

# Операционные системы

2 апреля 2019 г.

# Содержание

<b>1 Введение</b>	<b>4</b>
1.1 Преподаватель . . . . .	4
1.2 Операционные системы . . . . .	4
1.3 Ядро и прочее . . . . .	5
<b>2 Процессы</b>	<b>6</b>
2.1 Общее . . . . .	6
2.2 Модель памяти процесса . . . . .	7
2.3 Системные вызовы для работы с процессами . . . . .	7
2.4 PID и дерево процессов . . . . .	8
2.5 Calling convention . . . . .	9
2.6 API and ABI . . . . .	10
2.7 Процесс и ОС . . . . .	10
2.7.1 Scheduler . . . . .	10
2.7.2 Interruption . . . . .	11
2.7.3 Состояния процесса . . . . .	11
2.7.4 Переключение контекста . . . . .	11
2.8 Контекст процесса . . . . .	12
2.9 Системные процессы . . . . .	12
2.10 Литература . . . . .	12
2.11 Домашнее задание №1 . . . . .	13
<b>3 Файловые системы</b>	<b>14</b>
3.1 Носители . . . . .	14
3.1.1 HDD . . . . .	14
3.1.2 SSD . . . . .	15
3.1.3 Общее . . . . .	15
3.2 Быстродействие . . . . .	15
3.2.1 Интересные числа . . . . .	15
3.2.2 Выводы для HDD . . . . .	15
3.3 Structure packaging . . . . .	16
3.4 Алгоритмы элеватора . . . . .	16
3.5 Файл . . . . .	17
3.6 Директория . . . . .	17
3.6.1 Права — просто числа . . . . .	17
3.6.2 sticky bit . . . . .	18

3.7	Иерархия . . . . .	18
3.8	Монтирование . . . . .	19
3.9	Inode . . . . .	19
3.10	Проход по пути . . . . .	19
3.11	Атрибуты процесса . . . . .	20
3.12	Диски . . . . .	21
3.13	RAID . . . . .	22
3.14	Организация файловых систем . . . . .	23
3.15	Файловые системы . . . . .	25
3.16	Операции с файлами . . . . .	25
3.17	Системные вызовы . . . . .	26
3.18	Пару слов о типах . . . . .	27
3.19	Common pitfalls . . . . .	27
3.20	Литература . . . . .	27
3.21	Домашнее задание №2 . . . . .	27
<b>4</b>	<b>Виртуальная память</b>	<b>29</b>
4.1	Прерывания и исключения . . . . .	29
4.2	Память . . . . .	30
4.3	Подходы к организации памяти . . . . .	31
4.3.1	Досегментная организация(№1) . . . . .	31
4.3.2	Сегментная организация(№2) . . . . .	31
4.3.3	Страницчная организация(№3) . . . . .	31
4.3.4	Страницчная организация в x86(Реальность) . . . . .	31
4.4	MMU . . . . .	31
4.5	Переключение контекста . . . . .	31
4.6	Примеры использования . . . . .	31
4.6.1	Подкачка по требованию . . . . .	31
4.6.2	Copy on write . . . . .	31
4.6.3	Swap . . . . .	32
4.6.4	Другие . . . . .	33
4.7	Выделение памяти . . . . .	33
4.7.1	Пример №1 . . . . .	33
4.7.2	Syscalls . . . . .	33
4.7.3	Helpers . . . . .	34
4.7.4	Пример №2 . . . . .	34
4.8	Mapping . . . . .	35
4.8.1	Syscalls . . . . .	35
4.8.2	Пример №1 . . . . .	35
4.8.3	Пример №2 . . . . .	36
4.8.4	Additional . . . . .	37
4.8.5	Пример №3 . . . . .	37
4.9	Аллокаторы памяти . . . . .	38
4.10	Безопасность . . . . .	39
4.10.1	Meltdown . . . . .	39
4.10.2	ASLR . . . . .	39

4.11 Page Reclaiming . . . . .	39
4.12 Page Fault . . . . .	40
4.13 Литература . . . . .	40
4.14 Домашнее задание №3 . . . . .	40

# 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

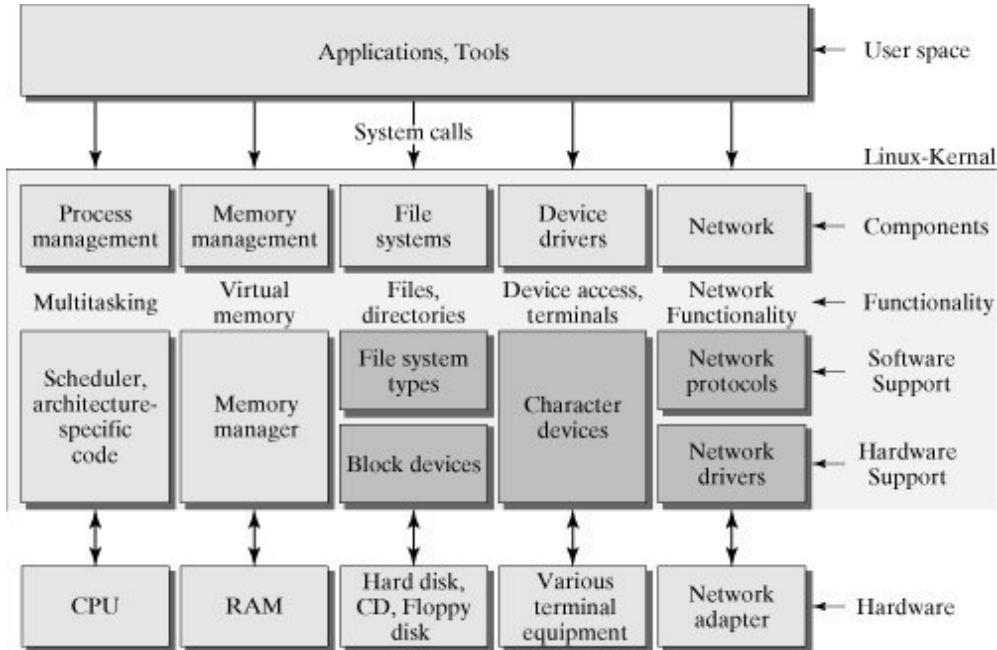


Рис. 1.1: Схема Kernelspace и Userspace

- Ядро Linux (*kernel*) — монолитное, это оправдано для ядра, но уязвимость одной части ядра ставит под 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.

