

# 1. Билет №13

**Файловая подсистема: особенности файловой подсистемы Unix/Linux.:** иерархическая структура файловой подсистемы. Виртуальная файловая система VFS в Linux. Четыре структуры VFS – `super_block`, `inode`, `dentry`, `file` их назначение. Адресация файлов большого размера в файловой системе `extX` и пример, показывающий доступ к файлу `/usr/ast/mbox`. Монтирование файловых систем. Команда `mount` и функции монтирования, пример из лаб. раб.

## 1.1. Файловая подсистема

Файл — важнейшее понятие в файловой подсистеме. Файл — информация, хранимая во вторичной памяти или во вспомогательном ЗУ с целью ее сохранения после завершения отдельного задания или преодоления ограничений, связанных в объеме основного ЗУ.

Файл — поименованная совокупность данных, хранимая во вторичной памяти (возможно даже целая). Файл — каждая индивидуально идентифицированная единица информации.

Существует 2 ипостаси файла:

1. файл, который лежит на диске;
2. открытый файл (с которым работает процесс).

Открытый файл — файл, который открывает процесс.

Файл != место на диске. В мире современной вычислительной техники файлы имеют настолько большие размеры, что не могут храниться в непрерывном физическом адресном пространстве, они хранятся вразброс (несвязанное распределение).

Файл может занимать разные блоки/сектора/дорожки на диске аналогично тому, как память поделена на страницы. В любой фрейм может быть загружена новая страница, как и файл.

Также, важно понимать адресацию.

Соответственно, система должна обеспечить адресацию каждого такого участка.

ОС является загружаемой программой, её не называют файлом, но когда компьютер включается, ОС находится во вторичной памяти. Затем с помощью нескольких команд, которые находятся в ПЗУ, ОС (программа) загружается в ОЗУ. При этом выполняется огромное количество действий, связанных с управлением памятью, и без ФС это сделать невозможно. Любая ОС без ФС не может быть полноценной.

Задача ФС — обеспечивать сохранение данных и доступ к сохраненным данным (обеспечивать работу с файлами).

Чтобы обеспечить хранение файла и последующий доступ к нему, файл должен быть изолирован, то есть занимать некоторое адресное пространство, и это адресное пространство должно быть защищено. Доступ обеспечивается по тому, как файл идентифицируется в системе (доступ осуществляется по его имени).

ФС — порядок, определяющий способ организации хранения, именования и доступа к данным на вторичных носителях информации.

File management (управление файлами) — программные процессы, связанные с общим управлением файлами, то есть с размещением во вторичной памяти, контролем доступа к файлам, записью резервных копий, ведением справочников (directory).

Основные функции управления файлами обычно возлагаются на ОС, а дополнительные — на системы управления файлами.

Доступ к файлам: open, read, write, rename, delete, remove.

Разработка UNIX началась с ФС. Без ФС невозможно создание приложений, работающих в режиме пользователя (сложно разделить user mode и kernel mode).

Файловая подсистема взаимодействует практически со всеми модулями ОС, предоставляя пользователю возможность долговременного хранения данных, а также ОС возможность работать с объектами ядра.

## 1.2. Особенности файловой подсистемы Unix/Linux

В Unix все файл, если что-то не файл, то это процесс.

В системе имеются спец. файлы, про которые говорят, что они больше чем файл: программные каналы, сокеты, внешние устройства.

Файловая система работает с регулярными (обычными) файлами и директориями. При этом Unix/Linux не делают различий между файлами и директориями.

Директория – файл, который содержит имена других файлов.

7 типов файлов в Unix:

1. '-' – обычный файл
2. 'd' – directory
3. 'l' – soft link
4. 'c' – special character device
5. 'b' – block device
6. 's' – socket
7. 'p' – named pipe

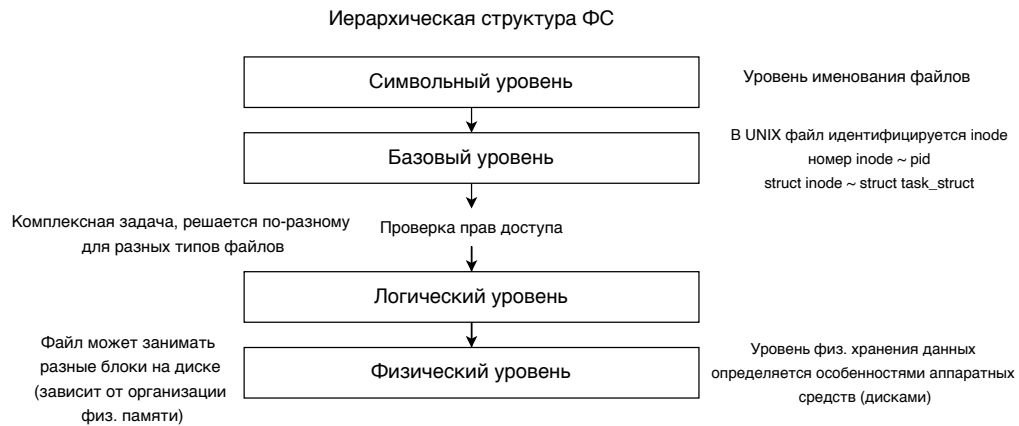
## 1.3. Иерархическая структура файловой подсистемы

Существует стандарт FileSystem Hierarchy Standard (FHS), который определяет структуру и содержимое каталогов в Linux distribution (Ubuntu поддерживает этот стандарт).

По этому стандарту корень файловой системы обозначается как «/» (корневой каталог) и его ветви обязательно должны составлять единую файловую систему, расположенную на одном носителе (диске или дисковом разделе). В нем должны располагаться все компоненты, необходимые для старта системы.

### Символьный уровень

Это уровень именования файлов. Сюда входит организация каталогов, подкаталогов.



В Unix/Linux имя файла не является его идентификатором. Один и тот же файл может иметь множество имён (hard link). Это делалось для того, чтобы к одному и тому же файлу можно было получать доступ из разных директорий. Файлы в системе идентифицируются с помощью inode.

Символьный уровень — самый верхний уровень файловой системы, именно он связан с именованием файлов и позволяет пользователю работать с файлами (так как помнить inode своих файлов сложно).

### Базовый уровень

Это уровень формирования дескриптора файлов. Должны быть соответствующие структуры, позволяющие хранить необходимую для файла информацию.

В ядре существует два типа inode (Index Node): дисковый и ядрёный. Чтобы получить доступ к файлу требуется перейти с символьного уровня к номеру inode, которым и идентифицируется в системе файл.

Обоснованием использования двух типов inode в системе является факт того, что Unix изначально создавалась как система которая поддерживает очень большие файлы. Для того чтобы адресовать данные которые находятся в этих файлах, необходимо иметь соответствующие структуры. Так как именно inode как сейчас принято говорить, является дескриптором файла, то такая информация должна храниться в дисковом inode.

### Логический уровень

Логическое адресное пространство файла аналогично адресному пространству процесса: оно начинается с нулевого адреса и представляет собой непрерывную последовательность адресов. Непрерывное адресное пространство, начинающееся с 0.

## Физический уровень

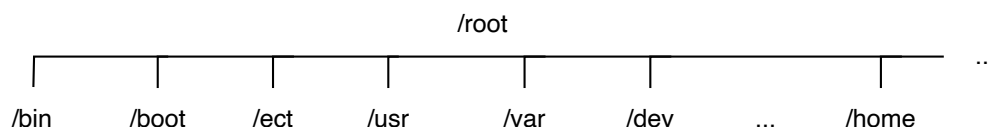
Это уровень хранения и доступа к данным.

## 1.4. Виртуальная файловая система VFS в Linux

Файловая система Linux.

Отличается от файловой системы UNIX предоставляемым интерфейсом: ОС Unix и Linux позволяют работать с большим количеством файловых систем.

Для этого UNIX предоставляет интерфейс VFS/vnode, а в Linux — VFS (отказались от vnode).



«Родные» файловые системы для Linux — ext (ext2, ext4, ...).

Структура слоев VFS:

## 1.5. Четыре структуры VFS — `super_block`, `inode`, `dentry`, `file` их назначение

Внутренняя организация VFS базируется на 4 структурах.

- `super_block`;
- `dentry`;
- `inode`;
- `file`.

