|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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

|  |  |
| --- | --- |
| Gerb-BMSTU_01 | **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.

**ЗАДАНИЕ**

**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 4](#_Toc93724238)

[1. Аналитическая часть 5](#_Toc93724239)

[1.1. Постановка задачи 5](#_Toc93724240)

[1.2. Загружаемые модули 5](#_Toc93724241)

[1.3. Распределение памяти: 6](#_Toc93724242)

[1.4. Объекты суперблока 6](#_Toc93724243)

[1.5. Объекты Inode 7](#_Toc93724244)

[1.6. Структура inode\_operations 8](#_Toc93724245)

[1.7. Объекты файлов 9](#_Toc93724246)

[1.8. Объект dentry 10](#_Toc93724247)

[1.9. Управление адресным пространством 10](#_Toc93724248)

[2. Конструкторская часть 12](#_Toc93724249)

[2.1. Взаимодействие между процессами и объектами VFS 12](#_Toc93724250)

[2.2. Некоторая дополнительная структура 12](#_Toc93724251)

[2.3. Инициализация и установка суперблока 14](#_Toc93724252)

[2.4. Методы, связанные с объектом inode 15](#_Toc93724253)

[2.4.1. Lookup 15](#_Toc93724254)

[2.4.2. Create 15](#_Toc93724255)

[2.4.3. Mkdir 16](#_Toc93724256)

[2.4.4. Rmdir 16](#_Toc93724257)

[2.4.5. Rename 17](#_Toc93724258)

[2.4.6. Link 18](#_Toc93724259)

[2.4.7. Unlink 18](#_Toc93724260)

[2.4.8. Symlink 20](#_Toc93724261)

[3. Технологическая часть 21](#_Toc93724262)

[3.1. Выбор языка программирования 21](#_Toc93724263)

[3.2. Исходный код программы 21](#_Toc93724264)

[4. Экспериментальная часть 24](#_Toc93724265)

[4.1. Условия эксперимента 24](#_Toc93724266)

[4.2. Загрузка модуля и интерфейс программы 24](#_Toc93724267)

[4.3. Результат работы программы 24](#_Toc93724268)

[4.4. Выгрузка модуля 26](#_Toc93724269)

[ЗАКЛЮЧЕНИЕ 27](#_Toc93724270)

[СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ 28](#_Toc93724271)

[Приложение 29](#_Toc93724272)

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. **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. **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. **Файловые подсистемы Linux:**

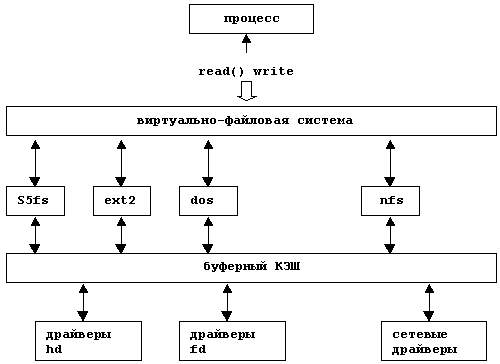
В Linux все файлы и каталоги размещаются в древовидной структуре. Самый верхний каталог файловой системы именуется как корневой (root) или просто “/” . Все прочие каталоги в Linux доступны из корневого и организованы в виде иерархической структуры.

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. **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. **Структуры, описывающие элементы файловой системы**
     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. **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. **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. **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 \*, size\_t, loff\_t \*); 3. ssize\_t (\*write) (struct file \*, const char \_\_user \*, size\_t, loff\_t \*); 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. **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.

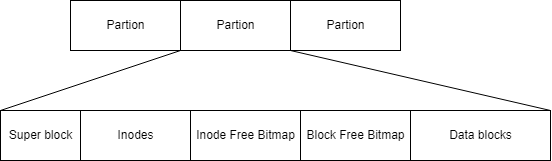
Листинг 8. Описание структуры dentry

|  |
| --- |
| 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. **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 будyт использоваться для управления свободным пространством:

• inode free bitmap - для свободных inodes

• block free bitmap -  для свободных блоков данных

* 1. **Вывод:**

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

1. Конструкторская часть
   1. **Инициализация и установка суперблока**

Struct ***file\_system\_typ*e** - основная структура данных, описывающая файловую систему в ядре. **Mount** и **kill\_sb** - 2 функции управления суперблоком.

