Shashlik Inc.

Операционные системы

Лекции Истратова А.Ю.

Сергей Мигалин & Денис Якимов
25.3.2018

Оглавление

1 Понятие операционной системы

1.1 Определение операционной системы

Операционная система – набор программ и данных, которые обеспечивают возможность использования аппаратуры компьютера и представляют интерфейс для реализации программного обеспечения, а также обработки информации.

Операционная система **взаимодействует** с аппаратными средствами, пользователем, прикладными программами, системными программами

1.2 Уровни работы операционной системы

№ уровня	Название	Тип
0	Физические устройства	Оборудование
1	Микроархитектура	
2	Система команд	
3	Ядро ОС	Операционная система
4	Прикладное *системное* ПО	
5	Прикладное ПО	Прикладное ПО

1.3 Функции операционной системы

Основными функциями операционных систем являются:

- разделение аппаратных ресурсов между программами
- определение интерфейса пользователя
- обеспечивание эффективного выполнения операций ввода\вывода
- осуществление восстановления информации и вычислительного процесса в случае ошибок
- планирование доступа к общим ресурсам
- управление процессами и потоками

1.4 Ресурсы операционной системы

Ресурсами, которыми управляет операционная система, являются:

- память
- процессорное время
- устройства ввода\вывода
- программы

Ресурсы делятся на выгружаемые и невыгружаемые, исчерпаемые и неисчерпаемые.

1.4.1 Выгружаемые и невыгружаемые ресурсы

Выгружаемый ресурс можно безболезненно забирать у владеющего им процесса.

Примеры: оперативная память, процессорное время.

Невыгружаемый нельзя отобрать у процесса.

Примеры: устройства ввода\вывода, диск.

1.4.2 Исчерпаемые и неисчерпаемые ресурсы

Исчерпаемые – память и т.п.

Неисчерпаемые – круговой конвейер (выход одной программы поступает на вход другой команды)

Пример:

```
ls | grep “*.zip” | ls
```

1.5 Операционная система Unix

ОС Unix была разработана корпорацией **AT&T** в **Bell Labs**.

Все работающие программы в Unix представлены в виде конкурирующих процессов и потоков. Процессорное время делится между процессами и потоками.

В ОС Unix реализован единый интерфейс для обмена информацией между процессами, файлами и внешними устройствами. Все источники и потребители ресурсов унифицированы от понятия файла. Все действия в рамках Unix систем координируются ядром системы.

- Управление процессами и потоками
- Распределение и перераспределение оперативной памяти
- Выполнение системных вызовов
- Распределение и перераспределение внешней памяти
- Управление устройствами ввода\вывода
- Обработка сигналов
- Учет и контроль за параметрами ОС

Достоинства ОС Unix связаны с ее инструментальностью (насыщенность различными программными средствами), мобильностью (удобство переноса с одной платформы на другую), сетевую направленность.

К **недостаткам** Unix систем можно отнести следующие: не поддерживается режим реального времени, снижение эффективности при решении однотипных задач.

2 Управление процессами

2.1 Понятие процесса

Процесс – это программа в стадии выполнения вместе с текущим значением счетчика команд, регистров и переменных.

Процессы называются **параллельными,** если они существуют одновременно.

Если параллельные процессы работают совершенно независимо друг от друга, они называются **асинхронными**.

2.2 Состояния процесса

В период своего существования процесс проходит через ряд дискретных состояний:

1. Процесс выполняется: в данный момент времени ему выделен процессор. На выполнение процесс запускает *планировщик*.
2. Процесс готов: может сразу использовать процессор, предоставленный в его распоряжение.
3. Процесс блокирован: ожидает появление какого-либо события. *Например*, идет ввод данных с клавиатуры или `sleep(5)`.
4. Приостановлен-готов
5. Приостановлен-блокирован

Примечание. Если поддерживаются *потоки*, то указанные выше состояния поддерживаются на уровне потоков.

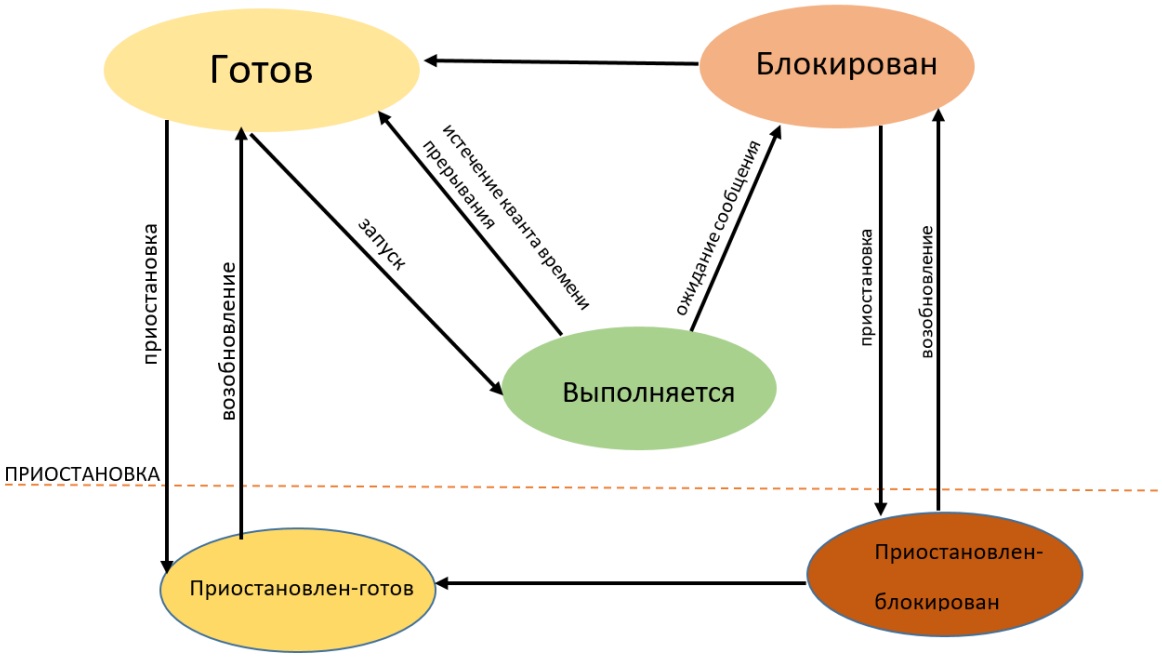


Рисунок 1 (Схема переходов состояний процессов)

Когда пользователь запускает программу на выполнение, ОС создает соответствующий процесс, который устанавливается в конец списка готовых процессов. Планировщик запускает самый высокоприоритетный процесс из списка на один квант времени (стандартно 100 мс). После того, как процесс отработает свой квант времени, он снова помещается в список готовых процессов. Если произойдет какое-либо событие, то процесс помещается в список блокированных.

2.3 Дескриптор процесса

Представителем процесса в составе Операционной системы является **блок управления процессом** (или **дескриптор** процесса).

Дескриптор – это номер строки в таблице процессов, где каждая строка – описание одного процесса или, иными словами, – это структура данных, содержащая:

- идентификатор процесса (PID – целое число)
- указатели памяти
- приоритет
- состояние
- указатель выделенных ресурсов
- область сохранения регистров

2.4 Операции над процессами

Операционная система предусматривает следующие **операции** над процессами:

- создание
- уничтожение
- возобновление
- приостановка
- изменение приоритета
- блокировка
- пробуждение
- запуск (выбор)

2.4.1 Создание процесса

Создание процесса состоит из следующих этапов:

- присвоение PID
- включение PID в список имен, известных системе (Найти первую пустую строку в таблице процессов и закинуть туда)
- определение начального приоритета
- выделение процессу начальных ресурсов
- выделение начальной памяти процессу

2.5 Общие ресурсы процессов

Когда процесс обращается к общим ресурсам с другими процессами, он исключает возможность одновременного обращения к этим ресурсам. Это явление называется **взаимно-исключением** процесса.

Семафор – это системная переменная (или переменная ядра операционной системы), доступная для всех процессов и потоков операционной системы. Семафор можно опрашивать и менять значение при помощи специальных операций. Семафоры могут принимать неотрицательные целые значения.

Mutex – упрощенная версия семафора, которая может принимать значения 0 и 1.

2.6 Потоки управления

У любого процесса есть по крайней мере один **поток управления** – последовательность команд для выполнения какого-то действия – и один **счетчик команд**. Но в некоторых операционных системах есть возможность создания нескольких потоков управления в одном процессе.

Такие потоки получили название **нити** (***Thread***) или **легковесных процессов**. У всех нитей процесса одно адресное пространство, но свои счетчики команд, регистры и состояния. Информация о нитях заносится в таблицу, которую ведет операционная система, и каждая запись этой таблицы описывает отдельный поток.

2.6.1 Волокно

Волокно (***Fiber***) – это поток, который выполняется только в адресном пространстве пользователя (не используя системные вызовы напрямую)

2.6.2 Задание

Задание (*Task)* – набор из одного или нескольких процессов, выполняемых как единое целое.

2.6.3 Событие

Событие (*Event)* – механизм синхронизации потоков (процессов). Событие принимает значение 0 или 1 и бывает двух типов:

- сигнализирующее (сбрасываемое вручную)
- несигнализирующее (сбрасываемое автоматически)

2.6.4 Критическая секция

Критическая секция (*Critical section)* – механизм для синхронизации, подобный мьютексам, но не являющийся частью ядра.

2.6.5 Тупик

Тупик (*Deadlock/клинч*) – процесс находится в состоянии тупика, если он ожидает события, которое никогда не произойдет.

