Операционные системы

Введение

Введение

Современный компьютер состоит из одного или нескольких процессоров, оперативной памяти, дисков, принтера, клавиатуры, мыши, дисплея, сетевых интерфейсов и других разнообразных устройств ввода-вывода => *получается довольно сложная система*

- Компьютеры оснащены специальным уровнем программного обеспечения, который называется **операционной системой,** в чью задачу входит управление пользовательскими программами, а также всеми ранее упомянутыми ресурсами.
- ОС работает непосредственно с аппаратным обеспечением и является основой остального программного обеспечения

Введение

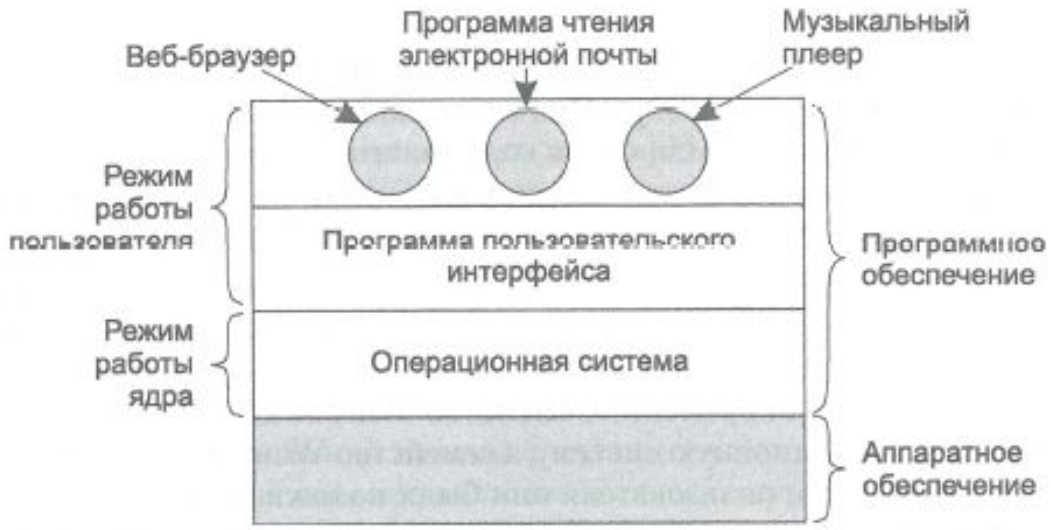
Примеры ОС: Windows, Linux или Max OS X

Программы пользовательского интерфейса (программы, с которыми взаимодействуют пользователи):

- **оболочка** – основана на применении текста (командный интерпретатор *cmd.exe* в Windows NT).
- **графический пользовательский интерфейс** (Graphical User Interface **(GUI))** – используются значки.

Данные программы фактически не являются частью операционной системы, хотя задействуют эту систему в своей работе.

Место операционной системы в структуре программного обеспечения



Программы пользовательского интерфейса — оболочка или GUI — находятся на самом низком уровне программного обеспечения, работающего в режиме пользователя, и по зволяют пользователю запускать другие программы, такие как веб-браузер, программа чтения электронной почты или музыкальный плеер

<u>Большинство компьютеров имеют два режима работы:</u>
режим ядра и режим пользователя.

- ОС – наиболее фундаментальная часть ПО, работающая в **режиме ядра** (этот режим называют еще **режимом супервизора).**
- ОС имеет полный доступ ко всему аппаратному обеспечению и может задействовать любую инструкцию, которую машина в состоянии выполнить.
- Вся остальная часть ПО работает в **режиме пользователя,** в котором доступно лишь подмножество инструкций машины.

Отличие ОС от обычного (работающего в режиме пользователя) ПО:

- Пользователь не может изменять программы, являющиеся частью ОС и защищенные на аппаратном уровне от любых попыток внесения изменений со стороны пользователя (например, обработчик прерываний системных часов).
- Работает в режиме ядра;
- ОС имеют большой объем, сложную структуру и длительные сроки использования (исходный код основы операционной системы типа Linux или Windows занимает порядка 5 млн строк)
- ОС сложно создавать и развиваются они в течении долгого времени

Операционная система в качестве менеджера ресурсов

Задачи операционной системы:

1. скрыть аппаратное обеспечение и существующие программы (и их разработчиков) под создаваемыми взамен них и приспособленными для нормальной работы красивыми, элегантными, неизменными абстракциями.

2. ОС в качестве менеджера ресурсов: обеспечить упорядоченное и управляемое распределение процессоров, памяти и устройств ввода-вывода между различными программами, претендующими на их использование.

Управление ресурсами

Управление ресурсами включает в себя **мультиплексирование** (распределение) ресурсов во времени и в пространстве.

- *во времени:* различные программы или пользователи используют ресурс по очереди: сначала ресурс получают в пользование одни, потом другие и т. д. (например, совместное использование принтера, процессора);
- *в пространстве:* каждый клиент получает какую-то часть разделяемого ресурса (например, оперативная память обычно делится среди нескольких работающих программ, или жесткий диск)

ОС выполняют две основные функции:

✔ предоставляют абстракции пользовательским программам

✔ управляют ресурсами компьютера.

- Взаимодействие пользовательских программ и операционной системы касается первой функции — взять, к примеру, операции с файлами: создание, запись, чтение и удаление.
- Управление ресурсами компьютера проходит большей частью незаметно для пользователей и осуществляется в автоматическом режиме.
- Т.о. интерфейс между пользовательскими программами и операционной системой строится в основном на абстракциях.

Понятия операционной системы

Процессы

- ***Процессом***, является программа во время ее выполнения.
- С каждым процессом связано его **адресное пространство** — список адресов ячеек памяти от нуля до некоторого максимума, откуда процесс может считывать данные и куда может записывать их. Адресное пространство содержит выполняемую программу, данные этой программы и ее стек.
- С каждым процессом связан набор ресурсов, который обычно включает регистры (в том числе счетчик команд и указатель стека), список открытых файлов, необработанные предупреждения, список связанных процессов и всю остальную информацию, необходимую в процессе работы программы.

*Т.о. **процесс** — это контейнер, в котором содержится вся информация, необходимая для работы программы.*

Запускаем программу редактирования видео и указываем конвертирование одночасового видеофайла в какой-нибудь определенный формат (процесс займет несколько часов), а затем переключаемся на блуждания по Интернету
Плюс: фоновый процесс, который периодически «просыпается» для проверки входящей электронной почты

Три активных процесса: видеоредактор, веб-браузер и программа получения (клиент) электронной почты

Действия ОС

- Периодически принимает решения остановить работу одного процесса и запустить выполнение другого.
- Если процесс приостанавливается, позже он должен возобновиться именно с того состояния, в котором был остановлен.

*Где хранить информацию о процессе чтобы вызов **read**, выполняемый после возобновления процесса, приводил к чтению нужных данных?*

Во многих ОС вся информация о каждом процессе, за исключением содержимого его собственного адресного пространства, хранится в таблице операционной системы, которая называется **таблицей процессов** и представляет собой массив (или связанный список) структур, по одной на каждый из существующих на данный момент процессов.

Процесс (в том числе приостановленный) состоит из:

- собственного адресного пространства, которое обычно называют **образом памяти;**
- записи в таблице процессов с содержимым его регистров, а также другой информацией, необходимой для последующего возобновления процесса.

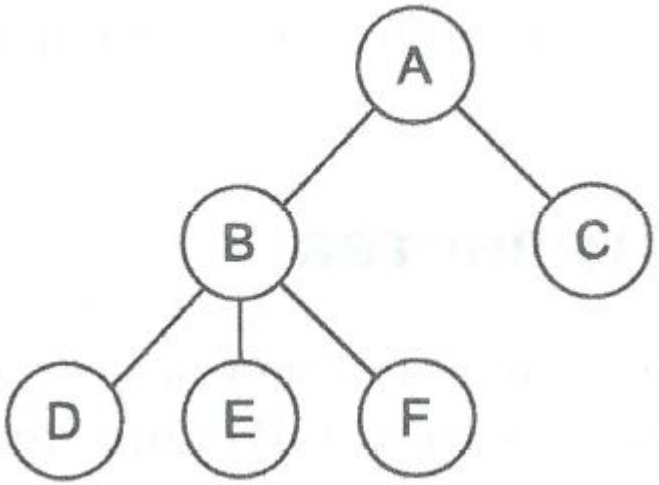
Системный вызов – обращение прикладной программы к ядру ОС для выполнения какой-либо операции.

- Архитектура современных процессоров предусматривает использование защищенного режима с несколькими уровнями привилегий: приложения обычно ограничены своим адресным пространством таким образом, что они не могут получить доступ или модифицировать другие приложения, исполняемые в операционной системе, либо саму операционную систему, и обычно не могут напрямую получать доступ к системным ресурсам (жесткие диски, видеокарта, сетевые устройства и т.д.).

