, пока внешняя реакция объекта остается без изменений, не имеют значения никакие изменения во внутренней реализации. Это дает возможность легко заменять одну реализацию объекта другой, например, в случае смены аппаратных средств; при этом сложное программное окружение, в котором находятся заменяемые объекты, не потребует никаких изменений.

С другой стороны, способность объектов представать в виде "черного ящика"позволяет упаковывать в них и представлять в виде объектов уже существующие приложения, ничего в них не изменяя.

Использование объектно-ориентированного подхода особенно эффективно при создании активно развивающегося программного обеспечения, например, при разработке приложений, предназначенных для выполнения на разных аппаратных платформах.

Полностью объектно-ориентированные операционные системы очень привлекательны для системных программистов, так как, используя объекты системного уровня, программисты смогут залезать вглубь операционных систем для приспособления их к своим нуждам, не нарушая целостность системы.

Но особенно большие перспективы имеет этот подход в реализации распределенных вычислительных сред. В то время, как сейчас разные пакеты, работающие в данный момент в сети, представляют собой статически связанные наборы программ, в будущем, с использованием объектно-ориентированного подхода, они могут превратиться в единую совокупность динамически связываемых объектов, где каждый объект оперативно устанавливает и разрывает связи с другими объектами для выполнения актуальных в данный момент задач. Приложения, созданные для такой сетевой среды, основанной на объектах, могут выполняться, динамически обращаясь к множеству объектов, независимо от их местонахождения в сети и независимо от их операционной среды.

Поскольку любое объектно-ориентированное приложение представляет собой набор объектов, разработчику желательно иметь стандартные средства для управления объектами и организации их взаимодействия. При использовании и разработке объектно-ориентированных приложений в неоднородных распределенных средах, нужны также средства, упрощающие доступ к объектам сети. При возникновении запроса к какому-либо объекту распределенной среды, независимо от того, находится требуемый объект на том же компьютере или на одном из удаленных, прозрачным образом должен быть выполнен поиск объекта, передача ему сообщения, и возврат ответа. Для обеспечения прозрачного обнаружения объектов, все они должны быть снабжены ссылками, хранящимися в каталогах. Отсюда вытекает очень сложная проблема орга-

низации службы каталогов, позволяющей программистам именовать и искать объекты в сети, которая, вообще говоря, может быть разбросана по всему миру.

Однако, несмотря на упомянутые сложности и проблемы, объектно-ориентированный подход является одной из самых перспективных тенденций в конструировании программного обеспечения.

11.2.6 Коммерческие объектно-ориентированные средства

Объектно-ориентированный подход к построению операционных систем, придающий порядок процессу добавления модульных расширений к небольшому ядру был принят на вооружение многими известными фирмами, такими как Microsoft, Apple, IBM, Novell/USL (UNIX Systems Laboratories) и Sun Microsystems - все они развернули свои операционные системы в этом направлении. Taligent, совместное предприятие IBM и Apple, надеется опередить всех со своей от начала до конца объектно-ориентированной операционной системой. Тем временем Next поставляет Motorola- и Intel-версии NextStep, наиболее продвинутой объектно-ориентированной операционной системы из имеющихся. Хотя NextStep и не имеет объектной ориентированности сверху донизу, как это планируется в разработках Taligent, но она доступна уже сегодня.

Одним из первых применений объектных систем для большинства пользователей станут основанные на объектах прикладные программы.

К объектно-ориентированным технологиям этого уровня, уже имеющимся сейчас или доступным в ближайшем будущем относятся:

- Microsoft OLE (Object Linking and Embedding - связывание и внедрение объектов),
- стандарт OpenDoc от фирм Apple, IBM, WordPerfect, Novell и Borland,
- DSOM (Distributed System Object Model - объектная модель распределенных систем) фирмы IBM,
- PDO (Portable Distributed Objects - переносимые распределенные объекты) фирмы Next,
- каркасы (frameworks) фирмы Taligent,
- архитектура CORBA объединения OMG.

Средства OLE

Для пользователей Windows объектно-ориентированный подход проявляется при работе с программами, использующими технологию OLE фирмы Microsoft. В первой версии OLE, которая дебютировала в Windows 3.1, пользователи могли вставлять объекты в документы-клиенты. Такие объекты устанавливали ссылку на данные (в случае связывания)

или содержали данные (в случае внедрения) в формате, распознаваемом программой-сервером. Для запуска программы-сервера пользователи делали двойной щелчок на объекте, посредством чего передавали данные серверу для редактирования. OLE 2.0, доступная в настоящее время в качестве расширения Windows 3.1, переопределяет документ-клиент как контейнер. Когда пользователь щелкает дважды над объектом OLE 2.0, вставленным в документ-контейнер, он активизируется в том же самом месте. Представим, например, что контейнером является документ Microsoft Word 6.0, а вставленный объект представляет собой набор ячеек в формате Excel 5.0. Когда вы щелкнете дважды над объектом электронной таблицы, меню и управляющие элементы Word как по волшебству поменяются на меню Excel. В результате, пока объект электронной таблицы находится в фокусе, текстовый процессор становится электронной таблицей.

Инфраструктура, требуемая для обеспечения столь сложных взаимодействий объектов, настолько обширна, что Microsoft называет OLE 2.0 "1/3 операционной системы". Хранение объектов, например, использует docfile, который в действительности является миниатюрной файловой системой, содержащейся внутри обычного файла MS-DOS. Docfile имеет свои собственные внутренние механизмы для семантики подкаталогов, блокировок и транзакций (т.е. фиксации-отката).

Наиболее заметный недостаток OLE - отсутствие сетевой поддержки, и это будет иметь наивысший приоритет при разработке будущих версий

OLE. Следующая основная итерация OLE появится в распределенной, объектной версии Windows, называемой Cairo (Каир), ожидаемой в 1995 году.

Стандарт OpenDoc

Apple, совместно с WordPerfect, Novell, Sun, Xerox, Oracle, IBM и Taligent, известными вместе как Component Integration Laboratory (Лаборатория по объединению компонентов), также занимается архитектурой объектно-ориентированных составных документов, называемой OpenDoc. Создаваемый для работы на разных платформах, этот проект значительно отстает по степени готовности от OLE 2.0.

Ключевыми технологиями OpenDoc являются механизм хранения Бенто (названный так в честь японской тарелки с отделениями для разной пищи), технология сценариев (scripting), позаимствованная в значительной степени из AppleSript, и SOM фирмы IBM. В Бенто-документе каждый объект имеет постоянный идентификатор, перемещающийся вместе с ним от системы к системе. Хранение не только является транзакционным, как в OLE, но и позволяет держать и отслеживать многочисленные редакции каждого объекта. Если имеется несколько редакций документа, то реально храниться будут только изменения от одной редакции к другой. Верхняя граница числа сохраняемых редакций будет определяться пользователем.

Команда Apple планирует сделать OpenDoc совместимым с Microsoft OLE. Если план завершится успехом, система OpenDoc сможет окружать объекты OLE слоем программ трансляции сообщений. Контейнер OpenDoc будет видеть встроенный объект OLE как объект OpenDoc, а объект OLE будет представлять свой контейнер, как контейнер OLE. Утверждается, что будет допустима также и обратная трансляция по этому сценарию, когда объекты OpenDoc функционируют в контейнерах OLE. Слой трансляции разрабатывается WordPerfect при помощи Borland, Claris, Lotus и других.

В основе OLE и OpenDoc лежат две соперничающие объектные модели: Microsoft COM (Component Object Model - компонентная объектная модель) и IBM SOM. Каждая определяет протоколы, используемые объектами для взаимодействия друг с другом. Основное их различие заключается в том, что SOM нейтральна к языкам программирования и поддерживает наследование, тогда как COM ориентирована на C++ и вместо механизма наследования использует альтернативный механизм, который Microsoft называет агрегацией.

Семейство CORBA

Hewlett-Packard, Sun Microsystems и DEC экспериментируют с объектами уже много лет. Теперь эти компании и много других объединились вместе, основав промышленную коалицию под названием OMG (Object Management Group), разрабатывающую стандарты для обмена

объектами. OMG CORBA (Common Object Request Broker Architecture - Общая архитектура посредника обработки объектных запросов) закладывает фундамент распределенных вычислений с переносимыми объектами. CORBA задает способ поиска объектами других объектов и вызова их методов. SOM согласуется с CORBA. Если вы пользуетесь DSOM под OS/2, вы сможете вызывать CORBA-совместимые объекты, работающие на HP, Sun и других архитектурах. Означает ли это возможность редактировать объект OpenDoc, сделанный на Macintosh, в документе-контейнере на RISC-рабочей станции? Вероятно, нет. CORBA может гарантировать только механизм нижнего уровня, посредством которого одни объекты вызывают другие. Для успешного взаимодействия требуется также понимание сообщений друг друга.

11.2.7 Множественные прикладные среды

В то время как некоторые идеи (например, объектно-ориентированный подход) непосредственно касаются только разработчиков и лишь косвенно влияют на конечного пользователя, концепция множественных прикладных сред приносит пользователю долгожданную возможность выполнять на своей ОС программы, написанные для других операционных систем и других процессоров.

И сейчас дополнительное программное обеспечение позволяет пользователям некоторых ОС запускать чужие программы (например, Mac

и UNIX позволяют выполнять программы для DOS и Windows). Но в зарождающемся поколении операционных систем средства для выполнения чужих программ становятся стандартной частью системы. Выбор операционной системы больше не будет сильно ограничивать выбор прикладных программ. Хотя столкновение пользовательских интерфейсов программ для Mac, Windows и UNIX на одном и том же экране и заставит пользователя немного потрудиться, но все равно, множественные прикладные среды операционных систем скоро станут такими же стандартными, как мыши и меню.

Множественные прикладные среды обеспечивают совместимость данной ОС с приложениями, написанными для других ОС и процессоров, на двоичном уровне, а не на уровне исходных текстов. Для пользователя, купившего в свое время пакет (например, Lotus 1-2-3) для MS DOS, важно, чтобы он мог запускать этот полюбившийся ему пакет без каких-либо изменений и на своей новой машине, построенной, например, на RISC-процессоре, и работающей под управлением, например, Windows NT.

При реализации множественных прикладных сред разработчики сталкиваются с противоречивыми требованиями. С одной стороны, задачей каждой прикладной среды является выполнение программы по возможности так, как если бы она выполнялась на "родной"ОС. Но потребности этих программ могут входить в конфликт с конструкцией современной операционной системы. Специализированные драйверы устройств могут

противоречить требованиям безопасности. Могут конфликтовать схемы управления памятью и оконные системы. Чисто экономические вопросы (например, стоимость лицензирования программ и угроза судебного преследования) также могут повлиять на дизайн чужих прикладных сред. Но самой большой потенциальной проблемой является производительность – прикладная среда должна выполнять программы с приемлемой скоростью.

Этому требованию не могут удовлетворить широко используемые ранее эмулирующие системы. Для сокращения времени на выполнение чужих программ прикладные среды используют имитацию программ на уровне библиотек. Эффективность этого подхода связана с тем, что большинство сегодняшних программ работают под управлением GUI (графических интерфейсов пользователя) типа Windows, Mac или UNIX Motif, при этом приложения тратят большую часть времени, производя некоторые хорошо предсказуемые вещи. Они непрерывно выполняют вызовы библиотек GUI для манипулирования окнами и для других связанных с GUI действий. И это то, что позволяет прикладным средам возместить время, потраченное на эмулирование команды за командой. Тщательно сделанная прикладная среда имеет в своем составе библиотеки, имитирующие внутренние библиотеки GUI, но написанные на родном коде, то есть она совместима с программным интерфейсом другой ОС. Иногда такой подход называют трансляцией для того, чтобы отличать его от более медленного процесса эмулирования кода по одной

команде за раз.