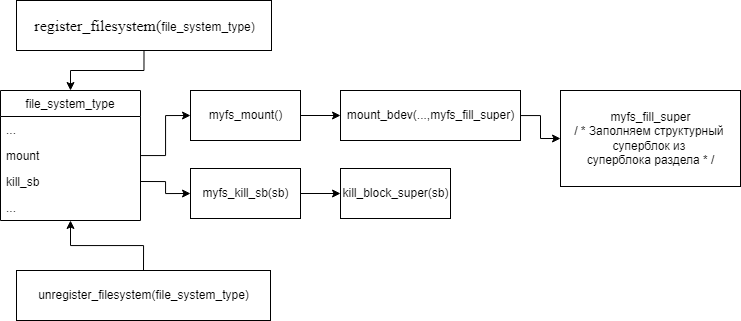


Рисунок 1. Инициализация и установка суперблока

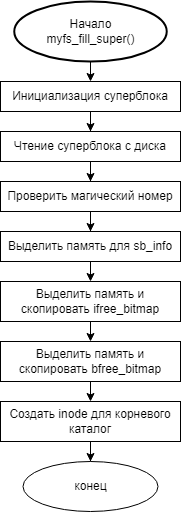


Рисунок 2. алгоритм работы функции myfs\_fill\_super

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

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

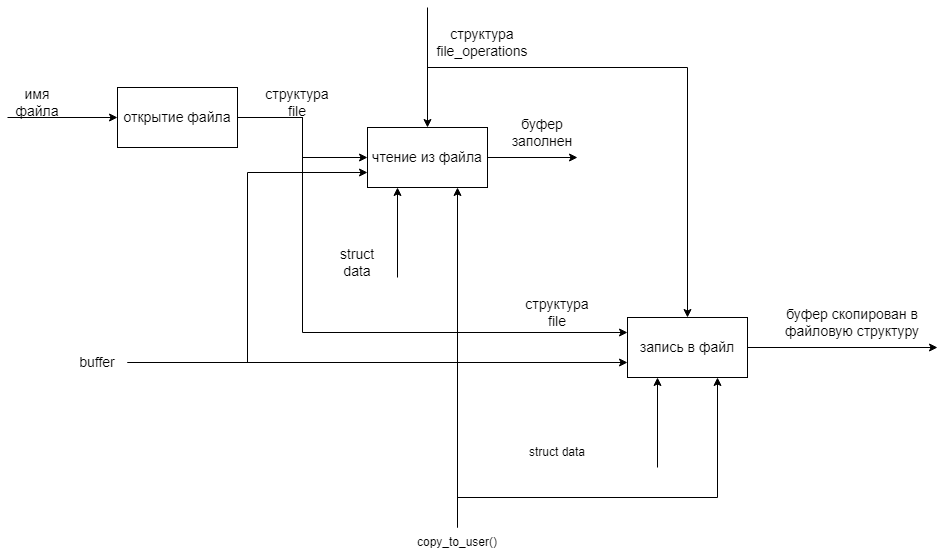


Рисунок 3. описывающая функции open(), read(), write()

* 1. **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.

* + 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.

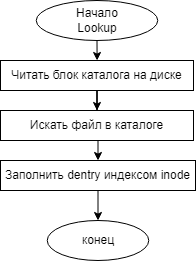
 На рисунке показан алгоритм работы операции Lookup().

Рисунок 4. алгоритм работы операции Lookup()

* + 1. **Create**

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

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

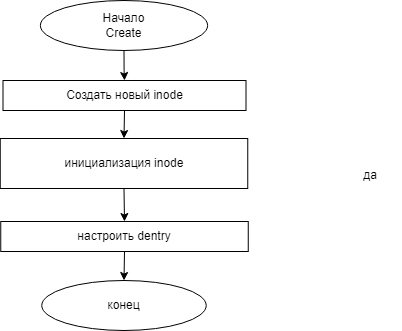
****

Рисунок 5. алгоритм работы операции create()

* + 1. **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**

* + 1. **Rmdir**

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

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

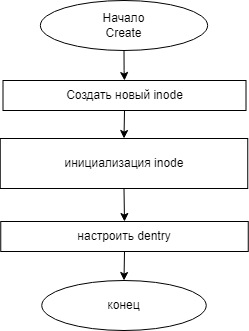
****

Рисунок 6. алгоритм работы операции rmdir()

* + 1. **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().*

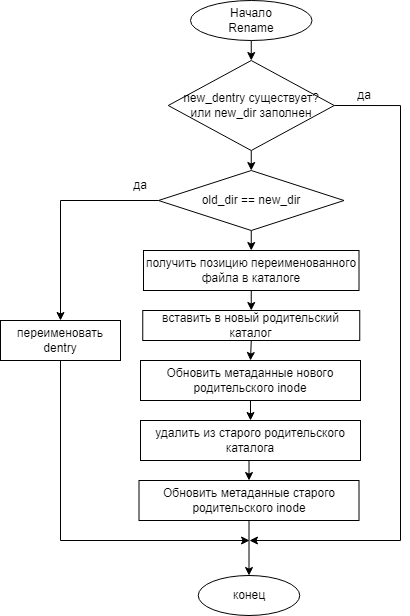
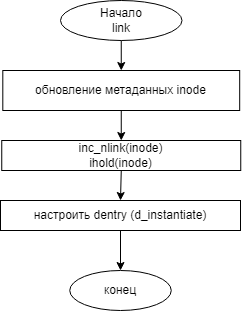
****

