НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Курс «Современные операционные системы»

Лекция 4

Основные понятия операционных систем

Москва
2016

Содержание

3.4. Клиент-серверные системы
3.5. Виртуальные машины
3.6. Экзоядра

1. Понятия операционных систем

1.1. Типы ОС

- ОС мейнфреймов – пакетная обработка, обработка транзакций, разделение времени (OS/390, UNIX, Linux).
- Серверные ОС (Solaris, FreeBSD, Linux и Windows Server 201x).
- Многопроцессорные ОС – специальные функции связи, сопряжения и синхронизации.
- ОС ПК – многозадачный режим (Linux, FreeBSD, Windows 7, Windows 8, Windows 10, OS X).
- ОС КПК (Android, iOS).
- Встроенные ОС – невозможна установка пользовательских программ, все программы записаны в ПЗУ (Embedded Linux, QNX, VxWorks).
- ОС сенсорных узлов (TinyOS).
- ОС реального времени – строгий контроль времени, жесткое и мягкое разделение времени.
- ОС смарт-карт – очень жесткие ограничения по мощности процессора и объему памяти.

1.2. Процессы

Процесс – программа во время ее выполнения.

Процесс содержит всю информацию, необходимую для работы программы. Он имеет свое ***адресное пространство*** в котором размещаются выполняемая программа, данные этой программы и ее стек. С процессом связан набор ресурсов: регистры (счетчик команд и указатель стека), список открытых файлов, необработанные предупреждения, список связанных процессов и др.

Процесс может приостанавливаться и возобновляться именно с того состояния, в котором был остановлен. На период приостановки вся информация о процессе (за исключением содержимого его собственного адресного пространства) хранится в ***таблице процессов*** – массиве (или связанном списке) структур.

Возможно создание ***дочерних процессов*** и *межпроцессное взаимодействие*.

1.3. Адресное пространство

Адресное пространство (образ памяти) – непрерывный набор адресов (с нуля и до некоторого максимума), создаваемый ОС, на которые может ссылаться процесс.

Адресное пространство отделено от физической памяти и может быть как больше, так и меньше нее. Если адресное пространство процесса превышает объем оперативной памяти, используется технология ***виртуальной памяти:*** ОС хранит часть адресного пространства в оперативной памяти, а часть – на диске, по необходимости меняя их фрагменты местами.

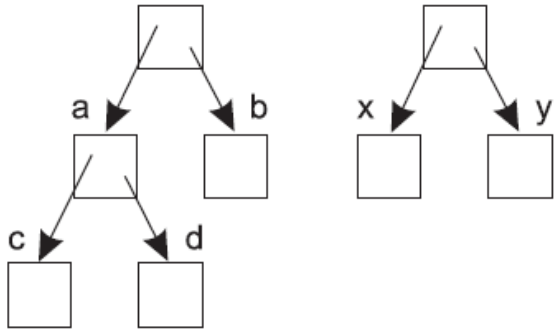
1.4. Файлы

В любой момент времени у каждого процесса есть текущий рабочий каталог, относительно которого рассматриваются пути файлов.

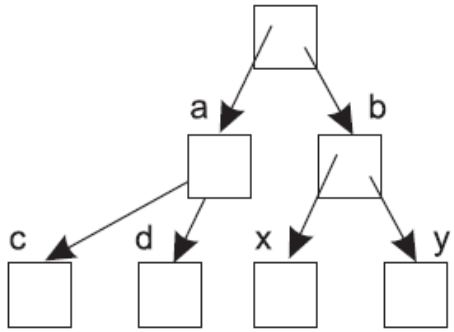
Перед тем как с файлом можно будет работать в режиме записи или чтения, он должен быть открыт. При этом происходит также проверка прав доступа. Если доступ разрешен, система возвращает целое число, называемое ***дескриптором файла***, который используется в последующих операциях. Если доступ запрещен, то возвращается код ошибки.

Файловые системы в UNIX.

Смонтированная файловая система, ***корневая*** файловая система.



Перед вызовом команды `mount`



После выполнения команды `mount`

Специальные файлы (хранятся в каталоге `/dev`): ***блочные*** (диск) и ***символьные*** (принтер, модем).

Канал — псевдофайл для соединения двух процессов.

Файлам в UNIX присваивается 9-битный код защиты. Этот код состоит из трех трехбитных полей по правам доступа пользователей:

- владелец,
- представитель группы, в которую он входит,
- всех остальные.

В каждом поле – по три ***rwx-бита*** (read, write, execute) для чтения, записи и выполнения. Например, код защиты

r	w	x	r	-	x	-	-	x
владелец			**группа**			**остальные**		

означает, что владельцу доступны чтение, запись или выполнение файла, остальным представителям его группы разрешается чтение или выполнение файла (но не запись), а всем остальным разрешено выполнение файла (но не чтение или запись). Для каталога x-бит означает разрешение на поиск. Дефис (минус) означает, что соответствующее разрешение отсутствует.

2. Системные вызовы

2.1. Передача управления ОС

Однопроцессорный компьютер одномоментно может выполнить только одну команду. Когда процесс выполняет программу в режиме пользователя и нуждается в какой-нибудь услуге ОС:

- процесс выполняет команду ***системного прерывания***, чтобы передать управление ОС;
- ОС по параметрам вызова определяет, что именно требуется вызывающему процессу;
- ОС обрабатывает системный вызов;
- ОС возвращает управление той команде, которая следует за системным вызовом.