# 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 процеса.
- С точки зрения программиста процесс — абстракция, которая позволяет думать что мы монопольно 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 shedulerивания
- 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 Модель памяти процесса

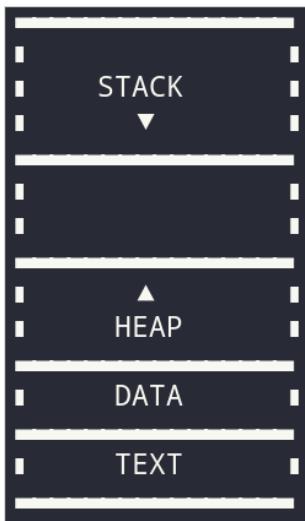


Рис. 2.1: Memory in process

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

## 2.3 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 — каждый дочерний процесс делает *fork()* и так далее

- *wait(pid)* — ждем процесс
- *exit()* — завершаемся

- `execve()` — запустить программу

execve-example.c

```
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);
}
```

- `kill()` — послать сигнал процессу
- `SIGKILL` — принудительное завершение другого процесса ( `\$ kill` )

## 2.4 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)

## 2.5 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 (*tmpmap*, *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*

## 2.6 API and ABI

- **API** (Application Program Interface)

Интерфейс коммуникации на уровне исходного кода (include, functions, etc)

- **ABI** (Application Binary Interface)

Интерфейс коммуникации на бинарном уровне (как параметры передаются функциям, кто очищает параметры функции и т.д.)

## 2.7 Процесс и ОС

### 2.7.1 Scheduler

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

## 2.7.2 Interruption

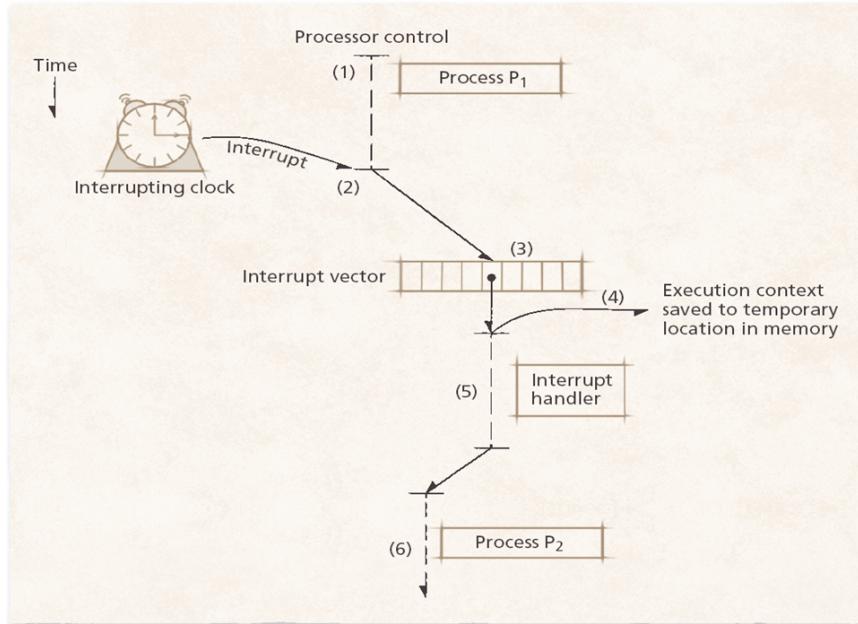
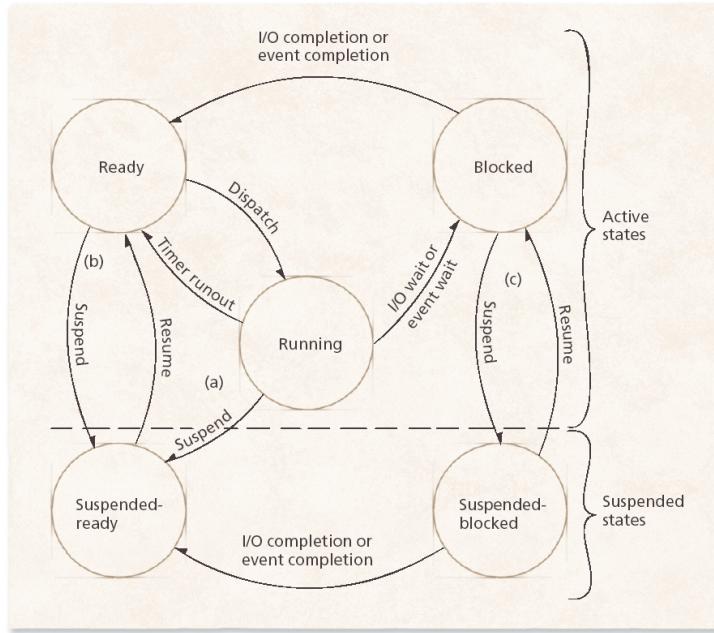


Рис. 2.2: Interruption of process

## 2.7.3 Состояния процесса



Процесс может быть в одном из следующих состояний

- **runnable**
- **running**
- sleeping: **interruptible**
- sleeping: **uninterruptible**
- **zombie**

Рис. 2.3: Диаграмма жизни процесса

## 2.7.4 Переключение контекста

Шедулер ОС раскидывает процессы и создает иллюзию одновременного выполнения

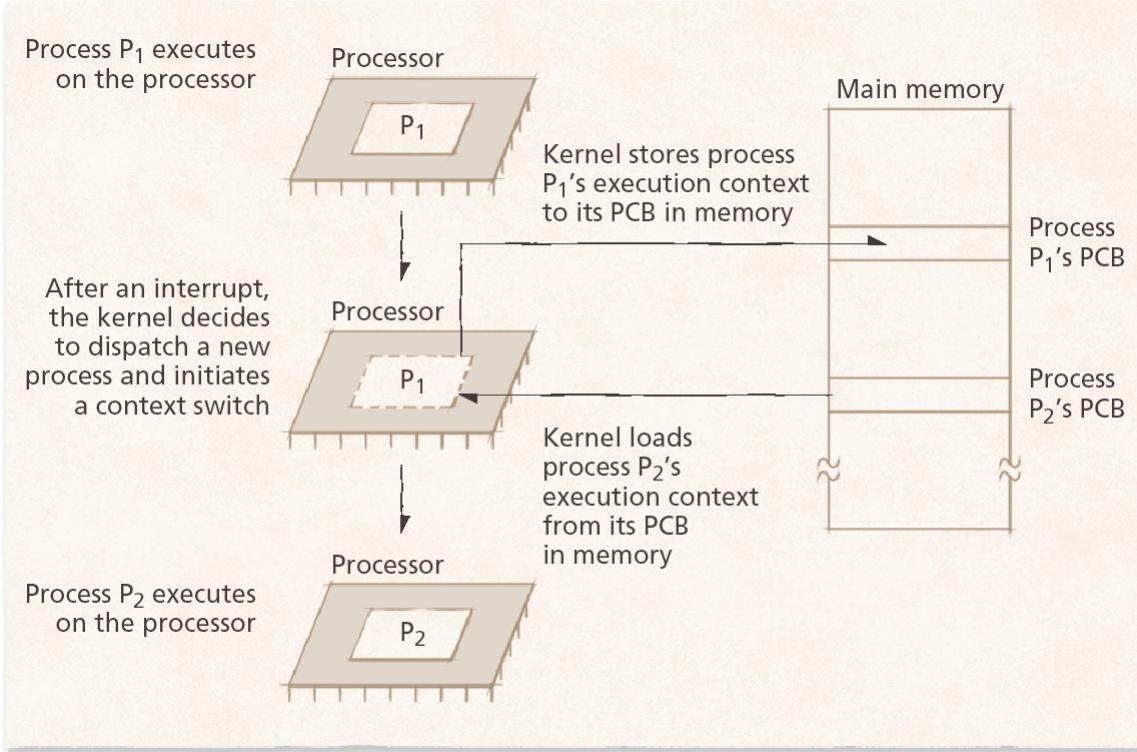


Рис. 2.4: Иллюзия многозадачности

## 2.8 Контекст процесса

TODO

## 2.9 Системные процессы

TODO What is this?

