Министерство образования Республики Беларусь

Учреждение образования «Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра электронных вычислительных машин

Дисциплина: Операционные системы и системное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

на тему

ПРОСТАЯ ФАЙЛОВАЯ СИСТЕМА (SFS) В ПРОСТРАНСТВЕ ПОЛЬЗОВАТЕЛЯ

БГУИР КР 6–05 0611–05 321 ПЗ

Студент: гр.350503 Санько А.С.

Руководитель Протько М.А.

Минск 2025

Министерство образования Республики Беларусь

Министерство образования Республики Беларусь

Учреждение образования

Белорусский государственный университет  
информатики и радиоэлектроники

Факультет: КСиС. Кафедра: ЭВМ.

Специальность: 6–05–0611–05 «Компьютерная инженерия».

Профилизация: «Вычислительные машины, системы и сети».

Утверждаю  
 Заведующий кафедрой ЭВМ  
 \_\_\_\_\_\_\_\_\_ Б.В. Никульшин  
 «\_\_\_\_» \_\_\_\_\_\_\_\_\_\_ 2025 г.

Задание  
по курсовому проекту студента

Санько Александра Сергеевича

**1** Тема проекта: «Простая файловая система (SFS) в пространстве пользователя»

**2** Срок сдачи студентом законченного проекта: 23 мая 2025 г.

**3** Исходные данные к проекту:

Операционная система: Linux; Язык программирования: C; Цель: разработать файловую систему, работающую в пространстве пользователя; Задачи: 1 анализ предметной области. 2 проектирование структуры файловой системы. 3 реализация операций работы с файловой системой. 4 обеспечение сохранности и целостности данных. 5 тестирование функциональности. 6 разработка документации. Алгоритмы, подлежащие разработке: алгоритм создания файловой системы, алгоритм монтирования файловой системы, алгоритм чтения файлов, алгоритм просмотра содержимого директорий, алгоритм создания корневой директории.

**4** Содержание пояснительной записки (перечень подлежащих разработке вопросов):

Введение. 1 Обзор источников литературы. 2 Системное проектирование. 3 Функциональное проектирование. 4 Разработка программных модулей. 5 Программа и методика испытаний. 6 Руководство пользователя. Заключение. Приложения.

**5** Перечень графического материала (с точным указанием обязательных чертежей):

5.1 Простая файловая система (SFS) в пространстве пользователя. Схема структурная.

5.2 Простая файловая система (SFS) в пространстве пользователя. Диаграмма последовательности.

5.6 Простая файловая система (SFS) в пространстве пользователя. Create\_home\_directory. Схема программы.

5.7 Простая файловая система (SFS) в пространстве пользователя. sfs\_mkfs. Схема программы.

Календарный план

|  |  |  |  |
| --- | --- | --- | --- |
| Наименование этапов курсового проекта | Объем этапа, % | Срок выполнения этапа | Примечания |
| Подбор и изучение литературы | 10 | 21.02 – 28.02 |  |
| Системное проектирование | 20 | 01.03 – 15.03 |  |
| Функциональное проектирование | 30 | 16.03 – 31.03 |  |
| Разработка программных модулей | 20 | 01.04 – 15.04 |  |
| Программа и методика испытаний | 5 | 16.04 – 30.04 |  |
| Руководство пользователя | 5 | 01.05 – 09.05 |  |
| Оформление пояснительной записки и графического материала | 10 | 10.05 – 22.05 |  |
| Защита курсового проекта |  | 23.05 – 06.06 |  |

Дата выдачи задания: 21 февраля 2025 г.

Руководитель М.А. Протько

Задание принял к исполнению \_\_\_\_\_\_\_\_\_ А.С. Санько

СОДЕРЖАНИЕ

[ВВЕДЕНИЕ 5](#_Toc198499765)

[1. ОБЗОР ЛИТЕРАТУРЫ 6](#_Toc198499766)

[1.1 Что такое файловая система 6](#_Toc198499767)

[1.1.1 Определение файловой системы 6](#_Toc198499768)

[1.1.2 Основные функции файловых систем 7](#_Toc198499769)

[1.1.3 Задачи файловой системы 7](#_Toc198499770)

[1.1.4 Файловые системы Linux 8](#_Toc198499771)

[1.2 Простая файловая система 9](#_Toc198499772)

[1.2.1 Метаинформация 9](#_Toc198499773)

[1.2.2 Хранение на диске 9](#_Toc198499774)

[1.3 Алгоритмы работы 10](#_Toc198499775)

[1.3.1 Чтение 10](#_Toc198499776)

[1.3.2 Дозапись 10](#_Toc198499777)

[1.3.3 Локальность хранения 11](#_Toc198499778)

[1.3.4 Сбои и восстановление 11](#_Toc198499779)

[1.4 FUSE 12](#_Toc198499780)

[2. СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ 13](#_Toc198499781)

[2.1 Введение в системное проектирование 13](#_Toc198499782)

[2.2 Архитектура системы 13](#_Toc198499783)

[2.2.1 Основные компоненты системы 13](#_Toc198499784)

[2.2.2 Структура данных и хранилище 14](#_Toc198499785)

[2.3 Алгоритмы работы с файловой системой 15](#_Toc198499786)

[2.3.1 Алгоритм записи файла 15](#_Toc198499787)

[2.3.2 Алгоритм чтения файла 16](#_Toc198499788)

[2.3.3 Алгоритм удаления файла 16](#_Toc198499789)

[2.3.4 Алгоритм создания директории 16](#_Toc198499790)

[2.4 Выбор технологий и инструментов 17](#_Toc198499791)

[2.4.1 Управление памятью 17](#_Toc198499792)

[2.4.2 Ввод/вывод 18](#_Toc198499793)

[2.5 Обеспечение надежности и производительности 18](#_Toc198499794)

[2.5.1 Надежность 18](#_Toc198499795)

[2.5.2 Производительность 18](#_Toc198499796)

[2.6 Заключение 18](#_Toc198499797)

[3. ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ 20](#_Toc198499798)

[3.1 Назначение 20](#_Toc198499799)

[3.2 Общая структура 20](#_Toc198499800)

[3.3 Superblock 20](#_Toc198499801)

[3.4 Inode 21](#_Toc198499802)

[3.5 DirectoryEntry 21](#_Toc198499803)

[3.6 Блоки данных 22](#_Toc198499804)

[3.7 Основные функции 22](#_Toc198499805)

[3.7.1 Форматирование диска 22](#_Toc198499806)

[3.7.2 Монтирование и размонтирование 22](#_Toc198499807)

[3.7.3 Помощь в использовании 22](#_Toc198499808)

[3.7.4 Помощь в использовании 23](#_Toc198499809)

[3.7.5 Создания домашнего каталога 23](#_Toc198499810)

[3.7.6 Рекурсивное удаление каталога 23](#_Toc198499811)

[3.7.7 Создание директорий 23](#_Toc198499812)

[3.7.8 Перемещение файлов 23](#_Toc198499813)

[3.7.9 Удаление директорий 23](#_Toc198499814)

[3.7.10 Смена директории 24](#_Toc198499815)

[3.7.11 Просмотр содержимого директории 24](#_Toc198499816)

[3.7.12 Создание файла 24](#_Toc198499817)

[3.7.13 Запись данных в файл 24](#_Toc198499818)

[3.7.14 Чтение данных из файла 25](#_Toc198499819)

[3.7.15 Удаление файла 25](#_Toc198499820)

[3.7.16 Разрешение пути 25](#_Toc198499821)

[3.8 Обработка ошибок 25](#_Toc198499822)

[4. РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ 26](#_Toc198499823)

[4.1 Разработка структурной схемы 26](#_Toc198499824)

[4.2 Разработка диаграммы последовательностей 26](#_Toc198499825)

[4.3 Разработка схем алгоритмов 26](#_Toc198499826)

[4.4 Описание алгоритма sfs\_ls\_dir(const char \*dirname) по шагам 26](#_Toc198499827)

[4.5 Описание алгоритма sfs\_read(const char \*filename) по шагам 27](#_Toc198499828)

[5. ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ 29](#_Toc198499829)

[5.1 Цель испытаний 29](#_Toc198499830)

[6. РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ 33](#_Toc198499831)

[6.1 Назначение программы 33](#_Toc198499832)

[6.2 Требования к окружению 33](#_Toc198499833)

[6.3 Установка и запуск 33](#_Toc198499834)

[6.4 Основные команды пользователя 33](#_Toc198499835)

[6.5 Особенности использования 34](#_Toc198499836)

[6.6 Возможные ошибки и их решения 34](#_Toc198499837)

[ЗАКЛЮЧЕНИЕ 35](#_Toc198499838)

[СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 36](#_Toc198499839)

[ПРИЛОЖЕНИЕ А 37](#_Toc198499840)

[ПРИЛОЖЕНИЕ Б 38](#_Toc198499841)

[ПРИЛОЖЕНИЕ В 39](#_Toc198499842)

[ПРИЛОЖЕНИЕ Г 40](#_Toc198499843)

[ПРИЛОЖЕНИЕ Д 41](#_Toc198499844)

[ПРИЛОЖЕНИЕ Е 64](#_Toc198499845)

## [**ВВЕДЕНИЕ**](#_top)

В современных операционных системах файловые системы играют ключевую роль, обеспечивая удобное и надежное управление данными. Они представляют собой неотъемлемую часть инфраструктуры, необходимую для работы практически всех программных приложений. Разработка и изучение файловых систем имеет большое значение как с точки зрения теоретической информатики, так и для практического использования в индустрии.

Целью данной курсовой работы является создание простой файловой системы (SFS) в пространстве пользователя. Простая файловая система представляет собой минималистичную реализацию, способную выполнять базовые операции, такие как создание, чтение, запись и удаление файлов и каталогов. Ее разработка позволяет изучить фундаментальные принципы работы файловых систем и исследовать их взаимодействие с операционной системой и аппаратным обеспечением.

Работа сосредоточена на реализации файловой системы в пространстве пользователя, что дает возможность изолировать проект от сложностей работы на уровне ядра. Данный метод способствует оптимизации процессов разработки, тестирования и диагностики программного кода, одновременно расширяя перспективы для экспериментального исследования архитектурных решений файловых систем.

В рамках курсовой работы будут рассмотрены основные аспекты проектирования файловой системы, включая структурирование данных, управление пространством хранения, обеспечение производительности и обработку ошибок. Также будут представлены результаты реализации и анализа работы SFS, а также выводы о возможностях ее дальнейшего совершенствования.

Таким образом, данный проект не только способствует углублению понимания работы файловых систем, но и развивает навыки разработки и анализа программного обеспечения, что является важной частью профессиональной подготовки в сфере информационных технологий.

## **ОБЗОР ЛИТЕРАТУРЫ**

### **Что такое файловая система**

#### **Определение файловой системы**

Обычно вся информация записывается, хранится и обрабатывается на различных цифровых носителях в виде файлов. Далее, в зависимости от типа файла, кодируется в виде расширений – \*exe, \*doc, \*pdf, которые открываются и обрабатываются в соответствующем программном обеспечении.

Операционная система воспринимает физический диск хранения информации как набор кластеров размером 512 байт и больше. Драйверы файловой системы организуют кластеры в файлы и каталоги, которые также являются файлами, содержащими список других файлов в этом каталоге. Эти же драйверы отслеживают, какие из кластеров в настоящее время используются, какие свободны, какие помечены как неисправные.

Запись файлов большого объема приводит к необходимости фрагментации, когда файлы не сохраняются как целые единицы, а делятся на фрагменты. Каждый фрагмент записывается в отдельные кластеры, состоящие из ячеек (размер ячейки составляет один байт). Информация о всех фрагментах, как части одного файла, хранится в файловой системе.

Файловая система связывает носитель информации (хранилище) с прикладным программным обеспечением, организуя доступ к конкретным файлам при помощи функционала взаимодействия программ API. Программа, при обращении к файлу, располагает данными только о его имени, размере и атрибутах. Всю остальную информацию, касающуюся типа носителя, на котором записан файл, и структуры хранения данных, она получает от драйвера файловой системы.

На физическом уровне драйверы ФС оптимизируют запись и считывание отдельных частей файлов для ускоренной обработки запросов, фрагментации и объединения хранящейся в ячейках информации. Данный алгоритм получил распространение в большинстве популярных файловых систем на концептуальном уровне в виде иерархической структуры представления метаданных. Технология снижает количество самых длительных дисковых операций – позиционирования головок при чтении произвольных блоков. Это позволяет не только ускорить обработку запросов, но и продлить срок службы HDD. В случае с твердотельными накопителями, где принцип записи, хранения и считывания информации отличается от применяемого в жестких дисках, ситуация с выбором оптимальной файловой системы имеет свои нюансы [1].

#### **Основные функции файловых систем**

Файловая система отвечает за оптимальное логическое распределение информационных данных на конкретном физическом носителе. Драйвер ФС организует взаимодействие между хранилищем, операционной системой и прикладным программным обеспечением. Правильный выбор файловой системы для конкретных пользовательских задач влияет на скорость обработки данных, принципы распределения и другие функциональные возможности, необходимые для стабильной работы любых компьютерных систем. Иными словами, это совокупность условий и правил, определяющих способ организации файлов на носителях информации.

Основными функциями файловой системы являются [1]:

– размещение и упорядочивание на носителе данных в виде файлов;

– определение максимально поддерживаемого объема данных на носителе информации;

– создание, чтение и удаление файлов;

– назначение и изменение атрибутов файлов (размер, время создания и изменения, владелец и создатель файла, доступен только для чтения, скрытый файл, временный файл, архивный, исполняемый, максимальная длина имени файла);

– определение структуры файла;

– поиск файлов;

– организация каталогов для логической организации файлов;

– защита файлов при системном сбое;

– защита файлов от несанкционированного доступа и изменения их содержимого.

