Основные понятия, системные вызовы, структура операционной системы

Лекция

Ревизия: 0.1

Последняя модификация: Vlad Kovtun

© NRJETIX 2000 - 2010

Дата последней модификации: 21.02.2010 13:01:00

История изменений

28.10.2010 – Версия 0.1. Первичный документ. Ковтун В.Ю.

Содержание

История изменений 2
Содержание 3
Лекция 2. Системные вызовы, структура операционной системы 4
Вопросы 4
Системные вызовы 4
СВ управления процессами 6
Базовые понятия операционной системы 6
Процессы 7
Взаимоблокировки 8
Управление памятью 8
Ввод-вывод данных 8
Файлы 8
Безопасность 11
Оболочка 11
Структура ОС 11
Монолитные ОС 11
Многоуровневые системы 12
Виртуальные машины 13
Экзоядро 15
Модель клиент-сервер 15
Литература 16

Лекция 2. Системные вызовы, структура операционной системы

Вопросы

1. Системные вызовы.

2. Понятие ОС.

3. Структура ОС.

Системные вызовы

Интерфейс между ОС и программами пользователя определяется набором **системных вызовов** (СВ), предоставляемых ОС. Чтобы на самом деле понять, что же делает, ОС, следует подробно рассмотреть этот интерфейс. Системные вызовы, доступные в интерфейсе, меняются от одной ОС к другой (хотя лежащая в их основе концепция практически одинакова).

Существует проблема выбора между (1) **неопределенными обобщениями** («ОС имеют системные вызовы для чтения файлов») и (2) **какой-либо конкретной системой** («в UNIX существует СВ для чтения с тремя параметрами: один для задания файла, второй — для того, чтобы указать, куда нужно поместить прочитанные данные, третий задает количество байтов, которое нужно прочитать»).

При втором способе следует проделать больше работы, но он обеспечивает лучшее понимание того, что в реальности происходит в ОС. Несмотря на то, что это обсуждение затрагивает конкретно стандарт POSIX (международный стандарт 9945-1), а, следовательно, также и ОС UNIX, System V, BSD, Linux, MINIX и т. д., у большинства других современных ОС есть СВ, выполняющие те же самые функции, хотя детали могут быть различны. Так фактический механизм обращения к системным функциям является в высокой степени машинно-зависимым и часто должен реализовываться на ассемблере, существуют библиотеки процедур, делающие возможным обращение к системным процедурам из программ на С и других языках с тем же успехом.

Полезно помнить следующее: **Любой компьютер с одним процессором в каждый конкретный момент времени может выполнить только одну команду**. Если процесс выполняет программу пользователя в пользовательском режиме и нуждается в системной службе, например чтении данных из файла, он должен выполнить прерывание или команду СВ для передачи управления ОС. Затем ОС по параметрам вызова определяет, что требуется вызывающему процессу. После этого она обрабатывает СВ и возвращает управление команде, следующей за СВ. В известном смысле выполнение СВ похоже на осуществление вызова процедуры, только первый проникает в ядро, а второй этого не делает.

Для того чтобы прояснить механизм СВ, кратко рассмотрим СВ `read`. Как упоминалось выше, у него есть три параметра: первый служит для задания файла, второй указывает на буфер, третий задает количество байтов, которое нужно прочитать. Как практически все СВ, он запускается из программы на С с помощью вызова библиотечной процедуры с тем же именем, что и СВ: read. Вызов из программы на С может выглядеть так:

```
count = read(fd, buffer, nbytes);
```

СВ (и библиотечная процедура) возвращает количество действительно прочитанных байтов в переменной `count`. Обычно эта величина совпадает с параметром `nbytes`, но может быть меньше, если, например, в процессе чтения процедуре встретился конец файла.

Если СВ не может быть выполнен или из-за неправильных параметров или из-за дисковой ошибки, значение счетчика `count` устанавливается равным -1, а номер ошибки помещается в глобальную переменную `errno`. Программы всегда должны проверять результат СВ, чтобы отслеживать появление ошибки.

СВ выполняются за серию шагов. Вернемся к упоминавшемуся выше примеру вызова `read` для того, чтобы разъяснить этот момент. Сначала при подготовке к вызову библиотечной процедуры `read`, которая фактически осуществляет СВ `read`, вызывающая программа помещает параметры в стек, как показано в шагах 1-3 на рис. 1. Компиляторы Си С++ помещают параметры в стек в обратном порядке, так исторически сложилось (чтобы первым был параметр для `printf`, то есть строка формата оказалась на вершине стека). Первый и третий параметры передаются по

значению, а второй параметр передается по ссылке, то есть передается адрес буфера (на то, что это ссылка, указывает символ &), а не его содержимое. Затем следует собственно вызов библиотечной процедуры (шаг 4). Эта команда процессора представляет собой обычную команду вызова процедуры и применяется для вызова любых процедур.