Рисунок 7. алгоритм работы операции rename()

* + 1. **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()

* + 1. **Unlink**

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

На рисунке показан алгоритм работы операции unlink()

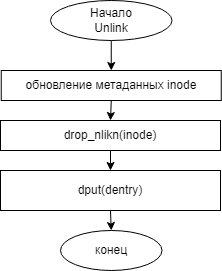
****

Рисунок 9. алгоритм работы операции unlink()

* 1. **Вывод**

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

1. Технологическая часть

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

* 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.

* 1. **Исходный код программы**

Листинг 10. Инициализация модуля

|  |
| --- |
| 1. static int \_\_init myfs\_init(void) 2. { 3. int ret = myfs\_init\_inode\_cache(); 4. if (ret) { 5. pr\_err("inode cache creation failed\n"); 6. goto end; 7. } 9. ret = register\_filesystem(&myfs\_file\_system\_type); 10. if (ret) { 11. pr\_err("register\_filesystem() failed\n"); 12. goto end; 13. } 15. pr\_info("module loaded\n"); 16. end: 17. return ret; 18. } |

Листинг 11. Выход из модуля

|  |
| --- |
| 1. static void \_\_exit myfs\_exit(void) 2. { 3. int ret = unregister\_filesystem(&myfs\_file\_system\_type); 4. if (ret) 5. pr\_err("unregister\_filesystem() failed\n"); 7. myfs\_destroy\_inode\_cache(); 9. pr\_info("module unloaded\n"); 10. } |

Листинг 12. Описание структуры file\_system\_type

|  |
| --- |
| 1. struct dentry \*myfs\_mount(struct file\_system\_type \*fs\_type, 2. int flags, 3. const char \*dev\_name, 4. void \*data) 5. { 6. struct dentry \*dentry = 7. mount\_bdev(fs\_type, flags, dev\_name, data, myfs\_fill\_super); 8. if (IS\_ERR(dentry)) 9. pr\_err("'%s' mount failure\n", dev\_name); 10. else 11. pr\_info("'%s' mount success\n", dev\_name); 13. return dentry; 14. } 16. /\* Unmount a myfs partition \*/ 17. void myfs\_kill\_sb(struct super\_block \*sb) 18. { 19. kill\_block\_super(sb); 21. pr\_info("unmounted disk\n"); 22. } 24. static struct file\_system\_type myfs\_file\_system\_type = { 25. .owner = THIS\_MODULE, 26. .name = "myfs", 27. .mount = myfs\_mount, 28. .kill\_sb = myfs\_kill\_sb, 29. .fs\_flags = FS\_REQUIRES\_DEV, 30. .next = NULL, 31. } |

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,  };   1. static struct dentry \*myfs\_lookup(struct inode \*dir, struct dentry \*dentry, unsigned int flags); 3. static int myfs\_create(struct inode \*dir, struct dentry \*dentry, umode\_t mode, bool excl); 5. static int myfs\_unlink(struct inode \*dir, struct dentry \*dentry); 7. static int myfs\_mkdir(struct inode \*dir, struct dentry \*dentry, umode\_t mode); 9. static int myfs\_rmdir(struct inode \*dir, struct dentry \*dentry); 11. static int myfs\_link(struct dentry \*old\_dentry,struct inode \*dir, struct dentry \*dentry); 13. static int myfs\_symlink(struct inode \*dir, struct dentry \*dentry, const char \*symname); |