- kthreadd
- kswapd
- ksoftirqd

## 2.10 Литература

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

## 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 - С или С++.

# Лекция 3

## Файловые системы

### 3.1 Носители

#### 3.1.1 HDD

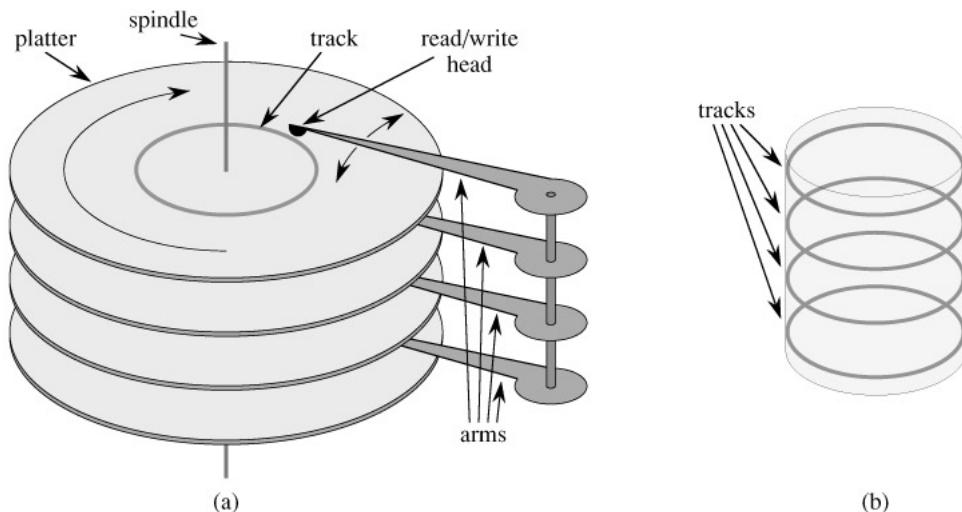


Рис. 3.1: Жесткий диск

- 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 наработка до отказа
- На **server** — сутки, **desktop** — часы (разница примерно в 3 раза, 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

- **IOPS** — input/output operations per second  
Показатель применяется для сравнения, например, какого-нибудь **HDD** с **SSD**
- **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 (переупорядочивают 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 которой 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** (Set Group ID up on execution)  
Специальный тип прав, который временно выдается запускающему (у него теперь права группы на файл/директорию)

### 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

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

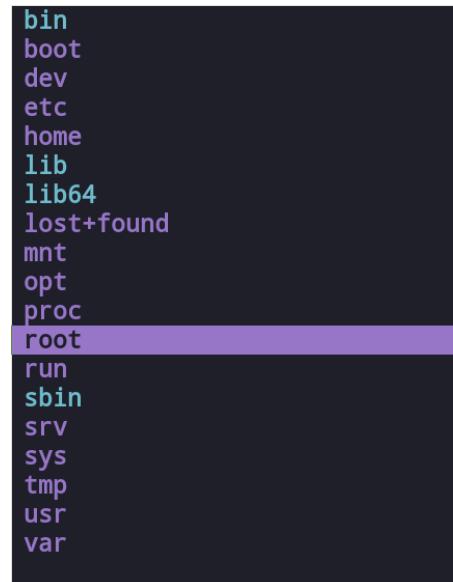


Рис. 3.2: Типичный вид корня в Linux

## 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

- Директория задает mapping имени файла в его inode
- **\$ ln**
- Hardlink — существует в рамках одной файловой системы
- Softlink(symlink) — text string
- **\$ stat** — информация о файле
- *atime* — время последнего доступа
- *ctime* — изменение мета-информации
- *mtime* — изменение содержимого файла
- inode корневой файловой системы фиксирован — 2

## 3.10 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 <-> номер inode)

## 3.11 Атрибуты процесса

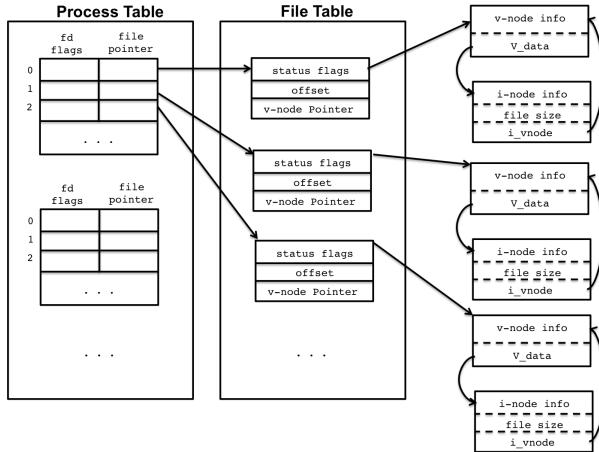


Рис. 3.3: Structures of Kernel

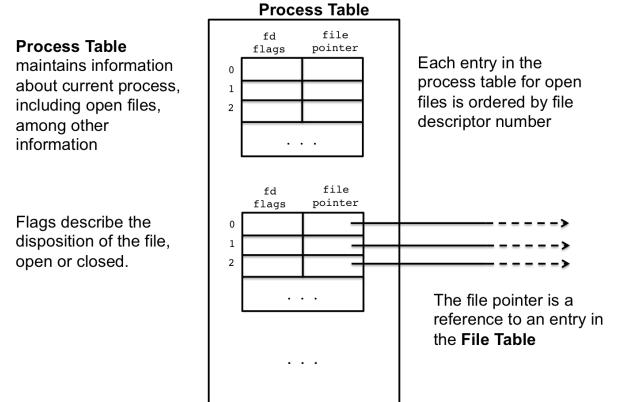


Рис. 3.4: Processes

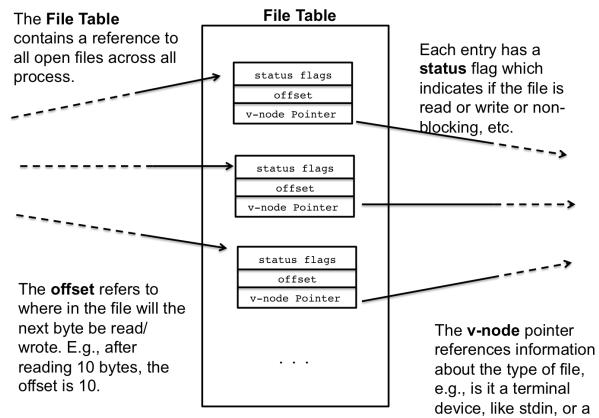


Рис. 3.5: Files

TODo ???