Системные вызовы похожи на вызовы процедур, но входят в ядро.

2.2. Системные вызовы POSIX

Пример: системный вызов чтения.

Вызов из программы на языке С с помощью вызова библиотечной процедуры `read`:

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

имеет три параметра:

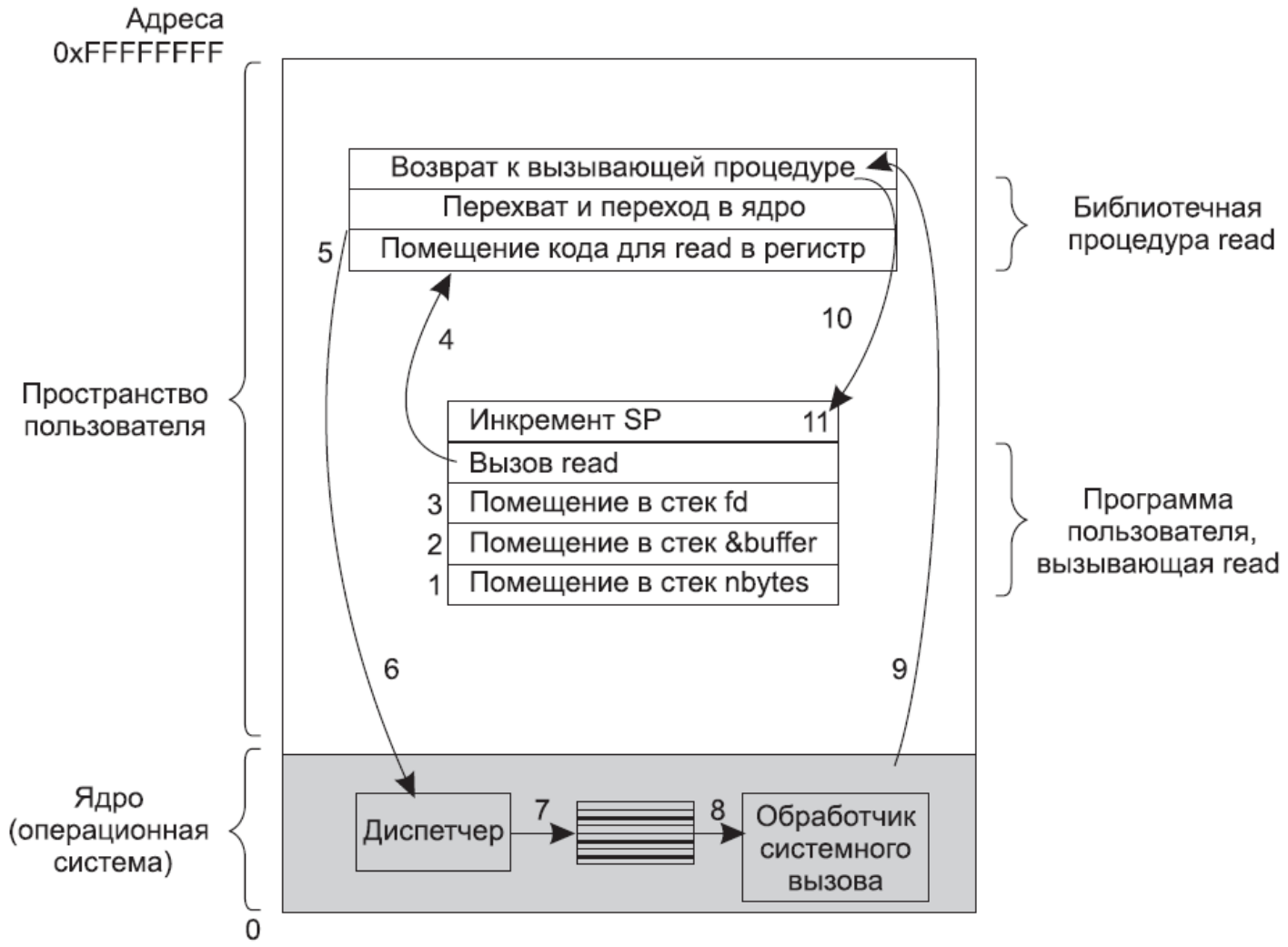
`fd` – служит для задания файла,

`buffer` – указывает на буфер,

`nbytes` – задает количество байтов, которое нужно прочитать.

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 помещается в глобальную переменную `errno`. Программам обязательно нужно проверять результаты системного вызова, чтобы отслеживать возникновение ошибки.



1,2,3 – пользовательская программа помещает параметры в стек;
4 – пользовательская программа вызывает библиотечную процедуру `read`;
5 – процедура `read` помещает номер системного вызова в регистр и выполняет команду TRAP;

6 – выполнение продолжается с фиксированного адреса, находящегося внутри ядра ОС;

7 – ядро проверяет номер системного вызова находит его в таблице указателей на обработчики;

8 – ядро передает управление нужному обработчику;

9 – после окончания работы обработчика ядро может

а) ввернуть управление процедуре `read` в пользовательской области памяти (после чего выполняется команда, которая следует за командой TRAP);

б) заблокировать вызывающую программу, пока не станут доступны считываемые данные;

10 – процедура `read` (после своего окончания) возвращает управление пользовательской программе;

11 – завершая процедуру `read`, пользовательская программа очищает стек и увеличивает указатель стека SP. Теперь программа может продолжить свою работу.

