

Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные среды и системное программирование

К защите допустить:

И.О. Заведующего кафедрой
информатики

_____ С. И. Сиротко

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

**ПРОСТАЯ ФАЙЛОВАЯ СИСТЕМА (SFS) В ПРОСТРАНСТВЕ
ПОЛЬЗОВАТЕЛЯ.**

БГУИР КП 1–40 04 01 012 ПЗ

Студент

Н. С. Сенько

Руководитель

Н. Ю. Гриценко

Нормоконтролер

Н. Ю. Гриценко

Минск 2025

СОДЕРЖАНИЕ

Введение.....	6
1 Архитектура вычислительной системы.....	7
1.1 История, версия и достоинства.....	7
1.2 Файловая система Linux.....	8
1.3 Виртуальные файловые системы (VFS).....	14
1.4 FUSE.....	16
2 Платформа программного обеспечения.....	18
2.1 Язык программирования C++.....	18
2.2 Micro.....	21
2.3 Arch Linux.....	22
3 Теоретическое обоснование разработки программного продукта.....	25
4 Проектирование функциональных возможностей программы.....	27
4.1 Обоснование и описание функциональных возможностей.....	27
5 Архитектура разрабатываемой программы.....	29
5.1 Общая структура программы.....	29
5.2 Функциональная схема программы.....	29
5.3 Блок-схема алгоритма.....	30
Список литературных источников.....	34

ВВЕДЕНИЕ

Целью данной работы является разработка и реализация простой файловой системы (*SFS*), функционирующей в пространстве пользователя, что позволит обеспечить гибкость управления данными и повысить безопасность за счёт изоляции от привилегированных компонентов операционной системы.

Файловая система – это программный компонент, отвечающий за организацию хранения, доступа и управления данными на носителях информации. В отличие от традиционных файловых систем, работающих на уровне ядра ОС, реализация *SFS* в пользовательском пространстве минимизирует риски нарушения стабильности системы, упрощает отладку и предоставляет разработчикам возможность экспериментировать с алгоритмами без модификации низкоуровневых механизмов. Такие системы особенно актуальны в контексте встраиваемых решений, образовательных проектов и специализированных приложений, требующих изолированного управления файлами.

В рамках данной работы будет проведено исследование архитектурных принципов файловых систем, проектирование *SFS* с поддержкой базовых операций (создание, чтение, запись, удаление файлов и директорий), а также её практическая реализация с использованием технологий пользовательского пространства, таких как *FUSE* (*Filesystem in Userspace*). Особое внимание уделено оптимизации метаданных, механизмам распределения памяти и обеспечению целостности данных.

Важным аспектом работы является интеграция *SFS* с интерфейсом *FUSE*, что позволяет абстрагироваться от сложностей взаимодействия с ядром ОС и сосредоточиться на логике работы системы. Это обеспечивает переносимость решения и упрощает его адаптацию для различных платформ.

Для оценки эффективности разработанной системы будет проведено тестирование, включающее измерение производительности при выполнении операций ввода-вывода, анализ задержек и сравнение с аналогичными решениями. Тестирование на различных типах данных (текстовые файлы, бинарные данные, множественные мелкие файлы) позволит определить устойчивость *SFS* к нагрузкам и её применимость в реальных сценариях. Результаты исследования послужат основой для выводов о целесообразности использования пользовательских файловых систем в современных вычислительных средах, а также обозначат направления для дальнейшей оптимизации и расширения функциональности.

Таким образом, работа вносит вклад в развитие методов проектирования файловых систем, демонстрируя возможности их реализации в условиях ограниченных привилегий, и предоставляет инструмент для образовательных и исследовательских задач в области управления данными.

1 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

1.1 История, версия и достоинства

Linux – свободно распространяемая операционная система, относящаяся к семейству *UNIX*, первоначально разработанная Линусом Торвальдсом в 1991 году. Её создание стало результатом коллективных усилий сообщества программистов и энтузиастов, объединившихся через интернет для совместной разработки. Важным отличием *Linux* от других *UNIX*-подобных систем является отсутствие в её ядре кодов *AT&T* или иных проприетарных компонентов. Основу пользовательского пространства *Linux* составляют инструменты, созданные в рамках проекта *GNU (Free Software Foundation, Cambridge, Massachusetts)*, что подчёркивает её открытость и соответствие принципам свободного программного обеспечения [1].

