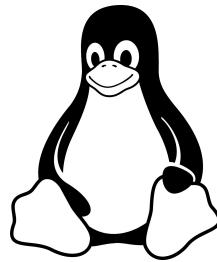
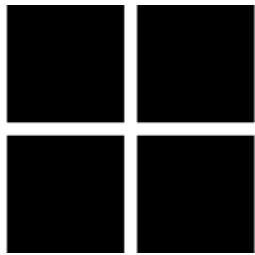
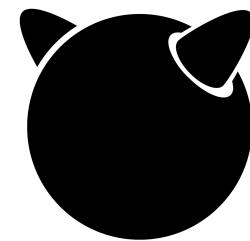


# Operating systems



macOS

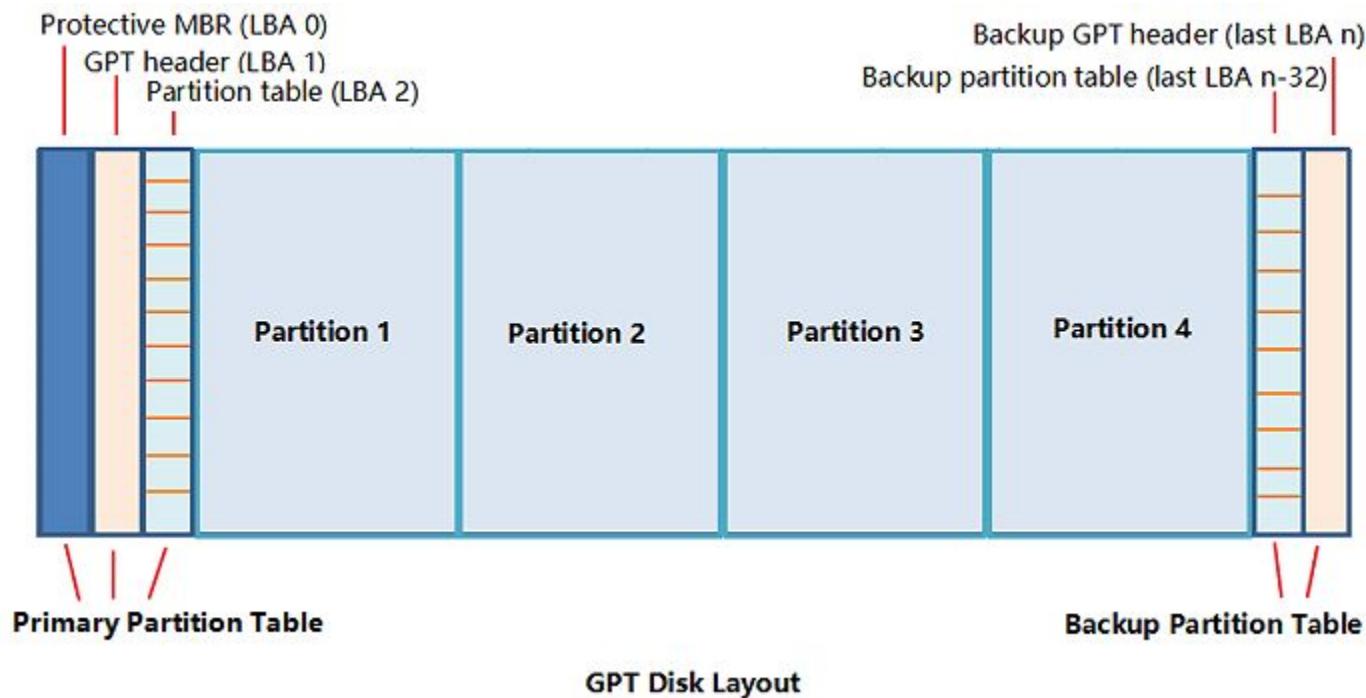


Файловые системы

# Назначение файловых систем

- Реализация единого интерфейса для построения файловой системы
  - Организация уровня абстракции от конкретной файловой системы
  - Стандартизация доступа к файловой системе на разных носителях
- 
- Носители называют **блочными устройствами**, они предоставляют работу с данными, организованными в **блоки** фиксированного размера
  - **Файловая система** предоставляет работу с этими блоками в виде **потока байтов + метаданные**