Например, для Windows-программы, работающей на Mac, при интерпретировании команд 80x86 производительность может быть очень низкой. Но когда производится вызов функции открытия окна, модуль прикладной среды может переключить его на перекомпилированную для 680x0 подпрограмму открытия окна. Так как библиотекам GUI не нужно дешифрировать и имитировать каждую команду, то в частях программы, относящихся к вызовам GUI ABI (Application Binary Interface - двоичный интерфейс прикладного программирования), производительность может резко вырасти. В результате на таких участках кода скорость работы программы может достичь (а возможно, и превзойти) скорость работы на своем родном процессоре.

Сегодня в типичных программах значительная часть кода занята вызовом GUI ABI. Apple утверждает, что программы для Mac тратят до 90 процентов процессорного времени на выполнение подпрограмм из Mac toolbox, а не на уникальные для этих программ действия. SunSelect говорит, что программы для Windows тратят от 60 до 80 процентов времени на работу в ядре Windows. В результате при эмуляции программы на основе GUI потери производительности могут быть значительно меньше. SunSelect заявляет, что его новая прикладная среда Windows, WABI (Windows Application Binary Interface - двоичный интерфейс прикладных программ Windows), благодаря сильно оптимизированным библиотекам, на некоторых платформах при исполнении одних и тех же тестов

может обогнать настоящий Microsoft Windows.

С позиции использования прикладных сред более предпочтительным является способ написания программ, при котором программист для выполнения некоторой функции обращается с вызовом к операционной системе, а не пытается более эффективно реализовать эквивалентную функцию самостоятельно, работая напрямую с аппаратурой. Отбить у программистов охоту "обращаться к металлу"сможет наличие в библиотеках мощных и сложных программ, к которым гораздо проще обращаться, чем писать самому.

Модульность операционных систем нового поколения позволяет намного легче реализовать поддержку множественных прикладных сред. В отличие от старых операционных систем, состоящих из одного большого блока для всех практических применений, разбитого произвольным образом на части, новые системы являются модульными, с четко определенными интерфейсами между составляющими. Это делает создание дополнительных модулей, объединяющих эмуляцию процессора и трансляцию библиотек, значительно более простым.

К усовершенствованным операционным системам, явно содержащим средства множественных прикладных сред, относятся: IBM OS/2 2.x и Workplace OS, Microsoft Windows NT, PowerOpen компании PowerOpen Association и версии UNIX от Sun Microsystems, IBM и Hewlett-Packard. Кроме того, некоторые компании переделывают свои интерфейсы пользователя в виде модулей прикладных сред, а другие поставщики пред-

лагают продукты для эмуляции и трансляции прикладных сред, работающие в качестве прикладных программ.

Существует много разных стратегий по воплощению идеи множественных прикладных сред, и некоторые из этих стратегий диаметрально противоположны. В случае UNIX, транслятор прикладных сред обычно делается, как и другие прикладные программы, плавающим на поверхности операционной системы. В более современных операционных системах типа Windows NT или Workplace OS модули прикладной среды выполняются более тесно связанными с операционной системой, хотя и обладают по-прежнему высокой независимостью. А в OS/2 с ее более простой, слабо структурированной архитектурой средства организации прикладных сред встроены глубоко в операционную систему.

Использование множественных прикладных сред обеспечит пользователям большую свободу выбора операционных систем и более легкий доступ к более качественному программному обеспечению.

Сетевой пакет DCE фирмы OSF

Распределенные вычисления имеют дело с понятиями более высокого уровня, чем физические носители, каналы связи и методы передачи по ним сообщений. Распределенная среда должна дать пользователям и приложениям прозрачный доступ к данным, вычислениям и другим ресурсам в гетерогенных системах, представляющих собой набор средств различных производителей. Стратегические архитектуры каж-

дого крупного системного производителя базируются сейчас на той или иной форме распределенной вычислительной среды (DCE). Ключом к пониманию выгоды такой архитектуры является прозрачность. Пользователи не должны тратить свое время на попытки выяснить, где находится тот или иной ресурс, а разработчики не должны писать коды для своих приложений, использующие местоположение в сети. Никто не заинтересован в том, чтобы заставлять разработчиков приложений становиться гуру коммуникаций, или же в том, чтобы заставлять пользователей заботиться о монтировании удаленных томов. Кроме того, сеть должна быть управляемой. Окончательной картиной является виртуальная сеть: набор сетей рабочих групп, отделов, предприятий, объединенных сетей предприятий, которые кажутся конечному пользователю или приложению единой сетью с простым доступом.

Одной из технологий, которая будет играть большое значение для будущего распределенных вычислений является технология DCE (Distributed Computing Environment) некоммерческой организации

Open Software foundation (OSF). DCE OSF - это интегрированный набор функций, независимых от операционной системы и сетевых средств, которые обеспечивают разработку, использование и управление распределенными приложениями. Из-за их способности обеспечивать управляемую, прозрачную и взаимосвязанную работу систем различных производителей и различных платформ, DCE является одной из самых важных технологий десятилетия.

Большинство из ведущих фирм-производителей ОС договорились о поставках DCE в будущих версиях своих системных и сетевых продуктов. Например, IBM, которая предлагает несколько DCE-продуктов, базирующихся на AIX, распространила их и на сектор сетей персональных компьютеров. В сентябре 1993 года IBM начала поставлять инструментальные средства для разработки DCE-приложений - DCE SDK для OS/2 и Windows, и в это же время выпустила на рынок свой первый DCE-продукт для пользователей персональных компьютеров - DCE Client for OS/2. Компания IBM достаточно далеко продвинулась в реализации спецификации DCE в своих продуктах, обогнав Microsoft, Novell и Banyan. Сейчас она продает как отдельный продукт клиентскую часть служб DCE для операционных сред MVS, OS/400, OS/2, AIX и Windows. IBM собирается реализовать отсутствующую в LAN Server единую справочную службу и новую службу безопасности в соответствии со спецификацией DCE и выпустить интегрированный вариант DCE/LAN Server в конце этого года.

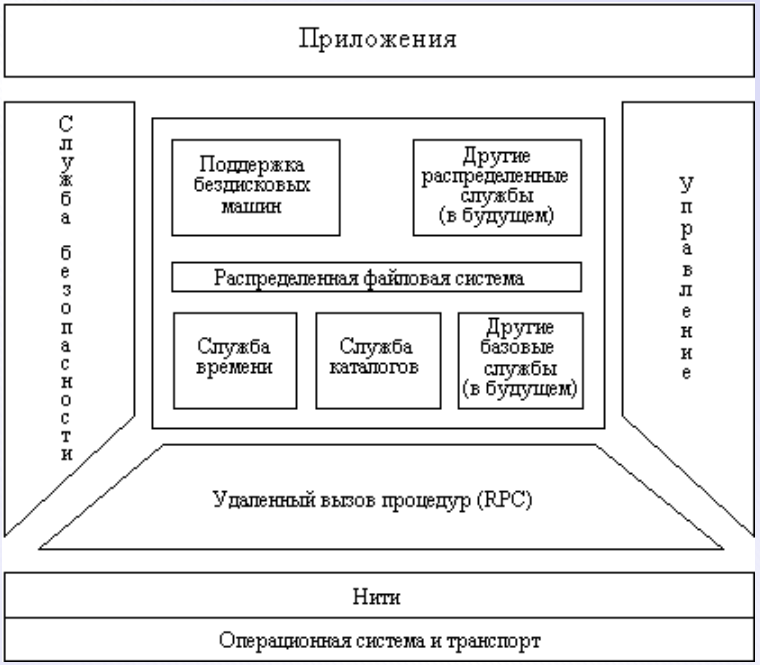


Рис. 11.4: Архитектура средств OSF DCE

Некоторые крупные фирмы-потребители программных продуктов обращаются непосредственно в OSF за лицензиями на первые версии DCE OSF.

В настоящее время DCE состоит из 7 средств и функций, которые делятся на базовые распределенные функции и средства разделения данных. Базовые распределенные функции включают:

1. нити,
2. RPC,
3. службу каталогов,
4. службу времени,
5. службу безопасности.

Функции разделения данных строятся над базовыми функциями и включают:

1. распределенную файловую систему (DFS),
2. поддержку бездисковых машин.

На рис. 11.4 показана архитектура DCE OSF. Эта архитектура представляет собой многоуровневый набор интегрированных средств. Внизу расположены базисные службы, такие как ОС, а на самом верхнем уровне находятся потребители средств DCE приложения. Средства безопасности и управления пронизывают все уровни. OSF резервирует место для функций, которые могут появиться в будущем, таких как спулинг, поддержка транзакций и распределенная объектно-ориентированная среда.

Обычно приложения имеют дело с процессами, каждый из которых состоит из одной нити управления.

Нити – это важная модель для выражения параллелизма внутри процесса, особенно для распределенного окружения. Например, возможности многонитевой обработки становятся особенно важными в контексте RPC. RPC является синхронным механизмом по своей природе: клиент делает вызов удаленной функции и ожидает выполнения вызова. При использовании нитей одна нить может сделать запрос, а другая начать обрабатывать данные от другого запроса. Следовательно, использование нитей может существенно улучшить производительность распределенного приложения. Нитевая модель предъявляет меньшие требования к искусству программиста, чем другие альтернативы параллелизма, такие как асинхронные операции или разделение памяти.

Поскольку поддержка нитей дает большой выигрыш в производительности, большинство современных операционных систем является многонитевыми. Однако многие используемые в настоящее время операционные системы таковыми не являются. DCE OSF предлагает специальный пакет, предназначенный для обеспечения многонитевости для организации распределенных приложений в ОС, не имеющих собственных встроенных средств поддержки многонитевости. Этот пакет выполнен как библиотека функций работы с нитями.

По сравнению со встроенными в ядро ОС функциями поддержки нитей, библиотечная их реализация имеет некоторые функциональные ограничения. С другой стороны, в этом случае предоставляются хорошие шансы для обеспечения совместимости свойств нитей в гетерогенных средах.

Все остальные службы DCE OSF - RPC, службы безопасности, каталогов и времени, распределенная файловая система - используют сервис этого пакета.

Рассмотрим пакет, реализующий нити в DCE OSF, более подробно. Как и большинство программного обеспечения DCE, этот пакет большой и сложный. Он состоит из 51 библиотечной функции, относящейся к нитям, которыми могут пользоваться прикладные программы. Многие из них не являются необходимыми, а разработаны только для удобства. Мы рассмотрим только основные.

Эти функции удобно разделить на 7 категорий, каждая из которых имеет дело с определенным аспектом работы с нитями. Первая категория имеет дело с управлением нитями. В нее входят вызовы:

- CREATE - создает новую нить
- EXIT - вызывается нитью при завершении
- JOIN - подобен системному вызову WAIT в UNIX
- DETACH - отсоединение нити-потомка

Эти вызовы позволяют создавать и завершать нити. Родительская нить может ждать потомка, используя вызов join, подобно вызову wait в UNIX'е. Если родительская нить не планирует этого, то она может отсоединиться от потомка, сделав вызов detach. В этом случае, когда нить-потомок завершается, ее память освобождается немедленно, вместо обычного ожидания родителя, который издал вызов join.

Вторая категория функций позволяет пользователю создавать, разрушать и управлять шаблонами для нитей, для семафоров (в данном случае имеются ввиду бинарные семафоры, имеющие два состояния, называемые мьютексами). Шаблоны могут использоваться с соответствующими начальными условиями. При создании объекта один из параметров вызова create является указателем на шаблон. Например, шаблон нити имеет по умолчанию размер стека 8 K. Все нити, созданные с помощью этого шаблона имеют размер стека 8 K. Смысл использования шаблонов в том, что они позволяют не задавать все параметры создаваемого объекта.

Существуют системные вызовы добавления новых атрибутов к шаблонам. Вызовы attr_create и attr_delete создают и удаляют шаблоны нитей. Другие вызовы позволяют программе читать и модифицировать атрибуты шаблона, такие как размер стека или параметры планирования. Имеются также шаблоны для создания и удаления семафоров.

Третья группа вызовов содержит пять функций для управления мьютексами: создание, удаление, блокирование, разблокирование.