Ядро *Linux*, распространяемое под лицензией *GNU GPL (General Public License)*, поддерживает широкий спектр технологий: от графической подсистемы *X Window* до сетевых протоколов *TCP/IP*, а также включает компиляторы *GNU C/C++* и инструменты для работы с текстом, такие как *TeX*. Это обеспечивает гибкость и адаптивность системы, позволяя использовать её в разнообразных вычислительных средах.

Ранние этапы разработки *Linux* отличались экспериментальным характером. Версия 0.01 (сентябрь 1991 г.) не предоставляла полноценного исполняемого кода, а требовала наличия ОС *Minix* для компиляции и модификации. Первая официальная версия 0.02, анонсированная 5 октября 1991 года, фокусировалась исключительно на развитии ядра, игнорируя вопросы документации, пользовательского интерфейса и распространения. Подобный подход отражал приоритет сообщества: на первом этапе стояла задача создания стабильного ядра, а вспомогательные аспекты считались вторичными.

Эволюция версий ядра демонстрирует рост проекта. После версии 0.03 нумерация скачком перешла к 0.10 из-за увеличения числа участников разработки. К марту 1992 года, с выходом версии 0.95, Линус Торвальдс обозначил близость к релизу, однако стабильная версия 1.0 была выпущена только в 1994 году. К декабрю 1993 года ядро достигло версии *0.99.pl14*, а на момент написания текста актуальной являлась версия 6.13.3, что иллюстрирует непрерывное развитие системы.

Современный *Linux* – полнофункциональная ОС, совместимая с ключевыми стандартами и технологиями, включая сетевые протоколы, среды разработки (*Emacs*), почтовые системы и инструменты взаимодействия с другими ОС, такими как *Windows*. Благодаря открытой модели разработки, для *Linux* доступны как свободные, так и коммерческие программные пакеты, что делает её универсальной платформой для научных, корпоративных и персональных задач. Сегодня *Linux* сохраняет статус одного из наиболее значимых проектов в истории open-source, сочетая инновации с преемственностью принципов *UNIX*.

1.2 Файловая система *Linux*

Файловая система (ФС) является важной частью любой операционной системы, которая отвечает за организацию хранения и доступа к информации на каких-либо носителях. Рассмотрим в качестве примера файловые системы для наиболее распространенных в наше время носителей информации – магнитных дисков. Как известно, информация на жестком диске хранится в секторах (обычно 512 байт) и само устройство может выполнять лишь команды считать/записать информацию в определенный сектор на диске. В отличие от этого файловая система позволяет пользователю оперировать с более удобным для него понятием – файл. Файловая система берет на себя организацию взаимодействия программ с файлами, расположенными на дисках. Для идентификации файлов используются имена. Современные файловые системы предоставляют пользователям возможность давать файлам достаточно длинные мнемонические названия.

Под каталогом в ФС понимается, с одной стороны, группа файлов, объединенных пользователем исходя из некоторых соображений, с другой стороны каталог – это файл, содержащий системную информацию о группе составляющих его файлов. Файловые системы обычно имеют иерархическую структуру, в которой уровни создаются за счет каталогов, содержащих информацию о файлах и каталогах более низкого уровня. Рассмотрим более подробно структуру жесткого диска. Базовой единицей жесткого диска является раздел, создаваемый во время разметки жесткого диска. Каждый раздел содержит один том, обслуживаемый какой-либо файловой системой и имеющий таблицу оглавления файлов – корневой каталог. Некоторые операционные системы поддерживают создание томов, охватывающих несколько разделов. Жесткий диск может содержать до четырех основных разделов. Это ограничение связано с характером организации данных на жестких дисках *IBM*-совместимых компьютеров. Многие операционные системы позволяют создавать, так называемый, расширенный (extended) раздел, который по аналогии с разделами может разбиваться на несколько логических дисков. [2]

Файловая система *Linux* [3] организована в форме иерархической структуры, которую образно можно представить в виде дерева. Она содержит в себе каталоги и подкаталоги, образуя таким образом вложенную структуру. Все файлы и каталоги начинаются от корневого каталога, который обозначается символом “/”, а далее распределяются по отдельным ветвям и листьям этого дерева. Таким образом создается логическая и удобная организация данных.