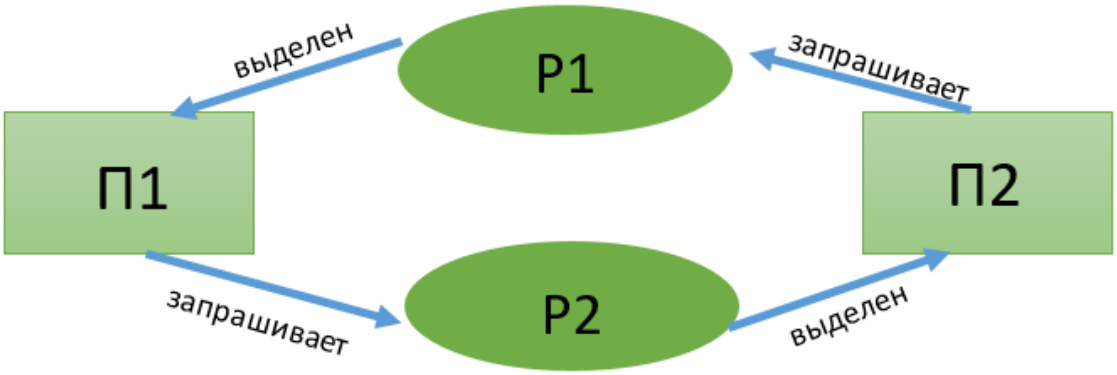


Рисунок 2 (Схема получения тупика)

2.7 Состояния процессора при эксплуатации ОС Unix

В каждый момент времени процессор может находиться в одном из 3-х состояний:

1. Система – если выполняются команды ядра ОС
2. Процесс – если выполняются команды работающего процесса
3. Ожидание – если в системе нет процессов или потоков, готовых к выполнению

Переход **процесс->система** может быть вызван одной из следующих причин:

- Выполнения системного вызова
- Прерывание: по таймеру, по сигналу

Процесс->ожидание:

- Выполняющийся процесс завершился, а других нет

Система->процесс:

- Завершилось выполнение системного вызова
- Завершилась обработка прерывания
- Запуск программы

Система->ожидание:

- Когда завершается обработка прерывания, а новых процессов\потоков для запуска нет

Ожидание->процесс:

- появляется процесс, готовый к выполнению

Ожидание->система:

- прерывание

2.8 Процессы в Unix

При загрузке процесса в оперативную память, процесс разделяется на 4 сегмента:

2.8.1 Контекст процесса

Таблица, в которой хранится информация о процессе

a. *Машинные регистры (область сохранения регистров, чтобы использовать для рестарта процесса)*
b. *Состояние системного вызова (информация о текущем системном вызове, включая параметры и результаты)*
c. *Таблица пользовательских дескрипторов файлов процесса*
d. *Учетная информация – указатель на таблицу, учитывающую процессорное время, используемое процессом*
e. *Стек ядра – фиксированный стек для работы процесса в режиме системы*

Доступ к контексту процесса имеют только системные вызовы. Контекст процесса не относится к адресному пространству процесса, однако контекст процесса подвергается своппингу совместно с процедурным сегментом и сегментом данных.

2. **Процедурный сегмент**
   Включает команды и константы
3. **Сегмент данных**
   Данные, инициируемые при компиляции
4. **Динамический сегмент**
   Данные, не инициируемые при компиляции

Процедурный сегмент и сегмент данных составляют **образ процесса**.

2.8.2 Дескриптор процесса

Кроме контекстов процессов, ядро поддерживает также таблицу процессов, являющуюся резидентной (всегда в оперативной памяти). Каждая запись этой таблицы – дескриптор процесса – описывает один процесс и содержит атрибуты:

- идентификатор процесса (PID – целое число)
- идентификатор родительского процесса PPID
- идентификатор владельца процесса UID
- идентификатор группы GID
- приоритет[1]
- состояние[1]
- счетчик использования процессорного времени CPU
- события, ожидаемые процессом (Events)
- сигнальная маска процесса
- время до истечения интервала будильника
- указатель на процедурный сегмент
- указатель на сегмент данных

[1] В многопоточных системах управление берет на себя поток

- указатель на динамический сегмент
- указатель внешней памяти

Если используется страничная организация памяти, то используются указатель на страницы.

2.8.3 Создание процесса в Unix

Все процессы в ОС Unix, за исключением процесса с PID=0, создаются использованием системного вызова fork().

Происходит обращение к ядру ОС, и этот процесс ищет первую свободную строку в таблице процессов, в которые можно записать данные о новом процессе. Когда такая строка находится, системный вызов fork() копирует туда данные родительского процесса. Затем fork выделяет память для сегмента данных, динамического сегмента, контекста процесса, куда копируются соответствующие сегменты родительского процесса. Процесс-сын получает идентификатор, затем настраивается его карта памяти и регистры, кроме того процессу-сыну предоставляется доступ к файлам родительского процесса, посредством копирования таблицы пользовательских дескрипторов файлов из контекста родительского процесса.

Карта памяти процесса-сына настраивается по следующему принципу: процессу-сыну выделяются новые таблицы страниц, но эти таблицы указывают на страницы родительского процесса, при этом они помечаются как доступные только для чтения. Когда процесс-сын пытается что-то записать в такую страницу, происходит прерывание, при обработке которого ядро выделяет процессу-сыну новую копию этой страницы с доступом на запись. Таким образом, копируются только те страницы, в которые процесс-сын пишет новые данные.

Страницы с процедурным сегментом не копируются.

2.8.4 Планирование в Unix

Определение последовательности выполнения или потоков предусматривает 2 алгоритма: **низкоуровневый** выбирает следующий процесс или поток из набора процессов или потоков в памяти, готовых к работе, **высокоуровневый** алгоритм перемещает процессы из памяти на диск и обратно (своппинг).

Оба алгоритма реализованы в процессе с PID=0 – swapper.

В низкоуровневом алгоритме используется несколько **очередей**. С каждой очередью связан диапазон непересекающихся значений приоритетов. Когда запускается низкоуровневый алгоритм он ищет очередь, начиная с самого высокого приоритета, пока не находит очередь, в которой хотя бы один процесс или поток. После этого из этой очереди выбирается и запускается первый процесс. Если процесс использует весь свой квант времени (100мс) он помещается обратно в конец очереди, а алгоритм планирования запускается снова. Раз в секунду приоритет каждого процесса пересчитывается по следующей формуле:

$$PRI = \mu CPU + \eta NICE + BASE$$

После пересчета приоритетов, процесс помещается в соответствующую очередь.

CPU – среднее значение тиков таймера, в течение которого процесс выполнялся. При каждом тике таймера счетчик использования процесса увеличивается на 1

$NICE$ – пользовательская составляющая приоритета процесса с диапазоном от -20 до 20

$BASE$ – отрицательное число, которое добавляется после выхода из состояния блокировки.

Когда процесс выполняет системный вызов, он блокируется, пока не будет выполнен вызов, и удаляется из очереди.

2.8.5 Информационная связь между процессами

[Отсутствует, необходимо взять]

3 Ядро операционной системы

3.1 Понятие ядра ОС

Ядро ОС – набор программ и данных, которые постоянно находятся в оперативной памяти компьютера.

Ядро ОС представляет собой лишь небольшую часть кода операционной системы в целом, однако оно относится к числу наиболее интенсивно используемых компонент системы. Ядро постоянно размещается в оперативной памяти, в то время как другие части операционной системы перемещаются во внешнюю память и обратно по мере необходимости.

3.2 Функции ядра операционной системы

Ядро операционной системы, как правило, содержит программы для реализации следующих функций:

- управление процессами (потоками)
- поддержка работы файловой системы
- управление устройствами ввода\вывода
- обработка прерываний
- поддержка распределения и перераспределения ОЗУ
- функции учета и контроля

3.3 Ядро ОС Unix

Ядро Unix состоит из двух секций:

- секция управляющих структур – все таблицы, которые ведет ядро ОС
- программная секция – набор программ, которые выполняют функции ядра
    - диспетчер процессов (практически не изменяется при переносе систем на другую машину)
        - определение последовательности процессов или потоков
        - обработка системных вызовов
        - распределение ресурсов системы
    - диспетчер устройств ввода\вывода (является машинно-зависимой и требует доработок при переходе на другую вычислительную платформу)
        - обеспечивает и контролирует передачу информации между оперативной памятью и внешними устройствами
            - драйверы символьных устройств
            - драйверы дисковых устройств
            - драйверы сетевых устройств
            - дисциплины линии

3.4 Обработка прерываний

Для обработки каждого из типов прерываний в составе каждой операционной системы есть программы, называемые **обработчиками прерываний**.

Когда происходит прерывание, операционная система запоминает состояние прерванного процесса и передает управление соответствующему обработчику прерываний. После завершения обработки прерывания операционная система запускает на выполнение либо тот процесс, который выполнялся в момент прерывания, либо процесс из списка готовых с наивысшим приоритетом.

3.4.1 Типы прерываний

Тип прерывания	Инициатор	Примеры
По вызову супервизора	Работающий процесс	
Ввода/вывода	Аппаратура ввода/вывода	завершается выполнение операций ввода или вывода происходит ошибка ввода\вывода устройство ввода\вывода переходит в другое состояние (занято, свободно)
Внешние прерывания	Прерывания от таймера	истекает квант времени работающего процесса
	Клавиша прерывания на пульте управления	Ctrl+C
	Прерывание от другого процессора	
	Прерывание по ошибке программы	Деление на ноль, переполнение стека
	Прерывание по рестарту	Пользователь нажимает на пульте управления кнопку рестарта или, когда другой процессор в мультипроцессорной системе присылает команду рестарта
	По ошибке компьютера	Прерывание по питанию, выход из строя аппаратуры

3.5 Системные вызовы

Когда во время выполнения программы необходимо выполнить другую программу или предоставить системные функции, используется аппарат системных вызовов. Синтаксически применение системного вызова похоже на вызов подпрограммы, однако исполняемый код системного вызова находится в ядре операционной системы, а не в загрузочном модуле.

3.5.1 Системные вызовы управления процессами

3.5.1.1 system

Позволяет запустить новый процесс в рамках уже выполняющегося процесса.

```
int system(char* cmd);
```

`cmd` – текст командной строки

Возвращаемое значение:
Возвращает `0` в случае успеха, `-1` в случае неудачи.

Пример:

```
system("ls"); //Вывести список файлов
system("./a.exe"); //Запуск программы в текущей директории
system("copy a.c b.c"); //Копировать
```

3.5.1.2 execl

Запускает новую программу вместо уже выполняющейся, без возврата вызывающей программы.

```
int execl(char* path, char* arg0, char* arg1, ...,
                                    NULL);
```

`path` – полное имя вызываемой программы
`argN` – внешние аргументы вызываемой программы

Возвращаемое значение:
Возвращает `0` в случае успеха, `-1` в случае неудачи.

