Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления
КАФЕДРА Программное обеспечение ЭВМ и информационные технологии

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОМУ ПРОЕКТУ

НА ТЕМУ:

“Виртуальная файловач система”

Студент ИУ7-76Б _____ Нгуен Ф. С.
(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель курсового проекта _____ Рязанова Н. Ю.
(Подпись, дата) (И. О. Фамилия)

2021

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

УТВЕРЖДАЮ

Заведующий кафедрой ИУ7
(Индекс)

_____

И.В.Рудаков
(И.О.Фамилия)

« ___ » _____ 2021 г.

З А Д А Н И Е
на выполнение курсового проекта

по дисциплине_____ Операционные системы_____

Виртуальная файловач система
(Тема курсового проекта)

Студент_____ Нгуен Ф. С. гр. ИУ7-76Б_____
(Фамилия, инициалы, индекс группы)

График выполнения проекта: 25% к _4 нед., 50% к _7_ нед., 75% к _11нед., 100% к 14 нед.

1. Техническое задание

Спроектировать и разработать в операционной системе Linux файловую систему с операциями: Монтирование, Создание, Удаление, Переименование, ...

2. Оформление курсового проекта

2.1. Расчетно-пояснительная записка на 25-30 листах формата А4.

Расчетно-пояснительная записка должна содержать постановку введение, аналитическую часть, конструкторскую часть, технологическую часть, экспериментально-исследовательский раздел, заключение, список литературы, приложения.

2.2. Перечень графического материала (плакаты, схемы, чертежи и т.п.)__На защиту проекта должна быть представлена презентация, состоящая из 15-20 слайдов. На слайдах должны быть отражены: постановка задачи, использованные методы и алгоритмы, расчетные соотношения, структура комплекса программ, диаграмма классов, интерфейс, характеристики разработанного ПО, результаты проведенных исследований.

Дата выдачи задания « ____ » _____ 20__ г.

Руководитель курсового проекта _____ ____**Рязанова Н. Ю.**
(Подпись, дата) (И.О.Фамилия)

Студент _____ _____**Нгуен Ф. С.**
(Подпись,дата) (И.О.Фамилия)

Оглавление

Введение

Один из ключей к успеху Linux - это его способность комфортно сосуществовать с другими системами. Вы можете прозрачно монтировать диски или разделы, на которых размещены форматы файлов, используемые Windows, другими системами Unix или даже системами с небольшой долей рынка, такими как Amiga. Linux удается поддерживать несколько типов дисков так же, как и другие варианты Unix, благодаря концепции, называемой виртуальной файловой системой.

Идея виртуальной файловой системы заключается в том, что внутренние объекты, представляющие файлы и файловые системы в памяти ядра, содержат широкий спектр информации; есть поле или функция для поддержки любой операции, предоставляемой любой реальной файловой системой, поддерживаемой Linux. Для каждой вызываемой функции чтения, записи или другой вызываемой функции ядро заменяет фактическую функцию, которая поддерживает собственную файловую систему Linux, файловую систему NT или любую другую файловую систему, в которой находится файл.

В этой работе обсуждаются цели, структура и реализация виртуальной файловой системы Linux. Он фокусируется на трех из пяти стандартных типов файлов Unix, а именно на обычных файлах, каталогах и символических ссылках.

1. Аналитическая часть

1.1. Постановка задачи

В соответствии с заданием на курсовую работу по курсу Операционные системы необходимо разработать виртуальную файловую систему.

Файловая система должна иметь следующие возможности:

- Для обычных файлов создавать удалять переименовать открыть прочитать записать
- Создавать удалять переименовать директории И поддтректории
- Создавать удалять переименовать символические ссылки

Для решения поставленной задачи необходимо

1. Проанализировать особенности файловой подсистемы Linux и интерфейса VFS
2. Проанализировать структуру файловой системы и структуры описывающие её элементы: *superblock, dentry, inode, file*
3. Разработать алгоритмы и структуры ПО
4. Разработать ПО виртуальной файловой системы
5. Наследовать работу ПО

1.2. Загружаемые модули

Одной из важных особенностей ос linux является способность расширения функциональности ядра без её перекомпиляции. Это обеспечивается возможность написания загружаемых модулей ядра которые загружаются в ядро и становятся его частью. Загружаемый модуль ядра представляет объектный код, который динамически подгружается в ядро командой «insmod» и удаляется из ядракомандой «rmmod».

Виртуальная файловая система может быть реализована в виде загрежаемого модуля ядра.

Ос Linux представляет специальные функции ядра для регистрации файловой системы и её дерегистрации

Регистрация файловой системы выполняется в функции инициализации модуля. Функция ядра **register_filesystem** предназначена для регистрации файловой системы и имеет следующий прототип:

int register_filesystem (struct file_system_type * fs);

Функция ядра **unregister_filesystem** предназначена для дерегистрации файловой системы и имеет следующий прототип:

int unregister_filesystem (struct file_system_type * fs);

Дерегистрации файловой системы вызывается в функции выхода загружаемого модуля.

Обе функции принимают как параметр указатель на структуру **file_system_type**, которая "описывает" создаваемую файловую систему. Эта структура описана в файле include / linux / fs.h.

Листинг 1. Описание структуры *file_system_type*

```
1. struct file_system_type {
2.     struct module *owner;
3.     const char *name;
4.     struct dentry *(*mount) (struct file_system_type *, int,
5.                 const char *, void *);
6.     void (*kill_sb) (struct super_block *);
7.     int fs_flags;
8.     struct file_system_type * next;
9.         }
```

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 функция будет вызвана при монтировании файловой системы

kill_sb функция будет вызвана при размонтированиифайловой системы

1.3. **Файловые подсистемы Linux:**

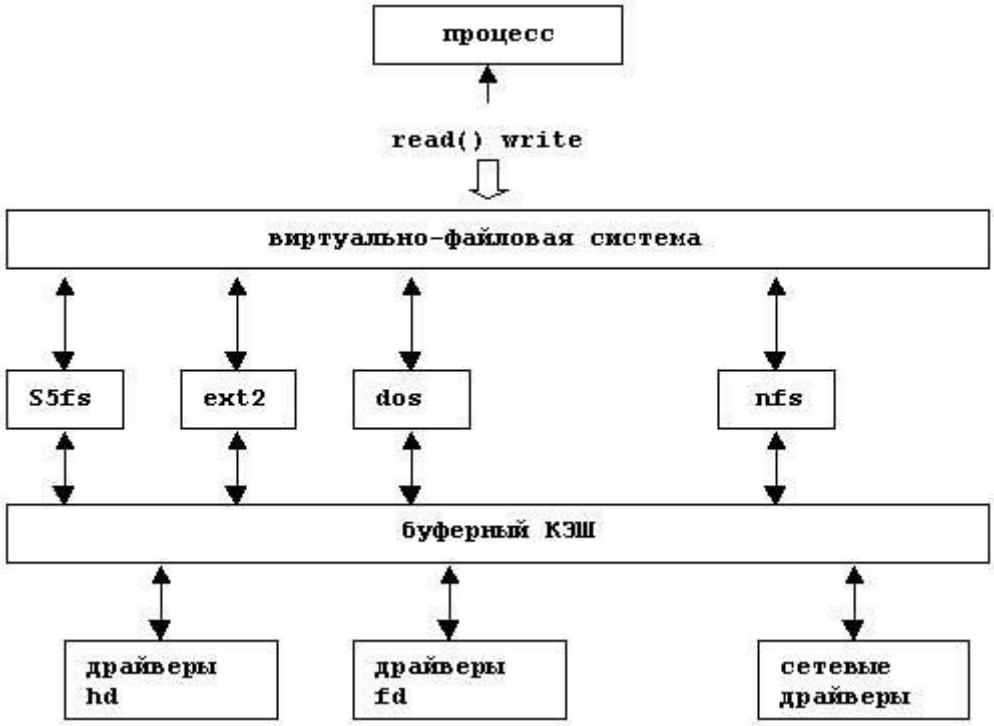
В Linux все файлы и каталоги размещаются в древовидной структуре. Самый верхний каталог файловой системы именуется как корневой (root) или просто “/” . Все прочие каталоги в Linux доступны из корневого и организованы в виде иерархической структуры.

Ext2, Ext3, Ext4 или Extended Filesystem - это стандартная файловая система для Linux. Она была разработана еще для Minix. Она самая стабильная из всех существующих, кодовая база изменяется очень редко и эта файловая система содержит больше всего функций. В ней было внесено много улучшений, в том числе увеличен максимальный размер раздела до одного экзабайта.

1.4.**Интерфейс VFS**

VFS содержит набор функций, которые должна поддерживать любая файловая система. Этот интерфейс состоит из ряда операций, которые оперируют тремя типами объектов: файловые системы, индексные дескрипторы и открытые файлы.

Указатели на функции, расположенные в дескрипторе файловой системы, позволяют VFS получить доступ к внутренним функциям файловой системы. Используются еще два типа дескрипторов: это inode и дескриптор открытого файла. Каждый из них содержит информацию, связанную с используемыми файлами и набором операций, используемых кодом файловой системы.