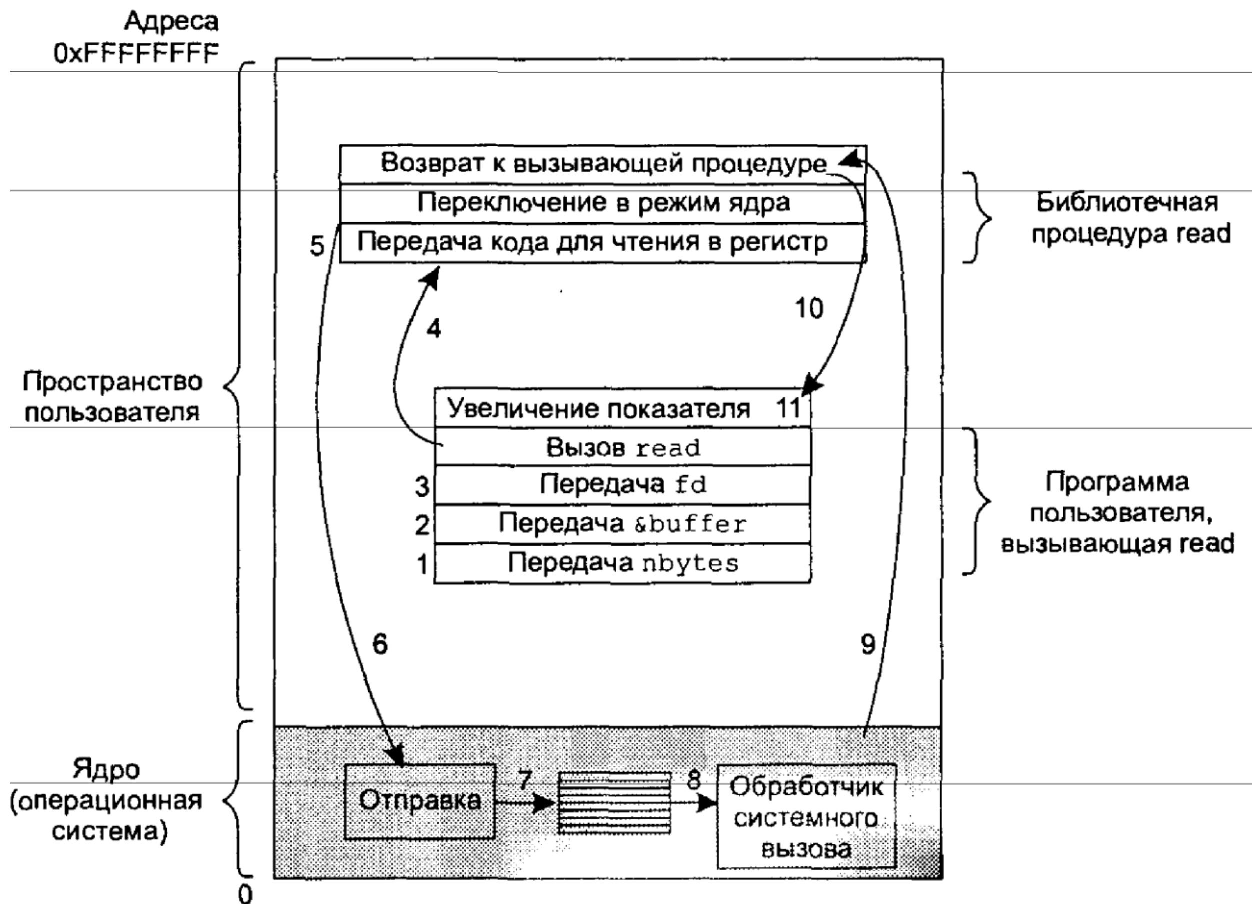

Рис. 1. Этапы вызова системного вызова `read(fd, buffer, nbytes)`

Библиотечная процедура, возможно, написанная на ассемблере, обычно помещает номер СВ туда, где его ожидает ОС, например в регистр (шаг 5). Затем она выполняет команду `TRAP` (эмулированное прерывание) для переключения из пользовательского режима в режим ядра и начинает выполнение с фиксированного адреса внутри ядра (шаг 6). Запускаемая программа ядра проверяет номер СВ и затем отправляет его нужному обработчику, как правило, используя таблицу указателей на обработчики СВ, индексированную по номерам вызовов (шаг 7). В этом месте начинает функционировать обработчик СВ (шаг 8). Как только он завершает свою работу, управление может возвращаться в пространство пользователя к библиотечной процедуре, к команде, следующей за командой `TRAP` (шаг 9). Эта процедура в свою очередь передает управление программе пользователя обычным способом, которым производится возврат из вызванной процедуры (шаг 10). Чтобы закончить работу, программа пользователя должна очистить стек, как это делается и после каждого вызова процедуры (шаг 11). Учитывая, что стек растет вниз, последняя команда увеличивает указатель стека ровно настолько, насколько нужно для удаления параметров, помещенных в стек перед запросом `read`. Теперь программа может продолжать свою работу.

На шаге 9 использовалось выражение «может возвращаться в пространство пользователя к библиотечной процедуре...» не просто так. СВ может блокировать вызвавшую его процедуру, препятствуя продолжению ее работы. Например, если она пытается прочесть что-то с клавиатуры, а там еще ничего не набрано, процедура должна быть блокирована. В этом случае ОС ищет процесс, который может быть запущен следующим. Позже, когда нужное устройство станет доступно, система вспомнит о блокированном процессе и шаги 9-11 будут выполнены.

В ОС присутствуют СВ для управления всеми ресурсами ОС, например:

- Процессами и потоками.
- Памятью.
- Файловой системой, в частности файлами и каталогами.
- Вводом-выводом.
- И т.д.

В качестве примера рассмотрим СВ для управления процессами в ОС.

СВ управления процессами

В каждой ОС предусмотрены СВ для управления процессами.

ОС Windows (Win32 API) [5]	FreeBSD, Linux, QNX [6]
• CreateProcess • CreateProcessAsUser • CreateProcessWithLogon • CreateProcessWithToken • ExitProcess • GetCurrentProcess • GetCurrentProcessId • GetExitCodeProcess • GetLogicalProcessorInformation & etc. • GetPriorityClass/ SetPriorityClass • GetProcessAffinityMask/ SetProcessAffinityMask • GetProcessHandleCount • GetProcessId • GetProcessIdOfThread • GetProcessIoCounters • GetProcessPriorityBoost/ SetProcessPriorityBoost • GetProcessShutdownParameters/ SetProcessShutdownParameters • GetProcessTime • GetProcessVersion • GetProcessWorkingSetSize/ SetProcessWorkingSetSize • GetStartupInfo • IsProcessInJob • NeedCurrentDirectoryForExePath • NtQueryInformationProcess • OpenProcess • TerminateProcess • WaitForInputIdle • WinExec	• system • exec (и все вариации) • spawn (и все вариации) • fork/vfork

Базовые понятия операционной системы

Для каждой ОС существует набор базовых понятий:

- процессы;
- память;

- файлы;

которые являются самыми важными для понимания общей идеи.

Процессы

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

Рассмотрим систему, работающую в режиме разделения времени. Периодически ОС решает остановить работу одного процесса и запустить другой, потому что первый израсходовал отведенную для него часть рабочего времени CPU в прошедшую секунду.

Если процесс был приостановлен подобным образом, позже он должен быть запущен заново из того же состояния, в каком его остановили. Следовательно, всю информацию о процессе нужно где-либо явно сохранить на время его приостановки. Например, процесс может иметь открытыми для чтения несколько файлов одновременно. Связанный с каждым файлом указатель дает текущую позицию (то есть номер байта или записи, которые будут прочитаны следующими). При временном прекращении процесса все указатели нужно сохранить так, чтобы команда чтения, выполненная после возобновления процесса, прочла правильные данные. Во многих ОС вся информация о каждом процессе, дополнительная к содержимому его собственного адресного пространства, хранится в таблице ОС. Эта таблица называется **таблицей процессов** и представляет собой массив (или связанный список) структур, по одной на каждый существующий в данный момент процесс.

