**Билет 10**

1. Создание собственной файловой системы. Структура, описывающая файловую систему и пример ее заполнения. Регистрация и дерегистрация 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.).

**Ответы**

**Структура, описывающая файловую систему и пример ее заполнения.**

Cтруктура, определяющая тип файловой системы - struct file\_system\_type.

struct [**file\_system\_type**](https://elixir.bootlin.com/linux/latest/ident/file_system_type) {

const char \*name;

int fs\_flags;

#define **[FS\_REQUIRES\_DEV](https://elixir.bootlin.com/linux/latest/ident/FS_REQUIRES_DEV)** 1 /\*требуется блочное устройство\*/

#define **[FS\_BINARY\_MOUNTDATA](https://elixir.bootlin.com/linux/latest/ident/FS_BINARY_MOUNTDATA)** 2 /\* с версии 2.6.5 монтируемые данные являются бинарными\*/

#define **[FS\_HAS\_SUBTYPE](https://elixir.bootlin.com/linux/latest/ident/FS_HAS_SUBTYPE)** 4

#define **[FS\_USERNS\_MOUNT](https://elixir.bootlin.com/linux/latest/ident/FS_USERNS_MOUNT)** 8 */\* Can be mounted by userns root \*/*

#define **[FS\_DISALLOW\_NOTIFY\_PERM](https://elixir.bootlin.com/linux/latest/ident/FS_DISALLOW_NOTIFY_PERM)** 16 */\* Disable fanotify permission events \*/*

#define **[FS\_RENAME\_DOES\_D\_MOVE](https://elixir.bootlin.com/linux/latest/ident/FS_RENAME_DOES_D_MOVE)** 32768 */\* FS will handle d\_move() during rename() internally. \*/*

int (\*init\_fs\_context)(struct **[fs\_context](https://elixir.bootlin.com/linux/latest/ident/fs_context)** \*);

const struct **[fs\_parameter\_spec](https://elixir.bootlin.com/linux/latest/ident/fs_parameter_spec)** \***[parameters](https://elixir.bootlin.com/linux/latest/ident/parameters)**;

struct **[dentry](https://elixir.bootlin.com/linux/latest/ident/dentry)** \*(\***[mount](https://elixir.bootlin.com/linux/latest/ident/mount)**) (struct **[file\_system\_type](https://elixir.bootlin.com/linux/latest/ident/file_system_type)** \*, int, const char \*, void \*);

void (\*kill\_sb) (struct **[super\_block](https://elixir.bootlin.com/linux/latest/ident/super_block)** \*); /\*прекращение доступа к суперблоку\*/

struct **[module](https://elixir.bootlin.com/linux/latest/ident/module)** \***[owner](https://elixir.bootlin.com/linux/latest/ident/owner)**; /\*счетчик ссылок на файловую систему\*/

struct **[file\_system\_type](https://elixir.bootlin.com/linux/latest/ident/file_system_type)** \* next;

struct **[hlist\_head](https://elixir.bootlin.com/linux/latest/ident/hlist_head)** fs\_supers;

struct **[lock\_class\_key](https://elixir.bootlin.com/linux/latest/ident/lock_class_key)** s\_lock\_key;

struct **[lock\_class\_key](https://elixir.bootlin.com/linux/latest/ident/lock_class_key)** s\_umount\_key;

struct **[lock\_class\_key](https://elixir.bootlin.com/linux/latest/ident/lock_class_key)** s\_vfs\_rename\_key;

struct **[lock\_class\_key](https://elixir.bootlin.com/linux/latest/ident/lock_class_key)** s\_writers\_key[**[SB\_FREEZE\_LEVELS](https://elixir.bootlin.com/linux/latest/ident/SB_FREEZE_LEVELS)**];

struct **[lock\_class\_key](https://elixir.bootlin.com/linux/latest/ident/lock_class_key)** i\_lock\_key;

struct **[lock\_class\_key](https://elixir.bootlin.com/linux/latest/ident/lock_class_key)** i\_mutex\_key;

struct **[lock\_class\_key](https://elixir.bootlin.com/linux/latest/ident/lock_class_key)** i\_mutex\_dir\_key;

};

Для каждого типа файловой системы существует только одна структура file\_system\_file, независимо от того, сколько таких файловых систем смонтировано и смонтирован ли хотя бы один экземпляр файловой системы.