Linux придерживается стандартов иерархии файловой системы (*Filesystem Hierarchy Standard, FHS*). Эти стандарты определяют основные правила организации и содержания каталогов в системах, аналогичных *UNIX*. Они обеспечивают единообразие в структуре файловой системы, делая

работу с разными дистрибутивами более предсказуемой и понятной для пользователей и разработчиков.

Система *FHS* создана рабочей группой *Linux Standard Base* с целью решения двух задач:

- 1 Унификация инфраструктуры каталогов.
- 2 Обеспечение совместимости разных дистрибутивов.

FHS определяет такие ключевые каталоги:

- 1 */bin* для исполняемых файлов;
- 2 */home* содержит личные папки пользователей, предоставляя каждому свое пространство для файлов;
- 3 */etc* служит для размещения настроек и конфигурационных файлов системы;
- 4 */var* для переменных данных, среди прочих.

Такая структура облегчает навигацию по системе, стандартизацию программ и скриптов, а также упрощает управление файлами и каталогами.

Рассмотрим самые известные файловые системы для Linux:

1 *Ext (Extended File System)*. Является первой файловой системой, разработанной специально для *Linux*. Она представляла собой значительный шаг вперед по сравнению с предыдущими решениями, предоставляя лучшую производительность и возможности для *Linux*-систем.

2 *Ext2 (Second Extended File System)*. Была разработана как улучшение *ext*, предлагая лучшую надежность и управление ресурсами, а также поддержку большего размера данных и файлов. Эта система не использует журналирование, что делает ее менее предпочтительной для проектов, где важна высокая надежность данных, но по-прежнему эффективной для портативных устройств хранения данных.

3 *Ext3 (Third Extended File System)*. Улучшенная версия *ext2*. Основным нововведением здесь является поддержка журналирования, что значительно повышает надежность и уменьшает время на восстановление после сбоев или некорректных выключений системы.

4 *Ext4 (Fourth Extended File System)*. Новейшее развитие в линейке *Ext*. Обладает улучшенной эффективностью, надежностью и масштабируемостью. Предлагает поддержку больших объемов хранения, а также ряд других технических усовершенствований.

5 *XFS*. Высокопроизводительная и масштабируемая файловая система от компании *SGI*. Известна своей способностью эффективно работать с большими файлами и обширными наборами данных, что делает ее популярным выбором для серверов и систем хранения данных.

6 *JFS (Journaled File System)*. Создана компанией *IBM*, обладает высокой надежностью и эффективным использованием ресурсов, обеспечивая хорошую производительность даже при повышенных нагрузках.

7 *BTrFS (B-Tree File System)*. Разработана *Oracle* для повышения гибкости управления данными и обеспечения высокого уровня отказоустойчивости. Она включает в себя такие возможности, как проверка и

восстановление данных на лету, эффективное сжатие и интеграцию множественных устройств в одну файловую систему.

8 *Swap*. Позволяет системе использовать часть дискового пространства в качестве виртуальной памяти, когда физическая память (*RAM*) исчерпана. Это не файловая система в традиционном понимании, но она является важным элементом управления памятью *Linux*.

Linux обладает уникальной способностью адаптироваться к различным требованиям и сценариям использования благодаря поддержке большого количества файловых систем. Такое разнообразие не только обогащает экосистему *Linux*, но и способствует инновациям, предоставляя пользователям свободу выбора. От стабильности и надежности, предлагаемых семейством Ext, до способности *XFS*, *JFS* и *Btrfs* управлять большими наборами данных и обеспечивать высокую производительность – каждая файловая система вносит свой вклад в адаптивность и возможности экосистемы. Это также отражает дух сотрудничества в сообществе *Linux*, где вклад независимых разработчиков способствует непрерывному развитию и инновациям.