1.5. Структуры, описывающие элементы файловой системы

1.5.1. Объекты *суперблока*

Эти структуры используются как «шлюз» к драйверам файловой системы, беря абстрактные идеи файловой системы и предоставляя ссылку на реализацию этих идей для каждой из файловых систем.

Листинг 2. Описание структуры *super_block*

```
1.struct super_block {
2.     unsigned long        s_blocksize;
3.     unsigned char        s_blocksize_bits;
4.     unsigned long        s_magic;
5.     const struct super_operations   *s_op;
6.     …
7.}
```

S_magic - 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;

s_blocksize : размер блока в байтах

s_blocksize_bits: размер блока в битах

s_op: указатель на структуру super_operations, которая содержит специальные методы связанные с суперблоком

Листинг 3. Описание структуры *super_operations*

```
1.struct super_operations {
2.    struct inode *(*alloc_inode)(struct super_block *sb);
3.    void (*destroy_inode)(struct inode *);
4.    int (*statfs) (struct dentry *, struct kstatfs *);
5.    void (*put_super) (struct super_block *);
6.    void (*write_inode) (struct inode *, int);
7.    int (*sync_fs) (struct super_block *);
8.}
```

1.5.2. Объекты *Inode*

Inode представлен структурой struct inode и операциями с ним, определенными в структуре ***struct inode_operations***.

Вся информация, необходимая файловой системе для обработки файла, включается в структуру данных, называемую индексным дескриптором. Имя файла - это случайно назначенная метка, которую можно изменить, но индексный дескриптор уникален для файла и остается неизменным, пока файл существует.

Листинг 4. Описание структуры *inode*

```
1.struct inode {
2.     owner
3.     umode_t i_mode
4.     const struct inode_operations    *i_op;
5.     const struct file_operations     *i_fop;
6.     loff_t           i_size;
7.     struct timespec64   i_atime;
8.     struct timespec64   i_mtime;
9.     struct timespec64   i_ctime;
10.         void           *i_private;
11.    }
```

I_mode : Тип файла и права доступа

I_size : Длина файла в байтах

I_atime : Время последнего доступа к файлу

I_mtime: Время последней записи файла

I_ctime: Время последнего изменения inode

I_op: Методы, связанные с объектом inode

I_fop: Методы, связанные с объектом file

Ядро Linux не может жестко запрограммировать конкретную функцию для обработки операции. Вместо этого он должен использовать указатель для каждой операции; указатель указывает на правильную функцию для конкретной файловой системы, к которой осуществляется доступ.

1.5.3. Структура inode_operations

Листинг 5. Описание структуры *inode_operations*

```
1. struct inode_operations {
2.     struct dentry * (*lookup) (struct inode *,struct dentry
   *, unsigned int);
3.     int (*create) (struct inode *,struct dentry *, umode_t,
   bool);
4.     int (*link) (struct dentry *,struct inode *,struct dentry
   *);
5.     int (*unlink) (struct inode *,struct dentry *);
6.     int (*symlink) (struct inode *,struct dentry *, const
   char *);
7.     int (*mkdir) (struct inode *,struct dentry *, umode_t);
8.     int (*rmdir) (struct inode *,struct dentry *);
9.     int (*rename) (struct inode *, struct dentry *, struct
   inode *, struct dentry *, unsigned int);
```

Только что перечисленные методы доступны для всех возможных индексных дескрипторов и типов файловых систем. Тем не мение, только часть из них применима к любому данному inode и файловой системе; поля, соответствующие нереализованные методы устанавливаются в NULL. Нам надо реализовать пропущенные методы.

Create(dir, dentry, mode, excl): Создает новый inode связанного с объектом dentry в некотором каталоге.

Lookup(dir, dentry, flags): Ищет в каталоге индексный дескриптор, соответствующий имени файла, включенному в объект dentry.

Mkdir(dir, dentry, mode): Создает новый inode для каталога, связанного с объектом dentry в некотором каталоге.

Rmdir (dir, dentry): Удаляет из каталога подкаталог, имя которого включено в объект dentry.

Link(old_dentry, dir, new_dentry): Создает новую жесткую ссылку, которая ссылается на файл, указанный в old_dentry в каталоге dir; новая жесткая ссылка имеет имя, указанное в new_dentry.

Unlink(dir, dentry): Удаляет жесткую ссылку на файл, указанный объектом dentry, из каталога.

Symlink(dir, dentry, symname): Создает новый inode для символической ссылки, связанной с объектом dentry в некотором каталоге.

Rename(old_dir, old_dentry, new_dir, new_dentry, flags): Перемещает файл, идентифицированный old_entry, из каталога old_dir в каталог new_dir. Новое имя файла включается в объект dentry, на который указывает new_dentry.

1.5.4. Объекты файлов

Файловый объект описывает, как процесс взаимодействует с файлом, который он открыл. Объект создается при открытии файла и состоит из файловой структуры. Основная информация, хранящаяся в файловом объекте, - это указатель файла, то есть текущая позиция в файле, с которой будет выполняться следующая операция. Поскольку несколько процессов могут обращаться к одному и тому же файлу одновременно, указатель файла не может храниться в объекте inode.

Листинг 6. Описание структуры *file*

```
1.struct file {
2.    struct path     f_path;
3.    struct inode *  inode
4.    struct file_operations*     f_ops;
5.    loff_t  f_pos;
6.    void *  private_data
```

```
7.     spinlock_t  f_lock;
8.     ….
9. }
```

F_pos : Текущее смещение файла (указатель файла)

F_op: Указатель на таблицу операций с файлами

Листинг 7. Описание структуры *file_operations*

```
1. struct file_operations {
2.     ssize_t (*read) (struct file *, char __user *, s
3.     ssize_t (*write) (struct file *, const char __use
4.     int (*open) (struct inode *, struct file *);
5. }
```

Структура *file_operations* содержит указатели на функции драйвера, которые отвечают за выполнение различных операций с устройством.

1.5.5. Объект dentry

Объект dentry создается ядром для каждого компонента пути, который ищет процесс; объект dentry связывает компонент с его соответствующим индексом. Например, при поиске пути /tmp/test ядро создает объект dentry для корневого каталога /, второй объект dentry для записи tmp корневого каталога и третий объект dentry для записи test в /tmp каталог.

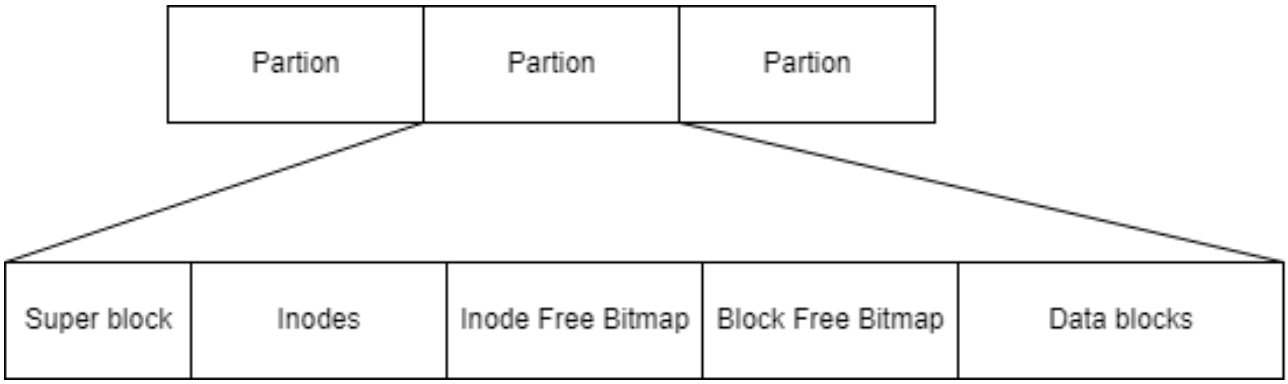
Листинг 8. Описание структуры dentry

```
1.Struct dentry {
2.     unsigned int d_flags;
3.     struct qstr d_name;
4.     struct inode *d_inode;
5.     struct  list_head  d_child;     /*  child  of
  parent list */
6.     struct list_head d_subdirs;
7.     struct dentry *d_parent;
8. }
```

Объекты Dentry хранятся в кэше распределителя slab, называемом dentry_cache; Таким образом, объекты dentry создаются и уничтожаются вызовом kmem_cache_alloc () и kmem_cache_free ().

1.6. **Распределение памяти:**

Каждый раздел (partition) представляет собой файловую систему. ФС содержит послед овательность блоков. Каждый блок имеет размер 4 KiB



Суперблок - это первый блок раздела (блок 0). Он содержит метаданные раздела, такие как количество блоков, количество inodes, количество свободных inodes / блоков,...

Inode Store - Содержит все inodes раздела. Максимальное количество inodes равно количеству блоков раздела.