#### **Задачи файловой системы**

Функционал файловой системы нацелен на решение следующих задач:

– присвоение имен файлам;

– программный интерфейс работы с файлами для приложений;

– отображение логической модели файловой системы на физическую организацию хранилища данных;

– поддержка устойчивости файловой системы к сбоям питания, ошибкам аппаратных и программных средств;

– содержание параметров файла, необходимых для правильного взаимодействия с другими объектами системы (ядро, приложения и пр.).

В многопользовательских системах реализуется задача защиты файлов от несанкционированного доступа, обеспечение совместной работы. При открытии файла одним из пользователей для других этот же файл временно будет доступен в режиме «только чтение». Вся информация о файлах хранится в особых областях раздела (томах). Структура справочников зависит от типа файловой системы. Справочник файлов позволяет ассоциировать числовые идентификаторы уникальных файлов и дополнительную информацию о них с непосредственным содержимым файла, хранящимся в другой области раздела.

#### **Файловые системы Linux**

В отличие от ОС Windows и macOS, ограничивающих выбор файловой системы предустановленными вариантами, Linux предоставляет возможность использования нескольких ФС, каждая из которых оптимизирована для решения определенных задач. Файловые системы в Linux используются не только для работы с файлами на диске, но и для хранения данных в оперативной памяти или доступа к конфигурации ядра во время работы системы. Все они включены в ядро и могут использоваться в качестве корневой файловой системы.

Основные файловые системы, используемые в дистрибутивах Linux:

– Ext2;

– Ext3;

– Ext4;

– JFS;

– ReiserFS;

– XFS;

– Btrfs;

– ZFS.

Ext2, Ext3, Ext4 или Extended Filesystem – стандартная файловая система, первоначально разработанная для Minix. Содержит максимальное количество функций и является наиболее стабильной в связи с редкими изменениями кодовой базы. Начиная с ext3 в системе используется функция журналирования. Сегодня версия ext4 присутствует во всех дистрибутивах Linux.

JFS или Journaled File System разработана в IBM в качестве альтернативы для файловых систем ext. Сейчас она используется там, где необходима высокая стабильность и минимальное потребление ресурсов (в первую очередь в многопроцессорных компьютерах). В журнале хранятся только метаданные, что позволяет восстанавливать старые версии файлов после сбоев.

ReiserFS также разработана в качестве альтернативы ext3, поддерживает только Linux. Динамический размер блока позволяет упаковывать несколько небольших файлов в один блок, что предотвращает фрагментацию и улучшает работу с небольшими файлами. Недостатком является риск потери данных при отключении энергии.

XFS рассчитана на файлы большого размера, поддерживает диски до 2 терабайт. Преимуществом системы является высокая скорость работы с большими файлами, отложенное выделение места, увеличение разделов на лету, незначительный размер служебной информации. К недостаткам относится невозможность уменьшения размера, сложность восстановления данных и риск потери файлов при аварийном отключении питания.

Btrfs или B–Tree File System легко администрируется, обладает высокой отказоустойчивостью и производительностью. Используется как файловая система по умолчанию в OpenSUSE и SUSE Linux.

Другие ФС, такие как NTFS, FAT, HFS, могут использоваться в Linux, но корневая файловая система на них не устанавливается, поскольку они для этого не предназначены [1].

### **Простая файловая система**

#### **Метаинформация**

Данные хранятся в виде блоков, а информация о принадлежности блоков к конкретным файлам фиксируется в специализированных структурных единицах – inode. Термин "inode" является производным от "index" и первоначально подразумевал размещение этих структур в таблице, обеспечивающей доступ к ним посредством индексирования. Такие структуры выполняют ключевую роль в файловых системах, позволяя организовывать адресацию и управление файловыми ресурсами на уровне операционной системы.

В inode записывается метаинформация файлов и каталогов, которая используется для работы с этими файлами. А также хранятся ссылки на блоки, где лежат данные файла.

Директория представляет собой особый вид файла, в содержимом которого перечислены файлы, принадлежащие данной директории и то, каким inode они соответствуют. В некоторых файловых системах могут быть ограничения на длину имени файлов. Имена файлов могут храниться просто списком, как в ранних файловых системах, в том числе ext2, но поиск по имени тогда будет требовать линейного прохода. В более новых файловых системах используются специальные структуры данных (например HTree в ext3/ext4). Специальные директории "." (current directory) и ".." (parent directory) реализуются путём хранения имён "." и ".." в директории (они автоматически создаются в момент инициализации директории и не могут быть удалены). Таким образом, всё дерево файлов и каталогов инкапсулируется в inode [3].

#### **Хранение на диске**

Пусть диск разбит на блоки по 4 КБ.

Сам inode – это структура, которая занимает десятки байт. Нулевой inode соответствует корню файловой системы. В одном блоке может содержаться большое количество inode. Идут они по порядку один за одним. Затем идут блоки с данными.

При удалении файлов inode становится свободным. Также освобождается блок, в котором хранились данные файла. Файловой системе нужно хранить некоторые структуры данных, чтобы находить свободные блоки. В первых версиях они связывались в некий список свободных блоков, позже от такого подхода отказались. Более производительное решение – хранить битовую маску свободных блоков. Каждому блоку соответствует один бит, для всех блоков получается битовый вектор. Также нужен битовый вектор для inode.

Первый блок обычно хранит метаинформацию о файловой системе и называется суперблоком. Это нужно, чтобы утилиты могли распознать тип файловой системы.

### **Алгоритмы работы**

#### **Чтение**

Описание алгоритма чтения файла /foo/bar:

1. Считывается информация из inode 0, там корневая директория. Приходится с диска вычитать целый блок, внутри которого есть нужный inode.

2. В этом нулевом inode есть ссылка на блок данных.

3. Читается этот блок, в нём содержимое корневого каталога.

4. Одна из записей там «foo, 5».

5. Читается блок, в котором находиться информация о inode 5.

6. В этом inode есть ссылка на блок данных, в котором находится содержимое каталога foo.

7. Считывается это содержимое.

8. Происходит поиск в нём и обнаруживается, запись bar с соответствующим ей номером inode.

9. Считывается информация из inode.

10. Считывается первый блок с данными нужного файла.

Итого необходимо шесть чтений с диска, для нахождения данных, содержащихся в файле. Это много. Исходя из этого в ОС организывывается кеш для хранения метаданных и для самих данных.

#### **Дозапись**

Описание алгоритма дозаписи данных в файл /foo/bar:

Кроме непосредственной записи данных, в inode обновляется размер файла. Таким образом, если обновленные данные не превосходят размер блока понадобиться две записи.

Если обновленные данные требуют выделения нового блока, будет ещё обновляться ссылка на блок, а также маска занятости. Все эти операции требуют обращения к разным местам диска. Следовательно необходимо организовать и для этих операций кеширование.

#### **Локальность хранения**

Чтобы сократить количество перемещений головки и улучшить локальность хранения, диск разбивается на зоны. Такой подход был применён в файловой системе ext2 (усовершенствование по сравнению с файловой системой ext).

В каждой группе есть свои битовые маски, inode и данные. Кроме того, в каждой группе присутствует копия суперблока.

При записи действуют некоторые эвристики. Файлы из одной директории обычно записываются в одну и ту же зону. Если файл очень большой, его можно распределить большими частями по многим группам (когда время чтения сравнимо со временем поиска, локальность уже не столь важна).

#### **Сбои и восстановление**

При записи приходится обновлять три блока: data bitmap, inode и блок данных. Если специально не предпринимать никаких мер, то при сбое данные могут быть утперяны, а файловая система окажется в неконсистентном состоянии.

Когда операционная система загружается, проверяется, корректно ли был отмонтирован раздел с файловой системой. Если нет, запускается утилита fsck, которая проверяет корректность файловой системы: inode корректны, счётчики ссылок согласованы, разные inode'ы указывают на разные данные, у директорий корректное содержимое.

1. Если при записи не обновилась битовая маска, а данные и изменения в inode записались успешно. Тогда новые данные могут быть записаны вместо старых, ведь блок помечен как свободный. При запуске системы такую ситуацию можно разрешить. Для этого надо прочитать информацию из всех inode. Если блок используемый, но в маске помечен как свободный, это необходимо исправить.

2. Если все данные записались, но не обновилась информация в inode. Данные записаны, но недоступны. При запуске ОС эту ситуацию тоже можно разрешить, прочитав информацию всех inode.

3. Если информация в inode и битовую маску записались, а данные – нет. Восстановление невозможно, данные утеряны.

4. Если записались данные только битовую маску. Ситуация некритичная, данные можно восстановить, как и в первых двух случаях.

Полное чтение при запуске системы по всем inode очень долго работает, нужно прочитать много данных. Размеры хранилищ год за годом увеличиваются, это становится непрактично. fsck может работать часами [3].

### **FUSE**

Filesystem in Userspace (FUSE) – модуль для ядер Unix–подобных операционных систем, позволяет разработчикам создавать новые типы файловых систем, доступные для монтирования пользователями без привилегий (прежде всего – виртуальных файловых систем); это достигается за счёт запуска кода файловой системы в пользовательском пространстве, в то время как модуль FUSE предоставляет связующее звено для актуальных интерфейсов ядра. C использованием средств FUSE разработаны, в частности, SSHFS, NTFS–3G, GlusterFS.

Доступна для Linux, FreeBSD, OpenBSD, NetBSD (as puffs), OpenSolaris, Minix 3, Android и macOS.

Для реализации новой файловой системы необходимо написать программу–обработчик и слинковать её с библиотекой libfuse. Основная цель этой программы – описать, как файловая система должна отвечать на запросы read/write/stat. Программа также используется для монтирования новой файловой системы. Во время монтирования файловой системы обработчик регистрируется в ядре. Если пользователь теперь выдает запросы read/write/stat для этой вновь смонтированной файловой системы, ядро перенаправляет эти IO–запросы обработчику, а затем отправляет ответ обработчика обратно пользователю [2].

Процесс работы:

1. Пользователь или процесс выполняет системный вызов (например, open("/mnt/fuse/myfile.txt")).

2. Ядро направляет запрос в модуль FUSE.

3. Модуль ядра передаёт запрос в пользовательскую программу через /dev/fuse.

4. Программа-обработчик, реализованная с помощью libfuse, анализирует запрос и возвращает результат обратно в модуль.

5. Модуль ядра возвращает результат системному вызову, и приложение получает данные или результат операции.

## **СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ**

### **Введение в системное проектирование**

Системное проектирование файловой системы – это ключевая стадия разработки, на которой происходит определение всех технических характеристик системы, а также выбор методов и алгоритмов для эффективного хранения и управления данными. Проектирование простой файловой системы (SFS), реализуемой в пространстве пользователя, предполагает создание не только структуры для хранения файлов, но и оптимизацию ее для работы в ограниченных условиях (например, без прав администратора). Это проектирование требует внимательного подхода к выбору алгоритмов для работы с данными, безопасности, производительности, а также надежности.

Файловая система является неотъемлемой частью любой операционной системы, поскольку она обеспечивает структурированное и упорядоченное хранение данных, а также управление доступом к ним. Однако традиционные файловые системы, такие как ext4, NTFS и FAT, действуют на уровне ядра операционной системы и имеют свои ограничения в плане гибкости. Проект SFS решает эту проблему, предоставляя возможность организовать файловую систему в пространстве пользователя, что позволяет избежать зависимости от конкретного ядра ОС и упрощает разработку и отладку.

Процесс проектирования будет включать несколько основных этапов: анализ требований, выбор структуры данных, создание алгоритмов для работы с файлами, оптимизация производительности и надежности, а также разработка интерфейсов взаимодействия с пользователем. Кроме того, особое внимание уделяется вопросам масштабируемости и расширяемости системы в будущем, так как требования к хранению данных могут меняться в зависимости от масштаба проекта и числа пользователей.

### **Архитектура системы**

Архитектура системы является основой для организации всех её компонентов. В проекте «Простая файловая система» структура будет достаточно простой, но с возможностью расширения и модификации в будущем. Архитектура будет включать несколько ключевых компонентов, каждый из которых выполняет свою специфическую роль. Ниже представлены основные компоненты, которые будут реализованы в рамках данного проекта.

#### **Основные компоненты системы**

1. Таблица метаданных (или индексная таблица). Каждый файл и директория имеют соответствующую запись в таблице метаданных, где хранится информация о файле: его имя, размер, дата создания, список блоков данных, а также другие параметры, такие как права доступа и время последнего изменения. Таблица метаданных будет использоваться для быстрого поиска информации о файлах и каталогах, а также для управления свободным пространством и операциями с данными.

2. Блоки данных. Блоки данных представляют собой наименьшие единицы хранения информации в системе. Каждый файл будет разбит на один или несколько блоков фиксированного размера. Например, если размер блока составляет 512 байт, то файл размером 1024 байта будет занимать два блока. Размер блока является важным параметром, который влияет на производительность системы, поскольку слишком маленький размер блока может привести к излишним операциям с маленькими частями данных, а слишком большой блок приведет к неэффективному использованию памяти. Блоки данных могут быть расположены не подряд, в случае если место на диске фрагментировано. Алгоритмы записи и чтения данных должны эффективно управлять фрагментацией, чтобы обеспечить стабильную производительность.

