

# 1. Билет №10

Создание собственной файловой системы. Структура, описывающая файловую систему и пример ее заполнения. Регистрация и deregistration файловых систем. Монтирование файловой системы. Структура `struct super_operations`. Структура `inode_operations`. Функции `simple` и `generic`. Точка монтирования. Функции монтирования. Функция `printk()`. Пример создания файловой системы, ее регистрация и монтирование (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 объемом 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. Создание собственной файловой системы

Чтобы создать собственную ф.с. В struct superblock есть поле file\_system\_type (структура ядра)

После описания ф.с., ядро предоставляет возможность зарегистрировать/удалить ф.с.

Структура описывающая конкретный тип ф.с. может быть только 1. При этом одна и та же ф.с. мб подмонтирована много раз.

Пример создания собств. ф.с.

Инициализация полей структуры file\_system\_type

```
1 struct file_system_type fs_type =  
2 {  
3     .owner = THIS_MODULE,  
4     .name = "myfs",  
5     .mount = myfs_mount,  
6     .kill_sb = kill_litter_super  
7 }
```

В функции myfs\_mount можно вызвать mount\_bdev/ mount\_nodev/ mount\_single

При создании ФС мы инициализируем лишь следующие поля:

- owner - нужно для организации счетчика ссылок на модуль (нужен, чтобы система не была выгружена, когда фс примонтирована).

- name - имя ФС.
- mount - указатель на функцию, которая будет вызвана при монтировании ФС.
- kill\_sb - указатель на функцию, которая будет вызвана при размонтировании ФС.

Разработчик ф.с. должен определить набор функций для работы с файлами в своей ф.с. Для этого используется struct file\_operations.

## 1.4. Регистрация и deregистрация файловой системы

Для регистрации ф.с. ядро предоставляет ф-цию register\_filesystem() (для удаления unregister\_filesystem). Функции register\_filesystem передается инициализированная структура file\_system\_type.

## 1.5. Монтирование файловой системы. Точка монтирования

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`

## 1.6. Структура `struct super_operations`

На любой структуре, описывающей объект ядра, определены функции для работы с объектом соответствующим типа (`struct file_operations`, `struct inode_operations`, `struct dentry_operations`).

```
1  struct super_operations {
2      struct inode *(*alloc_inode)(struct super_block *sb);
3      void (*destroy_inode)(struct inode *);
4
5      void (*dirty_inode) (struct inode *, int flags);
```

```

6      int (*write_inode) (struct inode *, struct writeback_control *wbc);
7      int (*drop_inode) (struct inode *);
8      void (*put_super) (struct super_block *);
9      ...
10     };

```

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

Ядро хранит копию таблицы inode-ов в памяти ядра, т.е. inode, к которому были обращения, кешируются для ускорения доступа к файлам. Сначала изменения вносятся в таблицу, к-ая находится в оперативной памяти.

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

write\_inode предназначена для записи inode на диск и помечает inode как измененный  
put\_super вызывается VFS при размонтировании фс.

## 1.7. Структура inode\_operations

Функции, определенные для работы с inode:

```

1      struct inode_operations {
2          struct dentry * (*lookup) (struct inode *, struct dentry *, unsigned
3              int);
4
5          int (*create) (struct inode *, struct dentry *,
6              umode_t, bool);
7
8          int (*mkdir) (struct inode *, struct dentry *,
9              umode_t);
10
11         int (*rename) ( struct inode *, struct dentry *,
12             struct inode *, struct dentry *, unsigned int);
13         ...
14     } ;

```

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.

## 1.8. Функции `simple` и `generic`

Функции `generic` - заглушки.

```
1 int generic_delete_inode(struct inode *inode)
2 {
3     return 1;
4 }
```

А функции `simple` выполняют некоторую простую последовательность действий, что освобождает разработчика от необходимости реализации этих действий каждый раз при реализации ФС.

```
1 int simple_statfs(struct dentry *dentry, struct kstatfs *buf)
2 {
3     buf->f_type = dentry->d_sb->s_magic;
4     buf->f_bsize = PAGE_CACHE_SIZE;
5     buf->f_namelen = NAME_MAX;
6     return 0;
7 }
```

## 1.9. Функция `printk()`

Функция `printk()` определена в ядре Linux и доступна модулям. Функция аналогична библиотечной функции `printf()`. Загружаемый модуль ядра не может вызывать обычные библиотечные функции, поэтому ядро предоставляет модулю функцию `printk()`. Функция пишет сообщения в системный лог.

## 1.10. Пример создания файловой системы, ее регистрация и монтирование (лаб. раб.)

```
1 static struct dentry *my_vfs_mount(struct file_system_type *type, int
    flags, const char *dev, void *data)
```

```

2  {
3      // my_vfs_fill_sb — указатель на функцию, которая будет вызвана из
4      mount_nodev для заполнения полей struct super_block
5      // nodev — не различает файловые системы символично-специальных и блочно-специальных устройств
6      struct dentry *const root_dentry = mount_nodev(type, flags, data,
7          my_vfs_fill_sb);
8          if (IS_ERR(root_dentry))
9      printk(KERN_ERR "+_can't mount_nodev\n");
10     else
11     printk(KERN_INFO "+_VFS_has_been_mounted.\n");
12
13     return root_dentry;
14 }
15
16 static struct file_system_type my_vfs_type = {
17     .owner = THIS_MODULE,
18     .name = "myvfs",
19     .mount = my_vfs_mount,
20     .kill_sb = my_kill_super,
21 };
22
23 static int __init my_vfs_init(void) {
24     int rc = register_filesystem(&my_vfs_type);
25     // ...
26     return 0;
27 }
28
29 static void __exit my_vfs_exit(void)
30 {
31     // ...
32     int rc = unregister_filesystem(&my_vfs_type);
33     // error handling
34 }
35
36 module_init(my_vfs_init);
37 module_exit(my_vfs_exit);

```