

*Федеральное государственное автономное учреждение  
высшего образования*

**Московский физико-технический институт  
(национальный исследовательский университет)**

---

**АРХИТЕКТУРА КОМПЬЮТЕРОВ И  
ОПЕРАЦИОННЫЕ СИСТЕМЫ**

---

**III СЕМЕСТР**

Физтех-школа: *ФПМИ*

Направление: *ПМИ*

Лектор: *Андреев Александр Николаевич*



Долгопрудный, Осень 2022 год.

# Содержание

<b>1</b>	<b>Вводная лекция</b>	<b>5</b>
1.1	Операционная система . . . . .	5
1.2	Из каких компонент состоит компьютер? . . . . .	5
1.2.1	Процессор . . . . .	6
1.2.2	Оперативная память . . . . .	6
1.3	Немного ассемблера . . . . .	7
1.4	Мультизадачность . . . . .	7
1.4.1	Суперскалярность . . . . .	7
1.4.2	CPU pipeline . . . . .	8
1.4.3	Мультипроцессорность . . . . .	8
1.5	Системные вызовы . . . . .	8
1.6	POSIX . . . . .	9
1.7	libc . . . . .	9
1.7.1	Пример . . . . .	9
1.8	Файловые дескрипторы . . . . .	10
<b>2</b>	<b>Представление данных в компьютере</b>	<b>10</b>
2.1	Беззнаковые типы . . . . .	10
2.1.1	Endianness . . . . .	10
2.2	Выравнивание . . . . .	11
2.2.1	Выравнивание структур . . . . .	12
2.3	Знаковые числа . . . . .	12
2.3.1	One's complement . . . . .	12
2.3.2	Two's complement . . . . .	13
2.4	Действительные числа . . . . .	14
2.4.1	Числа с фиксированной точкой . . . . .	14
2.4.2	Числа с плавающей точкой . . . . .	14

2.4.3	Decimals . . . . .	15
2.5	Кодировки . . . . .	15
2.5.1	Немного терминологии . . . . .	16
2.5.2	ASCII . . . . .	16
2.5.3	Unicode . . . . .	16
2.5.4	UTF-32 . . . . .	16
2.5.5	UTF-8 . . . . .	17
<b>3</b>	<b>Файлы</b>	<b>17</b>
3.1	Файлы и директории . . . . .	17
3.1.1	Имя файла . . . . .	17
3.1.2	Имя директории . . . . .	18
3.2	Файловые системы . . . . .	18
3.2.1	ext2 . . . . .	18
3.2.2	ext4 . . . . .	19
3.2.3	Другие файловые системы . . . . .	19
3.2.4	sysfs и procfs . . . . .	20
3.2.5	FUSE . . . . .	20
3.3	Файловые дескрипторы . . . . .	20
3.3.1	Работа с данными файла . . . . .	21
3.3.2	Работа с метаданными файла . . . . .	21
3.3.3	Права доступа . . . . .	21
3.3.4	Права доступа для директорий . . . . .	22
3.3.5	Регулярные файлы . . . . .	22
3.3.6	Директории . . . . .	22
3.3.7	Символические ссылки . . . . .	23
3.3.8	Жесткие ссылки . . . . .	23
3.3.9	Символьные устройства (character device) . . . . .	23

3.3.10	Блочные устройства (block device)	23
<b>4</b>	<b>Память</b>	<b>24</b>
4.1	Виртуальная память	24
4.1.1	Сегментная адресация	24
4.1.2	Страничная адресация	24
4.1.3	Multi-level page tables	25
4.1.4	Что хранится в PTE?	25
4.1.5	Устройство виртуального адреса	26
4.1.6	ОС и таблицы страниц	26
4.1.7	Выделение памяти: on-demand paging	26
4.1.8	Minor page fault	27
4.1.9	File memory mapping	27
4.1.10	Major page faults	27
4.1.11	Page cache	27
4.2	fsync	28
4.3	mmap и munmap	28
4.3.1	mmap: prot	28
4.3.2	mmap: flags	28
4.4	Псевдофайлы для контроля расхода памяти	29
4.5	Вытеснение страниц и swap	29
<b>5</b>	<b>Процессы</b>	<b>29</b>
5.1	Атрибуты процесса	30
5.2	Состояния процессов	30
5.3	fork	31
5.4	execve	31
5.4.1	execve: сохраняемые атрибуты	32
5.5	Атрибуты процесса: PID, PPID, TGID, ...	32

5.6	Атрибуты владельца процесса . . . . .	33
5.7	Работа с процессами . . . . .	33
5.7.1	exit . . . . .	33
5.7.2	wait . . . . .	34
5.8	wait4 . . . . .	34
5.8.1	Макросы для wait4 . . . . .	34
5.9	Лимит ресурсов процессов . . . . .	34
5.10	Механизмы изоляции . . . . .	35
5.11	ELF . . . . .	36
5.11.1	ELF: header . . . . .	37
5.11.2	ELF: секции . . . . .	37
5.11.3	ELF: .symtab . . . . .	38
5.11.4	ELF: сегменты . . . . .	38
5.11.5	ELF: program header . . . . .	38
5.12	execve: принципы работы . . . . .	38
<b>6</b>	<b>Linux scheduler</b>	<b>39</b>
6.1	Realtime scheduling . . . . .	39
6.2	Process niceness . . . . .	39
6.2.1	Timeslice . . . . .	39
6.2.2	Timeslice & niceness . . . . .	39
6.2.3	Timeslice & niceness: проблемы подхода . . . . .	40
6.3	completely Fair Scheduler . . . . .	40
6.3.1	CFS: проблемы . . . . .	40
<b>7</b>	<b>Сигналы</b>	<b>40</b>
7.1	Сигналы: примеры . . . . .	41
7.2	Доставка сигналов . . . . .	41
7.3	Signal safety . . . . .	41

7.4	Обработка сигналов . . . . .	41
7.5	Посылка сигналов . . . . .	42
7.5.1	Посылка сигналов: аргументы <code>kill</code> . . . . .	42
7.6	Доставка сигналов: маски сигналов . . . . .	42
7.6.1	Доставка сигналов: <code>sigprocmask</code> . . . . .	43
7.7	Обработка сигналов: <code>sigaction</code> . . . . .	43
7.7.1	Обработка сигналов: <code>siginfo_t</code> . . . . .	44
7.8	<code>SIGCHLD</code> . . . . .	44
7.9	Доставка сигналов во время системных вызовов . . . . .	45
7.10	Ожидание сигналов: <code>pause</code> , <code>sigsuspend</code> и <code>sigwaitinfo</code> . . . . .	45
7.11	Как устроены сигналы? . . . . .	45
7.12	Наследование сигналов: <code>fork</code> и <code>execve</code> . . . . .	45
7.13	Почему сигналы – это плохо? . . . . .	46
7.14	Почему лучше всегда использовать <code>sigaction</code> ? . . . . .	46
7.15	Real-time signals . . . . .	46
7.16	Обработка сигналов с помощью пайпов . . . . .	47
7.17	Использование сигналов . . . . .	47
7.17.1	<code>nginx</code> . . . . .	47
7.17.2	<code>golang</code> . . . . .	48

## 1 Вводная лекция

### 1.1 Операционная система

Операционная система – абстракция, которая связывает различные компоненты компьютера и пользовательские программы.

### 1.2 Из каких компонент состоит компьютер?

- Центральный процессор (CPU или ЦП)

- Чипсет и материнская плата
- Оперативная память (Random Access Memory = RAM)
- Накопители (HDD, SSD, NVMe)
- Аудиокарта
- Сетевая карта
- GPU
- Шина (PCI, I2C, ISA)

### 1.2.1 Процессор

- Исполняет команды или *инструкции*
- Регистры – самые быстрые доступные ячейки памяти
- Регистры определяют разрядность процессора
- Операндами могут быть либо константы, либо регистры, либо ссылки на память

### 1.2.2 Оперативная память

- Random Access Memory
- Адресное пространство – непрерывный массив байт от 0 до  $2^N$ , где  $N$  – разрядность процессора (64 бита)
- В реальности процессоры на текущий момент обычно адресуют не более 48 бит (256 терабайт)
- Инструкции процессора расположены также в RAM – архитектура Фон-Неймана

Сейчас оперативная память работает значительно медленнее процессора (доступ к RAM занимает несколько десятков инструкций процессора). Поэтому внутри процессора есть несколько уровней своей “оперативной памяти”: L1, L2, L3. Они устроены немного иначе, чем оперативная память, и стоят очень дорого. Если запрашивается доступ к 1 байту, а затем к следующему байту, то второе считывание будет сделано не из оперативной памяти, а из кэша (L1/L2/L3, в зависимости от их наполнения). О том, почему есть несколько уровней кэша, будет рассказано в следующих лекциях. Из-за существования кэшей, нам выгодно, чтобы данные лежали “рядом” в памяти. Один из примеров: [ускорение умножения матриц](#).

