Рассматривается реализация файловой системы, монтируемой на блочное устройство, в операционных системах семейства Unix/Linux. Версия ядра: 5.8

Форматирование блочного устройства.

Чтобы адресовать данные на блочном устройстве системными вызовами VFS операционной системы, необходимо предварительно форматировать блочное устройство, то есть записать на него структуру суперблок и таблицу файловых дескрипторов.

Заметим, что структура суперблок и файловые дескрипторы, расположенные на диске, отличаются от соответствующих структур пространства ядра. Однако, структура суперблок пространства ядра ставится в соответствие структуре суперблок на диске. Аналогично, файловый дескриптор пространства ядра ставится в соответствие файловому дескриптору, расположенному на диске, при выполнении действий файловой системы над данными.

Структура суперблок, расположенная на диске:

```
typedef struct myrwfs_super_block
{
    byte4_t magic_number; /* Magic number - идентифицировать файловую
систему */
    byte4_t block_size_bytes; /* размер блока в байтах */
    byte4_t bdev_size_blocks; /* размер устройства в блоках */
    byte4_t entry_size_bytes; /* размер дескриптора файла в байтах */
    byte4_t entry_table_size_blocks; /* размер таблицы дескрипторов в
блоках */
    byte4_t entry_table_block_start; /* смещение к началу таблицы
дескрипторов в блоках - она расположена после суперблока на устройстве
*/
    byte4_t entry_count; /* количество дескрипторов в таблице
дескрипторов */
    byte4_t data_block_start; /* смещение к началу данных файловой
системы на блочном устройстве */
    byte4_t reserved[MYRWFS_BLOCK_SIZE_BYTES / 4 - 8];
} myrwfs_super_block_t;
```

Файловый дескриптор, расположенный на диске:

```
typedef struct myrwfs_file_entry
{
    char name[MYRWFS_FILENAME_LEN + 1];
    byte4_t size; /* размер дескриптора файла в байтах */
    byte4_t perms; /* разрешения */
    byte4_t blocks[MYRWFS_DATA_BLOCK_CNT];
} myrwfs_file_entry_t;
```

Определим структуру, которая ставит в соответствие стуктуру суперблок на диске и структуру суперблок пространства ядра:

```
typedef struct myrwfs_info
{
    struct super_block *vfs_sb; /* обратный указатель на структуру суперблок пространства ядра */
    myrwfs_super_block_t myrwfs_sb; /* указатель на структуру суперблок, расположенную на устройстве */
    byte1_t *used_blocks; /* указатель на количество блоков, помеченных как занятые */
} myrwfs_info_t;
```

Обратим внимание! Поле byte1_t *used_blocks; содержит массив флагов (значения 0 или 1), показывающих, занят ли соответствующий блок блочного устройства. При записи данных на блок, его необходимо пометить, как занятый значением 1. При освобождении блока - пометить как свободный значением флага 0.

Кроме того, определим функции, ставящие в соответствие файловые дескрипторы на диске и пространства ядра. Отметим также, что файловые дескриптор корневого каталога файловой системы, располагается в таблице дескрипторов с индекса 1, так как нулевой inode зарезервирован для представления невалидного указателя.

```
#define ROOT_INODE_NUM (1)
#define MYRWFS_TO_VFS_INODE_NUM(i) (ROOT_INODE_NUM + 1 + (i))
#define VFS_TO_MYRWFS_INODE_NUM(i) ((i) - (ROOT_INODE_NUM + 1))
```

Введем также следующее соотношение: таблица файловых дескрипторов занимает 10% размера блочного устройства.

```
// источник: https://sysplay.in/blog/linux-device-drivers/2014/07/file-systems-the-semester-project/
#define MYRWFS_ENTRY_FRACTION 0.10 /* 10% блоков резервируем под таблицу дескрипторов - эмпирически */
```

Поскольку структура суперблок расположена в первом блоке блочного устройства, смещение к началу таблицы дескрипторов определим равным единице.

```
#define MYRWFS_ENTRY_TABLE_BLOCK_START 1
```

Для форматирования блочного устройства реализуем функции записи на него структуры суперблок и таблицы файловых дескрипторов:

```
void write_super_block(int fd, myrwfs_super_block_t *myrwfs_sb) {
    write(fd, myrwfs_sb, sizeof(myrwfs_super_block_t));
}
```

```
void write_file_entries_table(int fd, myrwfs_super_block_t *myrwfs_sb)
{
    byte1_t block[MYRWFS_BLOCK_SIZE_BYTES];

    for (int i = 0; i < myrwfs_sb->block_size_bytes / myrwfs_sb-
>entry_size_bytes; i++) {
        memcpy(block + i * myrwfs_sb->entry_size_bytes, &fe,
sizeof(fe));
    }

    for (int i = 0; i < myrwfs_sb->entry_table_size_blocks; i++) {
        write(fd, block, sizeof(block));
    }
}
```

После выполнения форматирования блочного устройства, может быть выполнено монтирование файловой системы.