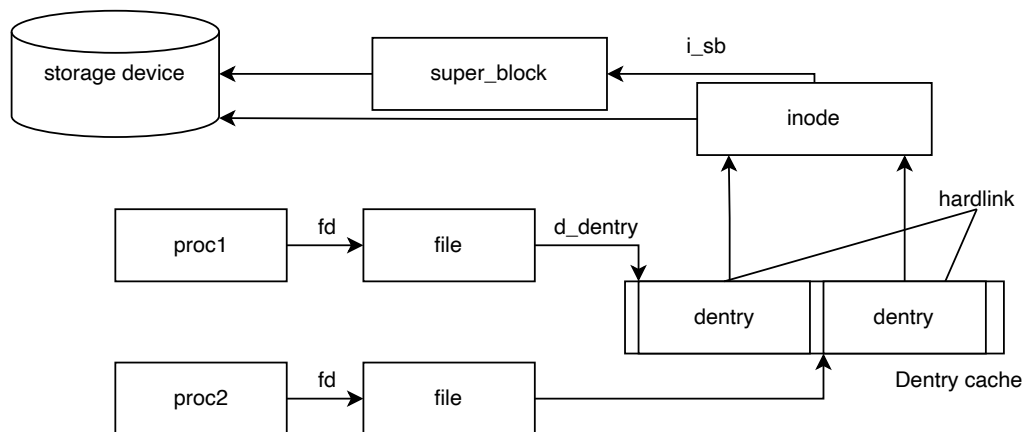
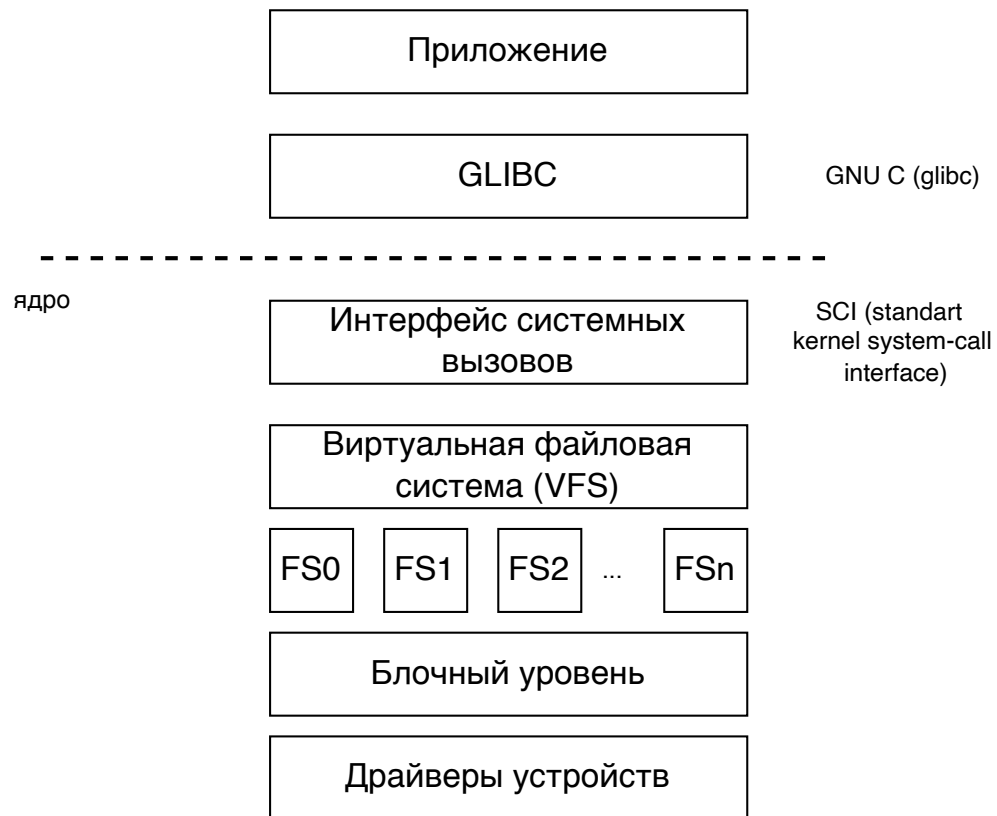
### 1.5.1. Связи структур

Связь структур VFS на основе полей структур — ключ к работе системы.

Одной из отправных точек являются системные вызовы, связанные с файлами: `read/write/lseek`.

Эти системные вызовы работают только с открытыми файлами: чтобы работать с файлом, его нужно открыть.

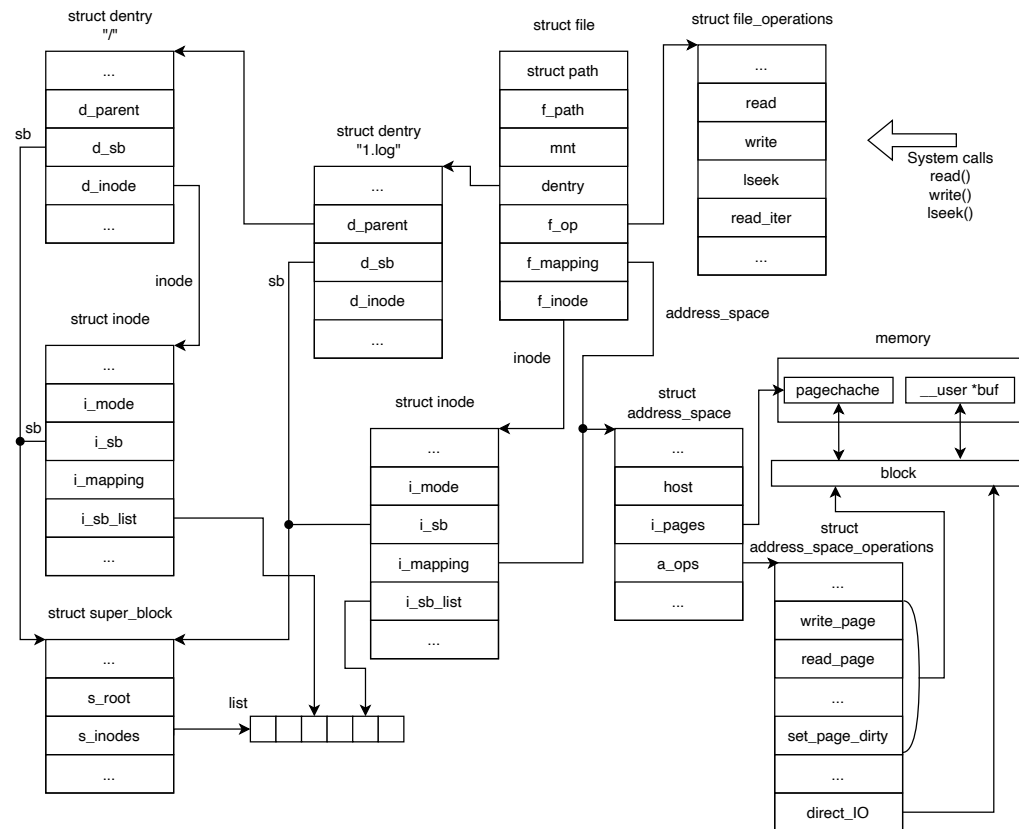
## Структура слоев VFS



### Упрощенная схема

Обычные файлы — регулярные. **inode** должен содержать информацию об адресах блоков диска, в которых хранится информация, записываемая в файл. Поэтому суперблок должен содержать соответствующую информацию о блоках на диске (свободен/занят), об **inode**'ах, иметь ссылку на соответствующую таблицу инодов.

### Связи структур при выполнении системных вызовов



### Связь между struct file и struct file\_operations

Файл должен быть открыт. Соответственно для открытого файла должен быть создан дескриптор. В этом дескрипторе имеется указатель на struct file\_operations. Это либо стандартные (установленные по умолчанию) операции на файлах для конкретной файловой системы, либо зарегистрированные разработчиком (собственные функции работы с файлами собственной файловой системы).

### Вспоминания о пояснениях

Указатель f\_mapping показывает связь структур, описывающих файлы в системе с памятью. Также в struct inode есть поле i\_mapping.

struct super\_block содержит список inode (s\_inodes). struct inode содержит указатель на соответствующий inode в списке (i\_sb\_list).

Любая файловая система имеет корневой каталог, а именно от корневого каталога формируется путь к файлу для конкретной файловой системы.