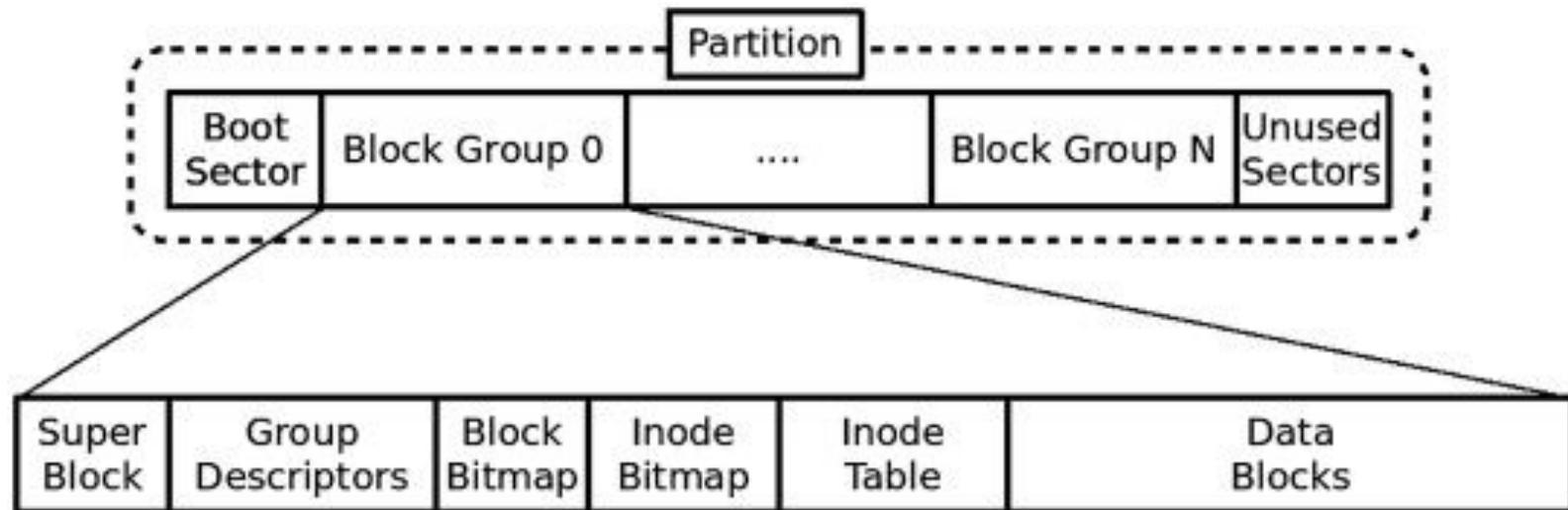
# Структура диска



# Базовые единицы

- **Logical Block Address (LBA)** – номер сектора на диске
  - Используется ОС и контроллером диска для обращения к секторам
- **Сектор** – это минимально адресуемая физическая единица чтения/записи на диске, линейка данных фиксированного размера
  - Современные диски (контроллеры) используют размер сектора в 4096 байт
  - Традиционно, размер сектора был 512 байт
- **Блок** – логическая единица на уровне ФС, является последовательной группой из нескольких секторов фиксированной длины
  - Адресация происходит по номеру блока, а не сектора или LBA
  - Размеры могут быть: 1024, 2048, 4096 байт (обычно кратно размеру сектора)

# Структура раздела на примере ext2



- Файловая система делится на группы блоков (от 8192 до 32768 блоков)

# Суперблок

- Суперблок содержит критические метаданные о всей файловой системе и необходим для ее монтирования и работы
- Содержит в себе следующую информацию:
  - Общее количество i-узлов в ФС (s\_inodes\_count)
  - Общее количество блоков (s\_blocks\_count)
  - Количество свободных блоков (s\_free\_blocks\_count)
  - Количество свободных i-узлов (s\_free\_inodes\_count)
  - Размер блока (s\_block\_size)
  - Время последнего монтирования и записи (s\_mtime и s\_wtime)
  - Состояние ФС: чистая или с ошибками (s\_state)
  - Номер первого блока данных (s\_first\_data\_block)
  - ...и прочие метаданные
- Их больше чем 1, их копии находятся в каждом нечётной группе блоков

