

struct dentry

Объекты `dentry` хранятся не в том виде, как они описаны, а как `inode` (как файлы) и создаются „на лету“;

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

1) free

Не содержит достоверной информации и не исп-ся VFS;
Соедв. обл-ть памяти обрабатывается
SLAB а `Posator'om`;

↳ `SLAB cache`

2) in use

В настоящее время ядром не исп-ся
Счётчик `d_count = 0`, но поле `d_inode` по-прежнему
указывает на соедв. индексный дескриптор;

Неиспользуемый объект `dentry` содержит достоверную информацию, но при необходимости он м.б. удалён и память м.б. освобождена;

3) in use

Используется ядром в текущий момент;
Счётчик `d_count > 0`;

У такого объекта есть `inode` (поле `d_inode` указывает на соедв. дескриптор);

Такой объект `dentry` не м.б. удалён;

4) negative (просто название)

Для него не существует соотв. entry inode;
Это возможно, если соотв. inode был
удалён с диска или объект entry был
создан как э-т пути несуществующему
файлу;

Тогда $d_inode = NULL$, но объект всё ещё
находится в кеше entry;

entry cache

Обращение к диску — очень длительная операция;

Информация об э-те пути (entry) имеет inode
(хранится на диске в виде файла);

Множеств э-тов пути к файлу м. б. много,
т.к. папки можно вкладывать одну в другую;
Каждое вложение — дополнительный объект
entry (элемент пути), т.е. файл;

При обращении к файлу просматриваются
все э-ты пути (каждый раз — обращение
к диску)

↳ поэтому объекты entry хранятся
в памяти (в кеше);

При этом они не удаляются просто так,
т.к. могут использоваться позже;

→ это существенно уменьшает время
обращения к конкретному файлу;

В Linux кеш dentry состоит из 2 типов структур:

- 1) set of dentry object в след. состояниях:
in use, unused, 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 включён в кеш dentry, то функция хеширования возвращ. нулевое значение;

Кеш dentry фактически действует как контроллер для cache inode;

И.е. 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 эл-та списка LRU хранятся соответственно в полях next и prev пер-ной dentry - unused;

Каждый объект dentry в состоянии in use включается в двусвязный список (поле i_dentry) соотв. объекта inode (i_dentry ← inode);

Поле d_alias хранит адреса соседних эл-тов в списке;

(struct list_head: `point next и prev`);
структура ядра

В ядре исп-ся двусвязные списки для обеспечения быстрого доступа за счёт реализации соотв. алгоритмов, простейшими из которых явл. бинарный поиск

Объекты `dentry in use` могут стать `negative`, если удаляется последний `hard link` на соотв. файл;

В этом случае объект `dentry` перемещается в список `lru unused_dentry`;

Хеш-таблица реализована как массив `dentry_hashtable`;

Spin-блокировка `dcache_lock` защищает структуру `dentry cache` от одновременного доступа к ней в многопроцессорных системах;

dentry_operations.

```
struct dentry_operations  
{  
    int (* d_revalidate)(struct dentry *, unsigned int);  
    ...  
    int (* d_hash)(const struct dentry *, unsigned int);  
    int (* d_compare)(const struct dentry *, unsigned int,  
        const char *, const struct qstr *);  
    int (* d_delete)(const struct dentry *);  
    int (* d_init)(const struct dentry *);  
    int (* d_release)(struct dentry *);  
    void (* d_input)(struct dentry *, struct inode *);  
    char * (* d_name)(struct dentry *, char *, int);  
    ...  
};
```

d_hash вызывается, когда VFS добавляет dentry в хеш-таблицу;

Первый dentry, который добавлен с помощью d_hash, явл. родительским каталогом;

d_compare вызывается для того, чтобы сравнить заданное имя с именем dentry;

При этом первый dentry явл. родителем первого dentry, который сравнивается;

В параметрах const struct qstr * - имя, с которым надо сравнить;

d_delete вызывается, если удаляется последняя ссылка на dentry;

d_init вызывается при создании;

d_release вызывается при освобождении;