Пример:

```
execl("/bin/ls", "ls", 0); //Вывести список файлов
//Все, что дальше, не выполняется
some_code();
```

3.5.1.3 fork

Позволяет запустить параллельный процесс. После выполнения программа разделяется на две идентичные копии, которые продолжают выполняться как два независимых процесса.

```
int fork();
```

Возвращаемое значение:
Возвращает `PID` дочернего процесса или `0` для дочернего процесса.

Пример:

```
if (fork() == 0){
```

```
        //процесс сын
} else {
        //процесс-отец
}
```

3.5.1.4 wait

Применяется в процессе-отце и ждет, когда завершится процесс-сын.

```
int wait(int* s);
```

Возвращаемое значение:
`PID` завершившегося дочернего процесса, а в свой аргумент `s` записывает причину гибели процесса-сына.

В `s` 7-мь младших битов (0-6) содержат нули, если сын завершен с помощью `exit()`, или номер сигнала (положительное целое число)

7-й бит содержит `0`, если образ процесса сохранен, `1` если не сохранен. С 8 по 15 содержат аргумент функции `exit()`

Пример:

```
if (fork() == 0){
        //процесс сын
} else {
        //процесс-отец
        pid = wait(&s);
        printf("s=%d\n", s);
}
```

3.5.2 Системные вызовы работы с файлами

Все операции ввода\вывода в операционной системе Unix связаны с вводом\выводом в файл. Файлы стандартного ввода, стандартного вывода и стандартного протокола открываются для любого вновь порожденного процесса и имеют пользовательские дескрипторы файлов 0, 1 и 2 соответственно. Когда завершается процесс, все открытые в нем файлы автоматически закрываются.

3.5.2.1 open

Открывает файл.

```
int open(char* name, int flag);
```

`name` – указатель на строку символов, содержащую полное имя открываемого файла
`flag` – целое значение, указывающее, какие права доступа к файлу разрешены текущему процессу

Возвращаемое значение:

При успешном завершении возвращает целое неотрицательное число, которое равно номеру первой свободной строки в таблице пользовательских дескрипторов файлов.

Пример:

```
int fd;
close(1);
fd = open("a.txt", 1); //fd===1
```

3.5.2.2 creat

Создает и открывает файл на запись.

```
int creat(char* name, int mode);
```

`name` – указатель на строку символов, содержащую полное имя открываемого файла
`mode` – права доступа к файлу

Возвращаемое значение:
Номер первой свободной строки в таблице пользовательских дескрипторов файлов в случае успеха.

Пример:

```
int fd;
close(1);
fd = creat("a.txt", 0777); //fd===1
```

3.5.2.3 dup

Позволяет получить копию пользовательского дескриптора открытого файла.

```
int dup(int old_fd);
```

`old_fd` – номер дескриптора, который нужно продублировать

Возвращаемое значение:
Номер первой свободной строки в таблице пользовательских дескрипторов файлов, в случае успеха, в которую скопировался дескриптор из аргумента.

Пример:

```
fd_copied = dup(1);
```

3.5.2.4 dup2

Возвращает копию old_fd, равную указанному значению new_fd, либо -1. Если значение указывает на файл, который открыт, то в результате применения dup2 этот файл будет закрыт.

```
int dup2 (int old_fd, int new_fd);
```

`old_fd` – номер дескриптора, который нужно продублировать
`new_fd` – номер дескриптора, куда нужно продублировать

Пример:

```
fd_copied = dup2(5,6);
```

3.5.3 Системные вызовы ввода-вывода

3.5.3.1 read

Для ввода-вывода информации в UNIX используются системные вызовы read() и write(). Они не предусматривают буферизации, а используют непосредственное обращение к ОС.

```
int read (int fd, char *buf, int n);
```

`fd` – пользовательские дескрипторы открытого файла.
`buf` – указатель на буфер, куда должны поступать данные, считанные из файла.
`n` – количество байт, которые необходимо считать.

Возвращаемое значение:
`read()` возвращает -1 в случае аварии. В случае успеха – это количество байтов, успешно прочитанных и сохраненных в аргументе `*buf`. Как правило, оно обычно равно `n`. Если файл содержит менее n байтов, то возвращается меньшее n число. Если встречается признак конца файла, то системный вызов возвращает 0.

Пример:

```
int fd; char buf[80];
fd = open (“a.txt”, 0);
fd = read (fd, buf, 80);
//fd ==39 //количество символов в строке, что в a.txt
0 == read (3, &buf[fd], 80) // read() == 0 признак конца файла
```

3.5.3.2 write

```
int write (int fd, char *buf, int n);
```

`fd` – пользовательские дескрипторы открытого файла для записи
`*buf` – указатель на буфер, откуда должны поступать данные
`n` – количество байт, которые необходимо записать.

Возвращаемое значение:
`write` возвращает -1 в случае аварии. Или количество байт, успешно записанных в файл в случае успеха.

Пример:

```
int fd = creat(“a.txt”, 0664);

2 == write (fd, “aaa”, 2);

// 10 == write (fd, “aaa”, 10); - другой пример
```

3.5.3.3 close

Закрывает открытый файл в случае успеха.

```
int close (int fd);
```

`fd` – пользовательский дескриптор открытого файла

Возвращаемое значение:
`close()` возвращает -1 в случае аварии. 0 в случае успеха.

Пример:

```
0 == close(1);

-1 == close(5);
```

3.5.3.4 pipe

Создает коммуникационный канал межпроцессорной связи между двумя взаимосвязанными процессами.

Синхронизация обмена через межпроцессорный канал построена таким образом, что, если процесс читает пустой канал, он будет ждать появление данных.

Если в канале осталось много несчитанной информации, записывающий процесс будет ждать освобождения места в канале. Если у канала доступ на запись закрыт, то будет получен код ответа 0, что означает конец файла. Если у канала доступ на чтение закрыт, то при записи в канал будет получено SIGPIPE (разрыв канала).

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

`Fd[2]` – массив куда засылаются два пользовательских дескриптора файла.
`Fd[0]` – для чтения
`Fd[1]` – для записи

Возвращаемое значение:
`close()` возвращает -1 в случае аварии. 0 в случае успеха.

Пример:

```
#include <stdio.h>
#include<sys/types.h>
#include <unistd.h>
#include <sys/wait.h>

void main () {
        int c, fd, s, d;
        if (fork()!=0)
              d = wait (&s);
        else {
              fd = creat("a.txt", 0644);
              close(1);
              dup2(fd, 1);
              close(fd);
```

```
                execl("/bin/ls","ls","-l",0);
                exit(1);
        }
}
==ЧТО БУДЕТ==
Отец
0 /dev/tty
1 /dev/tty
2 /dev/tty

Сын
0 /dev/tty
// 1 /dev/tty закрылся после close(1)
//После информация ls записывается в a.txt
2 ./ a.txt // после dup2
2 /dev/tty
// 3 ./ a.txt Закрылся после close(fd)
```

3.5.4 Системные вызовы для обработки сигналов

Каждая запись таблицы процессов в Unix содержит массив сигнальных флагов, по одному для каждого сигнала, определенного в системе. Когда во время выполнения поступает сигнал, ядро операционной системы в сигнальной маске устанавливает соответствующий флаг для процесса-получателя. Если процесс находится в режиме ожидания, то ядро будит этот процесс и ставит в очередь на выполнение.

Ядро будет проверять каким образом процесс будет реагировать на данный сигнал. Сигнальная маска процесса определяет, какие сигналы во время выполнения процесса блокируются. Разблокировка и обработка сигнала выполняется в данном процессе.

При создании процесс наследует сигнальную маску своего родителя, но ни один сигнал, ожидающий обработки родителем, к процессу-сыну не пропускается.

3.5.4.1 sigprocmask

Запрашивает и создает сигнальную маску.

```
int sigprocmask(int cmd, const sigset_t* new, sigset_t*
                                old);
```

new определяет набор сигналов, которые будут добавлены в сигнальную маску процесса или удалены.

cmd указывает, как значение new должно быть использовано:

- SIG_SETMASK заменяет сигнальную маску процесса аргументом *new*
- SIG_BLOCK добавляет сигналы из аргумента *new*
- SIG_UNBLOCK удаляет из сигнальной маске процесса сигналы аргумента *new*

Если new===NULL, то cmd игнорируется и сигнальная маска не изменяется.

old – это переменная, которой присваивается текущая маска процесса до вызова sigprocmask

Возвращаемое значение:
При успешном завершении возвращает 0, при неудаче – 1.

3.5.4.2 sigemptyset
Обнуляет все биты в переменной mask.

```
int sigemptyset(sigset_t* mask);
```

3.5.4.3 sigaddset
Устанавливает бит, соответствующий сигналу sig в переменной mask

```
int sigaddset(sigset_t* mask, int sig);
```

3.5.4.4 sigdelset
Обнуляет бит, соответствующий сигналу sig в переменной mask

```
int sigdelset(sigset_t* mask, int sig);
```

3.5.4.5 sigfillset
Устанавливает все биты в переменной mask

```
int sigfillset(sigset_t* mask);
```

3.5.4.6 sigismember
Возвращает бит, соответствующий сигналу sig в переменной mask

```
int sigismember(const sigset_t* mask, int sig);
```

3.5.4.7 sigaction
Определяет обработчик сигналов

```
int sigaction(int signum, const struct sigaction *act,
              struct sigaction *old_act);
```

Возвращаемое значение:
При успешном завершении возвращает 0, при неудаче – 1.

```
struct sigaction{
    void sa_handler(); // функция обработки сигнала
    sigset_t sa_mask;  // сигнальная маска процесса
    int sa_flags;       // позволяет имитировать системный вызов
сигнала
};
```

Пример:

```
int k = 0;
void func(){
```

```
    printf("K = %d", ++k);
}

int main()
{
    sigset_t mask;
    struct sigaction n;
    n.sa_handler = func;
    sigemptyset(&n.sa_mask);
    sigprocmask(0,0,&n.sa_mask);
    n.sa_flag = 0;
    sigaction(SIGINT, &n, 0);

    //some actions next
}
```

4 Управление файловой системой

4.1 Файл, запись, поле, байт

Файл – поименованная группа взаимосвязанных записей.