- Для взаимодействия с системными ресурсами приложения используют системные вызовы, которые дают возможность операционной системе обеспечить безопасный доступ к ним.
- Системные вызовы передают управление ядру операционной системы, которое определяет предоставлять ли приложению запрашиваемые ресурсы. Если ресурсы доступны, то ядро выполняет запрошенное действие, затем возвращает управление приложению.

Главными системными вызовами, используемыми при управлении процессами, являются вызовы, связанные с созданием и завершением процессов.

Пример: Процесс, называемый **интерпретатором команд,** или **оболочкой,** считывает команды с терминала. Пользователь только что набрал команду, требующую ком пиляции программы. Теперь оболочка должна создать новый процесс, запускающий компилятор. Когда этот процесс завершит компиляцию, он произведет системный вызов для завершения собственного существования.



Дерево процессов.
Процесс *A* создал два дочерних процесса, B и C. Процесс *B* создал три дочерних процесса *D, E, F*

Если процесс способен создавать несколько других процессов (называющихся **дочерними процессами),** а эти процессы в свою очередь могут создавать собственные дочерние процессы, то перед нами предстает дерево процессов.

Связанные процессы, совместно работающие над выполнением какой-нибудь задачи, зачастую нуждаются в обмене данными друг с другом и синхронизации своих действий. Такая связь называется **межпроцессным взаимодействием.**

- Каждому пользователю, которому разрешено работать с системой, системным администратором присваивается **идентификатор пользователя** (User IDentification **(UID)).**
- Каждый запущенный процесс имеет UID того пользователя, который его запустил. Дочерние процессы имеют такой же UID, как и у родительского процесса. Пользователи могут входить в какую-нибудь группу, каждая из которых имеет собственный **идентификатор группы** (Group IDentification **(GID)).**
- Пользователь с особым значением UID, называемый в UNIX суперпользователем (**superuser**), а в Windows администратором (**administrator**), имеет особые полномочия, позволяющие пренебрегать многими правилами защиты.

Адресные пространства

- В самых простых ОС в памяти присутствует только одна программа. Для запуска второй программы сначала нужно удалить первую, а затем на ее место загрузить в память вторую.
- Более изощренные ОС позволяют одновременно находиться в памяти нескольким программам. Чтобы исключить взаимные помехи (и помехи работе опе рационной системы), нужен какой-то защитный механизм. Несмотря на то что этот механизм должен входить в состав оборудования, управляется он ОС.

Управление адресным пространством процессов

- В простейшем случае максимальный объем адресного про странства, выделяемого процессу, меньше объема оперативной памяти. => процесс может заполнить свое адресное пространство и для его размещения в оперативной памяти будет достаточно места.

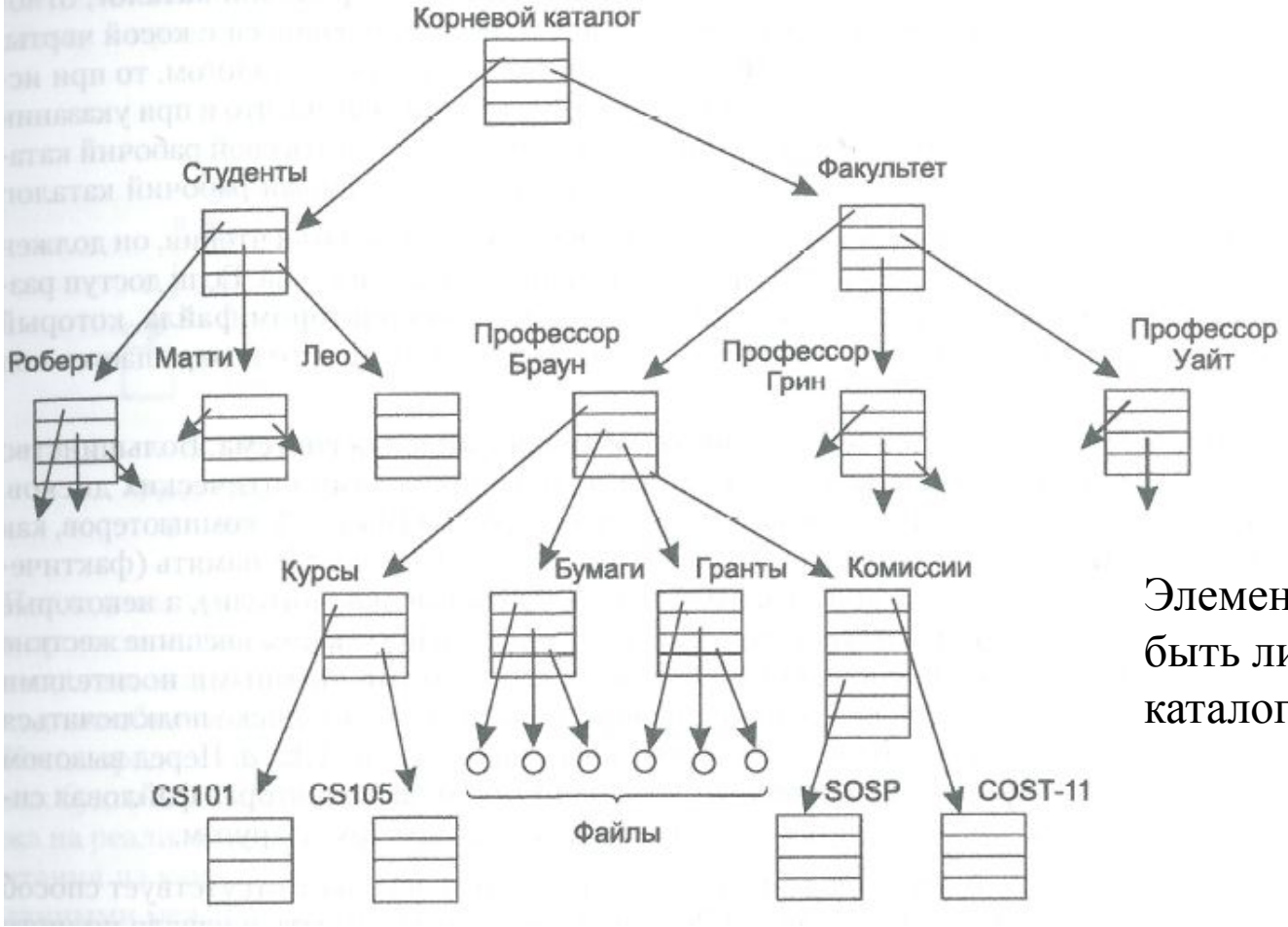
Что произойдет, если адресное пространство процесса превышает объем оперативной памяти, установленной на компьютере, а процессу требуется использовать все свое пространство целиком?

- В наше время, существует технология виртуальной памяти, при которой операционная система хранит часть адресного пространства в оперативной памяти, а часть — на диске, по необходимости меняя их фрагменты местами.
- ОС создает абстракцию адресного пространства в виде набора адресов, на которые может ссылаться процесс.
- Адресное пространство отделено от физической памяти машины и может быть как больше, так и меньше нее.

Файловая система

- Основная функция ОС — скрыть специфику дисков и других устройств ввода-вывода и предоставить программисту удобную и понятную абстрактную модель, состоящую из независимых от устройств файлов.
- Для создания, удаления, чтения и записи файлов предусмотрены ***системные вызовы***.
- Чтобы предоставить место для хранения файлов, многие операционные системы персональных компьютеров используют ***каталог*** как способ объединения файлов в группы.
- Для создания и удаления каталогов также используются системные вызовы.