# Описатель группы блоков

- **Дескрипторы группы блоков** содержат метаданные о расположении важных структур внутри группы блоков: таблиц i-узлов, битовых карт и т. д.
- Содержит в себе следующую информацию:
  - Номер блока, где хранится битовая карта блоков (bg\_block\_bitmap)
  - Номер блока с битовой картой i-узлов (bg\_inode\_bitmap)
  - Первый блок таблицы i-узлов в этой группе (bg\_inode\_table)
  - Количество свободных блоков данных (bg\_free\_blocks\_count)
  - Количество свободных i-узлов (bg\_free\_inodes\_count)
  - Количество использованных каталогов (bg\_used\_dirs\_count)
- Их больше чем 1, их копии находятся в каждом нечётной группе блоков

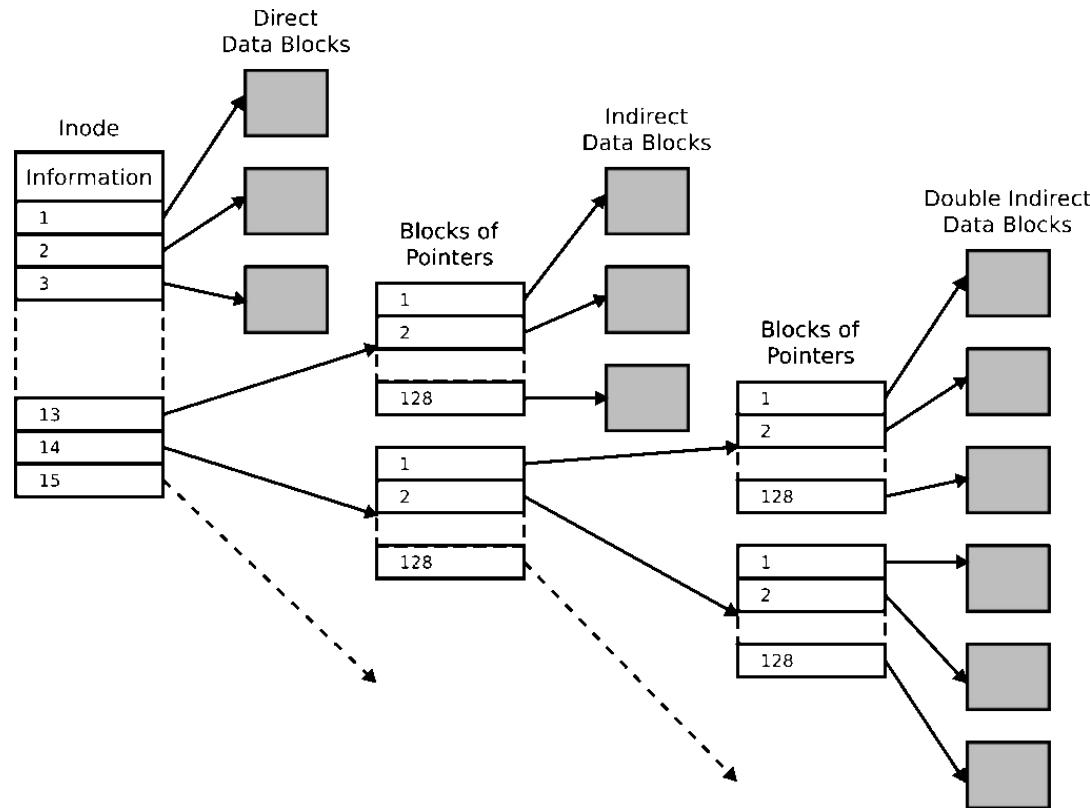
# Что за i-узлы?