Некоторые из наиболее востребованных системных вызовов (библиотечных процедур, осуществляющих системные вызовы) стандарта POSIX:

Вызов	Описание
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 элемент с 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

При возникновении ошибки возвращаемое значение кода завершения `s` равно `-1`. Имена возвращаемых значений: `pid` — id процесса, `fd` — дескриптор файла, `n` — количество байтов, `position` — смещение внутри файла, `seconds` — прошедшее время.

2.2.1. Системные вызовы для управления процессами.

Вызов	Описание
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. Аналогичные системные вызовы **`execl`**, **`execv`**, **`execle`** позволяют указать параметры различными способами. Все эти системные вызовы, обычно называют одним термином **`exec`**.
`exit(status)`	Завершает выполнение процесса и возвращает статус

При возникновении ошибки возвращаемое значение кода завершения `s` равно `−1`. Имена возвращаемых значений: `pid` — id процесса, `fd` — дескриптор файла.

Системный вызов **`fork`** (разветвление) является единственным в POSIX способом создания нового процесса. Он 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. Вызов `fork` возвращает нулевое значение для дочернего процесса и равное идентификатору дочернего процесса `pid` — для родительского.

Системный вызов **`waitpid`** ожидает, пока дочерний процесс с номером `pid` закончит свою работу. Когда запущено несколько дочерних процессов, вызов `waitpid` может ожидать завершения любого из них, если первый параметр `pid` 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 `options` определяются различные необязательные настройки.

Пример: работа системной оболочки, усеченной до минимума.

```
#define TRUE 1
while (TRUE) { /* бесконечный цикл */
     type_prompt( ); /* вывод приглашения на экран */
     read_command(command, parameters); /* чтение ввода с терминала */
     if (fork( ) != 0) { /* ответвление дочернего процесса */
          /* ... код родительского процесса ... */
          waitpid(-1, &status, 0); /* ожидание завершения дочернего процесса */
     } else {
          /* ... код дочернего процесса ... */
          execve(command, parameters, 0); /* выполнение command */
     }
}
```

После набора пользователем команды `command` оболочка с помощью системного вызова `fork` создает дочерний процесс, который должен выполнить эту команду пользователя. Дочерний процесс делает это, используя системный вызов **execve**, который полностью заменяет образ памяти процесса файлом, указанным в первом параметре.

Допустим пользователь в оболочке набрал команду для копирования файла:

```
cp file1 file2
```

Созданный оболочкой дочерний процесс находит и выполняет файл `cp` и передает ему имена исходного и целевого файлов. Основная программа `cp` (и большинство других основных программ на языке С) содержит объявление `main(argc, argv, envp)`, где

`argc` — количество элементов командной строки, включая имя программы
(в примере: `argc` равен `3`);

`argv` — указатель на массив строковых элементов командной строки
(в примере: `argv[0]` указывает на строку `cp`,
`argv[1]` — на строку `file1`,
`argv[2]` — на строку `file2`).

`envp` — указатель на массив переменных окружения, то есть на массив строк вида `имя = значение`, используемый для передачи программе такой информации, как тип терминала и имя домашнего каталога программ.

В рассматриваемом примере окружение дочернему процессу не передается, поэтому третий параметр `execve` имеет нулевое значение.

С точки зрения семантики системный вызов `exec` наиболее сложный из всех имеющихся в POSIX системных вызовов. Все остальные выглядят намного проще.

Системный вызов **`exit`** используется процессами, когда они заканчивают выполнение. Единственный параметр `status` — статус выхода (0–255) возвращается родительской программе через `statloc` в системном вызове `waitpid`.

2.2.2. Системные вызовы для управления файлами.

Вызов	Описание
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

При возникновении ошибки возвращаемое значение кода завершения `s` равно `-1`. Имена возвращаемых значений: `fd` — дескриптор файла, `n` — количество байтов, `position` — смещение внутри файла.

Системный вызов **open** используется для открытия файла в одном из режимов:
`O_RDONLY` – только для чтения,
`O_WRONLY` – только для записи,
`O_RDWR` – для чтения и записи,
`O_CREAT` – создание нового файла.

Системный вызов **lseek** изменяет значение указателя на текущую позицию, связанного с данным файлом. Параметры вызова: `fd` — дескриптор файла, `offset` — позиция в файле, `whence` — относительно чего задана позиция (начала файла, текущей позиции или конца файла). Вызов `lseek` возвращает абсолютную позицию в файле (в байтах) после изменения указателя.

Системный вызов **stat** сообщает информацию о файле. Параметры: `name` – файл, `buf` – указатель на структуру, в которую эта информация должна быть помещена. Для открытого файла то же самое делает системный вызов **fstat**.

2.2.3. Системные вызовы для управления каталогами и файловой системой.

Вызов	Описание
Управление каталогами и файловой системой	
`s = mkdir(name, mode)`	Создает новый каталог
`s = rmdir(name)`	Удаляет пустой каталог
`s = link(name1, name2)`	Создает новый элемент с 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

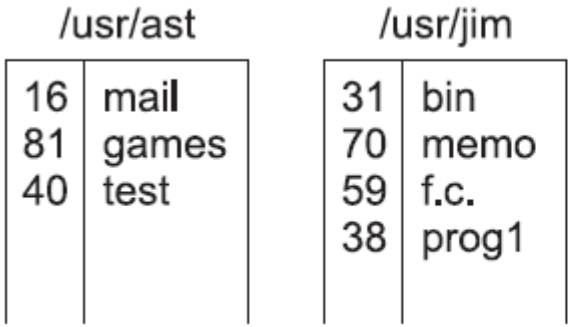
При возникновении ошибки возвращаемое значение кода завершения `s` равно `-1`.