Иерархической структура файловой системы университета

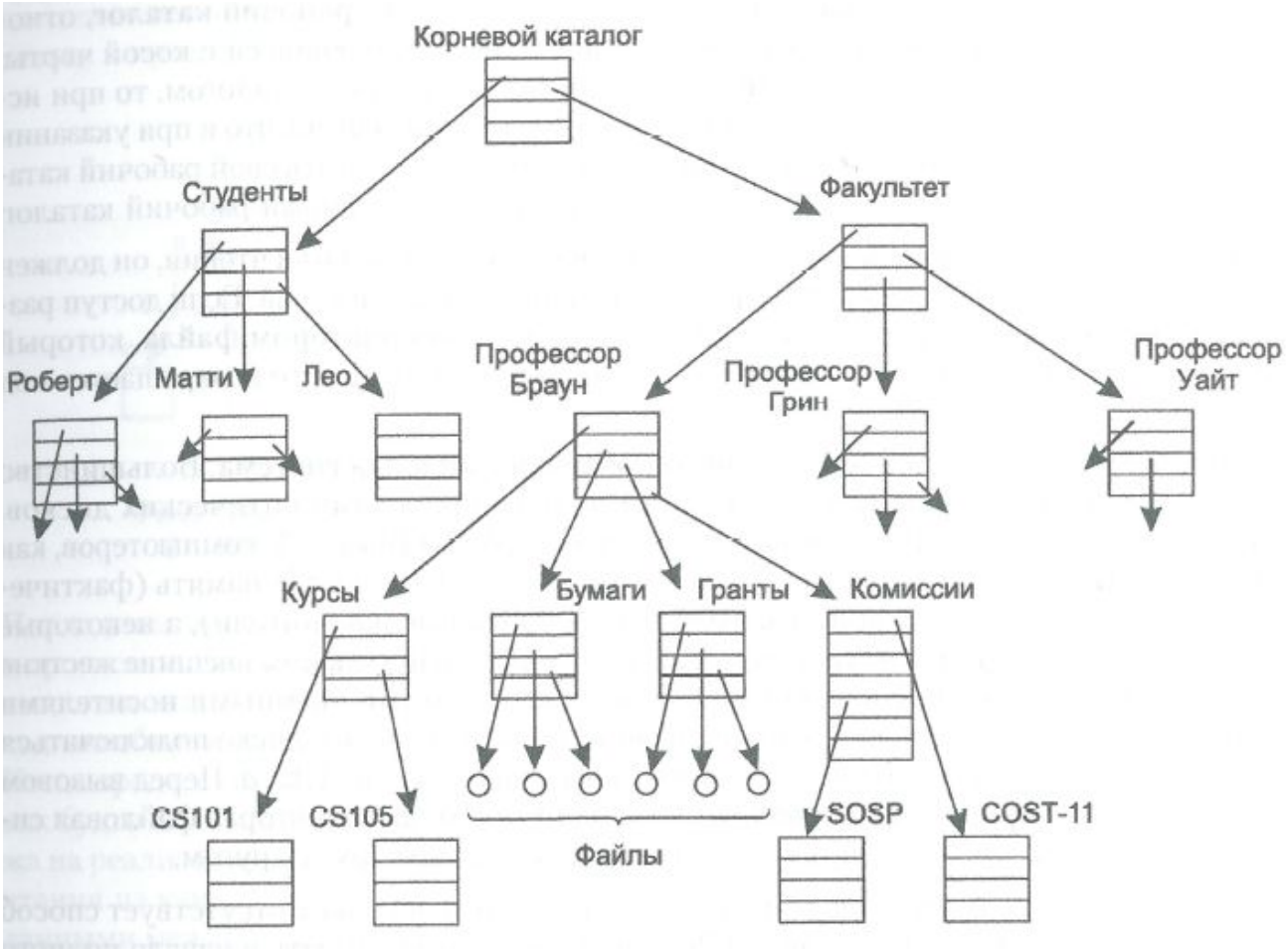


Элементами каталога могут быть либо файлы, либо другие каталоги

Иерархии файлов, как и иерархии процессов, организованы в виде деревьев.

Иерархии процессов	Иерархии файлов
глубина	
обычно не более трех уровней	имеют глубину в четыре, пять и более уровней
период существования	
не более нескольких минут	может существовать годами
Принадлежность и меры защиты	
только родительский процесс может управлять дочерним процессом или даже обращаться к нему	практически всегда существуют механизмы, позволяющие читать файлы и каталоги не только их владельцу, но и более широкой группе пользователей

- Каждый файл, принадлежащий иерархии каталогов, может быть обозначен своим **полным именем** с указанием пути к файлу, начиная с вершины иерархии — корневого каталога
- Этот абсолютный путь состоит из списка каталогов, которые нужно пройти от корневого каталога, чтобы добраться до файла, где в качестве разделителей компонентов служат символы косой черты (слеша)



У каждого процесса есть текущий **рабочий каталог,** относительно которого рассматриваются пути файлов, не начинающиеся с косой черты

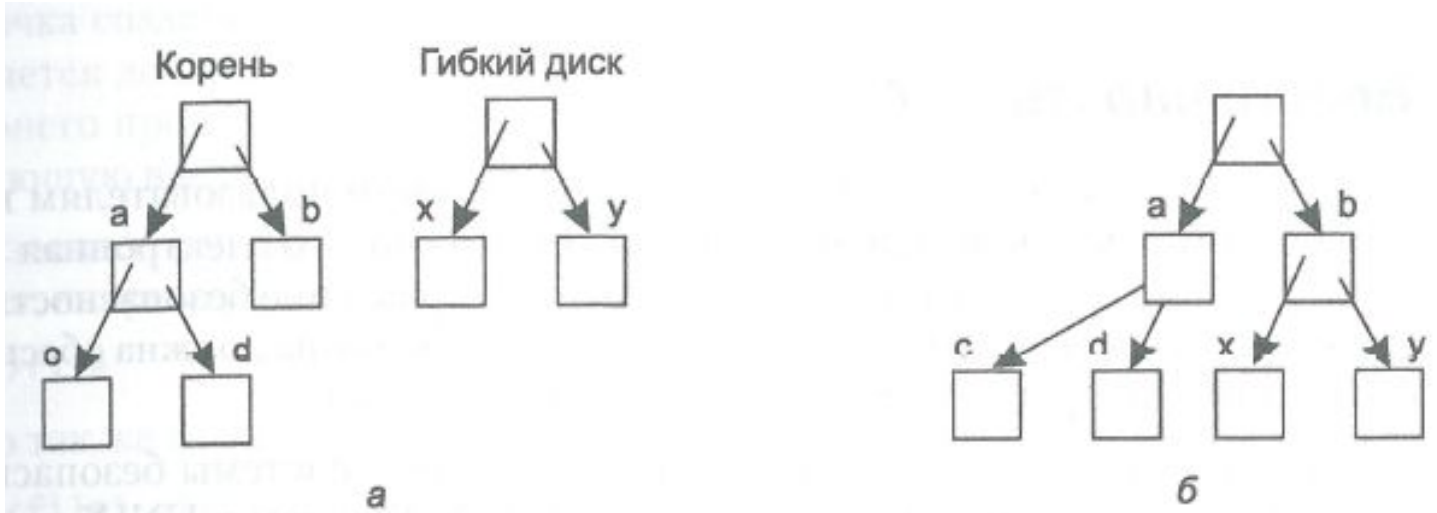
если /Faculty/Prof. Brown будет рабочим каталогом, то при ис пользовании пути Courses/CS101 будет получен тот же самый файл, что и при указании рассмотренного ранее абсолютного пути

путь к файлу CS101
/Faculty/Prof.Brown/Courses/CS101 (UNIX)
\Faculty\Prof.Brown \Courses\CS101 (Windows)

* первая косая черта является признаком использования абсолютного пути, который начинается в корневом каталоге

UNIX

Смонтированная файловая система . Для удобной работы со съемными носителями информации (диски, Blu-ray, USB) UNIX позволяет файловой системе на оптическом диске подключаться к основному дереву.



Файлы на компакт-диске: а — перед подключением недоступны; б_ после подключения становятся частью корневой файловой системы

Перед вызовом команды *mount* **корневая файловая система** на жестком диске и вторая файловая система на компакт-диске существуют отдельно и не связаны друг с другом

Системный вызов *mount* позволяет подключить файловую систему на компакт-диске к корневой файловой системе в том месте, где этого потребует программа

Unix

Специальные файлы – служат для того, чтобы устройства ввода-вывода были похожи на файлы

- можно проводить операции чтения и записи, используя те же системные вызовы, которые применяются для чтения и записи файлов

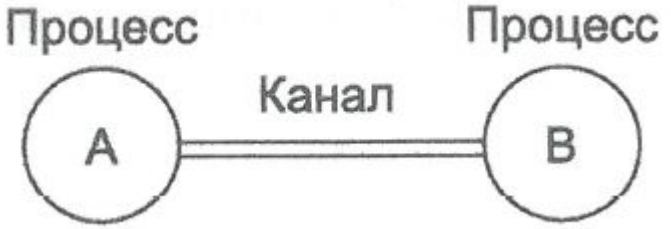
Существуют два вида специальных файлов: **блочные специальные файлы** и **символьные специальные файлы.**

- **Блочные специальные файлы** используются для моделирования устройств, содержащих набор блоков с произвольной адресацией, таких как диски. Открывая блочный специальный файл и считывая, скажем, блок 4, программа может напрямую получить доступ к четвертому блоку устройства независимо от структуры имеющейся у него файловой системы.
- **Символьные специальные файлы** используются для моделирования принтеров, модемов и других устройств, которые принимают или выдают поток символов.

В соответствии с принятым соглашением специальные файлы хранятся в каталоге **/dev**.

Например, путь /dev/lp может относиться к принтеру (который когда-то назывался строчным принтером — line printer).

Каналы



- **Каналы** имеют отношение как к процессам, так и к файлам.
- **Канал** — это разновидность псевдофайла, которым можно воспользоваться для соединения двух процессов.

Если процессам *A и B* необходимо обменяться данными с помощью канала, то они должны установить его заранее. Когда процессу *A* нужно отправить данные процессу *B,* он осуществляет запись в канал, как будто имеет дело с выходным файлом. Реализация канала очень похожа на реализацию файла.

Процесс *B* может прочитать данные, осуществляя операцию чтения из канала, как будто он имеет дело с входным файлом.