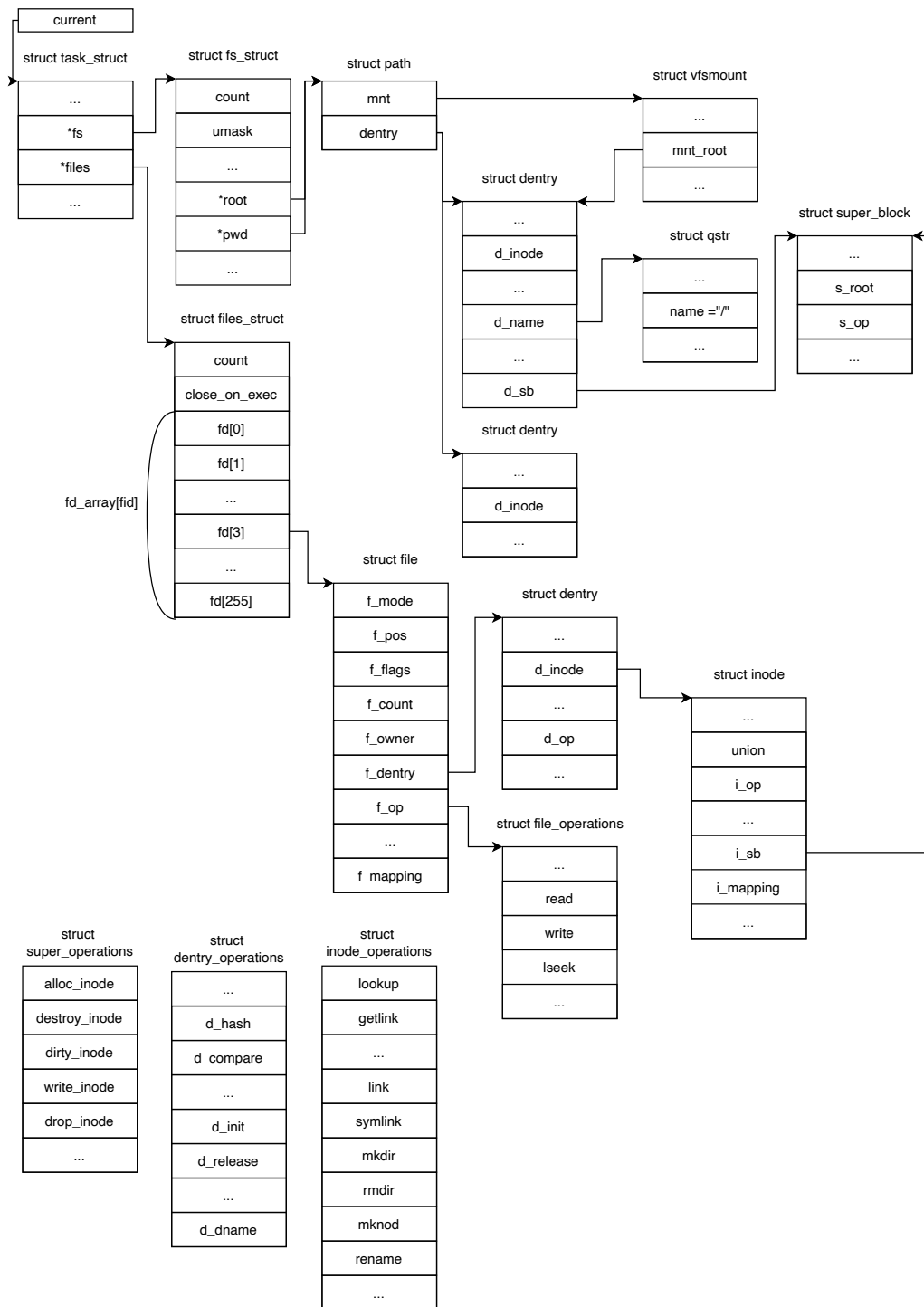
Отправная точка — системные вызовы (read, write, lseek, ...). Здесь нет open(), так как он открывает файл, а использование функций read, write, lseek возможно только при работе с открытым файлом.

### Связи структур относительно процесса

Теперь пойдем от процесса: Отправная точка — struct task\_struct; В struct task\_struct

есть 2 указателя:

- на **struct fs\_struct (\*fs)**; - Любой процесс относится к какой-то файловой системе
- на **struct files\_struct (\*files)** – дескриптор, описывающий файлы, открытые процессом (Любой процесс имеет собственную таблицу открытых файлов).



*Воспоминания о зарождении процесса*



Каждый процесс до того, как он был запущен, был файлом и принадлежал некоторой *вайбовой* системе, поэтому в **struct task\_struct** имеется указатель на фс, которой принадлежит файл программы, и указатель на таблицу открытых файлов процесса.

Очевидно, что **struct files\_struct** содержит массив дескрипторов открытых файлов (0,1,2,3,4,...).

При этом

- 0 – stdin
- 1 – stdout
- 02– stderr
- 03 – скорая помощь

Эти файлы открываются для процесса автоматически (файловые дескрипторы для этих файлов создаются автоматически).

Когда мы открываем файл, он может получить дескриптор, после этих трех (например, 3,4,5 и тд)

Всего в этой таблице может быть 256 дескрипторов.

**struct vfs\_mount** заполняется, когда файловая система монтируется. Имя – указатель на **struct qstr**.

В **struct super\_block** есть указатель на **struct super\_operations** (s\_op) и на root (s\_root), так как корневой каталог (точка монтирования) должен быть создан, чтобы иметь возможность смонтировать файловую систему.

## 1.5.2. struct super\_block

**struct super\_block** описывает подмонтированную файловую систему на диске.

Именно он обеспечивает возможность работы с файловой системой.

Содержит информацию для обеспечения доступа к файлам, которые хранятся на дисках вразброс, то есть адресацию соответствующих участков диска.

В **struct super\_block** содержится информация, необходимая системе для управления подмонтированной файловой системой.

У каждой файловой системы может быть один **super\_block**.

**Определение struct super\_block**

```
1 struct super_block {  
2     struct list_head      s_list;
```

```

3      dev_t                s_dev;                // устройство, на которо
           м находится ФС
4      unsigned long        s_blocksize;          // размер блока в байта
           x
5      unsigned char        s_dirt;                // флаг изменения супрбл
           ока
6      struct file_system_type s_type;              // в ядре есть структур
           а описывающая тип ФС
7      struct super_operations s_op;                // операции на суперблок
           е
8      struct block_device *b_dev // описывает устройство, на котором нах
           одится файловая система (соответствует драйверу блочного устрой
           ства)
9      unsigned long        s_magic;                // магический номер смон
           тированной ФС
10     struct dentry          *s_root;                // точка монтирования ФС
11     ...
12     int                   s_count;                // число ссылок
13     struct list_head      s_dirty;                // список измененных
           inode 'ов
14     char                  s_id[32];                // имя?
15 };

```

### Определение struct super\_operations

На любой структуре, описывающей объект ядра, определены функции для работы с объектом соответствующего типа (struct file\_operations, struct inode\_operations, struct dentry\_operations).

```

1 <linux/fs.h>
2
3 struct super_operations
4 {
5     struct inode *(alloc_inode)(struct superblock *sb);
6     void (*destroy_inode) (struct inode *);
7     ...
8     void (*dirty_inode)(struct inode *, int flags);
9     int (*write_inode)(struct inode*, struct wirteback_cintrol *wbc);
10    int (*drop_inode)(struct inode *);
11    ...

```

```

12 void (*put_super)(struct super_block *);
13 }

```

dirty\_inode вызывается VFS, когда в индекс (inode) вносятся изменения (функция используется для изменения соответствующей таблицы структуры).

Ядро хранит копию таблицы inode'ов в памяти ядра (так как доступ к диску — медленная операция), то есть inode, к которым были обращения, кешируются для ускорения доступа к файлам.

Сначала изменения вносятся в таблицу, которая находится в оперативной памяти.

Функция dirty\_inode позволяет отметить, что inode был изменен, и эту информацию надо скопировать в таблицу, которая находится на диске.

write\_inode предназначена для записи inode на диск и помечает inode как измененный.

put\_super вызывается VFS при размонтировании ФС.

### Подробное описание

Основная информация, которую хранит super\_block — информация, которая обеспечивает доступ к таблице inode'ов, и каждый дескриптор inode обеспечивает информацию для доступа к данным, хранящимся в файле. В struct super\_operations есть функция alloc\_inode (принимает super\_block, так как любой файл должен принадлежать конкретной файловой системе, то есть конкретному суперблоку; конкретный суперблок обеспечивает доступ к конкретному файлу).

Функция alloc\_super создает новый superblock. Вызывается при монтировании файловой системы. Возвращает указатель на новый superblock или NULL, если не удалось выделить superblock.

Функция alloc\_super создает новый superblock:

```

1 static struct super_block *alloc_super (
2     struct file_system_type *type ,
3     int flags ,
4     struct user_namespace *user_ns
5 )
6 {
7     struct superblock *s = kzalloc(sizeof(struct super_block) ,
8         GFP_USER);
9     static const struct super_operations default_ops;
10    if (!s) return NULL;
11    INIT_LIST_HEAD(&s->s_mounts);
12    ...
13    INIT_LIST_HEAD(&s->s_inodes);

```

```

13 |     ...
14 | }

```

`default_ops`: Для любой файловой системы определяется набор операций на суперблоке (система предоставляет разработчику определить эти операции).