Таким образом, приостановленный процесс состоит из собственного адресного пространства, обычно называемого образом памяти (**core image**), и компонентов таблицы процесса, содержащей, помимо других величин, его регистры.

Главными системными вызовами, управляющими процессами, являются вызовы, связанные с созданием и окончанием процессов.

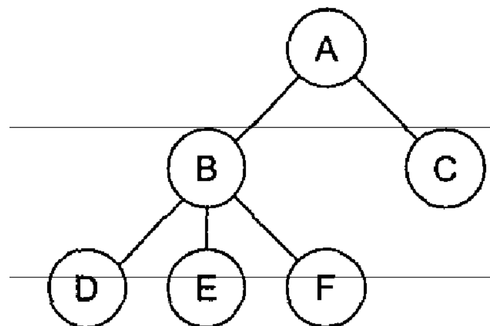

Рис. 2. Иерархия процессов в ОС. Процесс A создал два дочерних процесса B и C. Процесс B создал три дочерних процесса D, E и F

На рис. 2 представлена иерархия процессов. Процессы (родительские) могут создавать дочерние процессы. Отметим, что многие процессы могут взаимодействовать между собой посредством механизмов межпроцессного взаимодействия (IPC – Inter Process Communications) и синхронизированы между собой.

Каждый процесс в системе обладает идентификатором (PID – Process ID), родители и дети обладают одинаковым PID. Процессы могут быть организованы в группы с собственным идентификатором (GID – Group ID).

Более детально, процессы будут рассмотрены в дальнейших лекциях.

Взаимоблокировки

При взаимодействии двух и более процессов, они могут попадать в патовые ситуации, из которых невозможно выйти без посторонней помощи – **тупиковая ситуация (взаимоблокировка)**.

Процессы в ОС могут попадать в аналогичные ситуации, в которых они не могут выполняться дальше. Например, компьютер с накопителем на магнитной ленте и устройством записывающим компакт-диски. Теперь, каждому из двух процессов нужно записать данные с ленты на компакт-диск. Процесс 1 запрашивает и получает в пользование устройство с лентой. Затем процесс 2 запрашивает и получает устройство для записи компакт-дисков. После этого процесс 1 запрашивает устройство для записи компакт-дисков и приостанавливается до тех пор, пока процесс 2 не освободит его. Наконец, процесс 2 запрашивает устройство с лентой и также останавливается на время, потому что магнитофон уже занят процессом 1. Перед нами типичный тупик, из которого нет выхода. Более подробно, взаимоблокировки будут рассмотрены в дальнейших лекциях.

Управление памятью

В каждом компьютере есть оперативная память, используемая для хранения выполняющихся программ. В очень простых ОС в конкретный момент времени в памяти может находиться только одна программа. Для запуска второй программы сначала нужно удалить из памяти первую и загрузить на ее место вторую.

Более изощренные системы позволяют одновременно находиться в памяти нескольким программам. Для того чтобы они не мешали друг другу (и ОС), необходим некий защитный механизм. Хотя этот механизм располагается в аппаратуре, он управляется ОС.

Вышеизложенная точка зрения имеет отношение к управлению оперативной памятью компьютера и к ее защите. Другой, но не менее важный, связанный с памятью вопрос — это управление адресным пространством процессов. Обычно под каждый процесс отводится некоторый набор адресов, которые он может использовать, чаще всего начинающийся с 0 и продолжающийся до некого максимума. В простейшем случае максимальная величина адресного пространства для процесса меньше основной памяти. Тогда процесс может заполнить свое адресное пространство, и памяти хватит на то, чтобы содержать его целиком.

Однако на многих компьютерах адресация 32- или 64-разрядная, что дает для пространства адресов 2^{32} и 2^{64} байтов соответственно. Что произойдет, если адресное пространство процесса окажется больше, чем оперативная память компьютера, и процесс захочет использовать его целиком? На первых компьютерах подобным процессам просто не везло. В наши дни существует метод, называемый виртуальной памятью, при котором ОС держит часть адресов в оперативной памяти, а часть на диске и меняет их местами при необходимости. Эта важная функция ОС, а также другие понятия, связанные с управлением памятью, будут рассмотрены в следующих лекциях.

Ввод-вывод данных

Во всех компьютерах есть физическое устройство для получения входных данных и вывода информации. Посудите сами, что хорошего было бы в компьютере, если бы пользователи не могли сказать ему, что делать, и не могли получить результаты после завершения выполненной работы? Существует много видов устройств ввода-вывода, например клавиатуры, мониторы, принтеры и т.д. Всеми ими должна управлять ОС.

Каждая ОС имеет свою подсистему ввода-вывода для управления устройствами ввода-вывода. Некоторые из программ ввода-вывода являются независимыми от устройств, то есть их можно применить ко многим или ко всем устройствам ввода-вывода. Другая часть ПО ввода-вывода, в которую входят драйверы устройств, предназначена для определенных устройств ввода-вывода. В следующих лекциях, рассмотрим ПО ввода-вывода данных.

Файлы

Файловая система — это еще одно ключевое понятие, поддерживаемое виртуально всеми ОС. Как было замечено ранее, основной функцией ОС является скрытие особенностей функционирования с дисками и другими устройствами ввода-вывода и предоставление пользователю понятной и удобной абстрактной модели независимых от

устройств файлов. Системные вызовы очевидно необходимы для создания, удаления, чтения или записи файлов. Перед тем как прочитать файл, его нужно разместить на диске и открыть, а после прочтения его нужно закрыть. Все эти функции осуществляют системные вызовы.

Предоставляя место для хранения файлов, ОС используют понятие каталога (directory) как способ объединения файлов в группы. Например, студент может иметь по одному каталогу для каждого изучаемого им курса (для программ, необходимых в рамках этого курса), каталог для электронной почты, и еще один — для своей домашней web-страницы. Для создания и удаления каталогов также необходимы системные вызовы. Они же обеспечивают перемещение существующего файла в каталог и удаление файла из каталога. Содержимое каталогов могут составлять файлы или другие каталоги. Эта модель создает структуру — файловую систему, — как показано на рис. 6.