Два bitmaps будут использоваться для управления свободным пространством:

- inode free bitmap - для свободных inodes
- block free bitmap - для свободных блоков данных

1.7. **Вывод:**

В этом разделе рассмотрены особенности файловой подсистемы Linux, интерфейса VFS и структуры описывающие элементы файловой системы

2. Конструкторская часть

2.1. **Инициализация и установка суперблока**

Struct ***file_system_type*** - основная структура данных, описывающая файловую систему в ядре. **Mount** и **kill_sb** - 2 функции управления суперблоком.

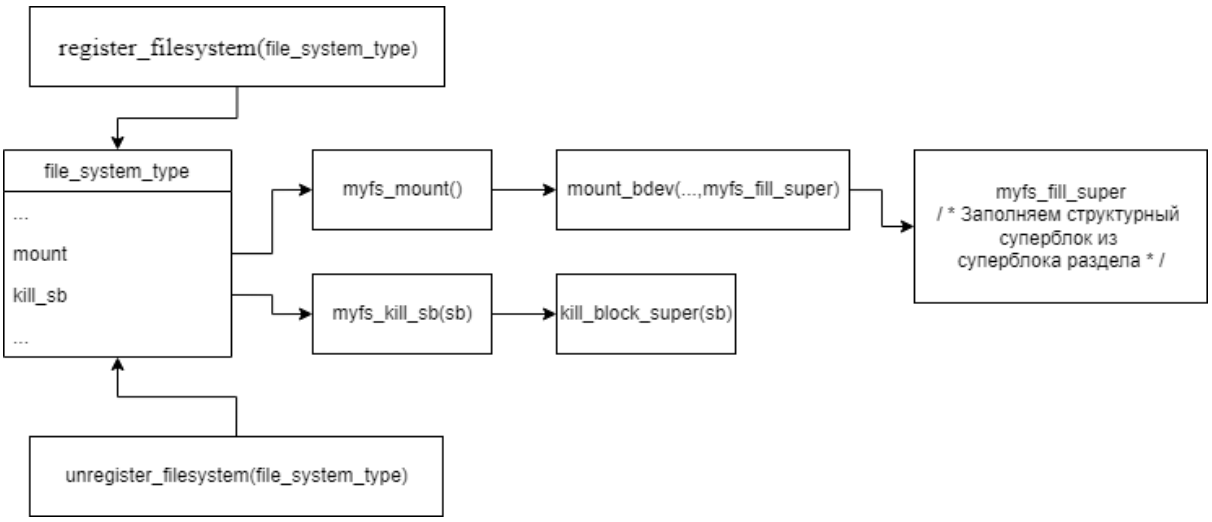


Рисунок 1. Инициализация и установка суперблока

Рисунок 2. алгоритм работы функции *myfs_fill_super*

2.2. **Методы, связанные с объектом file**

На рисунке 3 представлена IDEF0-диаграмма, описывающая функции *open(), read(), write()* структуры *file_operations*.

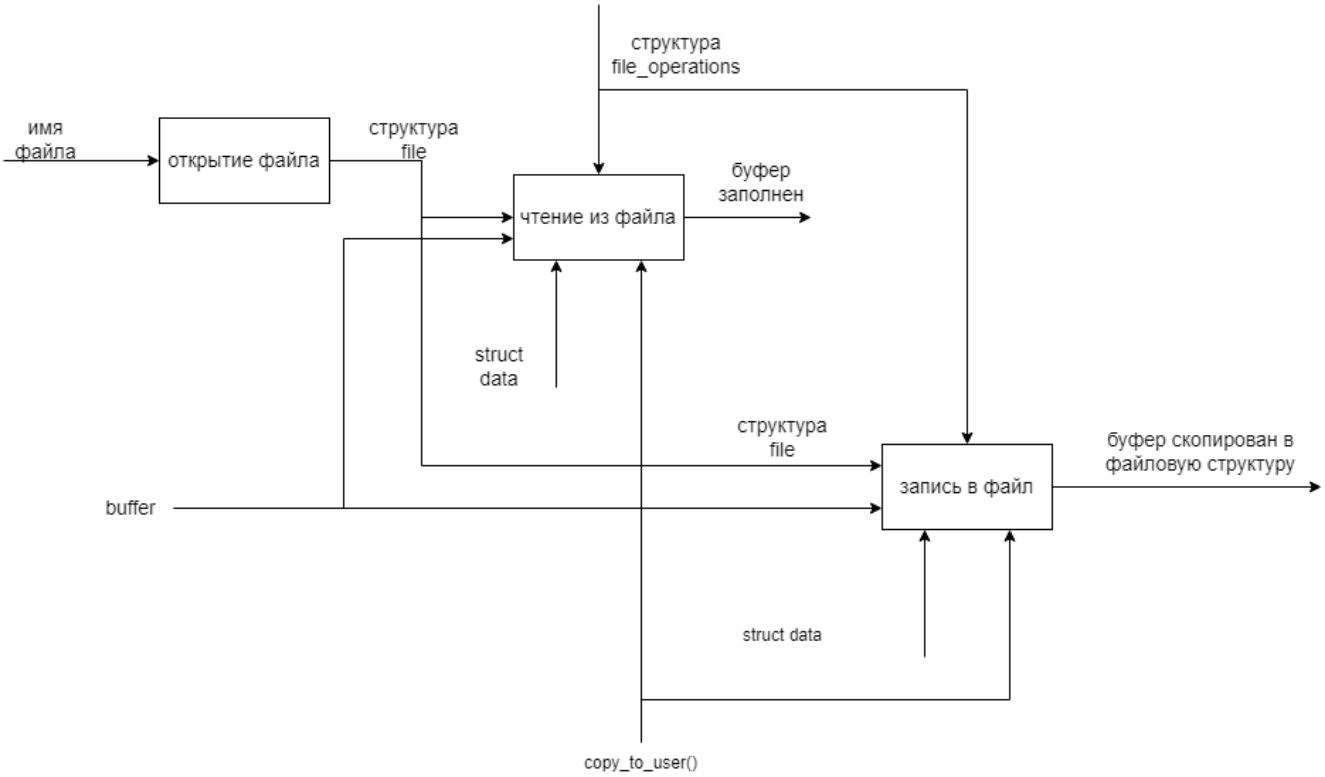


Рисунок 3. описывающая функции open(), read(), write()

2.3.**Методы, связанные с объектом inode**

Описание структуры *inode_operations*, используемой в программе:

Листинг 9. Описание структуры *inode_operations*

```
static const struct inode_operations myfs_inode_ops = {
    .lookup = myfs_lookup,
    .create = myfs_create,
    .unlink = myfs_unlink,
    .mkdir = myfs_mkdir,
    .rmdir = myfs_rmdir,
    .rename = myfs_rename,
    .link = myfs_link,
};
```

В следующем абзаце приведено описание конкретных функций в этой структуре.

2.3.1. Lookup

Искать dentry в справочнике. заполнить dentry значением NULL, если он не находится в каталоге, или соответствующим индексным дескриптором, если он найден. В случае успеха возвращает NULL.

На рисунке показан алгоритм работы операции Lookup().

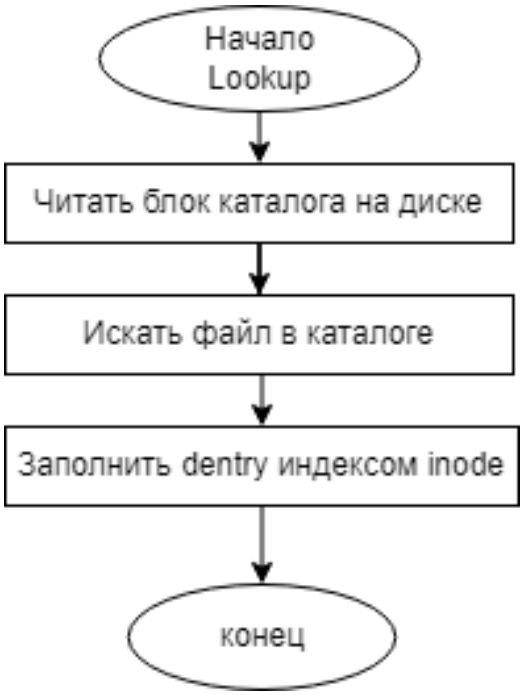


Рисунок 4. алгоритм работы операции *Lookup()*

2.3.2. Create

Create(dir, dentry, mode, excl): Создает новый inode связанного с объектом dentry в некотором каталоге.

На рисунке показан алгоритм работы операции *create()*.