Реализация файловой системы.

Файловая система реализована в виде загружаемого модуля ядра.

Рассмотрим функции инициализации, чтения и записи данных, расположенных на отформатированном блочном устройстве. Эти функции будут вызваны в реализациях вызовов VFS высокого уровня.

Для понимания работы функций предварительно рассмотрим вызовы чтения и записи данных на блочное устройство. Операционные системы Unix/Linux предоставляют системные вызовы sb_bread, brelse и mark_buffer_dirty для работы с блочными устройствами. Вспомним, что блочное устройство является буферизованным - то есть запись данных пользователем выполняется в буфер, после чего операционная система неявно для пользователя копирует данные из буфера на устройство.

sb_bread: https://elixir.bootlin.com/linux/latest/ident/sb_bread

- reads a block with the given number in a `buffer_head` structure; in case of success returns a pointer to the `buffer_head` structure, otherwise it returns `NULL`; the size of the read block is taken from the superblock, as well as the device from which the read is done;
- Если необходимо прочитать данные с блочного устройства, не передавая информацию о файловой системе, под которую оно было отформатировано, необходимо воспользоваться аналогичной функцией bread.

brealse: https://www.unix.com/man-page/netbsd/9/brelse/

- Unbusy a buffer and release it to the free lists.

mark_buffer_dirty:
https://manpages.debian.org/experimental/linux-manual-4.12/mark_buffer_dirty.9.en.html#SYNOPSIS

- mark a buffer_head as needing writeout

Рассмотрим реализации функций чтения и записи на блочное устройство.

```
static int read_sb_from_myrwfs(myrwfs_info_t *info,
myrwfs_super_block_t *myrwfs_sb) {
    struct buffer_head *bh;
    if (!(bh = sb_bread(info->vfs_sb, 0))) {
        return -1;
    }
    memcpy(myrwfs_sb, bh->b_data, MYRWFS_BLOCK_SIZE_BYTES);
    brelse(bh);
    return 0;
}

static int read_entry_from_myrwfs(myrwfs_info_t *info, int ino,
myrwfs_file_entry_t *fe) {
    byte4_t offset = ino * sizeof(myrwfs_file_entry_t);
    byte4_t len = sizeof(myrwfs_file_entry_t);
    byte4_t block = info->myrwfs_sb.entry_table_block_start;
    byte4_t block_size_bytes = info->myrwfs_sb.block_size_bytes;
    byte4_t bd_block_size = info->vfs_sb->s_bdev->bd_block_size;
    byte4_t abs;
    struct buffer_head *bh;

    abs = block * block_size_bytes + offset;
    block = abs / bd_block_size;
    offset = abs % bd_block_size;
    if (offset + len > bd_block_size) {
        return -1;
    }

    if (!(bh = sb_bread(info->vfs_sb, block))) {
        return -1;
    }

    memcpy((void *) fe, bh->b_data + offset, len);
    brelse(bh);

    return 0;
```

```
}

static int write_entry_to_myrwfs(myrwfs_info_t *info, int ino,
myrwfs_file_entry_t *fe) {
    byte4_t offset = ino * sizeof(myrwfs_file_entry_t);
    byte4_t len = sizeof(myrwfs_file_entry_t);
    byte4_t block = info->myrwfs_sb.entry_table_block_start;
    byte4_t block_size_bytes = info->myrwfs_sb.block_size_bytes;
    byte4_t bd_block_size = info->vfs_sb->s_bdev->bd_block_size;
    struct buffer_head *bh;

    byte4_t abs = block * block_size_bytes + offset;
    block = abs / bd_block_size;
    offset = abs % bd_block_size;
    if (offset + len > bd_block_size) {
        return -1;
    }

    if (!(bh = sb_bread(info->vfs_sb, block))) {
        return -1;
    }

    memcpy(bh->b_data + offset, (void *) fe, len);
    mark_buffer_dirty(bh);
    brelse(bh);
    return 0;
}
```