`s_mounts`: Одна и та же файловая система может быть смонтирована много раз, при этом она будет иметь один тип.

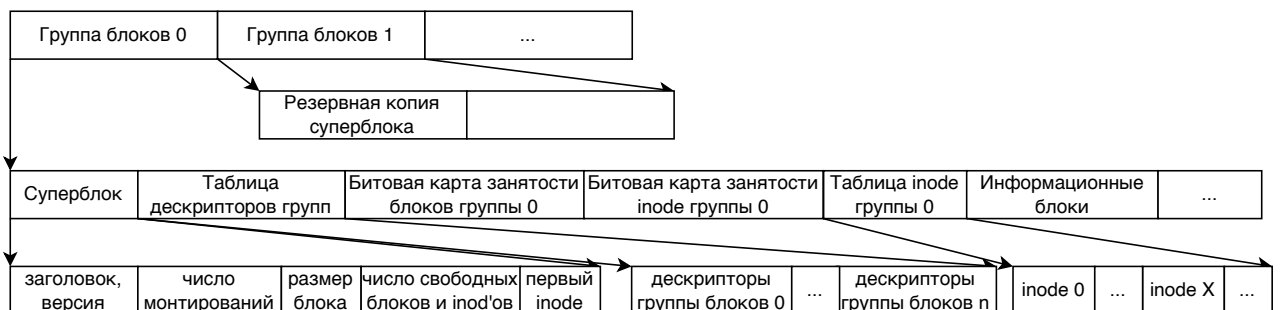
Любой экземпляр (объект) `super_block` описывает конкретную файловую систему, которая может быть подмонтирована, и только тогда файлы этой файловой системы будут доступны пользователю.

## Раздел жесткого диска и `super_block`

Если пользователь желает, чтобы фс стала доступна (пользователь сможет получить доступ к файлам и каталогам данной фс), она должна быть подмонтирована и для нее должен быть выделен раздел жесткого диска.

Раздел на диске – выделенное адресное пространство.

первая структура в разделе – `superblock`.



Размер блока (на диске) – непрерывное адресное пространство. В современных вычислительных система файлы относительно большие, поэтому она не могут храниться в непрерывном адресном пространстве. Она хранятся вразброс в свободных блоках. Система должна иметь возможность обращаться к каждому блоку, в котором хранится информация. Для этого в `inode` имеются соответствующие ссылки.

Чтобы система могла мобильно выделять файлам новые блоки, необходимо иметь соответствующую информацию.

`s_blocksize` – имя поля в `struct`

## `super_block`

Первый `inode` – `inode` корневого каталога фс.

В группе блоков 1 находится резервная копия суперблока. В каждой фс один `super_block`, и он располагается в начале раздела. У него есть копия для обеспечения надежности работы с фс, хранения файлов, `super_block` считывается в память ядра при монтировании фс и находится там до ее демонтирования (или завершения работы с системой).

`timestamps` – временные отметки, указывающие время модификации `inode`:

- `ctime` – время модификации `inode`
- `mtime` – время модификации файла
- `atime` – время последнего доступа к файлу.

Блок может быть адресован, следовательно мы знаем его адрес.

Битовая карта блоков (`block bit map`) – структура, в которой каждый бит показывает, занят блок или нет (отведен ли он какому-то файлу)

Битовая карта `inode` – выполняет аналогичную роль по отношению к таблице `inod'ов`. То есть показывает, какие индексные дескрипторы заняты, а какие свободны.

### 1.5.3. `struct dentry`

`struct dentry` (`directory entry`) описывает экземпляр директории, нигде не хранится, создается на основе информации, которая хранится в директориях на диске (`inode` с диска).

~~`struct dentry` не имеет `mapping`, то есть нигде не отображена.~~

~~Именно поэтому~~ имена поддиректорий хранятся как обычные файлы, так как эта информация нужна системе, чтобы предоставить в распоряжение пользователя имена директорий и поддиректорий.

~~Деревя каталогов не существуют, то есть оно строится "на лету" (например, утилитой `tree`) на основе той информации, которая сохранена на диске. Чтобы ускорить обращение к этой информации, она вся кешируется.~~

То есть когда происходит первое обращение к каталогу, он кешируется (существует соответствующая `struct list_head`, в которой будет храниться информация об этом каталоге).

`struct dentry` описывает элемент пути. Элемент пути — часть пути к файлу, которые отделяются друг от друга «/», начиная с корневого каталога.

Про объекты `dentry`:

- информация о любом элементе пути хранится как файл.
- существует кеш объектов `dentry`, в котором хранятся элементы пути, к которым уже были обращения. Это ускоряет доступ к файлам.

Объект `dentry` может находиться в одном из 4 состояний:

1. `free`

Не содержит достоверной информации и не используется VFS. Соответствующая область памяти обрабатывается SLAB allocator'ом.

2. `unused`

В настоящее время ядром не используется. Счетчик `d_count` равен нулю, но поле `d_inode` по-прежнему указывает на соответствующий индексный дескриптор. Неиспользуемый объект `dentry` содержит достоверную информацию, но при необходимости он может быть удален и память может быть освобождена.

3. `in use`

Используется ядром в текущий момент. Счетчик `d_count` больше нуля. У такого объекта есть `inode` (поле `d_inode` указывается на соответствующий дескриптор). Такой объект `dentry` не может быть удален.

4. `negative`

Для него не существует соответствующий ему `inode`. Это возможно, если соответствующий `inode` был удален с диска или объект `dentry` был создан как элемент пути несуществующему файлу.

Поле `d_inode = NULL`, но объект все еще находится в кеше `dentry`.

Объекты `dentry in use` могут стать `negative`, если удаляется последний `hard link` на соответствующий файл. В этом случае объект `dentry` перемещается в список LRU `unused_dentry`.

## Определение `struct dentry`

type	field	description
atomic_t	d_count	Кол-во использований объекта dentry
unsigned int	d_flags	Флаги, определенные для конкретного объекта dentry
struct dentry *	d_parent	Указатель на родительский каталог
struct list_head	d_hash	Указатель на список в хеш-таблице (указатели на соседние элементы в списке, связанные с одним и тем же значением хеш-функции)

type	field	description
struct list_head	d_lru	Указатель на список dentry в состоянии unused (очищается по алгоритму LRU, то есть вытесняются dentry, к которым дольше всего не было обращений) (организован по алгоритму LRU, так как какое-то время неиспользуемые dentry хранятся в списке для скорения обращения к файлам)
struct list_head	d_child	Список подкаталогов
...	...	...
int	d_mounted	Флаг, который установлен <=> dentry является точкой монтирования ФС
struct qstr	d_name	Имя файла
struct dentry_operations *	d_op	Функции (системные вызовы) для работы с dentry
struct super_block *	d_sb	Любая директория относится к конкретной файловой системе, то есть дерево каталогов - дерево конкретной файловой системы => объект dentry (как эл-т пути) всегда принадлежит конкретной файловой системе
unsigned long	d_vfs_flags	Флаги кеша dentry
struct list_head	d_alias	list of associated inodes



## Определение struct

### dentry\_operations

```
1 struct dentry_operations
2 {
3     int (*drevalidate)(struct dentry *, unsigned int);
4     ...
5     int (*d_hash)(const struct dentry *, unsigned int);
6     int (* d_compare)(const struct dentry *, unsigned int, const char *,
7         const struct
8         qstr *);
9     int (* d_delete)(const struct dentry *);
10    int (* d_init)(const struct dentry *);
11    int (* d_release) (struct dentry *);
12    void (* d_input)(struct dentry *, struct inode *);
13    char *(* d_name)(struct dentry *, char *, int);
14    ..
15 }
```

Краткое пояснение полей структуры:

- d\_hash — хеширование;
- d\_compare — сравнение: когда мы проходим по пути к файлу, мы сравниваем заданное имя и найденное.
- d\_init — выделение dentry.
- d\_release — освобождение dentry: освободить dentry можно, если на него нет ссылок;
- d\_dname — определение/создание пути к эл-ту (объекту) dentry.

Подробное пояснение полей структуры:

- d\_hash — вызывается, когда VFS добавляет dentry в хеш-таблицу.

Первый dentry, который добавлен с помощью d\_hash, является родительским каталогом.

- d\_compare — вызывается для того, чтобы сравнить заданное имя с именем dentry; При этом первый dentry является родителем того dentry, который сравнивается.

В параметрах `const struct qstr *` — имя, с которым надо сравнить.