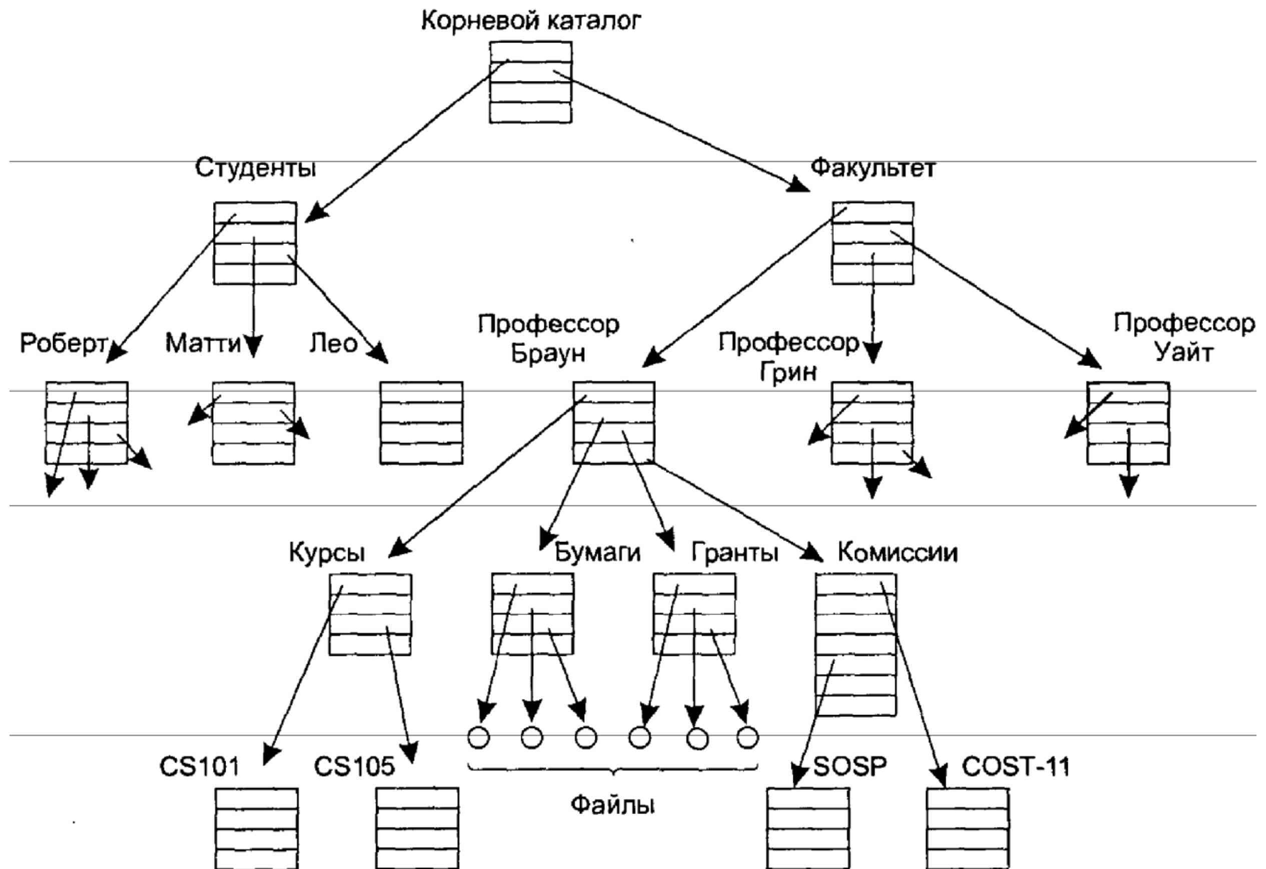

Рис. 3. Файловая система факультета

Иерархии процессов и файлов организованы в виде деревьев, но на этом сходство заканчивается. Иерархия процессов обычно не очень глубока (в ней редко бывает больше трех уровней), тогда как файловая структура достаточно часто имеет четыре, пять или даже больше уровней в глубину. Иерархия процессов обычно живет очень недолго, как правило, несколько минут, иерархия каталогов может существовать годами. Принадлежность и защита, также, различны для процессов и файлов. Обычно только родительский процесс может управлять или даже просто иметь доступ к дочернему процессу, однако практически всегда существует механизм, позволяющий читать файлы и каталоги не только владельцу файла, а более широкой группе пользователей.

В каждый момент времени у каждого процесса есть текущий **рабочий каталог**, в котором ищутся пути файлов. Процессы могут изменять свой рабочий каталог, используя системные вызовы.

Перед тем как прочесть или записать файл, его нужно открыть, в это же время проверяется разрешение доступа. Если доступ разрешен, система возвращает небольшое целое число, называемое **дескриптором файла** и используемое в последующих операциях. Если доступ запрещен, то возвращается код ошибки.

Другое важное понятие в UNIX — **это установленная (смонтированная) файловая система**. Почти все персональные компьютеры имеют один или два жестких диска, а

также съемные носители (flash-диски, CD/DVD). Чтобы предоставить возможность общения со сменными носителями, UNIX позволяет присоединять файловую систему съемного диска к главному дереву. Рассмотрим ситуацию на рис. 4, а. Перед вызовом системной процедуры `mount` корневая файловая система на жестком диске и вторая файловая система на съемном диске существуют раздельно и никак не связаны между собой.

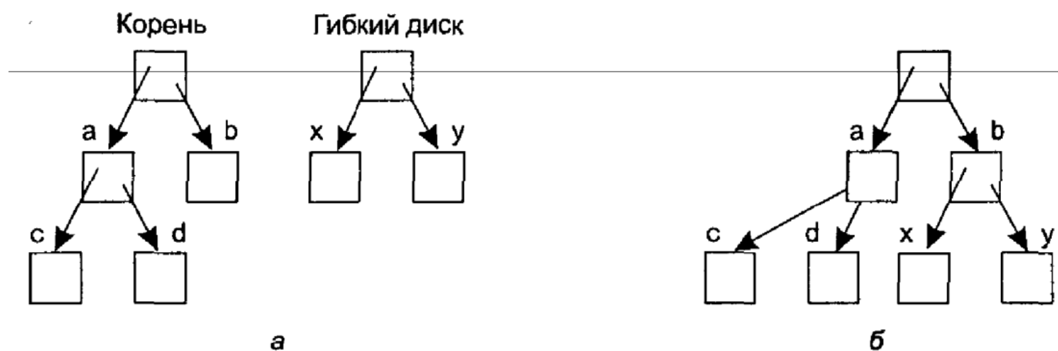

Рис. 4. Перед установкой файлы на диске 0 недоступны (а); после монтирования они становятся частью общей файловой структуры (б)

Однако файлы на гибком диске нельзя использовать, потому что для них невозможно определить путь. UNIX не позволяет присоединять к началу пути название диска или его номер, так как это привело бы к жесткой зависимости от устройств, которой ОС должна избегать. Вместо этого системный вызов `mount` позволяет присоединять файловую систему на съемном диске к корневой файловой системе в том месте, где этого захочет программа. На рис. 4, файловая система съемного диска была установлена в каталог **b**, таким образом, обеспечен доступ к файлам по путям **/b/x/** и **/b/y**. Если каталог **b** содержал какие-либо файлы, они будут недоступны, пока смонтирован гибкий диск, так как теперь **/b** ссылается на корневой каталог гибкого диска. (Невозможность доступа к этим файлам не так страшна, как кажется с первого взгляда: файловые системы почти всегда устанавливаются в пустые каталоги.) Если система содержит несколько жестких дисков, они все могут быть встроены в одно дерево таким же образом.