Т.о, обмен данными между процессами в UNIX очень похож на обычные операции записи и чтения файла.

Только сделав специальный системный вызов, процесс может узнать, что запись выходных данных на самом деле производится не в файл, а в канал.

Чтобы по-настоящему понять, что делает ОС, рассмотрим интерфейс.

Имеющиеся в интерфейсе системные вызовы варьируются в зависимости от используемой операционной системы (хотя основные понятия практически ничем не различаются).

Все, что будет рассматриваться, имеет непосредственное отношение к стандарту POSIX (Международный стандарт 9945-1), а следовательно к UNIX, System V, BSD, Linux, MINIX 3 и т. д.

Системные вызовы

Рассмотрим системный вызов чтения — *read.*

Read имеет три параметра:

- ✔ первый служит для задания файла,
- ✔ второй указывает на буфер,
- ✔ третий задает количество байтов, которое нужно прочитать.

Read осуществляется из программы на языке С с помощью вызова библиотечной процедуры, имя которой совпадает с именем системного вызова: *read.*

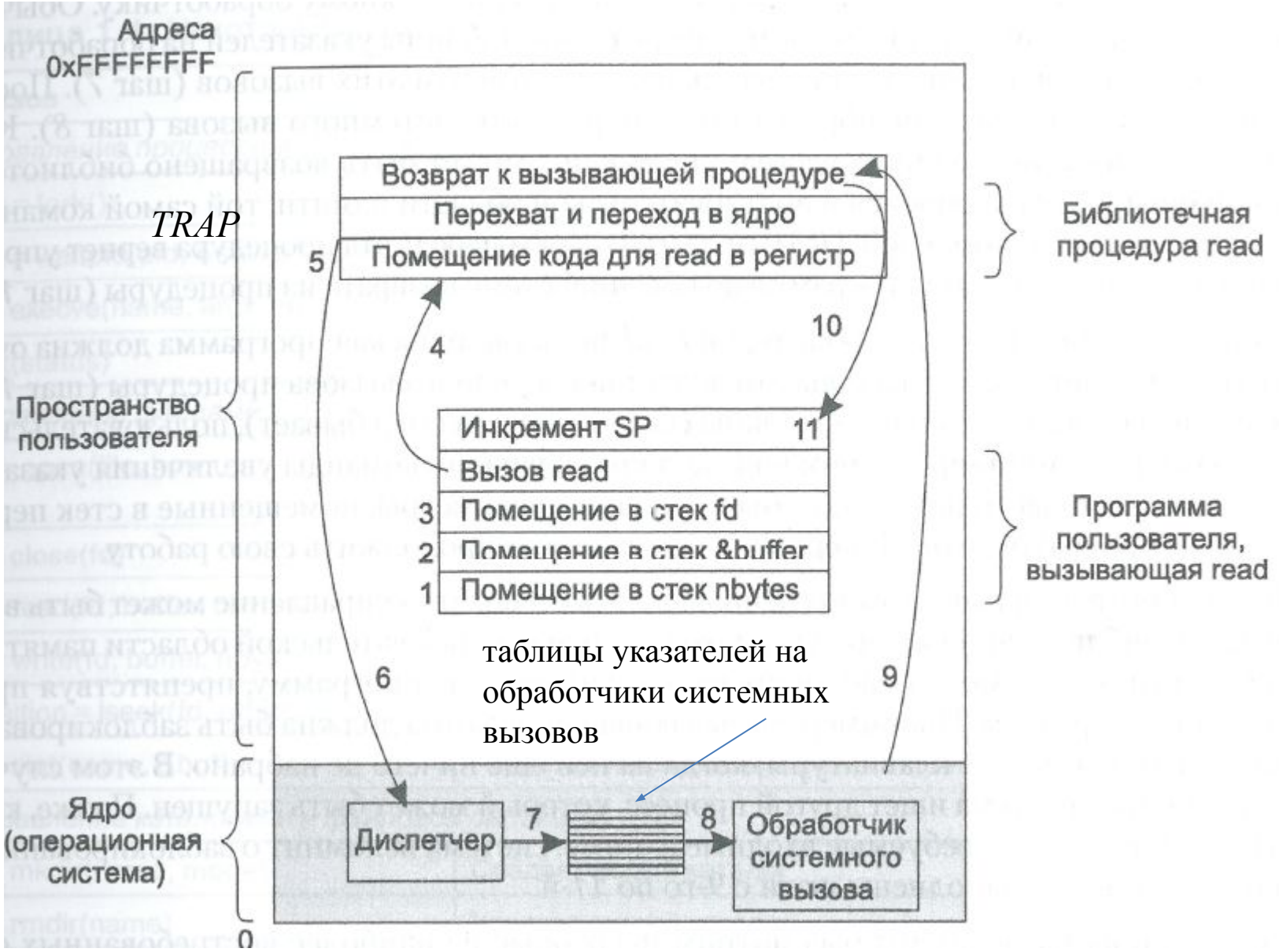
Пример вызова из программы на С :

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

- Системный вызов (и библиотечная процедура) возвращает количество фактически считанных байтов, которое сохраняется в переменной *count.* Обычно это значение совпадает со значением параметра *nbytes,* но может быть и меньше, если, например, в процессе чтения будет достигнут конец файла.
- Если системный вызов не может быть выполнен из-за неправильных параметров или ошибки диска, значение переменной *count* устанавливается в -1, а номер ошибки помешается в глобальную переменную *errno.*

count = read(fcL buffer, nbytes);

11 этапов выполнения системного вызова read(fd, buffer, nbytes)



Шаги выполнения системного вызова:

Шаги 1-3. При подготовке вызова библиотечной процедуры *read,* которая фактически и осуществляет системный вызов *read,* вызывающая программа помещает параметры в стек.

Компиляторы C и C++ помещают параметры в стек в обратном порядке, следуя исторически сложившейся традиции (чтобы на вершине стека оказался первый параметр функции *printf* — строка формата вывода данных).

Первый и третий параметры передаются по значению, а второй параметр передается по ссылке, поскольку это адрес буфера (о чем свидетельствует знак &), а не его содержимое.

Шаг 4. Осуществляется фактический вызов библиотечной процедуры. Эта команда представляет собой обычную команду вызова процедуры и используется для вызова любых процедур.

Шаг 5. Библиотечная процедура обычно помещает номер системного вызова туда, где его ожидает операционная система, например в регистр.

Шаг 6. Затем библиотечная процедура выполняет команду *TRAP для* переключения из пользовательского режима в режим ядра, и выполнение продолжается с фиксированного адреса, находящегося внутри ядра операционной системы.

Фактически команда *TRAP* очень похожа на команду вызова процедуры в том смысле, что следующая за ней команда берется из удаленного места, а адрес возврата сохраняется в стеке для последующего использования.

Шаг 7. Начавшая работу после команды *TRAP* часть ядра (диспетчер на рисунке) проверяет номер системного вызова, а затем передает управление нужному обработчику. Обычно передача управления осуществляется посредством таблицы указателей на обработчики системных вызовов, которая индексирована по номерам этих вызовов.

Шаг 8. После этого вступает в действие обработчик конкретного системного вызова.

Шаг 9. Как только обработчик закончит работу, управление может быть возвращено библиотечной процедуре, находящейся в пользовательской области памяти, той самой команде, которая следует за командой *TRAP*.

Шаг 10. В свою очередь эта процедура вернет управление пользовательской программе по обычной схеме возврата из процедуры.

Шаг 11. Чтобы завершить работу с процедурой *read,* пользовательская программа должна очистить стек, точно так же, как она это делает после любого вызова процедуры.

Примечание: на шаге *9* «управление может быть возвращено библиотечной процедуре, находящейся в пользовательской области памяти».

Системный вызов может заблокировать вызывающую программу, препятствуя продолжению ее работы.

Например, вызывающая программа должна быть заблокирована при попытке чтения с клавиатуры, когда на ней еще ничего не набрано. В этом случае операционная система ищет другой процесс, который может быть запущен. Позже, когда станут доступны требуемые входные данные, система вспомнит о заблокированном процессе и будут выполнены шаги с *9-то* по *11-й.*

Системные вызовы стандарта POSIX

В стандарте POSIX определено более 100 процедур, обеспечивающих обращение к системным вызовам.

- ✔ Системные вызовы включают в себя такие виды обслуживания, как создание и прерывание процессов, создание, удаление, чтение и запись файлов, управление каталогами, ввод и вывод данных.