3. Механизм работы с каталогами. Каталоги будут представлять собой специальные файлы, которые содержат записи о других файлах и каталогах. Каждый каталог будет хранить пары "имя файла – ссылка на файл", что позволяет эффективно работать с иерархией файловой системы. Каждый файл будет хранить в своих метаданных ссылку на каталог, в котором он находится. Таким образом, структура каталогов будет организована в виде древовидной иерархии.

4. Интерфейс взаимодействия с пользователем. Для взаимодействия с файловой системой будет разработан простой пользовательский интерфейс. Он может быть реализован через командную строку, где пользователь вводит команды для работы с файлами и каталогами, такими как создание, удаление, перемещение, копирование файлов и т.д.

#### **Структура данных и хранилище**

В основе структуры данных лежат несколько ключевых объектов: файлы, каталоги и блоки данных. Эти объекты будут связаны друг с другом через таблицу метаданных и механизмы адресации блоков данных.

1. Файлы. Каждый файл в системе будет иметь соответствующую запись в таблице метаданных. Для каждого файла хранятся следующие параметры:

– имя файла – строка, представляющая название файла;

– размер файла – количество данных в файле;

– список блоков данных – указатели на блоки памяти, которые содержат данные файла;

2. Каталоги. Каталог в этой системе будет представлять собой также файл, который содержит записи о других файлах или подкаталогах. Каждая запись будет содержать: имя файла или каталога, Ссылка на файл или подкаталог, где хранятся его метаданные. Таким образом, каталоги будут представлять собой иерархию файлов и подкаталогов, где каждый каталог может содержать множество файлов и подкаталогов.

3. Блоки данных. Данные в системе будут храниться в блоках фиксированного размера. Каждый блок будет содержать часть данных какого-либо файла. Размер блока может быть настроен, например, 512 байт или 1 КБ. Размер блока влияет на производительность системы, так как маленькие блоки могут приводить к большому количеству операций с диском, в то время как слишком большие блоки могут привести к неэффективному использованию пространства.

### **Алгоритмы работы с файловой системой**

Главной задачей при проектировании файловой системы является создание эффективных алгоритмов для записи, чтения, удаления и модификации данных. Каждый из этих процессов должен быть продуман таким образом, чтобы минимизировать потери производительности и обеспечить высокую скорость работы.

#### **Алгоритм записи файла**

Запись нового файла в файловую систему будет осуществляться в несколько шагов:

1. Поиск свободных блоков. Перед записью файла необходимо найти свободные блоки в системе, которые будут использоваться для хранения данных файла. Это может быть реализовано через использование карты свободных блоков или список свободных блоков. В случае фрагментации данных, блоки могут быть не последовательными, и система должна эффективно управлять этими разрозненными частями.

2. Запись метаданных. После того как свободные блоки найдены, создается запись о файле в таблице метаданных. Эта запись включает имя файла, его размер и ссылки на блоки, в которых будет храниться его содержимое.

3. Запись данных. Данные файла записываются в выбранные блоки. Если файл большой, система может разбить его на несколько блоков, которые могут быть расположены не подряд. В этом случае в таблице метаданных будет храниться информация о каждом блоке.

4. Обновление таблицы метаданных. После записи данных, информация в таблице метаданных обновляется, фиксируя окончательный размер файла и физические блоки, на которых он хранится.

#### **Алгоритм чтения файла**

Процесс чтения файла также имеет несколько этапов:

1. Поиск записи в таблице метаданных. Когда пользователь запрашивает файл, система сначала ищет запись о файле в таблице метаданных. В записи хранятся все данные о файле, включая список блоков, которые содержат его данные. Если файл не найдет, пользователь будет информирован об этом.

2. Чтение данных. После того как найдено местоположение файла, система извлекает данные из блоков, на которые ссылается запись в таблице метаданных.

3. Обработка ошибок. Извлеченные данные передаются пользователю через интерфейс. Если файл большой и состоит из нескольких блоков, данные могут передаваться по частям.

#### **Алгоритм удаления файла**

При удалении файла система должна выполнить несколько действий:

1. Запись о файле удаляется из таблицы метаданных. Это освобождает место в таблице для хранения информации о других файлах. Освобождение блоков данных. Все блоки, занятые файлом, помечаются как свободные и могут быть использованы для хранения других файлов.

2. Освобождение блоков данных: Все блоки, которые были использованы для хранения данных файла, помечаются как свободные. Они могут быть использованы для записи других данных.

3. Очистка данных. Полный алгоритм перед удалением файла предусматривает перезапись блоков данных случайными данными, чтобы предотвратить восстановление удаленной информации.

#### **Алгоритм создания директории**

Создание новой директории в файловой системе осуществляется в несколько этапов:

1. Проверка существования. Перед созданием необходимо убедиться, что директория с таким именем не существует в текущем каталоге. Также проверяется корректность имени (отсутствие запрещённых символов, допустимая длина).

2. Поиск свободного дескриптора. Находится свободный слот в таблице метаданных (или inode-таблице), который будет представлять новую директорию. Как и в случае с файлами, этот слот содержит имя, атрибуты и список блоков (пока пустой или содержащий структуру подкаталога).

3. Выделение блока под структуру директории. Система ищет свободный блок (или несколько блоков) для хранения содержимого новой директории, которое представляет собой список файлов и поддиректорий внутри неё (обычно в виде массива записей dir\_entry).

4. Инициализация директории. В блоке директории создаются специальные записи.

5. "." – ссылка на саму директорию.

6. ".." – ссылка на родительскую директорию. Это необходимо для корректной навигации и реализации команд cd, ls, pwd.

7. Добавление записи в родительскую директорию. В родительской директории создаётся запись о новой директории – аналогично добавлению файла. Эта запись содержит имя новой директории и ссылку на её дескриптор.

8. Обновление метаданных. Обновляется таблица метаданных: увеличивается размер родительской директории, фиксируется факт создания новой директории, устанавливаются временные метки.

### **Выбор технологий и инструментов**

Для реализации файловой системы будет использован язык программирования C. C – это низкоуровневый язык, который позволяет эффективно управлять памятью, что критично для создания файловых систем. Также C дает доступ к системным вызовам, таким как открытие, чтение, запись и закрытие файлов, что позволяет гибко взаимодействовать с операционной системой.

Важным аспектом является управление памятью. Для этого будут использоваться метаданные файловой системы, которые в зависимости от действия, выбранного пользователем, будут использоваться для изменения содержимого диска. Важно, чтобы система корректно управляла динамическим выделением памяти, избегала утечек и обеспечивала стабильную работу.

#### **Управление памятью**

Управление памятью будет реализовано через функции fseek, fwrite и чтение данных через fread. Это позволяет сразу записывать данные в правильное место на диске без чрезмерного использования буферов и оперативной памяти, что повышает безопасность хранения информации. Обеспечивается это тем, что между чтением и записью новой информации на диск проходит минимальное время.

Также важным аспектом оптимизации работы является кеширование метаданных. Кеширование позволяет значительно сократить операции чтения и записи с диска, заменяя их на чтения данных из оперативной памяти, которые в свою очередь используются при любом действии внутри файловой системы.

#### **Ввод/вывод**

Для ввода/вывода будет использоваться стандартные функции C, такие как open, fgets, printf, close для работы с файлами и блоками данных. Это позволяет максимально эффективно управлять данными и обеспечивать совместимость с большинством операционных систем.

### **Обеспечение надежности и производительности**

#### **Надежность**

Для обеспечения надежности будут реализованы следующие механизмы:

1. Проверка ошибок. Каждый этап работы с файловой системой (запись, чтение, удаление) будет сопровождаться проверками на ошибки, что позволит быстро обнаружить проблемы и предотвратить потерю данных.

2. Резервное копирование. В будущем можно добавить поддержку автоматического резервного копирования, что позволит избежать потери данных.

#### **Производительность**

Для повышения производительности файловая система будет оптимизирована по следующим направлениям:

1. Использование кэширования. Кэширование данных в памяти позволяет ускорить доступ к часто используемым файлам и уменьшить количество операций с диском.

2. Оптимизация фрагментации. Алгоритмы записи данных будут оптимизированы для минимизации фрагментации, что позволяет улучшить производительность и ускорить доступ к файлам.

### **Заключение**

Проектирование простой файловой системы (SFS) в пространстве пользователя предполагает создание эффективной и надежной структуры для хранения данных. Система будет обеспечивать работу с файлами и каталогами, реализуя все необходимые операции с данными, такие как создание, удаление, перемещение и копирование. Особое внимание уделено вопросам производительности, надежности и безопасности, а также возможностям масштабирования системы в будущем.

## **ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ**

### **Назначение**

Разрабатываемая система представляет собой простую файловую систему (SFS), реализованную в пространстве пользовательского процесса на языке C. Система предоставляет базовые операции для управления файлами и директориями, включая создание, удаление, перемещение, чтение и запись, а также функции монтирования и форматирования виртуального диска.

### **Общая структура**

Файловая система состоит из следующих логических компонентов:

– superblock – содержит глобальную информацию о файловой системе, включая общее количество и количество свободных inodes и блоков;

– inode-таблица – хранит метаданные для каждого объекта (файла или директории): флаги, размер, ссылки на блоки данных и связи с родительскими директориями;

– каталог файлов (Directory table) – содержит отображение между именами файлов и индексами соответствующих inodes;

– блоки данных – непосредственно хранят содержимое файлов;

– файловый дескриптор disk – абстракция виртуального устройства хранения, реализованного в виде файла на хостовой файловой системе.

### **Superblock**

Структура Superblock содержит метаинформацию, необходимую для функционирования пользовательской файловой системы. Она располагается в начале дискового пространства и используется при инициализации и монтировании. Superblock определяет размеры и текущее состояние системы, включая распределение блоков и inode.

Назначение и описание полей:

– total\_blocks – общее количество доступных блоков в файловой системе. Это фиксированное значение, определяющее объем хранилища;

– free\_blocks – количество свободных (неиспользуемых) блоков, готовых к размещению данных;

– total\_inodes – общее количество записей inode в файловой системе. Каждый inode соответствует одному файлу или каталогу;

– free\_inodes – текущее количество неиспользуемых inode. Значение уменьшается при создании новых файлов и увеличивается при их удалении;

– block\_bitmap – битовая карта занятости блоков, где каждый бит указывает, занят блок (1) или свободен (0). Используется для эффективного управления пространством.

### **Inode**

Структура Inode описывает метаинформацию о файле или каталоге. Она используется для хранения информации о размере, типе, количестве блоков и их расположении, а также для связи с соответствующими данными на диске. Каждый файл или каталог в файловой системе имеет свой уникальный inode.

Назначение и описание полей:

– is\_used – флаг, который указывает, занят ли данный inode. Значение 1 означает, что inode используется, 0 – что он свободен;

– is\_directory – флаг, указывающий, является ли объект директорией. Если значение равно 1, то объект – это каталог; если 0 – это файл;

– directory\_inode\_index – используется только для каталогов. Хранит индекс inode родительской директории для связи с другими каталогами;

– filename – имя файла или каталога. Это строка символов, длина которой ограничена константой MAX\_FILENAME\_LENGTH;

– size – размер файла в байтах. Это значение показывает полный размер данных, хранящихся в файле;

– block\_count – количество блоков данных, которые занимают данный файл. В одном inode может храниться до 16 блоков;

– blocks – массив, хранящий индексы блоков данных, на которых хранится содержимое файла. Максимальное количество блоков на файл – 16.

### **DirectoryEntry**

Структура DirectoryEntry используется для хранения информации о файлах и каталогах в директории. Каждый элемент DirectoryEntry представляет собой запись в каталоге, которая связывает имя файла с соответствующим inode.

Назначение и описание полей:

– inode\_index – индекс inode, который соответствует файлу или каталогу. Это позволяет связать имя файла с его метаинформацией в структуре inode;

– filename – имя файла или каталога. Это строка символов, длина которой ограничена константой MAX\_FILENAME\_LENGTH.

### **Блоки данных**

Являются основным хранилищем содержимого файлов. Каждый файл может занимать один или несколько блоков в зависимости от своего размера. Индексация осуществляется через массив blocks в inode.

Блоки данных могут использоваться для:

– хранения содержимого обычных файлов;

– организации структуры директорий (если они хранятся в виде отдельных блоков);

– будущего расширения (например, хранения косвенных ссылок или атрибутов).

### **Основные функции**

#### **Форматирование диска**

Функция sfs\_mkfs() инициализирует файловую систему, создаавя структуру Superblock, inode-таблицу и каталог. Все блоки помечаются как свободные. Данная функция вызывается внутри функции монтирования файловой системы в случае, если при монтировании возникли проьлемы с виртуальным диском такие как отсутствие последнего или нарушение его целостности. Функция создает виртуальный диск, который будет использоваться в дальнейшем.

#### **Монтирование и размонтирование**

Функции sfs\_mount() и sfs\_umount() обеспечивают открытие и сохранение состояния файловой системы. При монтировании происходит загрузка Superblock, inode-таблицы и директории в оперативную память. Загрузка метаданных в оперативную память необходима для ускорения работы файловой системы путем уменьшения количества обращений к диску для чтении информации. Так же при монтировании диск единажды открывается для чтения и записи информации, если виртуальный диск не существует вызывается функция которая его создаст. При размонтировании файловой системы все данные впоследний раз записываются на диск и закрывают его.

#### **Помощь в использовании**

Функция help() при вводе ключевого слова help выводит в консоль информацию о всех возможных командах, которые обрабатываются при использовании данной файловой системы. Так же для каждой команды указывается формат ввода данных для корректной работы этой команды.

