СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В современном мире цифровых технологий, где данные являются одним из ключевых активов, файловые системы служат фундаментальной основой для их организации, хранения и доступа. Традиционные файловые системы, такие как ext4, NTFS или APFS, являются сложными, высокопроизводительными компонентами, интегрированными непосредственно в ядро операционной системы. Однако разработка или модификация файловых систем на уровне ядра — чрезвычайно сложная и рискованная задача, требующая глубоких знаний внутренней архитектуры ОС и сопряжённая с высоким риском нарушения стабильности всей системы. Это создаёт значительный барьер для исследователей, студентов и разработчиков, желающих экспериментировать и глубже понять принципы работы хранения данных.

Именно здесь на сцену выходит технология FUSE (Filesystem in Userspace), которая произвела революцию в подходе к созданию файловых систем, особенно в среде операционных систем семейства Linux. FUSE предоставляет стабильный программный интерфейс, позволяющий реализовать логику файловой системы в виде обычного пользовательского процесса, который безопасно взаимодействует с ядром.

Данная курсовая работа, «Простая файловая система в пространстве пользователя», представляет собой полную реализацию драйвера файловой системы, совместимой с форматом FAT16, с использованием библиотеки FUSE. Актуальность её создания обусловлена образовательной необходимостью в наглядном и функциональном примере, который демонстрирует весь жизненный цикл файловых операций: от форматирования виртуального диска (файла-образа) и управления метаданными до создания, чтения, записи и удаления файлов и вложенных каталогов. В ходе разработки были использованы возможности библиотеки FUSE для интеграции с виртуальной файловой системой (VFS) ядра, а также эффективный механизм отображения файлов в память (mmap) для прямого взаимодействия с образом диска как с единым блоком памяти.

Объектом исследования является процесс реализации логики файловой системы в пространстве пользователя. Предметом — методы и средства для взаимодействия с ядром ОС посредством FUSE, принципы организации структур данных на диске (таблицы размещения файлов, записи каталогов) и эффективного доступа к ним через отображение файла-образа в память. Практическая значимость работы носит в первую очередь образовательный характер, предоставляя рабочий прототип, который позволяет на практике изучить внутреннее устройство одной из классических файловых систем, не прибегая к сложностям программирования на уровне ядра. Кроме того, проект может служить основой для дальнейших экспериментов, таких как реализация поддержки длинных имен файлов, журналирования или шифрования данных.

1 ОБЗОР ЛИТЕРАТУРЫ

1.1 Постановка задачи

В контексте непрерывного развития вычислительных систем и усложнения программно-аппаратных комплексов, вопросы эмуляции, виртуализации и анализа данных приобретают первостепенную важность. Центральной задачей, поставленной в рамках данного проекта, является теоретическая проработка и практическая реализация полнофункционального драйвера файловой системы, способного оперировать в изолированном пользовательском пространстве (userspace). Фундаментальным отличием разрабатываемого решения от классических системных драйверов является его объект взаимодействия: вместо физического носителя информации, такого как жесткий диск или его раздел, система оперирует целостной файловой структурой, инкапсулированной в единый файл-образ на диске.

Такой подход открывает широчайшие перспективы для решения целого спектра прикладных и исследовательских задач. В области цифровой криминалистики это обеспечивает возможность безопасного и контролируемого анализа образов дисков с сохранением неизменности цифровых улик. Для образовательных целей создается уникальная возможность наглядной демонстрации фундаментальных принципов организации и функционирования файловых систем без риска нарушения целостности основной операционной системы. Стандартные системные утилиты, хотя и предоставляют механизмы для работы с образами, зачастую требуют нетривиальных манипуляций, обладают высоким порогом вхождения для неподготовленного пользователя и не всегда обеспечивают необходимый уровень гибкости. Разрабатываемая программа призвана стать элегантным и универсальным решением этой проблемы, предоставляя прозрачный, интуитивно понятный и полностью интегрированный в операционную систему интерфейс для работы с файлами и каталогами внутри образа, как если бы он являлся стандартным блочным устройством.

1.2 FUSE и его роль в проекте

Архитектура современных операционных систем, в частности систем семейства UNIX, предполагает строгую иерархию и разделение привилегий между пространством ядра (kernel space) и пространством пользователя (user space). Традиционно, реализация файловых систем относилась к прерогативе ядра, что делало их разработку чрезвычайно сложным, трудоемким и сопряженным с высокими рисками процессом, где любая ошибка могла повлечь за собой фатальный сбой всей системы. Технология FUSE (Filesystem in Userspace) представляет собой парадигматический сдвиг в этом подходе, предлагая стабильный и безопасный программный интерфейс, который

действует как мост между ядром операционной системы и обычным пользовательским процессом.

В рамках данного проекта FUSE выступает в качестве краеугольного камня всей архитектуры. Его роль заключается в перехвате запросов, поступающих от слоя виртуальной файловой системы (VFS) ядра и адресованных к нашей точке монтирования. Каждый стандартный системный вызов — будь то чтение файла, создание каталога или запрос атрибутов — транслируется FUSE в вызов соответствующей, заранее зарегистрированной функции-обработчика в нашей программе. Таким образом, вся сложная логика, связанная с интерпретацией структуры FAT16, навигацией по цепочкам кластеров в таблице размещения файлов, анализом записей каталогов и непосредственной манипуляцией данными, инкапсулируется внутри пользовательского процесса. Это не только многократно упрощает процесс разработки и отладки, но и, что наиболее важно, полностью изолирует потенциальные сбои, гарантируя незыблемость и стабильность ядра операционной системы и превращая нашу программу в безопасную "песочницу" для реализации логики файловой системы.

1.3 Механизмы доступа к данным и их целостность

Эффективность работы любой файловой системы напрямую зависит от скорости выполнения операций ввода-вывода. Классический подход, основанный на последовательных операциях чтения и записи в файл, сопряжен со значительными накладными расходами из-за множественных дорогостоящих системных вызовов и обращений к дисковой подсистеме. Для преодоления этого узкого места в проекте был применен высокоэффективный механизм отображения файла в память, реализуемый посредством системного вызова mmap. Данная технология позволяет полностью отобразить содержимое файла-образа в виртуальное адресное пространство процесса, фактически стирая грань между файлом на диске и массивом в оперативной памяти.