- `d_delete` вызывается, если удаляется последующая ссылка на `dentry`.
- `d_init` вызывается при создании.
- `d_releae` вызывается при освобождении.
- `d_input` вызывается, когда `dentry` теряет `inode`.
- `d_name` вызывается, когда необходимо сгенерировать путь к элементу `dentry`.

## Кеш dentry

in english: `dentry_cache`

Обращение к диску — очень длительная операция.

Информация об элементе пути (`dentry`) имеет `inode` (хранится на диске в виду файла).

Таких элементов пути к файлу может быть много, так как папки можно вкладывать одну в другую. Каждое вложение — дополнительный объект `dentry` (элемент пути), то есть файл.

При обращении к файлу просматриваются все элементы пути (каждый раз — обращение к диску, поэтому объекты `dentry` хранятся в кеше, это существенно уменьшает время обращения к конкретному файлу). При этом они не удаляются просто так, так как могут использоваться позже.

В Linux кеш `dentry` состоит из 2 типов структур./

1. set of `dentry` object в следующих состояниях: `in use`, `unuse`, `negative`.
2. hash table для быстрого получения объекта `dentry`, который связан с заданным именем файла и заданным каталогом.

Если объект, к которому происходит обращение, не включен в кеш `dentry`, то функция хеширования возвращает нулевое значение.

Кеш `dentry` фактически действует как контроллер для `cache inode`. То есть кроме кеша `dentry` есть `cache inode` (`slab cache` - его часть).

Все эти кешы хранятся в оперативной памяти.

Все неиспользуемые данные включены в двусвязный список, который обновляется по алгоритму LRU.

Адреса первого и последнего элемента списка LRU хранятся соответственно в полях `next` и `prev` переменной `dentry_unused`.

Воспоминания о списках в ядре.

В ядре используются двусвязные списки для обеспечения быстрого доступа за счет реализации соответствующих алгоритмов, простейшим из которых является бинарный поиск.

Каждый объект dentry в состоянии in use включается в двусвязный список (поле i\_dentry) соответствующего объекта inode (i\_dentry <- inode).

Поле d\_alias хранит адреса соседних элементов в списке.

#### 1.5.4. struct inode

##### Кратко

struct inode – дескриптор физического файла. Существует 2 типа inode.

1. Дисковый – описывает физ. файл.
2. inode в ядре, к-ый позволяет предварительно контролировать доступ к файлу.

~~regular file -> обращается к дисковому inode; pipe, socket и др. спец. файлы должны существовать не в superblock (доступ к ним должен осуществляться не через superblock)~~

##### Подробнее

~~Информация в inode ядра актуальна для ядра, для динамического обращения.~~

В дисковом inode хранится информация о физ. расположении файла на диске. У struct inode есть номер (inode number) – индекс (смещение к inode в таблице inode-ов)

В UNIX/LINUX имя файла не явл его идентификатором. Идентификатором файла явл. номер inode.

Физ. файлы, если говорить об обычных файлах хранятся на диске. Чтобы создать файл для него нужно создать inode, а затем для него должно быть выделено адр. пр-во диска.

Обращение к файлу — обращение к inode.

Иерархическая структура каталогов очень удобна для доступа к файлам.

Доступ к любому элементу каталога осуществляется по его индексу (номеру inode).

На диске должны храниться файлы содержащие информацию о регулярных файлах и файлах-директориях, чтобы эта информация мб представлена в виде дерева каталогов.

inode содержат информацию и о файлах-директориях, и об обычных файлах (об их расположении на диске).

Команды для получения информации об inode

ls -li — увидеть inode

df — увидеть список файловых систем, при этом в списке мы увидим, сколько inode содержит ФС, сколько айнодов используется, сколько свободных айнодов, % использования айнодов и точку монтирования.

Информацию из айнод файла можно получить в user mode, используя команду stat:

file: user.txt

Size: 78 Blocks: 8 IO Block: 4096

Device

Access

### определение struct inode

Но кроме обычных файлов в UNIX/Linux есть прогр. каналы, сокеты, гибкие ссылки и внешние устройства. Они имеют inode.

struct inode содержит union, в котором перечисляются типы файлов, и union, в котором перечисляются inode соответствующих фс

```
1 struct inode {
2     struct list_head i_hash;
3     struct list_head i_list;
4     struct list_head i_dentry;
5     ...
6     unsigned long i_ino;
7     atomic_t i_count;
8     kdev_t i_rdev;
9     umode_t i_mode;
10    ...
11    loff_t i_size;
12    ...
13    // информация о времени модификации и доступа к inode
14    ...
15    // 6 полей, связ с блоками(только для ядра)
16    ...
17    unsigned int i_blkbits; // битовая карта блока
18    unsigned long i_blksize; // размер блоков
19    unsigned long i_blocks; // кол-во блоков
20    ...
21    const struct inode_operations *i_op; //перечень функций определенных для
        работы с inode и с открытыми файлами
22    const struct file_operations *i_fop;
```

```

23  struct super_block  *i_sb;
24  ...
25  struct list_head i_devices;
26  struct pipe_inode_info *i_pipe;
27  struct block_device *i_bdev;
28  struct char_device *i_cdev;
29  ...
30  unsigned long i_state;
31  unsigned int i_flags;
32  ...
33  union //типы фс
34  {
35  struct minix_inode_info minix_i;
36  struct ext2_inode_info ext2_i;
37  ....
38  struct ntfs_inode_info ntfs_i;
39  struct msdos_inode_info msdos_i;
40  ...
41  struct nfs_inode_info nfs_i; // сетевая фс
42  struct ufs_inode_info ufs_i;
43  ...
44  struct proc_inode_info proc_i;
45  struct socket socket_i;
46  ...
47
48  }
49  };

```

i\_hash – Информацию dentry хешируется для ускорения обращения к файлу и его имени (фактически dentry – часть имени файла).

Как правило, пользователь многократно обращается к одному и тому же файлу. Для этого в ядре имеются соотв. односвязные списки.

i\_sb: Любой inode принадлежит конкретной ФС. След-но, struct inode должна содержать указатель на superblock.

### 1.5.5. Структура

#### `inode_operations`

Функции, определенные для работы с inode:

```
1  struct inode_operations {
2  struct dentry * (*lookup) (struct inode *,struct dentry *, unsigned int)
    ;
3
4  int (*create) (struct inode *,struct dentry *,
5              umode_t, bool);
6
7  int (*mkdir) (struct inode *,struct dentry *,
8              umode_t);
9
10 int (*rename) ( struct inode *, struct dentry *,
11                struct inode *, struct dentry *, unsigned int);
12 ...
13 } ;
```

Для поиска inode требуется, чтобы VFS вызывала функцию `lookup()` родительского каталога inode. Этот метод устанавливается конкретной реализацией файловой системы, в которой находится inode. Как только VFS находит требуемый dentry (и, следовательно, inode), можно открывать файл системным вызовом `open` или получать информацию о файле функцией `stat`, которая просматривает данные inode и передает часть их в пространство пользователя.

### 1.5.6. Кеш inode

Задача кеш inode — ускорение поиска и доступа.

Кеш inode в Linux:

#### 1. Глобальный хеш-массив

`inode_hash_table`

В нем каждый inode хешируется по значению указателя на `superblock` и 32-разрядному номеру inode. Если `superblock` отсутствует, то inode добавляется к двусвязному списку `anon_hash_chain`. Такие inode называют *анонимными*. Например сокеты, которые создаются вызовом ф-ции `sock_alloc`, которая вызывает `get_empty_inode()`

2. Глобальный список `inode_in_use` содержит допустимые `inode`, у которых `i_count > 0`, `i_nlink > 0`. Только что созданные `inode` добавляются в этот список.
3. Глобальный список `inode_unused`. В нем находятся допустимые `inode` с `i_count=0`
4. Для каждого `superblock`, который содержит `inode` с `i_count > 0`, `i_nlink > 0` и `i_state - dirty` создается список этих `inode`. `inode` отмечается как грязный, когда он был изменен. Он добавляется в список `f_dirty`, но только если `inode` был хеширован
5. SLAB cache называется `inode_cacher`

Список `f_dirty` позволяет сократить время синхронизации: `inode` хранятся на диске, есть список измененных `inode`. Очевидно, что сначала измененный `inode` записывается в список в памяти (кеш), и уже потом данные о нем (наз. `dirty`) переносятся на диск.

### 1.5.7. Структура `inode` каталогов

Если говорить о физ файлах, к-ые хранятся во вторичной памяти, то ФС необходима информация о директориях. Т.е. о каталоге, который представляет из себя дерево директорий. Начиная с корневой директории мы, проходя по этому дереву, в конечном итоге попадаем в ту директорию, которую используем как рабочую.