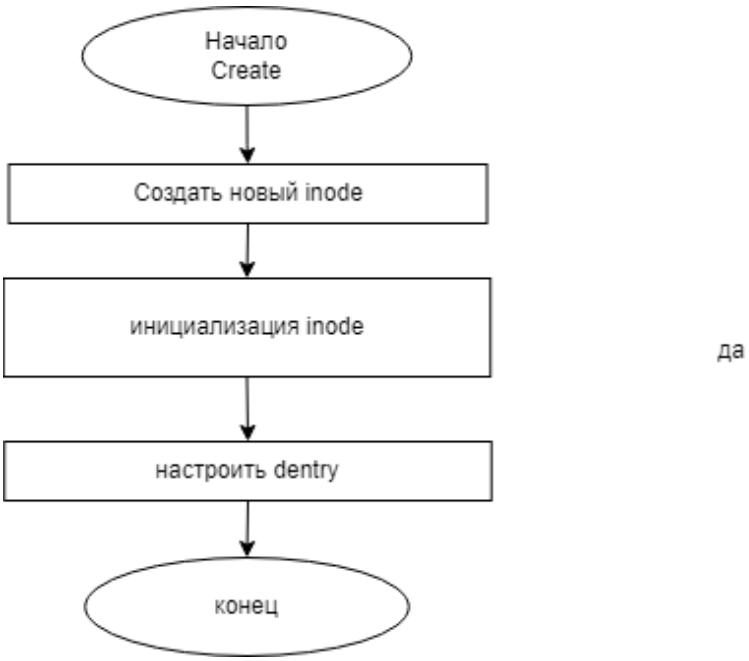


Рисунок 5. алгоритм работы операции *create()*

2.3.3. Mkdir

Mkdir(dir, dentry, mode): Создает новый inode для каталога, связанного с объектом dentry в некотором каталоге.

Создать каталог, вызвав функцию create с установкой флага S_IFDIR

2.3.4. Rmdir

Rmdir (dir, dentry): Удаляет из каталога подкаталог, имя которого включено в объект *dentry.*

На рисунке показан алгоритм работы операции *rmdir().*

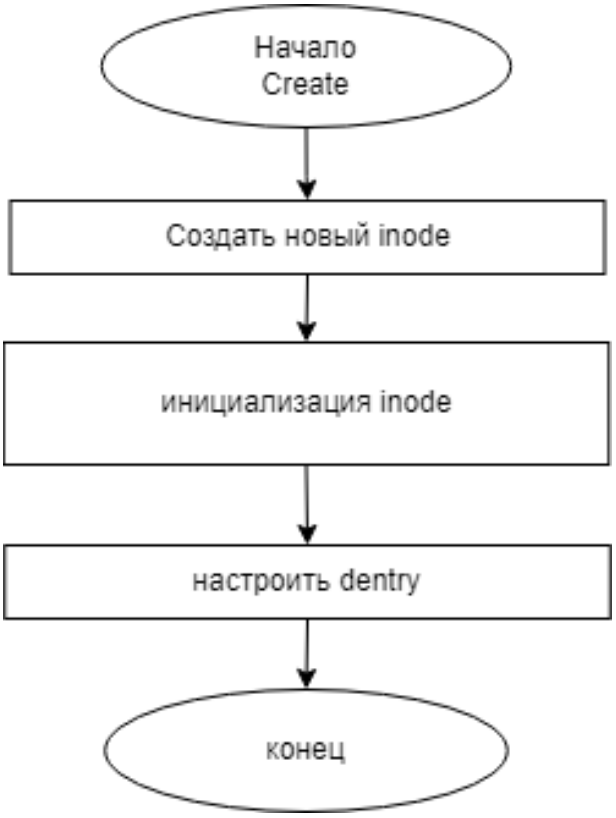


Рисунок 6. алгоритм работы операции rmdir()

2.3.5. Rename

Rename(old_dir, old_dentry, new_dir, new_dentry, flags): Перемещает файл, идентифицированный *old_entry*, из каталога *old_dir* в каталог new_dir.

Новое имя файла включается в объект *dentry*, на который указывает *new_dentry*.

На рисунке показан алгоритм работы операции *rename().*

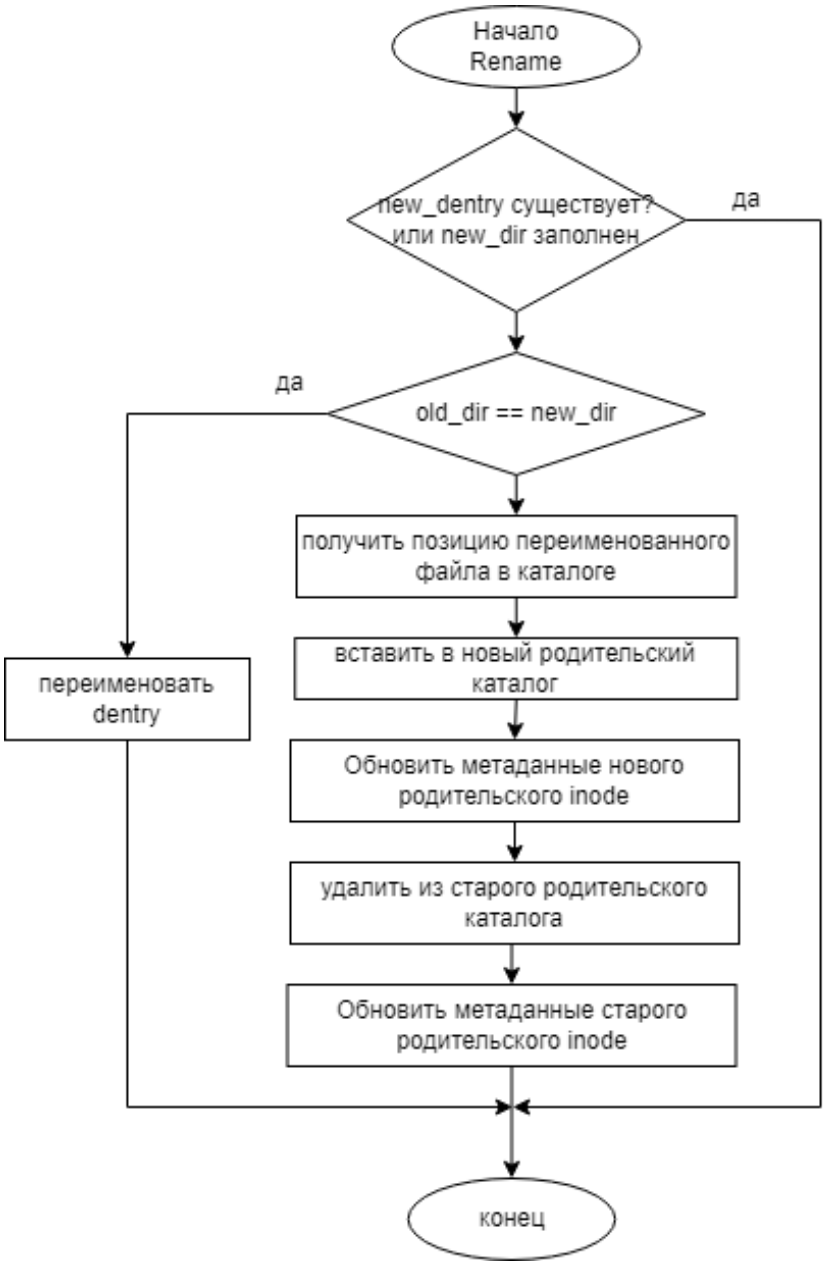


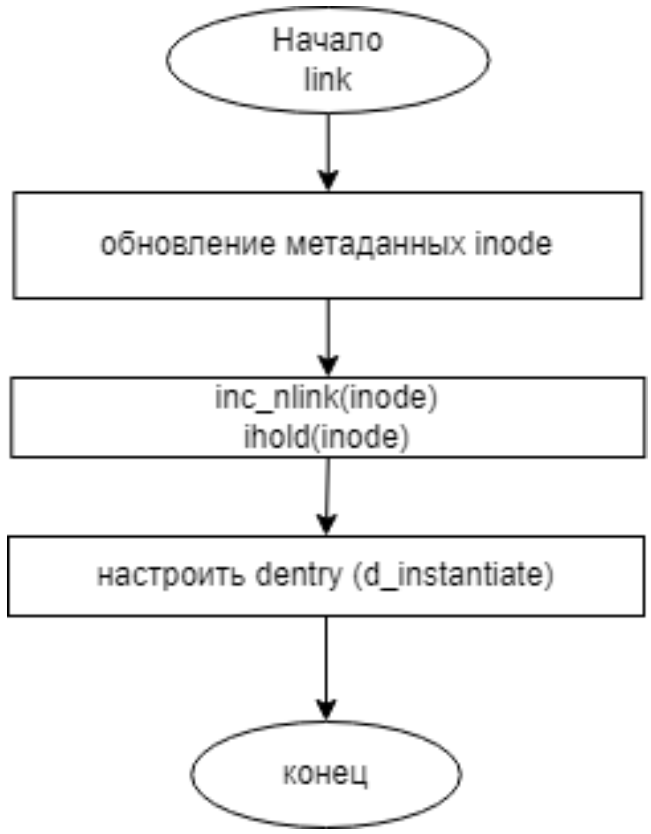
Рисунок 7. алгоритм работы операции rename()

2.3.6. Link

Link(old_dentry, dir, new_dentry): Создает новую жесткую ссылку, которая ссылается на файл, указанный в old_dentry в каталоге dir; новая жесткая ссылка имеет имя, указанное в new_dentry.

На рисунке показан алгоритм работы операции link()

Рисунок 8. алгоритм работы операции link()



2.3.7. Unlink

Unlink(dir, dentry): Удаляет жесткую ссылку на файл, указанный объектом dentry, из каталога.

