

# Операционные системы

25 марта 2019 г.

# Содержание

<b>1</b>	<b>Введение</b>	<b>3</b>
1.1	Преподаватель . . . . .	3
1.2	Операционные системы . . . . .	3
1.3	Ядро и прочее . . . . .	4
<b>2</b>	<b>Процессы</b>	<b>5</b>
2.1	Общее . . . . .	5
2.2	Sheduler . . . . .	6
2.3	API and ABI . . . . .	6
2.4	Модель памяти процесса . . . . .	7
2.5	Системные вызовы для работы с процессами . . . . .	7
2.6	PID и дерево процессов . . . . .	8
2.7	Calling convention . . . . .	8
2.8	Процесс и ОС . . . . .	10
2.9	Переключение контекста . . . . .	11
2.10	Литература . . . . .	11
2.11	Домашнее задание №1 . . . . .	12
<b>3</b>	<b>Файловые системы</b>	<b>13</b>
3.1	Носители . . . . .	13
3.1.1	HDD . . . . .	13
3.1.2	SSD . . . . .	14
3.1.3	Общее . . . . .	14
3.2	Быстродействие . . . . .	14
3.2.1	Интересные числа . . . . .	14
3.2.2	Выводы для HDD . . . . .	14
3.3	Structure packaging . . . . .	15
3.4	Алгоритмы элеватора . . . . .	15
3.5	Файл . . . . .	15
3.6	Директория . . . . .	16
3.6.1	Права — просто числа . . . . .	16
3.6.2	sticky bit . . . . .	17
3.7	Иерархия . . . . .	17
3.8	Монтирование . . . . .	17
3.9	Inode . . . . .	18
3.10	Атрибуты процесса . . . . .	18

3.11	Файловые системы . . . . .	18
3.12	Проход по пути . . . . .	18
3.13	Диски . . . . .	19
3.14	RAID . . . . .	19
3.15	Организация файловых систем . . . . .	21
3.16	Операции с файлами . . . . .	23
3.17	Системные вызовы . . . . .	23
3.18	Пару слов о типах . . . . .	23
3.19	Common pitfalls . . . . .	23
3.20	Литература . . . . .	23
3.21	Домашнее задание №2 . . . . .	23

# Лекция 1

## Введение

### 1.1 Преподаватель

Банщиков Дмитрий Игоревич

**email:** me@ubique.spb.ru

### 1.2 Операционные системы

- Операционная система — это уровень абстракции между пользователем и машиной. Цель курса в том, чтобы объяснить что происходит в системе от нажатия кнопки в браузере до получения результата.
- Курс будет посвящен Linux, потому что иначе говорить особо не о чем. Linux - это операционная система общего назначения, для машин от самых маленьких почти без ресурсов до мощнейших серверов. Простой ответ почему Linux настолько популярен, а не Windows — в некоторых случаях он бесплатный.
- Почему полезно разрушить абстракцию черного ящика? Чтобы писать более оптимизированный и функциональный код. Иногда встречаются проблемы которые не могут быть решены без знания внутренней работы ОС.

## 1.3 Ядро и прочее

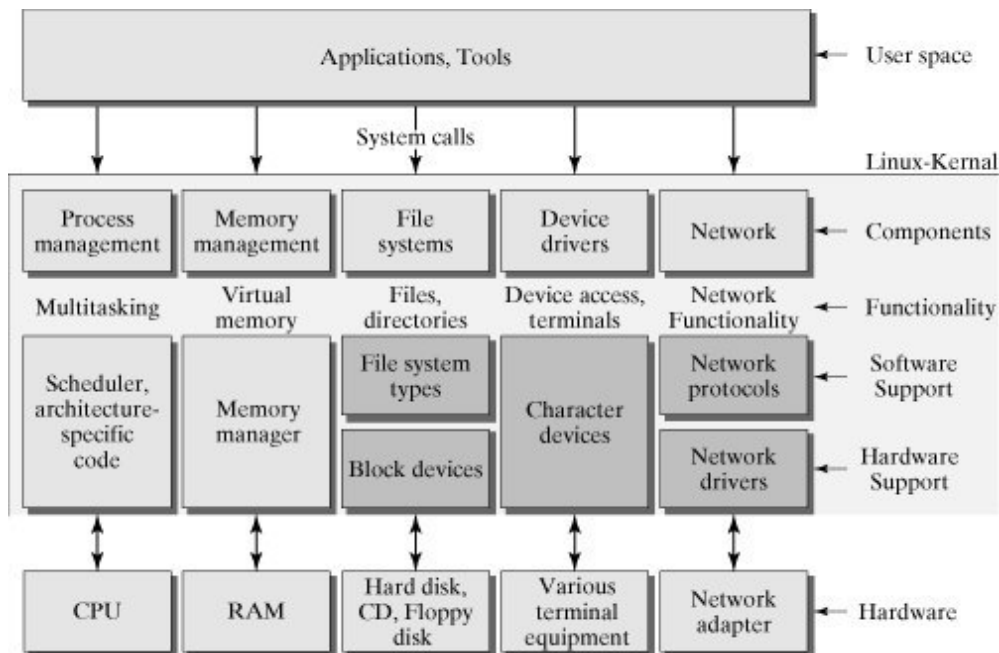


Рис. 1.1: Kernel scheme

- Ядро Linux (*kernel*) — монолитное, это оправдано для ядра, но уязвимость одной части ядра ставит в угрозу все остальные части.
- Микроядерные ОС - альтернатива монолитным (мы не будем их изучать), но с ними сложно работать, потому что протоколы общения между частями требуют ресурсов.
- *UNIX-like* системы - это системы предоставляющие похожий на *UNIX* интерфейс.