Запись – это группа взаимосвязанных полей.

Поле – это группа последовательных байтов или взаимосвязанных символов.

Байт – это 8 бит (набор из 8 ноликов и единичек)

4.2 Организация файлов

Под **организацией файлов** понимают способ расположения записей файлов во внешней памяти.

Различают следующие виды организации файлов:

- **Последовательная** – записи располагаются в физическом порядке.
- **Индексно-последовательная** – при индексно-последовательной организации записи располагаются в соответствие со значением ключей, содержащимся в каждой записи. Доступ к ним может осуществляться последовательно в порядке возрастания или убывания ключей, или прямо по ключу.
- **Прямая (произвольная)** – доступ к записям производится напрямую по физическому адресу на запоминающих устройствах прямого доступа

Библиотечная – представляет собой файл, состоящий из последовательных подфайлов

4.3 Файловая система

Файлами управляет операционная система. Их структура, именование, защита, использование относятся к той части операционной системы, которая называется **файловая система**.

Файловые системы содержат следующие **средства**:

- средство доступа к файлу
- средство управления файлами
- средство размещения файлов во внешней памяти
- средство сохранения целостности файла

4.4 Дескриптор файла

Информация, необходимая операционной системе для выполнения различных операций с файлами, содержится в **дескрипторе** файла (блок управления файла). Это структура данных, которая обычно хранится во внешней

памяти и передается в оперативную память только во время открытия файла. Строка в таблице дескрипторов файлов, содержащая:

- имя файла
- адрес
- размер
- дата и время создания\изменения
- права доступа
- тип файла
- диспозиция (постоянный, временный, рабочий)

4.5 Топология файловой системы

4.5.1 Реализация файлов

4.5.1.1 Неразрывные файлы

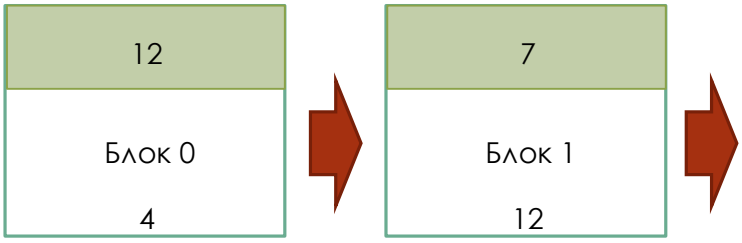
Наиболее простая схема реализации – **неразрывные файлы.** При работе с неразрывными файлами производительность обращения к диску очень высокая, так как весь файл может быть прочитан за одну операцию

Недостатки:

- Высокая фрагментированность. Если захотим убрать файл 1, то потом другой файл может не уместиться на это же место.
- Внесение изменений в диск вызывает сложности.

4.5.1.2 Список блоков

Файл представляется в виде однонаправленного **списка блоков** диска. Первое слово каждого блока указывает на слово, с которого начинается следующий блок. В остальной части блока хранятся данные. В каталоге нужно хранить только адрес первого блока.



Недостатки:

- Чтобы получить доступ к k-ому блоку, необходимо прочитать k-1 блок по очереди

4.5.1.3 Список блоков с индексацией

Указатели на следующие блоки диска хранятся не прямо в блоке, а в отдельной таблице, загруженной в памяти (**File allocation table**)

0	
1	
2	
3	
4	12
5	
…	…
12	7
…	…

Недостатки:

Хранение таблицы в оперативной памяти.

500Мб диска требует таблицы 2 Мб

Пример:

FAT

4.5.1.4 i-node (индексные дескрипторы)

С каждым файлом связана структура данных (**I-узел / индексные дескриптор**), содержащая **атрибуты файла** (адреса блоков, владелец и т.д.). Каждый конкретный I-узел находится в оперативной памяти только тогда, когда соответствующий файл открыт. Если каждый I-узел занимает n байтов, а одновременно открыто k файлов, то в оперативной памяти будет занято n*k байт. Обычно эта величина значительно меньше FAT таблицы и не зависит от объема диска.

Каждый индексный дескриптор содержит следующие поля:

- Права доступа
- Счетчик жестких ссылок
- ID владельца
- ID группы
- Размер файла в байтах
- Время последнего доступа к файлу
- Время последней модификации файла
- Время изменения прав доступа к файлу
- Номер индексного дескриптора файла
- Идентификатор файловой системы
- Тип файла
- Тип организации файла (последовательная, произвольный доступ, библиотечная, индексно-последовательная)
- Диспозиция (постоянная, временная, рабочая)
- Физический адрес на диске

4.5.1.5 NTFS (New technology file system)

При этой схеме реализации файл состоит из атрибутов, каждый из которых представляется набором байтов. Большинство файлов имеет три атрибута:

- имя файла
- 64-битовый идентификатор файла
- поток с данными

При этой схеме реализации предполагается, что каждый **том** (дисковый раздел C: D: E:) содержит файлы, каталоги, байтовые и битовые массивы данных. Каждый том организован как линейная последовательность блоков (в Microsoft

блоки == кластеры). Обращение к блоку осуществляется по их смещению от начала блока.

Основная структура – **таблица MFT.** Каждая запись таблицы имеет длину в 1 Кб. Каждая запись описывает один файл или каталог. Если файл слишком большой, то создается дополнительная запись, чтобы вместить список всех блоков файла. Список свободных записей в таблице учитывается с помощью **битовой карты.**

Первая запись в MFT-таблице описывает сам файл MFT – расположение блоков. Номер первого блока MFT таблицы содержится в загрузочном блоке диска.

Прежде чем работать с файлом, его надо открыть. При открытии файла ОС оперирует указанным именем файла. Запись в каталоге содержит информацию о необходимом блоке диска. Это может быть дисковый адрес, номер I-узла и т.д.

В первых трех случаях запись в каталоге должна содержать информацию и об атрибутах файла (размер файла, время создания)

4.6 Поддержка файловых систем и файлов ОС Unix

Unix поддерживает любые файловые системы. Для каждой файловой системы формируется идентификатор и создается таблица индексных дескрипторов файлов, в которой хранится информация обо всех файлах. Каждая запись в таблице индексных дескрипторов файлов содержит все перечисленные атрибуты файлы и определяется по идентификатору файловой системы и номеру индексного дескриптора.

Идентификатор присваивается файловой системе при выполнении команды `mount`.

Всякий раз, когда создается новый файл, ядро ОС Unix создает новую запись в таблице индексных дескрипторов файлов для сохранения информации о нем. Кроме того, ядро добавляет его имя и номер дескриптора в соответствующий каталог файловой системы. Таблицы индексных дескрипторов файлов содержатся в соответствующих файловых системах на диске, но ядро ведет их копии и в оперативной памяти

Помимо таблицы индексных дескрипторов файлов, ядро Unix ведет также таблицу файлов, в которой отслеживаются все открытые в системе файлы. Каждый процесс в операционной системе Unix поддерживает таблицу пользовательских дескрипторов файлов, в которой отслеживаются все файлы, открытые процессом.

Если индексный дескриптор файла доступен для процесса, ядро ищет в таблице пользовательских дескрипторов файлов первую свободную запись. Номер записи будет возвращен процессу в качестве индексного дескриптора файла.

В таблице пользовательских дескрипторов файлов делается ссылка на номер этой записи в таблице файлов.

В записи таблицы файлов делается ссылка на запись в таблице индексных дескрипторов файлов, в которой хранится индексный дескриптор открываемого файла. В записи таблицы файлов формируется указатель текущей позиции в файле. В записи таблицы файлов заносится информация, в каком режиме открыт

файл. В записи таблицы файлов значение счетчика ссылок устанавливается в соответствие с тем, сколько пользовательских дескрипторов файлов из процессов обращается из данной записи.

Значения счетчика ссылок индексного дескриптора файла увеличивается на единицу. Счетчик определяет сколько файлов таблицы файлов указывает на этот индексный дескриптор. Ядро будет использовать пользовательский дескриптор файла для поиска записи в таблице файлов, соответствующих данному открытому файлу. Далее ядро использует указатель для доступа к записи ТИДФ. Кроме того, ядро использует указатель на позицию файла, чтобы определить, с какого места в файле производить чтение или запись. Ядро проверяет тип файла и вызывает соответствующий драйвер, чтобы начать фактический обмен данными с физическими устройствами.

Резюме. При работе с файлами ОС поддерживает в памяти 2 таблицы: ТИДФ и ТФ, а ТПДФП поддерживается каждым процессом

4.7 Логическая и физическая организация файловой системы

В ОС Unix логическая и физическая структуры файлов не совпадают. Логически файл представляет собой непрерывную цепочку блоков. Физически файл представляет собой набор блоков, которые могут быть разбросаны по всему дисковому пространству.

С точки зрения физической организации файловая система представляет собой совокупность блоков, расположенных на диске.

0. блок начальной загрузки
1. суперблок. Тут содержится заголовок ФС
    a. Размер файловый системы fsize
    b. Количество индексных дескрипторов isize
    c. Указатель на список свободных блоков
2. Блоки, содержащие индексные дескрипторы файлов. Их количество определяется количеством isize.
3. Далее идут блоки, которые содержат данные или свободные блоки.

Для обеспечения быстрого доступа к файлам и эффективной работы файловой системы некоторые ее части во время функционирования ОС Unix должны располагаться в основной памяти: суперблок, ТИД. Дополнительно в оперативной памяти находится таблица файлов.

Поскольку поиск информации в каталогах производится линейно и может занять много времени, в ОС было добавлено кеширование имен. Прежде чем искать имя файла в каталоге, система ищет его в кеше.

Для увеличения производительности стали разбивать диск на группы цилиндров, у каждой из которых имеется свой суперблок, индексные дескрипторы и блоки с данными. Суть этого изменения – хранить ИД и блоки данных ближе друг к другу, чтобы снизить время доступа к ним жестким диском.