К этой директории существует путь, состоящий из поддиректорий, разделенных `'/'` (признак).

И только в конце в самой рабочей директории, находится файл, к которому можно обратиться по имени.

Впоследствии, после окончания работы с файлом и сохранении информации (надолго), мы обратимся к этому файлу.

Структура `inode` каталога

inode number 3470036	
.	3470036
..	3470017
folder1	3470031
file1	3470043
file2	3470023
folder2	3470024
file3	3470065
...	...

Имя файла сопоставлено с номером inode, имя директории – с номером inode. Именно обычные файлы и директории долговременно хранятся во вторичной памяти.

Невозможно не хранить имена директорий в долговременной памяти, так как иначе к ним не будет доступа (выключили комп, все имена исчезли и остались одни номера inode).

### 1.5.8. struct file

Существует 2 типа файлов — файл, к-ый лежит на диске и открытый файл. Открытый файл – файл, который открывает процесс

#### **Кратко**

struct file описывает открытый файл.

#### **Подробно**

Если файл просто лежит на диске, то через дерево каталогов можно увидеть это.

Увидеть можно только подмонтированную ФС.

А есть открытые файлы — файлы, с которыми работают процессы.

Открыть файл может только процесс. Если файл открывается потоком, то он в итоге все равно открывается процессом (как ресурс). Ресурсами владеет процесс.

### **Таблицы открытых файлов**

Помимо таблицы открытых файлов процесса (есть у каждого процесса), в системе есть одна таблица на все открытые файлы (на которую ссылаются таблицы процессов).

Причем в этой таблице на один и тот же файл (с одним и тем же inode) мб создано большое кол-во дескрипторов открытых файлов, т.к. один и тот же файл мб открыт много раз.

Каждое открытие файла с одним и тем же inode приведет к созданию дескриптора открытого файла.

При открытии файла его дескриптор добавляется:

1. в таблицу открытых файлов процесса (struct file\_struct)
2. в системную таблицу открытых файлов

Каждый дескриптор struct file имеет поле f\_pos. При работе с файлами это надо учитывать.

Один и тот же файл, открытый много раз без соотв. способов взаимоискл. будет атакован, что приведет к потере данных.

~~Гонки при разделении файлов — один и тот же файл мб открыт разными процессами.~~



## Определение struct file

```
1  struct file {
2  struct path      f_path;
3  struct inode      *f_inode;  /* cached value */
4  const struct file_operations *f_op;
5      ...
6  atomic_long_t      f_count; // кол-во жестких ссылок
7  unsigned int      f_flags;
8  fmode_t            f_mode;
9  struct mutex        f_pos_lock;
10 loff_t              f_pos;
11      ...
12 struct address_space *f_mapping;
13      ...
14 };
```

Как осуществляется отображение файла на физ. страницы? - дескриптор открытого файла имеет указатель на inode (файл на диске).

### Связь между struct file и struct file operations

Файл должен быть открыт. Соответственно для открытого файла должен быть создан дескриптор. В этом дескрипторе имеется указатель на struct file\_operations. Это либо стандартные (установленные по умолчанию) операции на файлах для конкретной файловой системы, либо зарегистрированные разработчиком (собственные функции работы с файлами собственной файловой системы).

```
1  struct file_operations {
2  struct module *owner;
3  loff_t (*llseek) (struct file *, loff_t, int);
4  ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
5  ssize_t (*write) (struct file *, const char __user *, size_t, loff_t *);
6      ...
7  int (*open) (struct inode *, struct file *);
8      ...
9  int (*release) (struct inode *, struct file *);
10      ...
11 } __randomize_layout;
```

Разработчики драйверов должны регистрировать свои функции read/write. В UNIX/Linux все файл как раз для того, чтобы свести все действия к однотипным операциям read/write

и не размножать их, а свести к большому набору операций.

Для регистрации своих функций используется(-лась) `struct file_operations`. С некоторой версии ядра 5.16+ (примерно) появилась `struct proc_ops`. В загружаемых модулях ядра можно использовать условную компиляцию:

```
1 #if LINUX_VERSION_CODE >= KERNEL_VERSION(5,6,0)
2 #define HAVE_PROC_OPS
3 #endif
4
5 #ifdef HAVE_PROC_OPS
6 static struct proc_ops fops = {
7     .proc_read = fortune_read ,
8     .proc_write = fortune_write ,
9     .proc_open = fortune_open ,
10    .proc_release = fortune_release ,
11 };
12 #else
13 static struct file_operations fops = {
14     .owner = THIS_MODULE,
15     .read = fortune_read ,
16     .write = fortune_write ,
17     .open = fortune_open ,
18     .release = fortune_release ,
19 };
20 #endif
```

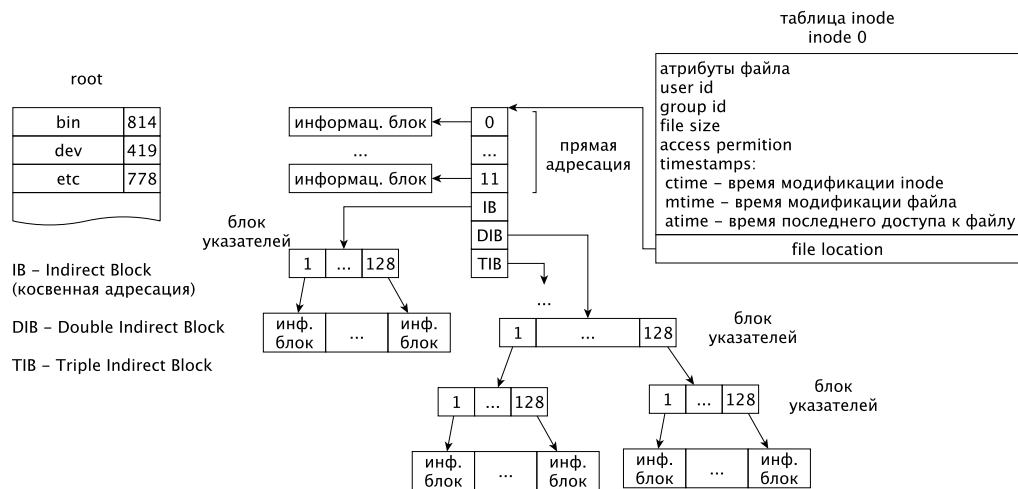
`proc_open` и `open` имеют одни и те же формальные параметры (указатели на `struct inode`, `struct file`). С другими функциями аналогично.

Зачем так сделано? — `proc_ops` сделана для того, чтобы не вешаться на `file_operations`, которые используются драйверами. Функции `file_operations` настолько важны для системы, что их решили освободить от работы с ФС `proc`.

## 1.6. Адресация файлов

### большого размера в файловой системе extX

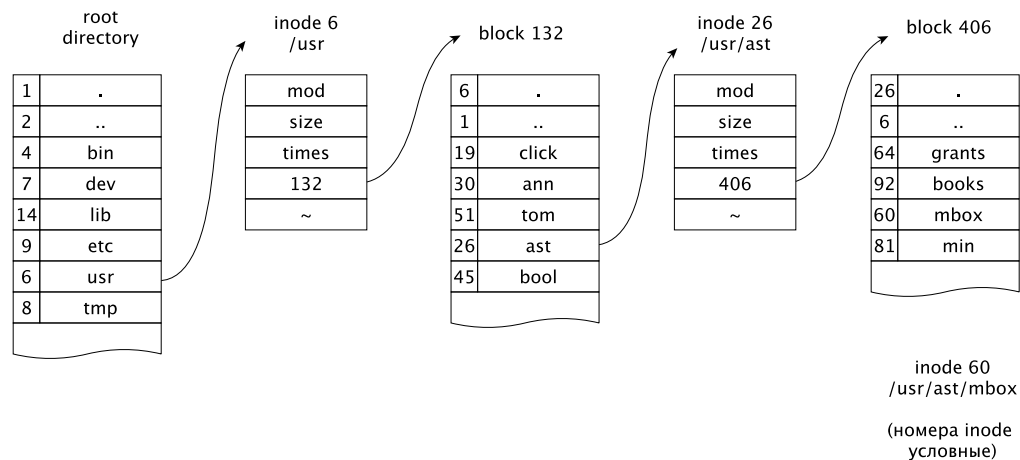
Чтобы иметь возможность хранить файлы очень большого размера, еще в 80-х была предложена схема с прямой и косвенной адресацией, двойной и тройной косвенной ад-



ресацией. Каждый адрес хранит адрес конкретного блока физического диска, в котором хранится информация, записанная в файл.