На рисунке показан алгоритм работы операции unlink()

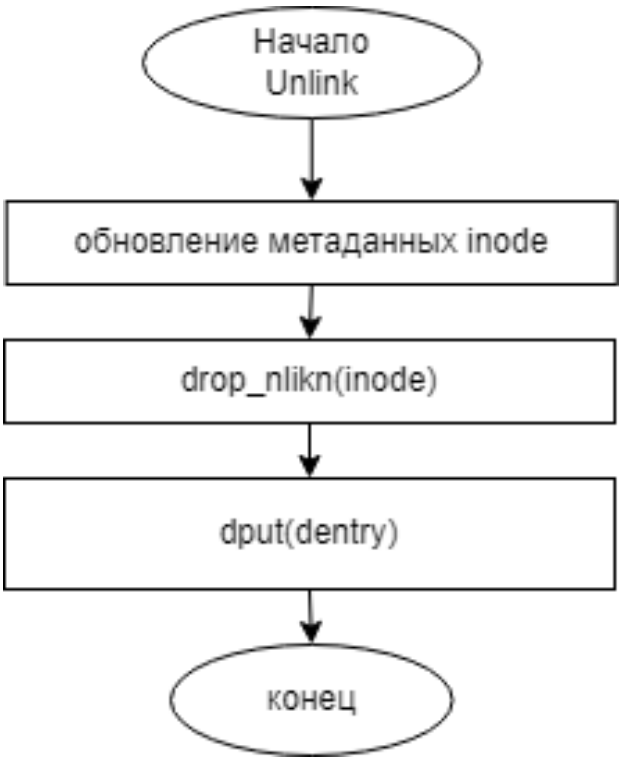


Рисунок 9. алгоритм работы операции unlink()

2.4. **Вывод**

В данном разделе были рассмотрены структуры программного обеспечения, предоставлены алгоритмов функции.

3. Технологическая часть

В данном разделе производится выбор средств для разработки и рассматривается реализация программного обеспечения.

3.1. Выбор языка программирования

В качестве языка программирования был выбран язык C. На этом языке реализованы все модули ядра и драйверы операционной системы Linux.

Компилятор -- gcc.

3.2. Исходный код программы

Листинг 10. Инициализация модуля

```
static int __init myfs_init(void)
{
    int ret = myfs_init_inode_cache();
    if (ret) {
        pr_err("inode cache creation failed\n");
        goto end;
    }

    ret = register_filesystem(&myfs_file_system_type);
    if (ret) {
        pr_err("register_filesystem() failed\n");
        goto end;
    }

    pr_info("module loaded\n");
end:
    return ret;
}
```

Листинг 11. Выход из модуля

```
static void __exit myfs_exit(void)
{
    int ret = unregister_filesystem(&myfs_file_system_type);
    if (ret)
        pr_err("unregister_filesystem() failed\n");

    myfs_destroy_inode_cache();

    pr_info("module unloaded\n");
}
```

Листинг 12. Описание структуры file_system_type

```
struct dentry *myfs_mount(struct file_system_type *fs_type,
                          int flags,
                          const char *dev_name,
                          void *data)
{
    struct dentry *dentry =
        mount_bdev(fs_type, flags, dev_name, data, myfs_fill_super);
    if (IS_ERR(dentry))
        pr_err("'%s' mount failure\n", dev_name);
    else
        pr_info("'%s' mount success\n", dev_name);

    return dentry;
}

/* Unmount a myfs partition */
void myfs_kill_sb(struct super_block *sb)
{
    kill_block_super(sb);

    pr_info("unmounted disk\n");
}

static struct file_system_type myfs_file_system_type = {
    .owner = THIS_MODULE,
    .name = "myfs",
    .mount = myfs_mount,
    .kill_sb = myfs_kill_sb,
    .fs_flags = FS_REQUIRES_DEV,
    .next = NULL,
}
```

Листинг 13. Описание структуры inode_operations

```
static const struct inode_operations myfs_inode_ops = {
    .lookup = myfs_lookup,
    .create = myfs_create,
    .unlink = myfs_unlink,
    .mkdir = myfs_mkdir,
    .rmdir = myfs_rmdir,
    .rename = myfs_rename,
    .link = myfs_link,
    .symlink = myfs_symlink,
};

static struct dentry *myfs_lookup(struct inode *dir, struct dentry *dentry,
unsigned int flags);
```

```
static int myfs_create(struct inode *dir, struct dentry *dentry, umode_t
mode, bool excl);

static int myfs_unlink(struct inode *dir, struct dentry *dentry);

static int myfs_mkdir(struct inode *dir, struct dentry *dentry, umode_t
mode);

static int myfs_rmdir(struct inode *dir, struct dentry *dentry);

static int myfs_link(struct dentry *old_dentry,struct inode *dir, struct
dentry *dentry);

static int myfs_symlink(struct inode *dir, struct dentry *dentry, const char
*symname);
```

Листинг 14. Описание структуры file_operations

```
const struct file_operations myfs_file_operations = {
    .open   = myfs_open,
    .read   = myfs_read_file,
    .write  = myfs_write_file,
}

static int myfs_open(struct inode *inode, struct file *filp)
{
    filp->private_data = inode->i_private;
    return 0;
}

static ssize_t myfs_read_file(struct file *filp, char *buffer,size_t count, lo
*offset)
{
    printk(KERN_INFO "Message: Start Reading");

    loff_t pos = *offset;
    struct file_data *fdata = (struct file_data *) filp->private_data;
    char *data = (char *) fdata->data;
    if (pos < 0)
        return -EINVAL;

    if (pos > MYFS_MAX_SIZE || !count)
        return 0;

    if (count > strlen(data) - pos)
        count = strlen(data) - pos;

    if (copy_to_user(buffer, data, count ))
```

```
    {
        return -EFAULT;
    }

    (*offset) += count;
    return count;
}

static ssize_t myfs_write_file(struct file *filp, const char *buffer, size_t co
loff_t *offset)
{
    loff_t pos = *offset;
    struct file_data *fdata = (struct file_data *) filp->private_data;
    char * data = fdata->data;
    char tmp[1024];
    if (pos < 0)
        return -EINVAL;

    size_t buffer_len = MYFS_MAX_SIZE;//strlen(buffer);

    if (pos > buffer_len || !count)
        return 0;


    if ( count > buffer_len - pos ) {
        count =  buffer_len - pos;
    }

    if (copy_from_user(tmp, buffer, count) )
    {
        return -EFAULT;
    }

    memcpy(data + (size_t)(*offset), tmp, count);

    (*offset) = pos + count;

    return count;
}
```

4. Исследовательская часть

4.1. Условия эксперимента

Исследование результатов выполнения программы производилось при следующем аппаратном обеспечении, выделенном виртуальной машине:

- процессор Intel(R) Core(TM) i5-7200U CPU @ 2.50ghz 2.70 ghz
- объем оперативной памяти: 4GB
- операционная система: OS Ubuntu 20.4
- версия ядра Linux Linux Kernel 5.13

4.2. Загрузка модуля и интерфейс программы

Sudo insmod myfs.ko

Mkdir -p myfs-Root

$ dd if=/dev/zero of=myfs.img bs=1M count=50

Sudo mount -o loop -t myfs myfs.img myfs-Root

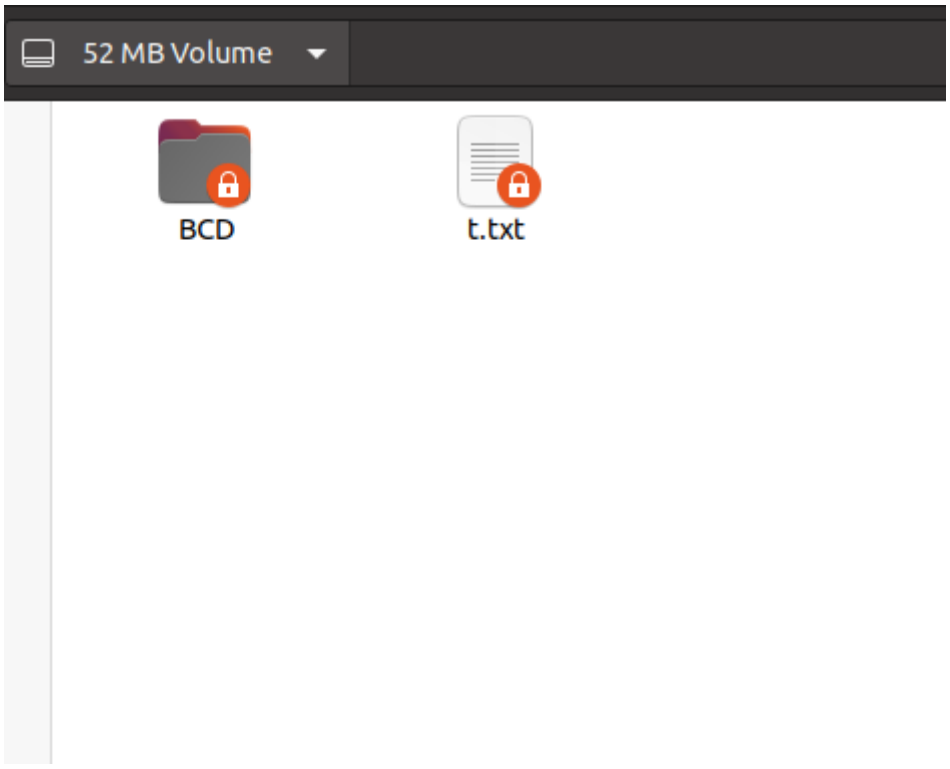


Рисунок 10. интерфейс программы

4.3. Результат работы программы

На рисунках 12-15 показаны результаты работы функций: ***create, rename, rmdir, read, write***