TODO Написать побольше

# Лекция 2

## Процессы

### 2.1 Общее

- Процесс — экземпляр запущенной программы. Процессы должны уметь договариваться чтобы сосуществовать, но в то же время не знать друг о друге и владеть монополией на ресурс машины.
- С точки зрения ОС процесс — это абстракция, позволяющая абстрагироваться от внутренностей процесса.
- С точки зрения программиста процесс — абстракция, которая позволяет думать что мы монополично владеем ресурсами машины.
- На момент выполнения процесс можно охарактеризовать полным состоянием его памяти и регистров. Чтобы приостановить процесс нам нужно просто сохранить его 'отпечаток', а чтобы возобновить нужно загрузить его память и регистры
- Батч-процессы (например, сборки или компиляции) не требуют отзывчивости пока жрут ресурсы.

Могут быть сформулированы следующие тезисы:

- Система не отличает между собой процессы
- Процессы в общем случае ничего не знают друг о друге
- Процесс - с одной стороны абстракция, которая позволяет не различать их между собой, с другой - конкретная структура
- Память и регистры - однозначно определяют процесс
- Способ выбора процесса - алгоритм *shedule*гивания
- Переключение с процесса на процесс - смена контекста процесса
- Контекст процесса - указатель на виртуальную память и значения регистров
- Как отличать процессы между собой - *pid*

## 2.2 Sheduler

### TODO Состояния процесса

- Заводит таймер для процесса(квант времени), после его истечения или когда процесс сам закончился выбирает другой процесс.
- Производительность - разбиение на несколько процессов
- Дизайн приложения

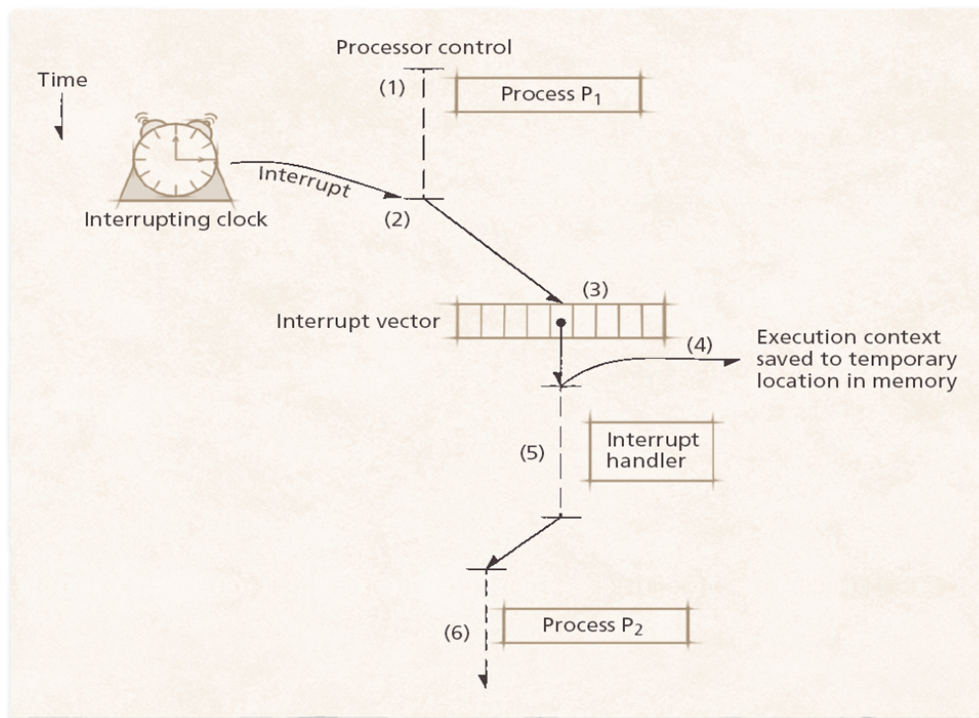
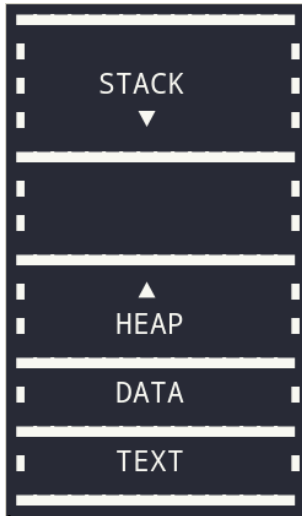