Обратим особое внимание на следующие преобразования:

```
byte4_t offset = ino * sizeof(myrwfs_file_entry_t);
    byte4_t len = sizeof(myrwfs_file_entry_t);
    byte4_t block = info->myrwfs_sb.entry_table_block_start;
    byte4_t block_size_bytes = info->myrwfs_sb.block_size_bytes;
    byte4_t bd_block_size = info->vfs_sb->s_bdev->bd_block_size;
    struct buffer_head *bh;

    byte4_t abs = block * block_size_bytes + offset;
    block = abs / bd_block_size;
    offset = abs % bd_block_size;
    if (offset + len > bd_block_size) {
        return -1;
    }
```

Функции принимают на вход индекс файлового дескриптора в таблице файловых дескрипторов. Для адресации данных на блочном устройстве, этому индексу должно быть поставлено в соответствие смещение к блоку, на котором расположен файловый дескриптор.

С учетом реализаций функций чтения и записи файловых дескрипторов на блочное устройство, функции создания и поиска файлового дескриптора будут иметь следующий вид:

```
int myrwfs_create(myrwfs_info_t *info, char *fn, int perms,
myrwfs_file_entry_t *fe) {
    int ino, free_ino, i;

    free_ino = -1;
    for (ino = 0; ino < info->myrwfs_sb.entry_count; ino++) {

        if (read_entry_from_myrwfs(info, ino, fe) < 0)
            return -1;

        if (!fe->name[0]) {
            free_ino = ino;
            break;
        }
    }

    if (free_ino == -1) {
        printk(KERN_ERR "No entries left\n");
        return -1;
    }

    strncpy(fe->name, fn, MYRWFS_FILENAME_LEN);
    fe->name[MYRWFS_FILENAME_LEN] = 0;
    fe->size = 0;
    fe->perms = perms;
    for (i = 0; i < MYRWFS_DATA_BLOCK_CNT; i++) {
        fe->blocks[i] = 0;
    }

    if (write_entry_to_myrwfs(info, free_ino, fe) < 0)
        return -1;

    return MYRWFS_TO_VFS_INODE_NUM(free_ino);
}
```

```
int myrwfs_lookup(myrwfs_info_t *info, char *fn, myrwfs_file_entry_t
*fe) {
    for (int ino = 0; ino < info->myrwfs_sb.entry_count; ino++) {

        if (read_entry_from_myrwfs(info, ino, fe) < 0)
            return -1;

        if (!fe->name[0])
            continue;

        if (strcmp(fe->name, fn) == 0)
            return MYRWFS_TO_VFS_INODE_NUM(ino);
    }

    return -1;
}
```

При реализации файловой системы для семейства Unix\Linux необходимо реализовать функции инициализации и уничтожения структуры суперблок пространства ядра. Однако в случае монтируемой файловой системы, необходимо кроме того определить функции инициализации и уничтожения структуры суперблок, расположенной на диске:

```
int fill_myrwfs_info(myrwfs_info_t *info) {

    if (read_sb_from_myrwfs(info, &info->myrwfs_sb) < 0) {
        return -1;
    }

    if (info->myrwfs_sb.magic_number != MYRWFS_MAGIC_NUMBER) {
        printk(KERN_ERR "Invalid MYRWFS detected. Giving up.\n");
        return -1;
    }

    byte1_t *used_blocks = (byte1_t *)(vmalloc(info-
>myrwfs_sb.bdev_size_blocks));
    if (!used_blocks) {
        return -ENOMEM;
    }

    int i;
    for (i = 0; i < info->myrwfs_sb.data_block_start; i++) {
        used_blocks[i] = 1;
```

```
    }
    for (; i < info->myrwfs_sb.bdev_size_blocks; i++) {
        used_blocks[i] = 0;
    }

    myrwfs_file_entry_t fe;
    for (int i = 0; i < info->myrwfs_sb.entry_count; i++) {
        if (read_entry_from_myrwfs(info, i, &fe) < 0) {
            vfree(used_blocks);
            return -1;
        }

        if (!fe.name[0])
            continue;

        for (int j = 0; j < MYRWFS_DATA_BLOCK_CNT; j++) {
            if (fe.blocks[j] == 0) break;
            used_blocks[fe.blocks[j]] = 1;
        }
    }

    info->used_blocks = used_blocks;
    info->vfs_sb->s_fs_info = info;
    return 0;
}

void kill_mywfs_info(myrwfs_info_t *info) {
    if (info->used_blocks)
        vfree(info->used_blocks);
}
```