#### **Создания домашнего каталога**

Процесс создания домашнего каталога представляет собой этап инициализации файловой системы, в ходе которого формируется начальная (корневая) директория. Она выполняет ключевую роль в управлении пользовательскими данными, обеспечивая структуру хранения и доступ к файлам.

#### **Рекурсивное удаление каталога**

Рекурсивное удаление каталога — это процесс удаления не только самого каталога, но и всех файлов и вложенных подкаталогов, содержащихся в нем. Этот метод применяется в случаях, когда необходимо полностью удалить файловую структуру без необходимости предварительно очищать ее вручную.

#### **Создание директорий**

Функция sfs\_create\_dir() позволяет создать поддиректорию в текущей или заданной директории. Реализована проверка на дублирование имен, выделение свободного inode и блока данных. Директория отображается в таблице directory и получает уникальный inode.

#### **Перемещение файлов**

Функция sfs\_move\_to\_dir() позволяет переместить файл или поддиректорию в другую директорию. При этом обновляется поле directory\_inode\_index в inode, отражающее родительскую директорию, и соответствующая запись в каталоге.

#### **Удаление директорий**

Функция sfs\_delete\_dir() удаляет пустые директории (кроме корня и текущей). Перед удалением осуществляется проверка на принадлежность к текущей директории и отсутствие вложенных файлов.

Если оба условия соблюдены, функция удаляет каталог, обновляет соответствующие метаданные в файловой системе и освобождает ресурсы, связанные с ним. Данный механизм предотвращает случайное удаление структур, содержащих важные данные, и минимизирует вероятность нарушения целостности файловой системы.

#### **Смена директории**

Функция sfs\_cd() позволяет перейти в другую директорию, изменяя глобальный указатель current\_directory\_inode. Для построения строкового представления пути используется функция build\_path\_from\_inode().

#### **Просмотр содержимого директории**

Функция sfs\_ls\_dir() отображает список всех объектов, находящихся в заданной или текущей директории. Выводится имя, тип объекта (файл или директория) и для файла размер в байтах. Для поиска содержимого необходимой директории используются метаданные.

#### **Создание файла**

Создает новый файл с заданным именем в текущей директории.

Описание работы:

1. Проверяет, существует ли уже файл с таким именем.

2. Находит свободный inode в таблице inode.

3. Обновляет структуру inode: устанавливает флаг is\_used, задает имя файла, обнуляет размер и количество блоков.

4. Добавляет новую запись в каталог (DirectoryEntry), связывая имя файла с номером его inode.

5. Обновляет суперблок (уменьшает количество свободных inode и обновляет таблицы).

#### **Запись данных в файл**

Записывает строку данных в файл с заданным именем.

Описание работы:

1. Ищет файл в текущем каталоге.

2. Проверяет, есть ли доступные блоки для хранения данных.

3. Разбивает данные на блоки и записывает их в свободные блоки данных.

4. Обновляет inode файла: устанавливает размер, количество блоков, сохраняет номера блоков в массиве blocks.

5. Обновляет битовую карту блоков и суперблок.

#### **Чтение данных из файла**

Читает и выводит содержимое файла с указанным именем:

1. Ищет файл в текущей директории.

2. Получает inode соответствующего файла.

3. Читает блоки данных, указанные в массиве blocks inode.

4. Выводит содержимое на стандартный вывод (обычно в консоль).

#### **Удаление файла**

Удаляет файл с заданным именем из файловой системы.

Описание работы:

1. Ищет файл в текущем каталоге.

2. Освобождает все блоки данных, указанные в inode файла, обновляя битовую карту блоков.

3. Освобождает inode (сбрасывает флаг is\_used).

4. Удаляет запись из текущего каталога (DirectoryEntry).

5. Обновляет суперблок.

#### **Разрешение пути**

Функция resolve\_path\_to\_inode() реализует логику разрешения путей к inode, включая поддержку относительных и абсолютных путей, специальных элементов, а также пошаговый анализ вложенности. Необходима для обработки специальных символов в пути к файлу и корректного нахождения необходимого инода.

### **Обработка ошибок**

Во всех функциях предусмотрены проверки на:

– корректность входных данных;

– наличие необходимых ресурсов (inode, блоков);

– целостность логической структуры (например, при удалении директорий);

– предотвращение запрещённых действий (удаление корня, текущей директории).

Сообщения об ошибках выводятся в стандартный поток с пояснениями.

## **РАЗРАБОТКА ПРОГРАММНЫХ МОДУЛЕЙ**

### **Разработка структурной схемы**

Для визуализации процесса взаимодействия различных логических блоков файловой системы была разработана структурная схема программы. Структурная схема представлена в приложении А.

### **Разработка диаграммы последовательностей**

Для визуализации процесса взаимодействия компонентов при монтировании и создании файла была разработана диаграмма последовательностей. Разработанная диаграмма представлена в приложении Б.

### **Разработка схем алгоритмов**

Схема метода create\_home\_directory()показана в приложении В.

Схема метода sfs\_mkfs(const char \*diskname) показана в приложении Г.

### **Описание алгоритма** sfs\_ls\_dir(const char \*dirname) **по шагам**

1. Начало.

2. Инициализация переменных target\_inode, basename[], parent\_inode.

3. Если dirname принимает значение NULL или её длина равна 0, присваивается переменной target\_inode значение current\_directory\_inode переход к шагу 5. Иначе выполняется шаг 4.

4. Присваивается переменной target\_inode значение соответствующее иноду по указанному пути, найденное функцией resolve\_path\_to\_inode().

5. Проверяется значение target\_inode, если оно равно -1, выводится сообщение об ошибке пути и выполняется выход из функции.

6. Если target\_inode не равно нулю и значение is\_directory для текущего пути установлено в 0, выводится сообщение, что это не директория и выполняется выход из программы.

7. Инициализация переменной dir\_path[].

8. Если значение target\_inode равно нулю, в dir\_path[] записывается путь корневой директории, то есть символ /.

9. В противном случае вызывается функция для получения пути по номеру инода.

10. Выводится сообщение о выводе информации о содержимом директории.

11. Инициализация переменной empty, записывается в неё значение 1.

12. Начало цикла А, продолжается до MAX\_FILES.

13. Если i-тый инод директории равен -1, пропускается элемент, переход к следующей итерации цикла.

14. Если в таблице инодов запись принадлежит нужной директории, выводится имя этой записи.

15. Конец цикла А.

16. Проверяется в какое значение установлен логиечский флаг-показатель директории, если 1 выводится надпись о том, что файл – директория, в проивном случае, файл и его размер.

17. Выводится символ перехода на следующую строку.

18. Сохраняется в переменную empty значение 0.

19. Проверка значения переменной empty, если 1 выводится сообщение о том, что директория пуста.

20. Конец.

### **Описание алгоритма** sfs\_read(const char \*filename) **по шагам**

1. Начало.

2. Начало цикла А, продолжается до MAX\_FILES.

3. Если функция сравнения переданного имени файла и имени файла на диске вернула 0 переход к шагу 4.

4. Инициализируется переменная inode адресом инода в котором хранится данный файл.

5. Инициализируются переменные data\_read нулем, size размером инода в котором храниться файл, buffer[size + 1].

6. Начало цикла Б, продолжается до inode->block\_count.

7. Инициализируется переменная block\_size с помощью тернарного оператора.

8. Перемещается позицию для чтения на диске по адресу inode->blocks[j] \* BLOCK\_SIZE.

9. Читается из диска объем информации равный размеру блока.

10. К переменной data\_read прибавляется размер блока.

11. Если data\_read == size выход из цикла Б.

12. Конец цикла Б.

13. В конец буфера по позиции size пишется символ конца строки.

14. Вывод сообщения и прочитанных данных, переход к шагу 15.

15. Конец цикла А.

16. Вывод сообщения, что файл не найден.

17. Конец.

## **ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ**

### **Цель испытаний**

Целью испытаний является проверка корректности работы разработанной простой файловой системы SFS (Simple File System) на языке C. Испытания должны подтвердить, что система обеспечивает основные функции работы с файлами и директориями, а также корректно сохраняет и восстанавливает данные после монтирования и размонтирования.

* 1. **Объект испытаний**

Объектом испытаний является программный модуль файловой системы SFS, реализующий следующие функции: sfs\_mount, sfs\_mkfs, help, sfs\_umount, sfs\_create, sfs\_write, sfs\_read, sfs\_delete, sfs\_create\_dir, sfs\_ls, sfs\_delete\_dir, sfs\_delete\_dir\_recursive, sfs\_move\_to\_dir, sfs\_cd.

* 1. **Методика испытаний**

Испытания проводятся вручную, с использованием тестовых сценариев, охватывающих все ключевые функции файловой системы. Результаты отображаются в терминале. Контроль корректности осуществляется путём анализа вывода функций и содержимого виртуального диска после операций.

Для каждой функции создаётся соответствующий тестовый сценарий с пошаговым выполнением команд и проверкой состояния системы.

* 1. **Результаты испытаний**

Функция sfs\_mount() автоматически монтирует файловую при запуске приложения, если диск файловой системы не найден для его создания вызывается функция sfs\_mkfs(), для наглядности выводятся комментарии о процессе монтирования, функция help информирует о способах взаимодействия с файловой системой (см. рисунок 5.1).

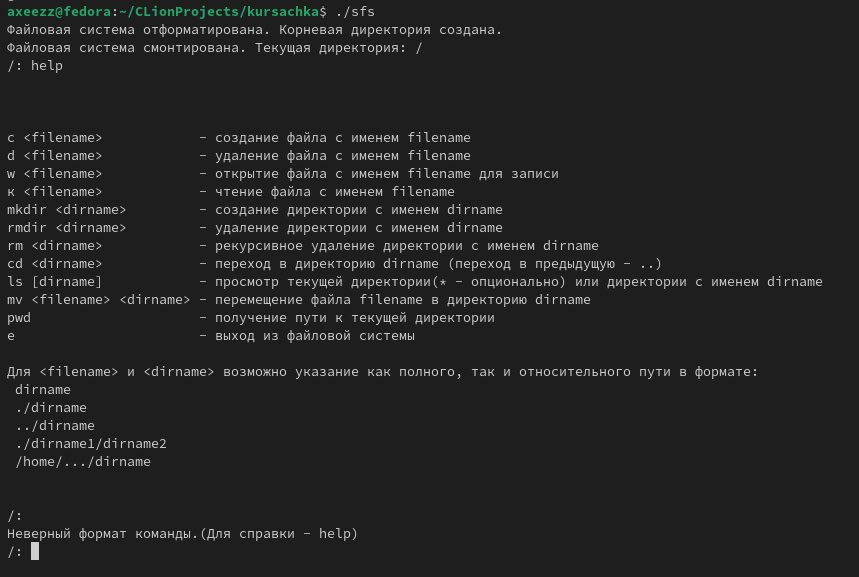


Рисунок 5.1 – Запуск файловой системы и функции help().

При повторном запуске файловой системы вызывается только функция sfs\_mount(). Файловая позволяет начать работу сразу же, так функция sfs\_ls() выводит содержимое корневой директории (см. рисунок 5.2).

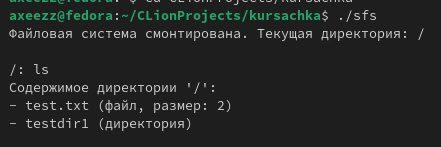


Рисунок 5.2 – Повторный запуск файловой системы и работа функции sfs\_ls().

В следующем сценарии тестирования показана работа функций удаления файла, перехода между директориями, удаления директории и рекурсивного удаления директории (см. рисунок 5.3).

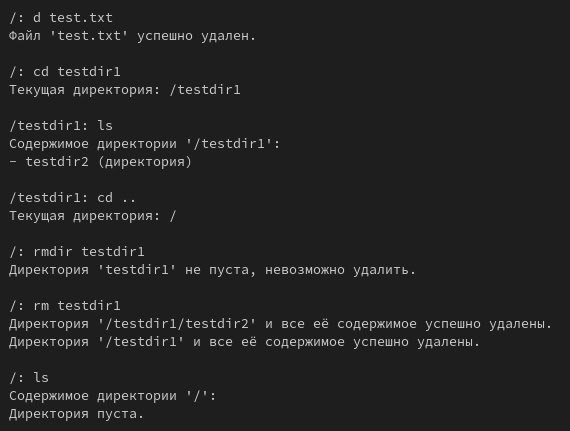


Рисунок 5.3 – Сценарий для тестирования №1.

Второй сценарий показывает работу функции создания директорий (см. рисунок 5.4).

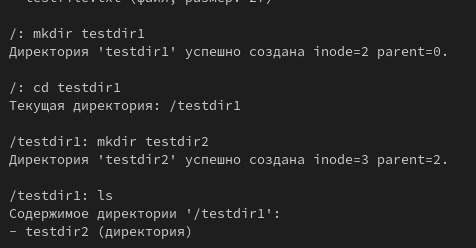


Рисунок 5.4 – Сценарий для тестирования №2.

Следующий сценарий тестирования показывает работу функция создания файла, записи в файл данных, чтения данных из файла (см. рисунок 5.5).

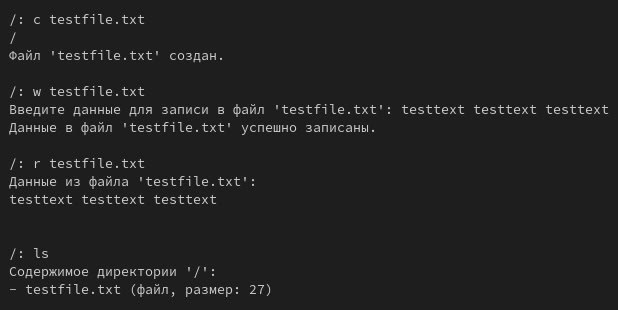


Рисунок 5.5 – Сценарий для тестирования №3.