Прямая адресация — быстрый доступ к блоку. Время доступа — плата за возможность адресации больших файлов.

## 1.7. Пример, показывающий доступ к файлу /usr/ast/mbox



В дисковом inode хранятся адреса блоков, в которых находится информация, записанная в соответствующий файл. struct inode содержит поля mod, size, times, ..., которые определяют время последнего обращения, модификации и номер блока (адрес блока). inode данной директории (/usr) содержит номер блока, в котором находится информация директории /usr. Из содержимого блока получаем номер inode /usr/ast — 26. Получаем номер inode /usr/ast/mbox — 60. далее произойдет обращение к соответствующей информации.

Всего 3 элемента пути, и столько обращений к внешней памяти. Поэтому все нужно кешировать.

## 1.8. Монтирование файловых систем

Фактически VFS — интерфейс, с помощью которого ОС может работать с большим количеством файловых систем.

Основной такой работы (базовым действием) является монтирование: прежде чем файловая система станет доступна (мы сможем увидеть ее каталоги и файлы) она должна быть смонтирована.

Монтирование — подготовка раздела диска к использованию файловой системы. Для этого в начале раздела диска выделяется структура `super_block`, одним из полей которой является список `inode`, с помощью которого можно получить доступ к любому файлу файловой системы.

Когда файловая система монтируется, заполняются поля `struct vfsmount`, которая представляет конкретный экземпляр файловой системы, или, иными словами, точку монтирования. Точкой монтирования является директория дерева каталогов.

Вся файловая система должна занимать либо диск, либо раздел диска и начинаться с корневого каталога.

Любая файловая система монтируется к общему дереву каталогов (монтируется в поддиректорию).

И эта подмонтированная файловая система описывается суперблоком и должна занимать некоторый раздел жесткого диска ("это делается в процессе монтирования").

Когда файловая система монтируется, заполняются поля структуры `super_block`.

`super_block` содержит информацию, необходимую для монтирования и управления файловой системой.

Пример: мы хотим посмотреть содержимое флешки. Флешка имеет свою файловую систему, она может быть подмонтирована к дереву каталогов, и ее директории, поддиректории и файлы, которые мы сохраним на флешке, будут доступны. Потом мы достаем флешку. "Хорошая" система контролирует это и сделает демонтаж файловой системы за нас.

Если в системе присутствует некоторый образ диска `image`, а также создан каталог, который будет являться точкой монтирования файловой системы `dir`, то подмонтировать файловую систему можно, используя команду: `mount -o loop -t myfs ./image ./dir`

Параметр `-o` указывает список параметров, разделенных запятыми. Одним из прогрессивных типов монтирования, является монтирование через петлевое (`loop`, по сути, это «псевдоустройство» (то есть устройство, которое физически не существует — виртуальное блочное устройство), которое позволяет обрабатывать файл как блочное устройство) устройство. Если петлевое устройство явно не указано в строке (а как раз параметр `-o loop` это задает), тогда `mount` попытается найти неиспользуемое в настоящий момент петлевое устройство и применить его.

Аргумент следующий за `-t` указывает тип файловой системы.

`./image` - это устройство. `./dir` - это каталог.

`umount` — команда для размонтирования файловой системы:

`umount ./dir`

## 1.9. Команда `mount` и функции монтирования

Любая файловая система может быть подмонтирована много раз. Для этого ядро предоставляет функцию `mount`.

В ядре определено несколько функций `mount`.

```
1 typedef int (*fill_super_t)(struct super_block *, void *, int);
2 struct dentry *mount_bdev(struct file_system_type *fs_type, int flags ,
   const char *dev_name, void *data, fill_super_t fill_super);
3 struct dentry *mount_nodev(struct file_system_type *fs_type, int flags ,
   void *data, fill_super_t fill_super);
4 struct dentry *mount_single(struct file_system_type *fs_type, int flags ,
   void *data, fill_super_t fill_super);
```

- `mount_bdev` — для монтирования ФС, находящейся на блочном устройстве,
- `mount_nodev` — для монтирования ФС, не связанной ни с каким устройством,
- `mount_single` — для монтирования ФС, точки монтирования которой разделяют один единственный экземпляр ФС.

И в `mount_bdev()`, и в `mount_nodev()` вызывается функция `fill_super()`, которая выполняет основные действия по инициализации `struct super_block`.

Эти функции возвращают объект `dentry`, и этим объектом должен быть `root`. Для файловой системы необходимо создать `root`. Это позволит выполнить монтирование файловой системы. Для `root` надо создать `inode`.

#### Пример из лабораторной VFS:

```
1  #include <linux/module.h>
2  #include <linux/kernel.h>
3  #include <linux/init.h>
4  #include <linux/init_task.h>
5  #include <linux/fs.h>
6  #include <linux/slab.h>
7  #include <linux/time.h>
8
9  MODULE_LICENSE("GPL");
10 MODULE_AUTHOR("Ekaterina_Karpova");
11
12 #define MAGIC_NUM 0x12528391
13
14 #define CACHE_SIZE 1024
15 #define CACHE_NAME "kittyfs_cache"
16 static struct kmem_cache *cache = NULL;
17 static struct kittyfs_inode **inode_cache = NULL;
18 static size_t cache_index = 0;
19
20 static struct kittyfs_inode
21 {
22     int i_mode;
23     unsigned long i_ino;
24 } kittyfs_inode;
25
26 static void kittyfs_kill_sb(struct super_block *sb){
27     printk(KERN_INFO "+_kittyfs:_kill_super_block");
28     kill_anon_super(sb);
29 }
30
31 static void kittyfs_put_sb(struct super_block *sb)
32 {
33     printk(KERN_INFO "+_kittyfs:_superblock_destroy_called");
34 }
```

```

35
36 static struct super_operations const kittyfs_sb_ops = {
37     .put_super = kittyfs_put_sb ,
38     .statfs = simple_statfs ,
39     .drop_inode = generic_delete_inode ,
40 };
41
42 static struct inode *kittyfs_new_inode(struct super_block *sb, int ino ,
    int mode)
43 {
44     struct inode *res;
45     res = new_inode(sb);
46     if (!res)
47         return NULL;
48     res->i_ino = ino;
49     res->i_mode = mode;
50     res->i_atime = res->i_mtime = res->i_ctime = current_time(res);
51     res->i_op = &simple_dir_inode_operations;
52     res->i_fop = &simple_dir_operations;
53     res->i_private = &kittyfs_inode;
54
55     if (cache_index >= CACHE_SIZE)
56     {
57         return NULL;
58     }
59
60     inode_cache[cache_index] = kmem_cache_alloc(cache, GFP_KERNEL);
61     if (inode_cache[cache_index])
62     {
63         inode_cache[cache_index]->i_ino = res->i_ino;
64         inode_cache[cache_index]->i_mode = res->i_mode;
65         cache_index++;
66     }
67     return res;
68 }
69
70 static int kittyfs_fill_sb(struct super_block *sb, void *data, int silent)
71 {

```

```

72     struct dentry *root_dentry;
73     struct inode *root_inode;
74     sb->s_blocksize = PAGE_SIZE;
75     sb->s_blocksize_bits = PAGE_SHIFT;
76     sb->s_magic = MAGIC_NUM;
77     sb->s_op = &kittyfs_sb_ops;
78     root_inode = kittyfs_new_inode(sb, 1, S_IFDIR | 0755);
79     if (!root_inode)
80     {
81         printk(KERN_INFO "+_kittyfs:_cannot_make_root_inode");
82         return -ENOMEM;
83     }
84     root_dentry = d_make_root(root_inode);
85     if (!root_dentry)
86     {
87         printk(KERN_ERR "+_kittyfs:_cannot_make_root_dentry");
88         return -ENOMEM;
89     }
90     sb->s_root = root_dentry;
91     return 0;
92 }
93
94 static struct dentry *kittyfs_mount(struct file_system_type *type, int
    flags, const char *dev, void *data)
95 {
96     struct dentry *const root_dentry = mount_nodev(type, flags, data,
        kittyfs_fill_sb);
97     if (IS_ERR(root_dentry))
98         printk(KERN_ERR "+_kittyfs:_cannot_mount");
99     else
100         printk(KERN_INFO "+_kittyfs:_mount_successful");
101     return root_dentry;
102 }
103
104 static void kittyfs_slab_constructor(void *addr)
105 {
106     memset(addr, 0, sizeof(struct kittyfs_inode));
107 }

