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

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 зашивается вся нужная 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 multitaksing – 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

1.4.3 Мультипроцессорность

- Тактовая частота процессоров не растет примерно с 2005 года
- Поэтому современные процессоры обычно имеют несколько ядер

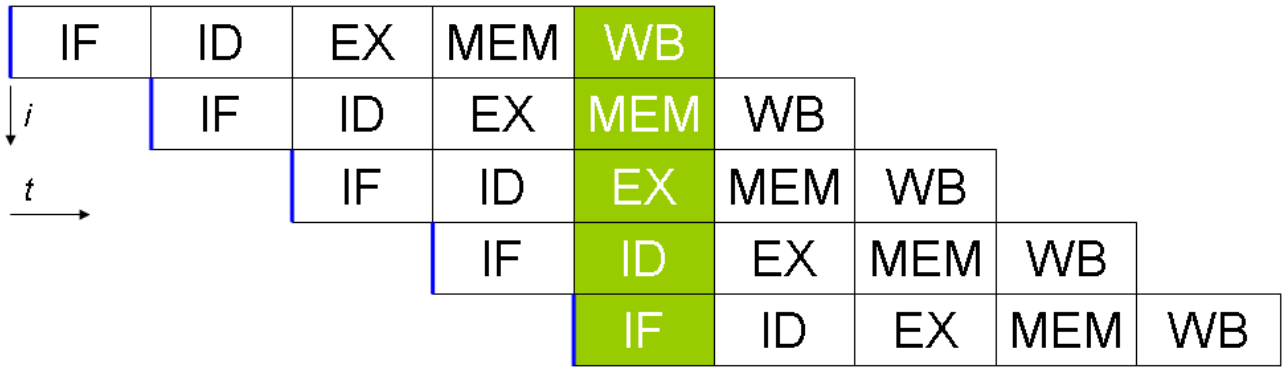

Рис. 1: CPU pipeline

- *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

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

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 Endianess

- 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 endianess:

- 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".

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

Рис. 2: Endianess

- `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 pad'ами
- Выравнивание структуры – максимальное выравнивание среди всех выравниваний её членов

2.3 Знаковые числа

2.3.1 One's complemnt

- $-A = BitwiseNot(A)$
- 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 число сопоставим отрицательное
- $-A = BitwiseNot(A) + 1$
- Одно представление нуля: $-0 = BitwiseNot(A) + 1 = 1111 + 1 = 0000 = +0$

- 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: $0010 = 00000010, 1000 = 11111000$
- "Перекос" диапазона представимых чисел
- `abs(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”: $M_{real} = 1.mmmmmm$

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, $sqrt(X)$, $\log(X)$, $X < 0$

Проблемы IEEE 754

- При вычислениях накапливается ошибка
- Сложение и умножение неассоциативно
- Умножение недистрибутивно
- NaN $\neq$ 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 (∼1.1 млн. code points)
- Code point’ы обозначаются как U+<число>
- ℵ = 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

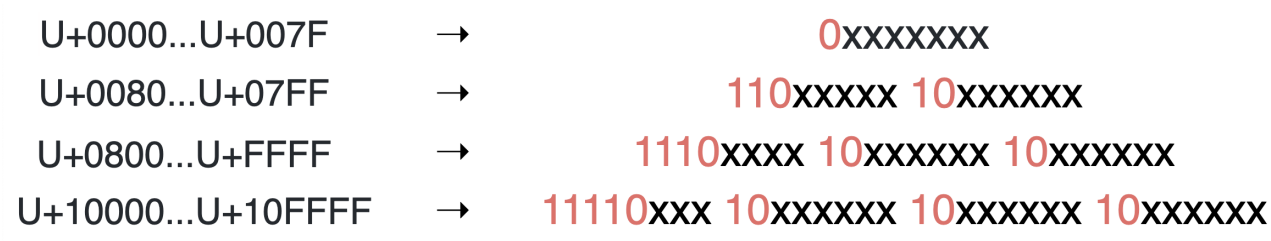

Рис. 3: UTF-8

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 метаинформацией + информация где он хранится
- Директории тоже хранятся в inode, так как директория – файл!