- **i-node (index node)** – это структура, которая хранит все метаданные файла или каталога, кроме имени
- Что в них хранится:
  - Тип файла (обычный файл, каталог, символьическая ссылка, устройство, сокет, FIFO)
  - Владелец (UID) и группа (GID)
  - Права доступа: rwxrwxrwx (9 бит + флаги: sticky, setuid и пр.)
  - **Размер файла в байтах**
  - Количество ссылок сколько имён (hard links) указывают на этот i-узел
  - Время доступа (atime) когда файл читали
  - Время изменения контента и метаданных (mtime и ctime)
  - **Указатели на блоки данных где лежат данные файла**
  - Флаги (immutable, append-only, noatime и др.)
- Имя файла хранится в каталоге как пара (имя – номер i-узла)
- Размер в 128 байт для ext2 и 256 байт для ext4

# Как i-узел указывает на данные?

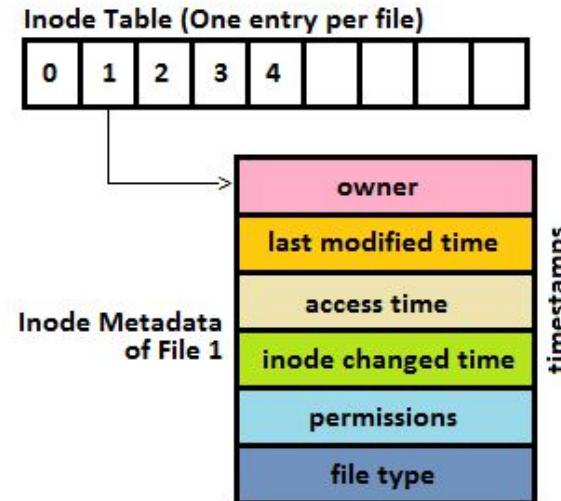
```
struct ext2_inode {  
    __le16  i_mode; // тип + права  
    __le16  i_uid;  
    __le32  i_size; // Размер файла в байтах  
    // Другие данные пропущены  
    __le32  i_blocks; // Сколько блоков занимает файл  
    __le32  i_block[15]; // Номера LBA, занимаемые файлом  
    // ...остальные данные  
};
```

# Как i-узел указывает на данные?



# Где хранятся i-узлы?

- I-узлы хранятся в таблице, которая просто является массивом, где значения по индексам этого массива – это LBA где находится описание i-узла
- Количество записей в таблице в каждой группе блоков обычно 8192 для ext4, но может быть от 8192 до 32768
- Битовая карта для i-узлов показывает занята ли запись в таблице



# Символическая ссылка

- Символическая ссылка – это специальный файл, который содержит путь к другому файлу
- Когда программа открывает символическую ссылку, операционная система открывает файл по пути, который в ней указан
- Если переименуем файл, на который указывает ссылка, то символическая ссылка не будет никуда указывать

# Жесткая ссылка

- Все файлы на диске имеют номера, записанные в i-узле
- Жесткая ссылка появляется, когда разные имена файлов указывают на один i-узел
- Любые изменения в одном файле приведут к изменениям в другом. Если удалить один из файлов, то второй файл останется.
- На i-узел есть reference counter, показывающий, сколько имен ссылаются на это содержимое. Когда reference counter = 0, ОС удаляет файл. Если открыть файл, то его reference counter на это время увеличится