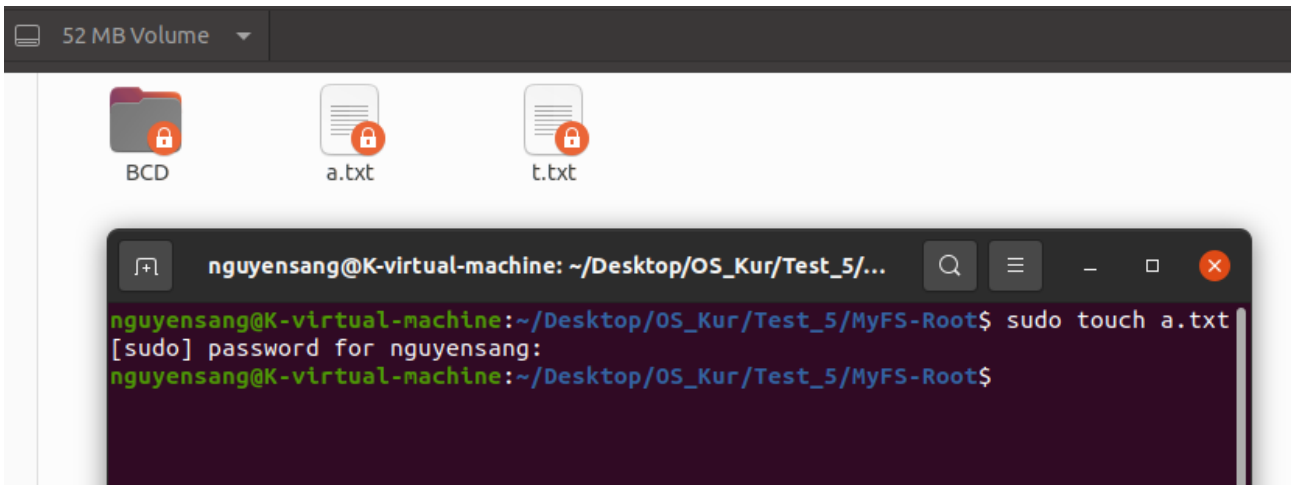


Рисунок 11. Работа операции ***create()***

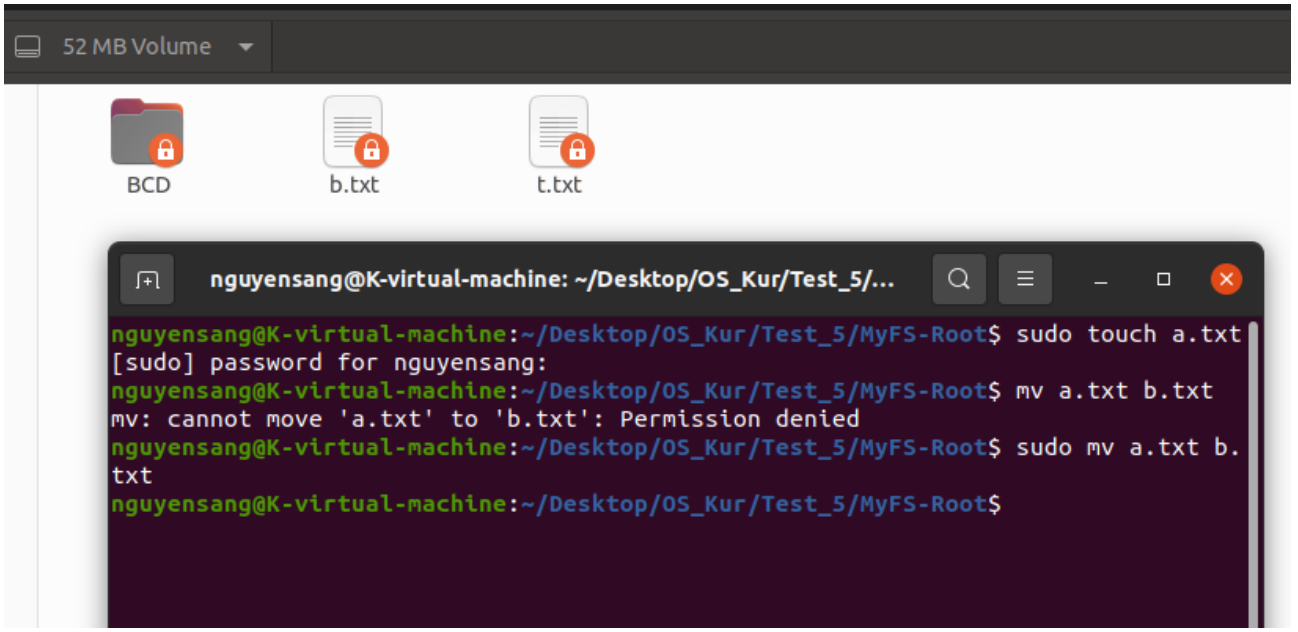


Рисунок 12. Работа операции ***rename()***

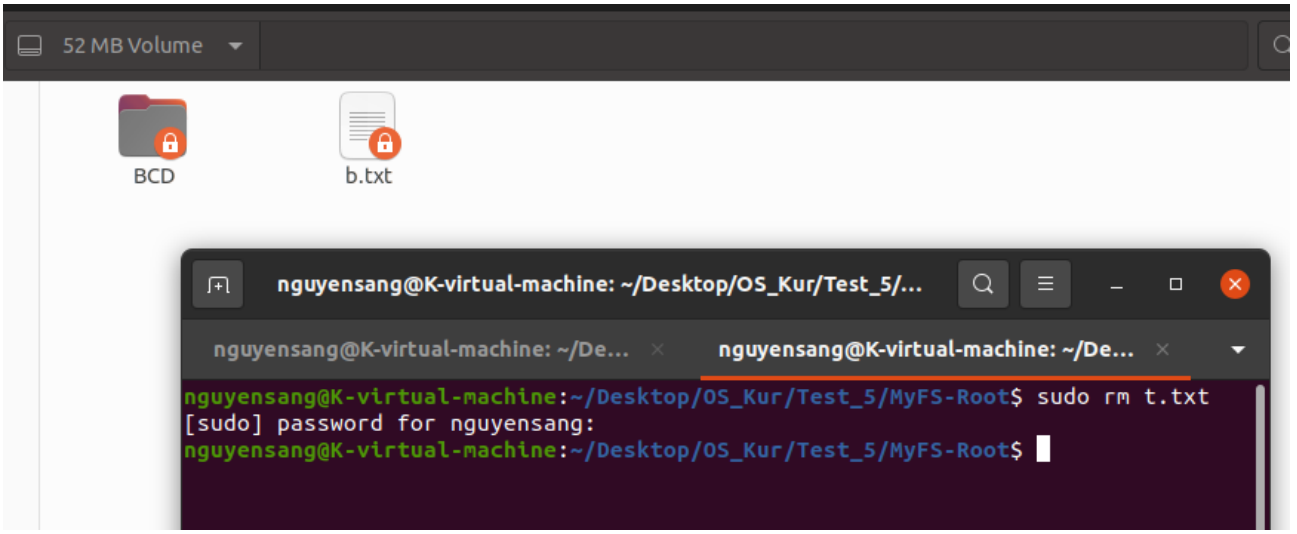


Рисунок 13. Работы операции ***rmdir()***

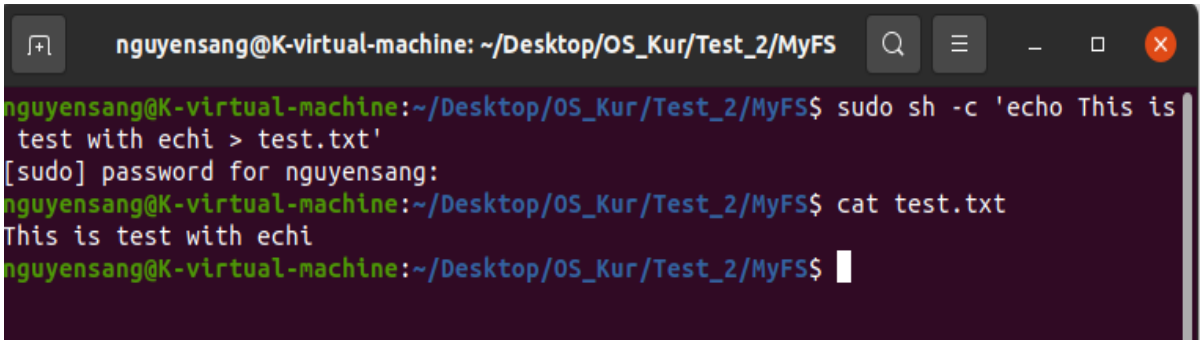


Рисунок 14. Работы операции ***read , write***

4.4. Выгрузка модуля

Unount –t myfs MyFS-Root

sudo rmmod myfs

ЗАКЛЮЧЕНИЕ

В результате выполнения данного курсового проекта был изучен метод создания виртуальной файловой системы, работа с ядром и ядерными функциями. Разработана ПО , в соответствии с техническим заданием. Разработанный программный продукт удовлетворяет поставленной задаче.