При форматировании используют блоки двух размеров. Для маленьких файлов – маленького размера, для больших – большого.

5 Управление оперативной памятью

5.1 Стратегии управления оперативной памятью

Управление оперативной памятью включает в себя следующие **стратегии** (группы алгоритмов):

1. **Выборки** (когда необходимо ввести в память очередной блок программы или данных)
    a. выборка по запросу – когда запрашивает выделение памяти
    b. упреждающая – когда не запрашивает
2. **Размещения** (в какое место оперативной памяти следует помещать поступающий блок программы или данных)
3. **Замещения** (определить, какой блок программы или данных следует вытолкнуть из оперативной памяти, чтобы освободить место для записи поступающих программ или данных)

5.2 Учет свободных областей

Существует 2 способа **учета свободных областей** оперативной памяти:

- **Битовая карта**
    - В битовой карте каждому свободному блоку соответствует бит, равный 0, а каждому занятому – 1
    - Применяется при страничной памяти
- **Связный список свободных**/занятых **блоков**
    - Связный список представляет собой набор записей. Каждая запись отображает область свободной памяти
    - Запись содержит
        - адрес области
        - длину
        - указатель на следующую запись

5.3 Алгоритмы замещения памяти

Существуют следующие алгоритмы замещения памяти:

a. Оптимальный алгоритм замещает тот блок, обращение к которому было раньше других, находящихся в памяти
b. Алгоритм FIFO (первый пришел, первый ушел). В связном списке отслеживается порядок загрузки блоков
c. Алгоритм FIFO вторая попытка. Проверяется, не используется ли он в данный момент. Если используется, то пропускаем блок.
d. Алгоритм LRU (Least Recently Used). Удаляется блок, не использующийся больше всего. Требует специального программного обеспечения
   Ageing алгоритм (программная реализация LRU
e. Not Recently Used. Удаляется блок, не использующийся в последнее время
f. Алгоритм часов.

5.4 Концепции распределения оперативной памяти

При проектировании современных операционных систем используется 2 **концепции распределения** оперативной **памяти**:

- Связная: каждая программа должна занимать один сплошной блок ячеек памяти
- Несвязная: программа разделяется на ряд блоков/сегментов, которые могут размещаться в оперативной памяти в участках, необязательно соседствующих друг с другом

Оверлейные перекрытия – для того, чтобы программа работала, мы подгружаем ее кусками.

5.5 Мультипрограммирование

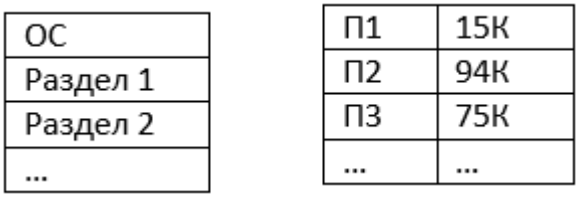


Рисунок 3 (мультипрограммирование с постоянными и переменными разделами)

При **мультипрограммировании с постоянными разделами** создается очередь программ или заданий. Используются абсолютные адреса. При использовании относительных адресов можно формировать единую очередь и помещаться в любой свободный раздел, размер которого это позволяет.

При **мультипрограммировании с переменными разделами** каждой программе выделяется столько памяти, сколько она требует.

5.6 Фрагментация, своппинг, виртуальная память

Фрагментация – очень много свободных «дыр» в памяти.

[Если П2 завершилась, то П3 «поднимется» наверх, и для П4 будет больше свободной оперативной памяти]

Своппинг (Swapping)

Подход своппинга не требует, чтобы программа постоянно находилась в оперативной памяти. В простых системах со своппингом в каждый момент времени оперативную память занимает только одна программа.

Виртуальная память

Суть концепции виртуальной памяти заключается в том, что адреса, к которым обращается процесс, являются вымышленными, то есть отличаются от реально существующих в оперативной памяти. Адреса, к которым обращается процесс, называются **виртуальными адресами**, а адреса, реально существующие в оперативной памяти, называются **реальными адресами.**

ОС загружает программу не целиком, а небольшими кусками.

Существует 2 способа реализации виртуальной памяти:

- страничная
- сегментная

Преобразование виртуальных адресов в реальные во время выполнения процесса обеспечивает операционная система. Она ведет таблицы, показывающие, какие ячейки виртуальной памяти на данный момент находятся в реальной памяти и где они размещаются. Чтобы сократить объем информации отображения, элементы информации группируются в блоки, и операционная система следит за тем, в каких участках оперативной памяти размещаются блоки оперативной памяти.

```
V | (b,d)
```

V-виртуальный адрес, b – номер блока, d – смещение

Преобразование виртуального адреса в реальный происходит следующим образом. Каждый процесс имеет собственную таблицу отображения блоков, которую ОС ведет в оперативной памяти [таблица страниц]. Адрес A этой таблицы в оперативной памяти загружается в специальный регистр ЦП. Номер блока b суммируется с базовым адресом A таблицы отображения блоков, образуя реальный адрес b' строки таблицы. Затем d и и b' суммируются, образуя реальный адрес памяти.

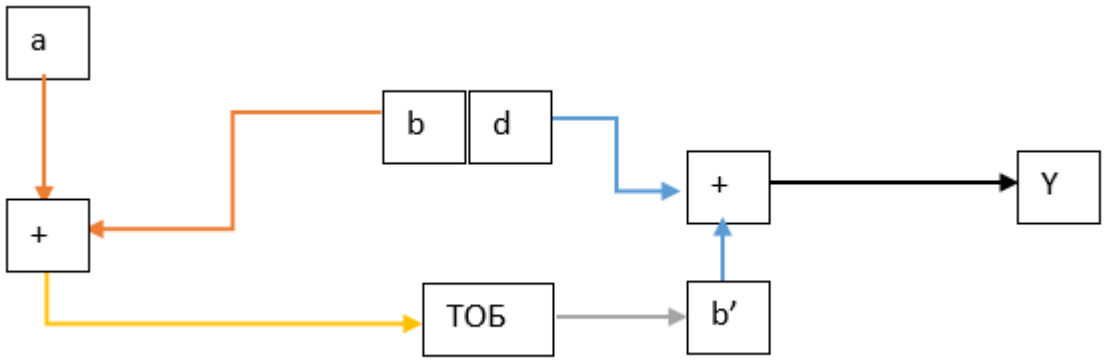


Рисунок 4 (Преобразование виртуального адреса в реальный)

5.7 Управление памятью в ОС Unix

5.7.1 Структура оперативной памяти Unix

Оперативная память в операционной системе Unix делится на три части:

1. *Операционная система*
2. *Карта памяти* – таблица, размер которой определен оперативной памятью. Характеризует каждую страницу памяти.

3. *Страницы* (страница процедурного сегмента, сегмента данных, динамического сегмента какого-либо процесса, либо находящаяся в списке свободных страниц)

Для каждой страницы в карте памяти есть запись фиксированной длинны. Первые два поля этой записи используются только тогда, когда соответствующие страницы находятся в **списке свободных страниц**. (Если два поля пустых, то страницы заняты).

Номер следующей записи	Номер предыдущей записи	Номер блока диска	Номер блока устройства	Хэш код блока	PID	Тип сегмента	Смещение в сегменте	Флаг(Бит) Использования
5	7							

У каждой страницы в оперативной памяти есть место, где она хранится на диске, в которое она помещается, когда выгружается из памяти. Следующие три блока **характеризует место хранения не диске**. Далее три поля характеризует, **к какому процессу** относятся страницы. **Флаг(бит)** использования нужен для алгоритма замещения.

При запуске (при выполнении) процесс может вызвать **страничное прерывание**, если одна или несколько его страниц не окажется в памяти. При страничном прерывании операционная система

1. Берет первую страницу из списка свободных страниц
2. Считывает в нее требуемую страницу
3. И удаляет ее из списка свободных страниц

Если список свободных страниц пуст, то выполнение процесса приостанавливается до тех пор, пока страничный демон не освободит одну или несколько страниц.

5.7.2 Управление памятью на основе своппинга

Большинство Unix систем для управления памятью используют подход своппинга. Передачу файлов между диском и памятью осуществляет верхний уровень планировщика (процесс 0 / своппер).

Выгрузка данных из памяти на диск инициируется, когда у ядра заканчивается свободная память из-за одного из следующих событий:

- появился новый процесс (fork())
- процесс хочет расширить сегмент данных (brk())
- динамическому сегменту требуется дополнительная память (malloc())
- добавляется (отображаемая\разделяемая) память
- когда наступает время запустить процесс уже долго находящийся на диске, часто бывает необходимо освободить место для него

Выбирая жертву, своппер рассматривает процессы, находящиеся в состоянии «блокирован». Если такие процессы нашлись, то из них выбирается процесс с наивысшим значением суммы приоритета и времени пребывания в

памяти (PRI+TIME). Если блокированных процессов нет, то на основе того же критерия выбирается процесс в состоянии «готов».

Каждые несколько секунд (по умолчанию 2 секунды) своппер исследует список выгруженных процессов, проверяя, не готов ли какой-нибудь из них к работе. Если процессы в состоянии «готов» обнаруживаются, из них выбирается процесс, дольше всех находящийся на диске. Дальше своппер проверяет, будет ли этот своппинг легкий или тяжелый. Легкий своппинг не требует дополнительного освобождения памяти, а тяжелый – это процедура, при которой для загрузки в оперативную память выгруженного процесса из памяти требуется удалить один или несколько других процессов. Эта процедура выполняется до тех пор, пока не выполнится одно из условий:

- на диске нет процессов в состоянии «готов»
- в памяти нет места

Чтобы не терять большую часть производительности на своппинг, ни один процесс не выгружается из памяти на диск, если он находится в памяти менее 2 секунд

5.7.3 Управление памятью на основе страничной подкачки

Для предоставления возможности работы с программами больших размеров практически во всех Unix системах к системе управления памятью добавлена страничная подкачка. Суть этого подхода заключается в следующем: чтобы работать, процессу не нужно целиком находиться в памяти. Все что требуется, это контекст процесса и таблица страниц. Если они загружены в память, то процесс считается находящимся в памяти и может быть загружен планировщиком.