# Директория

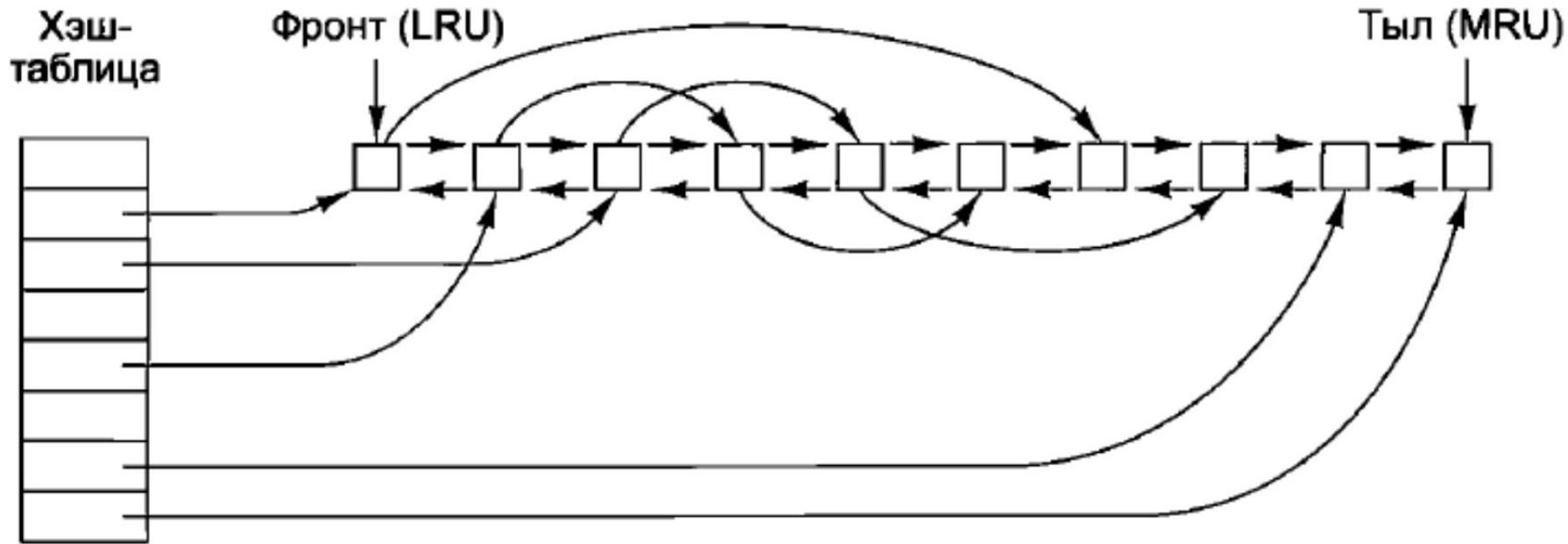
- В Unix “абсолютно всё – файлы”. Даже директории это файлы.
- Только этот “файл” содержит таблицу связывающую имя файла или директории с номером i-узла
- **dentry** (directory entry) представляет одну такую запись **в памяти** ядра в директории, а директория состоит из множества таких записей

```
struct dentry {  
    unsigned int d_flags;           // флаги (DCACHE_*)  
    struct inode *d_inode;          // указатель на inode  
    struct dentry *d_parent;         // родительский dentry  
    struct qstr d_name;             // имя (строка + длина)  
    struct list_head d_child;        // для дочерних (в родителе)  
    struct list_head d_subdirs;       // для подкаталогов (в родителе)  
    struct hlist_head d_bucket;      // для хэш-таблицы  
    // ... и ещё много полей  
};
```

# Ускорение поиска файла по имени

- Используется кэширование и хэширование
- В Linux **dcache** – это кэш dentry, которая представляет хэш-таблицу всех активных dentry, использует хэш по имени + родителю
- Жизненный цикл dentry:
  - Поиск по имени (`lookup()`)
  - Создать новый dentry (`d_alloc()`)
  - Привязать к i-узлу (`d_add()`)
  - Уменьшить счетчик ссылок (`dput()`)
  - Пометить как недействительный (`d_invalidate()`)
  - Удалить из памяти при нехватке RAM (`d_prune()`)

# Ускорение поиска файла по имени



# Основные операции с файлами

1. Создать (create), удалить (unlink), переименовать (rename)
2. Открыть (open), закрыть (close)
3. Прочитать (read), записать / добавить (write)
4. Найти (lseek)
5. Получить атрибуты и установить атрибуты (lstat, chmod, chown, ...)

# Операции с директориями

1. Создать (`opendir`), удалить (`rmdir`), переименовать (`rename`)
2. Открыть (`opendir`), закрыть (`closedir`)
3. Прочитать следующую запись в каталоге (`readdir`, `seekdir`)
4. `link`, `unlink`, `symlink` (как hard-, так и символические ссылки)

```
struct dirent {  
    ino_t          d_ino;        /* Inode number */  
    off_t          d_off;        /* Not an offset; see below */  
    unsigned short d_reclen;    /* Length of this record */  
    unsigned char   d_type;      /* Type of file */  
    char           d_name[256];  /* Null-terminated filename */  
};
```