Системный вызов **link** позволяет одному и тому же файлу появляться под двумя или более именами, зачастую в разных каталогах.

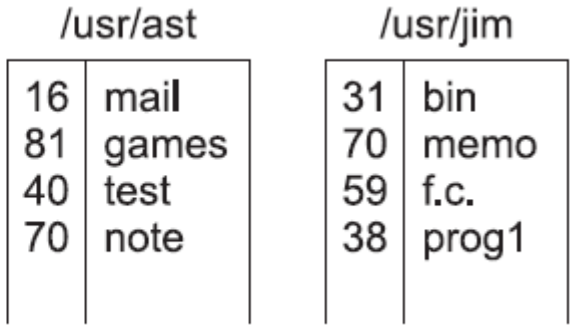
Например, у двух пользователей с именами `ast` и `jim` имеются каталоги с файлами. Если ast выполнит программу, содержащую системный вызов

```
link("/usr/jim/memo", "/usr/ast/note");
```

то файл `memo` в каталоге `jim` теперь будет входить в каталог `ast` под именем `note`. После этого `/usr/jim/memo` и `/usr/ast/note` будут ссылаться на один и тот же файл.



перед созданием ссылки



после создания ссылки

Каждый файл в UNIX имеет свой уникальный идентификатор – **i-номер** (i-number). Он является единственным для каждого файла индексом в таблице **i-узлов** (i-nodes). Каждый i-узел хранит информацию о том, кто является владельцем файла, где расположены его блоки на диске и т. д. Каталог — это просто файл, содержащий набор пар (i-номер, ASCII-имя).

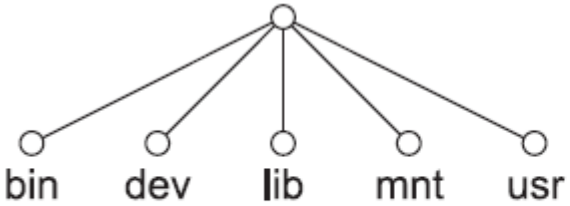
Системный вызов `link` просто создает новый элемент каталога (возможно, с новым именем), используя i-номер существующего файла.

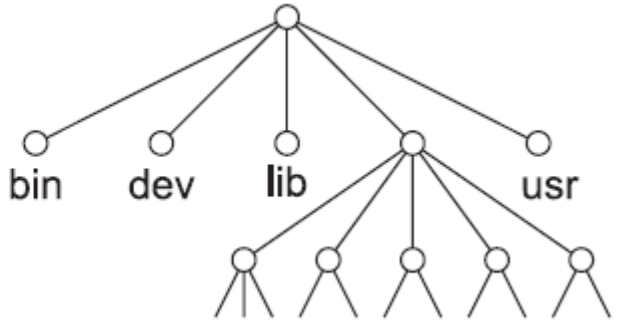
Если с помощью системного вызова **`unlink`** одна из этих записей будет удалена, то вторая останется нетронутой. Если будут удалены обе записи, то UNIX увидит, что записей для файла не существует (поле в i-узле отслеживает количество указывающих на данный файл элементов в каталогах), и удалит файл с диска.

Системный вызов **`mount`** позволяет объединять две файловые системы в одну. Пример подключения («монтирования») файловой системы USB-диска к корневой файловой системе:

```
mount("/dev/sdb0", "/mnt", 0);
```

где `/dev/sdb0` — это имя блочного специального файла для USB-устройства `0`,
`/mnt` — место в дереве, куда будет происходить подключение,
`0` (третий параметр) — режим подключения файловой системы (чтения и записи или только чтения).





Файловая система может быть отключена с помощью системного вызова **`umount`**.

2.2.4. Разные системные вызовы.

Вызов	Описание
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

Системный вызов **chdir** изменяет текущий рабочий каталог. После вызова

```
chdir("/usr/ast/test");
```

при открытии файла `xyz` будет открыт файл `/usr/ast/test/xyz`.

Системный вызов **chmod** позволяет изменять rwx-биты прав доступа к файлу.

Например, чтобы сделать файл для всех доступным для чтения, а для владельца — доступным для чтения и для записи, необходимо выполнить следующий системный вызов:

```
chmod("file", 0644);
```

Системный вызов **kill** позволяет пользователям и пользовательским процессам посылать сигналы. Если процесс готов принять сигнал, то при его поступлении запускается обработчик сигнала. Если процесс не готов к обработке сигнала, то его поступление уничтожает процесс.

2.3. Системные вызовы Win32 API

Операционные системы Windows и UNIX существенно отличаются в 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. В отличие от этого 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 событий***.

В ОС Windows фактические системные вызовы и используемые для их выполнения библиотечные вызовы намеренно разделены. Microsoft определен набор процедур, названный Win32 API (Application Programming Interface — интерфейс прикладного программирования), который должен использоваться для доступа к службам операционной системы. Отделяя API-интерфейс от фактических системных вызовов, Microsoft имеет возможность со временем изменять существующие системные вызовы, сохраняя работоспособность уже существующих программ. Microsoft гарантирует, что с течением времени процедуры Win32 не будут меняться.