Листинг 14. Описание структуры file\_operations

|  |
| --- |
| 1. const struct file\_operations myfs\_file\_operations = { 2. .open   = myfs\_open, 3. .read   = myfs\_read\_file, 4. .write  = myfs\_write\_file, 5. } 6. static int myfs\_open(struct inode \*inode, struct file \*filp) 7. { 8. filp->private\_data = inode->i\_private; 9. return 0; 10. } 12. static ssize\_t myfs\_read\_file(struct file \*filp, char \*buffer,size\_t count, loff\_t \*offset) 13. { 14. printk(KERN\_INFO "Message: Start Reading"); 16. loff\_t pos = \*offset; 17. struct file\_data \*fdata = (struct file\_data \*) filp->private\_data; 18. char \*data = (char \*) fdata->data; 19. if (pos < 0) 20. return -EINVAL; 22. if (pos > MYFS\_MAX\_SIZE || !count) 23. return 0; 25. if (count > strlen(data) - pos) 26. count = strlen(data) - pos; 28. if (copy\_to\_user(buffer, data, count )) 29. { 30. return -EFAULT; 31. } 33. (\*offset) += count; 34. return count; 35. } 37. static ssize\_t myfs\_write\_file(struct file \*filp, const char \*buffer, size\_t count, loff\_t \*offset) 38. { 39. loff\_t pos = \*offset; 40. struct file\_data \*fdata = (struct file\_data \*) filp->private\_data; 41. char \* data = fdata->data; 42. char tmp[1024]; 43. if (pos < 0) 44. return -EINVAL; 46. size\_t buffer\_len = MYFS\_MAX\_SIZE;//strlen(buffer); 48. if (pos > buffer\_len || !count) 49. return 0;  52. if ( count > buffer\_len - pos ) { 53. count =  buffer\_len - pos; 54. } 56. if (copy\_from\_user(tmp, buffer, count) ) 57. { 58. return -EFAULT; 59. } 61. memcpy(data + (size\_t)(\*offset), tmp, count); 63. (\*offset) = pos + count; 65. return count; 66. } |

1. **Исследовательская часть**
   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
  1. **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