## 1.3 Немного ассемблера

Ассемблер – это вид для человека, эти команды – не процессорные инструкции. На современных процессорах Intel длина инструкции обычно занимает от 1 до 8 байт. В этом курсе будет рассмотрена только архитектура x86. В инструкцию записывается вся необходимая информация: используемые константы, используемые адреса памяти и т.д. Подробнее об этом будет рассказано позже.

```
1 mov rax, qword ptr [rax]
2 add rax, 2
3 mov rbx, 1
4 add rax, rbx
```

`rax`, `rbx` – это регистры процессора. Всего различных регистров общего назначения 16.

Первая инструкция в этом коде берёт адрес регистра `rax`, считывает его содержимое, и записывает в него же, в `rax`.

Вторая команда прибавляет к содержимому `rax` 2.

Третья команда записывает в `rbx` 1.

Четвертая команда прибавляет `rbx` к `rax`.

## 1.4 Мультизадачность

- Мультизадачность – способность системы исполнять несколько задач (процессов) одновременно
- Cooperative multitasking – процессы добровольно передают управление друг другу
- Preemptive multitasking – процессы вытесняются ОС каждые несколько миллисекунд

Минус cooperative multitasking: если процесс завис, то он не передаст управление дальше, остается только reset.

Первая Windows, в которой появился multitasking, это Windows 95, до этого был singleprocess MS-DOS.

### 1.4.1 Суперскалярность

- Параллелизм уровня инструкций



- Если две инструкции независимы друг от друга, их можно выполнить параллельно
- Каждая инструкция состоит из нескольких этапов: fetch, decode, execute, memory access, register write back
- CPU pipeline

Пример процессора без суперскалярности: российский Эльбрус, в котором одна инструкция процессора содержит несколько операций, которые выполняются параллельно. Такой принцип называется VLIW – Very Long Instruction Word.

### 1.4.2 CPU pipeline

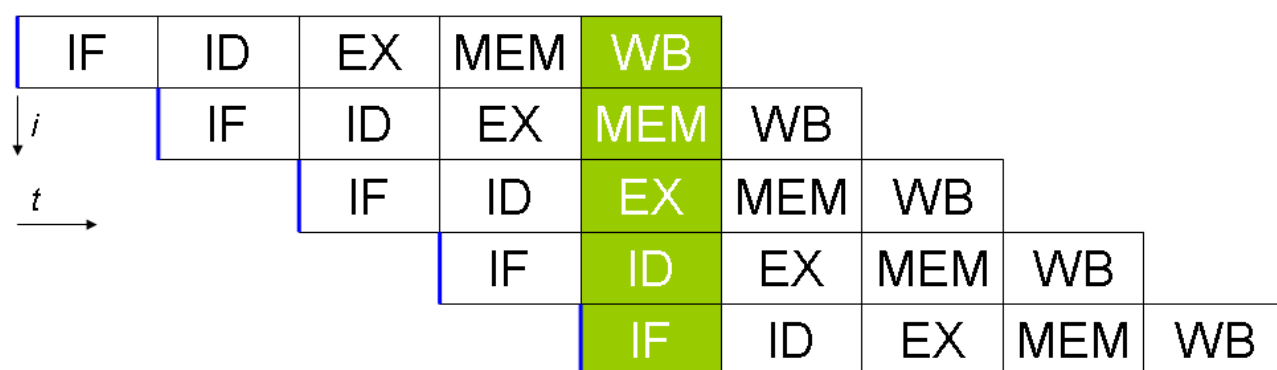


Рис. 1: CPU pipeline

### 1.4.3 Мультипроцессорность

- Тактовая частота процессоров не растет примерно с 2005 года
- Поэтому современные процессоры обычно имеют несколько ядер
- *Планировщик (scheduler)* ОС для каждого ядра процессора в каждый момент времени решает какой процесс будет запущен
- Возникают проблемы синхронизации

## 1.5 Системные вызовы

- Системные вызовы – это интерфейс операционной системы для процессов
- ABI = application binary interface

- SystemV ABI

Системный вызов – это очень дорогая операция. У каждой операционной системы свой ABI.

## 1.6 POSIX

- Portable Operating System Interface
- Стандарт, описывающий интерфейс операционных систем
- Системные вызовы – часть POSIX, но не все
- Например, POSIX описывает как должна быть устроена файловая система

Иными словами, POSIX – это стандарт написания операционных систем. Windows – не POSIX-совместимая система.

## 1.7 libc

- Стандартная библиотека C
- Реализует системные вызовы в виде функций C
- Ещё куча всяких полезных функций :)
- Много реализаций, glibc одна из самых больших

POSIX определяет, как устроены системные вызовы в виде функций языка C.

### 1.7.1 Пример

```
1 int res = read(0, &buf, 1024);
2 if (res < 0) {
3     char* err = strerror(errno);
4     // ...
5 }
```

Функция `read` возвращает `-1`, если считать не получилось, в противном случае – количество записанных байт.

`errno` – это глобальная переменная (внутри одного потока), в которой хранится последняя ошибка.

Вернуть массив из функции сложно (о причинах будет рассказано в следующих лекциях), поэтому обычно мы просим не вернуть результат, а записать его по некоторому адресу в памяти.

## 1.8 Файловые дескрипторы

- “Everything is a file!”
- Каждый файл имеет своё имя (или *путь*)
- Преобразовывать имя файла на каждый сисколл дорого
- Сначала нужно получить файловый дескриптор (например, через сисколл `open`)
- Все остальные операции без использования пути

Файловый дескриптор – это число. Например, 0 – это `stdin`, 1 – это `stdout`, 2 – `stderr`.

## 2 Представление данных в компьютере

### 2.1 Беззнаковые типы

- Представляют из себя  $N$ -битные положительные числа на отрезке  $[0, 2^N - 1]$
- Переполнение точно определено стандартом C (как сложение в  $\mathbb{Z}_{2^N}$ )
- $1111 + 0001 = 10000 = 0$

#### 2.1.1 Endianness

- Если  $N = 64$ , то  $64/8 = 8$  байт нужно, чтобы представить число в памяти
- Если  $N = 32$ , то  $32/8 = 4$  байта
- В какой последовательности хранить биты?

Есть 2 типа endianness:

- Little-endian: первые байты хранят младшие биты числа
- Big-endian: первые байты хранят старшие биты

Сейчас более распространен Little-endian.

Традиционно Big-endian используется в передаче данных по сети. Также первые процессоры использовали big-endian. PowerPC тоже использует big-endian.

На некоторых arm-процессорах есть инструкция, позволяющая менять endian “на лету”.



Рис. 2: Endianess

## 2.2 Выравнивание

- Числа быстрее считываются процессором, если они лежат по адресам, кратным их размерам
- Например: `sizeof(int) = 4`  $\Rightarrow$  выравнивание по границе 4 байт
- `char` – 1 байт
- `short` – 2 байта
- `int` – 4 байта
- `long long` – 4 байта

Работа с выровненными данными происходит быстрее.

Есть архитектуры, которые в принципе не позволяют читать по невыровненным адресам, например, arm. В процессорах Intel можно сделать так же.

### 2.2.1 Выравнивание структур

- Члены структур располагаются рядом
- Но если им не хватает выравнивания, компилятор “добивает” структуру padding'ами
- Выравнивание структуры – максимальное выравнивание среди всех выравниваний её членов

## 2.3 Знаковые числа

### 2.3.1 One's complement

- $-A = \text{BitwiseNot}(A)$
- Диапазон:  $[-2^{N-1} + 1, 2^N - 1]$

Преимущество такого представления: естественным образом реализуется сложение чисел. Однако есть проблема.

- $-1 = 1110$
- $+1 = 0001$
- $1110 + 0001 = 1111 = -0$

Получается 2 представления нуля. Это порождает еще проблемы:

- $-1 = 1110$
- $+2 = 0010$
- $1110 + 0010 = 10000 = 0$
- Упс...

### One's complement: end-around-carry

- Бит переноса отправляется назад, чтобы всё исправить
- $1110 + 0010 = 10000 = 0 + 1 = 1$

### One's complement: недостатки

- Два представления для 0:  $0000 = +0$  и  $1111 = -0$
- End-around-carry
- Зато сложение и вычитания одинаковое для знаковых и беззнаковых чисел (почти)!

### 2.3.2 Two's complement

- Определение отрицательных чисел:  $A + (-A) = 0$
- Давайте каждому положительному числу сопоставим отрицательное
- $-A = \text{BitwiseNot}(A) + 1$
- Одно представление нуля:  $-0 = \text{BitwiseNot}(A) + 1 = 1111 + 1 = 0000 = +0$
- Диапазон чуть больше, чем у one's complement:  $[-2^N, 2^N - 1]$
- Используется в современных процессорах

Теперь мы можем складывать знаковые и беззнаковые числа абсолютно одинаково.