Страницы с процедурным сегментом, сегментом данных и динамическим сегментом загружаются в память по мере необходимости обращения к ним. Страничная подкачка реализуется частично процессом с PID=0 и частично процессом «страничный демон» PID=2. Страничный демон периодически запускается и смотрит: есть ли для него работа. Если он обнаруживает, что количество страниц в списке свободных страниц мало, то страничный демон выполняет действия по освобождению дополнительных страниц в памяти.

5.7.4 Алгоритм замещения страниц в ОС Unix

Алгоритм замещения страниц выполняется страничным демоном. **Раз в 250 мс** он просыпается, чтобы сравнить количество свободных страниц системным параметром **lost free** (или min) равным, как правило, одной четверти объема памяти (по умолчанию).

Если число свободных страниц меньше параметра lost free или min, страничный демон начинает переносить страницы из памяти на диск, пока количество свободных страниц не станет равным параметру lost free (или max). Страничный демон **использует модифицированный алгоритм часов.**

Если операционная система обнаруживает, что частота подкачки страниц слишком высока, а количество свободных страниц все время ниже параметра lost free или минимума, swapper (процесс с идентификатором 0) начинает удалять один или несколько процессов (производить своппинг).

Сначала swapper проверяет, есть ли процесс, который бездействовал в течение 20 и более секунд. Если такие процессы есть, из них выбирается бездействующий в течение максимального срока процесс и выгружается на диск. Если таких процессов нет, swapper изучает 4 самых больших процесса (посчитать количество страниц для каждого процесса по карте памяти). Из этих процессов выбирается тот, который выполняется дольше всех и выгружается на диск и так до тех пор, пока не освободится достаточное количество памяти, то есть не будет достигнуто параметра lost free или min. Каждые 2 секунды swapper проверяет, есть ли на диске процесс в состоянии готов и выбирает процесс с наивысшим приоритетом. Загрузка выбранного процесса производится только при условии наличия достаточного количества свободные страниц, чтобы, когда случится неизбежное прерывание, для него нашлись свободные страничные блоки.

6 Управление вводом-выводом

Одна из важнейших функций ядра состоит в управлении устройствами ввода/вывода компьютера. Ядро операционной системы должно давать устройствам ввода\вывода команды, обрабатывать прерывания, обрабатывать ошибки и обеспечивать интерфейс с остальной частью ОС

6.1 Блочные и байтовые устройства

Блочными называют устройства, хранящие информацию в виде адресуемых блоков фиксированного размера. Обычно размеры блоков варьируются от 512 байт до 32768 байт. Важное свойство блочного устройства состоит в том, что каждый блок может быть прочитан **независимо** от остальных блоков. (Это устройство прямого доступа к памяти)

Байтовые устройства принимают или предоставляют поток символов без какой-либо структуры. Они не являются адресуемыми.

6.2 Принцип работы с устройствами ввода-вывода

Прикладные программы обычно общаются с абстрактными устройствами, а зависимую от устройств часть поддерживает ядро операционной системы (программы драйверов).

Устройства ввода/вывода, как правило, состоят из механической части и электронной части. Механический компонент находится в самом устройстве, а электронный компонент или контроллер устройства принимает форму печатной платы, которая сопряжена с одной из шин компьютера или вставляется в слот расширения объединительной платы.

Работа контроллера заключается в преобразовании последовательного потока битов в байтовую последовательность и в выполнении коррекции ошибок. У каждого контроллера есть несколько регистров или портов, с которым может общаться ЦП. У некоторых компьютеров такие регистры являются частью единого адресного пространства системы (часть оперативной памяти), у некоторых – отводится отдельное адресное пространство, в котором выделяются адреса для устройств.

В дополнение к регистрам ввода\вывода часто используются прерывания, при помощи которых контроллер может сообщить процессору, что его регистры готовы для записи или для чтения. Прерывание является электрическим сигналом. Линия запроса аппаратного прерывания Interrupt ReQuest line (IRQ) является одним из входов контроллера, и для каждого компьютера количество таких линий ограничено.

Каждая из линий IRQ связывается с вектором прерываний, который указывает на программу обработки прерывания. Операционная система обменивается с устройством ввода\вывода информацией, записывая команды в регистры контроллера. Передав команду контроллеру, процессор может продолжить свою работу. Затем, когда устройство выполнит команду, контроллер инициирует прерывание, чтобы привлечь ОС для проверки результата.

6.3 Программное обеспечение системы управления вводом-выводом

Программное обеспечение системы управления вводом\выводом разрабатывается независимо от устройств управления вводом\выводом, единообразного именования устройств и многослойность. Оно должно аккумулировать и обрабатывать ошибки устройств ввода\вывода, поддерживать синхронный (блокирующий) и асинхронный (управляемый прерываниями) способы переноса данных, предусматривать буферизацию и различать выделенные устройства ввода-вывода (монопольные) и устройства коллективного пользования.

Уровни программного обеспечения можно представить следующим образом:

1. **Обработчики прерываний** – программа, содержащая набор команд для обработки прерываний
2. **Драйверы** устройств ввода\вывода – программа управления устройством, воспринимающая абстрактные запросы от аппаратно-независимых программ верхних уровней и выполняющая их запросы
3. Независимые от аппаратуры программные модули операционной системы
4. Прикладные программы

Независимые от аппаратуры программные модули реализуют следующие функции:

- единообразный интерфейс для драйверов устройств
- единообразное именование устройств
- защита устройств
- обеспечение аппаратно-независимого размера блока
- буферизация
- сообщения об ошибках и обработка ошибок
- захват и освобождение монопольных устройств

Пользовательские программы:

- допускается использовать любые библиотечные функции
- можно писать свои функции

7 ИНТЕРПРЕТАТОР КОМАНД SHELL

7.1 ОБ ИНТЕРПРЕТАТОРЕ

7.1.1 Определение, стандарты и функции

Командная оболочка UNIX (англ. Unix shell, часто просто «шелл» или «sh») — командный интерпретатор, используемый в операционных системах семейства UNIX, в котором пользователь может либо давать команды операционной системе по отдельности, либо запускать скрипты, состоящие из списка команд.

Существуют **стандарты** POSIX.1, SYSTEM V, а также различные модификации, такие как:

- B-shell (bash) – Создатель Stephen Bourne
- C-shell – Создатель Bill Joy, Калифорнийский Университет Беркли
- K-shell – Создатель David Korn, Bell Labs
- P-shell (POSIX)

Интерпретатор shell **предназначен** для создания интерфейса между пользователем и ядром ОС и выполняет функции:

1. Интерпретация команд
2. Обрабатывает имена файлов, определенные через мета-символы
3. Осуществляет переадресацию ввода-вывода
4. Поддержка командного языка

7.1.2 Встроенные и внешние команды

Встроенные команды – это команды, являющиеся частью интерпретатора, они не требуют при своем выполнении запуск нового процесса.

Внешние команды – это команды интерпретатора, порождающие дополнительный процесс.

7.2 SHELL ПЕРЕМЕННЫЕ

Интерпретатор `shell` допускает создание **shell переменных**. Создание shell переменных выполняется либо оператором присваивания, либо командой read. Чтобы посмотреть значения shell переменной, к shell переменной добавляется знак $, а для обозначения shell переменных может быть использована любая строка.

```
a = 2b
b = саша
read c
echo %a b=&b c
chmod 0700 lab
```

Shell допускает использование **специальных** обозначений и переменных.

Пример. Пусть запущена программа `cp` с аргументами:

```
cp ac bc
```

`$#` - количество всех внешних аргументов

`$0` - cp

`$1` – ac

`$2` – bc

`$?` – результат выполнения последней команды

`$$` - PID текущего процесса

`` ` ` `` - Подстановка результата выполнения команды

`\` - отменяет специальный смысл следующего за ним символа

`' '` – отменяет смысл символов, заключенных в них

`" "` – отменяет смысл всех символов, заключенных в двойные, за исключением \ , “” , ‘’, $

7.3 Основные команды

Язык Shell содержит следующие встроенные команды:

`:` - нуль-команда, возвращает всегда true

`break` – конструкция, чтобы выйти из цикла

`cd` – изменение директории

`continue` – продолжай цикл, начинай следующую операцию

`echo` – записывает внешние аргументы в стандартный поток вывода

`eval` – считает аргумент и выполняет результирующую команду

`exec` – выполняет команду, но не в рамках shell

`exit` – выход из интерпретатора shell

`export` – экспортирование переменных (передача переменных из одного интерпретатора shell в другой)

`pwd` – отображает текущую директорию

`read` – считывает строку текста из стандартного файла ввода

`readonly` – преобразование переменной только для чтения

`return` – выход из функции с отображение кода возврата

`set` – отображение различных параметров для стандартного файла ввода

`shift` – смещает влево командную строку аргументов

`trap` – при получении сигнала выполняет команду

`unlimit` – отображает и устанавливает ресурсы shell

`umask` – отображает и устанавливает права доступа к файлу

`unset` – удаляет из памяти shell переменную или функцию

`wait` – ожидает окончание процесса потомка и сообщает о его завершении

7.3.1 Команда test

`test` – осуществляет проверку файлов, числовых величин, цепочку символов. Можно заменять на []

```
test <выражение> [<и\или\не> <выражение>] ... [...]
```

Проверка файлов:

```
test <флаг> <имя_файла>
```

-f – файл существует и обычный
-d – директория существует
-p – наличие канала
-b – наличие блочного файла
-c – наличие байт-ориентированного файла
-S – существует файл и является сокетом
-s – существует файл и имеет ненулевой размер
-r –w –x – проверка для данного пользователя возможности чтения, записи, исполнения

Проверка чисел

```
test <число> <отношение> <число>
```

-lt **<**
-le **<=**
-gt **>**
-ge **>=**
-eq **==**
-ne **!=**

Работа с символами

`test <строка> = <строка>` -- сравнение строк
`test –n <строка>` -- строка не пустая
`test –z <строка>` -- строка пустая

Смешанная проверка

```
test <выражение> –o <выражение> –a <выражение> !
                    <выражение>
```

-o **или**, -a **и**, ! **не**

7.4 Арифметические выражения

`expr` – вычисляет арифметическое выражение

Допускает сложение, вычитание, умножение, деление, взятие остатка.