Рис. 2.1: Process-interruption

## 2.3 API and ABI

### TODO

## 2.4 Модель памяти процесса



Общие соображения:

- **stack** — выделяется неявно, **heap** — должны выделять сами (malloc, new и тп),
- секции — **data**, **text**
- **data** — статические, глобальные переменные, **text**
- **stack** растёт вниз, **heap** - вверх
- **frame** — область памяти стека, хранящая данные об адресах возврата, информацию о локальных переменных
- Резидентная память та, которая действительно есть

## 2.5 Системные вызовы для работы с процессами

- *fork()* — для того чтобы создать новый процесс

fork-example.c

```
void f() {  
    const pid_t pid = fork();  
  
    if (pid == -1) {  
        // handle error  
    }  
    if (!pid) {  
        // we are child  
    }  
    if (pid) {  
        // we are parent  
    }  
}
```

*fork*-бомба — **TODO**

- *wait(pid)* — ждем процесс
- *exit()* — завершаемся
- *execve()* — запустить программу



```

int main(int argc, char *argv[]) {
    char *newargv[] = { NULL, "hello", "world", NULL };
    char *newenviron[] = { NULL };
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file-to-exec>\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    newargv[0] = argv[1];
    execve(argv[1], newargv, newenviron);
    perror("execve"); /* execve() returns only on error */
    exit(EXIT_FAILURE);
}

```

- *SIGKILL* — принудительное завершение другого процесса ( **\$ kill** ) **TODO Подробнее**

## 2.6 PID и дерево процессов

- У каждого *PID* есть *parentPID* (*PPID*)
- **\$ ps** — позволяет посмотреть специфичные атрибуты процесса
- Процесс *init(pid 0)* создается ядром и выступает родителем для большинства процессов, созданных в системе
- Можно построить дерево процессов ( **\$ pstree** )

Процесс делает *fork()*. Возможны 2 случая:

1. Процесс не делает *wait(childpid)*

Зомби-процесс (*zombie*) — когда дочерний процесс завершается быстрее, чем вы сделаете *wait*

2. Процесс завершается, что происходит с дочерним процессом?

Сирота (*orphan*) — процесс, у которого умер родитель. Ему назначается родителем процесс с *pid 1*, который время от времени делает *wait()* и освобождается от детей

*PID* - переиспользуемая вещь (таблица процессов)

## 2.7 Calling convention

**\$ man syscall** - как вызываются *syscall*

### syscall.h

```
#ifndef SYSCALL_H
#define SYSCALL_H

void IFMO_syscall();

#endif
```

### syscall.s

```
.data

.text
.global IFMO_syscall

IFMO_syscall:
    movq $1, %rax
    movq $1, %rdi
    movq $0, %rsi
    movq $555, %rdx
    syscall
    ret
```

### syscall-example.c

```
#include "syscall.h"

int main() {
    IFMO_syscall();
}
```

Что здесь происходит?

1. Вызываем `write()`
2. Просим ядро записать 555 байт начинающихся по адресу 0 в файловый дескриптор №1 (*stdout* — №1, *stdin* — №2, *stderr* — №3)
3. Ничего не происходит, так как:  
`write(1, NULL, 555)` возвращает -1 (*EFAULT* - Bad address)

Как со всем этим работать?

- **\$ strace** — трассировка процесса (подсматриваем за процессом, последовательность *syscall* с аргументами и кодами возврата)

Если *syscall* ничего не возвращает, то в выводе пишется ? вместо возвращаемого значения

- **\$ man errno** — ошибки

Если делаем *fork()* — проверяем код возврата (хорошая практика)

*char\* strerror(int errnum)* - возвращает строковое описание кода ошибки

Почему *char\**, а не *const char\**? Потому что всем было лень.

*thread\_local* — решение проблемы: переменная с ошибкой - общая для каждого потока

- До *main()* и прочего (конструкторы) происходит куча всего (*munmap*, *mprotect*, *mmap*, *access*) - размещение процесса в памяти и т.д.
- Программа не всегда завершается по языковым гарантиям (деструкторы)
- **\$ ptrace** — позволяет одному процессу следить за другим (используется, например, в *GDB*)
- *ERRNO* — переменная с номером последней ошибки, *strerror*
- *finalizers*, библиотечный вызов *exit*