### Two's complement: недостатки

- Операции сравнения теперь сложные
- Умножение требует sign extension:  $0010 = 00000010, 1000 = 11111000$
- “Перекося” диапазона представимых чисел
- $\text{abs}(\text{INT\_MIN}) = ???$

## 2.4 Действительные числа

### 2.4.1 Числа с фиксированной точкой

- $N$  бит на целую часть,  $M$  бит на дробную
- Всегда одинаковая точность
- Операции легко реализуются

### 2.4.2 Числа с плавающей точкой

Раньше процессоры имели отдельную плату для операций с числами с плавающей точкой.

- IEEE 754
- Стандарт 1985 года

Числа с плавающей точкой представлены 3 частями:

- Представление:  $(-1)^S \times M \times 2^E$
- $S$  – бит знака,  $M$  – мантисса,  $E$  – экспонента
- **float** (single):  $|S| = 1$ ,  $|M| = 23$ ,  $|E| = 8$
- **double**:  $|S| = 1$ ,  $|M| = 52$ ,  $|E| = 11$

### Нормализованные значения

- $|E| \neq 0$  и  $E \neq 2^{|E|} - 1$
- Экспонента хранится со смещением:  $E_{real} = E - 2^{|E|-1}$
- Мантисса имеет “виртуальную 1”:  $M_{real} = 1.mmmmmmm$

### Денормализованные значения

- $E = 0$
- $E_{real} = 1 - 2^{|E|} = 1$
- Это самые близкие к нулю числа и сам ноль (0.0 и +0.0)

## Специальные значения

- $E = 2^{|E|-1}$
- Если  $M = 0$ , то число представляет собой бесконечное значение
- Если  $M \neq 0$ , то число – NaN
  - Используются при операциях с неопределенным значением: например,  $\text{sqrt}(X)$ ,  $\log(X)$ ,  $X < 0$

## Проблемы IEEE 754

- При вычислениях накапливается ошибка
- Сложение и умножение неассоциативно
- Умножение недистрибутивно
- $\text{NaN} \neq \text{NaN}$  (??)
- 0.0 и +0.0

Более подробно о числах с плавающей точкой можно прочитать [здесь](#).

### 2.4.3 Decimals

- Представляются в виде двух чисел:  $N$  – знаменатель,  $M$  – числитель
- Все операции реализуются через приведение к общему знаменателю
- $N$  и  $M$  обычно используют длинную арифметику, поэтому в теории точность ограничена только оперативной памятью
- Используются в финансах

## 2.5 Кодировки

- Умеем оперировать числами, но как перевести числа в текст?
- Кодировки – “карты”, сопоставляющие наборы байт каким-то образом в символы



### 2.5.1 Немного терминологии

- Character – что-то, что мы хотим представить
- Character set – какое-то множество символов
- Coded character set (CCS) – отображение символов в уникальные номера
- Code point – уникальный номер какого-то символа

### 2.5.2 ASCII

- American Standard Code for Information Interchange, 1963 год
- 7-ми битная кодировка, то есть кодирует 128 различных символов
- Control characters: с 0 по 31 включительно, непечатные символы, мета-информация для терминалов

### 2.5.3 Unicode

- Codespace: 0 до 0x10FFFF ( $\sim 1.1$  млн. code points)
- Code point'ы обозначаются как  $U + \langle \text{число} \rangle$
- $\aleph = U + 2135$
- $r = U + 0072$
- Unicode – не кодировка: он не определяет как набор байт трактовать как characters

### 2.5.4 UTF-32

- Использует всегда 32 бита (4 байта) для кодировки
- Используется во внутреннем представлении строк в некоторых языках программирования (например, Python)
- Позволяет обращаться к произвольному code point'у строки за  $\mathcal{O}(1)$
- BOM определяет little vs. big-endian

Проблема: используется много места. Например, если мы пишем текст на английском, то под каждый символ будет выделено 4 байта, а можно было бы обойтись 1 (ASCII).

### 2.5.5 UTF-8

- Unicode Transformation Format
- Определяет способ как будут преобразовываться code point'ы
- Переменная длина: от 1 байта (ASCII) до 4 байт

U+0000...U+007F	→	0xxxxxxx
U+0080...U+07FF	→	110xxxxx 10xxxxxx
U+0800...U+FFFF	→	1110xxxx 10xxxxxx 10xxxxxx
U+10000...U+10FFFF	→	11110xxx 10xxxxxx 10xxxxxx 10xxxxxx

Рис. 3: UTF-8

#### UTF-8 overlong encoding

- 00100000 = U+0020
- 1100000010100000 = U+0020!
- overlong form или overlong encoding
- С точки зрения стандарта является некорректным представлением

## 3 Файлы

### 3.1 Файлы и директории

Файл – это сущность, которая содержит данные и имеет имя.

В Unix: Everything is a file!

#### 3.1.1 Имя файла

- Не более PATH\_MAX символов: 4 Кб на современных ОС, 256 байт для portability
- PATH\_MAX включает \0 в конце
- Разделитель пути – /
- Части пути не более 255 символов каждая

### 3.1.2 Имя директории

- Абсолютный путь: начинается с корня (например, /Users/carzil/mipt)
- Относительный путь: вычисляется от текущей директории (например, carzil/mipt)
- . – текущая директория (./carzil/mipt = carzil/mipt и ./carzil/././mipt = ./carzil/mipt)
- .. – директорий выше (/Users/carzil/mipt/.. = /Users/carzil)

## 3.2 Файловые системы

- Структура данных для организации хранения информации
- Метаданные – информация о файле: дата последнего изменения, права доступа, создатель и так далее
- Работают поверх хранилища (HDD, SSD, NVMe)
- Хранилище традиционно разбивается на *блоки*
- Размер блоков обычно 512 байт или 4Кб

### 3.2.1 ext2

- Linux, 1993 год
- inode – физическое представление файла на диске: заголовок с метаданной + информация где он хранится
- Директории тоже хранятся в inode, так как директория – файл!

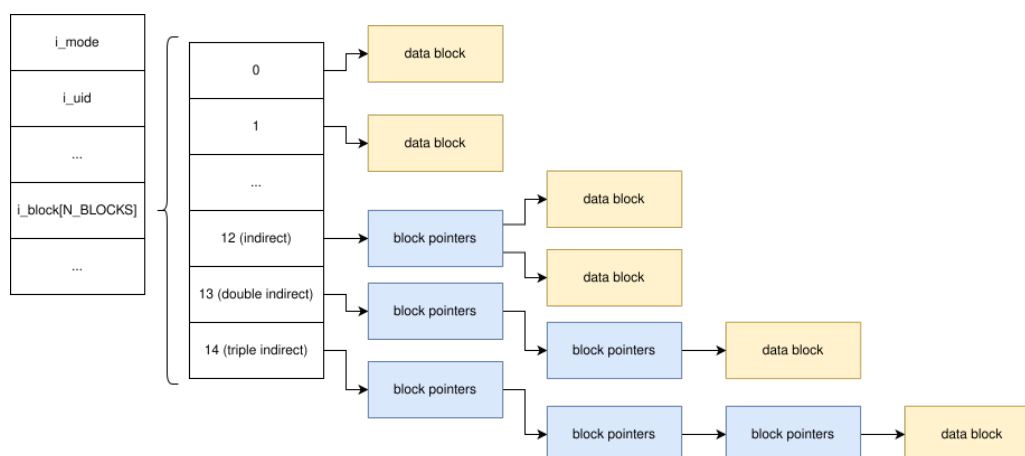


Рис. 4: inode

### 3.2.2 ext4

- 2006 год
- Де-факто стандартная файловая система для Linux
- Журналируемая
- Для больших директорий используется HTree

Во времена ext2 была проблема: считалось, что жесткие диски живут долго и работают безотказно, однако на самом деле это было не так. При записи файла, например, могла произойти ошибка на жестком диске, и из-за этого ext2 ломалась. По этой причине в ext3 сделали журнал. Журнал – это область на диске, в которую записываются логи всех операций, которые выполняются с данными. В ext3 в журнал записываются блоки, которые мы меняем на диске. Если в процессе изменения структуры данных файловой системы случился сбой диска, то в логи не будет записан маркер конца транзакции изменения файловой системы. И когда мы будем восстанавливать файловую систему по журналу, мы просто применим все записанные транзакции и получим какое-то её стабильное состояние. Также у нас есть возможность обратить изменения на диске. Это нам гарантирует, что если посреди операции возникла ошибка, мы не сломаем все данные, и сможем всё восстановить. Журналируемость можно отключить, благодаря этому файловая система будет работать быстрее.

Еще одно отличие ext2 и ext3: есть директории, в которых находятся очень много файлов. Из-за этого поиск в таких директориях работает долго: нужно пробежаться по всем записям в inode и сравнить названия файлов с тем, что мы ищем. Поэтому в ext3 (и в ext4 соответственно) была создана структура данных HTree, которая хранит кэши файловых имен. Благодаря этому можно за амортизированное  $O(1)$  искать файлы в директориях. Использование HTree тоже можно отключить.