В отличие от других ОС, где разные диски и разделы обладают собственными корневыми директориями, в *Linux* они монтируются в поддиректории внутри единой файловой иерархии.

В *Linux* есть несколько разновидностей файлов, которые выполняют свою уникальную функцию:

1 Обычные файлы (*regular files*). Это самый обычный тип файлов, который чаще всего используется. Сюда относятся данные, текст, исходный код программ, медиаматериалы и прочее.

2 Именованные каналы (*named pipes*). Необходимы для межпроцессного взаимодействия, позволяя одному процессу передавать данные другому.

3 Файлы устройств. Содержат в себе символьные (*char devices*) и блочные (*block devices*) файлы, которые предоставляют внешние аппаратные устройства (например, *HDD*, принтеры и прочие).

4 Ссылки. Включают два типа ссылок. Символические ссылки, или «симлинки», функционируют как ярлыки, указывающие на другие файлы или папки. Жесткие ссылки, в свою очередь, создают альтернативные пути доступа к одним и тем же физическим данным на диске, ведя себя как дубликаты файла без фактического дублирования содержимого.

5 Каталоги. Это своего рода папки, где хранятся ссылки на файлы и другие каталоги. Они помогают организовать данные, распределяя их по разным «отсекам», чтобы было легче найти нужную информацию.

6 Сокеты. Специальные файлы для обмена данными между разными процессами, как внутри одной системы, так и между разными компьютерами. Это своеобразные «почтовые ящики» для программ, через которые они могут «пересылать» друг другу информацию.

7 Двери (*Doors*). Механизм в некоторых операционных системах, предназначенный для взаимодействия между программными процессами.

В Linux каждый тип файла играет свою роль в обширной системе управления данными и процессами, придавая системе уникальную гибкость и мощь. Например, благодаря тому, что внешние устройства представлены в виде специальных файлов, взаимодействие с ними можно осуществлять стандартными методами работы с файлами.

В Linux каждая папка под корневым каталогом "/" служит определенной цели:

1 */home* – домашние пользовательские каталоги, которые нужны для хранения их персональных файлов и настроек.

2 */bin* и */sbin* – предназначены для хранения исполняемых файлов, то есть программ и команд. */bin* доступен для всех пользователей, а */sbin* содержит команды для администрирования и доступен только суперпользователю.

3 */lib*, */lib32*, */lib64* – служат для размещения библиотек, которые необходимы для нормальной работы программ и системных утилит, расположенных в */bin* и */sbin*.

4 */opt* – используется для размещения дополнительных программ, которые обычно устанавливаются из внешних источников, не входящих в стандартный набор дистрибутива.

5 */usr* – один из наиболее объемных каталогов, где находится большая часть пользовательского софта и библиотек. Структура похожа на корневой каталог с поддиректориями, например, */usr/bin*, */usr/lib*, */usr/local* и т.д.

6 */boot* – хранит файлы, которые нужны для процесса загрузки операционной системы.

7 */sys* – предоставляет интерфейс к аппаратному обеспечению, отражая состояние устройств и драйверов.

8 */tmp* – служит для хранения временных файлов, которые создаются системой и различными приложениями во время их выполнения.

9 */dev* – хранит специальные файлы, которые представляют собой аппаратные и виртуальные устройства, позволяя программам взаимодействовать с ними.

10 */run* – содержит сведения о системе, актуальные с момента последнего включения.

11 */root* – это специальная личная папка суперпользователя.

12 */proc* – это не реальная, а виртуальная файловая система, через которую доступна информация о работающих процессах и общем состоянии системы.

13 */srv* – предназначен для информации сервисов от системы, например, веб-сайтов и *FTP*.

Такая организация файловой системы обеспечивает четкую и логичную расстановку всех элементов, где каждый каталог и файл служит определенной цели. Это упрощает навигацию и использование системы, а также способствует эффективному управлению ресурсами.