## 2.8 Процесс и ОС

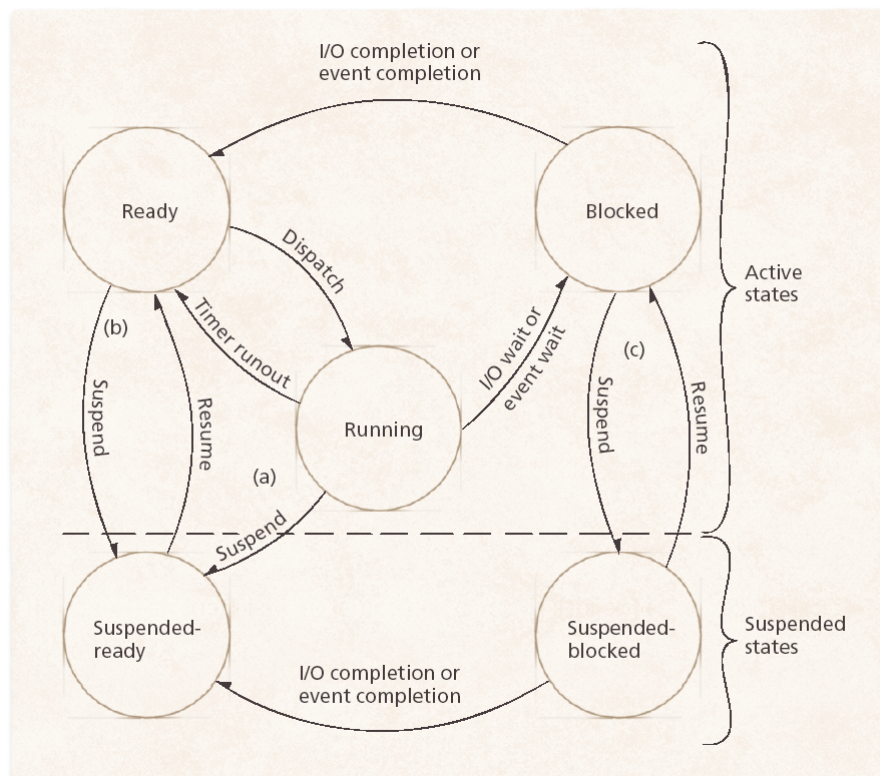


Рис. 2.2: Диаграмма времени жизни процесса и взаимодействия с ОС

## 2.9 Переключение контекста

Шедюлер ОС раскидывает процессы и создает иллюзию одновременного выполнения

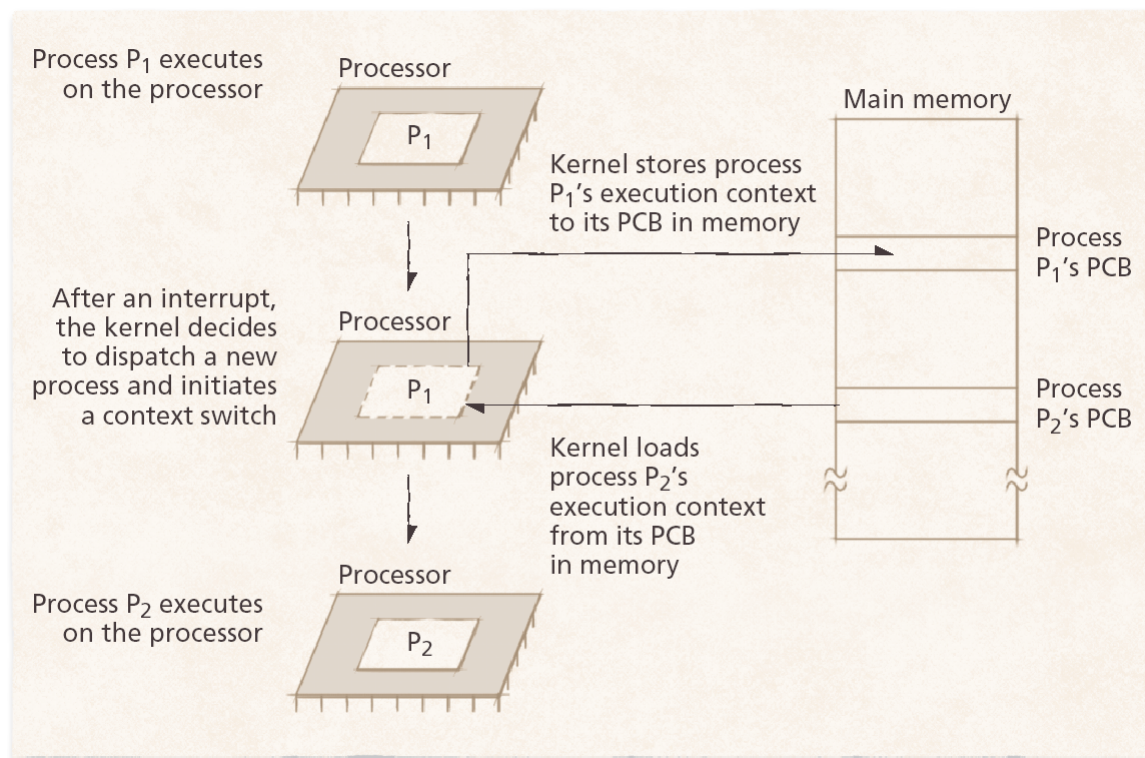


Рис. 2.3: Иллюзия многозадачности

## 2.10 Литература

- Windows Internals by Mark Russinovich
- Операционная система UNIX. Андрей Робачевский
- Unix и Linux. Руководство системного администратора. Эви Немец.

