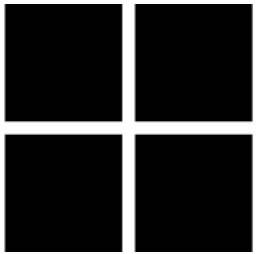
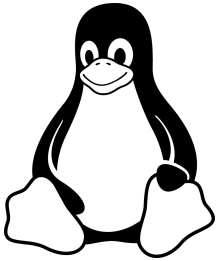
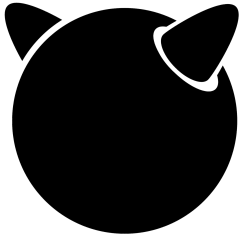
Operating systems





macOS



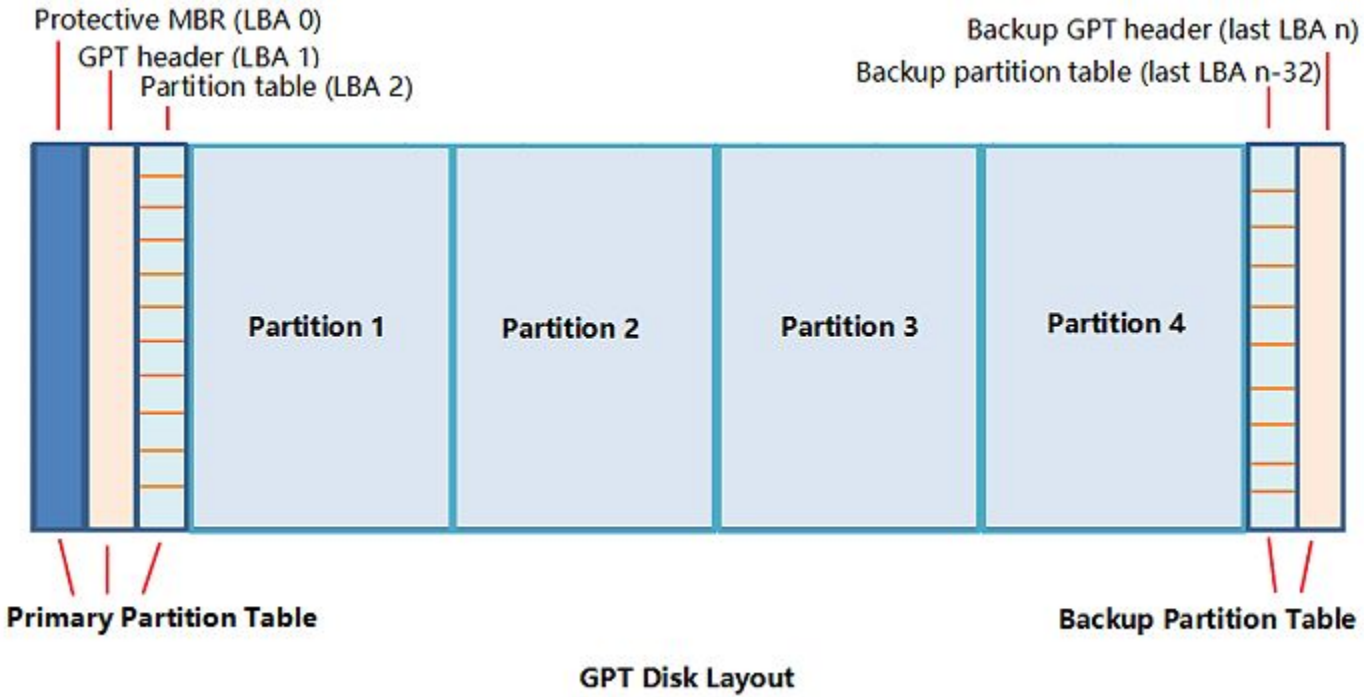
Файловые системы