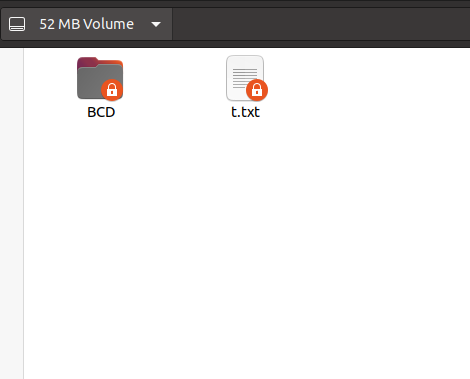


Рисунок 11. интерфейс программы

* 1. **Результат работы программы**

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

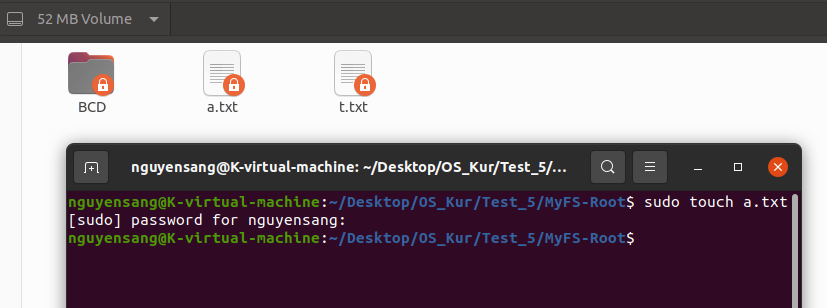


Рисунок 12. Работa операции **create()**

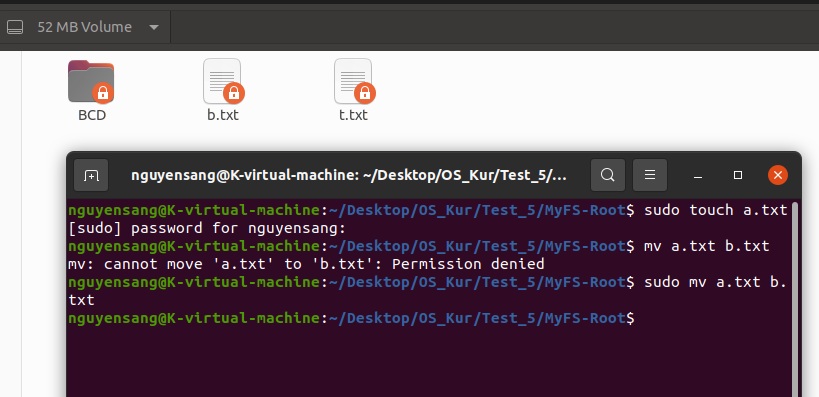


Рисунок 13. Работa операции **rename()**

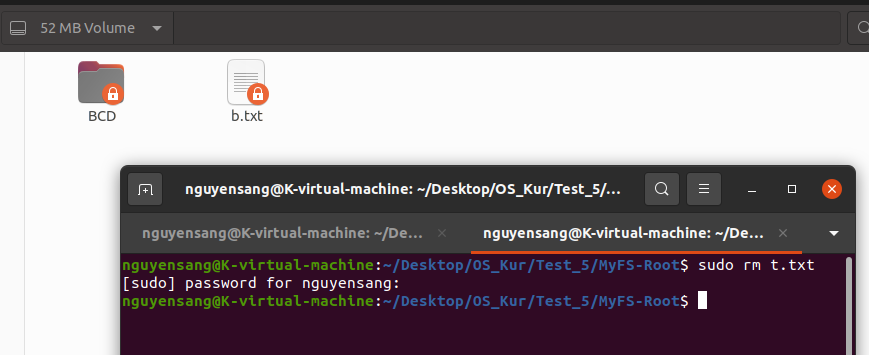


Рисунок 14. Работы операции **rmdir()**

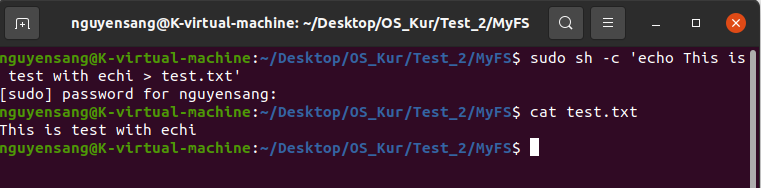


Рисунок 15. Работы операции **read , write**

* 1. **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]*

1. конспект лекций по курсу "Операционные системы"
2. The Linux Kernel's VFS Layer

*[https://www.usenix.org/legacy/publications/library/proceedings/usenix01/full\_papers/kroeger/kroeger\_html/node8.html]*

1. Using the page cache

*[https://cs4118.github.io/pantryfs/page-cache-overview.pdf]*

1. Linux File System: Virtual File System (VFS)

*[https://emmanuelbashorun.medium.com/linux-file-system-virtual-file-system-vfs-layer-part-3-79235c40a499]*

1. Виртуальная файловая система
2. *[http://www.cs.vsu.ru/~svv/ux/lecture%205.pdf]*
3. How to write a Linux VFS filesystem module

*[http://pages.cpsc.ucalgary.ca/~crwth/programming/VFS/VFS.php]*

1. 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.]*

1. Linux VFS

*[https://titanwolf.org/]*

# Приложение

Листинг 15. myfs.h

|  |
| --- |
| 1. #ifndef \_\_MYFS\_H\_\_ 2. #define \_\_MYFS\_H\_\_ 4. #include <linux/kernel.h> 5. #include <linux/init.h> 6. #include <linux/module.h> 7. #include <linux/pagemap.h>  /\* PAGE\_CACHE\_SIZE \*/ 8. #include <linux/fs.h>       /\* This is where libfs stuff is declared \*/ 9. #include <asm/atomic.h> 10. #include <asm/uaccess.h>    /\* copy\_to\_user \*/ 11. #include <linux/time.h> 13. #include <linux/buffer\_head.h> 14. #include <linux/slab.h> 15. #include <linux/statfs.h> 17. #define TMPSIZE 20 18. #define MYFS\_MAX\_SIZE 1024  21. /\* 22. \*  INODE 23. \*/ 25. int myfs\_rmdir(struct inode \*dir, struct dentry \*dentry); 27. int myfs\_rename(struct inode \*old\_dir, struct dentry \*old\_dentry, struct inode \*new\_dir, struct dentry \*new\_dentry, unsigned int dev); 29. int myfs\_unlink(struct inode \*dir, struct dentry \*dentry); 31. int myfs\_link(struct dentry \*old\_dentry, struct inode \*dir, struct dentry \*dentry); 33. struct dentry \*myfs\_lookup(struct inode \*dir, struct dentry \*dentry, unsigned int flags); 35. static struct inode \*myfs\_make\_inode(struct super\_block \*sb, int mode); 37. static int myfs\_create(struct inode \*dir, struct dentry \*dentry,umode\_t mode, bool excl); 39. const char \*myfs\_get\_link(struct dentry \*dentry, struct inode \*inode, struct delayed\_call \*done); 41. /\* 42. \*  FILE 43. \*/ 45. static int myfs\_open(struct inode \*inode, struct file \*filp); 47. static ssize\_t myfs\_read\_file(struct file \*filp, char \*buffer,size\_t count, loff\_t \*offset); 49. static ssize\_t myfs\_write\_file(struct file \*filp, const char \*buffer, size\_t count, loff\_t \*offset); 51. int myfs\_release(struct inode \*inode, struct file \*file); 53. static void lfs\_create\_files (struct super\_block \*sb, struct dentry \*root); 55. static struct dentry \*lfs\_create\_file (struct super\_block \*sb, struct dentry \*dir, const char \*name); 57. static struct dentry \*lfs\_create\_dir (struct super\_block \*sb, struct dentry \*parent, const char \*name); 59. struct file\_data { 60. char \* data; 61. size\_t size; 62. }; 64. struct file\_data \* create\_empty\_data(void);  67. /\* 68. \*  END FILE 69. \*/  72. static int myfs\_fill\_super (struct super\_block \*sb, void \*data, int silent); 74. static struct dentry \*myfs\_get\_super(struct file\_system\_type \*fst, int flags, const char \*devname, void \*data);  77. /\* 78. \*  STRUCT 79. \*/   83. const struct file\_operations myfs\_file\_operations = { 84. .open   = myfs\_open, 85. .read   = myfs\_read\_file, 86. .write  = myfs\_write\_file, 87. //.release= myfs\_release, 88. }; 90. const struct file\_operations myfs\_dir\_operations = { 91. .open       = dcache\_dir\_open, 92. .release    =  dcache\_dir\_close, 93. .llseek     = dcache\_dir\_lseek, 94. .read       = generic\_read\_dir, 95. .iterate    = dcache\_readdir, 96. .fsync      = noop\_fsync, 98. };   102. const struct super\_operations myfs\_s\_ops = { 103. .statfs     = simple\_statfs, 104. .drop\_inode = generic\_delete\_inode, 105. }; 107. struct file\_system\_type myfs\_type = { 108. .owner      = THIS\_MODULE, 109. .name       = "myfs", 110. .mount      = myfs\_get\_super, 111. .kill\_sb    = kill\_litter\_super, 112. }; 114. const struct inode\_operations myfs\_file\_inode\_operations= { 115. .rename = myfs\_rename, 116. .rmdir = myfs\_rmdir, 117. .unlink = myfs\_unlink, 118. .link = myfs\_link, 119. .lookup = myfs\_lookup, 120. .create = myfs\_create, 121. }; 123. const struct inode\_operations myfs\_dir\_inode\_operations = { 124. .rename = myfs\_rename, 125. .rmdir = myfs\_rmdir, 126. .unlink = myfs\_unlink, 127. .link = myfs\_link, 128. .lookup = myfs\_lookup, 129. .create = myfs\_create, 130. };  133. const struct inode\_operations myfs\_symlink\_inode\_operations = { 134. .get\_link = myfs\_get\_link, 135. }; 137. /\* 138. \*  END STRUCT 139. \*/ 140. #endif /\* \_\_MYFS\_H\_\_ \*/ |