При работе с Windows невозможно понять, что является системным вызовом (то есть выполняемым ядром), а что — просто вызовом библиотечной процедуры в пространстве пользователя. То, что было системным вызовом в одной версии Windows, может быть выполнено в пространстве пользователя в другой, и наоборот.

Win32 не является полностью единообразным и последовательным интерфейсом.

Вызовы Win32 API, приблизительно соответствующие вызовам UNIX (в Windows имеется очень большое количество других системных вызовов, большинство из которых не имеют соответствий в UNIX):

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

3. Структура операционных систем

Шесть основных конструкторских решений внутренней структуры ОС:

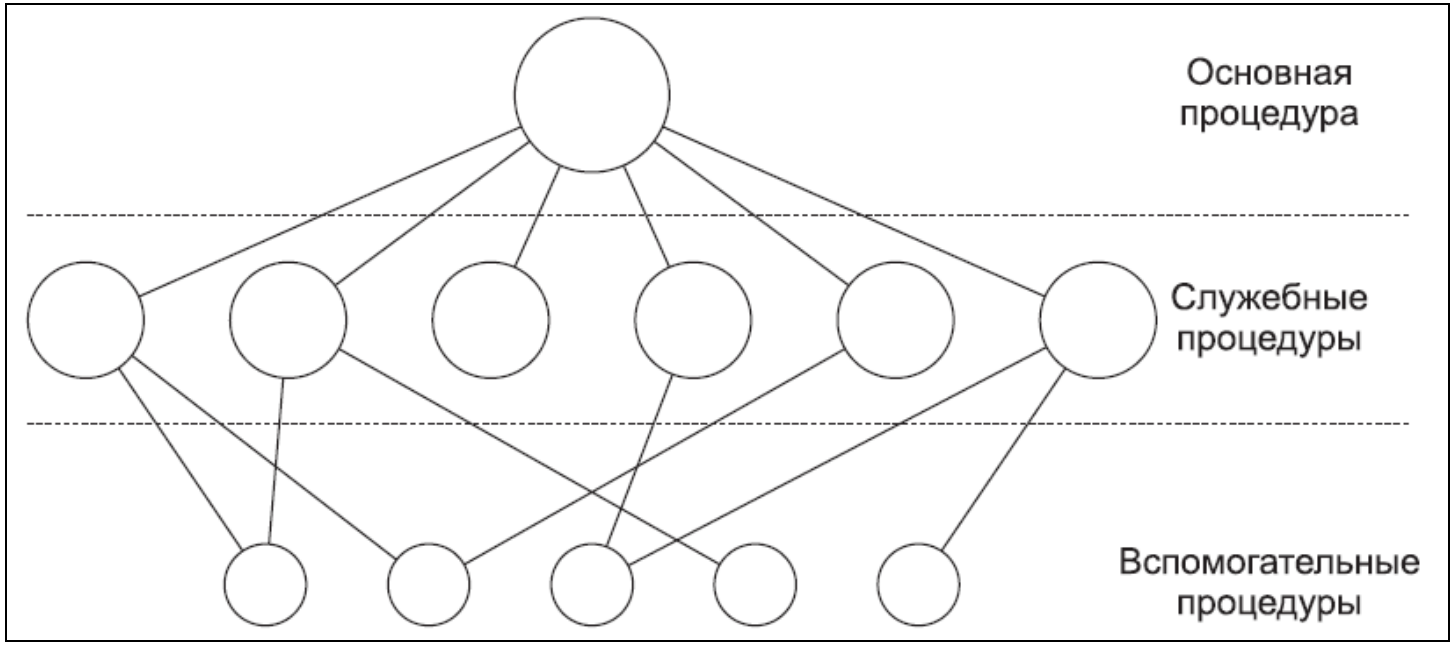
- ***монолитные системы,***
- ***многоуровневые системы,***
- ***микроядра,***
- ***клиент-серверные системы,***
- ***виртуальные машины,***
- ***экзоядра.***

3.1. Монолитные системы

Операционная система написана в виде набора процедур, связанных вместе в одну большую исполняемую программу, которая работает в режиме ядра. Каждая процедура может свободно вызвать любую другую процедуру, что приводит к весьма высокой эффективности работы системы, но делает ее громоздкой и непонятной. Отказ любой из этих процедур приведет к аварии всей операционной системы.

Для построения исполняемого файла монолитной системы необходимо сначала скомпилировать все отдельные процедуры (или файлы, содержащие процедуры), а затем связать их вместе, воспользовавшись системным компоновщиком. Здесь, по существу, отсутствует сокрытие деталей реализации — каждая процедура видна любой другой процедуре.

Службы (системные вызовы), предоставляемые ОС, запрашиваются путем помещения параметров в стек, а затем выполняется инструкция TRAP, которая переключает машину из пользовательского режима в режим ядра и передает управление ОС. Затем операционная система извлекает параметры и определяет, какой системный вызов должен быть выполнен и находит его обработчик (также одну из процедур) в таблице.