Управление процессом	
pid = fork()	Создает дочерний процесс, идентичный родительскому
pid = waitpid(pid, &statloc, options)	Ожидает завершения дочернего процесса
s = execve(name, argv, environp)	Заменяет образ памяти процесса
exit(status)	Завершает выполнение процесса и возвращает статус
Управление файлами	
fd = open (file, how...)	Открывает файл для чтения, записи или для того и другого
s = close(fd)	Закрывает открытый файл
n = read(fd, buffer, nbytes)	Читает данные из файла в буфер
n = write(fd, buffer, nbytes)	Записывает данные из буфера в файл
position = lseek(fd, offset, whence)	Перемещает указатель файла
s = stat(name, &buf)	Получает информацию о состоянии файла

Управление каталогами и файловой системой	
s = mkdir(name, mode)	Создает новый каталог
s = rmdir(name)	Удаляет пустой каталог
s = link(name1, name2)	Создает новый элементе именем name2, указывающий на name1
s = unlink(name)	Удаляет элемент каталога
s = mount(special, name, flag)	Подключает файловую систему
s = umount(special)	Отключает файловую систему
Разные	
s = chdir(dirname)	Изменяет рабочий каталог
s = chmod(name, mode)	Изменяет биты защиты файла
s = kill(pid, signal)	Посылает сигнал процессу
seconds = time(&seconds)	Получает время, прошедшее с 1 января 1970 года

- В UNIX имеется практически однозначная связь между системными вызовами (например, *read)* и библиотечными процедурами (с той же *read),* используемыми для обращения к системным вызовам.
- Т.е. для каждого системного вызова обычно существует одна библиотечная процедура, чаще всего одноименная, вызываемая для обращения к нему.

Системные вызовы для управления процессами

Системный вызов *fork()* (разветвление).

- ✔ вызов *fork* является единственным существующим в POSIX способом создания нового процесса (создает точную копию исходного процесса, включая все дескрипторы файлов, регистры и т. п.)
- ✔ после выполнения вызова *fork* исходный процесс и его копия (родительский и дочерний процессы) выполняются независимо друг от друга.
- ✔ на момент разветвления все их соответствующие переменные имеют одинаковые значения, но поскольку родительские данные копируются в дочерний процесс, последующие изменения в одном из них не влияют на изменения в другом.

- ✔ системный вызов *fork* возвращает нулевое значение для дочернего процесса и равное идентификатору дочернего процесса или PID — для родительского. Используя возвращенное значение PID, два процесса могут определить, какой из них родительский, а какой — дочерний.

- ✔ в большинстве случаев после вызова fork дочернему процессу необходимо выполнить программный код, отличный от родительского.

Пример работы системной оболочки:

- ✔ Системная оболочка считывает команду с терминала, создает дочерний процесс, ожидает, пока дочерний процесс выполнит команду, а затем считывает другую команду, если дочерний процесс завершается.
- ✔ Для ожидания завершения дочернего процесса роди тельский процесс выполняет системный вызов *waitpid,* который просто ждет, пока дочерний процесс не закончит свою работу (причем здесь имеется в виду любой дочерний процесс, если их несколько).

pid = waitpid(pid, &statloc, options)

- ✔ Системный вызов *waitpid* может ожидать завершения конкретного дочернего процесса или любого из запущенных дочерних процессов, если первый параметр имеет значение -1.
- ✔ Когда работа *waitpid* завершается, по адресу, указанному во втором параметре — *statloc,* заносится информация о статусе завершения дочернего процесса (нормальное или аварийное завершение и выходное значение).
- ✔ В третьем параметре определяются различные необязательные настройки.
- ✔ После набора команды оболочка создает дочерний процесс, который должен выполнить команду пользователя. Он делает это, используя системный вызов *execve,* который полностью заменяет образ памяти процесса файлом, указанным в первом параметре.

pid = waitpid(pid, &statloc, options)

s = execve(name, argv, environp)

Windows Win32 API

Операционные системы Windows и UNIX фундаментально отличаются друг от друга в соответствующих моделях программирования:

- ✔ Программы UNIX состоят из кода, который выполняет те или иные действия, при необходимости обращаясь к системе с системными вызовами для получения конкретных услуг.
- ✔ Программой Windows управляют, как правило, события. Основная программа ждет, пока возникнет какое-нибудь событие, а затем вызывает процедуру для его обработки.
  - Типичные события — это нажатие клавиши, перемещение мыши, нажатие кнопки мыши или подключение USB-диска. Затем для обслуживания события, обновления экрана и обновления внутреннего состояния программы вызываются обработчики. В итоге все это приводит к несколько иному стилю программирования, чем в UNIX.

Системные вызовы Windows

- ✔ Фактические системные вызовы и используемые для их выполнения библиотечные вызовы намеренно разделены.
- ✔ Корпорацией Microsoft определен набор процедур, названный **Win32 API** (Application Programming Interface — интерфейс прикладного программирования), который используется для доступа к службам ОС.
- ✔ Отделяя API-интерфейс от фактических системных вызовов, Microsoft поддерживает возможность со временем изменять существующие системные вызовы (даже от одной версии к другой), сохраняя работоспособность уже существующих программ => в самых последних версиях Windows содержится множество новых, ранее недоступных системных вызовов.

- Под Win32 будем понимать интерфейс, поддерживаемый всеми версиями Windows. Win32 обеспечивает совме стимость версий Windows.
- Количество имеющихся в Win32 API вызовов велико — исчисляется тысячами. Более того, наряду с тем, что многие из них действительно запускают системные вызовы, существенная часть целиком выполняется в пространстве пользователя. => при работе с Windows становится невозможно понять, что является системным вызовом (то есть выполняемым ядром), а что — просто вызовом библиотечной процедуры в пространстве пользователя. Фактически то, что было системным вызовом в одной версии Windows, может быть выполнено в пространстве пользователя в другой, и наоборот.

UNIX	Win32	Описание
fork	CreateProcess	Создает новый процесс
waitpid	WaitForSingleObject	Ожидает завершения процесса
execve	Нет	CreateProcess=fork+execve
exit	ExitProcess	Завершает выполнение процесса
open	Create File	Создает файл или открывает существующий файл
close	CloseHandle	Закрывает файл
read	Read File	Читает данные из файла
write	WriteFile	Записывает данные в файл
Iseek	SetFilePointer	Перемещает указатель файла
stat	GetFileAttributesEx	Получает различные атрибуты файла
mkdir	CreateDirectory	Создает новый каталог

UNIX	Win32	Описание
rmdir	RemoveDirectory	Удаляет пустой каталог
link	Нет	Win32 не поддерживает связи
unlink	DeleteFile	Удаляет существующий файл
mount	Нет	Win32 не поддерживает подключение к файловой системе
umount	Нет	Win32 не поддерживает подключение к файловой системе
chdir	SetCurrentDirectory	Изменяет рабочий каталог
chmod	Нет	Win32 не поддерживает защиту файла (хотя NT поддерживает)
kill	Нет	Win32 не поддерживает сигналы
time	GetLocalTime	Получает текущее время

- В Win32 API имеется огромное число вызовов для управления окнами, геометрическими фигурами, текстом, шрифтами, полосами прокрутки, диалоговыми окнами, меню и другими составляющими графического пользовательского интерфейса.
- Если графическая подсистема работает в памяти ядра (что справедливо для некоторых, но не для всех версий Windows), то их можно отнести к системным вызовам, в противном случае они являются просто библиотечными вызовами.

Структура операционной системы

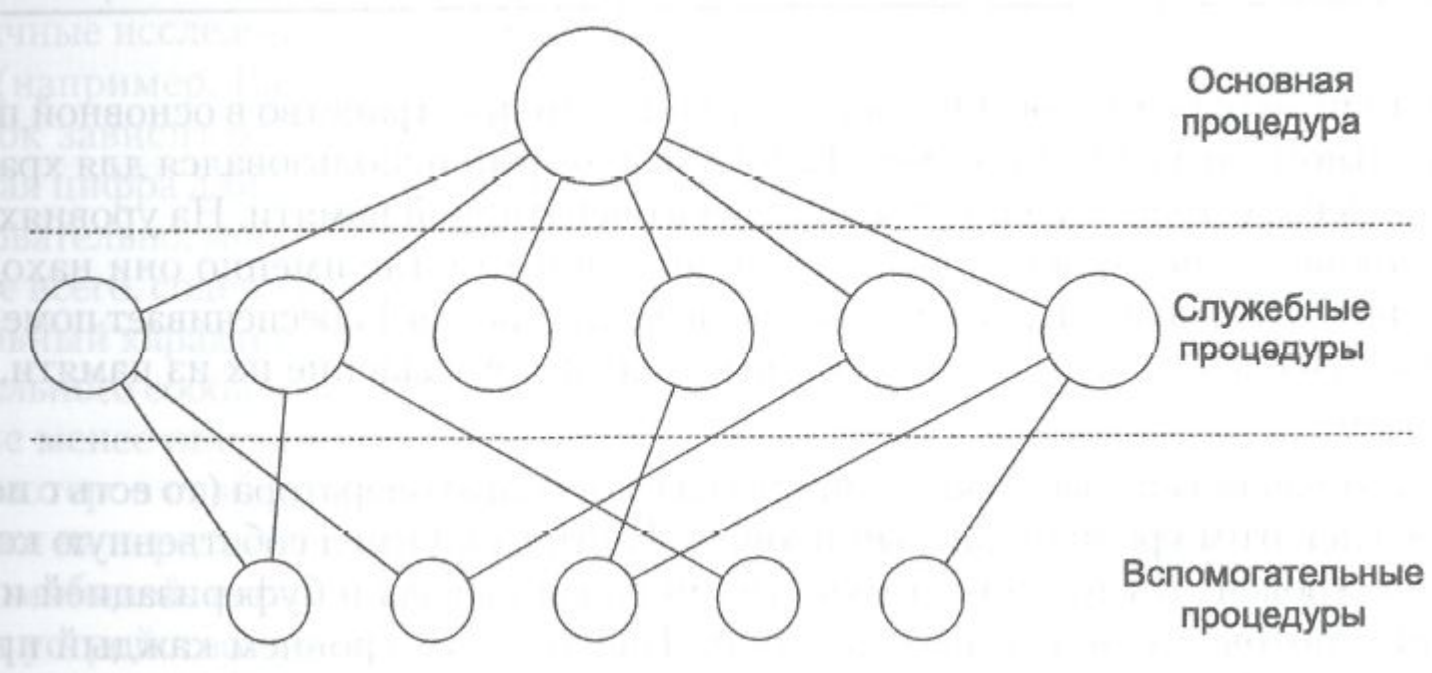
Выделяют шесть основных структур:

- ✔ монолитные системамы;
- ✔ многоуровневые системамы;
- ✔ микроядра;
- ✔ клиент-серверные системы;
- ✔ виртуальные машины;
- ✔ экзоядра.

Монолитные системы

- Вся ОС работает как единая программа в режиме ядра.
- ОС написана в виде набора процедур, связанных вместе в одну большую исполняемую программу.
- каждая процедура может свободно вызвать любую другую процедуру, если та выполняет какое-нибудь полезное действие, в котором нуждается первая процедура.
- возможность вызвать любую нужную процедуру приводит к весьма высокой эффективности работы системы, но наличие нескольких тысяч процедур, которые могут вызывать друг друга сколь угодно часто, нередко делает ее громоздкой и непонятной. Кроме того, отказ в любой из этих процедур приведет к аварии всей операционной системы.

Простая структурированная модель монолитной системы



Базовая структура монолитной ОС:

1. Основная программа, которая вызывает требуемую служебную процедуру.
2. Набор служебных процедур, выполняющих системные вызовы.
3. Набор вспомогательных процедур, содействующих работе служебных процедур.

- В этой модели для каждого системного вызова имеется одна ответственная за него служебная процедура, которая его и выполняет.
- Вспомогательные процедуры выполняют действия, необходимые нескольким служебным процедурам, в частности извлечение данных из пользовательских программ.
- Таким образом, процедуры делятся на три уровня.

В дополнение к основной операционной системе, загружаемой во время запуска компьютера, многие операционные системы поддерживают загружаемые расширения, в числе которых драйверы устройств ввода-вывода и файловые системы. Эти компоненты загружаются по мере надобности.

- В UNIX они называются библиотеками общего пользования.
- В Windows они называются **DLL**-библиотеками (Dynamic-LinkLibraries — динамически подключаемые библиотеки). Они находятся в файлах с расширениями имен .dll, и в каталоге C:\Windows\system32 на системе Windows их более 1000.

Многоуровневые системы

- Обобщением монолитной ОС, является организация операционной системы в виде иерархии уровней, каждый из которых является надстройкой над нижележащим уровнем.
- Первой системой, построенной таким образом, была система THE, созданная в Technische Hogeschool Eindhoven в Голландии Э. Дейкстрой (E. W Dijkstra) и его студентами в 1968 году.

Структура ОС системы THE

5	Оператор
4	Программы пользователя
3	Управление вводом-выводом
2	Связь оператора с процессом
1	Управление основной памятью и магнитным барабаном
0	Распределение ресурсов процессора и обеспечение многозадачного режима

Уровень 1 управлял памятью. На уровнях выше первого процессы не должны были беспокоиться о том, где именно они находятся, в памяти или на барабане.

Уровень 2 управлял связью каждого процесса с консолью оператора (то есть с пользо вателем). Над этим уровнем каждый процесс фактически имел собственную консоль оператора.

- **Уровень 3** управлял устройствами ввода-вывода и буферизацией информационных потоков в обоих направлениях. Над третьим уровнем каждый процесс мог работать с абстрактными устройствами ввода-вывода, имеющими определенные свойства.
- На **уровне 4** работали пользовательские программы, которым не надо было заботиться о процессах, памяти, консоли или управлении вводом-выводом.
- Процесс системного оператора размещался на **уровне 5**.

Система MULTICS: вместо уровней использовались серии концентрических колец, где внутренние кольца обладали более высокими привилегиями по отношению к внешним.

- Когда процедуре из внешнего кольца требовалось вызвать процедуру внутреннего кольца, ей нужно было создать эквивалент системного вызова, то есть выполнить инструкцию *TRAP,* параметры которой тщательно проверялись на допустимость перед тем, как разрешить продолжение вызова.
- Хотя вся операционная система в MULTICS являлась частью адресного пространства каждого пользовательского процесса, аппаратура позволяла определять отдельные процедуры как защищенные от чтения, записи или выполнения.

Преимущества кольцеобразного механизма: данный механизм мог быть легко расширен и на структуру пользовательских подсистем.

- Например, профессор может написать программу для тестирования и оценки студенческих программ и запустить ее в кольце *n,* а студенческие программы будут выполняться в кольце *n* + 1, так что студенты не смогут изменить свои оценки.

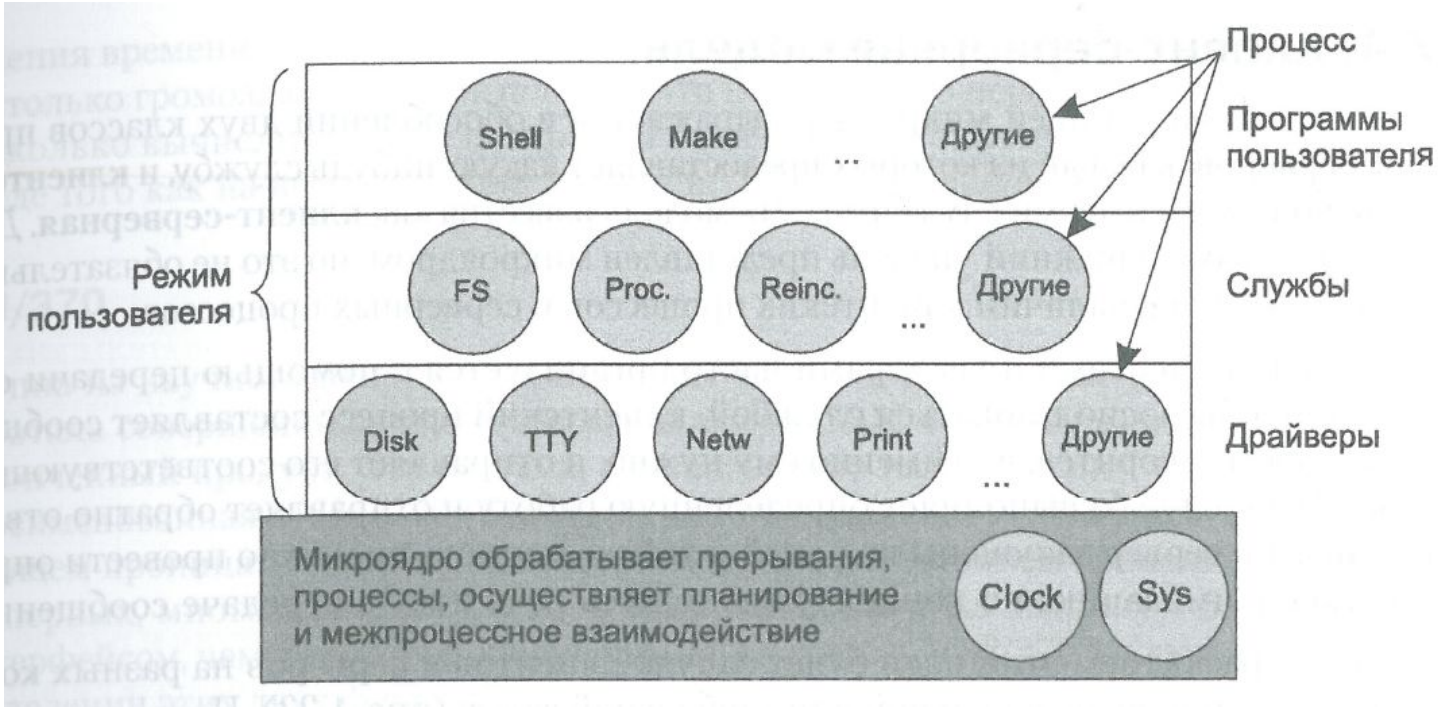
Микроядра

- При использовании многоуровневого подхода разработчикам необходимо выбрать, где провести границу между режимами ядра и пользователя.
- Традиционно все уровни входили в ядро, но это было не обязательно.
- Существуют очень весомые аргументы в пользу того, чтобы в режиме ядра выполнялось как можно меньше процессов, поскольку ошибки в ядре могут вызвать немедленный сбой системы.