Такое архитектурное решение дает колоссальное преимущество в производительности: доступ к любой области файловой системы, от суперблока до отдаленного кластера данных, сводится к молниеносной операции вычисления смещения и разыменования указателя. Это избавляет от необходимости постоянно отслеживать файловый курсор и позволяет работать со структурами данных напрямую, что существенно упрощает код и повышает его читаемость. Однако, работа в памяти требует обеспечения гарантий персистентности данных. Эту задачу решает системный вызов msync, который в момент размонтирования принудительно синхронизирует измененную область памяти с файлом на диске, выступая гарантом целостности и сохранности всех выполненных операций. Ритуал корректного завершения работы, включающий msync, munmap (отключение отображения)

и close (закрытие файлового дескриптора), обеспечивает грациозное размонтирование и предотвращает возникновение утечек ресурсов или потерю данных.

1.4 Описание используемых библиотек

Для построения надежного и функционального программного продукта была задействована комбинация специализированных системных интерфейсов и фундаментального инструментария, предоставляемого стандартной библиотекой языка C.

Несущей конструкцией всего приложения, безусловно, является библиотека libMfuse, доступ к которой осуществляется через заголовочный файл <fuse.h>. Она предоставляет исчерпывающий набор функций и структур для интеграции пользовательского процесса с ядром ОС. Центральным элементом взаимодействия является структура fuse_operations, которая представляет собой своего рода "контракт", где мы сопоставляем стандартные файловые операции с нашими собственными функциями-реализациями. Весь сложный механизм управления монтированием, организации цикла обработки запросов и коммуникации с ядром инкапсулирован внутри функции fuse_main.

Низкоуровневое взаимодействие с операционной средой, в частности с файловой системой и подсистемой управления виртуальной памятью, обеспечивается набором библиотек стандарта POSIX. Ключевую роль играет <sys/mman.h>, предоставляя API (mmap, munmap, msync) для реализации высокопроизводительного доступа к данным файла-образа. Заголовочные файлы <fcntl.h> и <unistd.h> содержат объявления функций, незаменимых для управления жизненным циклом файла-образа: open для его инициализации, ftruncate для задания исходного размера при форматировании и close для корректного освобождения системных ресурсов.

В качестве фундаментального инструментария, обеспечивающего базовую логику программы, выступают стандартные библиотеки языка C. <stdio.h>, <stdlib.h> и <string.h> предоставляют проверенные временем функции для всех рутинных операций: от вывода диагностических сообщений и динамического выделения памяти до манипуляций со строками и блоками памяти, что является неотъемлемой частью работы со структурами файловой системы. <errno.h> используется для точной диагностики ошибок системного уровня, <time.h> — для работы с временными метками, а <ctype.h> — для корректного преобразования имен файлов в канонический формат FAT.

2 СИСТЕМНОЕ ПРОЕКТИРОВАНИЕ

Проектируемая система реализована в виде пользовательского драйвера файловой системы (userspace driver), использующего технологию FUSE (Filesystem in Userspace). Данный подход является ключевым, поскольку он позволяет реализовать полноценную логику файловой системы в виде обычного приложения, которое взаимодействует с ядром операционной системы через четко определенный интерфейс. Вместо прямого взаимодействия с пользователем через консоль, система получает запросы от виртуальной файловой системы (VFS) ядра, которые инициируются стандартными утилитами командной строки (ls, mkdir, cat, rm и т.д.).

Архитектура системы построена на принципах строгой модульности, инкапсуляции логики и прямого манипулирования данными через отображение файла-образа в память (mmap). Такой подход обеспечивает высокую производительность за счет минимизации системных вызовов чтения/записи, а также четкое разделение ответственности между компонентами: модуль взаимодействия с FUSE, модуль логики FAT16 и модуль управления данными на «диске».

2.1 Общая архитектура и принципы построения

Вся система логически разделяется на пять ключевых компонентов:

- Модуль инициализации и жизненного цикла — отвечает за подготовку рабочей среды. Он обрабатывает запуск и остановку драйвера: открывает или создает файл-образ диска, отображает его в оперативную память с помощью mmap, инициализирует глобальные указатели на ключевые структуры (суперблок, таблица FAT, корневой каталог) и выполняет обратные операции при демонтировании (сохранение изменений на диск, освобождение ресурсов). Этот модуль гарантирует, что файловая система стартует в консистентном состоянии и корректно сохраняет все данные при завершении работы.

- Модуль трансляции операций FUSE (FUSE Bridge) — является ядром взаимодействия с операционной системой. Он представлен структурой fuse_operations, которая сопоставляет стандартные VFS-операции (такие как getattr, readdir, read, write, create, mMkdir) с конкретными функциями, реализованными в нашем драйвере. Этот компонент не имеет пользовательского интерфейса в привычном понимании; его "пользователем" является ядро Linux.

- Ядро навигации по файловой системе — реализует основную логику работы с путями и записями каталогов. Его центральная функция — find_path_entry — отвечает за разбор пути (например, /docs/report.txt), последовательный обход каталогов от корневого до целевого и нахождение дескриптора (directory entry) искомого файла или каталога. Этот модуль

является "мозгом" системы, преобразующим абстрактные пути в конкретные указатели на структуры данных в памяти.

- Слой управления данными и кластерами — отвечает за низкоуровневое управление дисковым пространством. В его ведении находится таблица размещения файлов (FAT), и он реализует ключевые операции: поиск свободного кластера для выделения места под новые данные (find_free_cluster), связывание кластеров в цепочки при записи в файл и их освобождение при удалении. Этот слой абстрагирует работу с "сырыми" блоками данных. Слой абстракции хранения (Memory-Mapped Image) — фундаментальный компонент, основанный на системном вызове mmap. Он представляет весь файл-образ диска (размером 16 МБ) как единый непрерывный массив байтов в оперативной памяти. Это позволяет всем остальным модулям работать с файловой системой так, как будто она полностью загружена в RAM, используя простые арифметические операции с указателями вместо дорогостоящих системных вызовов read/write для доступа к разным частям "диска". Данная структура обеспечивает четкое разделение ответственности, позволяет легко модифицировать и тестировать каждый компонент в отдельности и является основой для эффективной и стабильной работы всей файловой системы.

2.2 Основные блоки программы

2.2.1 Блок инициализации и управления образом

Этот блок, реализованный в функциях fat_init и fat_destroy, является точкой входа и выхода для всей файловой системы. При монтировании (fat_init) он выполняет критически важную последовательность действий: Пытается открыть существующий файл-образ диска, указанный в параметрах запуска. Если файл не найден, он создает его с заданным размером (DISK_SIZE) и производит "форматирование": размечает пространство под метаданные, таблицу FAT и область данных, а также инициализирует служебные записи в FAT. С помощью системного вызова mmap отображает весь файл-образ в виртуальное адресное пространство процесса. Это ключевой шаг, устраняющий необходимость в дисковых операциях ввода-вывода во время работы. Инициализирует глобальные указатели (meta, fat_table, root_dir, data_area) на соответствующие смещения внутри отображенной памяти для быстрого доступа. При демонтировании (fat_destroy) блок гарантирует целостность данных: с помощью msync принудительно записывает все изменения из памяти обратно в файл-образ на диске, после чего освобождает память (munmap) и закрывает файловый дескриптор.