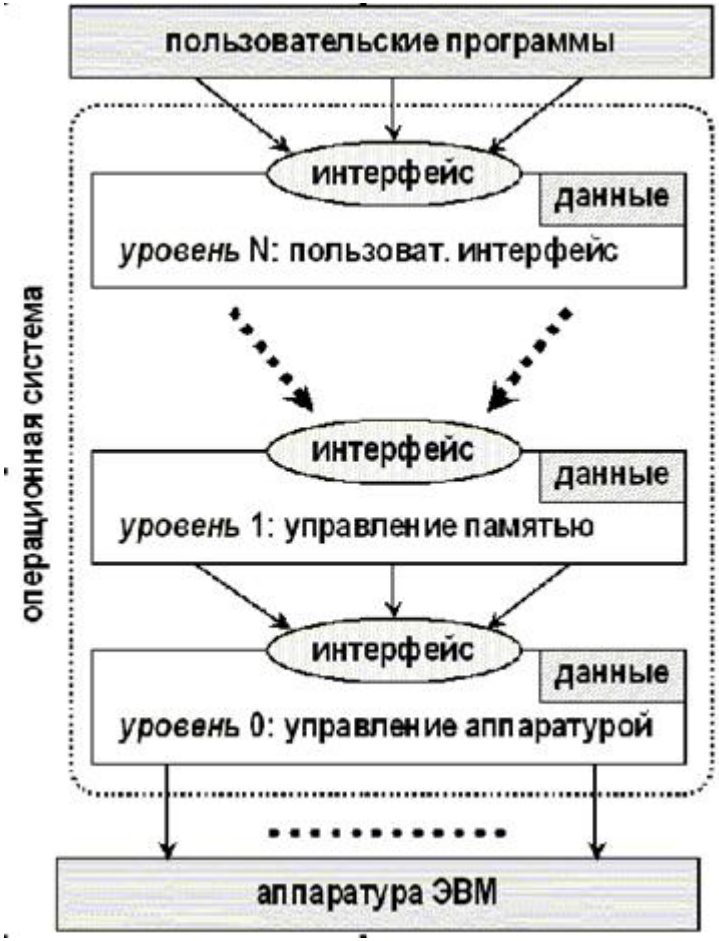


Структурированная модель монолитной системы

Многие ОС поддерживают расширения, которые загружаются по мере надобности (например, драйверы устройств ввода-вывода, файловые системы). В UNIX они называются ***библиотеками общего пользования***. В Windows они называются ***DLL-библиотеками*** (Dynamic-Link Libraries — динамически подключаемые библиотеки) и находятся в файлах с расширениями имен `.dll` в каталоге `C:\Windows\system32` (их более 1000).

3.2. Многоуровневые системы

Многоуровневая ОС организована в виде иерархии уровней, каждый из которых является надстройкой над нижележащим уровнем. Каждый уровень может обращаться за услугами только к соседнему нижележащему уровню через его интерфейс. Внутренние структуры данных каждого уровня не доступны другим уровням, а реализации процедур уровня скрыты.



3.3. Микроядра

Поскольку ошибки в ядре могут вызвать немедленный сбой всей системы, необходимо как можно меньше процессов выполнять режиме ядра.

Плотность ошибок в коде программ для солидных промышленных систем — приблизительно 10 ошибок на 1000 строк кода. Тогда монолитная ОС, состоящая из 5 000 000 строк кода, скорее всего, содержит от 10 000 до 50 000 ошибок ядра. Не все они имеют фатальный характер, некоторые ошибки могут представлять собой просто выдачу неправильного сообщения об ошибке в той ситуации, которая складывается крайне редко. Тем не менее операционные системы содержат столько ошибок, что производители компьютеров снабдили свою продукцию кнопкой перезапуска.