Для сравнения: пользовательские процессы могут быть настроены на обладание меньшими полномочиями, чтобы их ошибки не носили фатального характера.

Цель: достижение высокой надежности за счет разбиения операционной системы на небольшие, вполне определенные модули.

- Только один модуль – микроядро – запускается в режиме ядра, а все остальные запускаются в виде относительно слабо наделенных полномочиями обычных пользовательских процессов.
- Если запустить каждый драйвер устройства и файловую систему как отдельные пользовательские процессы, то ошибка в одном из них может вызвать отказ соответствующего компонента, но не сможет вызвать сбой всей системы. Т.о., ошибка в драйвере звукового устройства приведет к искажению или пропаданию звука, но не вызовет зависания компьютера. В отличие от этого в монолитной системе, где все драйверы находятся в ядре, некорректный драйвер звукового устройства может запросто сослаться на неверный адрес памяти и привести систему к немедленной вынужденной остановке



Упрощенная структура системы MINIX 3

За пределами ядра структура системы представляет собой три уровня процессов, которые работают в режиме пользователя.
Самый нижний уровень содержит драйверы устройств.
Над драйверами расположен уровень, содержащий службы, которые осуществляют основной объем работы операционной системы.
Все они работают в режиме пользователя.

MINIX 3 — это POSIX-совместимая система с открытым исходным кодом, находящаяся в свободном доступе по адресу www.minix3.org

Контроллер

- Контроллер представляет собой микросхему или набор микросхем, которые управляют устройством на физическом уровне. Он принимает от операционной системы команды, например считать данные с помощью устройства, а затем их выполняет.
- Программа, предназначенная для общения с контроллером, выдачи ему команды и получения поступающих от него ответов, называется **драйвером устройства.**
- Каждый производитель контроллеров должен поставлять вместе с ними драйверы для каждой поддерживаемой операционной системы.

Подключение драйвера к ОС

Драйвер нужно поместить в операционную систему, предоставив ему тем самым возможность работать в режиме ядра.

Три способа установки драйвера в ядро:

- заново скомпоновать ядро вместе с новым драйвером и затем перезагрузить систему;
- создать в специальном файле операционной системы запись, сообщающую ей о том, что требуется, и затем перезагрузить систему. Во время загрузки операционная система сама находит нужные ей драйверы и загружает их (Windows);
- динамическая загрузка драйверов — операционная система может принимать новые драйверы в процессе работы и оперативно устанавливать их, не требуя для этого перезагрузки.

В каждом контроллере для связи с ним имеется небольшое количество регистров.

Пример:

Простейший контроллер диска может иметь регистры для указания адреса на диске, адреса в памяти, счетчика секторов и направления передачи информации (чтение или запись).

Чтобы активизировать контроллер, драйвер получает команду от операционной системы, затем переводит ее в соответствующие значения для записи в регистры устройства.

Из совокупности всех регистров устройств формируется пространство портов ввода-вывода.

- На некоторых компьютерах регистры устройств отображаются в *адресное пространство операционной системы* (на те адреса, которые она может использовать), поэтому состояния регистров можно считывать и записывать точно так же, как и обычные слова в оперативной памяти.

- На других компьютерах регистры устройств помещаются в специальное *пространство портов ввода-вывода* (I/O port space), в котором каждый регистр имеет адрес порта. На таких машинах в режиме ядра доступны специальные команды ввода-вывода (обычно обозначаемые *IN и OUT),* позволяющие драйверам читать и записывать данные в регистры.

Ввод и вывод данных

Способ 1:

- Пользовательская программа производит системный вызов, который транслируется ядром в процедуру вызова соответствующего драйвера.
- После этого драйвер приступает к процессу ввода-вывода. В это время он выполняет очень короткий цикл, постоянно опрашивая устройство и отслеживая завершение операции (обычно занятость устройства определяется состоянием специального бита).
- По завершении операции ввода-вывода драйвер помещает данные (если таковые имеются) туда, куда требуется, и возвращает управление.
- Затем операционная система возвращает управление вызывающей программе. Этот способ называется **активным ожиданием** или **ожиданием готовности,** а его недостаток заключается в том, что он загружает процессор опросом устройства об окончании работы.

Ввод и вывод данных

Способ 2:

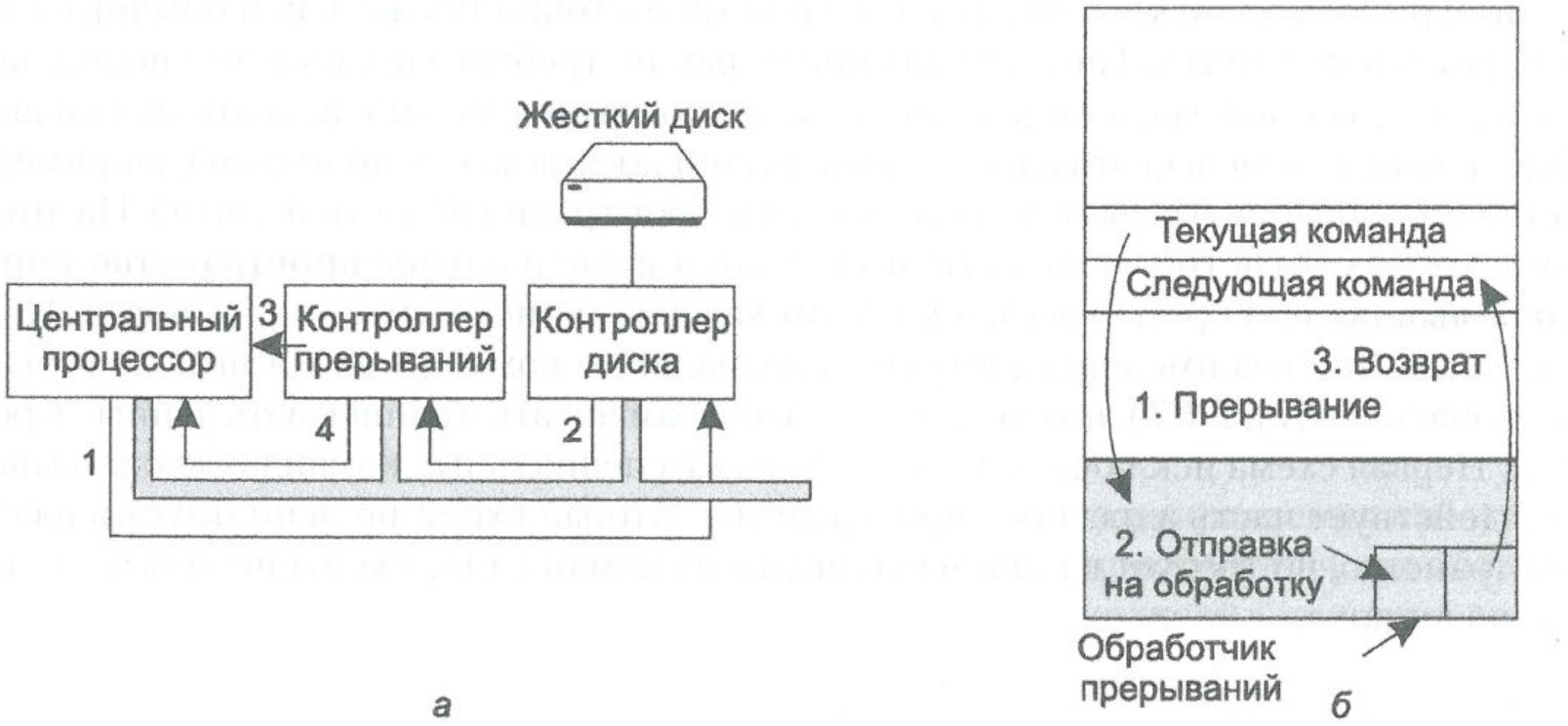
Драйвер запускает устройство и просит его выдать прерывание по окончании выполнения команды (завершении ввода или вывода данных).

Сразу после этого драйвер возвращает управление.

Затем операционная система блокирует вызывающую программу, если это необходимо, и переходит к выполнению других задач.

Когда контроллер обнаруживает окончание передачи данных, он генерирует **прерывание,** чтобы просигнализировать о завершении операции.