2.2.2 Блок-транслятор VFS-вызовов (FUSE)

Этот блок является интерфейсом между ядром ОС и логикой нашей файловой системы. Он не выполняет сложных вычислений, а действует как диспетчер. Центральным элементом является статическая структура fat_oper, поля которой (.getattr, .read, .write и т.д.) заполнены указателями на наши функции-обработчики. Когда пользователь выполняет команду, например ls -l /mydir, ядро через FUSE вызывает нашу функцию fat_getattr с путем /mydir. Функция fat_getattr, в свою очередь, использует другие блоки для поиска этого каталога и возвращает его атрибуты ядру. Таким образом, этот блок эффективно "переводит" высокоуровневые запросы операционной системы на язык функций нашей файловой системы.

2.2.3 Блок навигации и поиска

Это логический центр программы, отвечающий за интерпретацию путей. Его главная функция, find_path_entry, решает задачу нахождения любого объекта в файловой системе. Получив на вход путь, она разбивает его на компоненты (имена каталогов и файла) и последовательно "спускается" по иерархии. Начиная с корневого каталога (root_dir), она ищет в нем запись для первого компонента пути. Если найдена запись каталога, она использует номер его первого кластера (first_cluster) и функцию get_dir_from_cluster, чтобы получить указатель на данные следующего уровня иерархии, и повторяет поиск для следующего компонента. Этот процесс продолжается до тех пор, пока не будет найден целевой файл/каталог или пока не будет установлено, что путь не существует. Блок активно использует вспомогательные функции, такие как find_entry_in_dir (поиск внутри одного каталога) и to_fat_name (преобразование имен в формат 8.3).

2.2.4 Блок управления дисковым пространством (FAT)

Этот блок отвечает за жизненный цикл данных на "диске". Он оперирует двумя ключевыми сущностями: таблицей FAT и областью данных. Когда требуется выделить место для нового файла или расширить существующий (например, в функции fat_write), вызывается функция find_free_cluster. Она линейно сканирует массив fat_table, ища первую запись со значением FAT_ENTRY_FREE. Найдя свободный кластер, она помечает его как занятый (записывая туда, например, маркер конца файла FAT_ENTRY_EOF) и возвращает его номер. При записи данных, превышающих один кластер, этот блок связывает кластеры в цепочку, записывая в FAT-запись текущего кластера номер следующего. При удалении файла (в fat_unlink) блок проходит по всей цепочке кластеров файла и помечает каждую соответствующую запись в fat_table как свободную (FAT_ENTRY_FREE), делая это пространство

доступным для будущего использования.

3 ФУНКЦИОНАЛЬНОЕ ПРОЕКТИРОВАНИЕ

В рамках данного курсового проекта функциональное проектирование охватывает анализ требований к драйверу файловой системы FAT16, определение набора его ключевых функций, их логических взаимосвязей, а также детальное описание архитектуры взаимодействия с ядром через FUSE. Особое внимание уделяется критически важным аспектам, таким как реализация основных файловых операций (создание, чтение, запись, удаление), управление дисковым пространством через таблицу FMAT и навигация по иерархической структуре каталогов. Такой подход позволяет создать не только функциональный, но и стабильный программный продукт, способный корректно обрабатывать запросы от операционной системы и управлять данными на виртуальном диске.

3.1 Структура программы

Программа, построенная по принципу управляемой событиями архитектуры (event-driven), где "событиями" выступают вызовы от FUSE, начинает свою работу с этапа инициализации. В ходе этого этапа, реализуемого функцией fat_init, происходит открытие или создание файла-образа, его отображение в память (mmap), и инициализация глобальных указателей на ключевые области файловой системы: метаданные, таблицу FAT, корневой каталог и область данных. На этой же стадии происходит парсинг аргументов командной строки для получения пути к образу диска.

После успешной инициализации и монтирования, управление передается главному циклу FUSE (fuse_main), который ожидает запросов от ядра. Любая операция с файлами в смонтированном каталоге (например, ls, cat, mkdir) транслируется ядром в вызов соответствующей функции-обработчика из структуры fat_oper.

Основная логика программы заключается в обработке этих вызовов. Например, при запросе на чтение каталога (readdir) активируется функция, которая находит данные этого каталога в области данных и последовательно считывает его записи, преобразуя их в формат, понятный FUMSE. При записи в файл (write) запускается сложный механизм, который находит или выделяет новые кластеры данных, обновляет таблицу FAT, записывает сами данные в область данных и обновляет метаданные файла (размер, время модификации).

Завершение работы инициируется командой демонтирования файловой системы. В этом случае вызывается функция fat_destroy, которая обеспечивает атомарное сохранение всех накопленных в памяти изменений обратно в файл-образ с помощью msync, после чего корректно освобождает

все ресурсы (munmap, close), гарантируя целостность данных на "диске".

3.1.1 Подключение библиотек и глобальные переменные

Используются стандартные библиотеки языка С для работы с файлами (fcntl.h, unistd.h), строками (string.h), памятью (stdlib.h, sys/mman.h) и типами данных (stddef.h, stdint.h). Ключевой является заголовочный файл fuse.h, который предоставляет весь API для взаимодействия с FUSE.

Объявляются глобальные переменные, которые служат указателями на основные компоненты файловой системы, отображенные в память. Это позволяет избежать повторных вычислений смещений и обеспечивает быстрый доступ: fs_memory: указатель на начало всей области памяти, отображающей файл-образ. meta: указатель на структуру fs_metadata (суперблок) в начале fs_memory. fat_table: указатель на начало таблицы FAT. root_dir: указатель на начало корневого каталога. data_area: указатель на начало области данных, где хранятся файлы и подкаталоги. image_fd: файловый дескриптор открытого файла-образа.

3.1.2 Обнаружение файлов и навигация по каталогам

Эта логика реализована в ядре программы — функции find_path_entry. Она является основной для большинства операций, так как преобразует строковый путь в указатель на конкретную запись каталога (fat16_dir_entry). Алгоритм функции последовательно разбирает путь, разделенный символами '/', и для каждого компонента вызывает вспомогательную функцию find_entry_in_dir, которая осуществляет поиск внутри одного блока данных каталога. Если компонент найден и является каталогом, функция get_dir_from_cluster вычисляет адрес его данных для продолжения поиска на следующем уровне иерархии. Этот механизм позволяет программе эффективно перемещаться по файловой системе.