`let` – арифметическое выражение

Допускает + , -, /, %, > , <, >=, <=, =, !=, &, |, !

```
a = 2
a = ‘expr $a + 7’
b = ‘expr $a / 3’
c  = ‘expr $a ‘*’ $b’
echo a=$a $b c=$c
```

Конструкция expr позволяет сравнивать строки символов.

```
x = abcde
w = ‘expr = “$x” : ‘abc””
y = ‘expr $x : abd `‘{print $1}’`
echo $w $y
```

7.5 Сравнения, циклы и функции

7.5.1 Конструкция if, then, else

```
if <список>
then <список>
else <список>
fi
```

7.5.2 Конструкция case

```
case <значение> in
модель1) <список>;;
модель2) <список>;;
esac
```

7.5.3 Циклы while

```
while выражение
do
список команд
done
```

7.5.4 Циклы until

```
until выражение
do
список команд
done
```

Пример. Вывести 10 раз Hello

```
x=0
until [ $x –eq 10 ]
do
echo Hello
x=`expr $x + 1`
done
```

7.5.5 Циклы for

```
for переменная [in список]
do
список команд
done
```

Пример.

```
for i in 1 2 3 45
do
echo hello $i
done
#hello 1
#hello 2
#hello 3
#hello 45
```

7.5.6 Функции в языке Shell

```
<имя_функции> ()
{ <команда 1>
…
<Команда N>
}
```

7.6 Обработка имен файлов

Метасимвол `*` означает произвольную строку символов и может располагаться в любом месте имени файла, `?` означает произвольный символ, `[ ]` – альтернативный символ подстановки.

Пример.

```
cat f[1-5]
cat f[1 2 3 4 5 ]
cat *
cat f*
pr f*
cat f?
cat f[1 2 3-5]
```


> Означает перенаправление ввода-вывода

```
ls > a.txt
```

Если файл a.txt существовал, то его содержимое будет потеряно. Если файл a.txt не существовал, то он будет создан. Ошибка будет в случае ограниченных прав доступа.

>> Мягкое перенаправление вывода (слабая переадресация)

```
ls >> a.txt
```

Информация запишется в конце файла, перезаписи не будет.

Переадресация ввода:

```
wc < a.c
```

(если убрать <, то результат не поменяется)

```
mail log10 << let
```

7.7 Конвейеризация и последовательные операции

Интерпретатор shell допускает объединение нескольких команд в **конвейер** их совместного последовательного выполнения. В этом случае их связь осуществляется через межпроцессный канал и результат выполнения одной команды сразу поступает на вход другой команды.

```
who | sort

who | wc – l
```

С помощью операций `&&` и `||` можно формировать последовательные операции выполнения команд.

7.8 Примеры программ на языке Shell

Вывести все внешние аргументы

```
n=0
#Цикл по внешним аргументам выполняемой программы
for i
do
n=`expr $n + 1`
echo Argument $n : $i
done
```

Написать скрипт, который выводит в столбец внешние аргументы при вызове данного скрипта. Если при обращении к скрипту нет внешнего аргумента, то вывести идентификатор текущего процесса.

```
if [ $# -eq 0 ]
then echo PID is $$
else
echo “Arguments: “
while [ $1 = “” ]
do
echo $1
shift
done
fi
```

Вывести на экран имена текстовых файлов `/home/m703` которые представляют собой текстовые файлы и по объему не превышают некоторую заданную величину.

```
dir=`pwd`
cd /home/m703
files=`file * | grep text | cut –f 1 –d :`
for i in $files
do
n=`wc – c $i | awk ‘{ print $1 }’`
if test $n –lt $1
then echo $i
fi
done; cd $dir
```

Осуществлять подсчет итераций до тех пор, пока пользователь не прервет выполнение программы сигналом SIGINT

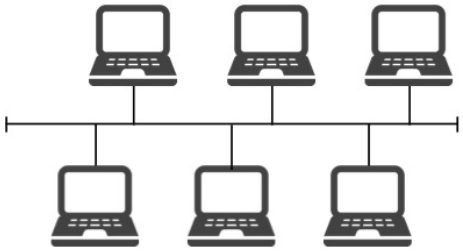
```
trap “func” 2
i=0
func()
{
echo “Количество итераций $i”
exit 1
}

while :
do
i=`expr $i + 1 `
echo $i
sleep 1
done
```

8 Работа с сетью

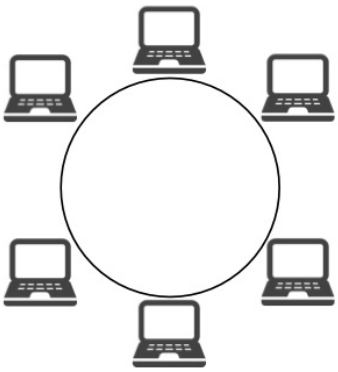
8.1 Построение локальных вычислительных систем

8.1.1 Топология сети

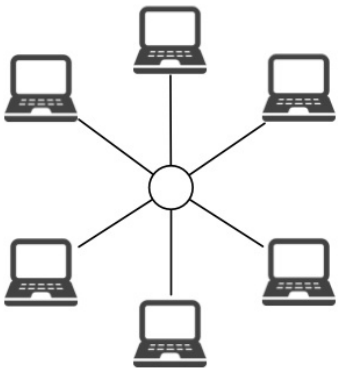
Топология сети – геометрическая форма и физическое расположение компьютеров по отношению к друг другу. Топология сети позволяет сравнивать и классифицировать различные сети.



Шинная топология



Кольцевая топология



Звездная топология

Рисунок 5 Топологии сетевого взаимодействия

Типы соединения:

- оптоволоконный
- витая пара
- коаксиальный кабель
- использовать радио

8.2 Дисциплина линии

Независимо от того, какие лини связи используются для соединения между двумя компьютерами, прежде чем они начнут передавать информацию друг другу, между двумя этими компьютерами должно быть принято соглашение по некоторым вопросам.

- Какой использовать метод синхронизации?
- Какую использовать скорость обмена?
- Какую использовать направленность передачи?
- Какую использовать систему кодирования?

Одна из проблем, которая должна быть решена при передаче информации, состоит в том, как определить, где символ или байт начинается, а где заканчивается (определить метод синхронизации).

8.2.1 Синхронный и асинхронный методы передачи

Одно из решений заключается в том, чтобы послать в обе стороны сообщение, синхронизирующее передачу. Этот метод называется **синхронным**. Используется, когда необходимо передать большой поток информации.

Асинхронный метод заключается в том, что каждый байт информации обрамляется парой битов. (*Прим. ред. это не верно, так как на самом деле асинхронный и синхронный протоколы отличаются тем, что первый оперирует кадрами, а второй – байтами*)

Контроль ошибок передачи:

- контрольная сумма
- избыточные коды, исправляющие часть ошибок

Возможная направленность передачи для линии, соединяющей двух абонентов:

- симплексная (в один конец)
- полудуплексная (в обе стороны по очереди)
- дуплексная (в обе стороны)

Скорость передачи информации измеряется в **бит/сек (бот)** и должна быть одинакова для пары абонентов.

8.3 Основные понятия

IP-адрес состоит из 4-х сегментов – чисел, разделенных точками. Одна часть – адрес сети, а другая – используется для обозначения конкретного компьютера в этой сети. Обычно, сетевая часть занимает первые три сегмента, адрес машины – последний сегмент. (192.168.0.12 – подчеркнута сетевая часть).

Кроме IP-адреса для подключения компьютера к сети необходимы:

- адрес сети
- широковещательный адрес
- адрес шлюза (если таковой имеется)
- адрес DNS-сервера
- маска сети

Адрес сети – это сетевая часть IP-адреса данного компьютера сети плюс ноль (192.168.0.0). Во время работы адрес сети определяет по IP-адресу с помощью маски сети путем поразрядной операции «и».

Широковещательный адрес позволяет данному компьютеру посылать сообщения одновременно всем компьютерам сети. Определяется он следующим образом: машинная часть в IP устанавливается как «255» (192.168.0.255).

Шлюз – один из компьютеров сети, обеспечивающий взаимодействие с другими сетями. Все соединения из данной сети с какой-либо другой сетью и наоборот осуществляются через этот шлюзовой компьютер. Как правило, адрес шлюза имеет ту же сетевую часть, что и IP-адрес данного компьютера. В машинной части стоит «1» (192.168.0.1).

Во многих сетях, включая интернет есть компьютеры, которые работают как **серверы доменных имен**. Задача: преобразовывать доменные имена сетей и хост-компьютеров в IP-адреса. Это позволяет идентифицировать ваш компьютер в сети, не пользуясь IP-адресом.

Маска сети используется для получения адреса сети, к которой подключен данный компьютер. При определении маски сети, IP-адрес хост-компьютера выступает в роли трафарета. Все числа в сетевой части устанавливаются равными «255», а машинной части – равные «0» (255.255.255.0).

Мост – это узел сети с двумя и более интерфейсными платами. Внутренний – разные платы, внешний – одинаковые платы.

Объединенная сеть представляет собой группу узлов сети, которая связывается с помощью моста с группой узлов другой сети. В объединенной сети с несколькими серверами сервер различается по имени сервера и по серийному номеру каждой системы. В объединённой сети помимо уникальных имен серверов каждая сеть имеет уникальный сетевой и межсетевой адрес. Каждая сетевая плата, соединенная в общую систему, должна иметь общий сетевой адрес. Сети, связанные через мосты, должны иметь различные сетевые адреса.

8.4 Протоколы передачи данных

В основном, все UNIX системы конфигурированы на подключение к сетям, в которых используются протоколы **TCP/IP** (Transfer Control Protocol – протокол управления передачей, Internet Protocol – межсетевой протокол). TCP отвечает за отправку и прием информации. IP отвечает за маршрутизацию. Эти 2 протокола базовые.