При работе с семафорами возникает очевидный вопрос, что случится, если использовать операцию "разблокировать"по отношению к мьютексу, который не заблокирован. Сохранится ли эта операция или будет потеряна? Эта проблема и вынудила Дейкстру ввести в свое время семафоры, при использовании которых порядок выполнения операций блокирования и разблокирования не имеет значения. В результате не возникает эффекта гонок. Логика работы мьютекса в DCE зависит от реализации, что, конечно, не позволяет писать переносимые программы.

Существует два типа мьютексов в DCE - быстрые и дружественные. Быстрый мьютекс является аналогом блокировки в СУБД. Если процесс пытается заблокировать незаблокированную запись, то эта операция выполняется успешно. Однако, если он попытается применить эту блокировку второй раз, то он сам заблокируется, ожидая, пока кто-нибудь не снимет блокировку с мьютекса.

Дружественный мьютекс позволяет нити заблокировать уже заблокированный мьютекс. Предположим, что главная программа блокирует семафор, а затем вызывает функцию, которая также блокирует мьютекс. Чтобы избежать клинча, принимается и вторая блокировка. Если количество открытий и закрытий мьютекса совпадает, связанные операции блокирования-разблокирования могут иметь произвольную глубину вложения. Разработчики пакета, очевидно не пришли к согласию относительно того, что делать с повторной блокировкой от той же нити, и решили реализовать обе версии.

Хорошо известный механизм для реализации распределенных вычислений, RPC, расширяет традиционную модель программирования - вызов процедуры - для использования в сети. RPC может составлять основу распределенных вычислений. Теоретически программист не должен становиться экспертом по коммуникационным средствам для того, чтобы написать распределенное сетевое приложение. При использовании RPC программист будет применять специальный язык для описания операций. Данное описание обрабатывается компилятором, который создает код как для клиента, так и для сервера. Для обеспечения такого типа функций средства RPC должны быть простыми, прозрачными, надежными и высокопроизводительными.

Средства RPC пакета DCE OSF обладают простотой. Они приближаются к модели вызова локальных процедур насколько это возможно. Они почти не требуют изменения концептуального мышления программиста, и поэтому уменьшают время его переподготовки. Это особенно важно для коллективов программистов, работающих в фирмах, которые не являются производителями коммерческих программных продуктов.

Неизменность протокола - это другая особенность RPC DCE OSF. Этот протокол четко определен и не может изменяться пользователем (в данном случае разработчиком сетевых приложений). Такая неизменность, гарантируемая ядром, является важным свойством в гетероген-

ных средах, требующих согласованной работы. В отличие от OSF, некоторые другие разработчики средств RPC полагают, что гибкость и возможность приспосабливать эти средства к потребностям пользователей являются более важными.

Средства RPC DCE OSF поддерживают ряд транспортных протоколов и позволяют добавлять новые транспортные протоколы, сохраняя при этом свои функциональные свойства.

DCE RPC эффективно используется в других службах DCE: в службах безопасности, каталогов и времени, в распределенной файловой системе. DCE RPC интегрируется с системой идентификации для обеспечения защиты доступа и с нитями клиента и сервера для того, чтобы, сохраняя синхронный характер выполнения вызова, обеспечить параллелизм. Способность RPC посылать и получать потоки типизированных данных неопределенной длины используется в распределенной файловой системе.

Распределенная служба каталогов

Задачей службы каталогов в распределенной сети является поиск сетевых объектов, то есть пользователей, ресурсов, данных или приложений. Служба каталогов (или иначе, имен) должна отобразить большое количество системных объектов (пользователей, организаций, групп, компьютеров, принтеров, файлов, процессов, сервисов) на множество понятных пользователю имен. Эта проблема сложна даже для гомогенных

сетей, так как персонал и оборудование перемещается, изменяет свои имена, местонахождение и т.д. В гетерогенных глобальных сетях служба каталогов становится намного более сложной, из-за необходимости синхронизации различных баз данных каталогов. Более того, при появлении в сети распределенных приложений служба каталогов должна начать отслеживание всех таких объектов и всех их компонентов.

Хорошая служба каталогов делает использование распределенного окружения прозрачным для пользователя. Пользователям не нужно знать расположение удаленного принтера, файла или приложения.

OSF определяет двухярусную архитектуру для службы каталогов для целей адресации межячеечного и глобального взаимодействия. Ячейка (cell) - это фундаментальная организационная единица для систем в DCE OSF. Ячейки могут иметь социальные, политические или организационные границы. Ячейки состоят из компьютеров, которые должны часто взаимодействовать друг с другом - это могут быть, например, рабочие группы, отделы, или отделения компаний. В общем случае компьютеры в ячейке географически близки. Размер ячеек изменяется от 2 до 1000 компьютеров, хотя OSF считает наиболее приемлемым диапазон от десятков до сотен компьютеров.

Некоторые производители и пользователи агитируют за реализацию X.500 как общей службы каталогов на всех уровнях. Но OSF полагает, что использование X.500 на уровне рабочей группы (то есть ячейки) было бы слишком громоздким из-за требований к программному обес-

печению по производительности - особенно, когда более гибкие средства службы каталогов уровня ячейки уже существуют на рынке.

Служба каталогов DCE состоит из 4-х элементов:

- CDS (Cell Directory Service) - служба каталогов ячейки. Ячейка сети - это группа систем, администрируемых как единое целое. CDS оптимизируется для локального доступа. Большинство запросов к службе каталогов относятся к ресурсам той же ячейки. Каждая ячейка сети нуждается по крайней мере в одной CDS.

- GDA (Global Directory Agent) - агент глобального каталога. GDA - это шлюз имен, который соединяет домен DCE с другими административными доменами посредством глобальной службы каталогов X.500 и DNS (domain name service - сервис имен домена). GDA передает запрос на имя, которое он не смог найти в локальной ячейке, службе каталогов другой ячейки, или глобальной службе каталогов (в зависимости от места хранения имени). Для того, чтобы отыскать имя, клиент делает запрос к локальному агенту GDA. Затем GDA передает запрос на междоменное имя службе X.500. Эта служба возвращает ответ GDA, который в свою очередь передает его клиенту. OSF GDA может быть совместим с любой схемой глобального именования.

- GDS (Global Directory Service) - глобальная служба каталогов. Основанная на стандарте X.500, GDS работает на самом верхнем уровне

иерархии и обеспечивает связь множества ячеек в множестве организаций.

- XDS (X/Open Directory Service) - обеспечивает поддержку X/Open API функций службы каталогов и позволяет разработчикам писать приложения, независимые от нижележащих уровней архитектуры службы каталогов. XDS-совместимые приложения будут работать одинаковым образом со службами каталогов DCE и X.500.

Распределенная служба безопасности

Имеется две больших группы функций службы безопасности: идентификация и авторизация. Идентификация проверяет идентичность объекта (например, пользователя или сервиса). Авторизация (или управление доступом) назначает привилегии объекту, такие как доступ к файлу.

Авторизация - это только часть решения. В распределенной сетевой среде обязательно должна работать глобальная служба идентификации, так как рабочей станции нельзя доверить функции идентификации себя или своего пользователя. Служба идентификации представляет собой механизм передачи третьей стороне функций проверки идентичности пользователя.

Служба безопасности OSF DCE базируется на системе идентификации Kerberos, разработанной в 80-е годы и расширенной за счет добавления элементов безопасности. Kerberos использует шифрование, осно-

ванное на личных ключах, для обеспечения трех уровней защиты. Самый нижний уровень требует, чтобы пользователь идентифицировался только при установлении начального соединения, предполагая, что дальнейшая последовательность сетевых сообщений исходит от идентифицированного пользователя. Следующий уровень требует идентификацию для каждого сетевого сообщения. На последнем уровне все сообщения не только идентифицируются, но и шифруются.

Система безопасности не должна сильно усложнять жизнь конечного пользователя в сети, то есть он не должен запоминать десятки паролей и кодов.

Весьма полезным сетевым средством для целей безопасности является служба прав доступа или, другими словами, авторизация. Служба авторизации OSF базируется на POSIX-совместимых списках прав доступа - ACL.

В то время как система Kerberos основана на личных ключах, в настоящее время широкое распространение получили методы, основанные на публичных ключах (например, метод RSA). OSF собирается сделать DCE-приложения переносимыми из Kerberos в RSA.

Распределенная файловая система DFS OSF

Распределенная файловая система DFS OSF предназначена для обеспечения прозрачного доступа к любому файлу, расположенному в любом узле сети. Главная концепция такой распределенной файловой системы

- это простота ее использования.

Распределенная файловая система должна иметь единое пространство имен. Файл должен иметь одинаковое имя независимо от того, где он расположен. Другими желательными свойствами являются интегрированная безопасность, согласованность и доступность данных, надежность и восстанавливаемость, производительность и масштабируемость до очень больших конфигураций без уменьшения производительности и независимое от места расположения управление и администрирование. Распределенная файловая система DFS OSF базируется на известной файловой системе AFS (The Andrew File System).

Файловая система AFS

AFS была разработана в университете Карнеги-Меллона и названа в честь спонсоров-основателей университета Andrew Carnegie и Andrew Mellon. Эта система, созданная для студентов университета, не является прозрачной системой, в которой все ресурсы динамически назначаются всем пользователям при возникновении потребностей. Несмотря на это, файловая система была спроектирована так, чтобы обеспечить прозрачность доступа каждому пользователю, независимо от того, какой рабочей станцией он пользуется.

Особенностью этой файловой системы является возможность работы с большим (до 10 000) числом рабочих станций.

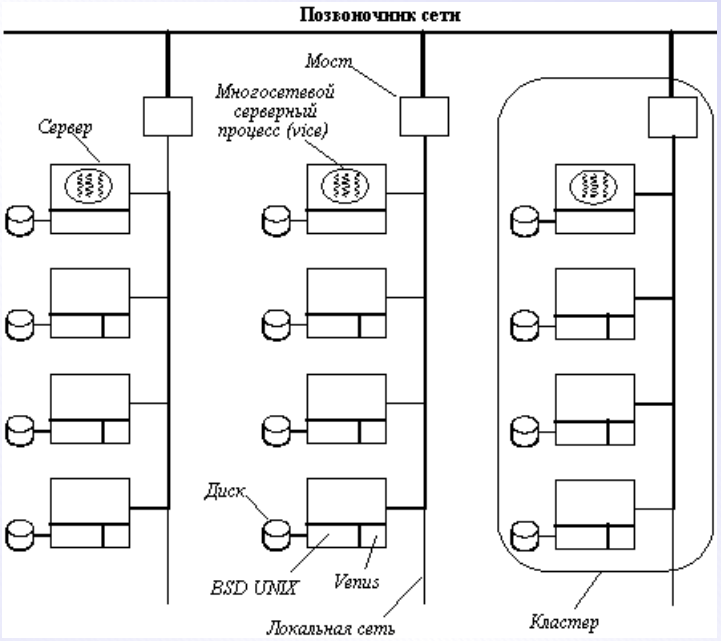


Рис. 11.5: Конфигурация системы, используемая AFS в университете Карнеги-Меллона

Конфигурация системы показана на Рис. 11.5. Она состоит из кластеров, каждый из которых включает файловый сервер и несколько десятков рабочих станций. Идея состоит в том, чтобы распределить большую часть трафика в пределах отдельных кластеров и тем самым уменьшить загрузку позвоночника сети.

Так как студенты могли входить в систему и в университете, и в обще-

житии, то иногда они оказывались далеко от сервера, который содержал их файлы. Несмотря на это, пользователь должен иметь возможность работать на произвольно выбранной рабочей станции, как на своем персональном компьютере.

Физически нет разницы между машиной клиента и сервера, и все они выполняют одну и ту же ОС BSD UNIX с его большим монолитным ядром. Однако над ядром выполняются совершенно различные программы серверов и клиентов. На клиент-машинах выполняются менеджеры окон, редакторы и другое стандартное программное обеспечение системы UNIX. Каждый клиент имеет также часть кода - venus, которая управляет интерфейсом между клиентом и серверной частью системы, называемой vice. Вначале venus выполнялся в пользовательском режиме, но позже он был перемещен в ядро для повышения производительности. Venus работает также в качестве менеджера кэша. В дальнейшем мы будем называть venus просто клиентом, а vice - сервером.