3.1.3 Получение детальной информации об объекте (getattr)

За эту задачу отвечает функция fat_getattr. Она использует find_path_entry для нахождения целевого файла или каталога. После успешного нахождения она анализирует поле attributes найденной записи. Если установлен флаг ATTR_DIRECTORY, она заполняет структуру stat информацией о каталоге (режим S_IFDIR | 0755). В противном случае, она заполняет ее данными о файле (режим S_IFREG | 0644, размер из поля file_size). Эта функция вызывается такими командами, как ls -l или stat.

3.1.4 Чтение и запись данных в фоновых потоках FUSE

FUSE выполняет операции в собственных рабочих потоках, поэтому наши функции fat_read и fat_write должны быть потокобезопасными.

fat_read: Получив запрос на чтение, функция сперва находит запись файла с помощью find_path_entry. Затем, на основе запрошенного смещения (offset), она вычисляет, с какого кластера в цепочке кластеров файла нужно начать чтение. Она проходит по таблице FAT, переходя от кластера к кластеру, пока не достигнет нужного. Далее она копирует запрошенное количество байт (size) из соответствующих кластеров в области data_area в буфер, предоставленный FUSE.

fat_write: Эта функция является одной из самых сложных. Она также находит файл, а затем определяет, сколько кластеров требуется для записи данных. Если существующих кластеров не хватает, она входит в цикл, в котором вызывает find_free_cluster для выделения новых кластеров и связывает их в цепочку в таблице FAT. После обеспечения достаточного дискового пространства она копирует данные из буфера FUSE в соответствующие кластеры и обновляет размер файла в его записи каталога.

3.1.5 Управление каталогами (mkdir, rmdir)

fat_mkdir: Функция находит родительский каталог, затем ищет в нем свободную запись (find_free_dir_entry) и выделяет новый кластер для данных нового каталога (find_free_cluster). Она заполняет новую запись каталога (имя, атрибуты, номер кластера), а затем инициализирует сам кластер данных нового каталога, создавая в нем обязательные записи "." (ссылка на себя) и ".." (ссылка на родителя).

fat_rmdir: Функция находит удаляемый каталог, проверяет, что он пуст (кроме записей "." и ".."), после чего помечает его запись в родительском каталоге как удаленную (filename[0] = 0xE5) и освобождает кластер, который занимали его данные, помечая соответствующую запись в fat_table как FAT_ENTRY_FREE.

3.1.6 Создание и удаление файлов (create, unlink)

fat_create: Эта операция вызывается, например, командой touch. Функция находит родительский каталог, ищет в нем свободную запись, заполняет ее именем нового файла, атрибутами и устанавливает начальный кластер в FAT_ENTRY_EOF (так как файл пустой) и размер в 0.

fat_unlink: Функция rm вызывает fat_unlink. Она находит файл, проходит по всей цепочке его кластеров в таблице FAT, освобождая каждый из них, после чего помечает запись файла в каталоге как удаленную.

3.1.7 Основная функция (main)

Функция main выступает в роли точки входа и конфигуратора. Она отвечает за парсинг аргументов командной строки, в частности, за извлечение пути к файлу-образу (--image=...). Она подготавливает структуру fat_options и передает ее вместе с остальными аргументами в fuse_main. fuse_main берет на себя всю дальнейшую работу: инициализацию FUSE, вызов нашего fat_init, запуск главного цикла обработки событий и, после демонтирования, вызов fat_destroy.

3.2 Основные функции программы

Основная функция программы — эмуляция работы файловой системы FAT16 поверх обычного файла в операционной системе Linux. На базовом уровне утилита способна создавать, форматировать и монтировать виртуальный диск, представленный файлом-образом. После монтирования она предоставляет ядру операционной системы полный набор стандартных файловых операций. Пользователь, работая со смонтированным каталогом через стандартные утилиты (ls, mkdir, cp, rm и т.д.), фактически вызывает соответствующие функции нашего драйвера.

Ключевой и наиболее сложной функцией является корректное управление дисковым пространством. Это включает в себя динамическое выделение и освобождение кластеров (блоков по 4 КБ) для хранения содержимого файлов и каталогов, а также ведение таблицы размещения файлов (FAT), которая хранит информацию о цепочках кластеров, принадлежащих каждому файлу. Программа способна работать с вложенными каталогами, обрабатывать длинные файлы, занимающие множество кластеров, и корректно изменять их размер. Все эти операции происходят прозрачно для пользователя и операционной системы, которая "видит" полноценный форматированный раздел.

3.2.1 Инициализация среды (fat_init)

Инициализация является основополагающим этапом, подготавливающим все необходимые ресурсы для работы файловой системы. При монтировании программа первым делом анализирует путь к файлу-образу. Если файл существует, он открывается и его содержимое отображается в память с помощью mmap. Если файл не существует, программа создает его, задает необходимый размер (DISK_SIZE) и производит его первичное форматирование: инициализирует суперблок (fs_metadata) с информацией о геометрии диска (размер кластера, смещения до ключевых областей), очищает

таблицу FAT, помечая все кластеры как свободные, и создает пустой корневой каталог. Независимо от сценария, в конце инициализации глобальные указатели (meta, fat_table, root_dir, data_area) указывают на правильные адреса в памяти, подготавливая систему к обработке запросов.

3.2.2 Управление файлами (create, unlink, write, read, truncate)

Система предоставляет полный жизненный цикл для управления файлами:

- Создание (fat_create): При создании нового файла в родительском каталоге находится свободный слот, который заполняется метаданными нового файла (имя, атрибуты). Файл изначально создается пустым, поэтому его размер равен 0, а указатель на первый кластер данных устанавливается в специальное значение "конец файла" (FAT_ENTRY_EOF).
- Запись (fat_write): Это центральная операция. При записи данных система определяет, сколько кластеров необходимо для хранения. Если существующих, выделенных файлу, не хватает, она ищет свободные кластеры в таблице FAT, выделяет их и достраивает цепочку кластеров файла. Затем данные копируются из буфера, предоставленного ядром, в соответствующие кластеры в области данных. Размер файла в его записи каталога обновляется.
- Чтение (fat_read): Система находит файл, по его записи каталога определяет номер первого кластера. Затем, следуя по цепочке в таблице FAT, она находит все кластеры, принадлежащие файлу, и копирует запрошенный диапазон байт из них в буфер ядра.
- Удаление (fat_unlink): При удалении файла система проходит по его цепочке кластеров в FAT, помечая каждый из них как свободный. После этого запись о файле в его родительском каталоге помечается как удаленная.
- Изменение размера (fat_truncate): Эта операция (в данной реализации) в основном изменяет логический размер файла в его метаданных. Реальное освобождение или выделение кластеров происходит при последующей записи.