Обратим особое внимание на участок кода:

```
...
     byte1_t *used_blocks = (byte1_t *)(vmalloc(info-
>myrwfs_sb.bdev_size_blocks));
    if (!used_blocks) {
        return -ENOMEM;
    }

    int i;
    for (i = 0; i < info->myrwfs_sb.data_block_start; i++) {
        used_blocks[i] = 1;
```

```
    }
    for (; i < info->myrwfs_sb.bdev_size_blocks; i++) {
        used_blocks[i] = 0;
    }

...
```

Вспомним, что для записи данных в i-й блок блочного устройства, его необходимо пометить как занятый, установив значение флага used_blocks[i] в 1. С учетом этой информации, реализуем функции аллокации и освобождения блока:

```
int myrwfs_get_data_block(myrwfs_info_t *info) {
    int i;

    for (i = info->myrwfs_sb.data_block_start; i < info-
>myrwfs_sb.bdev_size_blocks; i++) {
        if (info->used_blocks[i] == 0) {
            info->used_blocks[i] = 1;
            return i;
        }
    }
    return -1;
}

void myrwfs_put_data_block(myrwfs_info_t *info, int i) {
    info->used_blocks[i] = 0;
}
```

Теперь мы можем реализовать функции высокого уровня для работы с нашей файловой системой. Реализации этих функций будут вызваны операционной системой при выполнении вызовов VFS - ls, cat, touch и т.д.

В функции инициализации структуры суперблок пространства ядра необходимо вызвать также функцию инициализации стурктуры суперблок, расположенной на диске:

```
static int myrwfs_fill_super(struct super_block *sb, void *data, int
silent) {

    printk(KERN_INFO "** MYRWFS: myrwfs_fill_super\n");

    myrwfs_info_t *info;
    if (!(info = (myrwfs_info_t *)(kzalloc(sizeof(myrwfs_info_t),
GFP_KERNEL))))
        return -ENOMEM;
```

```
    info->vfs_sb = sb;
    if (fill_myrwfs_info(info) < 0) {
        kfree(info);
        return -1;
    }

    sb->s_magic = info->myrwfs_sb.magic_number;
    sb->s_blocksize = info->myrwfs_sb.block_size_bytes;
    sb->s_blocksize_bits = log_base_2(info-
>myrwfs_sb.block_size_bytes);
    sb->s_type = &myrwfs;
    sb->s_op = &myrwfs_sops;

    myrwfs_root_inode = iget_locked(sb, ROOT_INODE_NUM);
    if (!myrwfs_root_inode) {
        kill_mywfs_info(info);
        kfree(info);
        return -1;
    }

    if (myrwfs_root_inode->i_state & I_NEW) {
        myrwfs_root_inode->i_op = &myrwfs_iops;
        myrwfs_root_inode->i_mode = S_IFDIR | S_IRWXU | S_IRWXG |
S_IRWXO;
        myrwfs_root_inode->i_fop = &myrwfs_dops;
        unlock_new_inode(myrwfs_root_inode);
    }

    sb->s_root = d_make_root(myrwfs_root_inode);
    if (!sb->s_root) {
        iget_failed(myrwfs_root_inode);
        kill_mywfs_info(info);
        kfree(info);
        return -ENOMEM;
    }

    return 0;
}
```

Аналогично, в функции, выполняющей уничтожение структуры суперблок пространства ядра, необходимо также выполнить освобождение дескрипторов структур, расположенных на диске.

```
static void myrwfs_put_super(struct super_block *sb) {
    myrwfs_info_t *info = (myrwfs_info_t *)(sb->s_fs_info);

    printk(KERN_INFO "** MYRWFS: myrwfs_put_super\n");
    if (info) {
        kill_mywfs_info(info);
        kfree(info);
        sb->s_fs_info = NULL;
    }
}
```

Обратим внимание также на то, что в функции примонтирования файловой системы, необходимо вызвать mount_bdev и указать флаг FS_REQUIRES_DEV в дескрипторе файловой системы:

```
static struct dentry *myrwfs_mount(struct file_system_type *fs, int
flags, const char *devname, void *data) {
    printk(KERN_INFO "** MYRWFS: myrwfs_mount: devname = %s\n",
devname);
    return mount_bdev(fs, flags, devname, data, &myrwfs_fill_super);
}

static struct file_system_type myrwfs = {
    name: "myrwfs",
    fs_flags: FS_REQUIRES_DEV,
    mount:  myrwfs_mount,
    kill_sb: kill_block_super,
    owner: THIS_MODULE
};
```