Пространство имен, видимое пользовательскими программами, выглядит как традиционное дерево в ОС UNIX с добавленным к нему каталогом /cmu (рисунок 4.5). Содержимое каталога /cmu поддерживается AFS посредством vice-серверов и идентично на всех рабочих станциях. Другие каталоги и файлы исключительно локальны и не разделяются. Возможности разделяемой файловой системы предоставляются путем монтирования к каталогу /cmu. Файл, который UNIX ожидает найти в верхней части файловой системы, может быть перемещен символь-

ной связью из разделяемой файловой системы (например, /bin/sh может быть символьно связан с /cmu/bin/sh).

В основе AFS лежит стремление делать для каждого пользователя как можно больше на его рабочей станции и как можно меньше взаимодействовать с остальной системой. При открытии удаленного файла весь файл (или его значительная часть, если он очень большой) загружается на диск рабочей станции и кэшируется там, причем процесс, который сделал вызов OPEN, даже не знает об этом. По этой причине каждая рабочая станция имеет диск.

После загрузки файла на локальный диск он помещается в локальный каталог /cash, так что он выглядит для ОС как нормальный файл. Дескриптор файла, возвращаемый системным вызовом OPEN, соответствует именно этому файлу, так что вызовы READ и WRITE работают обычным путем, без участия клиента и сервера. Другими словами, хотя системный вызов OPEN значительно изменен, реализация READ и WRITE не изменилась.

Безопасность – это главный вопрос в системе с 10 000 пользователей. Так как пользователи вольны перегружать свои рабочие станции, когда захотят и могут выполнять на них модифицированные версии ОС, то главный принцип сервера - не доверять клиентским рабочим станциям. Все сообщения между рабочими станциями шифруются на уровне аппаратуры. Защита выполнена несколько необычным путем. Каталоги защищаются списками прав доступа (ACL), но файлы имеют обычные би-

ты RWX UNIX'а. Разработчики системы предпочитают механизм ACL, но так как многие UNIX-программы работают с битами RWX, то они оставлены для совместимости. Списки прав доступа могут содержать и отсутствие прав, так что можно, например, потребовать, чтобы доступ к файлу был разрешен для всех, кроме одного конкретного человека.

Диски рабочих станций используются только для временных файлов, кэширования удаленных файлов и хранения страниц виртуальной памяти, но не для постоянной информации. Это существенно упрощает управление системой, в этом случае нужно управлять и архивировать только файлы серверов, а рабочие станции не требуют никаких забот. Концептуально они могут начинать каждый рабочий день с чистого листа.

AFS сконструирована для расширения до масштабов национальной файловой системы. Система, показанная на рисунке, на самом деле представляет собой отдельную ячейку (cell). Каждая ячейка - это административная единица, такая как отдел или компания. Ячейки могут быть соединены друг с другом с помощью монтирования, так что дерево разделяемых файлов может покрывать многие города.

В дополнение к концепциям файла, каталога и ячейки AFS поддерживает еще одно важное понятие - том. Том - это собрание каталогов, которые управляются вместе. Обычно все файлы какого-либо пользователя составляют том. Таким образом поддерево, входящее в /usr/john, может быть одним томом, поддерево, входящее в /usr/mary, может быть дру-

гим томом. Фактически, каждая ячейка представляет собой ничто иное, как набор томов, соединенных вместе некоторым, подходящим с точки зрения монтирования, образом. Большинство томов содержат пользовательские файлы, некоторые другие используются для двоичных выполняемых файлов и другой системной информации. Тома могут иметь признак "только для чтения".

Семантика, предлагаемая AFS, близка к сессионной семантике. Когда файл открывается, он берется у подходящего сервера и помещается в каталог /cash на локальном диске на рабочей станции. Все операции чтения-записи работают с кэшированной копией. При закрытии файла он выгружается назад на сервер. Следствием этой модели является то, что когда процесс открывает уже открытый файл, то версия, которую он видит, зависит от того, где находится процесс. Процесс на той же рабочей станции видит копию в каталоге /cash, при этом выполняется вся семантика UNIX.

В то же время процесс на другой рабочей станции продолжает видеть исходную версию файла на сервере. Только после того, как файл будет закрыт и отослан обратно на сервер, последующая операция открытия увидит новую версию. После того, как файл закрывается, он остается в кэше, на случай, если он скоро будет снова открыт. Как мы видели ранее, повторное открытие файла, который находится в кэше, порождает проблему: "Как клиент узнает, последняя ли это версия файла?"В первой версии AFS эта проблема решалась прямым запросом клиен-

та к серверу. К сожалению эти запросы создавали большой трафик и впоследствии алгоритм был изменен. В новом алгоритме, когда клиент загружает файл в свой кэш, то он сообщает серверу, что его интересуют все операции открытия этого файла процессами на других рабочих станциях. В этом случае сервер создает таблицу, отмечающую местонахождение этого кэшированного файла. Если другой процесс где-либо в системе открывает этот файл, то сервер посылает сообщение клиенту, чтобы тот отметил этот вход кэша как недействительный. Если этот файл в настоящее время используется, то использующие его процессы могут продолжать делать это. Однако, если другой процесс пытается открыть его заново, то клиент должен свериться с сервером, действителен ли все еще этот вход в кэше, а если нет, то получить новую копию. Если рабочая станция терпит крах, а затем перезагружается, то все файлы в кэше отмечаются как недействительные.

Блокировка файла поддерживается с помощью системного вызова UNIX FLOCK. Если блокировка не снимается в течение 30 минут, то она снимается по тайм-ауту. Тома, предназначенные только для чтения, такие как системные двоичные файлы, реплицируются, а пользовательские файлы – нет.

Хотя прикладные программы видят традиционное пространство имен UNIX, внутренняя организация сервера и клиента использует совершенно другую схему имен. Они используют двухуровневую схему именования, при которой каталог содержит структуры, называемые fids (file

identifiers), вместо традиционных номеров i-узлов.

Fid состоит из трех 32-х битных полей. Первое поле - это номер тома, который однозначно определяет отдельный том в системе. Это поле говорит, на каком томе находится файл. Второе поле называется vnode, это индекс в системных таблицах определенного тома. Оно определяет конкретный файл в данном томе. Третье поле - это уникальный номер, который используется для обеспечения повторного использования vnode. Если файл удаляется, то его vnode может быть повторно использован, но с другим значением уникального номера, для того, чтобы обнаружить и отвергнуть все старые fids.

Протокол между сервером и клиентом использует fid для идентификации файла. Когда fid поступает в сервер, по значению номера тома производится поиск в базе данных, управляемой всеми серверами, чтобы обнаружить нужный сервер. Тома могут перемещаться между серверами, но не части томов, так что эта база данных требует периодического обновления, но перемещения томов случается редко, так что трафик обновления невелик. Перемещение тома является неделимым - сначала на сервере назначения делается копия тома, а затем удаляется оригинал. Этот механизм также используется для репликации томов только для чтения за исключением того, что исходный том не удаляется после его копирования. Этот же алгоритм используется для резервного копирования. Когда делается копия, то она помещается в файловую систему как том только для чтения. В течение последующих 24 часов процесс ско-

пирует этот том на ленту. Дополнительное преимущество этого метода - пользователь, который случайно удалил файл, все еще имеет доступ ко вчерашней копии.

Теперь рассмотрим общий механизм доступа к файлам в AFS. Когда приложение выполняет системный вызов OPEN, то он перехватывается оболочкой клиента, которая первым делом проверяет, не начинается ли имя файла с /cmu. Если нет, то файл локальный, и обрабатывается обычным способом. Если да, то файл разделяемый. Производится грамматический разбор имени, покомпонентно находится fid. По fid проверяется кэш, и здесь имеется три возможности:

1. Файл находится в кэше, и он достоверен.
2. Файл находится в кэше, и он не достоверен.
3. Файл не находится в кэше.

В первом случае используется кэшированный файл. Во втором случае клиент запрашивает сервер, изменялся ли файл после его загрузки. Файл может быть недостоверным, если рабочая станция недавно перезагружалась или же некоторый другой процесс открыл файл для записи, но это не означает, что файл уже модифицирован, и его новая копия записана на сервер. Если файл не изменялся, то используется кэшированный файл. Если он изменялся, то используется новая копия. В третьем случае файл также просто загружается с сервера. Во всех трех случаях

конечным результатом будет то, что копия файла будет на локальном диске в каталоге /cash, отмеченная как достоверная.

Вызовы приложения READ и WRITE не перехватываются оболочкой клиента, они обрабатываются обычным способом. Вызовы CLOSE перехватываются оболочкой клиента, которая проверяет, был ли модифицирован файл, и, если да, то переписывает его на сервер, который управляет данным томом.

Помимо кэширования файлов, оболочка клиента также управляет кэшем, который отображает имена файлов в идентификаторы файлов fid. Это ускоряет проверку, находится ли имя в кэше. Проблема возникает, когда файл был удален и заменен другим файлом. Однако этот новый файл будет иметь другое значение поля "уникальный номер так что fid будет выявлен как недостоверный. При этом клиент удалит вход (pass, fid) и начнет грамматический разбор имени с самого начала. Если дисковый кэш переполняется, то клиент удаляет файлы в соответствии с алгоритмом LRU.

Vice работает на каждом сервере как отдельная многонитевая программа. Каждая нить обрабатывает один запрос. Протокол между сервером и клиентом использует RPC и построен непосредственно на API. В нем есть команды для перемещения файлов в обоих направлениях, блокирования файлов, управления каталогами и некоторые другие. Vice хранит свои таблицы в виртуальной памяти, так что они могут быть произвольной величины.

Так как клиент идентифицирует файлы по их идентификаторам fid, то у сервера возникает следующая проблема: как обеспечить доступ к UNIX-файлу, зная его vnode, но не зная его полное имя. Для решения этой проблемы в AFS в UNIX добавлен новый системный вызов, позволяющий обеспечить доступ к файлам по их индексам vnode.

Реализация DFS на базе AFS дает прекрасный пример того, как работают вместе различные компоненты DCE. DFS работает на каждом узле сети совместно со службой каталогов DCE, обеспечивая единое пространство имен для всех файлов, хранящихся в DFS. DFS использует списки ACL системы безопасности DCE для управления доступом к отдельным файлам. Потоковые функции RPC позволяют DFS передавать через глобальные сети большие объемы данных за одну операцию.

Распределенная служба времени

В распределенных сетевых системах необходимо иметь службу согласования времени. Многие распределенные службы, такие как распределенная файловая система и служба идентификации, используют сравнение дат, сгенерированных на различных компьютерах. Чтобы сравнение имело смысл, пакет DCE должен обеспечивать согласованные временные отметки.

Сервер времени OSF DCE - это система, которая поставляет время другим системам в целях синхронизации. Любая система, не содержащая сервера времени, называется клерком (clerk). Распределенная служ-

ба времени использует три типа серверов для координации сетевого времени. Локальный сервер синхронизируется с другими локальными серверами той же локальной сети. Глобальный сервер доступен через расширенную локальную или глобальную сети. Курьер (courier) - это специальный локальный сервер, который периодически сверяет время с глобальными серверами. Через периодические интервалы времени серверы синхронизируются друг с другом с помощью протокола DTS OSF. Этот протокол может взаимодействовать с протоколом синхронизации времени NTP сетей Internet.

Многие фирмы-потребители программного обеспечения уже используют или собираются использовать средства DCE, поэтому ведущие фирмы-производители программного обеспечения, такие как IBM, DEC и Hewlett-Packard, заняты сейчас реализацией и поставкой различных элементов и расширений этой технологии.