3.2.3 Управление каталогами (mkdir, rmdir, readdir)

Система поддерживает иерархическую структуру каталогов:

- Создание (fat_mkdir): Для нового каталога выделяется кластер под его содержимое и создается запись в родительском каталоге. Внутри нового кластера автоматически создаются две служебные записи: . (ссылка на себя) и .. (ссылка на родительский каталог), что является стандартом для файловых систем.
- Удаление (fat_rmdir): Каталог можно удалить только в том случае, если он пуст (не содержит файлов или других каталогов, кроме . и ..). При удалении его кластер освобождается, а запись в родительском каталоге помечается как неиспользуемая.

- Чтение (fat_readdir): При запросе на листинг содержимого каталога система находит его данные, проходит по всем записям и передает имена файлов и подкаталогов ядру с помощью специальной callback-функции filler.

3.2.4 Получение атрибутов (getattr)

Эта функция является одной из самых часто вызываемых. Она отвечает на запросы о метаданных файла или каталога, такие как ls -l. Система находит запрошенный объект по пути и считывает его атрибуты из записи каталога (fat16_dir_entry). На основе этих атрибутов (флаг ATTR_DIRECTORY, размер файла file_size) она заполняет стандартную структуру stat, которую FUSE передает ядру. Эта информация включает тип объекта (файл или каталог), права доступа (эмулируются стандартные), размер, а также временные метки (в данной реализации эмулируются текущим временем).

3.2.5 Обработка пользовательских команд

Программа не имеет прямого интерактивного меню. "Пользовательские команды" — это стандартные утилиты Linux (ls, cp, mv, rm, mkdir, touch, cat, echo ... > file и т.д.), которые применяются к смонтированному каталогу. Ядро операционной системы перехватывает эти системные вызовы, и через FUSE они транслируются в вызовы соответствующих функций в нашей программе (fat_readdir, fat_write, fat_unlink и т.д.). Например, команда mkdir /mnt/fat/new_dir приведет к вызову fat_mkdir с аргументом path = "/new_dir". Вся логика обработки этих "команд" заключается в корректной реализации функций из структуры fat_oper.

3.2.6 Корректное завершение работы (destroy)

Обеспечение целостности данных при завершении работы является ключевым аспектом. Процесс завершения инициируется командой демонтирования (umount). Это приводит к вызову функции fat_destroy. Главная задача этой функции — гарантировать, что все изменения, сделанные в оперативной памяти (в fs_memory), будут записаны обратно в файл-образ на диске. Это достигается с помощью системного вызова msync(..., MS_SYNC), который принудительно синхронизирует отображенную область памяти с файлом. После успешной синхронизации память освобождается (munmap), и файловый дескриптор образа закрывается. Такой механизм гарантирует, что даже после интенсивных операций записи файловая система на диске останется в консистентном состоянии.

4 РАЗРАБОТКА ПРОГРАММЫХ МОДУЛЕЙ

В рамках данного проекта были разработаны модули для реализации логики файловой системы FMAT16, взаимодействия с ядром через FUSE, управления дисковым пространством (кластерами и таблицей FAT), а также для выполнения всех стандартных файловых операций. fat_getattr(), fat_readdir() – основные алгоритмы для получения атрибутов и чтения содержимого каталогов. fat_read(), fat_write() – ключевые алгоритмы для чтения и записи данных файлов, включая сложную логику выделения и связывания кластеров. find_path_entry() – центральный алгоритм навигации, используемый большинством других функций для поиска объектов в файловой системе. main(), fat_init(), fat_destroy() – реализуют жизненный цикл драйвера, его инициализацию, монтирование и корректное завершение работы с сохранением данных.

4.1 Разработка алгоритмов

4.1.1 Архитектурная схема алгоритма

В программе реализовано несколько взаимосвязанных ключевых алгоритмов, которые вместе формируют функциональность файловой системы.

Алгоритм навигации и поиска (find_path_entry):

- Инициализация: Поиск начинается с корневого каталога (root_dir). Путь к файлу/каталогу (например, /dir1/file.txt) принимается в качестве входных данных.

- Разбор пути: Путь разбивается на компоненты с помощью strtok.

- Итеративный поиск: Для каждого компонента пути выполняется поиск в текущем каталоге с помощью вспомогательной функции M. find_entry_in_dir преобразует имя компонента в формат FAT 8.3 (to_fat_name) и последовательно сравнивает его с записями в блоке данных каталога.

- Спуск по иерархии: Если компонент найден и является каталогом (атрибут ATTR_DIRECTORY), алгоритм получает номер его первого кластера (first_cluster) и с помощью функции get_dir_from_cluster вычисляет указатель на данные этого подкаталога. Этот указатель становится "текущим каталогом" для следующей итерации.

- Завершение: Поиск завершается, когда обработаны все компоненты пути. Функция возвращает указатель на запись целевого объекта и на данные его родительского каталога. Если на каком-либо этапе компонент не найден, функция возвращает ошибку -ENOENT.

Алгоритм записи в файл (fat_write):

- Поиск файла: Сначала с помощью find_path_entry находится запись (directory entry) целевого файла.

- Расчет потребностей: Вычисляется требуемое количество кластеров для

размещения данных с учетом смещения (offset) и размера (size).

- Выделение кластеров: Алгоритм сравнивает требуемое количество кластеров с уже выделенным. Если требуется больше кластеров:

- Он входит в цикл, в котором многократно вызывает find_free_cluster для поиска свободных ячеек в таблице FAT.

- Каждый найденный свободный кластер связывается с предыдущим, формируя или продолжая цепочку в таблице FAT. Последний кластер в цепочке помечается как FAT_ENTRY_EOF.

- Копирование данных: После обеспечения достаточного дискового пространства алгоритм проходит по цепочке кластеров файла, вычисляет нужные смещения и копирует данные из буфера, предоставленного FUSE, в соответствующие кластеры в области data_area.

- Обновление метаданных: Размер файла в его записи каталога (entry->file_size) обновляется, если он увеличился.