Через терминал в Linux доступен обширный арсенал команд для работы с файлами и каталогами.

К основным из них можно отнести следующие:

1 *ls* – позволяет просмотреть, что находится внутри папки. Можно использовать с различными опциями, например, *ls -l* для подробного списка.

2 *mkdir* – можно использовать для создания новой папки.

3 *cat* – выводит текст файла прямо в окно терминала.

4 *less* – дает возможность пролистывать содержимое указанного файла по страницам.

5 *touch* – позволяет инициировать создание нового файла без содержимого, при условии, что файл с таким именем еще не существует.

6 *rm* – удаляет указанный файл.

7 *rm -r* – удаляет каталог и все его содержимое рекурсивно.

8 *ln -s* – формирует символическую ссылку, указывающую на выбранный файл или директорию.

9 *pwd* – выводит текущее местоположение пользователя в структуре каталогов.

10 *which* – показывает полный путь к исполняемому файлу программы.

13 *cd* – изменяет текущую директорию на указанную.

14 *cp* – копирует файлы из одного места в другое.

15 *nano* – открывает текстовый редактор Nano для редактирования файла.

16 *mv* – может как переместить, так и переименовать файлы или папки.

17 *locate* – помогает быстро найти файлы, соответствующие заданному шаблону.

Эти команды можно считать некой основой для взаимодействия с файловой системой Linux. При необходимости их можно комбинировать или расширять с помощью различных опций для выполнения сложных задач.

В *Linux* организация данных на устройствах хранения, таких как жесткие диски и *SSD*, осуществляется через сложную структуру. Она содержит в себе блоки данных и индексные узлы (или *inodes*). Когда создается новый файл, система назначает ему один или несколько индексных узлов, в которых сохраняются метаданные этого файла.

Каждый индексный узел представляет собой уникальный идентификатор, необходимый для отслеживания файла в системе. Он хранит все сведения о файле, кроме его имени и фактического содержимого.

Фактическое содержимое файла содержится в блоках данных. Их размер определяется при организации файловой системы. А в дальнейшем она контролирует размещение этих блоков на физическом носителе, оптимизируя доступ к информации и минимизируя фрагментацию.

Кроме того – файловая система отвечает за эффективное использование дискового пространства, размещая файлы таким образом, чтобы снизить задержку при доступе и уменьшить фрагментацию данных. Это также предполагает использование алгоритмов для определения наилучшего расположения файлов и папок на диске с учетом текущего использования и доступного пространства.

Такая организация повышает гибкость и производительность системы, позволяя ей быстро находить и обрабатывать данные. Это также облегчает восстановление и управление файлами благодаря четкой компоновке индексных узлов и блоков данных.

Архитектура *Linux* представляет собой модульную и слоистую структуру, которая состоит из различных компонентов, взаимодействующих друг с другом для обеспечения полноценной работы.

Основные элементы этой архитектуры включают:

1 Ядро (*Kernel*). Является основным компонентом системы, контролирует аппаратные ресурсы, управляет памятью и процессами, а также обрабатывает системные вызовы. Ядро действует как посредник между аппаратным и программным обеспечением на уровне пользовательских приложений.

2 Системные библиотеки. Предоставляют набор стандартизированных функций и интерфейсов, которые упрощают разработчикам создание программ, позволяя им использовать общие решения для стандартных задач.

3 Системные утилиты. Содержат в себе базовые инструменты и программы, которые нужны для управления, настройки и мониторинга системы.

4 Пользовательские приложения. Охватывают широкий спектр программного обеспечения от сторонних разработчиков, включая графические среды рабочего стола, офисные пакеты, браузеры, игры и множество других приложений.

Архитектура *Linux* строится вокруг концепции разделения привилегий, в рамках которой действует два основных режима работы:

1 Пользовательский режим (*user mode*). В этом режиме работают все пользовательские приложения с ограниченными правами. Таким образом достигается безопасность системы.

2 Режим ядра (*kernel mode*). Этот режим дает полный доступ к аппаратным ресурсам и критически важным компонентам системы. Применяется ядром и драйверами устройств для выполнения задач, требующих повышенных привилегий.