Процесс ввода-вывода

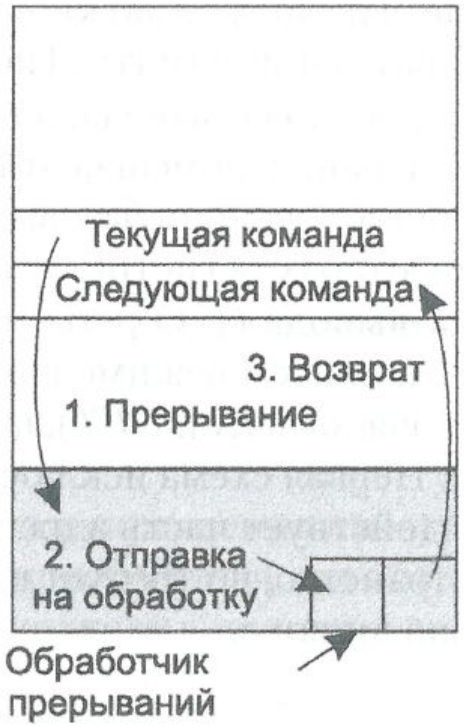


Этапы: а — запуск устройства ввода-вывода и получения прерывания; *б* —обработки прерывания (включает в себя получение прерывания, выполнение программы обработки прерывания и возвращение управления программе пользователя)

Процесс ввода-вывода

Этап 1. Драйвер передает команду контроллеру, записывая информацию в его регистры. Затем контроллер запускает само устройство.
Этап 2. Когда контроллер завершает чтение или запись заданного ему количества байтов, он выставляет сигнал для микросхемы контроллера прерываний, используя для этого определенные линии шины.

- **Этап 3.** Если контроллер прерываний готов принять прерывание (а он может быть и не готов к этому, если обрабатывает прерывание с более высоким уровнем приоритета), он выставляет сигнал на контакте микросхемы центрального процессора, информируя его о завершении операции.
- **Этап 4.** Контроллер прерываний выставляет номер устройства на шину, чтобы процессор мог его считать и узнать, какое устройство только что завершило работу (поскольку одновременно могут работать сразу несколько устройств).



- Как только центральный процессор решит принять прерывание, содержимое счетчика команд и слова состояния программы помещаются, как правило, в текущий стек и процессор переключается в режим ядра.
- Номер устройства может быть использован как индекс части памяти, используемой для поиска адреса обработчика прерываний данного устройства. Эта часть памяти называется **вектором прерываний.**
- Когда обработчик прерываний (являющийся частью драйвера устройства, выдающего запрос на прерывание) начинает свою работу, он извлекает помещенные в стек содержимое счетчика команд и слова состояния программы и сохраняет их, а затем опрашивает устройство для определения его состояния.
- После завершения обработки прерывания обработчик возвращает управление ранее работавшей пользовательской программе — на первую же еще не выполненную команду.

Способ 3:

Используется специальный контроллер **прямого доступа к памяти** (Direct Memory Access **(DMA)),** который может управлять потоком битов между оперативной памятью и некоторыми контроллерами без постоянного вмешательства центрального процессора.

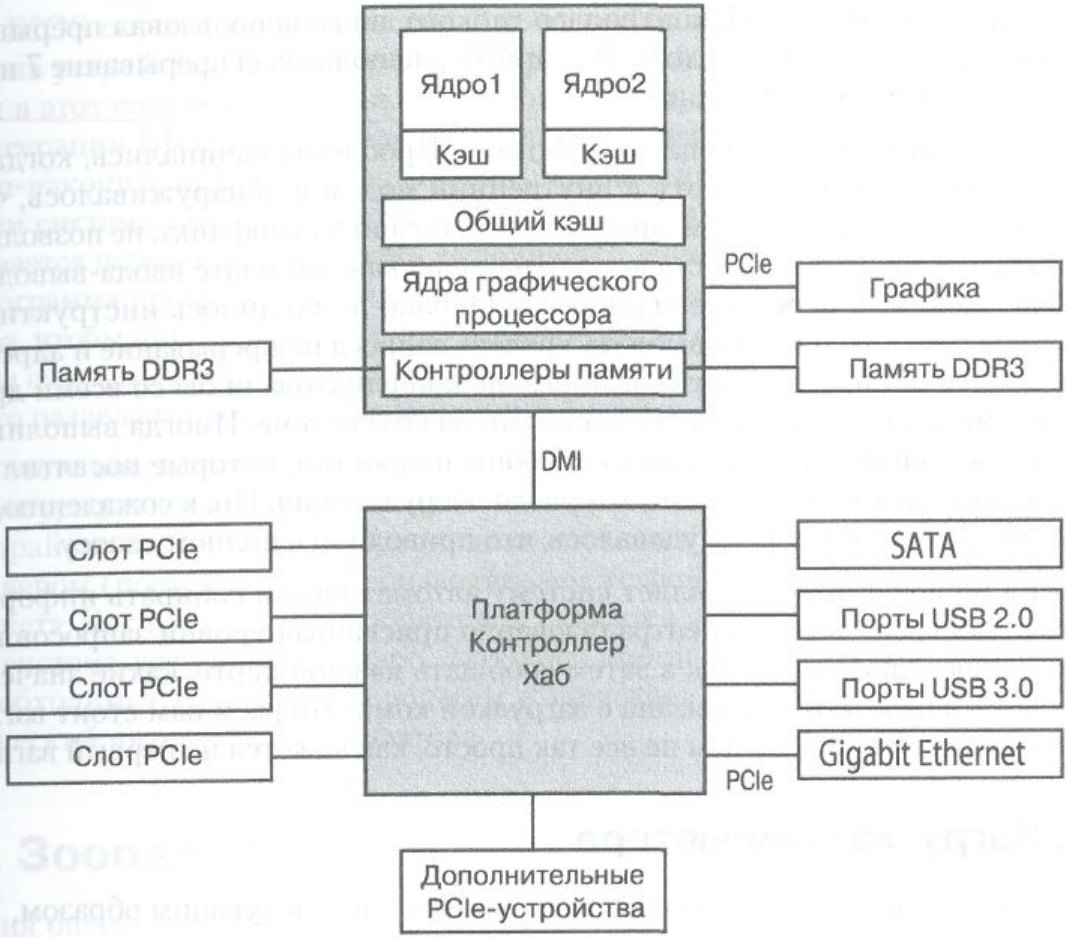
Центральный процессор осуществляет настройку контроллера DMA, сообщая ему, сколько байтов следует передать, какое устройство и адреса памяти задействовать и в каком направлении передать данные, а затем дает ему возможность действовать самостоятельно.

Когда контроллер DMA завершает работу, он выдает запрос на прерывание, который обрабатывается в ранее рассмотренном порядке.

Прерывания

Прерывания часто происходят в очень неподходящие моменты, например во время работы обработчика другого прерывания. Поэтому центральный процессор обладает возможностью запрещать прерывания с последующим их разрешением. Пока прерывания запрещены, любые устройства, закончившие свою работу, продолжают выставлять свои запросы на прерывание, но работа процессора не прекращается, пока прерывания снова не станут разрешены. Если за время запрещения прерываний завершится работа сразу нескольких устройств, контроллер решает, какое из них должно быть обработано первым, полагаясь обычно на статические приоритеты, назначенные каждому устройству. Побеждает устройство, имеющее наивысший приоритет, которое и обслуживается в первую очередь. Все остальные устройства должны ожидать своей очереди.

Шины



Структура большой системы семейства x86

- У этой системы имеется множество шин (например, шина кэш-памяти, шина памяти, а также шины PCIe, PCI, USB, SATA и DMI), каждая из которых имеет свою скорость передачи данных и свое предназначение.
- Операционная система для осуществления функций настройки и управления должна знать обо всех этих шинах.
- Основной шиной является PCI (Peripheral Component Interconnect — интерфейс периферийных устройств).

- Центральный процессор общается с памятью через быструю шину DDR3, со внешним графическим устройством — через шину PCIe, а со всеми остальными устройствами — через концентратор по шине **DMI** (Direct Media Interface — интерфейс непосредственной передачи данных)
- Концентратор в свою очередь соединяет все другие устройства, используя для обмена данными с USB-устройствами универсальную последовательную шину, для обмена данными с жесткими дисками и DVD-приводами — шину SATA и для передачи Ethernet-кадров — шину PCIe.
- Шина **USB** (Universal Serial Bus — универсальная последовательная шина) была разработана для подключения к компьютеру всех низкоскоростных устройств ввода-вывода вроде клавиатуры и мыши.
- **SCSI** (Small Computer System Interface — интерфейс малых вычислительных систем) является высокоскоростной шиной, предназначенной для высокопроизводительных дисков, сканеров и других устройств, нуждающихся в значительной пропускной способности. В наши дни эти шины встречаются в основном в серверах и рабочих станциях.

Plug and play

- ОС должна знать о том, какие периферийные устройства подключены к компьютеру, и сконфигурировать эти устройства. Это требование заставило корпорации Intel и Microsoft разработать для PC-совместимых компьютеров систему, называемую plug and play (подключи и работай). Она основана на аналогичной концепции, первоначально реализованной в Apple Macintosh. До появления plug and play каждая плата ввода-вывода имела фиксированный уровень запроса на прерывание и постоянные адреса для своих регистров ввода-вывода. Технология plug and play заставляет систему автоматически собирать информацию об устройствах ввода-вывода, централизованно присваивая уровни запросов на прерывания и адреса ввода-вывода, а затем сообщать каждой карте, какие значения ей присвоены.