Пример заполнения:

static struct file\_system\_type myfs\_type = {

.owner = THIS\_MODULE,

.name = "myfs",

.mount = myfs\_mount,

.kill\_sb = kill\_little\_super,

};

* Поле owner отвечает за счетчик ссылок на модуль, чтобы его нельзя было случайно выгрузить.
* Поле name хранит название файловой системы. Именно это название будет использоваться при ее монтировании.
* mount и kill\_sb два поля хранящие указатели на функции. Первая функция будет вызвана при монтировании файловой системы, а вторая при размонтировании.

kill\_little\_super() -- это generic-функция, предоставляемая VFS, она просто освобождает все внутренние структуры при размонтировании ФС; т.о. авторы простых виртуальных файловых систем не должны заботится об этом аспекте.

Функция myfs\_mount должна примонтировать устройство и вернуть структуру, описывающую корневой каталог файловой системы. Для этого она вызывает функцию mount\_bdev(type, flags, dev, data, myfs\_fill\_sb) (Если создается виртуальная файловая система, не связанная с каким-либо носителем, то нет необходимости использовать функцию mount\_bdev(). Вместо нее можно использовать функцию mount\_nodev()), где myfs\_fill\_sb - указатель на функцию, которая будет вызвана из mount\_bdev, чтобы проинициализировать суперблок. Сама функция myfs\_mount должна вернуть структуру dentry (переменная entry), представляющую корневой каталог нашей файловой системы, а создаст его функция myfs\_fill\_sb:

static int myfs\_fill\_sb(struct super\_block\* sb, void\* data, int silent)

{

struct inode\* root = NULL;

sb->s\_blocksize = PAGE\_SIZE;

sb->s\_blocksize\_bits = PAGE\_SHIFT;

sb->s\_magic = MYFS\_MAGIC\_NUMBER;

sb->s\_op = &myfs\_super\_ops;

root = myfs\_make\_inode(sb, S\_IFDIR | 0755);

if (!root)

{

printk (KERN\_ERR "MYFS inode allocation failed !\n") ;

return -ENOMEM;

}

root->i\_op = &simple\_dir\_inode\_operations;

root->i\_fop = &simple\_dir\_operations;

sb->s\_root = d\_make\_root(root) ;

if (!sb->s\_root)

{

printk(KERN\_ERR " MYFS root creation failed !\n") ;

iput(root);

return –ENOMEM;

}

return 0;

}

В первую очередь заполняется структура super\_block. Магическое число, по которому драйвер файловой системы может проверить, что на диске хранится именно та самая файловая система, а не что-то еще или прочие данные; операции для суперблока, его размер. Для магического числа можно использовать, например, значение 0x13131313.

*Для заполнения операций структуры super\_block используется структура super\_operations.* Ее реализация в данном примере:

static void myfs\_put\_super(struct super\_block \* sb)

{

printk(KERN\_DEBUG "MYFS super block destroyed!\n" ) ;

}

static struct super\_operations const myfs\_super\_ops = {

.put\_super = myfs\_put\_super,

.statfs = simple\_statfs,

.drop\_inode = generic\_delete\_inode,

};

В put\_super сохраняется деструктор нашего суперблока. Остальные поля заполняются заглушками из libfs. Функция myfs\_put\_super печатает в системный лог одну строчку. Функция myfs\_put\_super будет вызвана внутри kill\_block\_super перед уничтожением структуры super\_block, т.е. при размонтировании файловой системы.

Проинициализировав суперблок, функция myfs\_fill\_sb() выполняет построение корневого каталога нашей ФС. *Первым делом для него создается inode вызовом myfs\_make\_inode*. Он нуждается в указателе на суперблок и аргументе mode, который задает разрешения на создаваемый файл и его тип (маска S\_IFDIR говорит функции, что мы создаем каталог). Файловые и inode-операции, назначаемые новому inode, взяты из libfs, т.е. предоставляются ядром:

const struct **[inode\_operations](https://elixir.bootlin.com/linux/v4.4/ident/inode_operations)** **[simple\_dir\_inode\_operations](https://elixir.bootlin.com/linux/v4.4/ident/simple_dir_inode_operations)** = {