### 3.2.3 Другие файловые системы

- FAT32
- NTFS (проприетарная, используется в Windows)
- ReiserFS (оптимизирует работу с большим количеством маленьких файлов)
- ZFS (может работать с несколькими устройствами)

### 3.2.4 sysfs и procfs

- “Метафайловые системы”
- Не имеют никаких данных на диске, возвращают информацию напрямую из ядра Linux
- Часто используются, чтобы не добавлять новые сисколла

### 3.2.5 FUSE

- Код файловой системы обычно расположен в ядре – это неудобно
- FUSE = file system in userspace

Когда мы хотим поменять файл, мы сообщаем об этом операционной системе, а та, видя, что мы обращаемся к FUSE-файловой системе, передаст этот запрос процессу файловой системы.

Пример использования: можно работать с файлами на удаленной машине, как будто они находятся локально на нашем компьютере. В таком случае все изменения файловой системы будут пересылаться по сети. Это можно сделать при помощи SSHFS.

## 3.3 Файловые дескрипторы

- Преобразование имени файла в inode – очень дорогая операция, которая может требовать много обращений к диску
- Этот процесс “кэшируют” с помощью файловых дескрипторов
- Файловый дескриптор – число больше 0
- Новый файловый дескриптор будет минимальным доступным числом

Благодаря файловым дескрипторам можно только один раз делать преобразование имени файла в inode (делая сисколл open). При этом привязка произойдет именно к inode, а не к имени файла.

- За каждым файловым дескриптором скрывается [специальная структура](#) в ядре
- Указатель на inode, позиция в файле, флаги (чтения/запись/блокирование), различные локи и так далее

### 3.3.1 Работа с данными файла

```
1 #include <unistd.h>
2
3 int open(const char* pathname, int flags, mode_t mode);
4 ssize_t read(int fd, void* buf, size_t count);
5 ssize_t write(int fd, const void* buf, size_t count);
6 int close(int fd);
```

### 3.3.2 Работа с метаданными файла

```
1 #include <sys/stat.h>
2
3 int stat(const char* path, struct stat* buf);
4 int fstat(int fd, struct stat* statbuf);
5 int lstat(const char* pathname, struct stat* statbuf);
6
7 struct stat {
8     dev_t      st_dev;
9     ino_t      st_ino;
10    mode_t      st_mode;
11    nlink_t     st_nlink;
12    uid_t       st_uid;
13    gid_t       st_gid;
14    dev_t       st_rdev;
15    off_t       st_size;
16    blksize_t   st_blksize;
17    blkcnt_t    st_blocks;
18    struct timespec st_atime/st_mtime/st_ctime;
19 };
```

### 3.3.3 Права доступа

- rwx = Read/Write/Execute
- 9 бит, 3 группы; права владельца, права группы и права для остальных
- Часто записываются как числа в восьмиричной системе счисления

- $777_8 = 11111111_2 = \text{rwxrwxrwx}$
- $664_8 = 110100100_2 = \text{rw-r--r--}$

### 3.3.4 Права доступа для директорий

- `r` – листинг директории
- `w` – создание файлов внутри директории
- `x` – возможность перейти в директорию (`cd`), а также доступ к файлам

### 3.3.5 Регулярные файлы

- `S_ISREG(stat.st_mode)`
- Обычные файлы с данными

### 3.3.6 Директории

- `S_ISDIR(stat.st_mode)`
- Специальный API для чтения, обычные `read/write` не работают
- Создание и удаление: `mkdir/rmdir`

```
1 #include <dirent.h>
2
3 struct dirent* readdir(DIR* dirp);
4
5 struct dirent {
6     ino_t      d_ino;
7     off_t      d_off;
8     unsigned short d_reclen;
9     unsigned char d_type;
10    char        d_name[256];
11 };
12
13 int closedir(DIR* dirp);
```

### 3.3.7 Символические ссылки

- `S_ISLINK(stat.st_mode)`
- Аналог `std::weak_ptr` для `inode`
- Могут быть `dangling`: то есть ссылаться на файл, которого нет
- Отдельный тип файла
- Путь, на который она ссылается, записан в блоках

### 3.3.8 Жесткие ссылки

- Аналог `std::shared_ptr` для `inode`
- Только внутри одной файловой системы
- Если количество жестких ссылок стало равно 0, то `inode` становится свободной
- Не файл, а сущность файловой системы

### 3.3.9 Символьные устройства (`character device`)

- `S_ISCHR(stat.st_mode)`
- Устройства, из которых можно последовательно читать
- Клавиатура, звуковая карта, сетевая карта
- Такие файлы создаются драйверами ядра

### 3.3.10 Блочные устройства (`block device`)

- `S_ISBLK(stat.st_mode)`
- Разбиты на блоки одинакового размера
- Можно прочитать любой блок
- HDD, SSD, NAS



## 4 Память

### 4.1 Виртуальная память

Есть 2 проблемы: нужно выделить каждому процессу память, при этом так, чтобы процессы не могли смотреть в память друг друга. Вторая проблема: есть ассемблерные инструкции, которые прыгают в какие-то места памяти, при этом программа должна правильно работать независимо от того, в каком месте памяти она была запущена.

#### 4.1.1 Сегментная адресация

- Память делится на куски разного размера – сегменты
- За каждым процессом закрепляются несколько сегментов: сегмент с кодом, сегмент с данными, сегмент со стеком и так далее
- У такого подхода есть проблема: фрагментация памяти. В какой-то момент может быть ситуация, когда каждый второй байт занят и программа требует половину памяти; несмотря на то, что в реальности 50% памяти свободно, выделить её невозможно

#### 4.1.2 Страничная адресация

- Вся физическая память делится на **фреймы** – куски равного размера (4096 байт на x86)
- Каждому процессу выделяется своё *адресное пространство* или *виртуальная память*
- Виртуальная память делится на **страницы** аналогично фреймам
- Каждой странице в адресном пространстве может соответствовать какой-то фрейм

Мы будем использовать термин **страница** для виртуальной памяти, а термин **фрейм** – для физической памяти.

- Как хранить отображения страниц во фреймы?
- Всего существует  $\frac{2^{64}}{2^{12}} = 2^{52}$  страниц памяти
- Если каждая страница описывается 8 байтами, то потребуется  $2^{60}$  байт в памяти
- Нужен более экономный способ хранить это отображение

### 4.1.3 Multi-level page tables

- Идея: давайте сделаем таблицы многоуровневыми – сначала поделим всё пространство на части, каждую из этих частей еще на части и так далее
- Не храня лишние “дыры” мы будем экономить место
- Под x86 используются четырёхуровневые таблицы: P4, P3, P2, P1. Встречаются и пятиуровневые, но они сейчас мало распространены.
- Каждая таблица занимает ровно 4096 байт, содержит ровно 512 байт ( $PTE = page\ table\ entry$ ) по 8 байт и находится в начале страницы
- Каждая запись ссылается на начало следующей таблицы, последняя таблица ссылается на адрес фрейма