Другие протоколы:

UDP. User Datagram Protocol. Аналогичен TCP, только в UDP информация может потеряться (он менее безопасный – в TCP используется проверка целостности пакетов). UDP при этом быстрее и используется для передачи больших объемов данных, когда нам не принципиальна потеря нескольких байт.

ARP. Address Resolution Protocol. Протокол разрешения адресов. Определяет физические адреса сетевых интерфейсов по их IP-адресам.

DNS. Domain Name System. Служба доменных имен. Устанавливает соответствие «доменное имя – IP адрес». Пример: hse.ru – 192.168.0.12.

hostname(); - системный вызов который позволяет узнать IP компьютера.

ICMP. Internet Control Message Protocol. Межсетевой протокол управляющих сообщений. Осуществляет выдачу сообщений, позволяющих корректировать маршрутизацию, осуществляемую с помощью протокола IP.

RIP. Routing Information Protocol. Протокол маршрутной информации. Определяет маршрутизацию. Не определяет «куда», а определяет «каким путем». Присутствует сейчас не везде. Иногда заменяется на протокол **OSPF**. Open Shortest Path First. OSPF эффективнее, чем RIP.

RARP. Reverse Address Resolution Protocol. Протокол обратного разрешения адресов. По физическим адресам сетевых интерфейсов определяет IP-адреса.

FTP. File Transfer Protocol. Протокол пересылки файлов из одной системы в другую с использованием протокола TCP. Например, из Unix в Windows.

TFTP. Trivial File Transfer Protocol. Протокол пересылки файлов из одной системы в другую с использованием протокола UDP. Выгодно использовать при большом объеме пересылаемой информации.

TELNET. Протокол прикладного уровня для обеспечения интерактивного доступа к ресурсам удаленного компьютера.

SMTP. Simple Mail Transfer Protocol. Используется для передачи сообщений по электронной почте.

POP. Post Office Protocol. Используется для получения почты с почтовых серверов.

HTTP. Hyper Text Transfer Protocol. Протокол передачи гипертекстовой информации (т.е. документов HTML).

EGP. External Gate Protocol. Протокол внешнего шлюза. Предназначен для маршрутизации по внешним сетям.

IGP. Internal Gate Protocol. Протокол внутреннего шлюза. Предназначен для маршрутизации по локальной сети.

GGP. Gateway-to-Gateway Protocol. Межшлюзовый протокол. Обеспечивает маршрутизацию и пересылку сообщений между интернет шлюзами.

NFS. Net File System. Сетевая файловая система.

NIS, Network Information Service. Сетевая информационная служба. Обеспечивает поддержку пользователя при работе в сети.

SNMP. Simple Network Management Protocol. Протокол управления сетью. Выдает сообщения о состоянии и конфигурации протоколов TCP и IP.

8.5 Настройка и поддержка сети в ОС Unix

Адрес сети, ip адрес компьютера, широковещательный адрес, адрес сервера доменных имён и маска сети вводятся во время инсталляции системы.

8.5.1 Системные файлы, связанные с сетью

/etc/hosts отвечает за связь доменных имен с ip-адресами

Содержит адреса абонентов, с которыми «общение» планируется наиболее часто. В первую очередь смотрится этот файл, прежде чем обращаться к DNS-серверу. За ведение этого списка отвечает администратор системы. Этот файл выглядит так:

```
127.0.0.1    turtle.trek.com    localhost

192.168.0.12    slava.miem.edu.ru
```

Первоначально в файле всегда должна присутствовать запись для локального компьютера с адресом 127.0.0.1 (зарезервированный ip-адрес, который позволяет пользователям ОС Unix связываться с собой в локальном режиме. Иначе его называют «закольцованный интерфейс»)

/etc/networks отвечает за связь доменных имен с адресами сетей

В нем хранятся доменные имена и ip-адреса сетей, с которыми есть соединение в данный момент времени. Выглядит так:

```
loopback    127.0.0.0

miem.edu.ru    192.168.0.0

…
```

/etc/hostname содержит доменное имя компьютера

Содержится имя системы. Этот файл можно редактировать с помощью программы netcfg. Команда называется netconfig. Также есть ifconfig и route (они важны).

/etc/rc.d/init.d/inet (**/etc/rec.d/rc.inet**1, **/etc/rc.inet1** – в разных ОС) содержит команды конфигурирования сетевого интерфейса при начальной загрузке системы

В этом файле находятся команды, обеспечивающие конфигурирование сетевого соединения. Многие записи в этом файле автоматически создаются при использовании команды netconfig в процессе инсталляции. В этом файле находятся команды ifconfig, route, заданы имя системы, адрес сети и другие необходимые адреса. Этот файл можно редактировать при одном условии: если вы владеете навыками программирования на shell

/etc/host.conf содержит опции конфигурирования

/etc/resolv.conf содержит список серверов доменных имен

8.5.2 Сетевые интерфейсы и маршруты

Соединение с сетью система устанавливает посредством конкретного аппаратного интерфейса. Это может быть сетевая плата или модем. Команда **ifconfig** позволяет конфигурировать сетевые интерфейсы. Пакет, являющийся частью данных, по пути функции его назначения проходит по определенному маршруту. Маршрут определяет начальную точку передачи пакета и конечную

точку. Команда **route** обеспечивает необходимую маршрутизацию. Маршрут может быть **статическим** или **динамическим**. Динамический может захватывать больше узлов, но работать быстрее. Это верно с силу ограниченной пропускной способности узлов. В наше время статической маршрутизацией почти не пользуются.

Маршруты содержатся в файле **/proc/net/route**. Этот файл можно просмотреть программой **route**.

```
Destination     Gateway         Genmask         Flags
MSS Window    Use Iface

192.168.189.0   0.0.0.0         255.255.255.0   U
256    0        0 eth3

192.168.189.1   0.0.0.0         255.255.255.255 U
256    0        0 eth3

192.168.189.255 0.0.0.0         255.255.255.255 U
256    0        0 eth3
```

Destination – конечный пункт маршрута

Gateway – хост-имя шлюза, используемое на данном маршруте

Genmask – маска сети

Flags – тип или состояние маршрута (U – активный)

MSS – TCP maximum segment size – максимальный размер сегмента на транспортном уровне

Window – размер окна соединений для данного маршрута

Use – количество пакетов, пересланых по данному маршруту

Iface – тип интерфейса на данном маршруте

В файле маршрутизации должна содержаться хотя бы одна запись, предназначенная для закольцвывающего интрфейса.

8.5.2.1.1 Примеры

```
route add 192.168.0.1
```

add имеет несколько спецификаторов. Если вводится статический маршрут, то спецификатор понадобится для таких параметров как маска сети, шлюз, адрес пункта назначения. Если интерфейс уже конфигурирован командой **ifconfig**, то вся необходимая информация будет взята из данных конфигурации интерфейса, поэтому маршрут обычно задается спецификатором net

```
route add –net 192.168.0.1
```

Если компьютер подключен к сети, то в файле маршрутизации помимо записи о закольцовывающем интерфейсе, должна быть запись, задающая маршрут по умолчанию. Пункт назначения для такого маршрута задается ключевым словом default.

При каждом запуске системы сетевые интерфейсы и таблицы маршрутизации конфигурируются заново. Поэтому для каждого сетевого интерфейса команды **ifconfig** и **route** записываются в файл инициализации.

8.5.2.2 Команда ifconfig:

```
ifconfig интерфейс [флаг] адрес [опции]
```

Интерфейс – имя специального файла, который характеризует данную сетевую плату.

Флаг – может быть «host», либо «net». host – адрес, который идет дальше является адресом host-машины. net – данный ip-адрес – это адрес сети. По умолчанию «host».

Адрес – конкретный ip-адрес.

Опции (ключи) (наиболее популярные):

- **broadcast [адрес]** – широковещательный адрес сети
- **-broadcast** – отмена широковещательного режима.
- **netmask** – ip-маска сети.
- **mtu n** – максимальное число байтов, которое может быть передано через данный интерфейс за 1 передачу. Иначе говоря, максимальный размер передаваемого пакета данных.
- **aftype** – задает имя поддерживаемого семейства адресов (inet / unix). inet по умолчанию.
- **dstaddr [адрес]** – задается ip-адрес конечного соединения типа «точка-точка».
- **[-]allmulti** – включает/выключает режим при котором все поступающие кадры (пакеты) отслеживаются на уровне ядра системы. Позволяет узнать точное число полученных пакетов.
- **pointtopoint [адрес]** – работа указанного интерфейса в режиме «точка-точка», если указан адрес, то он присваивается удаленной системе.

8.5.2.2.1 Примеры

```
ifconfig eth0 192.168.0.1 broadcast 192.104.244.137
netmask 255.255.255.0

ifconfig eth0 192.168.0.1

ifconfig plip0 199.35.209.72 pointtopoint 204.166.254.14

ifconfig ppp0 199.35.209.72 pointtopoint 204.166.254.14

ifconfig adsl0 199.35.209.72 pointtopoint 204.166.254.14
```

```
# plip ppp adsl – интерфейсы для связи с провайдерами

ifconfig lo 127.0.0.1 – процедура конфигурирования
закольцовывающего интерфейса

# Команда ifconfig также полезна для проверки статуса
интерфейса.

ifconfig eth0
```

8.5.2.3 Контроль за состоянием сети

```
ping <ip> #проверка связи с узлом

netstat #позволяет в реальном времени получить состояние
сетевых соединений, а также таблицу маршрутизации

netstat –a #информация о сокетах на системе

netstat –n #IP удаленной и локальной систем

netstat –r #выдает таблицу маршрутизации

netstat –t #информация о TCP гнездах

netstat –u #информация о UDP гнездах

netstat –x #информация о доменных гнездах типа UNIX
```

9 CHANGELOG

06.03.2018 – initial commit

25.03.2018 – добавлены темы «Алгоритм замещения страниц в ОС Unix» и «Структура оперативной памяти ОС Unix». Исправлены ошибки (Contributor: Вячеслав Богданов):

- 3.5.3.4 исправлен заголовок «close» на «pipe»
- «create» изменен на «creat»
- Изменен прототип функции sigismember
- Изменен прототип функции sigaction

28.08.2018 – добавлен раздел 8 (Работа с сетью)

© Shashliki team 2018