При этом файловая система играет ключевую роль в архитектуре *Linux*, предоставляя единую иерархическую структуру для размещения данных на диске. Она обеспечивает логическое размещение файлов и папок, предоставляет простой доступ к ним и упрощает управление.

В дополнение к традиционным файловым системам в *Linux* часто применяются виртуальные и специализированные файловые системы, которые предназначены для конкретных задач.

Вот некоторые из них:

1 *EncFS* (*Encrypted File System*). Файловая система на основе *FUSE* (*Filesystem in Userspace*), которая предназначена для прозрачного шифрования файлов.

2 *AUFS* (*Another Union File System*). Слоистая файловая система, позволяющая объединять содержимое нескольких каталогов в один

виртуальный каталог. Таким образом обеспечивается удобное управление директориями из различных источников. Эта система часто применяется в *Live CD* и *Docker*-контейнерах.

3 *NFS (Network File System)*. Сетевая файловая система, которая дает возможность пользователям работать с файлами и папками на удаленных компьютерах, так, как будто они находятся непосредственно на их собственном устройстве.

4 *ZFS (Zettabyte File System)*. Изначально разработана для *Solaris (Sun Microsystems)*, а сейчас доступна и для некоторых дистрибутивов *Linux* (через проект *OpenZFS*). *ZFS* известна своими возможностями в области управления хранилищем, включая поддержку высокого уровня интеграции данных и пула хранения, снимки состояния, клонирование и встроенное шифрование.

Эти системы и технологии дают дополнительные возможности для достижения эффективности, надежности и гибкости в управлении данными и хранилищами в *Linux*. Они оптимизированы под конкретные сценарии использования и их можно выбирать с учетом требований к системе хранения данных.

1.3 Виртуальные файловые системы (*VFS*)

VFS – это важный уровень абстракции в современных операционных системах, предназначенный для облегчения взаимодействия между различными файловыми системами и пользовательскими приложениями. *VFS* функционирует как посредник, позволяя приложениям получать доступ к различным файловым системам через единый интерфейс, независимо от специфических характеристик или структуры базовой файловой системы [4]. *VFS* управляет различными подключенными файловыми системами, поддерживая структуры данных, которые описывают всю виртуальную файловую систему и реальные подключенные файловые системы. В *VFS* используются суперблоки и индексные дескрипторы, аналогично файловой системе *EXT2*, для описания файлов и каталогов в системе. Каждая файловая система регистрируется в *VFS* во время инициализации операционной системы. Модули файловой системы загружаются по мере необходимости, что позволяет *VFS* считывать их суперблоки.

Определения функций, которые принадлежат к базовым типам *VFS*, находятся в файлах *fs/*.c* исходного кода ядра, в то время как подкаталоги *fs/* содержат определенные файловые системы. В ядре также содержатся сущности, такие как *cgroups*, */dev* и *tmpfs*, которые требуются в процессе загрузки и поэтому определяются в подкаталоге ядра *init/*. Заметьте, что *cgroups*, */dev* и *tmpfs* не вызывают «большую тройку» функций *file_operations*, а напрямую читают и пишут в память.

На приведенной ниже диаграмме показано, как *userspace* обращается к различным типам файловых систем, обычно монтируемых в системах *Linux*. Не показаны такие конструкции как *pipes*, *dmesg* и *POSIX clocks*, которые

также реализуют структуру *file_operations*, доступ к которым проходит через слой *VFS*.