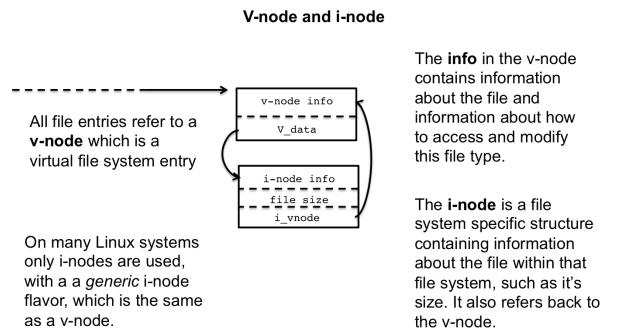
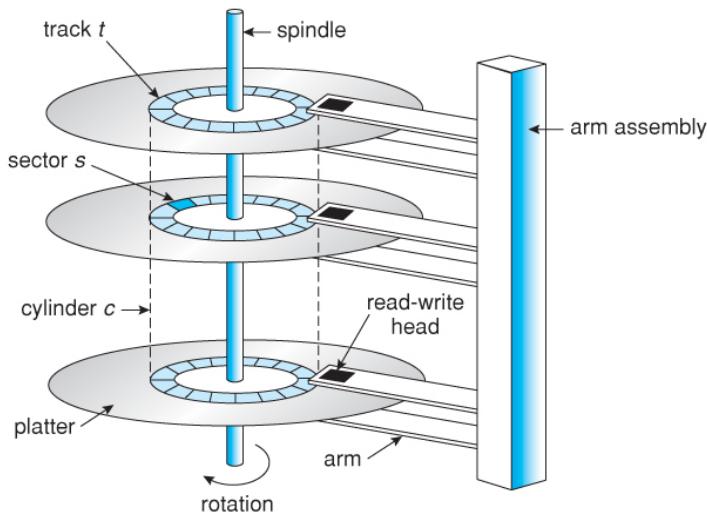


Рис. 3.6: Vnode and Inode

## 3.12 Диски



Устройство диска

- сектор
  - header: метаданные для контроллера диска
  - данные
  - trailer: ECC
- цилиндр
- пластина
- трэк
- шпиндель

Рис. 3.7: Устройство диска

- При записи данных на диск в сектора считаем и записываем **ECC**, при чтении считаем и затем сверяем (пытаемся исправить, если не соплось)
- **CLV** – Constant Linear Velocity(**CDROM**)
- **CAV** – Constant Angular Velocity(**HDD**)
- 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), они трактуются как "отдельные" диски
- 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 — bootstrap program at fixed location
- **MBR** — master boot record — boot code + partition table

### 3.13 RAID

Redundant Arrays of Independent Disks (Избыточный массив независимых дисков)

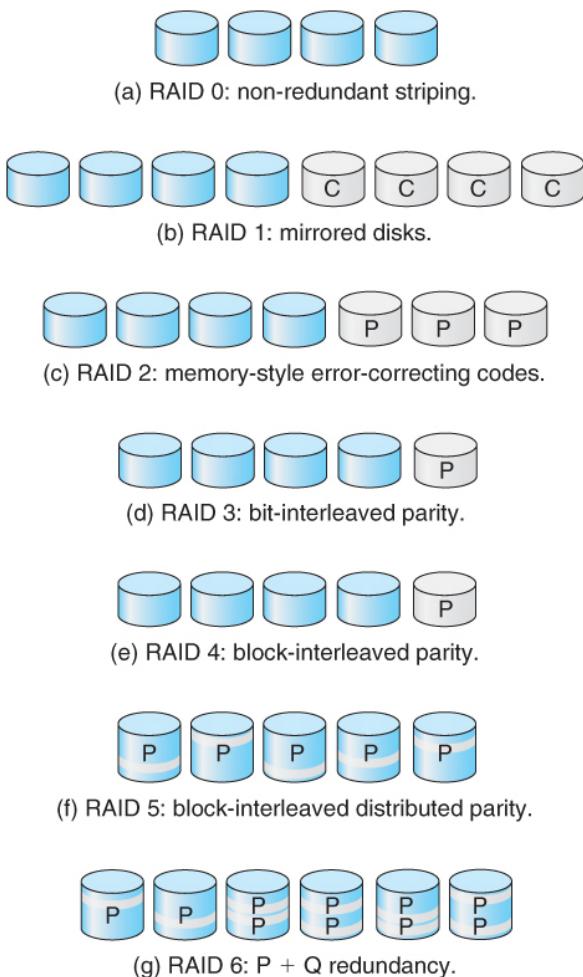


Рис. 3.8: RAID levels

- Reliability (надежность, hacks for more long time of complex usage)
- Performance (striping, суммирование IOPS)
- 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 батарейка, которая "улучшает" производительность (сначала на батарейку, потом на диск, когда будет удобно)
- TODO Байка про SpaceWeb

## 3.14 Организация файловых систем

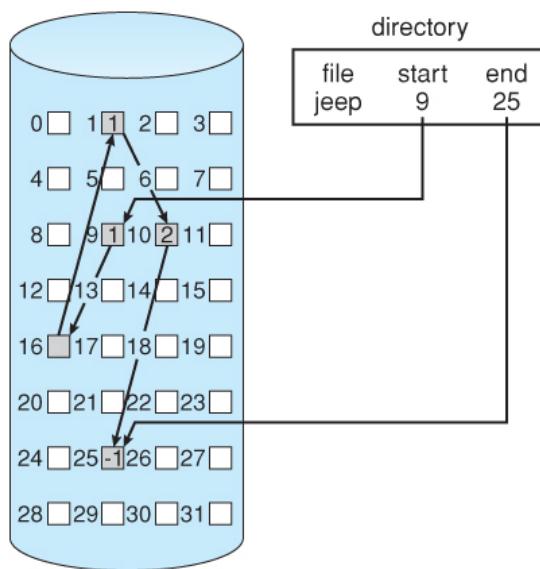
Структура директорий: связный список или хэш-таблица

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

Свободные сектора

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

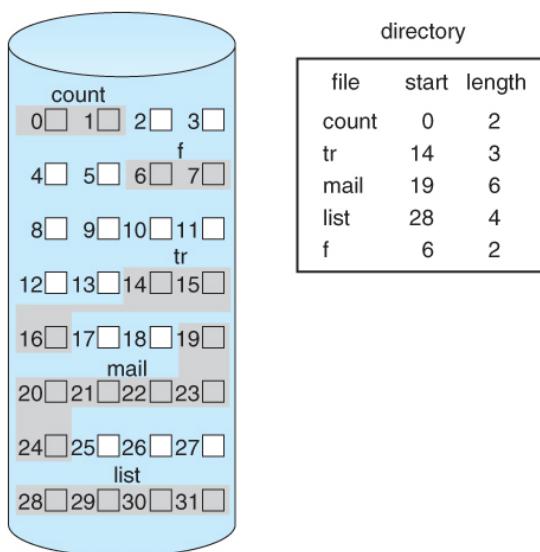
**Выделение памяти (allocation)**



### Линейное

- Объект задается началом и концом (здесь возникают проблемы внешней и внутренней фрагментации)
- Линейное чтение, меньше обращений
- Performance: sequential, random