```



```

108
109 static struct file_system_type kittyfs_type = {
110     .owner = THIS_MODULE,
111     .name = "kittyfs",
112     .mount = kittyfs_mount,
113     .kill_sb = kittyfs_kill_sb,
114 };
115
116 static int __init kittyfs_init(void)
117 {
118     int err = register_filesystem(&kittyfs_type);
119
120     if (err != 0){
121         printk(KERN_ERR "+_kittyfs:_cannot_register_filesystem");
122         return err;
123     }
124
125     if ((inode_cache = kmalloc(sizeof(struct) kittyfs_inode)*CACHE_SIZE,
126         GFP_KERNEL)) == NULL)
127     {
128         printk(KERN_ERR "+_kittyfs:_Can't_kmalloc.\n");
129         return -ENOMEM;
130     }
131
132     if ((cache = kmem_cache_create(CACHE_NAME, sizeof(struct) kittyfs_inode
133         ), 0, SLAB_HWCACHE_ALIGN, kittyfs_slab_constructor)) == NULL)
134     {
135         printk(KERN_ERR "+_kittyfs:_cannot_create_cache");
136         kmem_cache_destroy(cache);
137         kfree(inode_cache);
138         return -ENOMEM;
139     }
140
141     printk(KERN_INFO "+_kittyfs:_module_loaded");
142     return 0;
143 }
144
145 static void __exit kittyfs_exit(void)

```

```

144 {
145     int err;
146     int i;
147
148     for (i = 0; i < cache_index; i++)
149         kmem_cache_free(cache, inode_cache[i]);
150
151     kmem_cache_destroy(cache);
152     kfree(inode_cache);
153
154     err = unregister_filesystem(&kittyfs_type);
155     if (err != 0)
156         printk(KERN_ERR "+_kittyfs:_cannot_unregister_filesystem");
157     else
158         printk(KERN_INFO "+_kittyfs:_module_is_unloaded");
159 }
160
161 module_init(kittyfs_init);
162 module_exit(kittyfs_exit);

```

## Загадки

Подмонтированные файловые системы через `proc` -> `/proc/mounts`

Зарегистрированные файловые системы через `proc` -> `/proc/filesystems`

Лоор — виртуальное блочное устройство (драйвер диска, который пишет данные не на физическое устройство, а в файл (образ диска))

Что означает `nODEV` — для монтирования не требуется блочное устройство

Какой минимальный набор действий для того, чтобы зарегистрировать собственную ФС? — заполнить поле `name` в `struct file_system_type` и зарегистрировать функцией ядра `register_filesystem`

Какой минимальный набор действий для того, чтобы можно было смонтировать собственную ФС? — заполнить поля `mount` и `name`. Мы используем функция ядра `mount_nODEV`. Для функции `mount` мы должны передать функцию `fill_sb`, которая будет заполнять `superblock` нашей ФС.

Если не указать оунера, то можно выгрузить модуль, не отмонтировав ФС, он считает количество монтирований (ссылок на модуль).

Для размонтирования обязательно указать `нейм` и `килл суперблок`.

Что за структура `superblock`, для чего нужна? — описывает ПОДМОНТИРОВАННУЮ файловую систему.

Что нужно сделать внутри `fill_sb`? Просто заполнить `superblock` же недостаточно? — функция `fill_sb` должна вернуть `dentry` КОРНЕВОГО каталога -> необходимо создать и проинициализировать `dentry` (`d_make_root`) -> необходимо создать и проинициализировать `inode`

Какой функцией создаете `inode`? — `new_inode`

Почему `blocksize = PAGE_SIZE`? — VFS расположена в оперативной памяти -> выделение оперативной памяти производится страницами

Слаб-кэш — это высокопроизводительный аллокатор памяти, используемый для ускорения доступа к файлу и для повторного использования проинициализированных слабых (брусков — брусок характеризуется фиксированным размером) (их не придется заново инициализировать)

Создали свою структуру для `инода`, так как кэш под системный `инод` создается автоматически.

Зачем на каждый `дентри` там нужен `айнод`? — ДЕНТРИ СОЗДАЕТСЯ НА ОСНОВЕ АЙНОДА: для долговременного хранения файла нужен `айнод`, так как он описывает физический файл, а `дентри` создается для `айнода` и нигде не хранится — информация о папках хранится на диске, директория — это тоже файл, специального типа `d`. А поскольку это файл, ему нужен `айнод`. `Аинод` содержит информацию об адресах блоков, в которых хранится информация. В случае директории это информация о других файлах (тут можно нарисовать пример с `/usr/ast/mbox`). — `dentry` существуют только в оперативной памяти, поэтому при выключении системы информация об элементах пути, если не хранить ее в виде файлов на диске (энергонезависимой части памяти), будет утеряна.

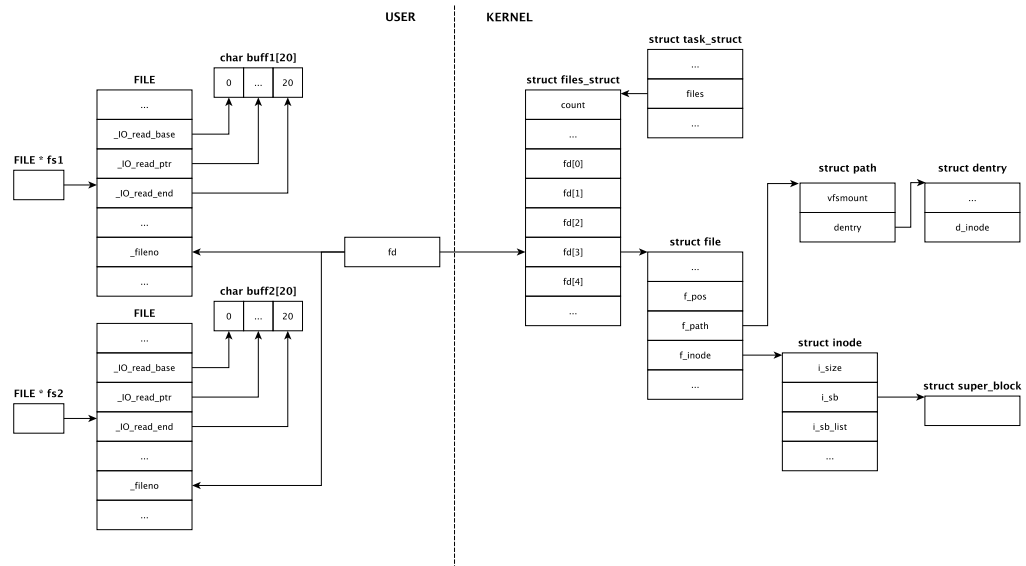
Что такое `стракт аинод`? — дескриптор физического файла

Что такое монтирование? — это выделение раздела диска под ФС, заполнение полей суперблока и помещение его в начало выделенного раздела. В суперблоке хранится массив `айнодов`, с помощью которого получаем доступ ко всем файлам ФС.

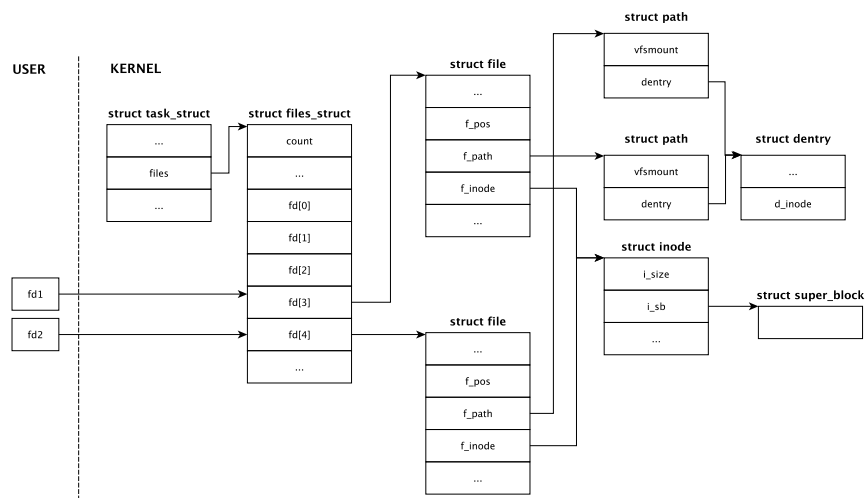
Основное действие при монтировании? — это выделение раздела диска под ФС

### **Связи структур из лабы на буферы**

*1 open, 2 fdopen, буферизация, читали 20 и 6 байт, выводили на экран*



*2 open, 2 дескриптора, без буферизации, посимвольно читали и выводили*



*2 open, без буферизации и с ней, или от а до з писали по очереди, 2 разных дескриптора, свои фпоз, записался либо по последнему фклоуз (при буф), либо по райт (посимвольно затирается без буф)*