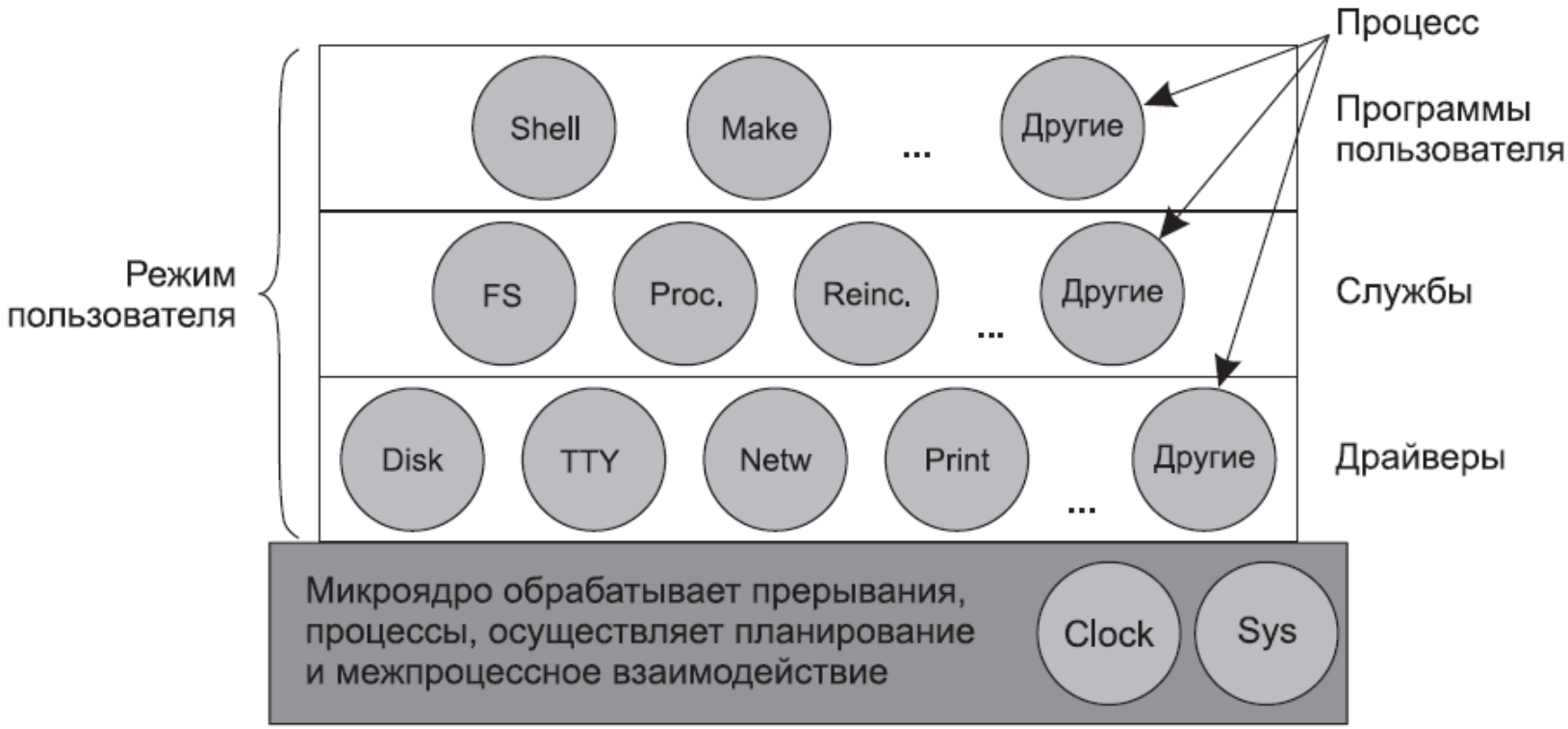
В конструкции микроядра достигается высокая надежность за счет разбиения операционной системы на небольшие модули. Только один из них — микроядро — запускается в режиме ядра, а все остальные запускаются в виде относительно слабо наделенных полномочиями обычных пользовательских процессов.

За исключением OS X, которая основана на микроядре Mach, широко распространенные ОС настольных ПК микроядра не используют. Но микроядра доминируют в приложениях, работающих в реальном масштабе времени в промышленных устройствах, авионике и военной технике, которые выполняют особо важные задачи и должны отвечать очень высоким требованиям надежности.

Часть общеизвестных микроядер представляют ОС Integrity, K42, L4, PikeOS, QNX, Symbian и MINIX 3.

Микроядро ОС **MINIX 3** — POSIX-совместимой системы с открытым исходным кодом, находящаяся в свободном доступе (`www.minix3.org`) — состоит из около 12 000 строк кода на языке С и 1400 строк кода на ассемблере, который использован для самых низкоуровневых функций.

Служба **reincarnation server** выполняет проверку функционирования других служб и драйверов. В случае обнаружения отказа одного из компонентов, он автоматически заменяется без вмешательства со стороны пользователя.



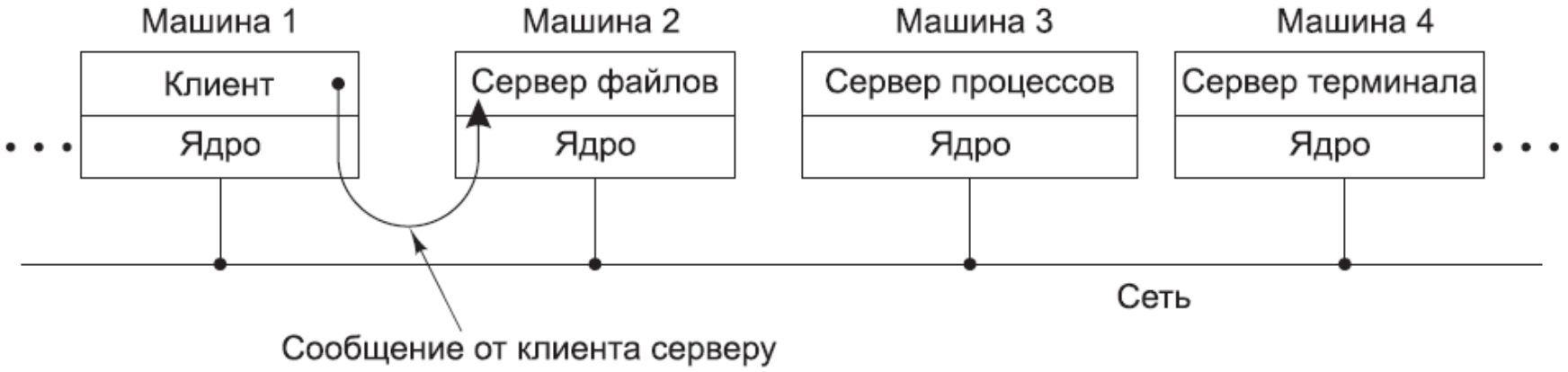
Структура ОС MINIX 3

3.4. Клиент-серверные системы

В клиент-серверной модели обособлены два класса процессов: ***серверы***, каждый из которых предоставляет какую-нибудь службу, и ***клиенты***, которые пользуются этими службами.

Связь между клиентами и серверами организуется с помощью передачи **сообщений**. Чтобы воспользоваться службой, клиентский процесс составляет сообщение, в котором говорится, что именно ему нужно, и отправляет его соответствующей службе. Затем служба выполняет определенную работу и отправляет обратно ответ.