Рис. 3.9: Linked allocation



### Список

- В каждом "блоке" указатель на следующий
- Плюсы: решает проблему внешней фрагментации
- Минусы: надежность, прыгаем по памяти
- Performance: sequential, awful random

Рис. 3.10: Contiguous allocation

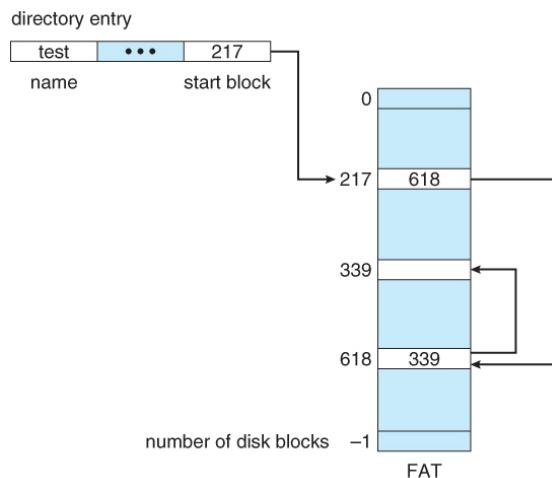


Рис. 3.11: FAT allocation

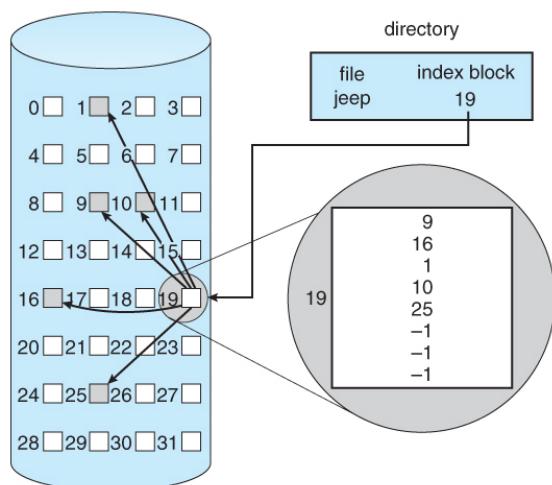


Рис. 3.12: Indexed allocation

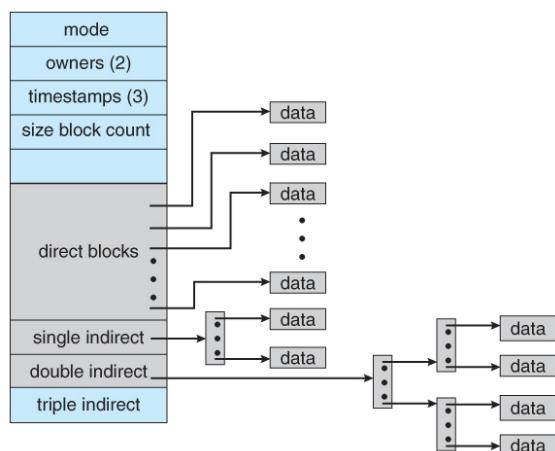


Рис. 3.13: UNIX allocation

## FAT

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

## Индексированное

- Отдельный блок для ссылок на данные
- Внутренняя фрагментация

## UNIX

- Комбинированная
- Косвенная многоуровневая адресация

### 3.15 Файловые системы

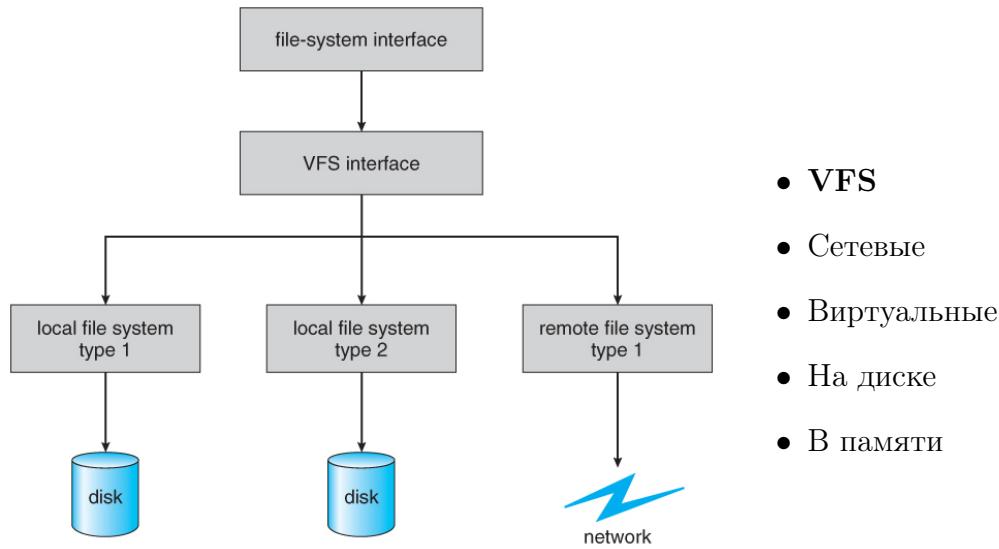


Рис. 3.14: VFS

### 3.16 Операции с файлами

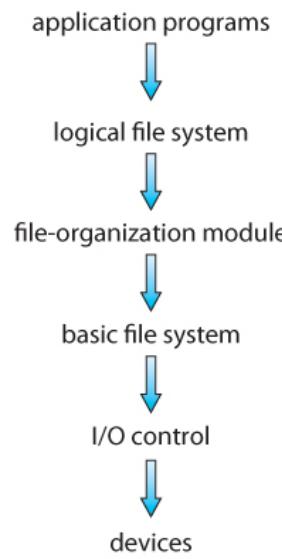


Рис. 3.15: Layered

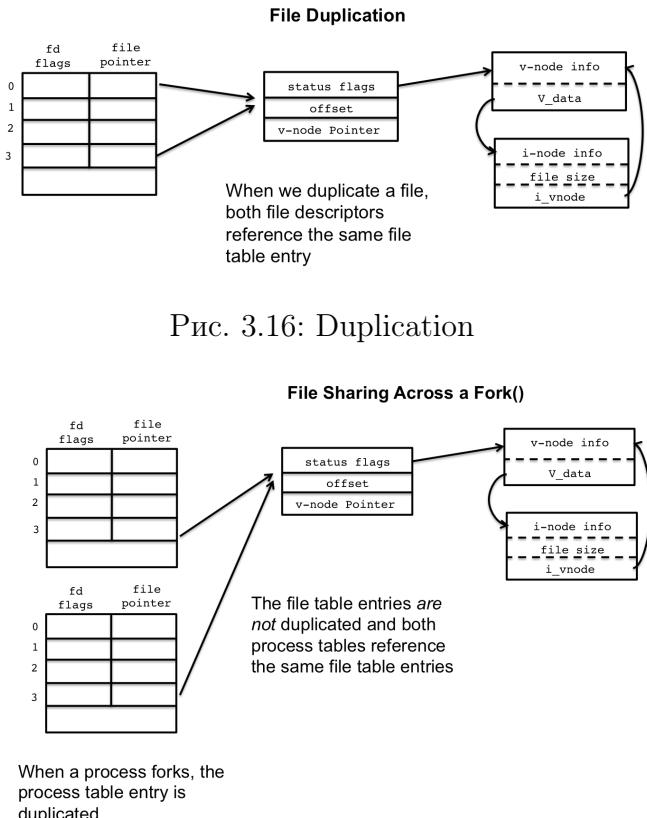


Рис. 3.16: Duplication

Рис. 3.17: Sharing

### 3.17 Системные вызовы

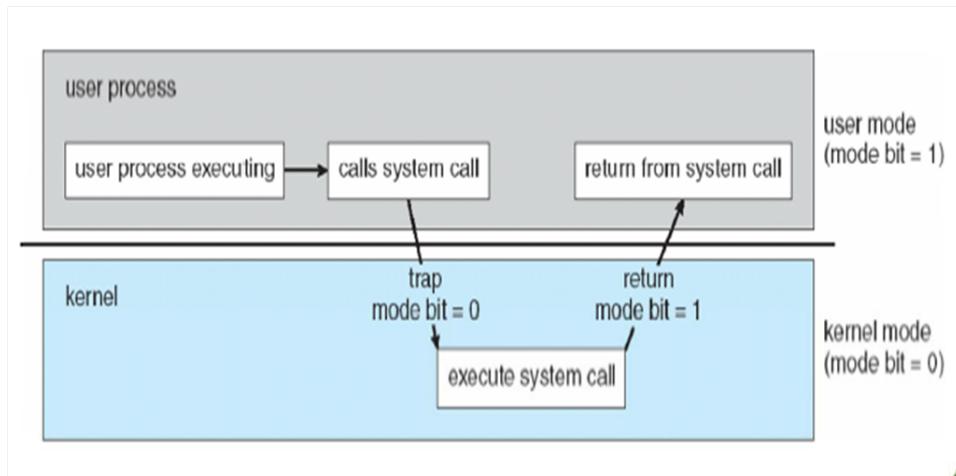


Рис. 3.18: Systemcall

- Дескриптор (например: `stdin`, ...) — интерфейс связи с ресурсом
- Файловый дескриптор

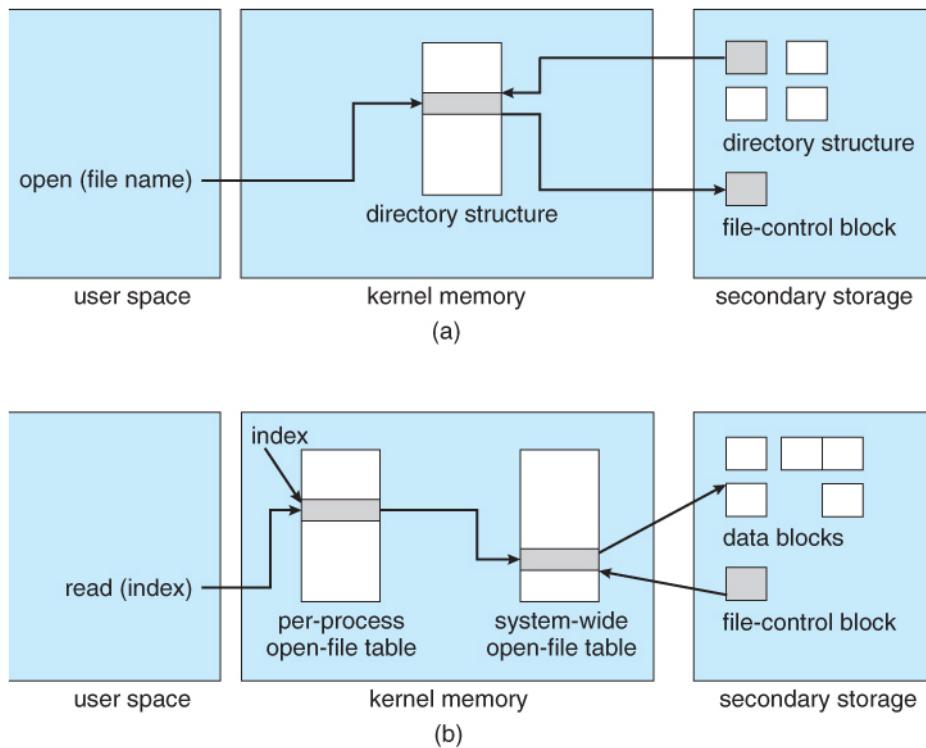


Рис. 3.19: Read() and Open()

TODO Доделать

## 3.18 Пару слов о типах

Лучше всего использовать следующие типы данных:  
**TODO Why?**

- `off_t`
- `size_t`
- `ssize_t`

## 3.19 Common pitfalls

- Неатомарные операции (окно `race`)
- **ТОСТОУ** (пример `race condition`) — класс багов, связанных с изменением состояния объекта между проверкой и реальным использованием. (проверили что файл есть, захотели открыть, его кто-то удалил между этими действиями, проиграли).