# Virtual File System

- Представляет собой единый абстрактный интерфейс в ядре ОС, который позволяет работать с разными файловыми системами
  - Без VFS каждый `open()`, `read()` и др. сис. вызовы должны знать, какая ФС содержит этот файл и как с ней работать
  - С VFS программа вызывает сис. вызов `open()`, а ядро само решает, как это сделать
- Ядро поддерживает в памяти объекты, которые характеризуют виртуальную файловую систему
- Эти объекты в памяти имеют указатели на функции, которые умеют работать с конкретной файловой системой и ее спецификой
- Предоставляет интерфейс для разных типов файловых систем: физических (`ext2,3,4`, `btrfs`, `xfs`, `zfs`, и др.) и виртуальных (`tmpfs`, `procfs`, `devfs`, `sysfs`, и прочие FUSE-based)

# Virtual File System: суперблок и i-узел

```
struct super_block {
    struct dentry *s_root;          // корневой dentry
    struct super_operations *s_op;  // методы: read_inode, sync_fs и др.
    struct file_system_type *s_type; // тип ФС (ext4, nfs и т.д.)
    // ...
};

struct inode {
    umode_t i_mode;                // тип + права
    uid_t i_uid;
    struct inode_operations *i_op;  // методы: create, unlink, mkdir
    struct file_operations *i_fop; // методы для открытых файлов
    void *i_private;               // данные конкретной ФС
    // ...
};
```

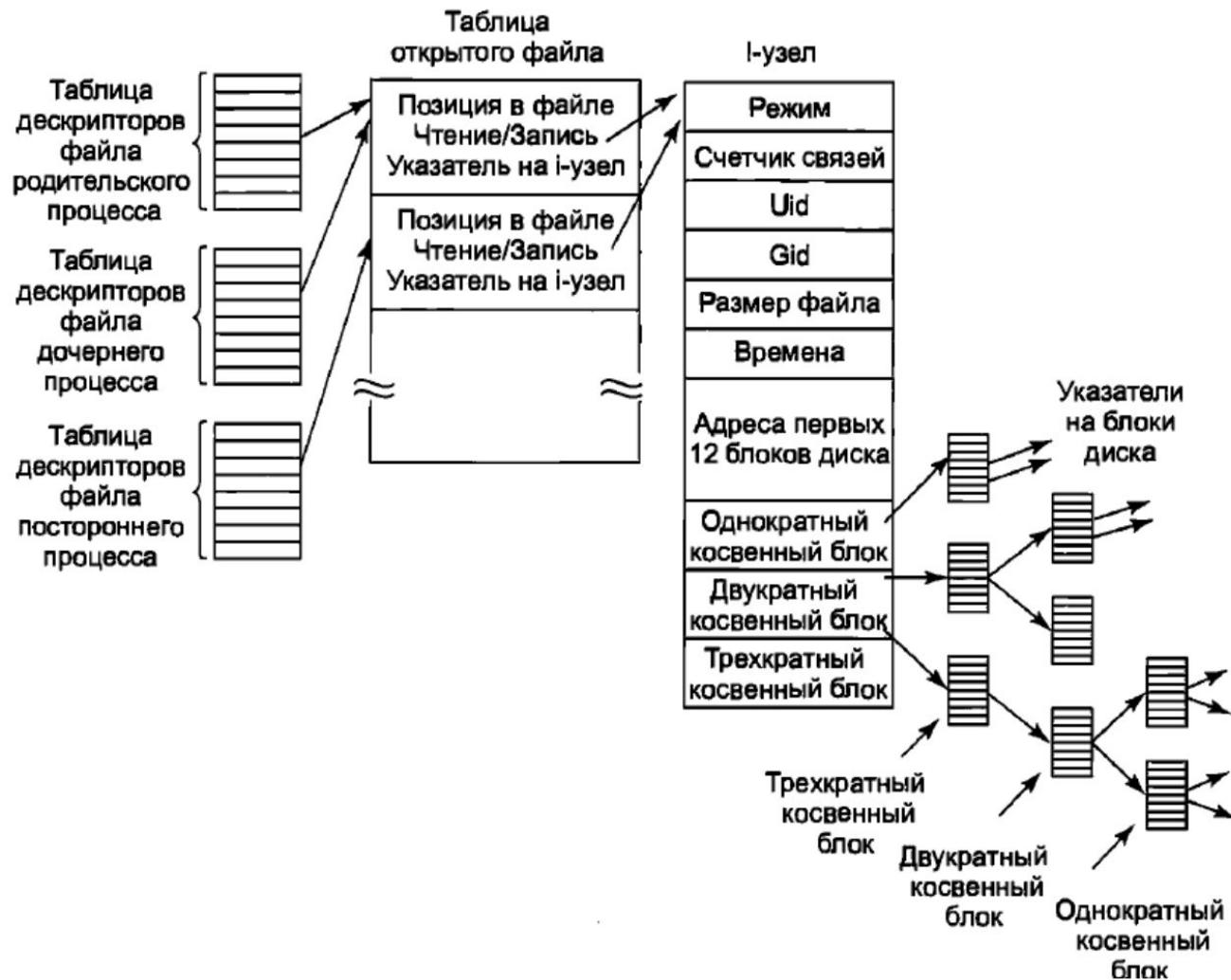
# Virtual File System: директория и файл

```
struct dentry {  
    struct inode *d_inode;    // Указатель на inode  
    struct qstr d_name;      // Название файла  
    struct dentry *d_parent; // Родительский dentry  
    // ...  
};  
  
struct file {  
    struct path f_path;        // { dentry, mount }  
    struct file_operations *f_op; // read, write, lseek и пр.  
    loff_t f_pos;              // текущая позиция  
    // ...  
};
```