Одной из главных особенностей и достоинств пакета DCE OSF является тесная взаимосвязь всех его компонентов. Это свойство пакета иногда становится его недостатком. Так, очень трудно работать в комбинированном окружении, когда одни приложения используют базис DCE, а другие - нет. В версии 1.1 совместимость служб пакета с аналогичными средствами других производителей улучшена. Например, служба Kerberos DCE в текущей версии несовместима с реализацией Kerberos MIT из-за того, что Kerberos DCE работает на базе средств RPC DCE, а Kerberos MIT - нет. OSF обещает полную совместимость с Kerberos MIT

в версии 1.1. Имеются и положительные примеры совместимости пакета DCE со средствами других производителей, например со средствами Windows NT. Хотя Windows NT и не является платформой DCE, но их совместимость может быть достигнута за счет полной совместимости средств RPC. Поэтому, после достаточно тщательной работы на уровне исходных кодов, разработчики могут создать DCE-сервер, который сможет обслуживать Windows NT-клиентов, и Windows NT-сервер, который работает с DCE-клиентами.

Для того, чтобы стать действительно распространенным базисом для создания гетерогенных распределенных вычислительных сред, пакет DCE должен обеспечить поддержку двух ключевых технологий - обработку транзакций и объектно-ориентированный подход. Поддержка транзакций совершенно необходима для многих деловых приложений, когда недопустима любая потеря данных или их несогласованность. Две фирмы - IBM и Transarc - предлагают дополнительные средства, работающие над DCE и обеспечивающие обработку транзакций. Что же касается объектно-ориентированных свойств DCE, то OSF собирается снабдить этот пакет средствами, совместимыми с объектно-ориентированной архитектурой CORBA, и работающими над инфраструктурой DCE. После достаточно тщательной работы на уровне исходных кодов, разработчики могут создать DCE-сервер, который сможет обслуживать Windows NT-клиентов, и Windows NT-сервер, который работает с DCE-клиентами.

Для того, чтобы стать действительно распространенным базисом для

создания гетерогенных распределенных вычислительных сред, пакет DCE должен обеспечить поддержку двух ключевых технологий - обработку транзакций и объектно-ориентированный подход. Поддержка транзакций совершенно необходима для многих деловых приложений, когда недопустима любая потеря данных или их несогласованность. Две фирмы - IBM и Transarc - предлагают дополнительные средства, работающие над DCE и обеспечивающие обработку транзакций. Что же касается объектно-ориентированных свойств DCE, то OSF собирается снабдить этот пакет средствами, совместимыми с объектно-ориентированной архитектурой CORBA и работающими над инфраструктурой DCE.

11.3 Обзор сетевых операционных систем

Большое разнообразие типов компьютеров, используемых в вычислительных сетях, влечет за собой разнообразие операционных систем: для рабочих станций, для серверов сетей уровня отдела и серверов уровня предприятия в целом. К ним могут предъявляться различные требования по производительности и функциональным возможностям, желательно, чтобы они обладали свойством совместимости, которое позволило бы обеспечить совместную работу различных ОС.

Сетевые ОС могут быть разделены на две группы: масштаба отдела и масштаба предприятия. ОС для отделов или рабочих групп обеспечивают набор сетевых сервисов, включая разделение файлов, приложений и

принтеров. Они также должны обеспечивать свойства отказоустойчивости, например, работать с RAID-массивами, поддерживать кластерные архитектуры. Сетевые ОС отделов обычно более просты в установке и управлении по сравнению с сетевыми ОС предприятия, у них меньше функциональных свойств, они меньше защищают данные и имеют более слабые возможности по взаимодействию с другими типами сетей, а также худшую производительность.

Сетевая операционная система масштаба предприятия прежде всего должна обладать основными свойствами любых корпоративных продуктов, в том числе:

- масштабируемостью, то есть способностью одинаково хорошо работать в широком диапазоне различных количественных характеристик сети,
- совместимостью с другими продуктами, то есть способностью работать в сложной гетерогенной среде интерсети в режиме plug-and-play.

Корпоративная сетевая ОС должна поддерживать более сложные сервисы. Подобно сетевой ОС рабочих групп, сетевая ОС масштаба предприятия должна позволять пользователям разделять файлы, приложения и принтеры, причем делать это для большего количества пользователей и объема данных и с более высокой производительностью. Кроме

того, сетевая ОС масштаба предприятия обеспечивает возможность соединения разнородных систем - как рабочих станций, так и серверов. Например, даже если ОС работает на платформе Intel, она должна поддерживать рабочие станции UNIX, работающие на RISC-платформах. Аналогично, серверная ОС, работающая на RISC-компьютере, должна поддерживать DOS, Windows и OS/2. Сетевая ОС масштаба предприятия должна поддерживать несколько стеков протоколов (таких как TCP/IP, IPX/SPX, NetBIOS, DECnet и OSI), обеспечивая простой доступ к удаленным ресурсам, удобные процедуры управления сервисами, включая агентов для систем управления сетью.

Важным элементом сетевой ОС масштаба предприятия является централизованная справочная служба, в которой хранятся данные о пользователях и разделяемых ресурсах сети. Такая служба, называемая также службой каталогов, обеспечивает единый логический вход пользователя в сеть и предоставляет ему удобные средства просмотра всех доступных ему ресурсов. Администратор, при наличии в сети централизованной справочной службы, избавлен от необходимости заводить на каждом сервере повторяющийся список пользователей, а значит избавлен от большого количества рутинной работы и от потенциальных ошибок при определении состава пользователей и их прав на каждом сервере.

Важным свойством справочной службы является ее масштабируемость, обеспечиваемая распределенностью базы данных о пользователях и ресурсах.

Такие сетевые ОС, как Banyan Vines, Novell NetWare 4.x, IBM LAN Server, Sun NFS, Microsoft LAN Manager и Windows NT Server, могут служить в качестве операционной системы предприятия, в то время как ОС NetWare 3.x, Personal Ware, Artisoft LANtastic больше подходят для небольших рабочих групп.

Критериями для выбора ОС масштаба предприятия являются следующие характеристики:

- Органичная поддержка многосерверной сети;
- Высокая эффективность файловых операций;
- Возможность эффективной интеграции с другими ОС;
- Наличие централизованной масштабируемой справочной службы;
- Хорошие перспективы развития;
- Эффективная работа удаленных пользователей;
- Разнообразные сервисы: файл-сервис, принт-сервис, безопасность данных и отказоустойчивость, архивирование данных, служба обмена сообщениями, разнообразные базы данных и другие;
- Разнообразные программно-аппаратные хост-платформы: IBM SNA, DEC NSA, UNIX;

- Разнообразные транспортные протоколы: TCP/IP, IPX/SPX, NetBIOS, AppleTalk;
- Поддержка многообразных операционных систем конечных пользователей: DOS, UNIX, OS/2, Mac;
- Поддержка сетевого оборудования стандартов Ethernet, Token Ring, FDDI, ARCnet;
- Наличие популярных прикладных интерфейсов и механизмов вызова удаленных процедур RPC;
- Возможность взаимодействия с системой контроля и управления сетью, поддержка стандартов управления сетью SNMP.

Конечно, ни одна из существующих сетевых ОС не отвечает в полном объеме перечисленным требованиям, поэтому выбор сетевой ОС, как правило, осуществляется с учетом производственной ситуации и опыта. **Таблица 6** содержит основные характеристики популярных и доступных в настоящее время сетевых ОС.

Таблица 6 Основные характеристики сетевых операционных систем

Novell NetWare 4.1	Специализированная операционная система, оптимизированная для работы в качестве файлового сервера и принт-сервера Ограниченные средства для использования в качестве сервера приложений: не имеет средств виртуальной памяти и вытесняющей многозадачности, а поддержка симметричного мультипроцессирования отсутствовала до самого недавнего времени. Отсутствуют API основных операционных сред, используемых для разработки приложений, - UNIX, Windows, OS/2 Серверные платформы: компьютеры на основе процессоров Intel, рабочие станции RS/6000 компании IBM под управлением операционной системы AIX с помощью продукта NetWare for UNIX Поставляется с оболочкой для клиентов: DOS, Macintosh, OS/2, UNIX, Windows (оболочка для Windows NT разрабатывается компанией Novell в настоящее время, хотя Microsoft уже реализовала клиентскую часть NetWare в Windows NT) Организация одноранговых связей возможна с помощью ОС PersonalWare Имеет справочную службу NetWare Directory Services (NDS), поддерживающую централизованное управление, распределенную, полностью реплицируемую, автоматически синхронизируемую и обладающую отличной масштабируемостью Поставляется с мощной службой обработки сообщений Message Handling Service (MHS), полностью интегрированную (начиная с версии 4.1) со справочной службой Поддерживаемые сетевые протоколы: TCP/IP, IPX/SPX, NetBIOS, Appletalk Поддержка удаленных пользователей: ISDN, коммутируемые телефонные линии, frame relay, X.25 - с помощью продукта NetWare Connect (поставляется отдельно) Безопасность: аутентификация с помощью открытых ключей метода шифрования RSA; сертифицирована по уровню C2 Хороший сервер коммуникаций Встроенная функция компрессии диска Сложное обслуживание

Banyan VINES 6.0 и ENS (Enterprise Network Services) 6.0	Серверные платформы: ENS for UNIX: работает на RISC-компьютерах под управлением SCO UNIX, HP-UX, Solaris, AIXENS for NetWare: работает на Intel платформах под управлением NetWare 2.x, 3.x, 4.x VINES работает на Intel-платформах Клиентские платформы: DOS, Macintosh, OS/2, UNIX, Windows for Workgroups, Windows NT Хороший сервер приложений: поддерживаются вытесняющая многозадачность, виртуальная память и симметричное мультипроцессирование в версии VINES и в ENS-версиях для UNIX. Поддерживаются прикладные среды UNIX, OS/2, Windows Поддержка одноранговых связей - отсутствует Справочная служба - Streettalk III, наиболее отработанная из имеющихся на рынке, с централизованным управлением, полностью интегрированная с другимисетевыми службами, распределенная, реплицируемая и автоматически синхронизируемая, отлично масштабируемая Согласованность работы с другими сетевыми ОС: хорошая; серверная оболочка работает в средах NetWare и UNIX; пользователи NetWare, Windows NT и LAN Server могут быть объектами справочной службы Streettalk III Служба сообщений - Intelligent Messaging, интегрирована с другими службами Поддерживаемые сетевые протоколы: VINES IP, TCP/IP, IPX/SPX, Appletalk Поддержка удаленных пользователей: ISDN, коммутируемые телефонные линии, X.25 Служба безопасности: поддерживает электронную подпись (собственный алгоритм), избирательные права доступа, шифрацию; не сертифицирована Простое обслуживание Хорошо масштабируется Отличная производительность обмена данными между серверами, хуже – при обмене сервер-ПК
Microsoft LAN Manager	Широкая распространенность; работает под OS/2 и UNIX поддерживает мощные серверные платформы один сервер может поддерживать до 2 000 клиентов

Microsoft Windows NT Server 3.51 и 4.0	Серверные платформы: компьютеры на базе процессоров Intel, PowerPC, DEC Alpha, MIPS Клиентские платформы: DOS, OS/2, Windows, Windows for Workgroups, Macintosh Организация одноранговой сети возможна с помощью Windows NT Workstation и Windows for Workgroups Windows NT Server представляет собой отличный сервер приложений: он поддерживает вытесняющую многозадачность, виртуальную память и симметричное мультипроцессирование, а также прикладные среды DOS, Windows, OS/2, POSIX Справочные службы: доменная для управления учетной информацией пользователей (Windows NT Domain Directory service), справочные службы имен WINS и DNS Хорошая поддержка совместной работы с сетями NetWare: поставляется клиентская часть (редиректор) для сервера NetWare (версий 3.x и 4.x в режиме эмуляции 3.x, справочная служба NDS поддерживается, начиная с версии 4.0), выполненная в виде шлюза в Windows NT Server или как отдельная компонента для Windows NT Workstation; недавно Microsoft объявила о выпуске серверной части NetWare как оболочки для Windows NT Server Служба обработки сообщений - Microsoft Mail, основанная на DOS- платформе, в этом году ожидается версия для платформы Windows NT - Microsoft Message Exchange, интегрированная с остальными службами Windows NT Server Поддерживаемые сетевые протоколы: TCP/IP, IPX/SPX, NetBEUI, Appletalk Поддержка удаленных пользователей: ISDN, коммутируемые телефонные линии, frame relay, X.25 - с помощью встроенной подсистемы Remote Access Server (RAS) Служба безопасности: мощная, использует избирательные права доступа и доверительные отношения между доменами; узлы сети, основанные на Windows NT Server, сертифицированы по уровню C2 Простота установки и обслуживания Отличная масштабируемость