## 2.11 Домашнее задание №1

Необходимо создать игрушечный интерпретатор.

Цель — получить представление о том, как работают командные интерпретаторы.

- Программа должна в бесконечном цикле считывать с *stdin* полный путь к исполняемому файлу, который необходимо запустить и аргументы запуска. Дождавшись завершения процесса необходимо вывести на *stdout* код его завершения.
- Необходимо использовать прямые системные вызовы для порождения новых процессов, запуска новых исполняемых файлов и получения статуса завершения системного вызова.
- Все возвращаемые значения системных вызовов должны быть проверены и в случае обнаружения ошибок необходимо выводить текстовое описание ошибки.
- На входе могут быть некорректные данные.
- Дополнительные баллы - поддержка переменных окружения.
- Язык имплементации - C или C++.

TODO Добавить еще одну картинку из *images*

TODO Секция Контекст процесса

TODO Секция Системные процессы

# Лекция 3

## Файловые системы

### 3.1 Носители

#### 3.1.1 HDD

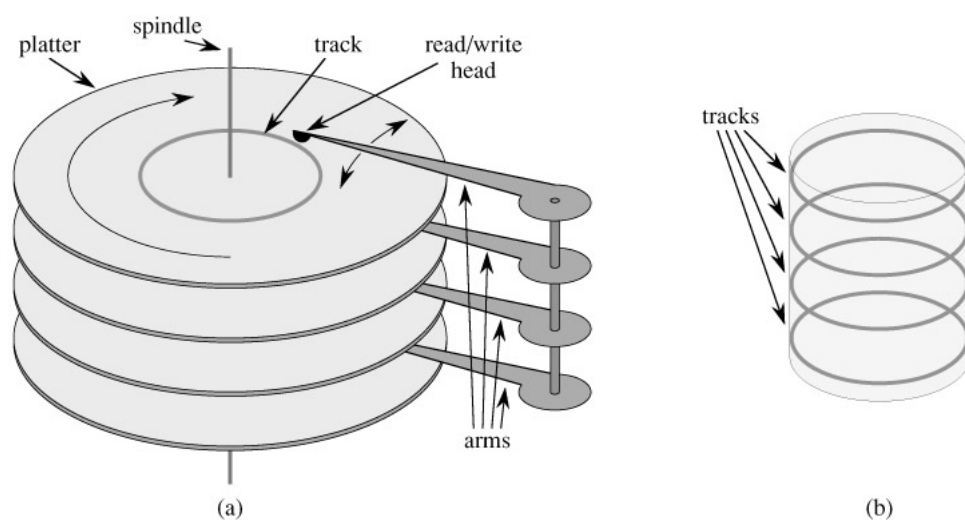


Рис. 3.1: Жесткий диск

- Обороты в минуту( $O$ ) — 5400, 7200, 10000, ...
- $\frac{1}{2*O}$  — минимальное время доступа (случайное чтение)
- В мире Unix не существует дефрагментации (ОС должна сама заботиться)
- Время отказа (**MTBF** — min time before failure) — условное количество циклов наработки до отказа
- На server — сутки, desktop — часы (разница в 3 раза примерно, если одно и то же число циклов)
- Плюсы: стоимость, объем
- Минусы: время доступа, надежность

### 3.1.2 SSD

- **SATA** и **NVME** — протоколы для дисков
- **NVME** — новомодная штука для **SSD**
- Плюсы: время доступа
- Минусы: надежность, стоимость, объем

### 3.1.3 Общее

- **EOPS** — **TODO**
- **seek** — рандомное чтение (512 байт)
- Минимум информации: сектор — 512 байт -> 4096 байт
- Чтение одного байта равносильно чтению всего сектора с этим байтом
- Запись одного байта — считать один сектор, заменить байт и записать один сектор
- Аналогия — процессор-память — **cacheline**  
(кэшируется линиями, а на диск записывается и считывается секторами)

## 3.2 Быстродействие

### 3.2.1 Интересные числа

Числа, которые должен знать каждый программист

Cycle	1 ns
Main memory reference	100 ns
Read 4K randomly from SSD	150 us
Read 1 MB sequentially from SSD	1 ms
Disk seek	10 ms
Read 1 MB sequentially from disk	20 ms

### 3.2.2 Выводы для HDD

- Читать нужно последовательно
- Обращения к диску следует минимизировать
- Стоимость доступа сильно дороже передачи данных

### 3.3 Structure packaging

Сколько будет занимать памяти следующая структура?

hole1.c

```
struct hole {  
    uint64_t a;  
    uint32_t b;  
    uint64_t c;  
    uint32_t d;  
}
```

Ответ: 32 байта, так как *b* и *d* будут выравнены по MAX\_ALIGNMENT  
Очевидное решение проблемы:

hole2.c

```
struct hole {  
    uint64_t a;  
    uint32_t b;  
    uint32_t d;  
    uint64_t c;  
}
```

Данная структура будет занимать 24 байта на x86\_64.