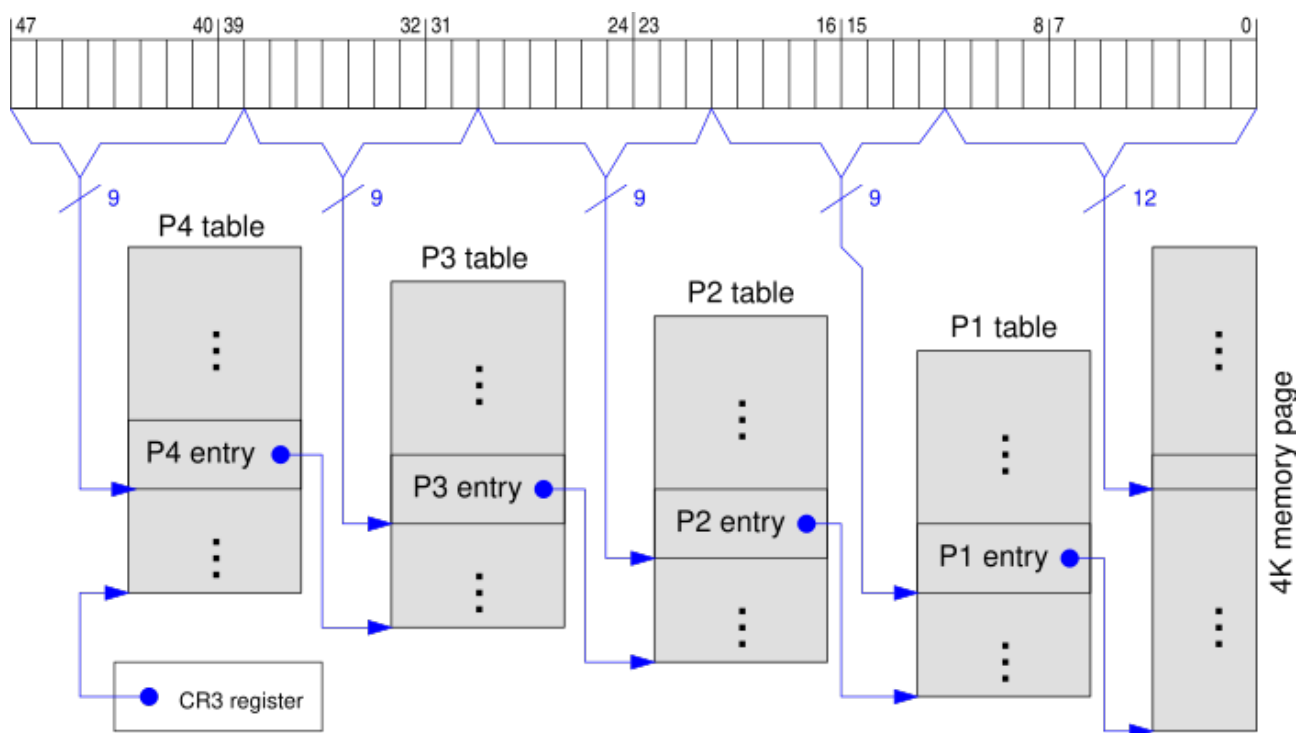


Рис. 5: Multi-level page table

### 4.1.4 Что хранится в PTE?

- Индексация следующей таблицы или фрейма не занимает все 8 байт PTE
- Кроме неё в PTE есть еще специальные *флаги страниц*
- Например, 1й бит отвечает за то, будет ли страница доступна на запись

- 63й – за то, будет ли процессор исполнять код на этой странице
- Также в некоторые биты процессор сам пишет флаги, например, dirty-бит устанавливается всегда, когда происходит запись в страницу
- Флаги имеют иерархическую видимость: если в P2 writeable-бит равен 0, а в P4 – 1, то страница будет доступна на запись

#### 4.1.5 Устройство виртуального адреса

- На текущий момент x86-64 позволяет адресовать 48 бит физической памяти
- Старшие биты (с 48 по 63) должны быть sign extended копиями 47го бита
- Следующие биты (с 38 по 47) адресуют PTE в P4
- Биты с 29 по 37 адресуют PTE в P3
- Биты с 21 по 28 адресуют PTE в P2
- Биты с 12 по 20 адресуют PTE в P1, которая ссылается непосредственно на фрейм
- Биты с 0 по 11 адресуют смещение внутри фрейма

#### 4.1.6 ОС и таблицы страниц

- Операционная система хранит таблицы страниц для каждого процесса
- Таблица страниц сменяется каждый раз, когда процессор переходит в другой поток
- В реальности каждое обращение к памяти не вызывает прыжки по таблицам, оно кэшируется в TLB (translation lookaside buffer)
- При переключении процесса TLB полностью сбрасывается (с оговорками)

#### 4.1.7 Выделение памяти: on-demand paging

- Обычно современные ОС не выделяют всю запрошенную память сразу
- Page fault – ситуация, когда нет запрашиваемой страницы в текущей таблице страниц
- Идея состоит в том, чтобы детектировать с помощью page fault'ов реальные обращения к памяти и только тогда её выделять

#### 4.1.8 Minor page fault

- Кроме самих таблиц ОС обычно хранят свои отображения, запрошенные пользователем
- В Linux такие отображения называются VMA = virtual memore area
- Во время выделения памяти, ядро создаёт новый VMA
- При первом обращении происходит page fault, ядро выделяет фрейм и добавляет его в таблицу страниц
- Такой PF называют минорным (minor page fault)

#### 4.1.9 File memory mapping

- Кроме выделения памяти POSIX позволяет мапить файлы в память
- Можно указать файл, оффсет в нём и адрес памяти
- По этому адресу памяти в текущем пространстве будет лежать (изменяемая) копия файла
- Изменения в других процессах будут сразу отображены в память

#### 4.1.10 Major page faults

- За страницами, за которыми закреплён файл, скрывается механизм, который называется page caching
- Для них тоже используется on-demand paging: при первом обращении генерируется page fault, ядро перехватывает исключение, читает с диска файл и копирует его в память
- Такой PF называют мажорным (major page fault)

#### 4.1.11 Page cache

- Страницы с данными файла из всех процессов ссылаются на один и тот же фрейм
- Поэтому изменения файлов (в том числе через write) видны во всей ОС сразу
- Однако, write не гарантирует, что данные были записаны на диск

## 4.2 fsync

```
1 int fsync(int fd);
```

Сконструировать оборудование так, чтобы оно точно записало что-то на диск, сложно (если не невозможно). Иногда бывает, что жесткие диски даже не предоставляют интерфейса для синхронизации данных. Поэтому **fsync** гарантирует, что данные дойдут до диска, но рассчитывать, что они точно запишутся, нельзя. Например, может произойти сбой в работе диска, и информация не будет записана. Используются различные способы предотвращения такого поведения. Например, на Mac-ах есть резервный источник питания, который используется для того, чтобы при выключении компьютера все данные сначала записались на диск (используя этот источник), и только потом произошло выключение. Похожие техники используются на серверах.

## 4.3 mmap и munmap

```
1 void* mmap(void* addr, size_t length, int prot, int flags,  
2           int fd, off_t offset);  
3 int munmap(void* addr, size_t length);  
4 int mprotect(void* addr, size_t len, int prot);
```

### 4.3.1 mmap: prot

- PROT\_EXEC – процессор сможет выполнять код на этой странице
- PROT\_READ – страницу будет доступна на чтения
- PROT\_WRITE – страница будет доступна на запись
- PROT\_NONE – к странице никак нельзя будет обратиться

### 4.3.2 mmap: flags

- MAP\_ANONYMOUS – определяет, что область будет анонимной, `fd == -1`
- MAP\_SHARED – определяет, что область будет доступна детям текущего процесса
- MAP\_FIXED – говорит ядру использовать *в точности* адрес `addr` или вернуть ошибку

- MAP\_POPULATE – говорит ядру сразу выделить физическую память для этой области (не будет использован механизм on-demand paging  $\Rightarrow$  не будет minor fault-ов)
- Есть еще много флагов

#### 4.4 Псевдофайлы для контроля расхода памяти

- /proc/<pid>/maps хранит текущие VMA
- /proc/<pid>/status содержит статус процесса, есть куча информации о памяти
- /proc/<pid>/mem представляет собой память процесса (её можно читать и писать)
- /proc/<pid>/map\_files хранит список файлов, которые замappлены в процесс

#### 4.5 Вытеснение страниц и swap

- Если системе не хватает физической памяти для хранения анонимных страниц, она начинает их сбрасывать на диск
- Вытеснение анонимных страниц происходит в специальный swap файл или раздел диска (файл подкачки)
- Обычно это никак не заметно на приложениях, однако в условиях memory pressure это может приводить к странным последствиям

### 5 Процессы

- POSIX: “A process is an abstraction that represents an executing program. Multiple processes execute independently and have separate address spaces. Processes can create, interrupt, and terminate other processes, subject to security restrictions”
- В самом ядре Linux нет понятия “процесс”, вместо этого оно оперирует “задачами”
- То, что в POSIX процесс, в Linux называется thread group
- Процессы объединяются в *группы процессов*
- Группы процессов объединяются в *сессии*

## 5.1 Атрибуты процесса

- Сохранённый контекст процессора (регистры)
- Виртуальная память (анонимные, private/shared, файловые)
- Файловые дескрипторы
- Current working directory (cwd)
- Текущий корень (`man 2 chroot`)
- `umask`
- PID, PPID, TID, TGID, PGID, SID
- Resource limits
- Priority
- Capabilities
- Namespaces

## 5.2 Состояния процессов

Runnable – значит, что процесс прямо сейчас готов исполнять инструкции. Uninterruptible sleep – состояние, пока процесс ждет ответа от операционной системы (например, выделяет память) Interruptible sleep – состояние, когда процесс “спит”. Например, мы вызвали `sleep` Zombie – это особое состояние, про которое мы поговорим подробнее позже Stopped – состояние, часто используемое, например, в дебаггерах.