IBM LAN Server 4.0	Серверные платформы: операционные системы MVS и VM для мейнфреймов; AS/400 с OS/400, рабочие станции RS/6000 с AIX, серверы Intel 486 или Pentium под OS/2. Поставляется с оболочками для клиентов: DOS, Macintosh, OS/2, Windows, Windows NT, Windows for Workgroups Серверы приложений могут быть организованы с помощью LAN Server 4.0 в операционных средах MVS, VM, AIX,OS/2, OS/400. В среде OS/2 поддерживаются: вытесняющая многозадачность, виртуальная память и симметричное мультипроцессирование. Организация одноранговых связей возможна с помощью ОС Warp Connect Справочная служба - LAN Server Domain, то есть основа на доменном подходе Поддерживаемые сетевые протоколы: TCP/IP, NetBIOS, Appletalk Безопасность - избирательные права доступа, система не сертифицирована Служба обработки сообщений – отсутствует Высокая производительность Недостаточная масштабируемость
IBM и NCR LAN Manager	LAN Manager for UNIX хорошо распространена (15% объема мировых продаж сетевых ОС) LAN Manager for AIX поддерживает RISC компьютеры System/6000 в качестве файлового сервера Работает под UNIX, имеет все преимущества, связанные с использованием этой ОС

Лабораторный практикум

Лабораторная работа № 1

Командный интерпретатор Windows. Командные файлы пакетной обработки

Цель работы: Получение практических навыков работы с командным интерпретатором Windows, разработки командных файлов.

Содержание работы

1. Создайте текстовые файлы с именами **text.txt, text1.txt** и добавьте в них произвольный текст, воспользовавшись средствами **DOS Editor** либо **командой перенаправления ввода**. Сохраните эти файлы в заранее созданном вами каталоге:

   \OS\<Ваша_фамилия>\Lab_1.

2. Измените атрибуты «Только для чтения» и «Скрытый» файла

text.txt. Произведите сравнение данного файла с файлом **text1.txt**.

3. Создайте текстовый файл **text2.txt**, скопировав в него содержимое файлов **text.txt** и **text1.txt**.

4. Создайте подкаталог в вашем каталоге с именем **pLab_1**. Сделайте ваш рабочий каталог текущим.

5. Сохраните значение текущего активного каталога и перейдите в подкаталог **pLab_1**. Скопируйте все файлы с расширением **.txt** из вашего каталога в подкаталог, изменив их расширение на **.doc**, а затем вернитесь к предыдущему активному каталогу.

6. Произведите дозапись в файл **text2.txt** следующих данных: графическое представление структуры вашей папки, текущие дату и время, вашу фамилию, имя и отчество.

7. Переместите все файлы, находящиеся в каталоге **pLab_1** в ваш каталог и удалите **pLab_1**. Удалите ваш каталог. Подумайте над тем, какой командой нужно воспользоваться в первом, а какой – во втором случае.

8. Реализуйте задачи 1-7 с помощью пакетного bat-файла с именем **lab1_1**.

9. Внесите изменения в полученный пакетный файл, чтобы его исполнение приостанавливалось перед выполнением очередного задания, с выводом соответствующего сообщения.

10. Создайте пакетный файл с именем **lab1_2**, выполняющий постраничный вывод на экран содержимого указанного в параметрах каталога и содержимого всех текстовых файлов данного каталога.

11. Создайте пакетный файл с именем **lab1_3**, добавляющий указанный в параметрах путь в переменную окружения PATH, после чего выводит пути доступа на экран. Для добавления используйте команду SET PATH = <*заданный_путь*>;%PATH%.

Методические указания

Команды Windows для работы с файлами

Attrib – позволяет просматривать, устанавливать или снимать атрибуты файла или каталога, такие как «Только чтение», «Архивный», «Системный» и «Скрытый».

Chdir (Cd) – вывод имени текущего каталога или переход в другую папку

Copy – копирование одного или нескольких файлов

Del (erase) – удаление файлов.

Dir – вывод списка файлов и подкаталогов каталога

Edit – редактор MS-DOS

Help (/?) – вывод справочных сведений о командах

Fc – сравнение двух файлов и вывод различий между ними.

Find – поиск заданной строки текста в файле или нескольких файлах

Findstr – поиск образцов текста в файлах с использованием регулярных выражений.

Ftype – вывод или редактирование связи между типом файла и его расширением

Mkdir – создание папки

Move – служит для перемещения одного или нескольких файлов из одного каталога в другой.

PushD – сохранение значения текущей активной папки и переход к другой папке

PopD – восстановление предыдущего значения активной папки, сохраненного с помощью команды PushD

Rename (ren) – изменяет имя файла или набора файлов

Replace – заменяет файлы в одном каталоге файлами с теми же именами из другого каталога

Rmdir (rd) – удаляет каталог.

Tree – представляет графически дерево каталогов заданного пути или диска.

Xcopy – копирует файлы и каталоги, включая подкаталоги

Более подробную информацию по каждой команде можно получить,

если после команды **help** в качестве параметра набрать имя интересующей команды или имя команды, а затем ключ **/?**.

Справочная информация по различным командам свидетельствует, что командой, набираемой в командной строке, является собственно имя команды, за которым могут следовать *ключи (опции)* – указания, модифицирующие поведение команды. *Квадратные скобки* ([ ]) в пояснениях обозначают, что эта информация не является обязательной при наборе команды. Ключи начинаются со *знака слэша* / и состоят из одного или нескольких символов. Кроме ключей, после команды могут следовать *аргументы (параметры)* – названия объектов, над которыми должна быть выполнена команда. Очень часто аргументами служат имена файлов и каталогов.

Ввод команды заканчивается нажатием клавиши Enter, после чего команда передается на исполнение командному процессору. В результате выполнения команды на экране дисплея могут появиться сообщения о ходе выполнения команды или об ошибках, а появление очередного приглашения (мигающего курсора) свидетельствует об успешном выполнении введенной команды и ожидании ввода следующей.

В сложных сценариях работы, когда используются длинные последовательности вводимых команд, всегда имеется возможность повторить или отредактировать одну команду (или часть команд) из выполненного перечня. Для этого следует выбрать нужную команду из журнала (истории). Клавиша управления курсором вверх выводит предыдущую

команду, вниз - последующую, клавиши вправо-влево позволяют перейти к посимвольному редактированию очередной команды.

По умолчанию предполагается, что результаты выполнения вводимых команд отображаются на экране дисплея. При желании можно использовать так называемое перенаправление ввода-вывода. Для этого используются следующие символы: символ <– для перенаправления ввода, а символы > и >>– для вывода. Например, строка вида

команда<имя_файла,

содержащая символ <, обозначает, что данные для выполнения команды должны браться из файла, обычно текстового. Еще один пример. Команда

dir>fl.txt

обозначает, что содержимое текущего каталога требуется записать в текстовый файл **fl.txt**. Если такового файла в текущем каталоге нет, то средствами операционной системы он создается. Запись осуществляется в начало файла, т.е. перед записью предыдущее содержимое, если оно было, стирается. При желании сохранить старое содержимое файла запись следует производить в конец файла, используя команду вида **dir>>fl.txt**.

Сведения о создании командных файлов см. в директории **Литература\О_командных_файлах.pdf**

Замечания

– Для сдачи работы необходимо:

1. **самостоятельно** выполнить работу, изучив необходимую теорию;
2. представить отчет по лабораторной работе;
3. ответить на вопросы преподавателя по теме работы.

– В ходе выполнения заданий оформляется **письменный отчет**. Отчет включает тему лабораторной работы, цель, содержание работы, а также краткое изложение выполненных работ (в соответствии с содержанием работы). Отчеты по всем работам **хранятся** студентом до выполнения отчетных мероприятий по дисциплине (получения зачетов и экзаменов).

Лабораторная работа № 2

Часть 1

Тема: Управление процессами и потоками в операционной системе MS Windows
Цель: самостоятельная работа с документацией команд, получение практических навыков управления процессами и потоками

Содержание работы

1. Составьте справочник для команд управления процессами, расписав какие параметры для чего нужны.
2. Поработайте с этими командами: запуская, запуская по времени, меняя приоритеты, уничтожая процессы.
3. Изучите возможности утилит Диспетчер задач Windows и Process Explorer. Сделайте выводы об отличиях между этими средствами.

Методические указания

Команды Windows для работы с процессами

at – запуск программ в заданное время
schtasks – настраивает выполнение команд по расписанию

start – запускает определенную программу или команду в отдельном окне.

taskkill – завершает процесс

tasklist – выводит информацию о работающих процессах

Для получения более подробной информации, можно команду help (**например: help at**)

cmd.exe - запуск командной оболочки Windows.

Диспетчер задач ОС Windows

Диспетчер задач в операционных системах семейства Microsoft Windows — утилита для вывода на экран списка запущенных процессов и потребляемых ими ресурсов (в частности статус, процессорное время и потребляемая оперативная память). Также есть возможность некоторой манипуляции процессами. Диспетчер задач можно вызвать нажав одновременно клавиши Ctrl+Shift+Esc или Ctrl+Alt+Del. Диспетчер задач можно также запустить по названию taskmgr (через диалоговое окно «Запуск программы» или кликнув правым кнопкой мыши по панели задач и выбрав соответствующий пункт в всплывающем меню.

Часть 2

Тема: Управление процессами и потоками в операционной системе MS Windows
Цель: самостоятельная работа с документацией команд, получение практических навыков управления процессами и потоками

Содержание работы

Напишите программу в **IDE Visual Studio 2010 (на языке программирования C#)**, выполняющую функции создания и уничтожения произвольно задаваемых процессов (реализовать проверку некорректно заданных процессов). Предусмотреть возможность создания нескольких экземпляров процессов с корректным уничтожением каждого (можно использовать любую структуру, хранящую сведения об идентификаторах процессов). Работу написанной программы отследить в Process Explorer'е. Сделать выводы.

Методические указания

1. Г. Шилдт «C# 4.0: Полное руководство» - стр. 833-835, 882-884;
2. Э. Троелсен «Язык программирования C# 2010 и платформа .NET» - стр. 585-593.

Лабораторная работа № 3

Разработка многопоточных приложений в среде Visual Studio

Цель: Приобрести навыки написания простого многопоточного приложения в ОС Windows

Общее задание: Написать приложение с одним вспомогательным потоком, вычисляющим заданное выражение и выводящее результат в любой визуальный компонент. В программе должны быть предусмотрены функции приостановки, возобновления и полной остановки выполнения потока с выводом соответствующего сообщения. Обработать исключения.