- Утечка дескрипторов
- Файловая система гарантирует, что до тех пор пока ты держишь файловый дескриптор на файл с ним ничего не произойдет извне (функции оканчивающиеся на "`at`", защита от **ТОСТОУ**)
- `openat`

## 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

Необходимо написать подмножество утилиты `find`.

Программа должна:

- Первым аргументом принимать абсолютный путь, в котором будет производиться поиск файлов.
- По умолчанию выводить в стандартный поток вывода все найденные файлы по этому пути
- Поддерживать аргумент **-inum num**. Аргумент задает номер инода
- Поддерживать аргумент **-name name**. Аргумент задает имя файла

- Поддерживать аргумент **-size [−=+]size**. Аргумент задает фильтр файлов по размеру(меньше, равен, больше)
- Поддерживать аргумент **-nlinks num**. Аргумент задает количество hardlink'ов у файлов
- Поддерживать аргумент **-exec path**. Аргумент задает путь до исполняемого файла, которому в качестве единственного аргумента нужно передать найденный в иерархии файл
- Поддерживать комбинацию аргументов. Например хочется найти все файлы с размером больше 1GB и скомпилировать их утилите /usr/bin/sha1sum.
- Выполнять поиск рекурсивно, в том числе во всех вложенных директориях.
- Сильные духом призываются к выполнению задания с использованием системного вызова *getdents(2)*. Остальные могут использовать *readdir* и *opendir* для чтения содержимого директории.