Далее рассматривается функция перемещения файлов между директориями (см. рисунок 5.6).

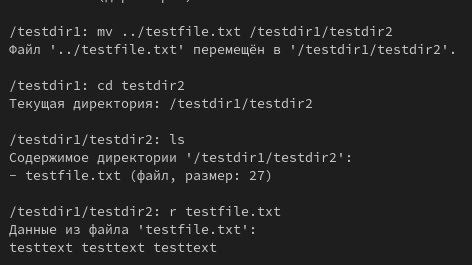


Рисунок 5.6 – Сценарий для тестирования №4.

Все указанные функции показали корректную работу в предоставленных сценариях. Таким образом можно сделать вывод о правильной работе файловой системы.

## **РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ**

### **Назначение программы**

Программа представляет собой простую файловую систему (SFS – Simple File System), реализованную на языке C, предназначенную для работы с виртуальным диском. Пользователь может выполнять базовые операции с файлами и директориями в рамках смонтированной файловой системы.

### **Требования к окружению**

– Операционная система: Linux;

– компилятор: GCC;

– объём оперативной памяти: не менее 128 МБ;

– доступ к терминалу/командной строке.

### **Установка и запуск**

1. Перейдите в директорию с проектом.

2. Скомпилируйте проект, команда make в командной строке.

3. Запустите исполняемый файл, команда ./sfs

4. Для знакомства с возможностями файловой системы и форматом команд используйте команду help (см. рисунок 6.1).

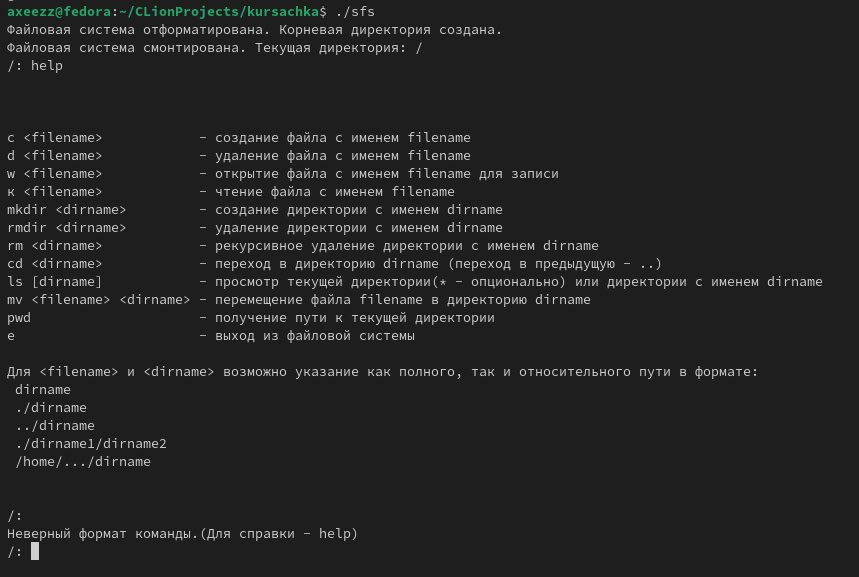


Рисунок 6.1 – Команда help.

### **Основные команды пользователя**

После запуска пользователь взаимодействует с системой через ввод команд в терминале. При работе с файловой системой следует соблюдать формат ввода команд (см. рисунок 6.1). При некорректном вводе команд будет выведено уведомление о некорректной команде.

### **Особенности использования**

– При первом монтировании создаётся корневая директория home;

– все файлы и директории хранятся во внутренней структуре и сохраняются на диск при umount;

– поддерживаются только одноуровневые имена (без / в имени);

– максимальный размер файла ограничен заранее заданным количеством блоков;

– максимальная длина имени файла/директории: 28 символов.

### **Возможные ошибки и их решения**

Возможные сообщения об ошибках и их решения представлены в таблице 1.

Таблица 1 – Сообщениях об ошибках.

|  |  |  |
| --- | --- | --- |
| **Сообщение об ошибке** | **Причина** | **Решение** |
| Файл уже существует | Попытка создать файл с существующим именем | Используйте уникальное имя |
| Файл не найден | Указанный файл отсутствует | Проверьте правильность имени и пути к файлу |
| Нет свободных inode или блоков | Файловая система переполнена | Удалите ненужные файлы |
| Неверная команда | Введена неизвестная команда | Ознакомьтесь с допустимыми командами |

## **ЗАКЛЮЧЕНИЕ**

В процессе выполнения курсовой работы была спроектирована и реализована программная модель простой файловой системы SFS (Simple File Systemя). Разработка охватывала как структуру хранения данных, так и реализацию основных команд, что позволило создать функциональный и расширяемый прототип файловой системы.

На этапе проектирования были определены ключевые логические структуры: суперблок, таблица inode, таблица директорий и блоки хранения данных. Особое внимание уделялось вопросам целостности данных, поиску свободных inode и блоков, а также поддержке базовой иерархии директорий, включая автоматическое создание корневой директории и навигацию по относительным и абсолютным путям.

На этапе реализации была создана система, поддерживающая:

– монтирование и размонтирование файловой системы с сохранением состояния на виртуальном диске;

– создание и удаление файлов;

– чтение и запись в файлы;

– навигацию по директориям (cd, pwd);

– просмотр содержимого текущей директории (ls);

– создание директорий (mkdir) и удаление пустых директорий (rm);

– отображение структуры путей и автоматическое построение абсолютного пути на основе inode текущей директории.

Была разработана последовательная архитектура, позволяющая гарантировать устойчивую работу системы даже в случае частичных сбоев. При этом использовались стандартные средства языка C для работы с файлами, буферами, строками и структурами. Для сохранения данных и их восстановления после перезапуска реализована система сериализации всех ключевых структур в файл-диск.

В качестве дополнения к программной реализации была разработана диаграмма последовательностей, отражающая поведение системы при монтировании, создании, удалении и чтении файлов.

В процессе испытаний программный комплекс продемонстрировал устойчивую работу во всех предусмотренных сценариях.

Возможными путями развития программного обеспечения является добавление проверки целостности файловой системы после сбоев. Подробное журналирование действий для возможности восстановления данных или отмены действий.

Таким образом, цели, поставленные для данной курсовой работы, были достигнуты. Разработанная система представляет собой законченный, работоспособный программный продукт, иллюстрирующий ключевые концепции построения собственной файловой системы.

## **СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ**

[1] Community [Электронный ресурс]. – 2017 – Режим доступа: <https://timeweb.com/ru/community/articles/tipy-faylovyh-sistem-ih-prednaznachenie-i-otlichiya> – Дата доступа – 29.03.2025.

[2] iRunner Wiki [Электронный ресурс]. – 2019 – Режим доступа: <https://acm.bsu.by/wiki/Unix2019b/Файловая_система#FUSE> – Дата доступа – 29.03.2025.

[3] iRunner Wiki [Электронный ресурс]. – 2019 – Режим доступа: https://acm.bsu.by/wiki/Unix2019b/Файловая\_система – Дата доступа – 30.03.2025.

Стахнов А.А. Linux в подлиннике: наиболее полное руководство / А. А. Стахнов – СПб.: БХВ-Петербург, 2011. – 752с.

## **ПРИЛОЖЕНИЕ А**

(обязательное)

Структурная схема

## **ПРИЛОЖЕНИЕ Б**

(обязательное)

Диаграмма последовательностей

## **ПРИЛОЖЕНИЕ В**

(обязательное)

Схема метода create\_home\_directory()

## **ПРИЛОЖЕНИЕ Г**

(обязательное)

Схема метода sfs\_mkfs(const char \*diskname)

## **ПРИЛОЖЕНИЕ Д**

(обязательное)

Исходный код программы

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

**Файл main.c**

1. #include <stdio.h>

2. #include <stdlib.h>

3. #include <string.h>

4. #include "sfs.h"

5.

6. int main() {

7. const char \*diskname = "virtual\_disk.img";

8. char command[256];

9. char data[BLOCK\_SIZE];

10.

11. sfs\_mkfs(diskname);

12. sfs\_mount(diskname);

13.

14. while (1) {

15. printf("\n");

16. sfs\_pwdm();

17. fgets(command, sizeof(command), stdin);

18. command[strcspn(command, "\n")] = 0; // Убираем символ новой строки

19.

20. char \*cmd = strtok(command, " "); // Получаем команду (например, c)

21. char \*arg = strtok(NULL, " "); // Получаем аргумент (например, kaka21)

22.

23. if (cmd == NULL) {

24. printf("Неверный формат команды.(Для справки - help)\n");

25. continue;

26. }

27.

28. if (\*cmd != 'l' && \*cmd != 'e' && strcmp(cmd, "pwd") && strcmp(cmd, "help") && arg == NULL) {

29. printf("Неверный формат команды.(Для справки - help)\n");

30. continue;

31. }

32.

33. if (strcmp(cmd, "c") == 0) {

34. sfs\_create(arg);

35. } else if (strcmp(cmd, "d") == 0) {

36. sfs\_delete(arg);

37. } else if (strcmp(cmd, "w") == 0) {

38. printf("Введите данные для записи в файл '%s': ", arg);

39. fgets(data, sizeof(data), stdin);

40. sfs\_write(arg, data);

41. } else if (strcmp(cmd, "r") == 0) {

42. sfs\_read(arg);

43. } else if (strcmp(cmd, "e") == 0) {

44. sfs\_umount();

45. break;

46. } else if (strcmp(cmd, "mkdir") == 0) {

47. sfs\_create\_dir(arg);

48. } else if (strcmp(cmd, "mv") == 0) {

49. char \*dirname = strtok(NULL, " ");

50. sfs\_move\_to\_dir(arg, dirname);

51. } else if (strcmp(cmd, "ls") == 0) {

52. sfs\_ls\_dir(arg);

53. } else if (strcmp(cmd, "rmdir") == 0) {

54. sfs\_delete\_dir(arg);

55. } else if (strcmp(cmd, "cd") == 0) {

56. if (arg != NULL) {

57. sfs\_cd(arg);

58. } else {

59. printf("Необходимо указать директорию для перехода.\n");

60. }

61. } else if (strcmp(cmd, "pwd") == 0) {

62. sfs\_pwd();

63. } else if (strcmp(cmd, "rm") == 0) {

64. sfs\_delete\_dir\_recursive(arg);

65. } else if (strcmp(cmd, "help") == 0) {

66. help ();

67. } else {

68. printf("Неизвестная команда.(Для справки - help)\n");

69. }

70. }

71.

72. return 0;

73. }

**Файл sfs.c**

1 #include "sfs.h"

2 #include <stdio.h>

3 #include <string.h>

4

5 // Глобальные переменные

6 FILE \*disk = NULL;

7 Superblock superblock;

8 Inode inode\_table[MAX\_FILES];

9 DirectoryEntry directory[MAX\_FILES];

10 int current\_directory\_inode = 0;

11 char current\_directory[MAX\_FILENAME\_LENGTH] = "";

12

13 // Основные функции файловой системы

14

15 void sfs\_mkfs(const char \*diskname) {

16 FILE \*test = fopen(diskname, "rb");

17 if (test) {

18 fclose(test);

19 printf("Файл '%s' уже существует. Используйте mount для доступа.\n", diskname);

20 return;

21 }

22

23 disk = fopen(diskname, "wb+");

24 if (!disk) {

25 printf("Не удалось создать файл диска.\n");

26 return;

27 }

28

29 // Initialize superblock

30 superblock.total\_blocks = MAX\_BLOCKS;

31 superblock.free\_blocks = MAX\_BLOCKS;

32 superblock.total\_inodes = MAX\_FILES;

33 superblock.free\_inodes = MAX\_FILES - 1;

34 memset(superblock.block\_bitmap, 0x00, sizeof(superblock.block\_bitmap));

35

36 // Initialize inodes and directory

37 for (int i = 0; i < MAX\_FILES; i++) {

38 inode\_table[i].is\_used = 0;

39 inode\_table[i].is\_directory = 0;

40 inode\_table[i].directory\_inode\_index = -1;

41 directory[i].inode\_index = -1;

42 memset(directory[i].filename, 0, MAX\_FILENAME\_LENGTH);

43 }

44

45 // Create root directory

46 inode\_table[0].is\_used = 1;

47 inode\_table[0].is\_directory = 1;

48 inode\_table[0].directory\_inode\_index = -1;

49 strncpy(inode\_table[0].filename, "/", MAX\_FILENAME\_LENGTH);

50 directory[0].inode\_index = 0;

51 strncpy(directory[0].filename, "/", MAX\_FILENAME\_LENGTH);

52

53 create\_home\_directory();

54

55 // Write to disk

56 fseek(disk, 0, SEEK\_SET);

57 fwrite(&superblock, sizeof(Superblock), 1, disk);

58 fwrite(inode\_table, sizeof(Inode), MAX\_FILES, disk);

59 fwrite(directory, sizeof(DirectoryEntry), MAX\_FILES, disk);

60 fflush(disk);

61

62 printf("Файловая система отформатирована. Корневая директория создана.\n");

63 }

64

65 void sfs\_mount(const char \*diskname) {

66 disk = fopen(diskname, "r+b");

67 if (!disk) {

68 printf("Файл диска не существует. Создать новую файловую систему? (y/n): ");