Варианты:

№ п\п	Выражение вычисляемое потоком
1	$1+\frac{1}{1!}+\frac{1}{2!}+\frac{1}{3!}+\ldots+\frac{1}{n!}+\ldots$
2	$1-\frac{1}{1!}+\frac{1}{2!}-\frac{1}{3!}+\ldots\pm\frac{1}{n!}\mp\ldots$
3	$1+\frac{1}{2}+\frac{1}{4}+\ldots+\frac{1}{2^n}+\ldots$
4	$1-\frac{1}{2}+\frac{1}{4}-\frac{1}{8}+\ldots\pm\frac{1}{2^n}\mp\ldots$
5	$\frac{1}{1\cdot3}+\frac{1}{3\cdot5}+\ldots+\frac{1}{(2n-1)(2n+1)}+\ldots$
6	$1+\frac{x^2}{2!}+\frac{x^4}{4!}+\ldots+\frac{x^{2n}}{(2n)!}+\ldots$
7	$x+\frac{x^3}{3}+\frac{x^5}{5}+\ldots+\frac{x^{2n+1}}{(2n+1)!}+\ldots$
8	$1+\frac{1}{2}(2\cdot3x+3\cdot4x^2+5\cdot6x^3+\ldots)$
9	$1+\frac{x\ln a}{1!}+\frac{(x\ln a)^2}{2!}+\ldots+\frac{(x\ln a)^n}{n!}+\ldots$
10	$\frac{\pi}{2}-\left(x-\frac{x^3}{3}+\frac{x^5}{5}-\ldots+(-1)^n\frac{x^{2n+1}}{2n+1}\pm\ldots\right)$
11	$\frac{1}{2}+\frac{3}{2^2}+\ldots+\frac{2n-1}{2^n}+\ldots$
12	$1-\frac{2}{2^2}+\frac{1}{2^3}-\frac{1}{4^2}+\ldots\pm\frac{1}{n^2}\mp\ldots$
13	$1+\frac{x}{1!}+\frac{x^2}{2!}+\ldots+\frac{x^n}{n!}+\ldots$
14	$1+\frac{1}{2^4}+\frac{1}{3^4}-\ldots\pm\frac{1}{n^4}\mp\ldots$

Лабораторная работа № 4

Файловая система ОС Linux. Командная оболочка bash

Цель: получение практических навыков работы с командной оболочной Linux, изучение принципов построения файловой системы Linux

Содержание работы

1. Перейдите в домашний каталог /**home**. Выведите содержимое активного каталога на экран. Поэкспериментируйте с командой вывода: выведите содержимое в один столбец, получите информацию о типах файлах, информацию о владельцах и правах доступа, отсортируйте содержимое по суффиксам имен файлов, по размерам файлов, по дате и времени, отобразите список скрытых файлов и информацию о правах доступа в одном списке.

2. Создайте подкаталог домашнего каталога /**home**/<**Фамилия**>/**lab_2** (использовать команды оболочки). Затем создайте текстовый файл с именем **text1.txt** и добавьте в него произвольный текст (использовать команду **nano**). Сохраните этот файл в вышеназванном подкаталоге. Сделайте этот каталог текущим.

3. Создайте еще один текстовый файл с именем **text2.txt**, добавьте в него произвольный текст (использовать перенаправление выходного потока). Выполните конкатенацию (объединение) файлов **text1.txt** и **text2.txt** с записью результата в файл **text3.txt** (создавать этот файл заранее не нужно). Выполните постраничный вывод содержимого получившегося файла. Создайте дополнительно текстовые файлы с именами **text[4-15].txt** (можно использовать команду **touch** или редактор **nano**).

4. Создайте подкаталог в вашем каталоге с именем **plab_2**, в этом подкаталоге создайте текстовый файл с произвольным содержимым **text4.txt** (используйте конструкцию выполнения команд при успешном завершении предыдущих). Переименуйте файл **text4.txt** в **text9.txt**. Выполните копирование текстовых файлов с именами **text[1-8].txt** из директории **lab_2** в директорию **./plab_2**.

5. Смените текущий каталог на ./plab_2. Переименуйте файлы text[1-3].txt соответственно в файлы text[16-18].doc, text[4-6].txt в text[19-21].abc, text[7-9].txt в text[22-24].zed. Переместите получившийся набор файлов text[16-24].* в каталог .. . Вернитесь к значению предыдущего активного каталога.

6. Произведите дозапись в файл **text1.txt** следующих данных: текущие дата и время, ваша фамилию, имя и отчество.

7. Произведите сжатие и архивацию всех файлов в директории /<**Фамилия**>/**lab_2** с помощью команды zip с созданием файла texts.zip с уровнем сжатия 9, заданием пароля. Удалите исходные файлы. Разархивируйте получившийся файл. Удалите каталог **plab_2**. Удалите каталог **lab_2**. Подумайте над тем, какой командой нужно воспользоваться при удалении каталога **plab_2**, а какой — при удалении **lab_2**.

Методические указания

Список основных команд bash

ls – вывод списка файлов и каталогов

Опции:

-R – содержимое подкаталогов
-1 – вывод в один столбец
-a – отображение скрытых файлов и каталогов
-F – отображение информации о типах файлов
-l – информация о правах доступа и владельцах
-r – вывод информации в обратном порядоке
-X – сортировка содержимого по суффиксам имен файлов
-t – сортировка по дате и времени
-S – сортировка по размеру

pwd – опредеделение пути к текущему каталогу
cd – переход к другому каталогу

- **cd** - - переход к предыдущему каталогу
- **cd** $\sim$ - переход в рабочий каталог

touch – изменение сведений о времени (если файл существует, если нет – то создает его)
mkdir – создание нового каталога

Опции:

- -p – создание со всеми подкаталогами
- -v – информация о действиях, выполняемых командой

cp – копирование файлов

Опции:

- -i – предотвращения копирования поверх важных файлов
- -R – копирование каталогов

mv – перемещение и переименование
rm – удаление файлов

Опции:

- -i – предотвращение удаления важных файлов
- -R – удаление всех подкаталогов
- -f – отключение напоминаний о том, что каталог непустой

rmdir – удаление пустого каталога
su – переключение пользователя
man – получение информации о командах
cat – конкатенация файлов с выводом в stdout cat file1 file2
less – постраничный вывод текста
zip – архивирование и сжатие

Пример:

- zip -4 test.zip *.txt, 4 – уровень сжатия

Опции:

- -[0-9] – уровень сжатия
- -P, -e – защита паролем

unzip – разархивирование

Опции:

- -t – проверка файлов, предназначенных для разархивирования
- -v – получение детальной информации о выполняемых действиях
- -l – список файлов для разархивирования

Запуск Debian Linux

1. Запускаем программу Sun VirtualBox, выбираем из списка слева систему Debian, нажимаем на кнопку «Старт»;
2. По запросу системы о вводе логина и пароля вводим соответственно **root** и **123**.
3. Выполняем предложенные задания.

Замечания

Если ОС Debian Linux отсутствует в списке, то выполняем следующие действия:

1. Открываем Total Commander, в пункте меню «Сеть» выбираем «Новое ftp-соединение...», вводим в поле «Соединиться с...» адрес **mf.brsu.by** и нажимаем на кнопку «OK». После этого в появившемся разделе открываем директорию «soft\virtualmachines» и копируем файл **netdeb.vdi** в локальную директорию.
2. Создаем новую виртуальную машину, после чего указываем в качестве имени загрузочного диска путь к файлу из пункта 1.

Лабораторная работа № 5

Администрирование систем Unix

Цель: Получение практических навыков администрирования в системе Unix

Задания

1. Просмотрите файлы **/etc/shadow**, **/etc/group** и **/etc/passwd**.

2. Зарегистрируйте пользователя **test1**, имеющего домашний каталог **/home/nouser** и являющегося членом групп **mail** и **users**. Пользователь должен иметь **UID = 1777**. Ознакомьтесь с соответствующей записью в **/etc/passwd**. После этого измените **UID** пользователя **test1** на **1001**. Проверьте изменения в соответствующей записи в **/etc/passwd**. Изучите соответствующую запись в **/etc/shadow**. Установите для данного пользователя пароль. Изучите изменения в **/etc/shadow**.

3. Создайте учетную запись для пользователя **test2** с настройками по умолчанию. Проверьте, создался ли домашний каталог пользователя. Измените имя созданного пользователя **test2** на **test3**. Изучите **/etc/group**. Удалите пользователя **test3**. Изучите **/etc/group**.

4. Установите дату устаревания пароля для пользователя **test1** на 31 декабря 2010 года. Проверьте изменения с помощью команды **chage**. Заблокируйте учетную запись пользователя **test1**. Проверьте изменения в **/etc/shadow**.

5. Создайте группу пользователей **xusers** с **GID = 1010**.

6. Зарегистрируйте свою учетную запись **guest** в качестве участника группы xusers. Ознакомьтесь с соответствующей записью в **/etc/group**.

7. Измените имя группы **xusers** на **yusers**. Проверьте изменения в **/etc/group**.

8. Удалите все созданные вами учетные записи и группы пользователей (**учетную запись guest удалять не нужно!!!**).

Методические указания

Для выполнения лабораторной работы использовать следующие команды: sudo, useradd, usermod, userdel, groupadd, groupmod, groupdel, passwd, cat, chage. Синтаксис этих команд изучить самостоятельно.

Замечания

– Для сдачи работы необходимо:

1. **самостоятельно** выполнить работу, изучив необходимую теорию;
2. представить отчет по лабораторной работе;
3. ответить на вопросы преподавателя по теме работы.

– В ходе выполнения заданий оформляется **письменный отчет**. Отчет включает тему лабораторной работы, цель, содержание работы, а также краткое изложение выполненных работ (в соответствии с содержанием работы). Отчеты по всем работам **хранятся** студентом до выполнения отчетных мероприятий по дисциплине (получения зачетов и экзаменов).

Лабораторная работа № 6

Сценарии оболочки bash. Права доступа Unix

Цель: Получение практических навыков написания сценариев bash

Задания

Команда chmod позволяет задавать права доступа для файлов и каталогов. Требуется создать сценарий оболочки с именем gchmod, который позволяет искать в заданной директории файлы с заданными правами и менять эти права на другие.

Входные данные:

- каталог, в котором нужно производить поиск
- права для поиска
- целевые права

Методические указания

Смотрим в файле bash.pdf

Замечания

– Для сдачи работы необходимо:

1. **самостоятельно** выполнить работу, изучив необходимую теорию;
2. представить отчет по лабораторной работе;
3. ответить на вопросы преподавателя по теме работы.

– В ходе выполнения заданий оформляется **письменный отчет**.

Лабораторная работа № 7

Процессы в ОС UNIX. Создание и работа с процессом

Цель: Изучить программные средства создания процессов, получить навыки управления и синхронизации процессов, а также простейшие способы обмена данными между процессами. Ознакомиться со средствами динамического запуска программ в рамках порожденного процесса, изучить механизм сигналов ОС UNIX, позволяющий процессам реагировать на различные события, и каналы, как одно из средств обмена информацией между процессами.

Задание:

1. Разработать программу, реализующую действия, указанные в вариантах заданий к лабораторной работе с учётом следующих требований:

   - все действия, относящиеся как к родительскому процессу, так и к порожденным процессам, выполняются в рамках ;
   - обмен данными между процессом-отцом и процессом-потомком предлагается выполнить посредством : процесс-отец после порождения процесса-потомка постоянно опрашивает временный файл, ожидая появления в нем информации от процесса-потомка;

- если процессов-потомков несколько, и все они подготавливают некоторую информацию для процесса-родителя, каждый из процессов помещает в файл некоторую структурированную запись, при этом в этой структурированной записи содержатся сведения о том, какой процесс посылает запись, и сама подготовленная информация.

2. Модифицировать ранее разработанную программу с учётом следующих требований:

   - действия процесса-потомка реализуются , запускаемой по одному из системных вызовов execl(), execv() и т.д. из процесса-потомка;
   - процесс-потомок, после порождения, должен начинать и завершать свое функционирование по сигналу, посылаемому процессом-предком (это же относится и к нескольким процессам-потомкам);
   - обмен данными между процессами необходимо осуществить через .

Варианты заданий

1. Разработать программу, вычисляющую интеграл на отрезке $[A; B]$ от функции $\exp(x)$ методом трапеций, разбивая интервал на K равных отрезков. Для нахождения exp() программа должна породить

процесс, вычисляющий её значение путём разложения в ряд по формулам вычислительной математики.

2. Разработать программу, вычисляющую плотность распределения Пуассона с параметром λ в точке k (k – целое) по формуле $f(k) = \frac{\lambda^k \exp(-\lambda)}{k!}$. Для нахождения факториала и $\exp(-\lambda)$ программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

3. Разработать программу, вычисляющую плотность выпуклого распределения в точке по формуле $f(x) = \frac{1-\cos x}{\pi x^2}$. Для нахождения π и cos программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

4. Разработать программу, вычисляющую интеграл на отрезке $\left[0; \frac{\pi}{2}\right]$ от функции $\frac{\sin x}{x}$ методом трапеций, разбивая интервал на K равных отрезков. Для нахождения $\sin x$ программа должна породить процесс, вычисляющий её значение путём разложения в ряд по формулам вычислительной математики.