.**[lookup](https://elixir.bootlin.com/linux/v4.4/ident/lookup)** = **[simple\_lookup](https://elixir.bootlin.com/linux/v4.4/ident/simple_lookup)**,

};

Пример реализации функции myfs\_make\_inode:

static struct inode \* myfs\_make\_inode(struct super\_block \* sb, int mode)

{

struct inode \* ret = new\_ inode(sb);

if (ret)

{

inode\_init\_owner(ret, NULL, mode);

ret->i\_size = PAGE\_SIZE;

ret->i\_atime = ret->i\_mtime = ret->i\_ctime = current\_time(ret);

ret->i\_privat = myfs\_inode;

}

return ret;

}

Она размещает новую структуру inode (системным вызовом new\_inode()) и заполняет ее значениями: размером и временами (citme, atime, mtime). Аргумент mode определяет не только права доступа к файлу, но и его тип - регулярный файл или каталог.

Далее для корневого каталога создается структура dentry, через которую он помещается в directory-кэш. Суперблок имеет специальное поле, хранящее указатель на dentry корневого каталога, которое также устанавливается myfs\_fill\_sb.

**Регистрация и дерегистрация файловых систем.**

Зарегистрировать файловую систему можно с помощью системного вызова register\_filesystem(). Регистрация файловой системы выполняется в функции инициализации модуля. Для дерегистрации файловой системы используется функция unregister\_filesystem(), которая вызывается в функции выхода загружаемого модуля.

**Монтирование файловой системы**

Монтирование это – система действий, в результате которой файловая система устройства становиться доступной.

Монтировать файловую систему просто означает сделать конкретную файловую систему доступной в определенный момент в дереве каталогов Linux.

Монтирование корневой файловой системы (/) является частью процесса инициализации ОС. Все остальные файловые системы невозможно использовать до тех пор, пока они не будут смонтированы в определенных точках монтирования. Файловые системы, перечисленные в /etc/fstab монтируются в процессе загрузки системы. Файл / etc / fstab содержит список записей в следующей форме:

[File System] [Mount Point] [File System Type] [Options] [Dump] [Pass]

Для монтирования файловой системы в заданном месте (точке монтирования) в Linux используется команда mount в следующей форме:

mount [OPTION...] DEVICE\_NAME DIRECTORY

Прежде чем можно будет смонтировать файловую систему в определенный каталог, нужно войти в систему как root (некоторые файловые системы могут монтироваться обычным пользователем). Каталог, в который монтируется файловая система, должен существовать.

Чтобы отсоединить смонтированную файловую систему, используется команда umount, за которой следует либо каталог, в котором она была смонтирована (точка монтирования), либо имя устройства:

$ umount DIRECTORY[[1]](#footnote-1)

$ umount DEVICE\_NAME

**Точка монтирования**

Точкой монтирования является обычная директория дерева каталогов. После подключения файловой системы точка монтирования становится корневым каталогом смонтированной файловой системы.

**Функция printk()**

Позволяет классифицировать сообщения согласно, назначенным им, приоритетам. Обычно, приоритет задается через макроопределение.

В заголовочном файле <linux/kernel.h> определены восемь возможных уровней приоритета.

KERN\_EMERG

Используется для “непредвиденных” сообщений, особенно для тех, которые предшествуют краху.

KERN\_ALERT

Ситуация, требующая немедленного вмешательства.

KERN\_CRIT

Критическая ситуация. Часто связанная с серьезными неисправностями оборудования или программным проблемам.

KERN\_ERR

Используется для отчета об условиях возникшей ошибки. Драйвера устройств часто используют KERN\_ERR для сообщениях о проблемах, связанных с оборудованием.

KERN\_WARNING

Сообщение о проблемных ситуациях, которые сами по себе не создают проблем для системы.

KERN\_NOTICE

Ситуация не проблемная, но заслуживает внимания. Часто используется в сообщениях системы безопасности.

KERN\_INFO

Информационное сообщение. Используется в сообщениях о найденном оборудовании при загрузке драйверов.

KERN\_DEBUG

Используется для отладочных сообщений.

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

*Код 8 лабы*

1. [↑](#footnote-ref-1)