69 int answer = getchar();

70 while (getchar() != '\n'); // Clear input buffer

71

72 if (answer == 'y' || answer == 'Y') {

73 sfs\_mkfs(diskname);

74 disk = fopen(diskname, "r+b");

75 if (!disk) {

76 printf("Ошибка при создании файловой системы.\n");

77 return;

78 }

79 } else {

80 return;

81 }

82 }

83

84 // Check filesystem validity

85 if (!is\_valid\_filesystem(disk)) {

86 printf("Файл не содержит валидной ФС. Инициализировать? (y/n): ");

87 int answer = getchar();

88 while (getchar() != '\n');

89

90 if (answer == 'y' || answer == 'Y') {

91 fclose(disk);

92 sfs\_mkfs(diskname);

93 disk = fopen(diskname, "r+b");

94 if (!disk) {

95 printf("Ошибка при создании файловой системы.\n");

96 return;

97 }

98 } else {

99 fclose(disk);

100 disk = NULL;

101 return;

102 }

103 }

104

105 // Read structures

106 fseek(disk, 0, SEEK\_SET);

107 if (fread(&superblock, sizeof(Superblock), 1, disk) != 1) {

108 printf("Ошибка чтения суперблока.\n");

109 fclose(disk);

110 disk = NULL;

111 return;

112 }

113

114 fseek(disk, sizeof(Superblock), SEEK\_SET);

115 if (fread(inode\_table, sizeof(Inode), MAX\_FILES, disk) != MAX\_FILES) {

116 printf("Ошибка чтения таблицы inode.\n");

117 fclose(disk);

118 disk = NULL;

119 return;

120 }

121

122 fseek(disk, sizeof(Superblock) + sizeof(Inode) \* MAX\_FILES, SEEK\_SET);

123 if (fread(directory, sizeof(DirectoryEntry), MAX\_FILES, disk) != MAX\_FILES) {

124 printf("Ошибка чтения директории.\n");

125 fclose(disk);

126 disk = NULL;

127 return;

128 }

129

130 // Set current directory

131 current\_directory\_inode = 0;

132 strcpy(current\_directory, "/");

133

134 // Try to find /home

135 for (int i = 0; i < MAX\_FILES; i++) {

136 if (directory[i].inode\_index != -1 &&

137 strcmp(directory[i].filename, "home") == 0 &&

138 inode\_table[directory[i].inode\_index].is\_directory) {

139 current\_directory\_inode = directory[i].inode\_index;

140 strcpy(current\_directory, "/home");

141 break;

142 }

143 }

144

145 printf("Файловая система смонтирована. Текущая директория: %s\n", current\_directory);

146 }

147

148 void sfs\_umount() {

149 if (disk) {

150 // Save all changes

151 fseek(disk, 0, SEEK\_SET);

152 fwrite(&superblock, sizeof(Superblock), 1, disk);

153

154 fseek(disk, sizeof(Superblock), SEEK\_SET);

155 fwrite(inode\_table, sizeof(Inode), MAX\_FILES, disk);

156

157 fseek(disk, sizeof(Superblock) + sizeof(Inode) \* MAX\_FILES, SEEK\_SET);

158 fwrite(directory, sizeof(DirectoryEntry), MAX\_FILES, disk);

159

160 fflush(disk);

161 fclose(disk);

162 disk = NULL;

163

164 printf("Файловая система размонтирована. Все данные сохранены.\n");

165 }

166 }

167

168 // Вспомогательные функции

169 int find\_free\_inode() {

170 for (int i = 0; i < MAX\_FILES; i++) {

171 if (inode\_table[i].is\_used == 0) {

172 return i;

173 }

174 }

175 return -1;

176 }

177

178 int find\_free\_block() {

179 for (int i = 0; i < MAX\_BLOCKS; i++) {

180 if ((superblock.block\_bitmap[i / 8] & (1 << (i % 8))) == 0) {

181 return i;

182 }

183 }

184 return -1;

185 }

186

187 void allocate\_block(int block\_index) {

188 superblock.block\_bitmap[block\_index / 8] &= ~(1 << (block\_index % 8));

189 superblock.free\_blocks--;

190 }

191

192 void free\_block(int block\_index) {

193 superblock.block\_bitmap[block\_index / 8] |= (1 << (block\_index % 8));

194 superblock.free\_blocks++;

195 }

196

197 void print\_current\_directory() {

198 if (current\_directory\_inode == 0) {

199 printf("/\n");

200 return;

201 }

202 char path[MAX\_FILENAME\_LENGTH];

203 build\_path\_from\_inode(current\_directory\_inode, path, MAX\_FILENAME\_LENGTH);

204 printf("%s\n", path);

205 }

206

207 void build\_path\_from\_inode(int inode, char \*path, size\_t path\_size) {

208 if (inode == 0) {

209 strncpy(path, "/", path\_size);

210 return;

211 }

212 char parent\_path[MAX\_FILENAME\_LENGTH] = "";

213 int parent\_inode = inode\_table[inode].directory\_inode\_index;

214 if (parent\_inode != 0) {

215 build\_path\_from\_inode(parent\_inode, parent\_path, sizeof(parent\_path));

216 }

217 for (int i = 0; i < MAX\_FILES; i++) {

218 if (directory[i].inode\_index == inode) {

219 if (strcmp(parent\_path, "/") == 0) {

220 snprintf(path, path\_size, "/%s", directory[i].filename);

221 } else {

222 snprintf(path, path\_size, "%s/%s", parent\_path, directory[i].filename);

223 }

224 break;

225 }

226 }

227 }

228

229 void sfs\_pwd() {

230 printf("Текущая директория: %s\n", current\_directory);

231 }

232

233 void sfs\_pwdm() {

234 printf("%s: ", current\_directory);

235 }

236

237 int is\_valid\_filesystem(FILE \*f) {

238 if (!f) return 0;

239

240 // 1. Проверяем размер файла

241 fseek(f, 0, SEEK\_END);

242 long size = ftell(f);

243 fseek(f, 0, SEEK\_SET);

244

245 if (size < sizeof(Superblock) + sizeof(Inode)\*MAX\_FILES) {

246 return 0;

247 }

248

249 // 2. Проверяем сигнатуру суперблока

250 Superblock sb;

251 if (fread(&sb, sizeof(Superblock), 1, f) != 1) {

252 return 0;

253 }

254

255 // Простые проверки (можно добавить magic number)

256 if (sb.total\_blocks != MAX\_BLOCKS || sb.total\_inodes != MAX\_FILES) {

257 return 0;

258 }

259

260 // 3. Проверяем корневую директорию

261 Inode root\_inode;

262 fseek(f, sizeof(Superblock), SEEK\_SET);

263 if (fread(&root\_inode, sizeof(Inode), 1, f) != 1) {

264 return 0;

265 }

266

267 if (!root\_inode.is\_used || !root\_inode.is\_directory) {

268 return 0;

269 }

270

271 return 1;

272 }

273

274 long get\_block\_offset(int block\_index) {

275 return sizeof(Superblock)

276 + sizeof(Inode) \* MAX\_FILES

277 + sizeof(DirectoryEntry) \* MAX\_FILES

278 + block\_index \* BLOCK\_SIZE;

279 }

280

281 void help() {

282 printf("\n\n\nc <filename> - создание файла с именем filename\n");

283 printf("d <filename> - удаление файла с именем filename\n");

284 printf("w <filename> - открытие файла с именем filename для записи\n");

285 printf("r <filename> - чтение файла с именем filename\n");

286 printf("mkdir <dirname> - создание директории с именем dirname\n");

287 printf("rmdir <dirname> - удаление директории с именем dirname\n");

288 printf("rm <dirname> - рекурсивное удаление директории с именем dirname\n");

289 printf("cd <dirname> - переход в директорию dirname (переход в предыдущую - ..)\n");

290 printf("ls [dirname] - просмотр текущей директории(\* - опционально) или директории с именем dirname\n");

291 printf("mv <filename> <dirname> - перемещение файла filename в директорию dirname\n");

292 printf("pwd - получение пути к текущей директории\n");

293 printf("е - выход из файловой системы\n\n");

294 printf("Для <filename> и <dirname> возможно указание как полного, так и относительного пути в формате:\n dirname\n ./dirname\n ../dirname\n ./dirname1/dirname2\n /home/.../dirname\n\n\n");

295 }

**Файл sfs.h**

1 #ifndef SFS\_H

2 #define SFS\_H

3

4 #include <stdio.h>

5 #include <string.h>

6

7 // Константы

8 #define MAX\_FILENAME\_LENGTH 256

9 #define MAX\_BLOCKS 1024

10 #define MAX\_FILES 128

11 #define BLOCK\_SIZE 4096

12

13 // Структуры

14 typedef struct {

15 int total\_blocks;

16 int free\_blocks;

17 int total\_inodes;

18 int free\_inodes;

19 unsigned char block\_bitmap[MAX\_BLOCKS/8];

20 } Superblock;

21

22 typedef struct {

23 int is\_used;

24 int is\_directory;

25 int directory\_inode\_index;

26 char filename[MAX\_FILENAME\_LENGTH];

27 int size;

28 int block\_count;

29 int blocks[16]; // Максимум 16 блоков на файл

30 } Inode;

31

32 typedef struct {

33 int inode\_index;

34 char filename[MAX\_FILENAME\_LENGTH];

35 } DirectoryEntry;

36

37 // Глобальные переменные (объявлены как extern)

38 extern FILE \*disk;

39 extern Superblock superblock;

40 extern Inode inode\_table[MAX\_FILES];

41 extern DirectoryEntry directory[MAX\_FILES];

42 extern int current\_directory\_inode;

43 extern char current\_directory[MAX\_FILENAME\_LENGTH];

44

45 // Прототипы функций

46 void sfs\_mkfs(const char \*diskname);

47 void sfs\_mount(const char \*diskname);

48 void sfs\_umount();

49 void sfs\_pwdm();

50 void sfs\_pwd();

51 void help();

52 int is\_valid\_filesystem(FILE \*f);

53 void create\_home\_directory();

54 long get\_block\_offset(int block\_index);

55

56 // Функции для файлов

57 void sfs\_create(const char \*filename);

58 void sfs\_write(const char \*filename, const char \*data);

59 void sfs\_read(const char \*filename);

60 void sfs\_delete(const char \*filename);

61

62 // Функции для директорий

63 void sfs\_cd(const char \*dirname);

64 void sfs\_create\_dir(const char \*dirname);

65 void sfs\_ls\_dir(const char \*dirname);

66 void sfs\_delete\_dir(const char \*dirname);

67 void sfs\_delete\_dir\_recursive(const char \*dirname);

68

69 // Вспомогательные функции

70 int find\_free\_inode();

71 int find\_free\_block();

72 void allocate\_block(int block\_index);

73 void free\_block(int block\_index);

74 void print\_current\_directory();

75 void build\_path\_from\_inode(int inode, char \*path, size\_t path\_size);

76 int resolve\_path\_to\_inode(const char \*path, int \*parent\_inode\_index, char \*basename);

77 void sfs\_move\_to\_dir(const char \*file\_input, const char \*dir\_input);

78

79 #endif

**Файл sfs\_file.c**

1 #include "sfs.h"

2

3 // Функции для работы с файлами

4 void sfs\_create(const char \*filename) {

5 // Проверка длины имени

6 if (strlen(filename) >= MAX\_FILENAME\_LENGTH) {

7 printf("Слишком длинное имя файла.\n");

8 return;

9 }

10

11 // Проверка существования файла

12 for (int i = 0; i < MAX\_FILES; i++) {

13 if (directory[i].inode\_index != -1 &&

14 inode\_table[directory[i].inode\_index].directory\_inode\_index == current\_directory\_inode &&

15 strcmp(directory[i].filename, filename) == 0) {

16 printf("Файл '%s' уже существует.\n", filename);

17 return;

18 }

19 }

20

21 // Создание файла

22 int inode\_index = find\_free\_inode();

23 if (inode\_index == -1) {

24 printf("Нет свободных inode.\n");

25 return;

26 }

27

28 int block\_index = find\_free\_block();

29 if (block\_index == -1) {

30 printf("Нет свободных блоков.\n");

31 return;

32 }

33

34 int dir\_entry\_index = -1;

35 for (int i = 0; i < MAX\_FILES; i++) {

36 if (directory[i].inode\_index == -1) {

37 dir\_entry\_index = i;

38 break;

39 }

40 }

41

42 if (dir\_entry\_index == -1) {

43 printf("Нет места в директории.\n");

44 return;

45 }

46

47 // Заполнение структур

48 inode\_table[inode\_index].is\_used = 1;

49 inode\_table[inode\_index].is\_directory = 0;

50 inode\_table[inode\_index].directory\_inode\_index = current\_directory\_inode;

51 strncpy(inode\_table[inode\_index].filename, filename, MAX\_FILENAME\_LENGTH);

52 inode\_table[inode\_index].size = 0;

53 inode\_table[inode\_index].block\_count = 1;

54 inode\_table[inode\_index].blocks[0] = block\_index;

55

56 directory[dir\_entry\_index].inode\_index = inode\_index;

57 strncpy(directory[dir\_entry\_index].filename, filename, MAX\_FILENAME\_LENGTH);

58

59 superblock.free\_inodes--;

60 allocate\_block(block\_index);

61

62 // Сохранение на диск

63 fseek(disk, 0, SEEK\_SET);

64 fwrite(&superblock, sizeof(Superblock), 1, disk);

65 fseek(disk, sizeof(Superblock), SEEK\_SET);