В частности:

- Проанализированы особенности файловой подсистемы Linux и интерфейса VFS
- Проанализирована структуру файловой системы и структуры описывающие её элементы: *superblock, dentry, inode, file*
- Разработаны алгоритмы и структуры ПО
- Разработано ПО виртуальной файловой системы

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Overview of the Linux Virtual File System
   [https://www.kernel.org/doc/html/latest/filesystems/vfs.html]
2. конспект лекций по курсу "Операционные системы"
3. The Linux Kernel's VFS Layer
   [https://www.usenix.org/legacy/publications/library/proceedings/usenix01/full_papers/kroeger/kroeger_html/node8.html]
4. Using the page cache
   [https://cs4118.github.io/pantryfs/page-cache-overview.pdf]
5. Linux File System: Virtual File System (VFS)
   [https://emmanuelbashorun.medium.com/linux-file-system-virtual-file-system-vfs-layer-part-3-79235c40a499]
6. Виртуальная файловая система
7. *[http://www.cs.vsu.ru/~svv/ux/lecture%205.pdf]*
8. How to write a Linux VFS filesystem module
   [http://pages.cpsc.ucalgary.ca/~crwth/programming/VFS/VFS.php]
9. Understanding the Linux Kernel, Daniel P. Bovet, Marco Cesati
   [https://www.oreilly.com/library/view/understanding-the-linux/0596005652/ch12s02.html#:~:text=Each%20VFS%20object%20is%20stored,specialized%20behavior%20for%20the%20object.]
10. Linux VFS
    [https://titanwolf.org/]

Приложение

Листинг 15. myfs.h

```
#ifndef __MYFS_H__
#define __MYFS_H__

#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/pagemap.h>  /* PAGE_CACHE_SIZE */
#include <linux/fs.h>       /* This is where libfs stuff is declared */
#include <asm/atomic.h>
#include <asm/uaccess.h>    /* copy_to_user */
#include <linux/time.h>

#include <linux/buffer_head.h>
#include <linux/slab.h>
#include <linux/statfs.h>

#define TMPSIZE 20
#define MYFS_MAX_SIZE 1024


/*
 *  INODE
 */

int myfs_rmdir(struct inode *dir, struct dentry *dentry);

int myfs_rename(struct inode *old_dir, struct dentry *old_dentry, struct
inode *new_dir, struct dentry *new_dentry, unsigned int dev);

int myfs_unlink(struct inode *dir, struct dentry *dentry);

int myfs_link(struct dentry *old_dentry, struct inode *dir, struct dentry
*dentry);

struct dentry *myfs_lookup(struct inode *dir, struct dentry *dentry,
unsigned int flags);

static struct inode *myfs_make_inode(struct super_block *sb, int mode);

static int myfs_create(struct inode *dir, struct dentry *dentry,umode_t mode,
bool excl);

const char *myfs_get_link(struct dentry *dentry, struct inode *inode, struct
delayed_call *done);

/*
```

```
 *  FILE
 */

static int myfs_open(struct inode *inode, struct file *filp);

static ssize_t myfs_read_file(struct file *filp, char *buffer,size_t count,
loff_t *offset);

static ssize_t myfs_write_file(struct file *filp, const char *buffer, size_t
count, loff_t *offset);

int myfs_release(struct inode *inode, struct file *file);

static void lfs_create_files (struct super_block *sb, struct dentry *root);

static struct dentry *lfs_create_file (struct super_block *sb, struct dentry
*dir, const char *name);

static struct dentry *lfs_create_dir (struct super_block *sb, struct dentry
*parent, const char *name);

 struct file_data {
    char * data;
    size_t size;
 };

 struct file_data * create_empty_data(void);


/*
 *  END FILE
 */


static int myfs_fill_super (struct super_block *sb, void *data, int silent);

static struct dentry *myfs_get_super(struct file_system_type *fst, int flags,
const char *devname, void *data);


/*
 *  STRUCT
 */



const struct file_operations myfs_file_operations = {
    .open    = myfs_open,
    .read    = myfs_read_file,
    .write   = myfs_write_file,
```

```
.     //.release= myfs_release,
. };
.
. const struct file_operations myfs_dir_operations = {
.     .open       = dcache_dir_open,
.     .release    =  dcache_dir_close,
.     .llseek     = dcache_dir_lseek,
.     .read       = generic_read_dir,
.     .iterate    = dcache_readdir,
.     .fsync      = noop_fsync,
.
. };
.
0.
1.
2.      const struct super_operations myfs_s_ops = {
3.          .statfs     = simple_statfs,
4.          .drop_inode = generic_delete_inode,
5.      };
6.
7.      struct file_system_type myfs_type = {
8.          .owner      = THIS_MODULE,
9.          .name       = "myfs",
0.          .mount      = myfs_get_super,
1.          .kill_sb    = kill_litter_super,
2.      };
3.
4.      const struct inode_operations myfs_file_inode_operations= {
5.          .rename = myfs_rename,
6.          .rmdir = myfs_rmdir,
7.          .unlink = myfs_unlink,
8.          .link = myfs_link,
9.          .lookup = myfs_lookup,
0.          .create = myfs_create,
1.      };
2.
3.      const struct inode_operations myfs_dir_inode_operations = {
4.          .rename = myfs_rename,
5.          .rmdir = myfs_rmdir,
6.          .unlink = myfs_unlink,
7.          .link = myfs_link,
8.          .lookup = myfs_lookup,
9.          .create = myfs_create,
0.      };
1.
2.
3.      const struct inode_operations myfs_symlink_inode_operations = {
4.          .get_link = myfs_get_link,
5.      };
```

```
6.
7.     /*
8.      *  END STRUCT
9.      */
. #endif /* __MYFS_H__ */
```