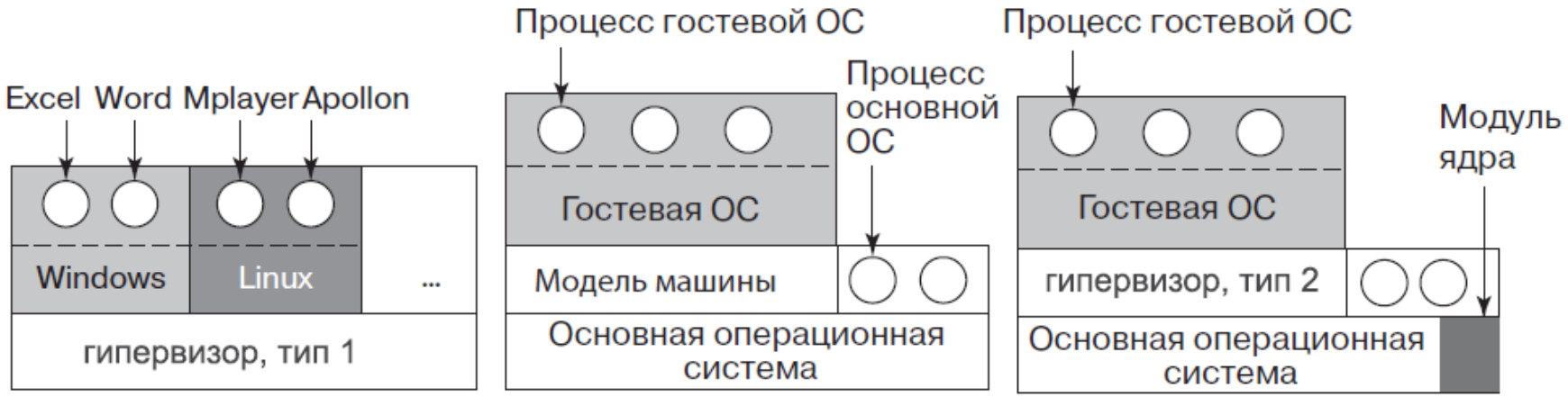
Клиент-серверная модель является абстракцией, которая может быть реализована как для отдельно взятого компьютера, так и для компьютеров, объединенных в сеть.



Клиент-серверная модель, реализованная с помощью сети

3.5. Виртуальные машины

Программа или аппаратная схема – ***монитор виртуальных машин (гапервизор)*** запускается непосредственно на оборудовании и обеспечивает многозадачность, предоставляя верхнему уровню несколько ***виртуальных машин***. Эти виртуальные машины являются точной копией исходной аппаратуры, включающей режим ядра и пользователя, устройства ввода-вывода, прерывания и все остальное, что есть у настоящей машины. Поскольку каждая виртуальная машина идентична настоящему оборудованию, на каждой из них способна работать любая ОС, которая может быть запущена непосредственно на самом оборудовании. На разных виртуальных машинах могут быть запущены разные ОС.



Гипервизор, тип 1 | Чистый гипервизор, тип 2 | Практический гипервизор, тип 2

Автономный гипервизор (тип 1) имеет свои встроенные драйверы устройств, модели драйверов и планировщик и поэтому не зависит от базовой ОС. Он более производителен, но проигрывает в производительности виртуализации на уровне ОС и паравиртуализации. Примеры: *VMware ESX, Citrix **Xen**Server.*

Гипервизор на основе базовой ОС (тип 2) работает вместе с ядром основной ОС. Гостевой код может выполняться прямо на физическом процессоре, но доступ к устройствам ввода-вывода компьютера из гостевой ОС осуществляется через обычный процесс основной ОС — ***монитор уровня пользователя***. Примеры: *Microsoft Virtual PC, VMware Workstation, QEMU, Parallels, VirtualBox.*

Добавление модуля ядра (практический гипервизор, тип 2) для выполнения ряда трудоемких задач улучшило производительность.

Гибридный гипервизор (тип 1+) состоит из двух частей: из тонкого гипервизора, контролирующего процессор и память, а также работающей под его управлением специальной сервисной ОС, через которую гостевые ОС получают доступ к физическому оборудованию. Примеры: *Microsoft Virtual Server, Sun Logical Domains, Citrix **Xen**Server, Microsoft Hyper-V*.

При ***паравиртуализации*** гостевые ОС подготавливаются для исполнения в виртуализированной среде, для чего их ядро незначительно модифицируется. Программа гипервизора предоставляет ОС гостевой API, вместо использования ресурсов напрямую. Код виртуализации локализуется непосредственно в гостевую ОС (для этого гостевые ОС должны иметь открытые исходные коды).

3.6. Экзоядра

Другой подход заключается в **разделении ресурсов** настоящей машины, вместо ее клонирования. При этом каждой виртуальной машине предоставляется подмножество ресурсов. Например, одна виртуальная машина может получить дисковые блоки от 0 до 1023, другая — блоки от 1024 до 2047 и т. д.

Самый нижний уровень, работающий в режиме ядра, — это программа ***экзоядро***. Ее задача состоит в распределении ресурсов между виртуальными машинами и отслеживании, чтобы ни одна из машин не пыталась использовать чужие ресурсы. Каждая виртуальная машина может запускать собственную ОС, но ограничена использованием ресурсов.