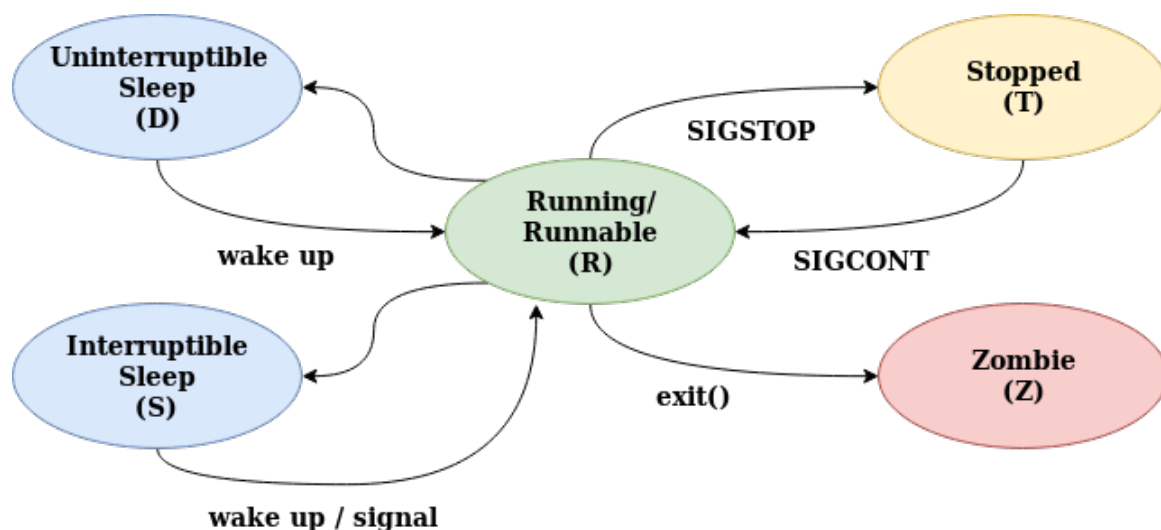


Рис. 6: Task states

## 5.3 fork

- Создать новый процесс можно только *скопировав* текущий с помощью `pid_t fork()`
- `fork` выйдет в двух процессах одновременно, но в одном вернёт PID ребенка, а в другом – 0.
- Ребенок будет полностью идентичен родителю, но файловые дескрипторы и адресное пространство будут *скопированы*
- Для оптимизации потребления памяти используется copy-on-write подход для копирования памяти

```
1 pid_t pid = fork();
2 if (pid < 0) {
3     // error! can't create new process
4 } else if (pid == 0) {
5     // We're in fork's child
6     // getpid() returns current pid
7 } else {
8     // We're in fork's parent and pid is the child's pid
9 }
```

Есть системный вызов `getpid`, возвращающий PID текущего процесса.

## 5.4 execve

- Создавать копии недостаточно, нужно уметь запускать произвольные файлы
- Для этого используется системный вызов `execve`
- Он заменяет текущий процесс процессом, созданным из указанного файла
- Это называется заменой образа процесса: заменяются только части адресного пространства
- Также в новом образе процесса останутся незакрытые файловые дескрипторы, не помеченные флагов `O_CLOEXEC`



```
1 extern char** environ;
2 int execl(const char* path, const char* arg0, ...);
3 int execlp(const char* path, const char* arg0, ...);
4 int execlp(const char* file, const char* arg0, ...);
5 int execv(const char* path, char* const argv[]);
6 int execve(const char* path, char* const argv[],
7             char* const envp[]); // system call!
8 int execvp(const char* file, char* const argv[]);
9 int fexecve(int fd, char* const argv[], char* const envp[]);
```

#### 5.4.1 execve: сохраняемые атрибуты

- В отличие от `fork` сохраняет меньше атрибутов
- Файловые дескрипторы (не помеченные флагом `O_CLOEXEC`)
- `cwd` и `root`

```
1 pid_t pid = fork();
2 if (pid == -1) {
3
4 } else if (pid == 0) {
5     char* argv[] = {"ls", "-lah", NULL};
6     char* envp[] = {"FOO=bar", "XYZ=abc", NULL};
7     execve("/usr/bin/ls", argv, envp);
8     // if you ended up here, something went wrong!
9 } else {
10
11 }
```

### 5.5 Атрибуты процесса: PID, PPID, TGID, ...

- PID = process ID
- PPID = parent process ID
- PGID = process group ID
- SID = session ID

- `pid_t` `getpid()`, `pid_t`, `getpid()`, `pid_t` `gettid()`
- `pid_t` `getpgid()`, `int` `setpgid(pid_t pid, pid_t pgid)`
- `pid_t` `setid()`, `pid_t` `getsid(pid_t pid)`
- `/proc/<pid>/status` ИЛИ В `/proc/<pid>/stat`

## 5.6 Атрибуты владельца процесса

- UID (user ID или real user ID) – ID владельца процесса, `void` `setuid(uid_t)`
- EUID (effective user ID) используется для проверок доступа, `seteuid(uid_t)`
- SUID (saved user ID) используется, чтобы можно было временно понизить привилегии
- FSUID (file system user ID) обычно совпадает с EUID, но может быть отдельно изменен через `int` `setfsuid(uid_t fsuid)`
- Непривилегированный процесс может выставить EUID равный только в SUID, UID или опять в EUID
- Также есть понятие `setuid/setgid` флагов (sticky flags), в отличие от обычных файлов, EUID такого процесса будет выставлен как UID *владельца файла*, а не *текущий пользователь*
- Есть аналогичные GID, EGID, SGID, FSGID

## 5.7 Работа с процессами

### 5.7.1 `exit`

- Завершает текущий процесс с определенным *кодом возврата* (exit code)
- `exit` vs. `_exit`
- Чтобы завершить текущую thread group можно воспользоваться `exit_group`
- `exit` закрывает все открытые файловые дескрипторы, освобождает выделенные страницы, etc
- Если у процесса были дети, то их родителем станет процесс с `PID == 1`
- После этого процесс становится *зомби-процессом*
- Ядро не хранит огромную структуру для него, а только его PID и exit code

### 5.7.2 wait

- Дождидается, пока процесс будет остановлен
- Для этого используются системные вызовы семейства `wait*`
- Они дожидаются завершения процесса (конкретного или любого) и возвращают специальный *exit status*
- Обычно *exit status* содержит то, что передали в `exit`

```
1 pid_t wait(int* stat_loc);
2 pid_t waitpid(pid_t pid, int* stat_loc, int options);
3 pid_t wait3(int* stat_loc, int options, struct rusage* rusage);
4 pid_t wait4(pid_t pid, int* stat_loc, int options,
5             struct rusage* rusage); // system call!
```

### 5.8 wait4

- `pid < 1`: ждёт любой дочерний процесс в группе процессов `-pid`
- `pid == -1`: ждёт любой дочерний процесс
- `pid == 0`: ждёт любой дочерний процесс в текущей группе процессов
- `pid > 0`: ждёт конкретного ребенка

#### 5.8.1 Макросы для wait4

```
1 WIFEXITED(status) // did the process terminate itself?
2 WEXITSTATUS(status) // get exit status
3 WIFSIGNALED(status) // did the process terminate by a signal?
4 WTERMSIG(status) // get the signal
5 WCOREDUMP(status) // did the process leave a coredump?
```

### 5.9 Лимит ресурсов процессов

- Лимиты делятся на два типа: `soft` и `hard`
- `Soft`-лимит или текущий лимит – по нему вычисляются проверки

- Hard-лимит – максимальное значение soft-лимита
- CPU-время, если процесс его превысит ему ядро отошлет **SIGXCP**, если превысит hard-лимит, то **SIGKILL**
- Размер записываемых файлов: write будет возвращать **EFBIG**
- Размер записываемых coredump-файлов
- На максимальный размер виртуальной памяти, выделенной процессу
- Количество одновременных процессов пользователя
- Количество файловых дескрипторов

```
1 #include <sys/time.h>
2 #include <sys/resource.h>
3
4 struct rlimit {
5     rlim_t rlim_cur; /* Soft limit */
6     rlim_t rlim_max; /* Hard limit (ceiling for rlim_cur) */
7 };
8
9 int getrlimit(int resource, struct rlimit* rlim);
10 int setrlimit(int resource, const struct rlimit* rlim);
```

## 5.10 Механизмы изоляции

- `man 7 namespaces` и `man 7 cgroups`
- Linux namespaces изолируют части отдельных процессов
- CGroups (control groups) обычно ограничивают потребляемое процессорное время и память
- Существующие неймспейсы: cgroup, IPC, network, mount, PID, time, UTS

## 5.11 ELF

- Executable & Linkable Format
- Исполняемый формат файлов для Linux
- Поддерживает разные архитектуры и битности
- [Спецификация](#)
- [Структуры внутри ядра](#)