Еще одно важное понятие в UNIX — это **специальный файл** - служат для того, чтобы устройства ввода-вывода выглядели как файлы. При этом можно прочесть информацию из специальных файлов или записать ее туда с помощью тех же самых системных вызовов, что используются для чтения и записи файлов. Существует два вида специальных файлов:

- **блочные специальные файлы** - используются для моделирования устройств, состоящих из набора произвольно адресуемых блоков, таких как диски. Открывая их и читая, скажем, блок 4, программа может напрямую получить доступ к четвертому блоку на устройстве, без обращения к содержащейся на нем файловой системе.
- **символьные специальные файлы** - используются для моделирования принтеров, модемов и других устройств, которые принимают или выдают поток символов. По соглашению специальные файлы хранятся в каталоге `/dev`. Например, `/dev/lp` может быть строковым принтером.

Каналы (pipe), имеющие отношение и к процессам и к файлам. Канал (также иногда называемый трубой) представляет собой псевдофайл, который можно использовать для связи двух процессов, как показано на рис. 5. Если процессы А и В захотят пообщаться с помощью канала, они должны установить его заранее. Когда процесс А хочет отправить данные процессу В, он пишет их в канал, как если бы это был выходной файлом с входными данными. Таким образом, соединение между процессами в UNIX выглядит очень похожим на обычное чтение и запись файлов. Более того, только сделав специальный системный вызов, процесс может обнаружить, что выходной файл, в который он пишет данные, не реальный файл, а канал. Файловые системы очень важны.

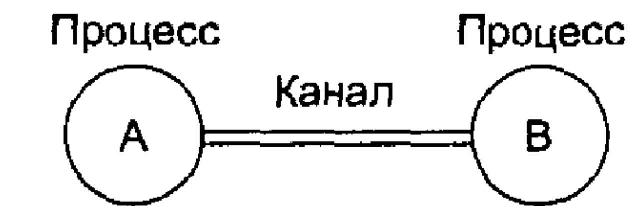

Рис. 5. Межпроцессное взаимодействие посредством канала

Безопасность

Компьютеры содержат большое количество информации, конфиденциальность которой пользователи зачастую хотят сохранить: электронную почту, бизнес-планы и многое другое. В задачу ОС входит управление системой защиты подобных файлов, так чтобы они, например, были доступны только пользователям, имеющим на это права.

В качестве простейшего примера, дающего представление о том, как работает система безопасности, рассмотрим систему UNIX. В UNIX для защиты файлов им присваивается 9-битовый двоичный код. Этот код защиты состоит из трех полей по три бита; одно для владельца, второе для других членов группы владельца (пользователи разделяются на группы системным администратором) и третье — для всех остальных. В каждом поле есть бит, определяющий доступ для чтения, бит, определяющий доступ для записи, и бит, разрешающий выполнение. Эти три бита называются rwx-битами (read, write, execute). Например, код защиты *rwxr-x—x* означает, что владелец файла может читать, писать или выполнять файл, другие члены группы могут читать или выполнять файл (но не писать в него), а остальные могут только выполнять файл (но не читать или писать). Для каталогов означает разрешение на поиск. Дефис означает, что соответствующее разрешение отсутствует.

Кроме защиты файлов, существует еще множество других вопросов безопасности: защита системы от нежелательных гостей, людей, и не только (вирусов). Обсудим различные вопросы, связанные с безопасностью, в следующих лекциях.

Оболочка

ОС представляет собой программу, выполняющую системные вызовы. Редакторы, компиляторы, ассемблеры, компоновщики и командные интерпретаторы не являются частью ОС, несмотря на их большую важность и полезность. Поскольку есть риск запутаться в этих вещах, в данном разделе мы кратко рассмотрим только командный интерпретатор UNIX, называемый **оболочкой (shell)** в MS DOS называют **режим командной строки**. Хотя она не входит в ОС, но во всю пользуется многими функциями ОС и поэтому является хорошим примером того, как могут применяться системные вызовы. Кроме этого, оболочка предоставляет основной интерфейс между пользователем, сидящим за своим терминалом, и ОС, если, конечно, пользователь не использует графический интерфейс.

Структура ОС

Монолитные ОС

В общем случае организация монолитной системы представляет собой «большой беспорядок». То есть структура как таковая отсутствует. ОС написана в виде набора процедур, каждая из которых может вызывать другие, когда ей это нужно. При использовании такой техники каждая процедура системы имеет строго определенный интерфейс в терминах параметров и результатов, и каждая имеет возможность вызвать любую другую для выполнения некоторой необходимой для нее работы.

Для построения монолитной системы необходимо скомпилировать все отдельные процедуры, а затем связать их в единый объектный файл с помощью компоновщика. Здесь, по существу, полностью отсутствует сокрытие деталей реализации — 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 иметь некоторую структуру. При обращении к СВ, поддерживаемым ОС, параметры помещаются в строги определенные

места — регистры или стек, после чего выполняется специальная команда прерывания, известная как вызов ядра или вызов супервизора. Эта команда переключает машину из режима пользователя в режим ядра и передает управление ОС, что видно на шаге 6 рис. 6. Затем ОС проверяет параметры вызова, чтобы определить, какой системный вызов должен быть выполнен. После этого ОС обращается к таблице как к массиву с номером системного вызова в качестве индекса. В i-ом элементе таблицы содержится ссылка на процедуру обработки системного вызова k (шаг 7 на рис. 9). Такая организация ОС предполагает следующую структуру:

1. Главная программа, которая вызывает требуемую служебную процедуру.

2. Набор служебных процедур, выполняющих СВ.

3. Набор утилит, обслуживающих служебные процедуры.