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**

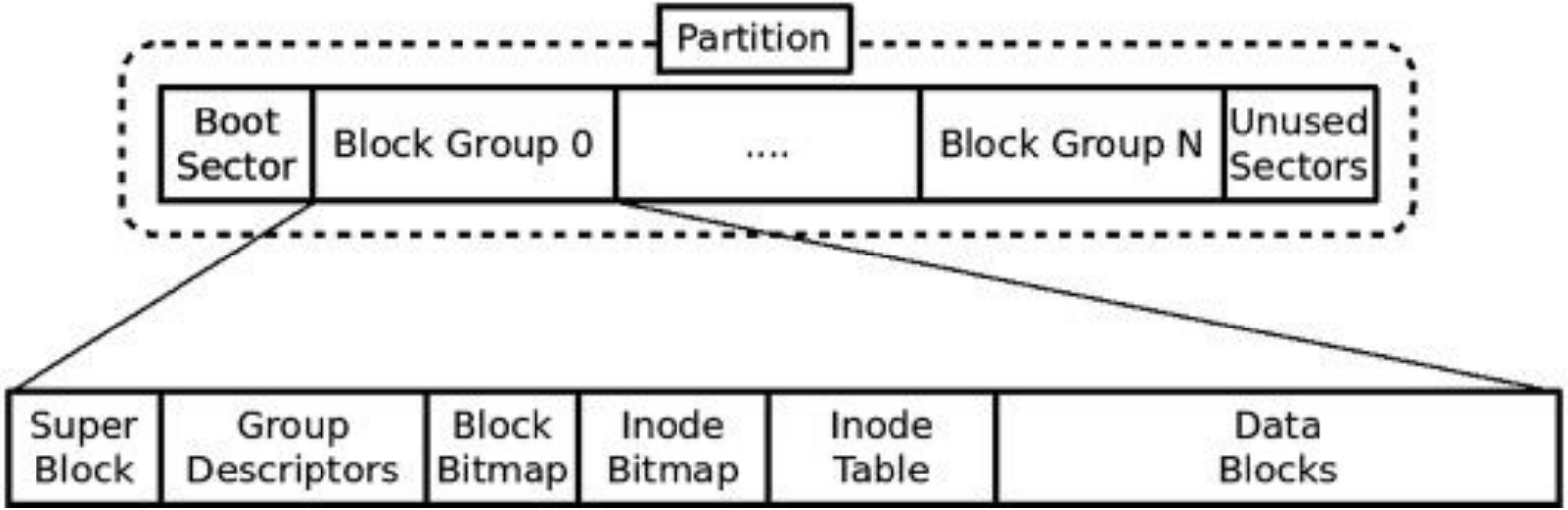
Структура диска



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

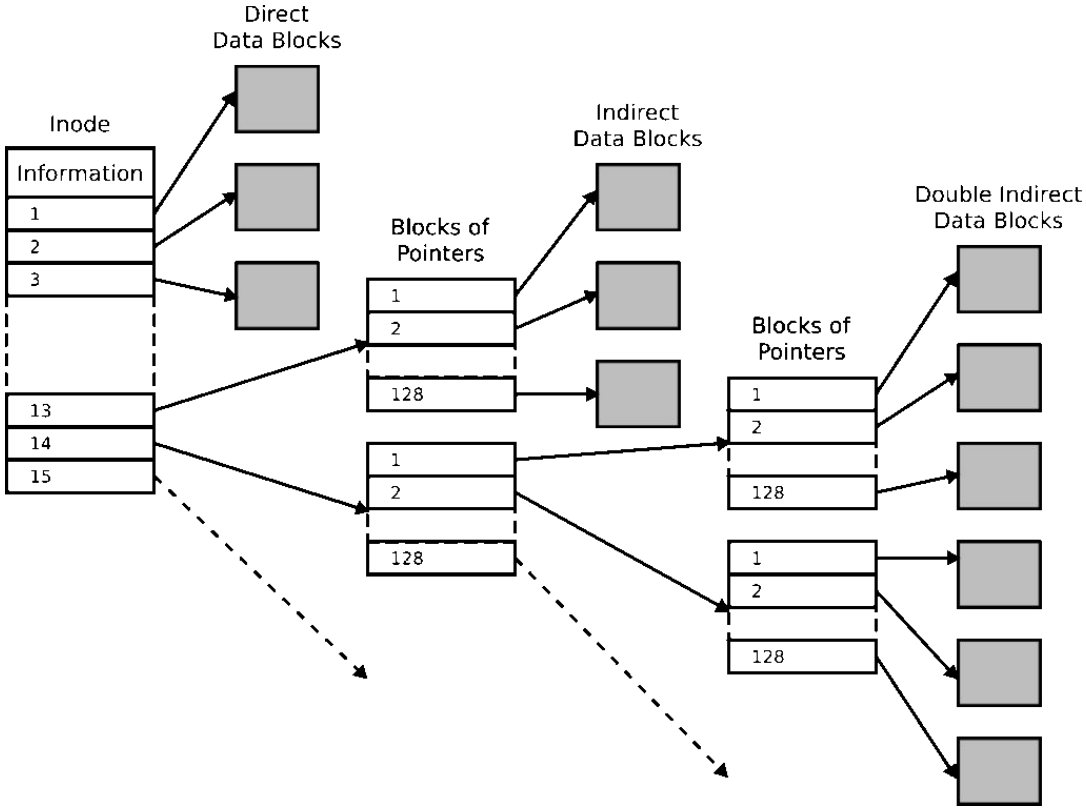
Что за i-узлы?