Листинг 16. myfs.c

|  |
| --- |
| 1. #include <linux/kernel.h> 2. #include <linux/init.h> 3. #include <linux/module.h> 4. #include <linux/pagemap.h>  /\* PAGE\_CACHE\_SIZE \*/ 5. #include <linux/fs.h>       /\* This is where libfs stuff is declared \*/ 6. #include <asm/atomic.h> 7. #include <asm/uaccess.h>    /\* copy\_to\_user \*/ 9. #include <linux/time.h> 10. #include "myfs.h" 12. #define LFS\_MAGIC 0x19920342  15. static int myfs\_fill\_super (struct super\_block \*sb, void \*data, int silent) 16. { 17. struct inode \*root; 18. struct dentry \*root\_dentry; 20. sb->s\_blocksize = VMACACHE\_SIZE; 21. sb->s\_blocksize\_bits = VMACACHE\_SIZE; 22. sb->s\_magic = LFS\_MAGIC; 23. sb->s\_op = &myfs\_s\_ops; 25. root = myfs\_make\_inode (sb, S\_IFDIR | 0755); 26. inode\_init\_owner(root, NULL, S\_IFDIR | 0755); 27. if (! root) 28. goto out; 29. root->i\_op = &myfs\_dir\_inode\_operations; 30. root->i\_fop = &myfs\_dir\_operations; 32. set\_nlink(root, 2); 33. root\_dentry = d\_make\_root(root); 34. if (! root\_dentry) 35. goto out\_iput; 37. lfs\_create\_files (sb, root\_dentry); 38. sb->s\_root = root\_dentry; 39. return 0; 41. out\_iput: 42. iput(root); 43. out: 44. return -ENOMEM; 45. } 47. static struct dentry \*myfs\_get\_super(struct file\_system\_type \*fst, int flags, const char \*devname, void \*data) 48. { 49. return mount\_nodev(fst, flags, data, myfs\_fill\_super); 50. }  53. static int \_\_init myfs\_init(void) 54. { 55. return register\_filesystem(&myfs\_type); 56. } 58. static void \_\_exit myfs\_exit(void) 59. { 60. unregister\_filesystem(&myfs\_type); 61. } 63. module\_init(myfs\_init); 64. module\_exit(myfs\_exit); 66. MODULE\_LICENSE("GPL");  69. /\* 70. \*      INODE 71. \*/ 73. const char \*myfs\_get\_link(struct dentry \*dentry, struct inode \*inode, struct delayed\_call \*done) 74. { 75. return inode->i\_link; 76. } 78. int myfs\_rmdir(struct inode \*dir, struct dentry \*dentry) 79. { 80. if (!simple\_empty(dentry)) 81. return -ENOTEMPTY; 83. drop\_nlink(d\_inode(dentry)); 84. simple\_unlink(dir, dentry); 85. drop\_nlink(dir); 86. return 0; 87. } 89. int myfs\_rename(struct inode \*old\_dir, struct dentry \*old\_dentry, struct inode \*new\_dir, struct dentry \*new\_dentry, unsigned int dev) 90. { 91. printk(KERN\_INFO "Message: Start Rename\n"); 93. struct inode \*inode = d\_inode(old\_dentry); 94. int they\_are\_dirs = d\_is\_dir(old\_dentry); 96. if (!simple\_empty(new\_dentry)) 97. return -ENOTEMPTY; 99. if (d\_really\_is\_positive(new\_dentry)) { 100. simple\_unlink(new\_dir, new\_dentry); 101. if (they\_are\_dirs) { 102. drop\_nlink(d\_inode(new\_dentry)); 103. drop\_nlink(old\_dir); 104. } 105. } else if (they\_are\_dirs) { 106. drop\_nlink(old\_dir); 107. inc\_nlink(new\_dir); 108. } 110. old\_dir->i\_ctime = old\_dir->i\_mtime = new\_dir->i\_ctime = 111. new\_dir->i\_mtime = inode->i\_ctime = current\_time(inode); 113. return 0; 114. } 116. int myfs\_unlink(struct inode \*dir, struct dentry \*dentry) 117. { 118. struct inode \*inode = d\_inode(dentry); 120. inode->i\_ctime = dir->i\_ctime = dir->i\_mtime = current\_time(inode); 121. drop\_nlink(inode); 122. dput(dentry); 123. return 0; 124. } 126. int myfs\_link(struct dentry \*old\_dentry, struct inode \*dir, struct dentry \*dentry) 127. { 128. struct inode \*inode = d\_inode(old\_dentry); 130. inode->i\_ctime = dir->i\_ctime = dir->i\_mtime = current\_time(inode); 131. inc\_nlink(inode); 132. ihold(inode); 133. dget(dentry); 134. d\_instantiate(dentry, inode); 135. return 0; 136. } 138. struct dentry \*myfs\_lookup(struct inode \*dir, struct dentry \*dentry, unsigned int flags) 139. { 140. printk(KERN\_INFO "Message: Start Lookup\n"); 142. if (dentry->d\_name.len > NAME\_MAX) 143. return ERR\_PTR(-ENAMETOOLONG); 144. if (!dentry->d\_sb->s\_d\_op) 145. d\_set\_d\_op(dentry, &simple\_dentry\_operations); 146. d\_add(dentry, NULL); 147. return NULL; 148. } 150. static struct inode \*myfs\_make\_inode(struct super\_block \*sb, int mode) 151. { 152. struct inode\* inode; 153. inode = new\_inode(sb); 154. if (!inode) { 155. return NULL; 156. } 157. inode->i\_mode = mode; 158. inode->i\_atime = inode->i\_mtime = inode->i\_ctime = current\_time(inode); 159. if (S\_ISREG(mode)) 160. { 161. inode->i\_fop = &myfs\_file\_operations; 162. inode->i\_op = &myfs\_file\_inode\_operations; 163. set\_nlink(inode, 1); 164. struct file\_data \* data = (struct file\_data \*) create\_empty\_data(); 165. inode->i\_private = data; 166. } 167. else if (S\_ISDIR(mode)) 168. { 169. inode->i\_fop = &myfs\_dir\_operations; 170. inode->i\_op = &myfs\_dir\_inode\_operations; 