# Лекция 4

## Виртуальная память

### 4.1 Прерывания и исключения

- Процессор с памятью не могут быть жить в вакууме (без ОС)
- Простыми словами: иногда процессор не знает что ему делать в конкретной ситуации, так как не знает контекста исполнения, тогда он просит помощи у внешней среды (чаще всего ОС)
- *Interrupt Deriving Architecture* — есть таблица, каждой ячейке которой соответствует какая-либо исключительная ситуация(например, поделить на нуль) и функция ее разрешающая.
- **IDTR** — регистр, в котором хранится адрес *Interrupt Descriptor Table* (глобальный)  
В некоторых архитектурах находится по фиксированному адресу (например, x86 — защищенный режим)

Проще говоря **callbacks**

- Другой подход — *Polling* (есть управляющий код, который периодически опрашивает устройство на предмет того, что нужно обработать; процессор выставляет флаг — "нуждаюсь в обработке").

Пример — сетевая карта

- У каждого подхода свои плюсы и минусы
- **cr2** — контрольный регистр, считывается функцией *do\_page\_fault* (которая занимается обработкой **page fault**)

## 4.2 Память

Хочется чтобы каждый процесс был защищен от любого другого  
Проблемы памяти:

1. Памяти мало, она дорогая
2. Памяти мало, программ много, как договориться?

Закон Парето - 80% обращений к 20% памяти в среднем у пользовательской программы

Может тогда выгружать неиспользуемую память на диск?

Может еще переиспользовать память? (например, сегмент **text** у **Chrome**)

3. Памяти мало, программ много, как защититься?

Неплохо было бы выложить в read-only какую-то память (сегмент **text** — *const*-переменные)

stackoverflow.cpp

```
#include <iostream>

void a() {
    char A[100]; // Локальная переменная
    std::cin >> A; // 132 байта
    // Начали писать вверх и переписали адрес возврата
    // (так как стек растет вниз, а адреса массива вверх)
    // Когда делаем RET, вернемся по испорченному адресу
}

void b() {
    a();
}
```

Способ защиты — память, которая может записываться не может выполняться

Канарейка на стеке — проверяем значение переменной 'канарейка', которую добавили после адреса возврата

Хотим чтобы память ядра была недоступна пользователям

Как можно это все сделать?

## 4.3 Подходы к организации памяти

TODO

### 4.3.1 Досегментная организация(№1)

### 4.3.2 Сегментная организация(№2)

### 4.3.3 Страницчная организация(№3)

### 4.3.4 Страницчная организация в x86(Реальность)

## 4.4 MMU

TODO

## 4.5 Переключение контекста

- Восстановить память процесса (регистр **cr3**)
- Восстановить поток выполнения (значения регистров)
- Как защитить память ядра? — привилегиями

TODO no image for this

## 4.6 Примеры использования

### 4.6.1 Подкачка по требованию

- Пусть у нас есть фильм. Давайте читать этот файл используя окно считывания. Прикольно. Но можем замалпить файл в память и читать его.
- Каждый раз когда читаем еще незагруженную страницу, **page fault**, далее ядро подгружает страницу.
- Еще пример — **\$ chrome help** (нам незачем инициализировать всю работу приложения)
- Ядро достаточно оптимально все это делает.

### 4.6.2 Copy on write

copyonwrite.cpp

```
#include <assert.h>
#include <sys/types.h>
#include <unistd.h>
```

```

int main() {
    int x = 42;
    const pid_t pid = fork();
    assert(pid != -1);

    if (!pid) {
        x = 43;
    }
}

```

Какое значение x будет в parent и в child?

Немного по-другому

copyonwrite2.cpp

```

#include <sys/types.h>
#include <unistd.h>
#include <iostream>
#include <cassert>

int main() {
    int x = 42;
    const pid_t pid = fork();
    assert(pid != -1);

    const bool am_i_child = !pid;

    if (am_i_child) {
        sleep(2);
        x = 43;
    } else {
        sleep(3);
    }

    std::cout << "Am I child: " << am_i_child << " " << x << " " << &x <<
        std::endl;
}

```

- Однаковые адреса, значение разное (child — 43, parent — 42)
- При создании процесса — ничего не делаем
- За всей этой историей стоит **MMU**

### 4.6.3 Swap

- Скидывание на диск холодной data
- **MMU** ходит по кругу и проверяет использование страниц

#### 4.6.4 Другие

- Рост стека
- Разделяемый текст => Разделяемые библиотеки
- Разделяемая память

### 4.7 Выделение памяти

#### 4.7.1 Пример №1

mem1.c

```
#include <stdlib.h>

int main() {
    printf("Hello\n");
    const char * a = malloc(100);
}
```

```
$ cc mem1.c -o mem1
$ strace -f ./mem1
```

mem1.trace

```
execve("./mem1", ["../mem1"], 0x7fff24c75d48 /* 35 vars */) = 0
...
...
...
brk(NULL)                      = 0x55a077d5e000
brk(0x55a077d7f000)            = 0x55a077d7f000
write(1, "Hello\n", 6Hello     ) = 6
exit_group(0)                  = ?
+++ exited with 0 +++
```

- Нет никакого выделения памяти через *malloc* (потому что это не системный вызов)
- **\$ man 2 brk, sbrk** — явное увеличение/уменьшение кучи
- Для того же самого со стеком системных вызовов нет (только ассемблерные инструкции)

#### 4.7.2 Syscalls

- *malloc/calloc/realloc* через прослойку выделяют себе блок памяти, режут его на куски и выдают клиенту-программисту

- *free()* — отдает блок памяти не ядру, а прослойке
- *malloc()* — неинициализированная память (сколько байт)
- *calloc()* — память инициализированная нулями (сколько объектов хотим)
- *realloc()* — все вместе (может еще перемещать память, так как возвращается не всегда тот указатель, который передали)
- Операторы **new** и **delete** выражаются через функции С

### 4.7.3 Helpers

- **Valgrind** — выполняет инструкцию за инструкцией программы, которую ему передали (строит внутри себя модель и отслеживает выделения памяти), долго работает (значительно снижает производительность)
- **Sanitizer (ASAN)** — инструментируют код, добавляют дополнительные проверки (тоже внутри себя строит модель)
- Программы с санитайзером работают не на порядок медленнее, а в 2-3 раза.