- **i-node (index node) –** это структура, которая хранит все метаданные файла или каталога, кроме имени
- Что в них хранится:
  - **Тип файла** (обычный файл, каталог, символическая 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?

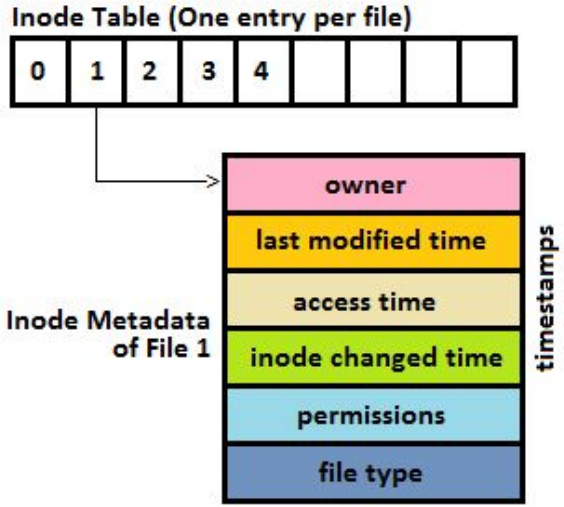
```
struct ext2_inode {
    __le16  i_mode; // тип + права
    __le16  i_uid;
    __le32  i_size; // Размер файла в байтах
    // Другие данные пропущены
    __le32  i_blocks; // Сколько блоков занимает файл
    __le32  i_block[15]; // Номера LBA, занимаемые файлом
    // ...остальные данные
};
```