4.1.2 Подробный разбор этапов алгоритма

Выделение и связывание кластеров при записи:

Центральной и самой сложной частью алгоритма fat_write является динамическое расширение файла.

```
// Код в функции fat_write

// Расчет необходимого количества кластеров
size_t required_clusters = (offset + size + meta->cluster_size - 1) / meta->cluster_size;

// Проверка, нужно ли выделять новые кластеры
if (required_clusters > current_clusters_count) {
   // Находим последний кластер в существующей цепочке
   uint16_t last_cluster = ...;

   // Цикл выделения недостающих кластеров
   for (size_t i = current_clusters_count; i < required_clusters; ++i) {
      // 1. Найти свободный кластер
      uint16_t new_cluster = find_free_cluster();
      if (new_cluster == 0)
         break; // Место на диске закончилось

      // 2. Пометить его как конец новой цепочки
      fat_table[new_cluster] = FAT_ENTRY_EOF;
```

```
        // 3. Связать его с предыдущим кластером
        if (last_cluster == 0) { // Если файл был пуст
            entry->first_cluster = new_cluster;
        } else {
            fat_table[last_cluster] = new_cluster;
        }
        last_cluster = new_cluster; // Обновить указатель на последний
кластер
    }
}
```

Этот фрагмент демонстрирует, как система динамически "наращивает" файл. Сначала определяется, сколько всего кластеров понадобится. Если их больше, чем уже есть, алгоритм ищет последний кластер в текущей цепочке и в цикле начинает добавлять новые. Каждый новый кластер, найденный функцией find_free_cluster, "привязывается" к концу цепочки путем записи его номера в FAT-запись предыдущего кластера. Этот механизм обеспечивает целостность файла даже при фрагментации данных на "диске".

4.2 Реализация многопоточности и синхронизации

4.2.1 Архитектура многопоточного взаимодействия

В отличие от интерактивных приложений, где потоки создаются и управляются вручную, в драйвере FUSE многопоточность является неотъемлемой частью самой библиотеки FUSE. Основной поток: После вызова fuse_main, основной поток программы блокируется и передает управление FUSE. Он существует только для поддержания работы драйвера. Рабочие потоки FUSE: Библиотека FUSE создает пул рабочих потоков (worker threads) для обработки входящих запросов от ядра. Когда несколько приложений одновременно обращаются к смонтированной файловой системе (например, одно читает файл, другое создает каталог), FUSE распределяет эти запросы по разным потокам. Это означает, что наши функции-обработчики (fat_read, fat_write, fat_mkdir и т.д.) могут выполняться одновременно в разных потоках.

4.2.2 Запуск и завершение потоков

Программист не управляет жизненным циклом рабочих потоков FUSE напрямую. Запуск: Потоки создаются самой библиотекой FUSE во время выполнения fuse_main. Завершение: Когда файловая система демонтируется, fuse_main завершает работу всех своих потоков, после чего вызывает нашу

функцию fat_destroy и возвращает управление в main.

4.2.3 Синхронизация потоков

Поскольку все операции модифицируют общие структуры данных в памяти (fs_memory), потенциально возможны состояния гонки. Например, два потока одновременно пытаются выделить свободный кластер, что может привести к тому, что один и тот же кластер будет выделен двум разным файлам.

В данной реализации эта проблема не решается явными механизмами синхронизации, такими как мьютексы. Программа полагается на то, что большинство операций с файловой системой на высоком уровне сериализуются ядром, а также на атомарность простых операций записи в память на современных архитектурах.

Однако это является значительным упрощением и потенциальной точкой отказа в высоконагруженных сценариях. В промышленной реализации необходимо было бы защищать критические секции кода мьютексами:

- Защита таблицы FAT: Любые операции, изменяющие fat_table (выделение/освобождение кластеров, построение цепочек), должны быть обернуты в мьютекс. Это касается функций find_free_cluster, fat_write, fat_unlink, fat_mkdir.
- Защита каталогов: Операции, изменяющие содержимое каталога (создание/удаление записей), также требуют блокировки для предотвращения повреждения структуры каталога.

В текущей учебной реализации предполагается, что одновременные модифицирующие операции маловероятны или разнесены во времени достаточно, чтобы не вызывать коллизий.

4.3 Реализация вспомогательных модулей

4.3.1 Модуль управления структурами данных FMAT

Этот модуль представлен набором небольших, но критически важных функций, которые инкапсулируют логику работы со структурами FAT16:

- to_fat_name(const char *path, char *fat_name, char *fat_ext): Эта функция является шлюзом для преобразования стандартных имен файлов (например, "document.txt") в формат 8.3, используемый в FAT. Она разделяет имя и расширение, переводит их в верхний регистр и дополняет пробелами до 8 и 3 символов соответственно. Это обеспечивает совместимость с форматом записи на "диске".
- find_free_cluster(): Инкапсулирует логику поиска первого свободного блока данных. Она выполняет простой линейный поиск по массиву fat_table, ища ячейку со значением FAT_ENTRY_FREE.

- get_dir_from_cluster(uint16_t cluster): Эта функция-хелпер преобразует номер кластера в прямой указатель на область памяти, где хранятся данные этого кластера. Она использует базовый адрес data_area и вычисляет смещение на основе номера кластера и его размера. Это позволяет абстрагироваться от ручных вычислений адресов.

- find_free_dir_entry(fat16_dir_entry *dir): Аналогично find_free_cluster, эта функция ищет свободное место, но уже внутри блока данных каталога. Она ищет запись, первый байт имени которой равен 0x00 (никогда не использовалась) или 0xE5 (была удалена).

4.3.2 Модуль инициализации и форматирования

Этот логический модуль сосредоточен в функции fat_init и отвечает за подготовку "диска" к работе, если файл-образ создается впервые. Основные шаги: Выделение файла: С помощью ftruncate файлу-образу задается точный размер DISK_SIZE. Разметка памяти: Вся область fs_memory обнуляется с помощью memset. Инициализация суперблока: Структура meta заполняется вычисленными значениями: общим количеством кластеров, смещениями до таблицы FAT, корневого каталога и области данных. Эти значения вычисляются на основе констант DISK_SIZE и CLUSTER_SIZE и являются "картой" всего диска. Инициализация FAT: Первые две записи в таблице fat_table заполняются служебными значениями 0xFFF8 и FAT_ENTRY_EOF в соответствии со стандартом FAT16. Этот модуль гарантирует, что при первом запуске создается корректно отформатированная и готовая к использованию файловая система.

5 ПРОГРАММА И МЕТОДИКА ИСПЫТАНИЙ

В ходе испытаний была проведена комплексная проверка работоспособности драйвера файловой системы FAT16 в различных сценариях использования. Целью тестирования являлась проверка корректности реализации всех стандартных файловых операций, устойчивости к граничным условиям, корректности управления дисковым пространством и сохранения целостности данных после демонтирования. Ниже приведены основные этапы тестирования, проводимые с использованием стандартных утилит командной строки Linux в смонтированном каталоге.