66 fwrite(inode\_table, sizeof(Inode), MAX\_FILES, disk);

67 fseek(disk, sizeof(Superblock) + sizeof(Inode) \* MAX\_FILES, SEEK\_SET);

68 fwrite(directory, sizeof(DirectoryEntry), MAX\_FILES, disk);

69

70 print\_current\_directory();

71

72 printf("Файл '%s' создан.\n", filename);

73 }

74

75 void sfs\_write(const char \*filename, const char \*data) {

76 for (int i = 0; i < MAX\_FILES; i++) {

77 if (strncmp(directory[i].filename, filename, MAX\_FILENAME\_LENGTH) == 0) {

78 Inode \*inode = &inode\_table[directory[i].inode\_index];

79

80 int data\_written = 0;

81 int size = strlen(data);

82

83 for (int j = 0; j < inode->block\_count; j++) {

84 int block\_size = (size - data\_written > BLOCK\_SIZE) ? BLOCK\_SIZE : size - data\_written;

85

86 fseek(disk, get\_block\_offset(inode->blocks[j]), SEEK\_SET);

87 fwrite(data + data\_written, 1, block\_size, disk);

88

89 data\_written += block\_size;

90 if (data\_written == size) break;

91 }

92 fflush(disk);

93

94 inode->size = size;

95

96 fseek(disk, sizeof(Superblock), SEEK\_SET);

97 fwrite(inode\_table, sizeof(Inode), MAX\_FILES, disk);

98

99 fflush(disk);

100

101 printf("Данные в файл '%s' успешно записаны.\n", filename);

102 return;

103 }

104 }

105 printf("Файл '%s' не найден.\n", filename);

106 }

107

108 void sfs\_read(const char \*filename) {

109 for (int i = 0; i < MAX\_FILES; i++) {

110 if (strncmp(directory[i].filename, filename, MAX\_FILENAME\_LENGTH) == 0) {

111 Inode \*inode = &inode\_table[directory[i].inode\_index];

112

113 int data\_read = 0;

114 int size = inode->size;

115 char buffer[size + 1];

116

117 for (int j = 0; j < inode->block\_count; j++) {

118 int block\_size = (size - data\_read > BLOCK\_SIZE) ? BLOCK\_SIZE : size - data\_read;

119

120 fseek(disk, get\_block\_offset(inode->blocks[j]), SEEK\_SET);

121 fread(buffer + data\_read, 1, block\_size, disk);

122

123 data\_read += block\_size;

124 if (data\_read == size) break;

125 }

126

127 buffer[size] = '\0';

128

129 printf("Данные из файла '%s':\n%s\n", filename, buffer);

130 return;

131 }

132 }

133 printf("Файл '%s' не найден.\n", filename);

134 }

135

136 void sfs\_delete(const char \*filename) {

137 if (disk == NULL) {

138 fprintf(stderr, "Ошибка: файловая система не смонтирована.\n");

139 return;

140 }

141

142 if (filename == NULL || strlen(filename) == 0) {

143 printf("Ошибка: не указано имя файла.\n");

144 return;

145 }

146

147 // Ищем файл в текущей директории

148 int dir\_entry\_index = -1;

149 int file\_inode\_index = -1;

150

151 for (int i = 0; i < MAX\_FILES; i++) {

152 if (directory[i].inode\_index != -1 &&

153 inode\_table[directory[i].inode\_index].directory\_inode\_index == current\_directory\_inode &&

154 strcmp(directory[i].filename, filename) == 0) {

155

156 // Проверяем, что это не директория

157 if (inode\_table[directory[i].inode\_index].is\_directory) {

158 printf("Ошибка: '%s' является директорией. Используйте ddir для удаления директорий.\n", filename);

159 return;

160 }

161

162 dir\_entry\_index = i;

163 file\_inode\_index = directory[i].inode\_index;

164 break;

165 }

166 }

167

168 if (dir\_entry\_index == -1) {

169 printf("Файл '%s' не найден в текущей директории.\n", filename);

170 return;

171 }

172

173 Inode \*file\_inode = &inode\_table[file\_inode\_index];

174

175 // Освобождаем все блоки файла

176 for (int i = 0; i < file\_inode->block\_count; i++) {

177 int block\_index = file\_inode->blocks[i];

178

179 if (block\_index < 0 || block\_index >= MAX\_BLOCKS) {

180 printf("Предупреждение: некорректный индекс блока %d, пропускаем.\n", block\_index);

181 continue;

182 }

183

184 // Очищаем блок на диске

185 char zero\_block[BLOCK\_SIZE] = {0};

186 fseek(disk, block\_index \* BLOCK\_SIZE, SEEK\_SET);

187 fwrite(zero\_block, BLOCK\_SIZE, 1, disk);

188

189 // Освобождаем блок в битовой карте

190 free\_block(block\_index);

191 }

192

193 // Освобождаем inode

194 memset(file\_inode, 0, sizeof(Inode));

195 superblock.free\_inodes++;

196

197 // Удаляем запись из директории

198 directory[dir\_entry\_index].inode\_index = -1;

199 memset(directory[dir\_entry\_index].filename, 0, MAX\_FILENAME\_LENGTH);

200

201 // Сохраняем изменения на диск

202 fseek(disk, 0, SEEK\_SET);

203 fwrite(&superblock, sizeof(Superblock), 1, disk);

204

205 fseek(disk, sizeof(Superblock), SEEK\_SET);

206 fwrite(inode\_table, sizeof(Inode), MAX\_FILES, disk);

207

208 fseek(disk, sizeof(Superblock) + sizeof(Inode) \* MAX\_FILES, SEEK\_SET);

209 fwrite(directory, sizeof(DirectoryEntry), MAX\_FILES, disk);

210

211 fflush(disk); // Гарантируем запись на диск

212

213 printf("Файл '%s' успешно удален.\n", filename);

214 }

**Файл sfs\_dir.c**

1 #include "sfs.h"

2

3 void create\_home\_directory() {

4 // Проверка существования /home

5 for (int i = 0; i < MAX\_FILES; i++) {

6 if (directory[i].inode\_index != -1 &&

7 strcmp(directory[i].filename, "home") == 0) {

8 return;

9 }

10 }

11

12 // Создание /home

13 int home\_inode = find\_free\_inode();

14 if (home\_inode == -1) return;

15

16 inode\_table[home\_inode].is\_used = 1;

17 inode\_table[home\_inode].is\_directory = 1;

18 inode\_table[home\_inode].directory\_inode\_index = 0;

19 strncpy(inode\_table[home\_inode].filename, "home", MAX\_FILENAME\_LENGTH);

20

21 for (int i = 1; i < MAX\_FILES; i++) {

22 if (directory[i].inode\_index == -1) {

23 directory[i].inode\_index = home\_inode;

24 strncpy(directory[i].filename, "home", MAX\_FILENAME\_LENGTH);

25 break;

26 }

27 }

28

29 superblock.free\_inodes--;

30 current\_directory\_inode = home\_inode;

31 strncpy(current\_directory, "/home", MAX\_FILENAME\_LENGTH);

32 printf("Директория /home создана.\n");

33 }

34

35 void sfs\_cd(const char \*dirname) {

36 if (dirname == NULL || strlen(dirname) == 0) {

37 printf("Ошибка: путь не указан.\n");

38 return;

39 }

40

41 char basename[MAX\_FILENAME\_LENGTH];

42 int parent\_inode;

43 int target\_inode = resolve\_path\_to\_inode(dirname, &parent\_inode, basename);

44

45 // Проверяем, что директория существует

46 if (target\_inode == -1) {

47 printf("Директория '%s' не найдена.\n", dirname);

48 return;

49 }

50

51 // Проверяем, что это действительно директория

52 if (!inode\_table[target\_inode].is\_directory) {

53 printf("'%s' не является директорией.\n", dirname);

54 return;

55 }

56

57 // Обновляем текущую директорию

58 current\_directory\_inode = target\_inode;

59

60 // Обновляем строковое представление пути

61 if (target\_inode == 0) {

62 strcpy(current\_directory, "/");

63 } else {

64 // Строим полный путь по inode

65 build\_path\_from\_inode(target\_inode, current\_directory, MAX\_FILENAME\_LENGTH);

66 }

67

68 printf("Текущая директория: %s\n", current\_directory);

69 }

70

71 void sfs\_create\_dir(const char \*dirname) {

72 // Проверка длины имени

73 if (strlen(dirname) >= MAX\_FILENAME\_LENGTH) {

74 printf("Слишком длинное имя директории.\n");

75 return;

76 }

77

78 // Находим родительскую директорию (текущую или корневую)

79 int parent\_inode = current\_directory\_inode;

80 if (strcmp(dirname, "home") == 0) {

81 parent\_inode = 0; // Корень для /home

82 }

83

84 // Проверяем, нет ли уже такой директории

85 for (int i = 0; i < MAX\_FILES; i++) {

86 if (directory[i].inode\_index >= 0 &&

87 inode\_table[directory[i].inode\_index].directory\_inode\_index == parent\_inode &&

88 strcmp(directory[i].filename, dirname) == 0) {

89 printf("Директория '%s' уже существует.\n", dirname);

90 return;

91 }

92 }

93

94 // Ищем свободный inode

95 int inode\_index = find\_free\_inode();

96 if (inode\_index == -1) {

97 printf("Нет свободных inode для новой директории.\n");

98 return;

99 }

100

101 // Ищем свободный блок

102 int block\_index = find\_free\_block();

103 if (block\_index == -1) {

104 printf("Нет свободных блоков для хранения данных.\n");

105 return;

106 }

107

108 // Создаем inode для директории

109 Inode \*inode = &inode\_table[inode\_index];

110 inode->is\_used = 1;

111 inode->is\_directory = 1;

112 inode->directory\_inode\_index = parent\_inode; // Устанавливаем родителя

113 inode->size = 0;

114 inode->block\_count = 1;

115 inode->blocks[0] = block\_index;

116

117 // Находим свободную запись в directory

118 int dir\_entry\_index = -1;

119 for (int i = 0; i < MAX\_FILES; i++) {

120 if (directory[i].inode\_index == -1) {

121 dir\_entry\_index = i;

122 break;

123 }

124 }

125

126 if (dir\_entry\_index == -1) {

127 printf("Нет свободных записей в directory.\n");

128 return;

129 }

130

131 // Заполняем запись в directory

132 DirectoryEntry \*entry = &directory[dir\_entry\_index];

133 strncpy(entry->filename, dirname, MAX\_FILENAME\_LENGTH);

134 entry->inode\_index = inode\_index;

135

136 superblock.free\_inodes--;

137 allocate\_block(block\_index);

138

139 // Обновляем данные на диске

140 fseek(disk, 0, SEEK\_SET);

141 fwrite(&superblock, sizeof(Superblock), 1, disk);

142 fseek(disk, sizeof(Superblock), SEEK\_SET);

143 fwrite(inode\_table, sizeof(Inode), MAX\_FILES, disk);

144 fseek(disk, sizeof(Superblock) + sizeof(Inode) \* MAX\_FILES, SEEK\_SET);

145 fwrite(directory, sizeof(DirectoryEntry), MAX\_FILES, disk);

146 fflush(disk);

147

148 printf("Директория '%s' успешно создана.\n", dirname);

149 }

150

151 void sfs\_ls\_dir(const char \*dirname) {

152 // Определяем директорию для вывода содержимого

153 int target\_inode;

154 char basename[MAX\_FILENAME\_LENGTH];

155 int parent\_inode;

156

157 if (dirname == NULL || strlen(dirname) == 0) {

158 // Если путь не указан - используем текущую директорию

159 target\_inode = current\_directory\_inode;

160 } else {

161 // Разрешаем переданный путь

162 target\_inode = resolve\_path\_to\_inode(dirname, &parent\_inode, basename);

163 }

164

165 // Проверяем, что объект существует

166 if (target\_inode == -1) {

167 printf("'%s' не найдено.\n", dirname ? dirname : "текущий путь");

168 return;

169 }

170

171 // Проверяем, что это директория (если не корень)

172 if (target\_inode != 0 && !inode\_table[target\_inode].is\_directory) {

173 printf("'%s' не является директорией.\n", dirname ? dirname : "текущий путь");

174 return;

175 }

176

177 // Получаем полное имя директории для вывода

178 char dir\_path[MAX\_FILENAME\_LENGTH];

179 if (target\_inode == 0) {

180 strcpy(dir\_path, "/");

181 } else {

182 build\_path\_from\_inode(target\_inode, dir\_path, MAX\_FILENAME\_LENGTH);

183 }

184

185 printf("Содержимое директории '%s':\n", dir\_path);

186

187 // Выводим содержимое директории

188 int empty = 1;

189 for (int i = 0; i < MAX\_FILES; i++) {

190 if (directory[i].inode\_index != -1) {

191 // Для файлов и поддиректорий этой директории

192 if (inode\_table[directory[i].inode\_index].directory\_inode\_index == target\_inode) {

193 printf("- %s", directory[i].filename);

194 if (inode\_table[directory[i].inode\_index].is\_directory) {

195 printf(" (директория)");

196 } else {

197 printf(" (файл, размер: %d)", inode\_table[directory[i].inode\_index].size);

198 }

199 printf("\n");

200 empty = 0;

201 }

202 }

203 }

204

205 if (empty) {

206 printf("Директория пуста.\n");

207 }

208 }

209

210 void sfs\_delete\_dir(const char \*dirname) {

211 if (disk == NULL) {

212 fprintf(stderr, "Ошибка: файловая система не смонтирована.\n");

213 return;