### 3.4 Алгоритмы элеватора

[Ссылка на презентацию](#)

1. SLIDE 6

Алгоритмы элеватора обрабатывают последовательности запросов к диску (перепорядочивают их)

2. SLIDE 7

**FCFS** (FIFO) — самый простой и медленный

3. SLIDE 8-9

**SSTF** (Shortest Seek Time First)— сортировка (очередной запрос определяется наименьшим временем seek)

4. SLIDE 10 - ...

Различные способы упорядочивания(**SCAN**)

### 3.5 Файл

- Абстракция для данных (для Kernelspace)



- Последовательность байтов (для Userspace)
- Формат не определен
- **Unix** — все есть файл (абстракция-интерфейс внутри ядра)
- Типы файлов
  - regular
  - directory
  - symlink
  - socket, fifo
  - character device, block device

## 3.6 Директория

- Содержит имена находящихся в ней файлов
- `.` — ссылка на текущую
- `..` — ссылка на родителя
- `$ cd` — сменить директорию
- `$ pwd` — текущая директория
- `$ ls` — формирование дерева
- `$ find` — поиск
- *filename vs pathname*: `$ realpath`

### 3.6.1 Права — просто числа

- `$ view /etc/passwd`
- `$ view /etc/group`
- `$ id` - показывает идентификаторы того, кто ее вызывал
- `$ execute` — search
- `$ read` — directory listing
- `$ write` — changing directory
- Темные директории (переход в директорию внутри директории, для который ты не можешь посмотреть все файлы)
- Права `rwX` (read, write, execute)

- **\$ chmod** — меняет права доступа  
**\$ chmod 123** — 1 - user, 2 - group, 3 - other
- У процесса есть информация о том, кто его запустил
- SGID — **TODO**

### 3.6.2 sticky bit

- Изменение поведения при создании нового файла
- /tmp
- Создаешь директорию со *sticky bit* и все, кто создают файлы в этой директории имеют на них права

## 3.7 Иерархия

**TODO** root.png

- /
  - bin/
  - dev/
  - etc/
  - sbin/
  - home/
  - var/
  - usr/
    - \* bin/
    - \* sbin/
  - tmp

## 3.8 Монтирование

- Есть корень и есть узлы, в которые можно монтировать другие файловые системы (часть из них виртуальная)
- **\$ mount**
- Для / обычно используется **ext4** (использует журналирование)
- Для /boot может использоваться **ext2** — так как это более проверено временем (на Ubuntu)
- Файловая система для узла — это не константа, ее можно менять
- **\$ df -h** , **\$ du -hs**

## 3.9 Inode

TODO More from presentation

- Директория задает mapping имени файла в его inode
- **\$ ln**
- Hardlink — существует в рамках одной файловой системы
- Softlink(symlink) — бит 1
- **\$ stat** — информация о файле
- *atime* — время последнего доступа
- *ctime* — изменение мета-информации
- *mtime* — изменение содержимого файла

## 3.10 Атрибуты процесса

TODO

## 3.11 Файловые системы

TODO

## 3.12 Проход по пути

- Рекурсивный процесс (увеличиваем индекс при проходе в глубину)
- Количество seek по диску зависит от длины пути
- `namei` (name-innode) — lru-cache (файл  $\leftrightarrow$  номер inode)

### 3.13 Диски

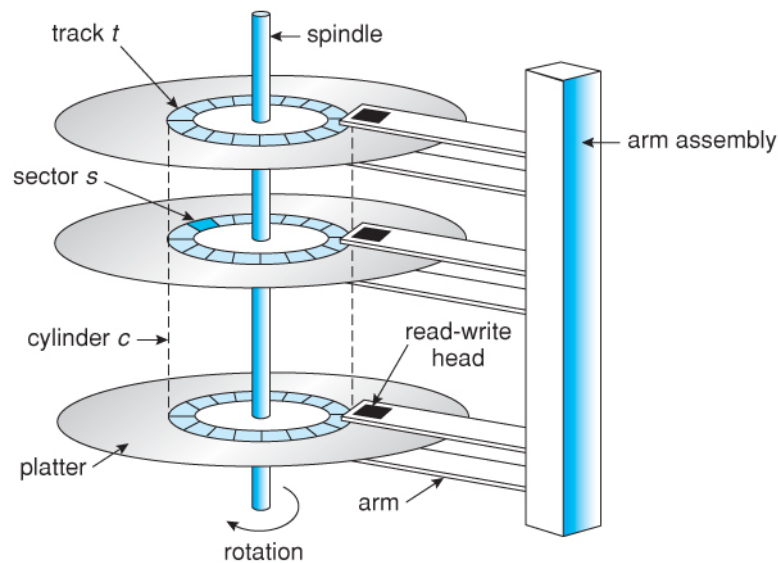


Рис. 3.2: Устройство диска

#### TODO Информация из презентации

- На внешних цилиндрах больше секторов, чем на внутренних => чем ближе к центру тем меньше скорость нужна (CD)
- На жестких дисках — постоянная угловая скорость (в центре больше плотность)
- *Partitioning* — разделение диска на несколько логических частей (партиции, на каждой своя файловая система)
- Существует другой подход - "собственная" файловая система на "сыром" диске (MySQL)
- Современный контроллер жесткого диска может находить механически поврежденные блоки (bad blocks) и делать remap их на некоторые запасные (sector sparing: replace bad sectors with spare)
- **\$ man 1 badblocks**
- Bootblock — **TODO**