5. Разработать программу, вычисляющую значение плотности лог-нормального распределения в точке $(x > 0)$ по формуле $f(x) =$

$\frac{\frac{1}{2}\exp^{(-\frac{1}{2})\ln x^2}}{x\sqrt{2\pi}}$. Для нахождения π, $\exp(x)$ и $\ln(x)$ программа должна породить три параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

6. Разработать программу, вычисляющую число размещений n элементов по r ячейкам $N = \frac{n!}{n_1!} \cdot n_2! \cdot \ldots \cdot n_r!$, удовлетворяющее требованию, что в ячейку с номером i попадает ровно n_i элементов $(i = 1...r)$ и $n_1 + n_2 + \ldots + n_r = n$. Для вычисления каждого факториала необходимо породить процесс-потомок.

7. Разработать программу, вычисляющую интеграл на отрезке $[0; 1]$ от функции $\cos x^2 \cdot \sin x^2$ методом Симпсона, разбивая интервал на четное число $K = 2m$ равных отрезков. Для нахождения $\cos x$ и $\sin x$ программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

8. Разработать программу, вычисляющую интеграл на отрезке $[0; 1]$ от функции $\exp(x^2)$ методом трапеций, разбивая интервал на K равных отрезков. Для нахождения $\exp(x)$ программа должна породить процесс, вычисляющий её значение путём разложения в ряд по формулам вычислительной математики.

9. Разработать программу, вычисляющую число сочетаний $C(k, n) =$

$\frac{n!}{k!(n-k)!}$. Для вычисления факториалов $n!$, $k!$, $(n-k)!$ должны быть порождены три параллельных процесса-потомка.

10. Разработать программу, вычисляющую плотность нормального распределения в точке x по формуле $f(x) = \frac{exp(-\frac{x^2}{2})}{\sqrt{2\pi}}$. Для нахождения π и $\exp(-\frac{x^2}{2})$ программа должна породить два параллельных процесса, вычисляющих эти величины путём разложения в ряд по формулам вычислительной математики.

11. Разработать программу, вычисляющую интеграл в диапазоне от 0 до 1 от подинтегрального выражения $f(x) = \frac{4}{1+x^2}$ с помощью последовательности равномерно распределённых на отрезке $[0;1]$ случайных чисел, которая генерируется процессом-потомком параллельно. Процесс-потомок должен завершиться после заранее заданного числа генераций N.

Методические указания к лабораторной работе

Для порождения нового процесса (процесс-потомок) используется системный вызов fork(). Формат вызова:

```
int fork();
```

Порожденный таким образом процесс представляет собой точную копию своего процесса-предка. Единственное различие между ними заключается в том, что процесс-потомок в качестве возвращаемого значения

системного вызова fork() получает 0, а процесс-предок - идентификатор процесса-потомка. Кроме того, процесс-потомок наследует и весь контекст программной среды, включая дескрипторы файлов, каналы и т.д. Наличие у процесса идентификатора дает возможность и ОС UNIX, и любому другому пользовательскому процессу получить информацию о функционирующих в данный момент процессах.

Ожидание завершения процесса-потомка родительским процессом выполняется с помощью системного вызова wait():

```
int wait(int *status);
```

В результате осуществления процессом системного вызова wait() функционирование процесса приостанавливается до момента завершения порожденного им процесса-потомка. По завершении процесса-потомка процесс-предок пробуждается и в качестве возвращаемого значения системного вызова wait() получает идентификатор завершившегося процесса-потомка, что позволяет процессу-предку определить, какой из его процессов-потомков завершился (если он имел более одного процесса- потомка). Аргумент системного вызова wait() представляет собой указатель на целочисленную переменную status, которая после завершения выполнения этого системного вызова будет содержать в старшем байте код завершения процесса-потомка, установленный последним в качестве системного вызова exit(), а в младшем - индикатор причины завершения процесса-потомка.

Формат системного вызова exit(), предназначенного для завершения функционирования процесса:

```
int exit(int status);
```

Аргумент status является статусом завершения, который передается отцу процесса, если он выполнял системный вызов wait().

Для получения собственного идентификатора процесса используется системный вызов getpid(), а для получения идентификатора процесса-отца - системный вызов getppid():

```
int getpid(); int getppid();
```

Вместе с идентификатором процесса каждому процессу в ОС UNIX ставится в соответствие также идентификатор группы процессов. В группу процессов объединяются все процессы, являющиеся процессами-потомками одного и того же процесса. Организация новой группы процессов выполняется системным вызовом getpgrp(), а получение собственного идентификатора группы процессов - системным вызовом getpgrp(). Их формат:

```
int setpgrp(); int getpgrp();
```

С практической точки зрения в большинстве случаев в рамках порожденного процесса загружается для выполнения программа, определенная одним из системных вызовов execl(), execv(),... Каждый из этих системных вызовов осуществляет смену программы, определяющей функционирование данного процесса:

```
execl(name, arg0, arg1, ... , argn, 0) char *name, *arg0,
*arg1, ... , *argn; execv(name, argv) char *name, *argv[];
execle(name, arg0, arg1, ... , argn, 0, envp) char *name,
*arg0, *arg1, ... , *argn, *envp[]; execve(name, argv, envp)
char *name, *arg[], *envp[];
```

Сигналы - это программное средство, с помощью которого может быть прервано функционирование процесса в ОС UNIX. Механизм сигналов позволяет процессам реагировать на различные события, которые могут произойти в ходе функционирования процесса внутри него самого или во внешнем мире. Каждому сигналу ставятся в соответствие номер сигнала и строковая константа, используемая для осмысленной идентификации сигнала. Эта взаимосвязь отображена в файле описаний signal.h. Для посылки сигнала используется системный вызов, имеющий формат:

```
void kill(int pid, int sig);
```

В результате осуществления такого системного вызова сигнал, специфицированный аргументом sig, будет послан процессу, который имеет идентификатор pid. Если pid не превосходит 1, сигнал будет послан целой группе процессов.

Использование системного вызова signal() позволяет процессу самостоятельно определить свою реакцию на получение того или иного события (сигнала):

```
int sig; int (*func)(); int *signal(sig, func) ();
```

Реакцией процесса, осуществившего системный вызов signal() с аргументом func, при получении сигнала sig будет вызов функции func().

Системный вызов pause() позволяет приостановить процесс до тех пор, пока не будет получен какой-либо сигнал:

```
void pause();
```

Системный вызов alarm(n) обеспечивает посылку процессу сигнала SIGALARM через n секунд.

В ОС UNIX существует специальный вид взаимодействия между процессами - программный канал. Программный канал создается с помощью системного вызова pipe(), формат которого:

```
int fd[2]; pipe(fd);
```

Системный вызов pipe() возвращает два дескриптора файла: один для записи данных в канал, другой - для чтения. После этого все операции передачи данных выполняются с помощью системных вызовов ввода-вывода read/write. При этом система ввода-вывода обеспечивает приостановку процессов, если канал заполнен (при записи) или пуст (при чтении). Таких программных каналов процесс может установить несколько. Отметим, что установление связи через программный канал опирается на наследование файлов. Взаимодействующие процессы должны быть родственными.

Вопросы к экзамену

1. Определение и понятие операционной системы (ОС).
2. Эволюция ОС.
3. Классификация ОС по типам функционирования.
4. Классификация ОС по типам аппаратных платформ и областям использования.
5. Базовые концепции организации ОС.
6. Подсистема управления процессами.
7. Алгоритмы планирования процессов.
8. Типы процедур планирования процессов.
9. Средства синхронизации и взаимодействия процессов.
10. Обобщающее средство синхронизации процессов.
11. Тупики.
12. Концепции ресурса.
13. Необходимые условия возникновения тупика.

14. Предотвращение тупиков.
15. Алгоритм банкира.
16. Обнаружение тупиков.
17. Восстановление после тупиков.
18. Управление памятью.
19. Распределение памяти.
20. Методы распределения памяти с использованием дискового пространства.
21. Иерархия запоминающих устройств. Принцип кэширования данных.
22. Файловая система. Имена и типы файлов. Атрибуты.

Итоговый тест

Выполните, пожалуйста, этот тест!

Литература

Основная литература

1. Таненбаум Э. Современные операционные системы. 2-е изд. / Э. Таненбаум. – СПб. : Питер, 2004. – 1040 с.

2. Рихтер, Дж. C#. Создание эффективных Win-32 приложений с учетом специфики 64-разрядной версии Windows / Дж. Рихтер. Пер. с англ., 4-е изд., СПб Питер: Москва. Из-во торг. дом «Русская редакция», 2001. – 752 с.

3. Рихтер, Дж. C#. Программирование серверных приложений для Microsoft Windows 2000. Мастер-класс / Дж. Рихтер, Дж. Д. Кларк. Пер. с англ., 4-е изд., СПб Питер: Москва. Из-во торг. дом «Русская редакция», 2001. – 592 с.

4. Русинович, М. Внутреннее устройство Microsoft Windows: Windows Server 2003, Windows XP и Windows 2000. Мастер-класс / М. Русинович, Д. Соломон. Пер. с англ., 4-е изд., Москва: Из-во торг. дом «Русская редакция», СПб: Питер 2006. – 992 с.

5. Побегайло А.П. Системное программирование в Windows / А. П. Побегайло — СПб.: Питер, 2006. — 1056 с.

6. Харт., Джонсон М. Системное программирование в среде Win32, 2-е изд. /Москва. Пер. с англ.: Изд. Дом «Вильямс», 2001. – 464 с.

7. Вахалие, Ю. Unix изнутри / Ю. Вахалие — СПб.: Питер, 2003. — 844 с.

8. Безверхий А.А. Введение в операционные системы. Учебное пособие / А. А. Безверхий, С.И. Кашкевич — Минск, УП «ИВЦ Минфина». — 168с.

9. Silberschatz A. Operating System Principles (Seventh Edition) / A. Silberschatz, P.B. Galvin, G. Gagne — Copyright 2006 by John Wiley&Sons (Asia) Pte.Ltd.

10. Лав, Р. Linux. Системное программирование / Р. Лав — СПб.: Питер, 2014. — 448 с.

11. Линн, С. Администрирование Microsoft Windows Server 2012 / С. Линн — СПб.: Питер, 2014. — 314 с.

Дополнительная литература

12. Вудхалл, А. Операционные системы: разработка и реализация/ А. Вудхалл, Э. Таненбаум: Питер, 2006. – 576 с.

13. Дейтел, Х.М. Операционные системы. Основы и принципы: Третье издание / Х.М. Дейтел, П.Дж. Дейтел, Д.Р. Чофенс. Пер. с англ. – Москва: ООО «Бином-Пресс», 2006. – 1024 с.

14. Дейтел, Х.М. Операционные системы. Распределенные системы, сети, безопасность: Третье издание / Х.М. Дейтел, П.Дж. Дейтел, Д.Р. Чофенс. Пер. с англ. – Москва: ООО «Бином-Пресс», 2007. – 704 с.

15. Чан, Т. Системное программирование на C++ для Unix / Т. Чан. Пер. с англ. – Киев: Издательский отдел BHV, 1997.

16. Робачевский, А. Операционная система Unix / А. Робачевский. — БХВ.: Питер, 1999.

17. Стахов, А.А. Linux / А.А. Стахов – СПб.: БХВ – Питер, 2003. – 912 с.

18. Бэкон, Д. Операционные системы / Д. Бэкон, Г. Харрис. Пер. с англ. – Спб.: Питер; Киев. Изд. Гр. BHV, 2004. – 800 с.

19. Столлингс, В., Операционные системы / В. Столингс. Вильямс, 2002. – 848 с.

20. Вильямс, А. Системное программирование в Windows 2000 / А. Вильямс. СПб.: Питер, 2001. – 624 с.

21. Петцольд, Ч. Программирование для Microsoft Windows – 8 (6 – е издание)/ Ч. Петцольд – СПб.: – Питер, 2014. – 1008 с.