VFS

Примечание: При описании данной лабораторной учитывается, что была выполнена лабораторная по загрузке модуля ядра, и при реализации данной лабораторной необходимо использовать наработки предыдущей (например makefile).

Прежде всего необходимо подключить все необходимые библиотеки:

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/fs.h>
#include <linux/time.h>
```

Жизнь любой файловой системы начинается с регистрации. Зарегистрировать файловую систему можно с помощью системного вызова `register_filesystem`. Мы будем регистрировать файловую систему в функции инициализации модуля. Для разрегистрации файловой системы используется функция `unregister_filesystem`, и вызывать мы ее будем в функции выхода нашего модуля.

Обе функции принимают как параметр указатель на структуру `file_system_type` — она будет "описывать" файловую систему. Среди всех полей этой структуры нас интересуют лишь некоторые из них, поэтому при определении структуры инициализируем только следующие поля:

```
static struct file_system_type myfs_type = {
    .owner = THIS_MODULE,
    .name = "myfs",
    .mount = myfs_mount,
    .kill_sb = kill_block_super,
};
```

Где поле `owner` отвечает за счетчик ссылок на модуль, чтобы его нельзя было случайно выгрузить. Например, если файловая система была примонтирована, то выгрузка модуля может привести к краху, но счетчик ссылок не позволит выгрузить модуль пока он используется, т.е. пока мы не размонтируем файловую систему.

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. Достаточно реализовать всего одну, а вместо второй будем использовать `kill_block_super`, которую предоставляет ядро.

Теперь рассмотрим функцию `myfs_mount`. Она должна примонтировать устройство и вернуть структуру описывающую корневой каталог файловой системы:

```
static struct dentry *myfs_mount(struct file_system_type *type, int
   flags, char const *dev, void *data)
{
    struct dentry *const entry = mount_bdev(type, flags, dev, data,
       myfs_fill_sb);
    if (IS_ERR(entry))
        printk(KERN_ERR "MYFS mounting failed!\n");
    else
```

```
        printk(KERN_DEBUG "MYFS mounted!\n");
    return entry;
}
```

По факту, большая часть работы происходит внутри функции `mount_bdev`, но нас интересует лишь ее параметр `myfs_fill_sb` — это 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`.

Итак, `myfs_fill_sb` выглядит так:

```
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 значение 0x13131313.

Проинициализировав суперблок, `myfs_fill_sb` берется за построение корневого каталога нашей ФС. Первым делом для него создается `inode` вызовом `myfs_make_inode`, реализация которого будет показана ниже. 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, которые мы назначаем новому `inode`, взяты из `libfs`, т.е. предоставляются ядром.

Далее для корневого каталога создается структура `dentry`, через которую он помещается в directory-кэш. Заметим, что суперблок имеет специальное поле, хранящее указатель на

dentry корневого каталога, которое также устанавливается `myfs_fill_sb`.

Теперь по порядку: вернемся к структуре `super_block` и ее операциям, нас будет интересовать только одно ее поле — `put_super`. В `put_super` мы сохраним якобы «деструктор» нашего суперблока. Остальные поля мы заполним заглушками из `libfs`:

```
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,
};
```

Функция `myfs_put_super` ни делает ничего полезного, она используется исключительно чтобы напечатать в системный лог еще одну строчку. Функция `myfs_put_super` будет вызвана внутри `kill_block_super` (см. выше) перед уничтожением структуры `super_block`, т.е. при размонтировании файловой системы.

Теперь посмотрим, как работает `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);
    }

    return ret;
}
```

Примечание: В оригинальном источнике вместо `current_time(ret)` используется макрос `CURRENT_TIME`. Не на всех системах этот макрос работает корректно, так что в данном примере используется именно функция `current_time()`.

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

Остается только написать код для инициализации модуля и его выгрузки:

```
static int __init myfs_init(void)
{
    int ret = register_filesystem(&myfs_type);
    if (ret != 0)
    {
        printk(KERN_ERR "MYFS_MODULE cannot register filesystem!\n");
```

```
        return ret;
    }

    printk(KERN_DEBUG "MYFS_MODULE loaded!\n");
    return 0;
}

static void __exit myfs_exit(void)
{
    int ret = unregister_filesystem(&myfs_type);
    if (ret != 0)
        printk(KERN_ERR "MYFS_MODULE cannot unregister
           filesystem!\n");

    printk(KERN_DEBUG "MYFS_MODULE unloaded!\n");
}
```

Каркас файловой системы готов, пора его проверить. Сборка и загрузка драйвера файловой системы ничем не отличается от сборки и загрузки обычного модуля, т.е. используются уже знакомые команды `insmod` и `rmmod`.

Вместо реального диска для экспериментов будем использовать loop устройство. Это такой драйвер "диска", который пишет данные не на физическое устройство, а в файл (образ диска). Создадим образ диска, пока он не хранит никаких данных, поэтому все просто:

```
touch image
```

Кроме того, нужно создать каталог, который будет точкой монтирования (корнем) файловой системы:

```
mkdir dir
```

Теперь, используя этот образ, примонтируем файловую систему:

```
sudo mount -o loop -t myfs ./image ./dir
```

Если операция завершилась удачно, то в системном логе можно увидеть сообщения от модуля (`dmesg`). Чтобы размонтировать файловую систему делаем так:

```
sudo umount ./dir
```

И опять проверяем системный лог.

Ссылки на источники:
1. Облегченная версия: `https://habrahabr.ru/company/spbau/blog/218833/`;
2. Усложненная версия: `http://opennet.ru/base/dev/virtual_fs.txt`;