# Что происходит при открытии файла?

Например в коде был вызван `open( "/etc/passwd" )`

1. `syscall open( )` обращается в ядро
2. `namei( )` разбирает путь до файла в цепочку dentry: { “/”, “etc”, “passwd” }
3. Происходит `lookup` в `dcache`, должен быть найден dentry для `passwd`
4. После успешной находки, `d_inode` в dentry `password` будет содержать номер i-узла этого файла
5. `inode->i_op->lookup()` узнает инфу про i-узел для `passwd`
6. `alloc_file()` создаёт `struct file` для этого i-узла
7. `file->f_op = inode->i_fop` структура наследует методы для работы с файлом от i-узла
8. Ядро находит свободный индекс дескриптора в таблице дескрипторов процесса, аллоцируется новый дескриптор и возвращается как результат





# Исполняемые файлы

- На рисунке изображён исполняемый файл формата ELF (Unix, Linux, macOS)
- Исполняемый файл состоит из заголовков и секций
- Про секции `text`, `data`, `bss`, `rodata` мы уже знаем
- Программа пользуется библиотеками, где находятся внешние функции. Информация о том, где искать адреса внешних функций и глобальных данных находится в **таблице символов**

# Как ядро запускает программу на примере Linux

Когда вызывается системный вызов `exec()`, происходит следующее:

- Ядро читает заголовок файла, чтобы определить его формат. В Linux это ELF, но поддерживаются и другие (скрипты, `a.out` и т.д.)
- Если это ELF-файл (магическое число: `0x7F 'E' 'L' 'F'`), ядро парсит заголовок: проверяет архитектуру (x86, ARM и т.д.), тип (exec или shared object) и секции (`.text` для кода, `.data` для данных и т.д.)
- Если файл — скрипт (начинается с `#!`), ядро запускает интерпретатор (например, `/bin/bash` для .sh-скриптов) и передаёт ему скрипт как аргумент
- Если формат не поддерживается, возвращается ошибка (EACCES или ENOEXEC)

# Как ядро запускает программу на примере Linux

- Ядро очищает текущее адресное пространство процесса, но сохраняет PID и другие метаданные
- Используя системный вызов `mmap()`, ядро отображает секции ELF-файла в виртуальную память:
  - `.text` (код): отображается с правами на чтение и исполнение (`PROT_READ | PROT_EXEC`), обычно `shared` для экономии памяти
  - `.data` и `.rodata`: с правами на чтение/запись (`PROT_READ | PROT_WRITE` для `.data`)
  - `.bss`: инициализируется нулями (не хранится в файле, но выделяется в памяти)
- Адреса назначаются с учётом ASLR (Address Space Layout Randomization) для безопасности, чтобы адреса не были предсказуемыми
- Если программа динамически скомпонована, ядро загружает динамический загрузчик (`linker`) `/lib/ld-linux.so`. Этот загрузчик затем загружает библиотеки (например `libc.so`) также через `mmap()`