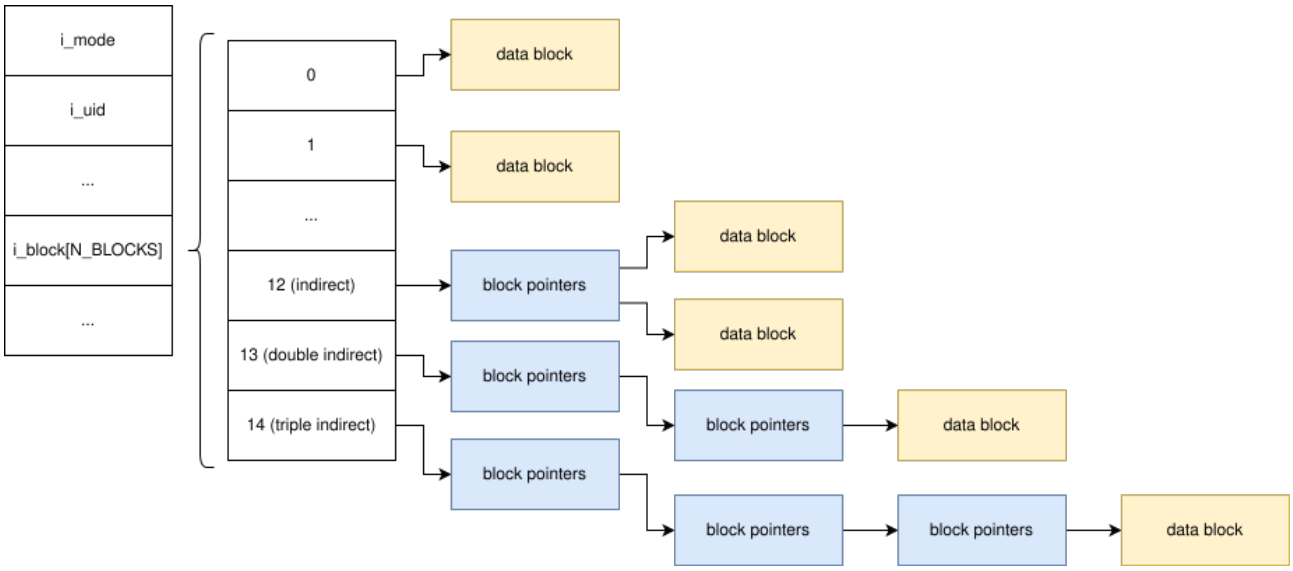

Рис. 4: inode

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 $\mathcal{O}(1)$ 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)
- ZPS (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 сисколлы

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

```c
#include <unistd.h>

int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
int close(int fd);
```

3.3.2 Работа с метаданными файла

```c
#include <sys/stat.h>

int stat(const char* path, struct stat* buf);
int fstat(int fd, struct stat *statbuf);
int lstat(const char* pathname, struct stat* statbuf);

struct stat {
  dev_t st_dev;
  ino_t st_ino;
  mode_t st_mode;
  nlink_t st_nlink;
  uid_t st_uid;
  gid_t st_gid;
  dev_t st_rdev;
  off_t st_size;
  blksize_t st_blksize;
  blkcnt_t st_blocks;
  struct timespec st_atime/st_mtime/st_ctime;
};
```

3.3.3 Права доступа

- rwx = **R**ead/**W**rite/e**X**ecute
- 9 бит, 3 группы; права владельца, права группы и права для остальных
- Часто записываются как числа в восьмиричной системе счисления
- $777_8 = 111111111_2 =$ `rwxrwxrwx`
- $664_8 = 110100100_2 =$ `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

```c
#include <dirent.h>

struct dirent *readdir(DIR *dirp);

struct dirent {
  ino_t d_ino;
  off_t d_off;
  unsigned short d_reclen;
  unsigned char d_type;
  char d_name[256];
};

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