### 4.7.4 Пример №2

Когда мы делаем *free()*, память может остаться в userspace.

mem2.c

```
#include <stdlib.h>

int main() {
    printf("Hello\n");
    char * a = malloc(100);
    a[0] = 'X';
    free(a);
    printf("%c\n", a[0]);
}
```

Что будет? — **Undefined Behavior**

Но ведь ничего не напечаталось

mem2.trace

```
execve("./mem2", ["../mem2"], 0x7ffeb9fe31b8 /* 35 vars */) = 0
...
...
...
brk(NULL)                      = 0x557938333000
brk(0x557938354000)            = 0x557938354000
write(1, "Hello\n", 6Hello      ) = 6
```

```

write(1, "\0\n", 2
)                      = 2
exit_group(0)           = ?
+++ exited with 0 +++

```

На самом деле все напечаталось, просто мы этого не видим

## 4.8 Mapping

### 4.8.1 Syscalls

*mmap()*, *munmap()*

- Представьте, что у вас есть файл, и вы заранее не знаете к какой части файла будете обращаться.
- Замаппив этот файл в память своего процесса, нужно понимать, что ядро автоматически не полностью подгружает его в память.
- Допустим, есть куча процессов, которые читают свой исполняемый файл  
Если бы мы делали это не через маппинг, то каждый раз надо было бы считывать что-то в локальные буфера
- С помощью маппинга можно избежать этой проблемы

### 4.8.2 Пример №1

map1.c

```

#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    const int fd = open("/etc/passwd", O_RDONLY);

    struct stat st;
    assert(fstat(fd, &st) != -1);
    void *addr = mmap(NULL, st.st_size, PROT_READ, MAP_ANONYMOUS | MAP_PRIVATE,
                      fd, 0);
    //Отображение файла начиная с нуля

    close(fd);
}

```

```
}
```

### map1.trace

```
execve("./map1", ["../map1"], 0x7ffd039a12a8 /* 35 vars */) = 0
...
...
...
openat(AT_FDCWD, "/etc/passwd", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=1053, ...}) = 0
mmap(NULL, 1053, PROT_READ, MAP_PRIVATE|MAP_ANONYMOUS, 3, 0) = 0x7f7afa213000
close(3) = 0
exit_group(0) = ?
+++ exited with 0 +++
```

В выводе *openat()*, так как мы делали не прямой системный вызов

### 4.8.3 Пример №2

Печать содержимого файла */etc/passwd*

### map2.c

```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

int main() {
    const int fd = open("/etc/passwd", O_RDONLY);

    struct stat st;
    assert(fstat(fd, &st) != -1);
    char *addr = mmap(NULL, st.st_size, PROT_READ, MAP_PRIVATE, fd, 0);

    for (size_t i = 0; i < st.st_size; ++i) {
        printf("%c", addr[i]);
    }

    munmap(addr, st.st_size);

    close(fd);
}
```

### map2.trace

```
execve("./map2", ["../map2"], 0x7ffc9c1e94e8 /* 35 vars */) = 0
...
...
openat(AT_FDCWD, "/etc/passwd", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=1053, ...}) = 0
mmap(NULL, 1053, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fe33146a000
fstat(1, {st_mode=S_IFREG|0644, st_size=2251, ...}) = 0
brk(NULL) = 0x5607682c3000
brk(0x5607682e4000) = 0x5607682e4000
munmap(0x7fe33146a000, 1053) = 0
close(3) = 0
write(1, "root:x:0:0::/root:/bin/bash\nbin:"..., 1053root:x:0:0::/root:/bin/bash
...
...
...
exit_group(0) = ?
+++ exited with 0 +++
```

#### 4.8.4 Additional

- *pause()* — системный вызов, останавливает программу, пока она не получит сигнал
- Ядро всегда зануляет память, которую отдает
- Как работают санитайзеры: выделяют справа и слева блоки памяти, в которых устанавливают флаги и потом чекают на access к ним

#### 4.8.5 Пример №3

\$ man 2 mprotect

### map3.c

```
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <unistd.h>

#include <assert.h>
#include <stdio.h>
#include <stdlib.h>

#define PAGE_SIZE 4096

int main() {
```

```

const size_t size = 4 * PAGE_SIZE;
char *addr = mmap(NULL, size, PROT_READ | PROT_WRITE, MAP_PRIVATE |
    MAP_ANONYMOUS, -1, 0);
assert(addr != MAP_FAILED);

for (size_t i = 0; i < size; ++i) {
    addr[i] = 'X';
}

mprotect(&addr[3 * PAGE_SIZE], PAGE_SIZE, PROT_NONE);

addr[3 * PAGE_SIZE - 1] = 'Y';
addr[3 * PAGE_SIZE] = 'Y'; //Segmentation fault

munmap(addr, size);
}

```

map3.trace

```

execve("./map3", ["../map3"], 0x7ffc727faa38 /* 35 vars */) = 0
...
...
...
mmap(NULL, 16384, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
    0x7f1eb681e000
mprotect(0x7f1eb6821000, 4096, PROT_NONE) = 0
--- SIGSEGV {si_signo=SIGSEGV, si_code=SEGV_ACCERR, si_addr=0x7f1eb6821000} ---
+++ killed by SIGSEGV (core dumped) +++

```

\$ man 2 madvise

## 4.9 Аллокаторы памяти

Зачем писать свой аллокатор памяти?

Случай:

1. Есть особенность данных, которую хотим использовать
2. Когда нужно дебажить свой аллокатор

*Slab*-аллокаторы эксплуатируют то, что мы выделяем и освобождаем память одного и того же размера

slab.c

```

struct SLAB {
    uint64_t index;
    object_t objects[64];
    SLAB *prev;
    SLAB *next;
}

```

```
};

struct pool {
    SLAB *full;
    SLAB *parted;
    SLAB *empt;
};
```

## 4.10 Безопасность

### 4.10.1 Meltdown

- Состоит из нескольких этапов
- Это атака позволяет читать непrivилегированную для процесса память ядра
- Использует следующие особенности:
  - Спекулятивное выполнение
  - Утечка данных через сторонний канал
    - \* Процессоры имеют кэши
    - \* Оценка времени доступа к массиву позволяет понять, находятся ли данные в кэше

### 4.10.2 ASLR

Address space layout randomization

aslr.c

```
#include <stdio.h>

int main() {
    char a;

    printf("%p\n", &a);
    printf("%p\n", main);
}
```

Почему значения почти одинаковые при каждом запуске?  
Они рандомизируются (еще один рубеж защиты)

## 4.11 Page Reclaiming

TODO presentation

## 4.12 Page Fault

TODO presentation

## 4.13 Литература

- Understanding the Linux Kernel by Daniel P. Bovet & Marco Cesati  
(Достаточно хорошо описана архитектура)
- Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3  
(Руководство от Intel)
- x86 Instruction Set Architecture by Tom Shanley  
(Выжимка руководства от Intel)
- What every programmer should know about memory by Ulrich Drepper  
(Очень полезно)
- Безопасное программирование на С и С++. Роберт С. Сиакорд  
(Про уязвимости)

## 4.14 Домашнее задание №3

Кусочек **JIT** компилятора

Цель — получить знакомство с системными вызовами, используемыми для получения/освобождения памяти от ядра. Получить представление о том, как может работать **JIT** компилятор.

Программа должна

- Выделить память с помощью *mmap(2)*.
- Записать в выделенную память машинный код, соответствующий какой-либо функции.
- Изменить права на выделенную память - чтение и исполнение (*mprotect(2)*).
- Вызвать функцию по указателю на выделенную память.
- Освободить выделенную память.

Что может помочь?

- `$ man objdump`
- `help disassemble` в `gdb`

Extra points

- Сильные духом призываются к возможности модификации кода выполняемой функции в runtime.
- Например, вы можете получить аргументом вызова вашей программы какое-то число и пропатчить машинный код этим числом. Эта часть задания будет оцениваться в дополнительные баллы.