### 3.14 RAID

Redundant Arrays of Independent Disks (Избыточный массив независимых дисков)

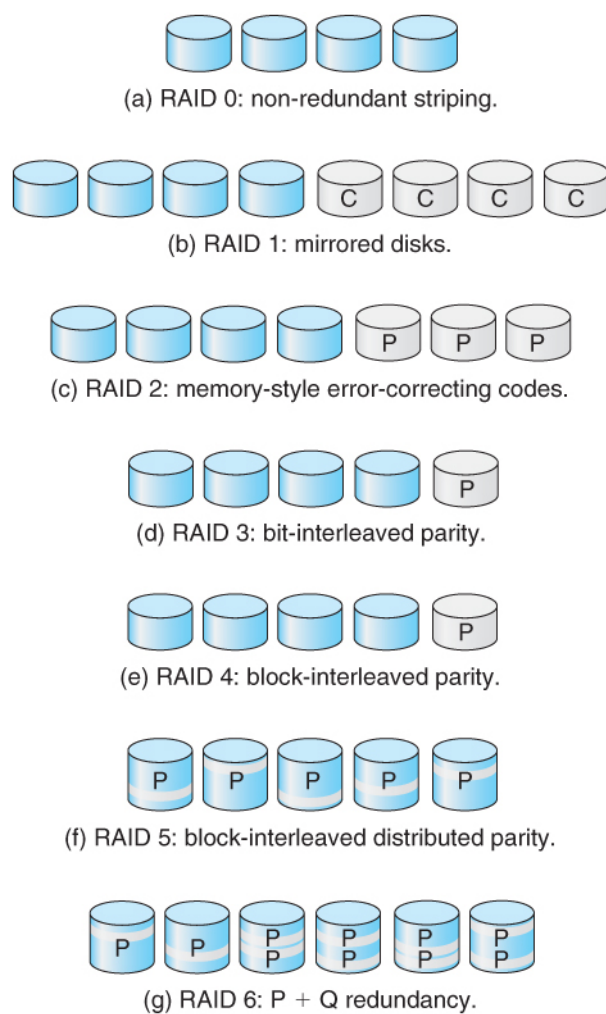


Рис. 3.3: Уровни RAID

- Reliability (надежность, hacks for more long time of complex usage)
- Performance (striping, суммирование EOPS)
- Levels:
  - 0 — pure striping (1 блок на 1 диске, 2 блок на 2 диске и т.д. — один диск вышел из строя — fail)
  - 1 — pure mirroring (пара дисков, данные продублированы)
  - 0 + 1, 1 + 0
  - 2, 3, 4, 5 — используются не так часто (хранение доп. данных)
- Rebuild — падает производительность
- Hardware RAID — проблемы: "залоченность" на производителе (vendor lock in), драйвера, как правило, не очень

- Software RAID — гипотетически медленно, но на практике нужная производительность достигается
- У аппаратных RAID — есть батарея, которая "улучшает" производительность (сначала на батарею, потом на диск, когда будет удобно)

### 3.15 Организация файловых систем

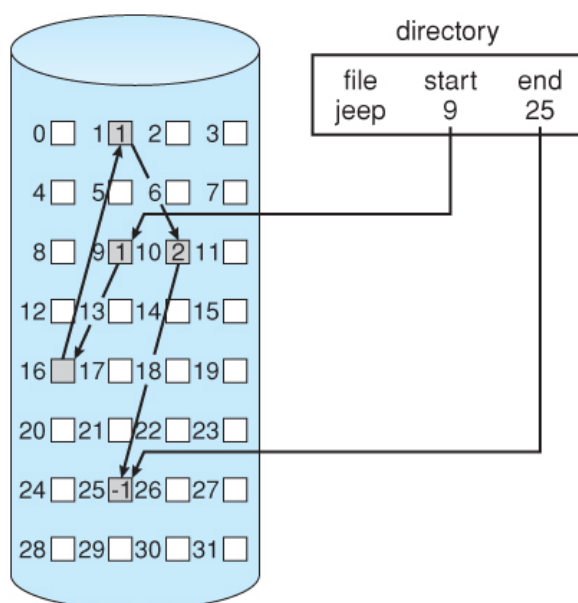
Структура директорий: связный список и хэш-таблица