В этой модели для каждого системного вызова имеется одна служебная процедура. Утилиты выполняют функции, которые нужны нескольким служебным процедурам. Деление процедур на три уровня показано на рис. 6.

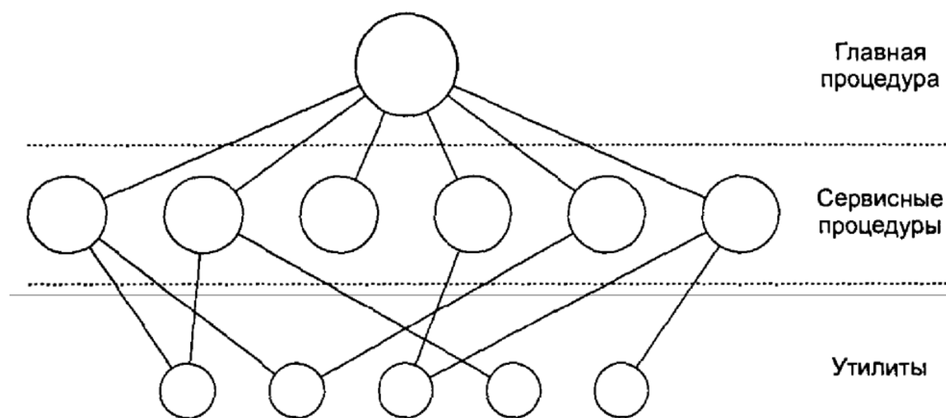

Рис. 6. Простая модель монолитной ОС

Многоуровневые системы

Обобщением подхода, изображенного на рис. 6, является организация ОС в виде иерархии уровней. Первой системой, построенной таким образом, была система THE, созданная в Technische Hogeschool Eindhoven (Нидерланды) Э. Дейкстрой (E.W. Dijkstra) и его студентами в 1968 году. Она была простой пакетной системой для голландского компьютера Electrologica X8, память которого состояла из 32К 27-разрядных слов. Система включала 6 уровней, как показано в таблице 1. Уровень 0 занимался распределением времени процессора, переключая процессы при возникновении прерывания или при срабатывании таймера. Над уровнем 0 система состояла из последовательных процессов, каждый из которых можно было запрограммировать, не заботясь о том, что на одном процессоре запущено несколько процессов. Другими словами, уровень 0 обеспечивал базовую многозадачность процессора.

Таблица 1. Структура операционной системы THE

Уровень	Функция
5	Оператор
4	Программы пользователя
3	Управление вводом-выводом
2	Связь оператор-процесс
1	Управление памятью и барабаном
0	Распределение процессора и многозадачность

Уровень 1 управлял памятью. Он выделял процессам пространство в оперативной памяти и на магнитном барабане объемом 512К слов для тех частей процессов

(страниц), которые не помещались в оперативной памяти. Процессы более высоких уровней не заботились о том, находятся ли они в данный момент в памяти или на барабане. Программное обеспечение уровня 1 обеспечивало попадание страниц в оперативную память по мере необходимости.

Уровень 2 управлял связью между консолью оператора и процессами. Таким образом, все процессы выше этого уровня имели свою собственную консоль оператора. Уровень 3 управлял устройствами ввода-вывода и буферизовал потоки информации к ним и от них. Любой процесс выше уровня 3, вместо того чтобы работать с конкретными устройствами, с их разнообразными особенностями, мог обращаться 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.

Виртуальные машины

Исходная версия OS/360 была системой исключительно пакетной обработки. Однако множество пользователей OS/360 желали работать в системе с разделением времени, поэтому различные группы программистов как в самой корпорации IBM, так и вне ее решили написать для этой машины системы с разделением времени. Официальная система с разделением времени от IBM, которая называлась TSS/360, поздно вышла в свет и оказалась настолько громоздкой и медленной, что на нее перешли немногие. В конечном счете от нее отказались, но уже после того, как ее разработка потребовала около 50 млн. долларов. Группа из научного центра IBM в Кембридже, штат Массачусетс, разработала в корне отличающуюся от нее систему, которую IBM в результате приняла как законченный продукт. Сейчас она широко используется на еще оставшихся мэйнфреймах.

Эта система, в оригинале называвшаяся CP/CMS, а позже переименованная в VM/370, была основана на следующем проницательном наблюдении: система с разделением времени обеспечивает (1) многозадачность и (2) расширенную машину с более удобным интерфейсом, чем тот, что предоставляется оборудованием напрямую. VM/370 основана на полном разделении этих двух функций.

Сердце системы, называемое **монитором виртуальной машины**, работает с оборудованием и обеспечивает многозадачность, предоставляя верхнему слою не одну, а несколько виртуальных машин, как показано на рис. 7. Но, в отличие от всех других ОС, эти виртуальные машины не являются расширенными. Они не поддерживают файлы и прочие удобства, а представляют собой точные копии голой аппаратуры, включая режимы ядра и пользователя, ввод-вывод данных, прерывания и все остальное, присутствующее на реальном компьютере.

Поскольку каждая виртуальная машина идентична настоящему оборудованию, на каждой из них может работать любая ОС, которая запускается прямо на аппаратуре. На разных виртуальных машинах могут (а зачастую так и происходит) функционировать различные ОС. На некоторых из них для обработки пакетов и транзакций работают потомки OS/360, а на других структура VM/370 с системой CMS для интерактивного разделения времени пользователей работает однопользовательская интерактивная система CMS (Conversational Monitor System — система диалоговой обработки).

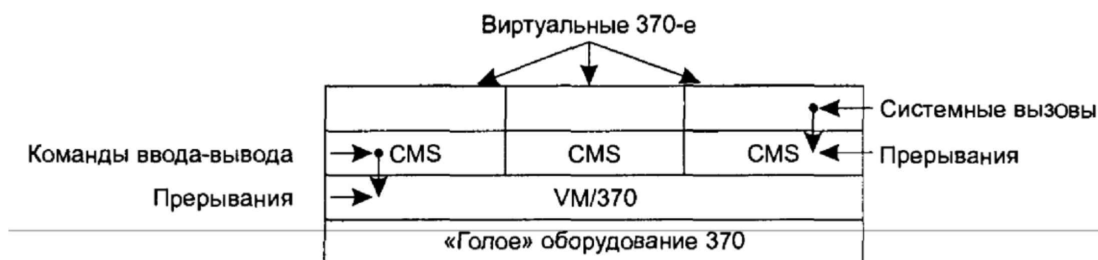

Рис. 7. Структура VM/370 с системой CMS

Когда программа ОС CMS выполняет системный вызов, он прерывает ОС на своей собственной виртуальной машине, а не на VM/370, как произошло бы, если бы он работал на реальной машине, вместо виртуальной. Затем CMS выдает обычные команды ввода-вывода для чтения своего виртуального диска или другие команды,

которые ей могут понадобиться для выполнения Вызова. Эти Команды ввода-вывода перехватываются VM/370, которая выполняет их в рамках моделирования реального оборудования. При полном разделении функций многозадачности и предоставления расширенной машины каждая часть может быть намного проще, гибче и удобней для обслуживания.