171. set\_nlink(inode, 2); 172. } 173. inode->i\_ino = get\_next\_ino(); 174. return inode; 176. } 178. static int myfs\_create(struct inode \*dir, struct dentry \*dentry,umode\_t mode, bool excl) 179. { 180. printk(KERN\_INFO "Message: Start Create\n"); 181. struct inode \*inode; 183. inode = myfs\_make\_inode(dir->i\_sb, mode | S\_IFREG); 184. if (!inode) 185. goto out; 186. inode\_init\_owner(inode, dir, mode | S\_IFREG); 187. //d\_instantiate(dentry, inode); 188. d\_add(dentry, inode); 189. dget(dentry); 191. out: 192. return 0; 193. }  196. static void lfs\_create\_files (struct super\_block \*sb, struct dentry \*root) 197. { 198. struct dentry \*subdir; 200. lfs\_create\_file(sb, root, "test.txt"); 202. subdir = lfs\_create\_dir(sb, root, "Dir1"); 203. if (subdir) 204. lfs\_create\_file(sb, subdir, "test.txt"); 205. }  208. static struct dentry \*lfs\_create\_dir (struct super\_block \*sb, 209. struct dentry \*parent, const char \*name) 210. { 211. struct dentry \*dentry; 212. struct inode \*inode; 214. dentry = d\_alloc\_name(parent, name); 215. if (! dentry) 216. goto out; 218. inode = myfs\_make\_inode(sb, S\_IFDIR | 0755); 219. if (! inode) 220. goto out\_dput; 221. inode->i\_op = &myfs\_dir\_inode\_operations; 223. d\_add(dentry, inode); 224. return dentry; 226. out\_dput: 227. dput(dentry); 228. out: 229. return 0; 230. }  233. static struct dentry \*lfs\_create\_file (struct super\_block \*sb, struct dentry \*dir, const char \*name) 234. { 235. struct dentry \*dentry; 236. struct inode \*inode; 238. dentry = d\_alloc\_name(dir, name); 239. if (! dentry) 240. goto out; 241. inode = myfs\_make\_inode(sb, S\_IFREG | 0644); 242. if (! inode) 243. goto out\_dput; 244. strcpy(((struct file\_data \*) inode->i\_private)->data, name); 246. d\_add(dentry, inode); 247. return dentry; 249. out\_dput: 250. dput(dentry); 251. out: 252. return 0; 253. }  256. static int myfs\_open(struct inode \*inode, struct file \*filp) 257. { 258. filp->private\_data = inode->i\_private; 259. return 0; 260. } 262. static ssize\_t myfs\_read\_file(struct file \*filp, char \*buffer,size\_t count, loff\_t \*offset) 263. { 264. printk(KERN\_INFO "Message: Start Reading"); 266. loff\_t pos = \*offset; 267. struct file\_data \*fdata = (struct file\_data \*) filp->private\_data; 268. char \*data = (char \*) fdata->data; 269. if (pos < 0) 270. return -EINVAL; 272. if (pos > MYFS\_MAX\_SIZE || !count) 273. return 0; 275. if (count > strlen(data) - pos) 276. count = strlen(data) - pos; 278. if (copy\_to\_user(buffer, data, count )) 279. { 280. return -EFAULT; 281. } 283. (\*offset) += count; 284. return count; 285. } 287. static ssize\_t myfs\_write\_file(struct file \*filp, const char \*buffer, size\_t count, loff\_t \*offset) 288. { 289. printk(KERN\_INFO "Message: Start Writing offset = %lld", \*offset); 290. loff\_t pos = \*offset; 291. struct file\_data \*fdata = (struct file\_data \*) filp->private\_data; 292. char \* data = fdata->data; 293. char tmp[1024]; 294. printk(KERN\_INFO "Message: Write Check 1"); 296. if (pos < 0) 297. return -EINVAL; 299. printk(KERN\_INFO "Message: Write Check 2"); 301. size\_t buffer\_len = MYFS\_MAX\_SIZE;//strlen(buffer); 303. printk(KERN\_INFO "Message: Write Check 3"); 305. if (pos > buffer\_len || !count) 306. return 0;  309. if ( count > buffer\_len - pos ) { 310. count =  buffer\_len - pos; 311. } 313. printk(KERN\_INFO "Message: Write Check 4"); 315. if (copy\_from\_user(tmp, buffer, count) ) 316. { 317. return -EFAULT; 318. } 320. printk(KERN\_INFO "Message: Write Check 5 "); 322. memcpy(data + (size\_t)(\*offset), tmp, count); 324. printk(KERN\_INFO "Message: Write Check 6"); 325. (\*offset) = pos + count; 327. return count; 328. } 330. int myfs\_release(struct inode \*inode, struct file \*file) 331. { 332. kfree(file->private\_data); 333. return 0; 334. }  337. struct file\_data \* create\_empty\_data(void) 338. { 339. struct file\_data \* new = (struct file\_data \*) kzalloc(sizeof(struct file\_data), GFP\_KERNEL); 340. new->data = (char \*)kzalloc(MYFS\_MAX\_SIZE, GFP\_KERNEL); 341. memset(new->data, 0, MYFS\_MAX\_SIZE); 342. new->size = 0; 343. return new; 344. } 345. /\* 346. \*      END FILE 347. \*/ |

Листинг 17. Makefile

|  |
| --- |
| 1. obj-m += fs.o 2. fs-objs := myfs.o 4. CURRENT = $(shell uname -r) 5. KDIR = /lib/modules/$(CURRENT)/build 6. PWD = $(shell pwd) 8. all: 9. $(MAKE) -C $(KDIR) M=$(PWD) modules 10. clean: 11. @rm -f \*.o .\*.cmd .\*.flags \*.mod.c \*.order 12. @rm -f .\*.\*.cmd \*~ \*.\*~ TODO.\* 13. @rm -fR .tmp\* 14. @rm -rf .tmp\_versions 15. disclean: clean 16. @rm \*.ko \*.symvers |