5.1 Запуск программы и монтирование диска

Первым шагом является запуск и монтирование файловой системы. Тестирование проводится для двух сценариев:

Запуск с несуществующим файлом-образом:

```
./fat16_fuse mnt --image=new_disk.img
```

Проверяется, что программа корректно выводит сообщение о создании и форматировании нового образа, создает файл new_disk.img заданного размера (16 МБ) и успешно монтируется.

```
Файл-образ не найден. Создание и форматирование new_disk.img...
Форматирование завершено.
FAT16 FUSE FS инициализирована.
```

Запуск с существующим файлом-образом: Проверяется, что программа распознает существующий образ, выводит соответствующее сообщение и успешно монтируется, загружая ранее сохраненное состояние файловой системы. После монтирования первым шагом в методике испытаний является выполнение команды ls -la mnt/. Проверяется, что команда выполняется без ошибок и показывает пустой каталог (содержащий только . и ..), что подтверждает базовую работоспособность readdir и getattr на корневом каталоге.

5.2 Проверка корректности монтирования

Проверка корректности создания, отображения и удаления файлов и каталогов.

Создание каталогов (mkdmir):

```
mkdir mnt/docs
mkdir mnt/docs/work
ls -l mnt/
ls -l mnt/docs/
```

Проверяется, что каталоги успешно создаются, в том числе и вложенные. Команда ls -l должна корректно отображать их как каталоги (d...) с правами 0755.

Создание файлов (touch, echo):

```
touch mnt/docs/file1.txt
echo "hello world" > mnt/docs/work/report.txt
```

```
ls -l mnt/docs/work/
```

Проверяется создание пустого файла (touMch) и файла с содержимым (echo). Команда ls -l должна отображать report.txt как файл (- ...) с правами 0644 и корректным размером (12 байт).

Удаление файлов и каталогов (rm, rmdir):

```
rm mnt/docs/file1.txt
rmdir mnt/docs/work
```

Проверяется, что rmdir не может удалить непустой каталог. После удаления содержимого (report.txt) rmdir должна сработать корректно. Проверяется, что после удаления объекты исчезают из вывода ls.

5.3 Тестирование операций чтения, записи и изменения файлов

Этот этап адаптируется для тестирования операций чтения, записи и изменения файлов, которые являются ключевыми для файловой системы.

Чтение данных (cat):

```
cat mnt/docs/work/report.txt
```

Проверяется, что содержимое файла, записанное на предыдущем шаге, считывается и выводится корректно (hello world).

Дозапись в файл (>>):

```
echo " another line" >> mnt/docs/work/report.txt
cat mnt/docs/work/report.txt
```

Проверяется, что дозапись в конец файла работает правильно. cat должен вывести hello world another line. Тестируется, что размер файла в выводе ls -l корректно увеличился.

Запись большого файла (тест на выделение нескольких кластеров):

```
# Создаем файл размером > 4 КБ (размер кластера)
dd if=/dev/urandom of=mnt/largefile.bin bs=1K count=10
ls -l mnt/largefile.bin
```

Проверяется способность системы выделять несколько кластеров для

одного файла и связывать их в цепочку в таблице FAT. Команда ls -l должна показать корректный размер файла (10240 байт).

Копирование файлов (cp):

```
cp mnt/largefile.bin mnt/docs/largefile_copy.bin
diff mnt/largefile.bin mnt/docs/largefile_copy.bin
```

Тестируется комплексная операция, включающая чтение одного файла и одновременную запись другого. Утилита diff используется для проверки побайтного совпадения оригинального файла и его копии, что подтверждает корректность работы read и write.

Демонтирование и проверка целостности:

```
sudo umount mnt
```

После выполнения всех операций файловая система демонтируется. Затем она монтируется снова, и выполняется проверка (ls, cat), что все созданные файлы и каталоги, а также их содержимое, сохранились в файле-образе new_disk.img. Это подтверждает корректную работу fat_destroy и msyMnc.

5.4 Тестирование устойчивости и обработки ошибок

В рамках испытаний была проведена проверка устойчивости программы к нештатным ситуациям и граничным условиям:

Некорректные пути:

```
ls mnt/non_existent_dir/
cat mnt/no_file.txt
mkdir mnt/docs/work
```

Проверяется, что при обращении к несуществующим файлам или каталогам команды возвращают стандартные ошибки ОС (No such file or directory, File exists). Это подтверждает корректную обработку ошибок в find_path_entry.

Переполнение диска:

Создается цикл, который записывает большие файлы до тех пор, пока не закончится место на виртуальном диске (16 МБ).

```
while true; do dd if=/dev/zero of=mnt/fill.$RANDOM bs=1M count=1;
done
```

Проверяется, что при попытке записи после исчерпания свободного места команда dd завершается с ошибкой "No space left on device". Это тестирует корректную обработку ситуации, когда find_free_cluster возвращает 0.

Удаление непустого каталога:

```
mkdir mnt/testdir
touch mnt/testdir/a.txt
rmdir mnt/testdir
```

Проверяется, что rmdir возвращает ошибку "Directory not empty", что подтверждает логику проверки на пустоту в fat_rmdir.

Имена файлов:

Тестируется создание файлов с именами разной длины, с расширениями и без, а также с использованием разных регистров. Проверяется, что все они корректно преобразуются в формат 8.3 и сохраняются.

```
touch mnt/test.
touch mnt/UPPERCASE.TXT
touch mnt/longfilename.longext
ls mnt/
```

Проверяется, как readdir отображает эти имена обратно пользователю (в нижнем регистре, с усечением до 8.3).

Целостность после сбоя:

Для симуляции сбоя процесс fat16_fuse может быть принудительно завершен (kill -9). После этого проверяется состояние файла-образа. Так как mMmap с MAP_SHARED периодически сбрасывает данные на диск, часть данных может сохраниться. Этот тест показывает важность корректного демонтирования для гарантии целостности.

6 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

Данный раздел представляет собой руководство пользователя, предназначенное для освоения разработанного в рамках проекта драйвера файловой системы FAT16. Целью программы является эмуляция

полноценного дискового раздела, отформатированного в FAT16, поверх обычного файла-образа в операционной системе Linux. Приложение позволяет создавать, монтировать и работать с таким виртуальным диском с помощью стандартных системных утилит, что делает его полезным инструментом для разработки, тестирования и изучения принципов работы файловых систем.

6.1 Запуск программы

Перед началом работы с программой убедитесь, что в вашей системе установлены все необходимые компоненты: компилятор C (например, gcc) и пакет для разработки с библиотекой FUSE (например, libfuse-dev в системах на базе Debian/Ubuntu).

Сборка программы:

Компиляция исходного кода в исполняемый файл fat16_fuse выполняется стандартной командой в терминале из каталога с проектом:

```
gcc -Wall fat16_fuse.c -o fat16_fuse `pkg-config fuse --cflags --libs`
```

Эта команда скомпилирует исходный код, связывая его с библиотекой FUSE.