Функции создания и поиска файлового дескриптора в таблице файловых дескрипторов будут иметь следующий вид - отметим, что в реализации этих функций вызываются функции низкого уровня, выполняющие чтение и запись данных, расположенных на блочном устройстве.

```
static int myrwfs_inode_create(struct inode *parent_inode, struct
dentry *dentry, umode_t mode, bool excl) {
    char fn[dentry->d_name.len + 1];
    int perms = 0;
    myrwfs_info_t *info = (myrwfs_info_t *)(parent_inode->i_sb-
>s_fs_info);
    int ino;
    struct inode *file_inode;
    myrwfs_file_entry_t fe;
```

```
    printk(KERN_INFO "** MYRWFS: myrwfs_inode_create\n");

    strncpy(fn, dentry->d_name.name, dentry->d_name.len);
    fn[dentry->d_name.len] = 0;
    if (mode & (S_IRUSR | S_IRGRP | S_IROTH))
        mode |= (S_IRUSR | S_IRGRP | S_IROTH);
    if (mode & (S_IWUSR | S_IWGRP | S_IWOTH))
        mode |= (S_IWUSR | S_IWGRP | S_IWOTH);
    if (mode & (S_IXUSR | S_IXGRP | S_IXOTH))
        mode |= (S_IXUSR | S_IXGRP | S_IXOTH);
    perms |= (mode & S_IRUSR) ? 4 : 0;
    perms |= (mode & S_IWUSR) ? 2 : 0;
    perms |= (mode & S_IXUSR) ? 1 : 0;
    if ((ino = myrwfs_create(info, fn, perms, &fe)) == -1)
        return -1;

    file_inode = new_inode(parent_inode->i_sb);
    if (!file_inode)
    {
        myrwfs_remove(info, fn);
        return -ENOMEM;
    }
    printk(KERN_INFO "** MYRWFS: Created new VFS inode for #%d, let's
fill in\n", ino);
    file_inode->i_ino = ino;
    file_inode->i_size = fe.size;
    file_inode->i_mode = S_IFREG | mode;
    file_inode->i_fop = &myrwfs_fops;
    if (insert_inode_locked(file_inode) < 0)
    {
        make_bad_inode(file_inode);
        iput(file_inode);
        myrwfs_remove(info, fn);
        return -1;
    }
    d_instantiate(dentry, file_inode);
    unlock_new_inode(file_inode);

    return 0;
}

static struct dentry *myrwfs_inode_lookup(struct inode *parent_inode,
struct dentry *dentry, unsigned int flags) {
```

```
    myrwfs_info_t *info = (myrwfs_info_t *)(parent_inode->i_sb-
>s_fs_info);
    char fn[dentry->d_name.len + 1];
    int ino;
    myrwfs_file_entry_t fe;
    struct inode *file_inode = NULL;

    printk(KERN_INFO "** MYRWFS: myrwfs_inode_lookup\n");

    if (parent_inode->i_ino != myrwfs_root_inode->i_ino)
        return ERR_PTR(-ENOENT);z

    strncpy(fn, dentry->d_name.name, dentry->d_name.len);
    fn[dentry->d_name.len] = 0;
    if ((ino = myrwfs_lookup(info, fn, &fe)) == -1)
      return d_splice_alias(file_inode, dentry);

    file_inode = iget_locked(parent_inode->i_sb, ino);
    if (!file_inode)
        return ERR_PTR(-EACCES);

    if (file_inode->i_state & I_NEW) {
        file_inode->i_size = fe.size;
        file_inode->i_mode = S_IFREG;
        file_inode->i_mode |= ((fe.perms & 4) ? S_IRUSR | S_IRGRP |
S_IROTH : 0);
        file_inode->i_mode |= ((fe.perms & 2) ? S_IWUSR | S_IWGRP |
S_IWOTH : 0);
        file_inode->i_mode |= ((fe.perms & 1) ? S_IXUSR | S_IXGRP |
S_IXOTH : 0);
        file_inode->i_fop = &myrwfs_fops;
        unlock_new_inode(file_inode);
    }

    d_add(dentry, file_inode);
    return NULL;
}

static struct inode_operations myrwfs_iops = {
    create: myrwfs_inode_create,
    lookup: myrwfs_inode_lookup
};
```