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

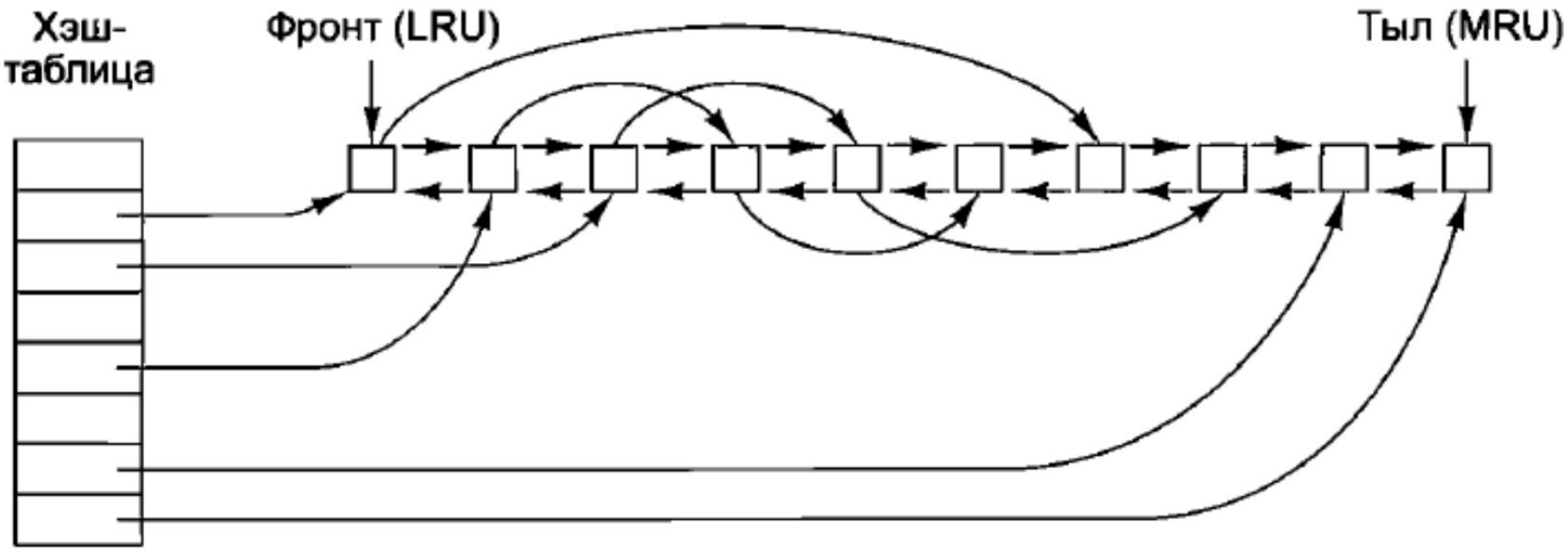
- В Unix “абсолютно всё – файлы”. Даже директории это файлы.
- Только этот “файл” содержит таблицу связывающую имя файла или директории с номером i-узла
- **dentry** (directory entry) представляет одну такую запись ***в памяти*** ядра в директории, а директория состоит из множества таких записей

```
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

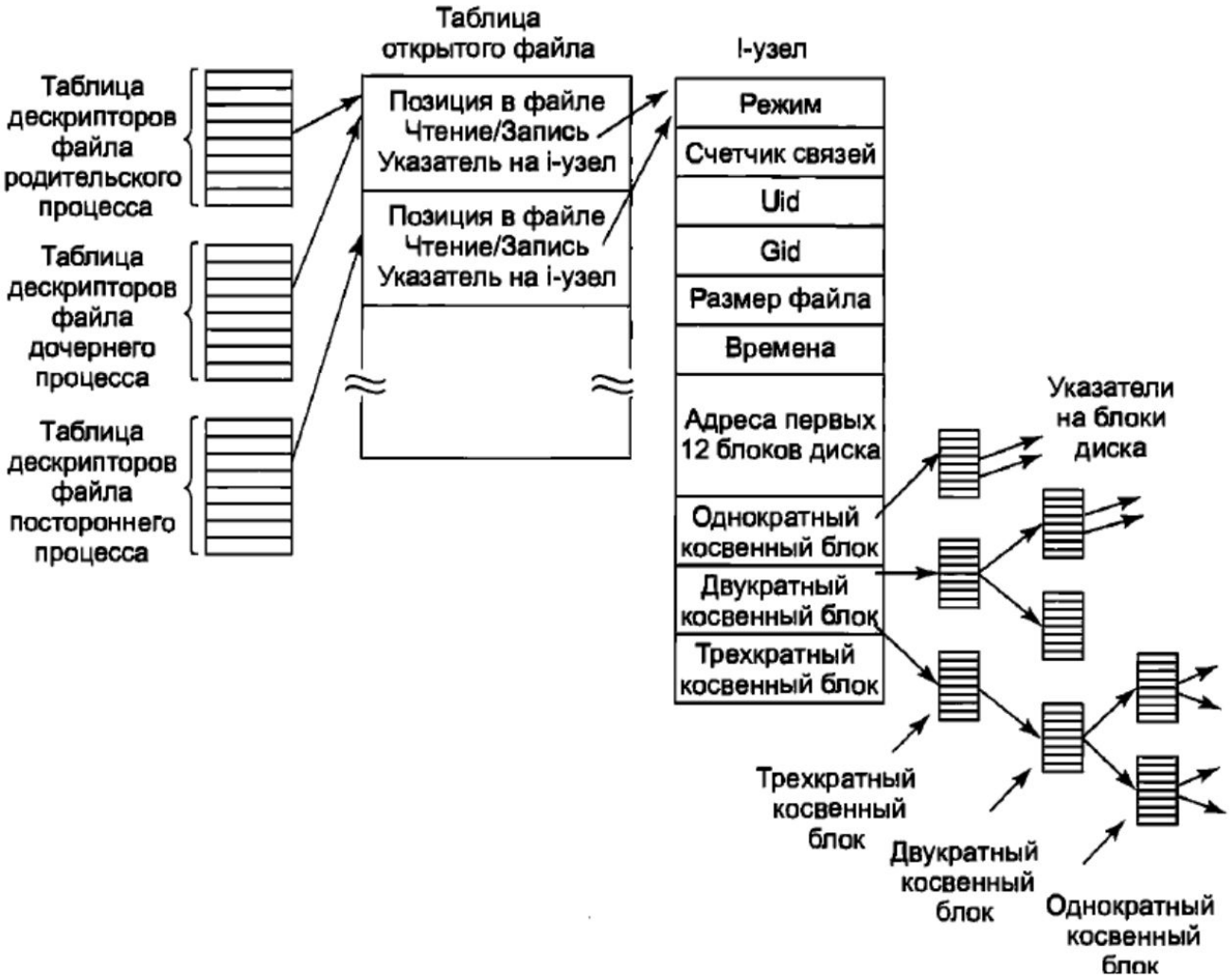


Таблица открытого файла
I-узел
Таблица дескрипторов файла родительского процесса
Таблица дескрипторов файла дочернего процесса
Таблица дескрипторов файла постороннего процесса
Позиция в файле
Чтение/Запись
Указатель на i-узел
Позиция в файле
Чтение/Запись
Указатель на i-узел
Режим
Счетчик связей
Uid
Gid
Размер файла
Времена
Адреса первых 12 блоков диска
Однократный косвенный блок
Двукратный косвенный блок
Трехкратный косвенный блок
Указатели на блоки диска
Трехкратный косвенный блок
Двукратный косвенный блок
Однократный косвенный блок