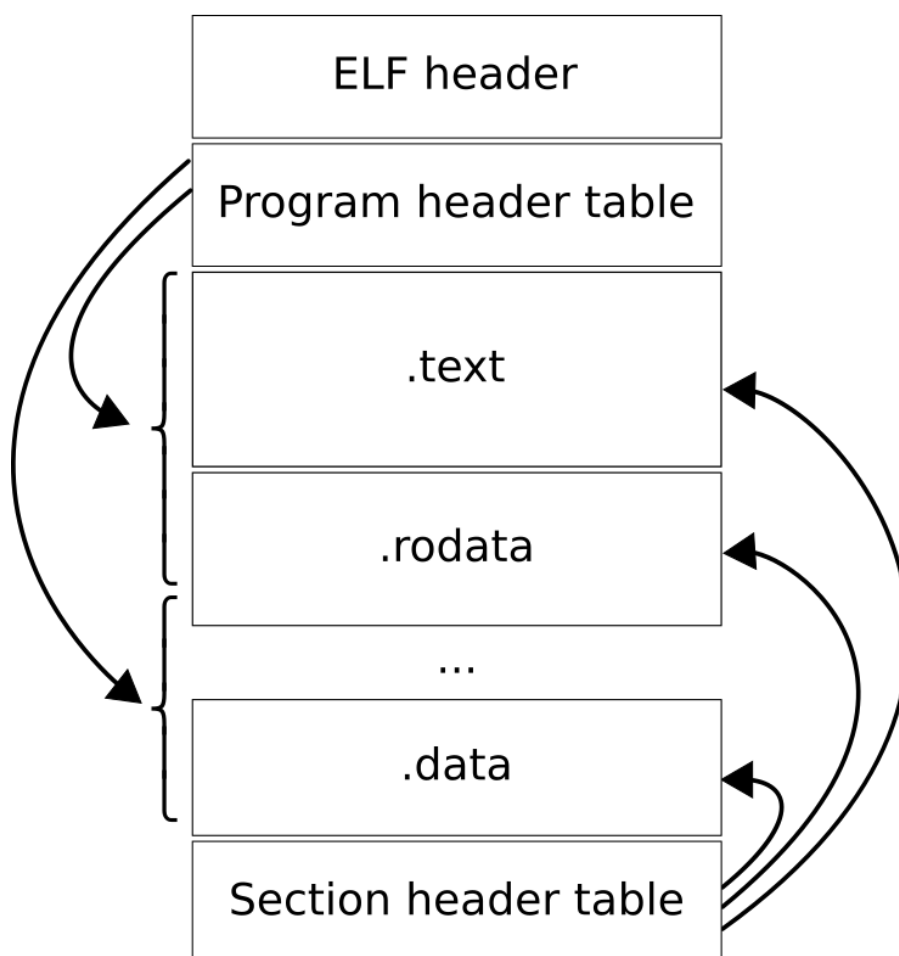


Рис. 7: ELF structure

### 5.11.1 ELF: header

```
1 typedef struct elf64_hdr {
2     unsigned char e_ident[EI_NIDENT]; /* ELF "magic number" */
3     Elf64_Half e_type;
4     Elf64_Half e_machine;
5     Elf64_Word e_version;
6     Elf64_Addr e_entry; /* Entry point virtual address */
7     Elf64_Off e_phoff; /* Program header table file offset */
8     Elf64_Off e_shoff; /* Section header table file offset */
9     Elf64_Word e_flags;
10    Elf64_Half e_ehsize;
11    Elf64_Half e_phentsize;
12    Elf64_Half e_phnum;
13    Elf64_Half e_shentsize;
14    Elf64_Half e_shnum;
15    Elf64_Half e_shstrndx;
16 } Elf64_Ehdr;
```

### 5.11.2 ELF: секции

- Секции хранят непосредственно данные
- Каждая секция имеет своё имя
- Перечень всех секций хранит *таблица секций*
- `.data` – данные
- `.text` – исполняемый код
- `.rodata` – read-only данные
- `.symtab` – таблица символов
- `.strtab` – таблица строк
- `.shstrtab` – таблица строк с названием секций
- `.rel/.rela` – таблица релокаций

### 5.11.3 ELF: .symtab

```
1 typedef struct {
2     uint32_t      st_name;
3     unsigned char st_info;
4     unsigned char st_other;
5     uint16_t      st_shndx;
6     Elf64_Addr    st_value;
7     uint64_t      st_size;
8 } Elf64_Sym;
```

### 5.11.4 ELF: сегменты

- Все секции объединяются в *сегменты*
- Каждый сегмент имеет адрес, по которому он загружается
- Program header array содержит все сегменты, располагается в начале файла

### 5.11.5 ELF: program header

```
1 typedef struct elf64_phdr {
2     Elf64_Word p_type;
3     Elf64_Word p_flags;
4     Elf64_Off  p_offset;    /* Segment file offset */
5     Elf64_Addr p_vaddr;    /* Segment virtual address */
6     Elf64_Addr p_paddr;    /* Segment physical address */
7     Elf64_Xword p_filesz;  /* Segment size in file */
8     Elf64_Xword p_memsz;   /* Segment size in memory */
9     Elf64_Xword p_align;   /* Segment alignment, file & memory */
10 } Elf64_Phdr;
```

Разница между `p_memsz` и `p_filesz` существует и обуславливается наличием сегмента `.bss`, в котором хранятся статические переменные, инициализируемые нулями. Такие переменные не хранятся на диске, но при запуске программы место под них отводится. Поэтому `p_filesz` – это размер на диске, а `p_memsz` – размер загружаемого файла.

## 5.12 ехесве: принципы работы

- Парсит первые несколько байт файла

- Ищет ELF magic или shebang
- Загружает образ (мmap'ит)
- Подготавливает окружение для старта процесса (стек, переменные окружение, etc)
- Запускает инструкцию по адресу `e_entry`

## 6 Linux scheduler

### 6.1 Realtime scheduling

- У каждого процесса есть собственный realtime приоритет
- Планировщик ищет процесс с наибольшим приоритетом и запускает его, пока он не выйдет
- Вытеснения нет!
- Round-robin для процессов с одним приоритетом

### 6.2 Process niceness

- Значения от -20 до +19
- Процесс с меньшим niceness должен получать больше процессорного времени

#### 6.2.1 Timeslice

- Квант процессорного времени
- В зависимости от типа системы от 1мс до 10-20мс
- CPU bound процессы: в основном выполняют инструкции
- I/O bound процессы: в основном спят

#### 6.2.2 Timeslice & niceness

- Зададим каждому приоритету определённый timeslice (например, 0 = 100ms, +20 = 5ms)
- Процессы с меньшим NI получают больше процессорного времени



### 6.2.3 Timeslice & niceness: проблемы подхода

- Два процесса с  $NI = +19$  и два процесса с  $NI = 0$
- Первые получают по 5ms процессорного времени
- Вторые по 100ms
- В обоих случаях 50% CPU, но переключение контекста у первых происходит чаще

И другие проблемы:

- Два процесса  $NI = 0$  и  $NI = 1$  vs. два процесса  $NI = +18$  и  $NI = +19$
- 5ms/10ms vs. 95ms/105ms
- Нелинейность относительно стартового значения

## 6.3 completely Fair Scheduler

- Эмулирует идеальный multitasking процессор
- Идея – раздавать процессам время пропорциональное  $1/N$ , где  $N$  – количество запущенных процессов
- Каждому процессу начисляется **vruntime** – количество затраченного CPU времени за определенный интервал времени
- Очередным берётся процесс с наименьшим **vruntime**

### 6.3.1 CFS: проблемы

- Минимальное время (гранулярность)  $\Rightarrow$  чем больше процессов, тем менее честным становится CFS
- Context switch считается zero-time операцией, но по факту это не так

## 7 Сигналы

- Асинхронное событие, которое может произойти после любой инструкции
- Используются для уведомления процессов о каких-то событиях
- Сигнал можно либо обрабатывать, либо игнорировать
- Однако SIGKILL и SIGSTOP нельзя обработать или проигнорировать

## 7.1 Сигналы: примеры

- Нажатие `^C` (Ctrl+C) в терминале генерирует `SIGINT`
- Нажатие `^\` (Ctrl+Backslash) в терминале генерирует `SIGQUIT`
- Запись в пайп только с write-концом генерирует `SIGPIPE`
- вызов `abort()` приводит к `SIGABORT`
- Обращение к несуществующей памяти генерирует `SIGSEGV`

## 7.2 Доставка сигналов

- Сигналы могут быть доставлены от ядра (например, `SIGKILL`, `SIGPIPE`)
- Либо от другого процесса (`SIGHUP`, `SIGINT`, `SIGUSR`)
- Или процесс может послать сигнал сам себе (`SIGABRT`)
- Сигналы могут быть доставлены в *любой момент* выполнения программы

## 7.3 Signal safety

- Во время обработки сигналы процессы могут быть в критической секции
- Поэтому в обработчиках сигналов нельзя использовать, например, `printf`
- Можно использовать только `async-signal-safe` функции
- `man 7 signal-safety`

## 7.4 Обработка сигналов

```
1 void signal_handler(int sig) {  
2     // ...  
3 }  
4  
5 int main() {  
6     signal(SIGINT, signal_handler);  
7     signal(SIGTERM, signal_handler);  
8     signal(SIGSEGV, SIG_IGN);  
9     signal(SIGABRT, SIG_DFL);  
10 }
```

## 7.5 Посылка сигналов

```
1 #include <signal.h>
2
3 int raise(int sig);
4 int kill(pid_t pid, int sig);
```

Название `kill` существует по историческим причинам: раньше был только один сигнал — `SIGKILL`.