214 }

215

216 if (dirname == NULL || strlen(dirname) == 0) {

217 printf("Ошибка: не указано имя директории.\n");

218 return;

219 }

220

221 // Разрешаем путь к директории

222 int parent\_inode;

223 char basename[MAX\_FILENAME\_LENGTH];

224 int dir\_inode = resolve\_path\_to\_inode(dirname, &parent\_inode, basename);

225

226 // Проверяем, что директория существует

227 if (dir\_inode == -1) {

228 printf("Директория '%s' не найдена.\n", dirname);

229 return;

230 }

231

232 // Нельзя удалить корневую директорию

233 if (dir\_inode == 0) {

234 printf("Ошибка: нельзя удалить корневую директорию.\n");

235 return;

236 }

237

238 // Проверяем, что это действительно директория

239 if (!inode\_table[dir\_inode].is\_directory) {

240 printf("Ошибка: '%s' не является директорией.\n", dirname);

241 return;

242 }

243

244 // Проверяем, не пытаемся ли удалить текущую директорию

245 if (dir\_inode == current\_directory\_inode) {

246 printf("Ошибка: нельзя удалить текущую директорию.\n");

247 return;

248 }

249

250 // Проверяем, пуста ли директория

251 for (int i = 0; i < MAX\_FILES; i++) {

252 if (directory[i].inode\_index != -1 &&

253 inode\_table[directory[i].inode\_index].directory\_inode\_index == dir\_inode) {

254 printf("Директория '%s' не пуста, невозможно удалить.\n", dirname);

255 return;

256 }

257 }

258

259 // Находим запись в directory для удаляемой директории

260 int dir\_entry\_index = -1;

261 for (int i = 0; i < MAX\_FILES; i++) {

262 if (directory[i].inode\_index == dir\_inode) {

263 dir\_entry\_index = i;

264 break;

265 }

266 }

267

268 if (dir\_entry\_index == -1) {

269 printf("Ошибка: не найдена запись в директории.\n");

270 return;

271 }

272

273 // Получаем полный путь для сообщения

274 char dir\_path[MAX\_FILENAME\_LENGTH];

275 build\_path\_from\_inode(dir\_inode, dir\_path, sizeof(dir\_path));

276

277 // Освобождаем inode

278 inode\_table[dir\_inode].is\_used = 0;

279 superblock.free\_inodes++;

280

281 // Удаляем запись из directory

282 directory[dir\_entry\_index].inode\_index = -1;

283 memset(directory[dir\_entry\_index].filename, 0, MAX\_FILENAME\_LENGTH);

284

285 // Сохраняем изменения на диск

286 fseek(disk, 0, SEEK\_SET);

287 fwrite(&superblock, sizeof(Superblock), 1, disk);

288

289 fseek(disk, sizeof(Superblock), SEEK\_SET);

290 fwrite(inode\_table, sizeof(Inode), MAX\_FILES, disk);

291

292 fseek(disk, sizeof(Superblock) + sizeof(Inode) \* MAX\_FILES, SEEK\_SET);

293 fwrite(directory, sizeof(DirectoryEntry), MAX\_FILES, disk);

294

295 fflush(disk); // Гарантируем запись на диск

296

297 printf("Директория '%s' успешно удалена.\n", dir\_path);

298 }

299

300 int resolve\_path\_to\_inode(const char \*path, int \*parent\_inode\_index, char \*basename) {

301 if (!path || !parent\_inode\_index || !basename) {

302 printf("[ERROR] Неверные аргументы resolve\_path\_to\_inode.\n");

303 return -1;

304 }

305

306 // Создаем копию пути для безопасной работы

307 char temp\_path[MAX\_FILENAME\_LENGTH \* 2]; // Увеличиваем буфер для сложных путей

308 strncpy(temp\_path, path, sizeof(temp\_path));

309 temp\_path[sizeof(temp\_path) - 1] = '\0';

310

311 //printf("[DEBUG] resolve\_path\_to\_inode: path = '%s'\n", temp\_path);

312

313 char \*token;

314 char \*rest = temp\_path;

315 int current\_inode = (path[0] == '/') ? 0 : current\_directory\_inode;

316 \*parent\_inode\_index = -1;

317 char last\_token[MAX\_FILENAME\_LENGTH] = "";

318

319 // Обрабатываем случай корневого пути

320 if (strcmp(temp\_path, "/") == 0) {

321 strncpy(basename, "/", MAX\_FILENAME\_LENGTH);

322 \*parent\_inode\_index = -1;

323 return 0; // Корневой inode

324 }

325

326 // Разбираем путь по токенам

327 while ((token = strtok\_r(rest, "/", &rest))) {

328 // printf("[DEBUG] token = '%s'\n", token);

329

330 // Пропускаем пустые токены и текущую директорию

331 if (strlen(token) == 0 || strcmp(token, ".") == 0) {

332 continue;

333 }

334

335 // Обработка перехода в родительскую директорию

336 if (strcmp(token, "..") == 0) {

337 if (current\_inode != 0) { // Не выходим за пределы корня

338 \*parent\_inode\_index = inode\_table[current\_inode].directory\_inode\_index;

339 current\_inode = \*parent\_inode\_index;

340 }

341 continue;

342 }

343

344 // Поиск токена в текущей директории

345 int found = 0;

346 for (int i = 0; i < MAX\_FILES; i++) {

347 if (directory[i].inode\_index >= 0 &&

348 inode\_table[directory[i].inode\_index].directory\_inode\_index == current\_inode &&

349 strncmp(directory[i].filename, token, MAX\_FILENAME\_LENGTH) == 0) {

350

351 \*parent\_inode\_index = current\_inode;

352 current\_inode = directory[i].inode\_index;

353 found = 1;

354 strncpy(last\_token, token, MAX\_FILENAME\_LENGTH);

355 break;

356 }

357 }

358

359 if (!found) {

360 // printf("[DEBUG] '%s' не найден в inode %d\n", token, current\_inode);

361 strncpy(basename, token, MAX\_FILENAME\_LENGTH);

362 return -1;

363 }

364 }

365

366 // Обработка случая, когда путь заканчивается слешем (например, "dir/")

367 if (strlen(last\_token) == 0 && path[strlen(path)-1] == '/') {

368 strncpy(basename, "", MAX\_FILENAME\_LENGTH);

369 return current\_inode;

370 }

371

372 // Сохраняем результат

373 strncpy(basename, last\_token, MAX\_FILENAME\_LENGTH);

374 //printf("[DEBUG] Успешно: basename='%s', inode=%d, parent\_inode=%d\n",

375 // basename, current\_inode, \*parent\_inode\_index);

376

377 return current\_inode;

378 }

379

380 void sfs\_move\_to\_dir(const char \*file\_input, const char \*dir\_input) {

381 int file\_parent\_inode, dir\_parent\_inode;

382 char file\_name[MAX\_FILENAME\_LENGTH];

383 char dir\_name[MAX\_FILENAME\_LENGTH];

384

385 // Получаем inode файла и родительской директории

386 int file\_inode\_index = resolve\_path\_to\_inode(file\_input, &file\_parent\_inode, file\_name);

387 if (file\_inode\_index == -1) {

388 printf("Файл '%s' не найден.\n", file\_input);

389 return;

390 }

391

392 // Получаем inode целевой директории

393 int dir\_inode\_index = resolve\_path\_to\_inode(dir\_input, &dir\_parent\_inode, dir\_name);

394 if (dir\_inode\_index == -1 || !inode\_table[dir\_inode\_index].is\_directory) {

395 printf("Директория '%s' не найдена или это не директория.\n", dir\_input);

396 return;

397 }

398

399 // Проверяем, нет ли файла с таким именем в целевой директории

400 for (int i = 0; i < MAX\_FILES; i++) {

401 if (directory[i].inode\_index >= 0 &&

402 inode\_table[directory[i].inode\_index].directory\_inode\_index == dir\_inode\_index &&

403 strcmp(directory[i].filename, file\_name) == 0) {

404 printf("Файл с именем '%s' уже существует в директории '%s'.\n", file\_name, dir\_input);

405 return;

406 }

407 }

408

409 // Удаляем запись из старой директории

410 for (int i = 0; i < MAX\_FILES; i++) {

411 if (directory[i].inode\_index == file\_inode\_index &&

412 inode\_table[directory[i].inode\_index].directory\_inode\_index == file\_parent\_inode) {

413 directory[i].inode\_index = -1; // Помечаем как свободную запись

414 break;

415 }

416 }

417

418 // Добавляем запись в новую директорию

419 for (int i = 0; i < MAX\_FILES; i++) {

420 if (directory[i].inode\_index == -1) { // Находим свободную запись

421 directory[i].inode\_index = file\_inode\_index;

422 strncpy(directory[i].filename, file\_name, MAX\_FILENAME\_LENGTH);

423 inode\_table[file\_inode\_index].directory\_inode\_index = dir\_inode\_index;

424 break;

425 }

426 }

427

428 // Сохраняем изменения

429 fseek(disk, sizeof(Superblock) + sizeof(Inode) \* MAX\_FILES, SEEK\_SET);

430 fwrite(directory, sizeof(DirectoryEntry), MAX\_FILES, disk);

431

432 printf("Файл '%s' перемещён в '%s'.\n", file\_input, dir\_input);

433 }

434

435 // Рекурсивное удаление директории

436 void sfs\_delete\_dir\_recursive(const char \*dirname) {

437 if (disk == NULL) {

438 fprintf(stderr, "Ошибка: файловая система не смонтирована.\n");

439 return;

440 }

441

442 if (dirname == NULL || strlen(dirname) == 0) {

443 printf("Ошибка: не указано имя директории.\n");

444 return;

445 }

446

447 // Разрешаем путь к директории

448 int parent\_inode;

449 char basename[MAX\_FILENAME\_LENGTH];

450 int dir\_inode = resolve\_path\_to\_inode(dirname, &parent\_inode, basename);

451

452 // Проверяем, что директория существует

453 if (dir\_inode == -1) {

454 printf("Директория '%s' не найдена.\n", dirname);

455 return;

456 }

457

458 // Нельзя удалить корневую директорию

459 if (dir\_inode == 0) {

460 printf("Ошибка: нельзя удалить корневую директорию.\n");

461 return;

462 }

463

464 // Проверяем, что это действительно директория

465 if (!inode\_table[dir\_inode].is\_directory) {

466 printf("Ошибка: '%s' не является директорией.\n", dirname);

467 return;

468 }

469

470 // Проверяем, не пытаемся ли удалить текущую директорию

471 if (dir\_inode == current\_directory\_inode) {

472 printf("Ошибка: нельзя удалить текущую директорию.\n");

473 return;

474 }

475

476 // Получаем полный путь для сообщений

477 char dir\_path[MAX\_FILENAME\_LENGTH];

478 build\_path\_from\_inode(dir\_inode, dir\_path, sizeof(dir\_path));

479

480 // Рекурсивно удаляем все содержимое директории

481 for (int i = 0; i < MAX\_FILES; i++) {

482 if (directory[i].inode\_index != -1 &&

483 inode\_table[directory[i].inode\_index].directory\_inode\_index == dir\_inode) {

484

485 // Для поддиректорий вызываем рекурсивное удаление

486 if (inode\_table[directory[i].inode\_index].is\_directory) {

487 char subdir\_path[MAX\_FILENAME\_LENGTH];

488 snprintf(subdir\_path, MAX\_FILENAME\_LENGTH, "%s/%s", dir\_path, directory[i].filename);

489 sfs\_delete\_dir\_recursive(subdir\_path);

490 }

491 // Для файлов вызываем обычное удаление

492 else {

493 sfs\_delete(directory[i].filename);

494 }

495 }

496 }

497

498 // Находим запись в directory для удаляемой директории

499 int dir\_entry\_index = -1;

500 for (int i = 0; i < MAX\_FILES; i++) {

501 if (directory[i].inode\_index == dir\_inode) {

502 dir\_entry\_index = i;

503 break;

504 }

505 }

506

507 if (dir\_entry\_index == -1) {

508 printf("Ошибка: не найдена запись в директории.\n");

509 return;

510 }

511

512 // Освобождаем блоки директории (если они есть)

513 for (int i = 0; i < inode\_table[dir\_inode].block\_count; i++) {

514 int block\_index = inode\_table[dir\_inode].blocks[i];

515 if (block\_index >= 0 && block\_index < MAX\_BLOCKS) {

516 free\_block(block\_index);

517 }

518 }

519

520 // Освобождаем inode

521 inode\_table[dir\_inode].is\_used = 0;

522 superblock.free\_inodes++;

523

524 // Удаляем запись из directory

525 directory[dir\_entry\_index].inode\_index = -1;

526 memset(directory[dir\_entry\_index].filename, 0, MAX\_FILENAME\_LENGTH);

527

528 // Сохраняем изменения на диск

529 fseek(disk, 0, SEEK\_SET);

530 fwrite(&superblock, sizeof(Superblock), 1, disk);

531

532 fseek(disk, sizeof(Superblock), SEEK\_SET);

533 fwrite(inode\_table, sizeof(Inode), MAX\_FILES, disk);

534

535 fseek(disk, sizeof(Superblock) + sizeof(Inode) \* MAX\_FILES, SEEK\_SET);

536 fwrite(directory, sizeof(DirectoryEntry), MAX\_FILES, disk);

537

538 fflush(disk); // Гарантируем запись на диск

539 printf("Директория '%s' и все её содержимое успешно удалены.\n", dir\_path);}

## **ПРИЛОЖЕНИЕ Е**

(обязательное)

Ведомость документов