Листинг 16. myfs.c

```
#include <linux/kernel.h>
#include <linux/init.h>
#include <linux/module.h>
#include <linux/pagemap.h>  /* PAGE_CACHE_SIZE */
#include <linux/fs.h>       /* This is where libfs stuff is declared */
#include <asm/atomic.h>
#include <asm/uaccess.h>    /* copy_to_user */

#include <linux/time.h>
.#include "myfs.h"
.
.#define LFS_MAGIC 0x19920342
.
.
.static int myfs_fill_super (struct super_block *sb, void *data, int silent)
.{
.     struct inode *root;
.     struct dentry *root_dentry;
.
.     sb->s_blocksize = VMACACHE_SIZE;
.     sb->s_blocksize_bits = VMACACHE_SIZE;
.     sb->s_magic = LFS_MAGIC;
.     sb->s_op = &myfs_s_ops;
.
.     root = myfs_make_inode (sb, S_IFDIR | 0755);
.     inode_init_owner(root, NULL, S_IFDIR | 0755);
.     if (! root)
.         goto out;
.     root->i_op = &myfs_dir_inode_operations;
.     root->i_fop = &myfs_dir_operations;
.
.     set_nlink(root, 2);
.     root_dentry = d_make_root(root);
.     if (! root_dentry)
.         goto out_iput;
.
.     lfs_create_files (sb, root_dentry);
.     sb->s_root = root_dentry;
.     return 0;
.
```

```
  out_iput:
    iput(root);
  out:
    return -ENOMEM;
}

static struct dentry *myfs_get_super(struct file_system_type *fst, int flags,
const char *devname, void *data)
{
    return mount_nodev(fst, flags, data, myfs_fill_super);
}


static int __init myfs_init(void)
{
    return register_filesystem(&myfs_type);
}

static void __exit myfs_exit(void)
{
    unregister_filesystem(&myfs_type);
}

module_init(myfs_init);
module_exit(myfs_exit);

MODULE_LICENSE("GPL");


/*
 *       INODE
 */

const char *myfs_get_link(struct dentry *dentry, struct inode *inode, struct
delayed_call *done)
{
    return inode->i_link;
}

int myfs_rmdir(struct inode *dir, struct dentry *dentry)
{
    if (!simple_empty(dentry))
        return -ENOTEMPTY;

    drop_nlink(d_inode(dentry));
    simple_unlink(dir, dentry);
    drop_nlink(dir);
    return 0;
}
```

```
.
. int myfs_rename(struct inode *old_dir, struct dentry *old_dentry, struct
 inode *new_dir, struct dentry *new_dentry, unsigned int dev)
. {
.         printk(KERN_INFO "Message: Start Rename\n");
.
.     struct inode *inode = d_inode(old_dentry);
.     int they_are_dirs = d_is_dir(old_dentry);
.
.     if (!simple_empty(new_dentry))
.         return -ENOTEMPTY;
.
.     if (d_really_is_positive(new_dentry)) {
0.             simple_unlink(new_dir, new_dentry);
1.             if (they_are_dirs) {
2.                 drop_nlink(d_inode(new_dentry));
3.                 drop_nlink(old_dir);
4.             }
5.         } else if (they_are_dirs) {
6.             drop_nlink(old_dir);
7.             inc_nlink(new_dir);
8.         }
9.
0.         old_dir->i_ctime = old_dir->i_mtime = new_dir->i_ctime =
1.             new_dir->i_mtime = inode->i_ctime = current_time(inode);
2.
3.         return 0;
4.     }
5.
6.     int myfs_unlink(struct inode *dir, struct dentry *dentry)
7.     {
8.         struct inode *inode = d_inode(dentry);
9.
0.         inode->i_ctime = dir->i_ctime = dir->i_mtime =
 current_time(inode);
1.         drop_nlink(inode);
2.         dput(dentry);
3.         return 0;
4.     }
5.
6.     int myfs_link(struct dentry *old_dentry, struct inode *dir, struct
 dentry *dentry)
7.     {
8.         struct inode *inode = d_inode(old_dentry);
9.
0.         inode->i_ctime = dir->i_ctime = dir->i_mtime =
 current_time(inode);
1.         inc_nlink(inode);
2.         ihold(inode);
3.         dget(dentry);
```

```
4.          d_instantiate(dentry, inode);
5.          return 0;
6.      }
7.
8.      struct dentry *myfs_lookup(struct inode *dir, struct dentry *dentry,
 unsigned int flags)
9.      {
0.          printk(KERN_INFO "Message: Start Lookup\n");
1.
2.          if (dentry->d_name.len > NAME_MAX)
3.              return ERR_PTR(-ENAMETOOLONG);
4.          if (!dentry->d_sb->s_d_op)
5.              d_set_d_op(dentry, &simple_dentry_operations);
6.          d_add(dentry, NULL);
7.          return NULL;
8.      }
9.
0.      static struct inode *myfs_make_inode(struct super_block *sb, int mode)
1.      {
2.          struct inode* inode;
3.              inode = new_inode(sb);
4.             if (!inode) {
5.                      return NULL;
6.              }
7.              inode->i_mode = mode;
8.              inode->i_atime = inode->i_mtime = inode->i_ctime =
 current_time(inode);
9.              if (S_ISREG(mode))
0.              {
1.                  inode->i_fop = &myfs_file_operations;
2.                  inode->i_op = &myfs_file_inode_operations;
3.                  set_nlink(inode, 1);
4.                  struct file_data * data = (struct file_data *)
 create_empty_data();
5.                  inode->i_private = data;
6.              }
7.              else if (S_ISDIR(mode))
8.              {
9.                  inode->i_fop = &myfs_dir_operations;
0.                  inode->i_op = &myfs_dir_inode_operations;
1.                  set_nlink(inode, 2);
2.              }
3.              inode->i_ino = get_next_ino();
4.          return inode;
5.
6.      }
7.
8.      static int myfs_create(struct inode *dir, struct dentry *dentry,umode_t
 mode, bool excl)
```

```
9.      {
0.          printk(KERN_INFO "Message: Start Create\n");
1.          struct inode *inode;
2.
3.          inode = myfs_make_inode(dir->i_sb, mode | S_IFREG);
4.          if (!inode)
5.              goto out;
6.          inode_init_owner(inode, dir, mode | S_IFREG);
7.          //d_instantiate(dentry, inode);
8.          d_add(dentry, inode);
9.          dget(dentry);
0.
1.      out:
2.          return 0;
3.      }
4.
5.
6.      static void lfs_create_files (struct super_block *sb, struct dentry
 *root)
7.      {
8.          struct dentry *subdir;
9.
0.          lfs_create_file(sb, root, "test.txt");
1.
2.          subdir = lfs_create_dir(sb, root, "Dir1");
3.          if (subdir)
4.              lfs_create_file(sb, subdir, "test.txt");
5.      }
6.
7.
8.      static struct dentry *lfs_create_dir (struct super_block *sb,
9.              struct dentry *parent, const char *name)
0.      {
1.          struct dentry *dentry;
2.          struct inode *inode;
3.
4.          dentry = d_alloc_name(parent, name);
5.          if (! dentry)
6.              goto out;
7.
8.          inode = myfs_make_inode(sb, S_IFDIR | 0755);
9.          if (! inode)
0.              goto out_dput;
1.          inode->i_op = &myfs_dir_inode_operations;
2.
3.          d_add(dentry, inode);
4.          return dentry;
5.
6.        out_dput:
```

```
7.          dput(dentry);
8.        out:
9.          return 0;
0.      }
1.
2.
3.      static struct dentry *lfs_create_file (struct super_block *sb, struct
 dentry *dir, const char *name)
4.      {
5.          struct dentry *dentry;
6.          struct inode *inode;
7.
8.          dentry = d_alloc_name(dir, name);
9.          if (! dentry)
0.              goto out;
1.          inode = myfs_make_inode(sb, S_IFREG | 0644);
2.          if (! inode)
3.              goto out_dput;
4.          strcpy(((struct file_data *) inode->i_private)->data, name);
5.
6.          d_add(dentry, inode);
7.          return dentry;
8.
9.        out_dput:
0.          dput(dentry);
1.        out:
2.          return 0;
3.      }
4.
5.
6.      static int myfs_open(struct inode *inode, struct file *filp)
7.      {
8.          filp->private_data = inode->i_private;
9.          return 0;
0.      }
1.
2.      static ssize_t myfs_read_file(struct file *filp, char *buffer,size_t
 count, loff_t *offset)
3.      {
4.          printk(KERN_INFO "Message: Start Reading");
5.
6.          loff_t pos = *offset;
7.          struct file_data *fdata = (struct file_data *) filp->private_data;
8.          char *data = (char *) fdata->data;
9.          if (pos < 0)
0.              return -EINVAL;
1.
2.          if (pos > MYFS_MAX_SIZE || !count)
3.              return 0;
```

```
4.
5.          if (count > strlen(data) - pos)
6.              count = strlen(data) - pos;
7.
8.          if (copy_to_user(buffer, data, count ))
9.          {
0.              return -EFAULT;
1.          }
2.
3.          (*offset) += count;
4.          return count;
5.      }
6.
7.      static ssize_t myfs_write_file(struct file *filp, const char *buffer,
 size_t count, loff_t *offset)
8.      {
9.          printk(KERN_INFO "Message: Start Writing offset = %lld",
 *offset);
0.          loff_t pos = *offset;
1.          struct file_data *fdata = (struct file_data *) filp->private_data;
2.          char * data = fdata->data;
3.          char tmp[1024];
4.          printk(KERN_INFO "Message: Write Check 1");
5.
6.          if (pos < 0)
7.              return -EINVAL;
8.
9.          printk(KERN_INFO "Message: Write Check 2");
0.
1.          size_t buffer_len = MYFS_MAX_SIZE;//strlen(buffer);
2.
3.          printk(KERN_INFO "Message: Write Check 3");
4.
5.          if (pos > buffer_len || !count)
6.              return 0;
7.
8.
9.          if ( count > buffer_len - pos ) {
0.              count =  buffer_len - pos;
1.          }
2.
3.          printk(KERN_INFO "Message: Write Check 4");
4.
5.          if (copy_from_user(tmp, buffer, count) )
6.          {
7.              return -EFAULT;
8.          }
9.
0.          printk(KERN_INFO "Message: Write Check 5 ");
```

```
1.
2.          memcpy(data + (size_t)(*offset), tmp, count);
3.
4.          printk(KERN_INFO "Message: Write Check 6");
5.          (*offset) = pos + count;
6.
7.          return count;
8.      }
9.
0.      int myfs_release(struct inode *inode, struct file *file)
1.      {
2.          kfree(file->private_data);
3.          return 0;
4.      }
5.
6.
7.       struct file_data * create_empty_data(void)
8.      {
9.          struct file_data * new = (struct file_data *) kzalloc(sizeof(struct
 file_data), GFP_KERNEL);
0.          new->data = (char *)kzalloc(MYFS_MAX_SIZE, GFP_KERNEL);
1.          memset(new->data, 0, MYFS_MAX_SIZE);
2.          new->size = 0;
3.          return new;
4.      }
5.      /*
6.       *       END FILE
7.      */
```

Листинг 17. Makefile

```
obj-m += fs.o
fs-objs := myfs.o

CURRENT = $(shell uname -r)
KDIR = /lib/modules/$(CURRENT)/build
PWD = $(shell pwd)

all:
    $(MAKE) -C $(KDIR) M=$(PWD) modules
clean:
    @rm -f *.o .*.cmd .*.flags *.mod.c *.order
    @rm -f .*.*.cmd *~ *.*~ TODO.*
    @rm -fR .tmp*
    @rm -rf .tmp_versions
disclean: clean
    @rm *.ko *.symvers
```