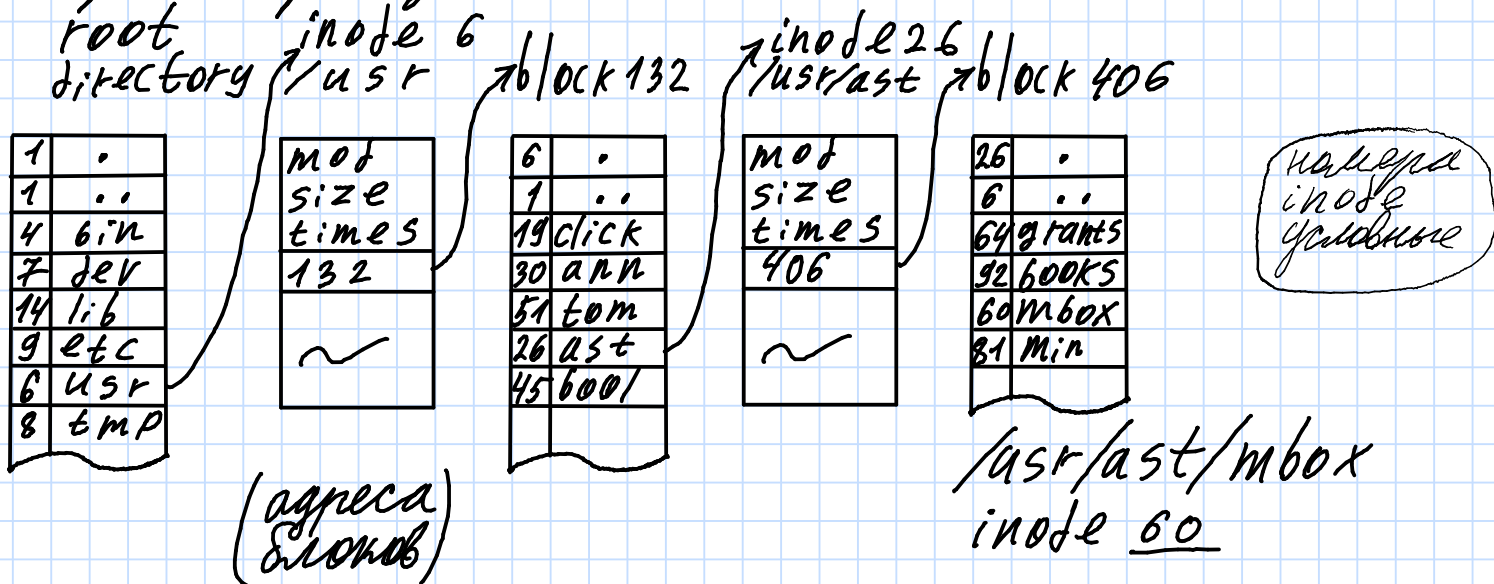
d_input вызывается, когда dentry меняет inode;

d_name вызывается, когда необходимо сгенерировать путь к эл-нту dentry;

Пример, показывающий действия, которые выполняются в системе для доступа к файлу mbox:

/usr/ast/mbox

Первое обращение:



В дисковом inode хранятся адреса блоков, в которых находится информация, записанная в соотв. файле;

struct inode содержит поля (mod, size, times, ...), которые определяют время последнего обращения, время последней модификации и номер блока (адрес блока);

inode данной директории (/usr) содержит номер блока, в котором находится информация директории /usr;

Из содержимого блока получаем inode^(номер)
/usr/ast - 26; (номер)

... Получаем inode /usr/ast/mbox - 60;
Далее произойдет обращение к соотв. информации;

Всего 3 э-та пути, и столько обращений
к внешней памяти...

Именно поэтому всё кэшируется;

struct file описывает открытые файлы,
которые нужны процессу для выт-я действий;

В системе существует одна таблица
открытых файлов;

struct file - дескриптор открытого файла;

Открыть файл может только процесс.

Если файл открывается потоком, то он
в итоге всё равно открывается процессом
(как ресурс);

Ресурсами владеет процесс;

```
struct file
{
```

```
    struct path f_path;
    struct inode *f_inode; /* cached value */
    const struct file_operations *f_op;
    ...
    atomic_t f_count; // кол-во установленных ссылок
    unsigned int f_flags;
    fmode_t f_mode;
    struct mutex f_pos_lock;
    loff_t f_pos;
    ...
    struct address_space *f_mapping;
```

```
};
```

курсовая?

?! как осуществ. отображ-е
.. файла на физ. страницы

Дескриптор открытого файла имеет указатель на inode (файл на диске);

```
struct file_operations
{
```

```
    struct module *owner;
    loff_t (*llseek)(struct file *, loff_t, int);
    ssize_t (*read)(struct file *, char __user *,
        size_t, loff_t *);
    ssize_t (*write)(struct file *, const char __user *,
        size_t, loff_t *);
```

```
    ...
    int (*open)(struct inode *, struct file *);
```

```
    ...
    int (*release)(struct inode *, struct file *);
    ...
```

```
}; (сравнить с proc_ops, см. семинары)
```


Разработчики драйверов должны регистрировать свои ф-ции read/write;

Зачем в UNIX/Linux всё так?

Для того, чтобы все действия свести к одноклассным операциям (read/write) и "не размыкаться" эти действия, а именно свести к небольшому набору операций;

Для регистрации своих ф-ций read/write в драйверах используется(-лась)
struct file_operations;

Р.с. прос виртуальная: информация создаётся на "лету";