Идея виртуальной машины очень часто используется в наши дни, но в несколько другом контексте: для работы старых программ, написанных для системы MS-DOS на Pentium (или на других 32-разрядных процессорах Intel). При разработке компьютера Pentium и его программного обеспечения обе компании, Intel и Microsoft, понимали, что возникнет острая потребность в работе старых программ на новом оборудовании. Поэтому корпорация Intel создала на процессоре Pentium режим виртуального процессора 8086. В этом режиме машина действует как 8086 (которая с точки зрения программного обеспечения идентична 8088), включая 16-разрядную адресацию памяти с ограничением объема памяти в 1 Мбайт.

Такой режим используется системой Windows и другими ОС для запуска программ MS-DOS. Программы запускаются в режиме виртуального процессора 8086. Пока они выполняют обычные команды, они работают напрямую с оборудованием. Но когда программа пытается обратиться по прерыванию к ОС, чтобы сделать системный вызов, или пытается напрямую осуществить ввод-вывод данных, происходит прерывание с переключением на монитор виртуальной машины.

Возможны два варианта устройства:

- Сама система MS-DOS загружена в адресное пространство виртуальной машины 8086, так что монитор виртуальной машины только отсылает прерывания назад к MS-DOS, как это происходит на реальной 8086. Когда затем MS-DOS пытается самостоятельно осуществить ввод-вывод, операция перехватывается и выполняется монитором виртуальной машины.
- Монитор виртуальной машины перехватывает первое прерывание и сам выполняет ввод-вывод, так как он знает все системные вызовы MS-DOS и имеет представление о том, что должно делать каждое прерывание. Этот вариант не столь безупречен, как первый, потому что, в отличие от первого варианта, он корректно моделирует только MS-DOS и никакие другие ОС.

С другой стороны, он намного быстрее работает, так как избегает проблем запуска MS-DOS для выполнения ввода-вывода. Существует еще один недостаток фактического запуска MS-DOS в режиме виртуальной 8086: MS-DOS очень часто оперирует флагом разрешения/запрещения прерывания, а моделирование этого требует больших затрат.

Стоит отметить, что ни один из двух описанных методов в действительности не является те же самым, чем была VM/370, потому что смоделированная машина представляет собой только 8086, а не полноценный Pentium. В системе VM/370 можно было запустить на виртуальной машине саму VM/370. На Pentium нельзя запустить, скажем, ОС Windows на виртуальной 8086, потому что не существует версий Windows, работающих на этой машине. Даже для самых старых версий Windows необходим как минимум 286-й процессор, а моделирование 286 не поддерживается (не говоря уже об эмуляции Pentium). Однако, если немного модифицировать двоичный код Windows, подобная модель становится возможна, и даже возможна ее коммерческая реализация.

Кроме того, виртуальные машины используются, правда, несколько другим способом, для работы программ Java. Когда корпорация Sun Microsystems придумала язык программирования Java, она также разработала виртуальную машину (то есть архитектуру компьютера), называемую JVM (Java Virtual Machine — виртуальная машина Java). Компилятор Java выдает код для JVM, который затем обычно выполняется программным интерпретатором JVM, Преимущество этого подхода заключается в том, что код JVM можно передавать через Internet на любой компьютер, имеющий интерпретатор JVM, и запускать там. Если бы компилятор выдавал двоичные программы, например, для компьютеров SPARC или Pentium, их было бы нельзя куда-либо передать и запустить в работу так просто, как это происходит с JVM. (Конечно, компания Sun могла бы разработать компилятор, который выдавал бы двоичные коды SPARC, и затем использовать интерпретатор SPARC, но структура JVM намного проще для интерпретации.) Другое преимущество JVM заключается в том, что когда интерпретатор реализован должным образом, что вовсе не тривиально, приходящие JVM-программы можно проверить в целях безопасности и затем запустить в защищенной среде, так что эти программы не смогут похитить данные или причинить какой-нибудь иной вред.

Экзоядро

В системе VM/370 каждый пользователь получает точную копию настоящей машины. На Pentium, в режиме виртуальной машины 8086, каждый пользователь получает точную копию другой машины. Развив эту идею дальше, исследователи из Массачусетсского технологического института изобрели систему, которая обеспечивает каждого пользователя абсолютной копией реального компьютера, но с подмножеством ресурсов. Например, одна виртуальная машина может получить блоки на диске с номерами от 0 до 1023, следующая — от 1024 до 2047 и т. д.

На нижнем уровне в режиме ядра работает программа, которая называется **экзоядро (exokernel).** В ее задачу входит распределение ресурсов для виртуальных машин, а после этого проверка их использования (то есть отслеживание попыток машин использовать чужой ресурс). Каждая виртуальная машина на уровне пользователя может работать с собственной ОС, как на VM/370 или виртуальных 8086-х для Pentium, с той разницей, что каждая машина ограничена набором ресурсов, которые она запросила и которые ей были предоставлены.

Преимущество схемы экзоядра заключается в том, что она позволяет обойтись без уровня отображения. При других методах работы каждая виртуальная машина считает, что она использует свой собственный диск с нумерацией блоков от 0 до некоторого максимума. Поэтому монитор виртуальной машины должен поддерживать таблицы преобразования адресов на диске (и всех других ресурсов). Необходимость преобразования отпадает при наличии экзоядра, которому нужно только хранить запись о том, какой виртуальной машине выделен данный ресурс. Такой подход имеет еще одно преимущество: он отделяет многозадачность (в экзоядре) от ОС пользователя (в пространстве пользователя) с меньшими затратами, так как для этого ему необходимо всего лишь не допускать вмешательства одной виртуальной машины в работу другой.

Модель клиент-сервер

Система VM/370 сильно выигрывает в простоте благодаря переносу значительной части кода традиционной ОС (обеспечивающего расширенную машину) в верхний уровень, систему CMS. Однако VM/370 и при этом останется сложной комплексной программой, потому что моделирование нескольких виртуальных 370-х машин само по себе не так просто (особенно если вы хотите сделать это достаточно эффективно).