# Как ядро запускает программу на примере Linux

- Ядро выделяет стек (stack) для процесса в верхней части адресного пространства (0x7fffffff000 на x86-64)

На стек помещаются:

- Аргументы командной строки (argv)
- Переменные окружения (envp, вроде PATH, HOME)
- Auxiliary vector (auxv) – это дополнительные данные, такие как адрес загрузчика, случайные байты для ASLR и т.д.
- Указатели на argv и envp устанавливаются в регистрах (rdi, rsi на x86-64)
- Если есть, настраивается VDSO (Virtual Dynamic Shared Object) — специальная область от ядра для быстрых syscall'ов (gettimeofday и т. д.), без перехода в kernel mode

# Как ядро запускает программу на примере Linux

- После настройки ядро устанавливает точку входа (entry point), то есть адрес поля `e_entry` в ELF-заголовке (обычно `_start` в `crt0.o`)
- Процесс возвращается в user mode
- Управление передается на `_start()`, который вызывает `__libc_start_main()`, а затем `main()` программы
- Если программа статически скомпонована, загрузчик не нужен — сразу в код

# Небольшой FAQ про memory mapping

- Q: Будет ли ядро и загрузчик выделять новую память, чтобы загрузить секции исполняемой программы и библиотек?
- A: Нет, потому что ядро отслеживает открытые программы и библиотеки и использует memory mapping, чтобы отобразить секции единственных загруженных экземпляров программы или библиотек. Так мы экономим память и пользуемся одной из важных возможностей виртуального адресного пространства
- Q: Как быть, если одна из секций (например data) экземпляра программы была изменена, но она была общей для всех программ
- A: Ядро пользуется принципом CoW

# Статическая компоновка программ

- Compile time: компилируем код, получая объектные файлы, которые предоставляют символы, которые в них определены, и требуют символов, которые в них используются;
- Link time: во время компоновки объектных файлов в исполняемый файл компоновщик разрешает (resolves) символы в их адреса и подставляет эти адреса в машинный код. То есть патчит заглушки, где нужны адреса
- Runtime: образ исполняемого файла требует, чтобы его секции загрузили в память по фиксированным адресам, и рассчитывает на это в своей работе

# Динамическая компоновка программ

- Link time: компоновщик оставляет некоторые символы неразрешёнными, но записывает в executable информацию о том, какие динамические библиотеки ему требуются для работы
- Runtime: динамический загрузчик разрешает используемые символы в их адреса, разыскивая символы в загруженных библиотеках

# Динамическое разрешение символов

- Ядро этим заниматься не должно
- Программа, которая этим занимается, называется **динамический загрузчик**. Она должна исполняться в адресном пространстве той программы, в которой она загружается

# Динамическое разрешение символов

- Когда запускается динамически скомпонованная программа, сначала загружается интерпретатор (interpreter), указанный в ELF-заголовке (обычно это `/lib64/ld-linux-x86-64.so.2`)
- Этот загрузчик (`ld.so`) анализирует зависимости программы (из секции `.dynamic` ELF-файла) и находит нужные `.so`-библиотеки (например, `libc.so`)
- Далее используется memory mapping, чтобы отобразить секции динамической библиотеки в виртуальное адресное пространство процесса

# Динамическое разрешение символов

- Для каждой библиотеки ld.so открывает файл (через open( ))
- Затем вызывается mmap несколько раз, чтобы отобразить разные секции ELF-файла в память:
  - .text (код): mmap с PROT\_READ | PROT\_EXEC, MAP\_SHARED для исполняемого кода, который может быть shared между процессами.
  - .data и .bss (данные): mmap с PROT\_READ | PROT\_WRITE, MAP\_PRIVATE для modifiable данных, с copy-on-write
  - .rodata (read-only данные): mmap с PROT\_READ, MAP\_SHARED
- Это происходит рекурсивно: библиотеки сами могут зависеть от других, так что ld.so загружает всю цепочку