Монтирование файловой системы:

Для работы с драйвером необходимо создать точку монтирования — пустой каталог, через который будет осуществляться доступ к файловой системе.

```
mkdir mnt
```

Запуск и монтирование файловой системы осуществляется следующей командой:

```
./fat16_fuse mnt --image=mydisk.img
```

Разберем аргументы: ./fat16_fuse: исполняемый файл нашего драйвера. mnt: путь к точке монтирования (каталогу, который мы создали). --image=mydisk.img: обязательный параметр, указывающий путь к файлу, который будет использоваться как образ диска. Сценарии первого запуска: Если файл mydisk.img не существует: Программа автоматически создаст его, отформатирует под файловую систему FAT16 размером 16 МБ и смонтирует. Вы увидите сообщение о форматировании. Если файл mydisk.img уже

существует: Программа откроет его как существующий диск и смонтирует, предоставляя доступ к ранее сохраненным на нем данным. После успешного запуска программа будет работать в фоновом режиме. Терминал, из которого был произведен запуск, будет занят процессом FUSE. Для работы с файловой системой необходимо открыть новый терминал.

6.2 Просмотр смонтированного каталога

"Просмотр устройств" — это просмотр содержимого смонтированного каталога. Для этого используются стандартные команды Linux. Чтобы увидеть список файлов и каталогов в корне нашего виртуального диска, выполните:

```
ls -la mnt/
```

Эта команда выведет список содержимого, права доступа, размеры и другую информацию, точно так же, как для любого другого каталога в системе.

6.3 Базовые операции по созданию объектов

Рассмотрим базовые операции по созданию объектов.
Создание каталога:

```
mkdir mnt/documents
```

Эта команда создаст новый каталог с именем documents на нашем виртуальном диске.
Создание файла:
Для создания пустого файла используется команда touch:

```
touch mnt/documents/report.txt
```

Для создания файла с содержимым можно использовать перенаправление вывода:

```
echo "Это " > mnt/documents/report.txt
```

6.4 Просмотр содержимого файлов

Просмотр содержимого файла:

```
cat mnt/documents/report.txt
```

Команда cat выведет в консоль содержимое файла report.txt.

Копирование файлов: Файлы можно копировать как на виртуальный диск, так и с него, с помощью стандартной утилиты cp:

```
# Копирование файла из домашнего каталога на наш диск
cp ~/some_document.pdf mnt/

# Копирование файла внутри диска
cp mnt/documents/report.txt mnt/documents/report_backup.txt
```

6.5 Работа с большими файлами

Работа с большими файлами, которая тестирует механизм выделения множества кластеров.

Запись большого файла: Можно использовать утилиту dd для создания файла заданного размера, например, 5 МБ:

```
dd if=/dev/urandom of=mnt/large_data_file.bin bs=1M count=5
```

Эта команда создаст на нашем виртуальном диске файл large_data_file.bin размером 5 мегабайт, заполненный случайными данными. Выполнение этой команды позволяет проверить, как файловая система справляется с данными, размер которых превышает один кластер (4 КБ).

6.6 Управление и завершение работы

Демонтирование файловой системы:

Для корректного завершения работы и сохранения всех изменений на диск необходимо демонтировать файловую систему. Это делается с помощью стандартной утилиты umount. Важно: демонтирование должно выполняться с правами суперпользователя (sudo).

```
sudo umount mnt
```

После выполнения этой команды процесс fat16_fuse в другом терминале завершится. Функция fat_destroy будет вызвана автоматически, чтобы записать все изменения из памяти в файл-образ mydisk.img. Это

гарантирует целостность данных. Принудительное завершение (не рекомендуется): Если завершить процесс fat16_fuse принудительно (например, через kill -9 или закрыв терминал), данные могут быть сохранены не полностью, что может привести к повреждению файловой системы на образе диска. Всегда используйте umount для штатного завершения работы.

ЗАКЛЮЧЕНИЕ

В ходе выполнения курсового проекта был разработан пользовательский драйвер файловой системы FAT16 для операционной системы Linux, функционирующий на основе технологии FUSE (Filesystem in Userspace). В работе были изучены и применены подходы к реализации стандартных операций файловой системы, включая управление дисковым пространством, работу с иерархической структурой каталогов и обработку данных файлов. Использование механизма отображения файла-образа в память (mMmap) позволило создать производительное решение, минимизирующее накладные расходы на операции дискового ввода-вывода.

Реализация программы на языке C с активным использованием библиотеки FUSE (fuse.h) обеспечила корректное взаимодействие с виртуальной файловой системой (VFS) ядра операционной системы. Ключевые функции, определенные в структуре fuse_operations (такие как getattr, readdir, mkdir, read, write, unlink), позволили транслировать стандартные системные вызовы в логику управления нашей файловой системой. Проведённые испытания с помощью стандартных утилит командной строки (ls, cp, rm, mkdir) подтвердили корректную работу всех заявленных режимов и целостность данных после цикла монтирования-демонтирования.

Практическая значимость проекта заключается в предоставлении наглядного и функционального примера реализации файловой системы. Программа может использоваться в образовательных целях для изучения фундаментальных принципов организации данных на диске, таких как работа с таблицей размещения файлов (FAT), управление кластерами и структурами каталогов. Также она может служить основой или прототипом для разработки специализированных файловых систем для встраиваемых устройств или для решения прикладных задач, требующих кастомной логики хранения данных.

В целом, поставленные в работе цели и задачи были успешно достигнуты. Полученные результаты и опыт могут быть использованы для дальнейшего развития проекта, например, для добавления поддержки длинных имен файлов (LFN), реализации механизмов синхронизации для обеспечения полной потокобезопасности в высоконагруженных сценариях или внедрения более сложных алгоритмов кэширования и распределения дискового пространства.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] FUSE (Filesystem in Userspace) Documentation [Электронный ресурс]. URL: https://www.kernel.org/doc/html/latest/filesystems/fuse.html (Дата обращения: 20.05.2024).

[2] A FAT filesystem specification [Электронный ресурс] // OSDev.org Wiki. URL: https://wiki.osdev.org/FAT (Дата обращения: 20.05.2024).

[3] Роберт Лав. Linux. Системное программирование. 2-е изд. – СПб.: Питер, 2014. – 448 с.

[4] Керниган Б., Ритчи Д. Язык программирования С. 2-е изд. – М.: Вильямс, 2015. – 304 с.

[5] mmap(2) — Linux manual page [Электронный ресурс] // The Linux man-pages project. URL: https://man7.org/linux/man-pages/man2/mmap.2.html (Дата обращения: 20.05.2024).

[6] Бовет Д., Чезати М. Ядро Linux. 3-е изд. – СПб.: БХВ-Петербург, 2007. – 1104 с.

ПРИЛОЖЕНИЕ А

Схема алгоритма

ПРИЛОЖЕНИЕ Б

Ведомость документов