### 7.5.1 Посылка сигналов: аргументы `kill`

- Если `pid == 0`, то сигнал будет доставлен текущей группе процессов
- Если `pid > 0`, то сигнал будет доставлен процессу `pid`
- Если `pid == -1`, то сигнал будет всем процессам, которым текущий процесс может его отправить
- Если `pid < -1`, то сигнал будет доставлен группе процессов `-pid`
- Если `sig == 0`, то сигнал не будет никому отправлен, а будет только осуществлена проверка ошибок (dry run)
- Возврат из `kill` не гарантирует, что сигнал обработался в получателе(-ях)!

## 7.6 Доставка сигналов: маски сигналов

- Маска сигналов — `bitset` всех сигналов
- У процесса есть две маски сигналов: *pending* и *blocked*
- *pending* — это те сигналы, которые должны быть доставлены, но ещё не успели
- Из этого следует, что если несколько раз отправить сигнал в один процесс, то он может быть обработан лишь единожды
- *blocked* — это те сигналы, которые процесс блокирует (блокировать и игнорировать — разные вещи)
- Если сигнал заблокирован, это значит, что он не будет доставлен вообще, если проигнорирован — то у него просто пустой обработчик

```
1 #include <signal.h>
2
3 int sigemptyset(sigset_t* set);
4 int sigfillset(sigset_t* set);
5 int sigaddset(sigset_t* set, int signum);
6 int sigdelset(sigset_t* set, int signum);
7 int sigismember(const sigset_t* set, int signum);
```

### 7.6.1 Доставка сигналов: sigprocmask

- `int sigprocmask(int h, sigset_t *set, sigset_t *oset);`
- `SIG_SETMASK` — установить маску заблокированных сигналов
- `SIG_BLOCK` — добавить сигналы `set` в заблокированные
- `SIG_UNBLOCK` — удалить сигналы `set` из заблокированных
- Если `oset != NULL`, то туда будет записана предыдущая маска

## 7.7 Обработка сигналов: sigaction

```
1 #include <signal.h>
2
3 int sigaction(int signum, const struct sigaction* act,
4               struct sigaction* oldact);
5
6 struct sigaction {
7     void (*sa_handler)(int);
8     void (*sa_sigaction)(int, siginfo_t*, void*);
9     sigset_t sa_mask;
10    int sa_flags;
11 };
```

- Выставляет обычный обработчик `sa_handler`
- Или расширенный: `sa_sigaction`
- При выполнении сигнала `signum` в заблокированные сигналы добавятся сигналы из `sa_mask`, а также сам сигнал

- `sa_flags` — флаги, меняющие поведение обработки
- Чтобы использовать `sa_sigaction`, нужно выставить `SA_SIGINFO`
- Если выставить `SA_RESETHAND`, то обработчик сигнала будет сброшен на дефолтный после выполнения
- Если выставить `SA_NODEFER`, то если сигнал не был в `sa_mask`, обработчик может быть прерван самим собой

### 7.7.1 Обработка сигналов: `siginfo_t`

```
1 #include <signal.h>
2
3 siginfo_t {
4     int      si_signo;      int      si_overrun;
5     int      si_errno;      int      si_timerid;
6     int      si_code;       void*    si_addr;
7     int      si_trapno;     long     si_band;
8     pid_t    si_pid;        int      si_fd;
9     uid_t    si_uid;        short   si_addr_lsb;
10    int      si_status;      void*    si_lower;
11    clock_t   si_utime;      void*    si_upper;
12    clock_t   si_stime;      int      si_pkey;
13    sigval_t  si_value;      void*    si_call_addr;
14    int      si_int;         int      si_syscall;
15    void*     si_ptr;        unsigned int si_arch;
16 }
```


## 7.8 SIGCHLD

- Еще один способ уведомлять процессы о завершении дочерних
- `si_status` хранит информацию о том, как завершился процесс (`CLD_KILLED`, `CLD_STOPPED`, etc)
- `si_pid` хранит PID дочернего процесса
- Если выставить `SIG_IGN`, то зомби не будут появляться

## 7.9 Доставка сигналов во время системных вызовов

- Если сигнал прервал выполнение блокирующего сисколла, то есть два поведения
- Если использован `sigaction` и в `sa_flags` есть `SA_RESTART`, то после того, как обработчик завершится, сисколл продолжит свою работу (syscall restarting)
- Если не указан, то сисколл вернёт ошибку и `errno == EINTR`

## 7.10 Ожидание сигналов: `pause`, `sigsuspend` и `sigwaitinfo`

- `int pause(void)`
- Блокируется до первой доставки сигналов (которые не заблокированы)
- `int sigsuspend(const sigset_t* mask)`
- Атомарно заменяет маску заблокированных сигналов на `mask` и ждёт первой доставки сигналов
- `int sigwaitinfo(const sigset_t* restrict set, siginfo_t* restrict info)`
- Требуется вызова 

## 7.11 Как устроены сигналы?

- Ядро проверяет, нужно ли доставить сигнал в текущий процесс (при выходе из сисколла, или в S-состоянии, или по таймеру)
- Конструируется специальный отдельный стек (`sigaltstack`)
- В начало стека кладётся фрейм, который содержит информацию о прерванной инструкции (`siginfo_t->ucontext`)
- Ядро «прыгает» в обработчик события, выполняя его на этом стеке
- Обработчик завершается и вызывает `sigreturn`
- Ядро возвращается в предыдущий стек, инструкцию и так далее

## 7.12 Наследование сигналов: `fork` и `execve`

- `fork` сохраняет маску сигналов и назначенные обработчики
- `execve` сохраняет *только* маску заблокированных сигналов

### 7.13 Почему сигналы — это плохо?

- Почти невозможно обработать сигналы без race condition'ов
- Обработчики сигналов могут вызываться во время работы других обработчиков
- Посылка нескольких сигналов может привести к посылке только одного
- Не используйте сигналы для IPC (interprocess communication)!

Если в процессе несколько тредов, какой из них получит сигнал?

*[...] A process-directed signal may be delivered to any one of the threads that does not currently have the signal blocked. If more than one of the threads has the signal unblocked, then the kernel chooses an arbitrary thread to which to deliver the signal [...]*

### 7.14 Почему лучше всегда использовать sigaction?

- Война поведений: BSD vs System-V
- В отличие от BSD, в System-V обработчик сигнала выполняется единожды, после чего сбрасывается на обработчик по умолчанию
- В BSD обработчик сигнала не будет вызван, если в это время уже выполняется обработчик того же самого сигнала
- В BSD используется syscall restarting, в System-V — нет
- BSD (`_BSD_SOURCE` или `-std=c99`): `SA_RESTART`
- System-V (`_GNU_SOURCE` или `-std=gnu99`): `SA_NODEFER|SA_RESETHAND`
- Всегда лучше использовать `sigaction` для однозначности поведения программы!

### 7.15 Real-time signals

- При посылке сигналов учитывается их количество и порядок
- Отделены от обычных: начинаются с `SIGRTMIN`, заканчиваются `SIGRTMAX`
- Вместе с сигналом посылается специальная метainформация (правда, только число), которую можно как-то использовать

- Получить эту дополнительную информацию можно через `siginfo_t->si_value`

```
1 #include <signal.h>
2
3 union sigval {
4     int     sival_int;
5     void*   sival_ptr;
6 };
7
8 int sigqueue(pid_t pid, int signum, const union sigval value);
```

## 7.16 Обработка сигналов с помощью пайпов

- Иногда не хочется возиться с атомарными счётчиками или хочется выполнить какие-то нетривиальные действия в обработчике
- Или в обработчике нет какого-то нужного контекста (экземпляра класса итд)
- Помогает трюк с пайпами
- В обработчике будем писать номер сигнала (или какую-то другую информацию) в пайп
- В основной программе будем делать read на другой конец пайпа
- **Важно:** write-конец пайпа должен быть с флагом `O_NONBLOCKING`, иначе возможен дедлок

## 7.17 Использование сигналов

### 7.17.1 nginx

- Если поменялся конфиг nginx, то как его подхватить заново?
- Постоянный трафик от клиентов ⇒ перезапуск невозможен
- Сигналы приходят на помощь!
- `SIGHUP` сигнализирует nginx о том, что конфиг поменялся и его нужно перечитать и применить



- `SIGTERM` сигнализирует `nginx` о том, что нужно остановить обработку запросов как можно скорее и завершиться
- `SIGUSR1` используется для `hot upgrade`

### 7.17.2 `golang`

- Реализация `preemptive multitasking`
- Внутри Go реализованы лёгкие потоки – горутины (`cooperative multitasking`)
- Иногда горутины могут зависать (если, например, много вычислений) и их нужно уметь принудительно вытеснять
- Отдельный тред `sysmon`, который следит за остальными тредами, исполняющими горутины
- Если какая-то горутина зависает больше, чем на 10 мс, посылается `SIGURG` и её выполнение прерывается