smart ( `$ smartctl` ) — оценка диска на практике

Свободные сектора

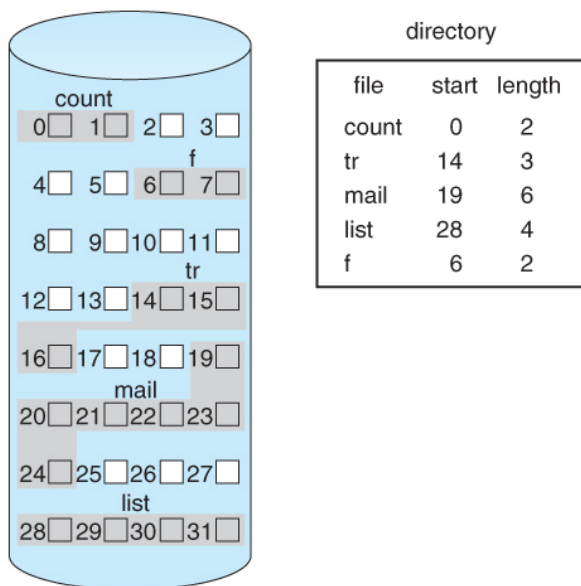
1. Bit Vector — fast, space usage
2. Список

Выделение памяти



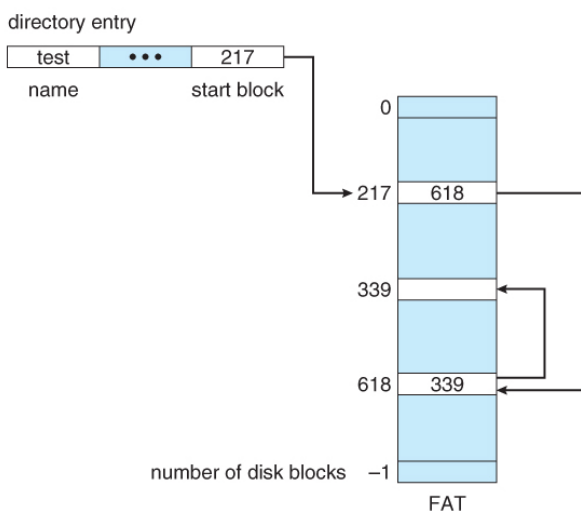
Линейное

- Объект задается началом и концом (здесь возникают проблемы внешней и внутренней фрагментации)
- Performance: +sequential, +random



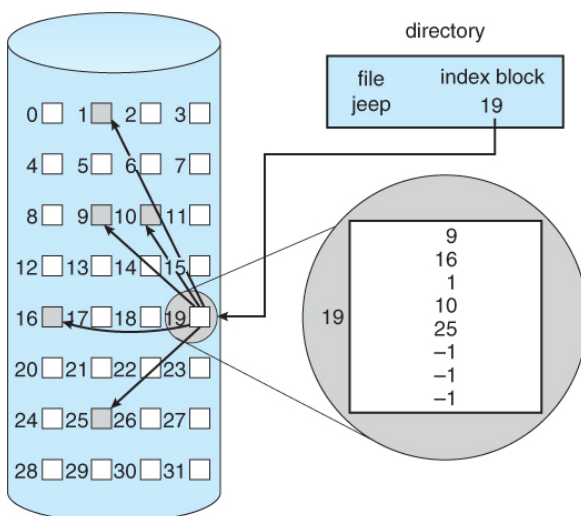
## Список

- Performance: -sequential, -random
- Надежность: -
- Решает проблему внешней фрагментации
- В каждом "блоке" указатель на следующий



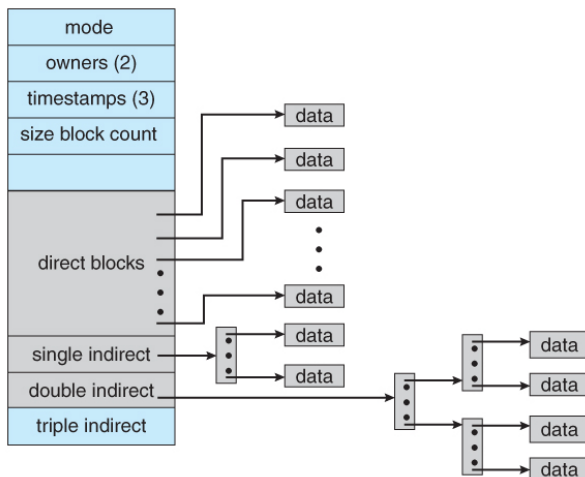
## FAT

- Все ссылки хранятся в начале диска — их можно эффективно кэшировать



## Индексированная

- Отдельный блок для ссылок на данные
- Внутренняя фрагментация



## UNIX

- Комбинированная
- Косвенная многоуровневая адресация

### 3.16 Операции с файлами

TODO 3 картинки + презентация

### 3.17 Системные вызовы

### 3.18 Пару слов о типах

### 3.19 Common pitfalls

### 3.20 Литература

- The Unix Programming Environment. Brian W. Kernighan, Rob Pike
- Advanced Programming in the Unix Environment. W. Richard Stevens

### 3.21 Домашнее задание №2