В развитии современных ОС наблюдается тенденция в сторону дальнейшего переноса кода в верхние уровни и удалении при этом всего, что только возможно, из режима ядра, оставляя минимальное микроядро. Обычно это осуществляется перекладыванием выполнения большинства задач ОС на средства пользовательских процессов. Получая запрос на какую-либо операцию, например чтение блока файла, пользовательский процесс (теперь называемый **обслуживаемым процессом** или **клиентским процессом)** посылает запрос **серверному** (обслуживающему) **процессу,** который его обрабатывает и высылает назад ответ.

В модели, показанной на рис. 8, в задачу ядра входит только управление связью между клиентами и серверами. Благодаря разделению ОС на части, каждая из которых управляет всего одним элементом системы (файловой системой, процессами, терминалом или памятью), все части становятся маленькими и управляемыми. К тому же, поскольку все серверы работают как процессы в режиме пользователя, а не в режиме ядра, они не имеют прямого доступа к оборудованию. Поэтому если происходит ошибка на файловом сервере, может разрушиться служба обработки файловых запросов, но это обычно не приводит к остановке всей машины целиком.

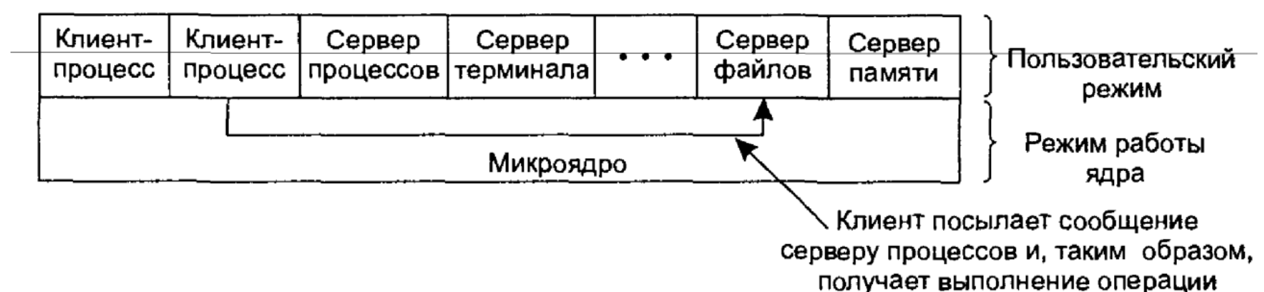

Рис. 8. Модель клиент-сервер

Другое преимущество модели клиент-сервер заключается в ее простой адаптации к использованию в распределенных системах (рис. 9). Если клиент общается с сервером, посылая ему сообщения, клиенту не нужно знать, обрабатывается ли его сообщение локально на его собственной машине или оно было послано по сети серверу на удаленной машине. С точки зрения клиента происходит одно и то же в обоих случаях; запрос был послан, и на него получен ответ.

Рассказанная выше история о ядре, управляющем передачей сообщений от клиентов к серверам и назад, не совсем реалистична. Некоторые функции ОС, такие как загрузка команд в регистры физических устройств ввода-вывода, трудно, если вообще возможно, выполнить из программ в пространстве пользователя. Есть два способа разрешения этой проблемы:

- Первый заключается в том, что некоторые критические серверные процессы (например, драйверы устройств ввода-вывода) действительно запускаются в режиме ядра, с полным доступом к аппаратуре, но при этом общаются с другими процессами при помощи обычной схемы передачи сообщений.
- Второй способ состоит в том, чтобы встроить минимальный механизм обработки информации в ядро, но оставить принятие политических решений за серверами в пользовательском пространстве. Например, ядро может опознавать сообщения, посланные по определенным адресам. Для ядра это означает, что нужно взять содержимое сообщения и загрузить его, скажем, в регистры ввода-вывода некоторого диска для запуска операции чтения диска.

В этом примере ядро даже может не обследовать байты сообщения, если они оказались допустимы или осмысленны; оно может вслепую копировать их в регистры диска. (Очевидно, должна использоваться некоторая схема, ограничивающая круг процессов, имеющих право отправлять подобные сообщения.) Разделение между механизмом и политикой является очень важным понятием, встречающимся в ОС в различном контексте постоянно.

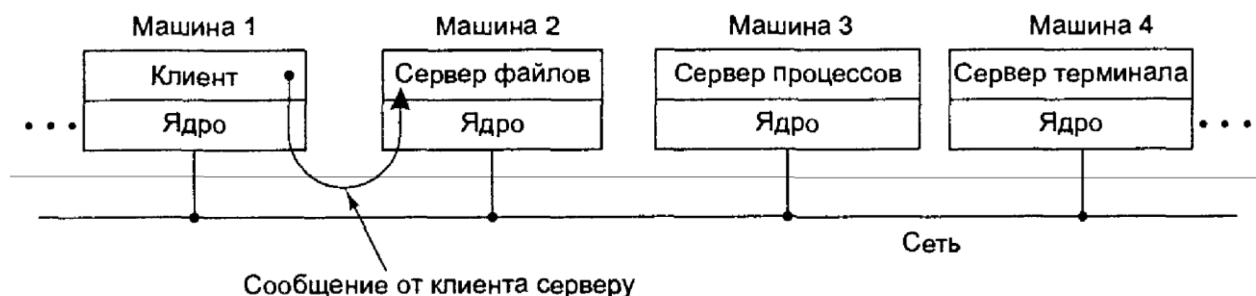

Рис. 9. Модель клиент-сервер в распределенной системе

Литература

1. Э. Таненбаум. Современные операционные системы. 2-ое изд. –СПб.: Питер, 2002. – 1040 с.

2. А. Шоу. Логическое проектирование операционных систем. Пер. с англ. –М.: Мир, 1981. –360 с.

3. С. Кейслер. Проектирование операционных систем для малых ЭВМ: Пер. с англ. –М.: Мир, 1986. –680 с.

4. Э. Таненбаум, А. Вудхалл. Операционные системы: разработка и реализация. Классика CS. –СПб.: Питер, 2006. –576 с.

5. Microsoft Development Network. URL: http://msdn.com

6. Роб Кертен. Введение в QNX/Neutrino 2. Руководство по программировани приложений реального времени в QNX Realtime Platform. –СПб.: Издательство «Петрополис», 2001. -480 с.