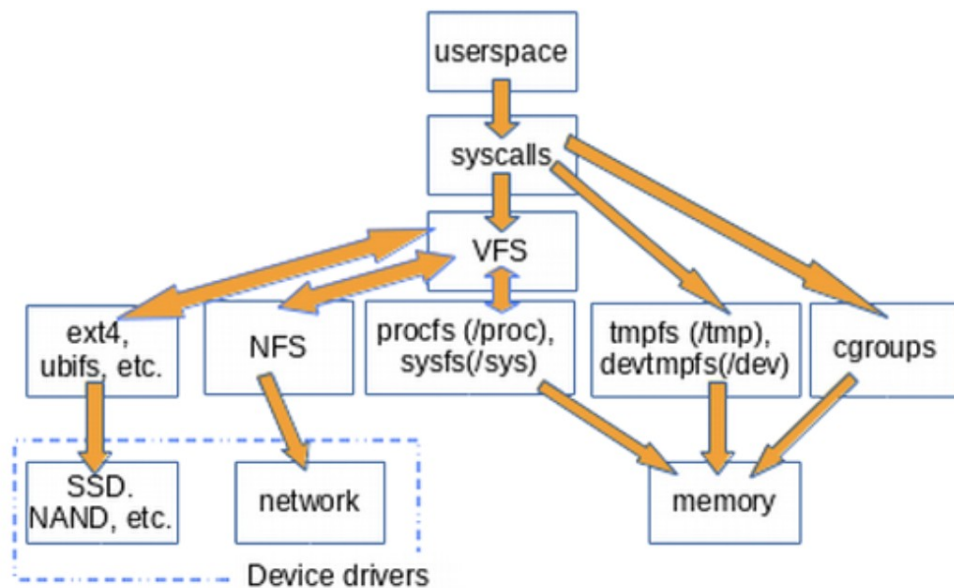


Рисунок 1.1 – Схема общения userspace к различным типам файловых систем

VFS также играет важную роль в разработке операционной системы, поскольку она не только обеспечивает независимость от файловой системы, но и обеспечивает такие преимущества, как повышенная безопасность или производительность. В этой статье мы также продемонстрируем, что расширяемость *VFS* является ценным преимуществом, позволяющим внедрять расширенные функции без необходимости внесения изменений в ядро.

1.3.1 Операции *VFS*

VFS предоставляет ряд операций для взаимодействия с файлами и каталогами. Эти операции реализованы в виде системных вызовов, таких как *open(2)*, *stat(2)*, *read(2)*, *write(2)* и *chmod(2)*, которые вызываются из контекста процесса.

Способ, которым это реализовано в Linux *VFS*, описан в статье Ричарда Гуч и Пекка Энберг[3]. Во время системного вызова *VFS* преобразует имя пути в запись каталога (*dentry*) путем поиска в кэше *dentry*. Однако, если кэш слишком мал, чтобы вместить все данные в оперативной памяти, *VFS*, возможно, придется создать *dentry* и загрузите индексные узлы, чтобы определить путь к ним. Каждый *dentry* обычно имеет указатель на индексный узел, который представляет объект файловой системы, который может находиться на диске или в памяти.

Когда файл открывается, файловая структура выделяется и инициализируется указателем на функции-члены *dentry* и *file operation*,

которые берутся из данных *inode*. Затем вызывается файловый метод *open()*, позволяющий реализации файловой системы выполнять свою работу. Наконец, файловая структура помещается в таблицу файловых дескрипторов для процесса. Для чтения, записи и закрытия файлов используется файловый дескриптор пользовательского пространства, который вызывает соответствующий метод файловой структуры. Пока файл открыт, соответствующие индексы *dentry* и *VFS* остаются в использовании.

1.4 FUSE

FUSE – свободный 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.

Файловые системы можно разрабатывать как *user space*-программы, без необходимости писать код в ядре ОС. *FUSE* поддерживает непривилегированное монтирование ФС, права супер-пользователя не нужны для запуска файловой системы.

Это значит, что на *FUSE* быстрее разрабатывать и проще отлаживать. Рассмотрим фреймворк подробнее.

Файловая система с точки зрения *FUSE* – это *daemon* в *user space*, его называют драйвером ФС. Файловые системы в *Linux* реализуют одинаковый интерфейс. За это отвечает *vfs* – подсистема ядра *Linux*. *Vfs* выполняет роль маршрутизатора: направляет запросы клиентов к экземплярам ФС [5].

FUSE-модуль ядра:

- получает запрос от *vfs*;
- направляет его драйверу ФС;
- ждёт от драйвера ответа;
- возвращает результат обратно в *vfs*.

Этот процесс показан на рисунке ниже.

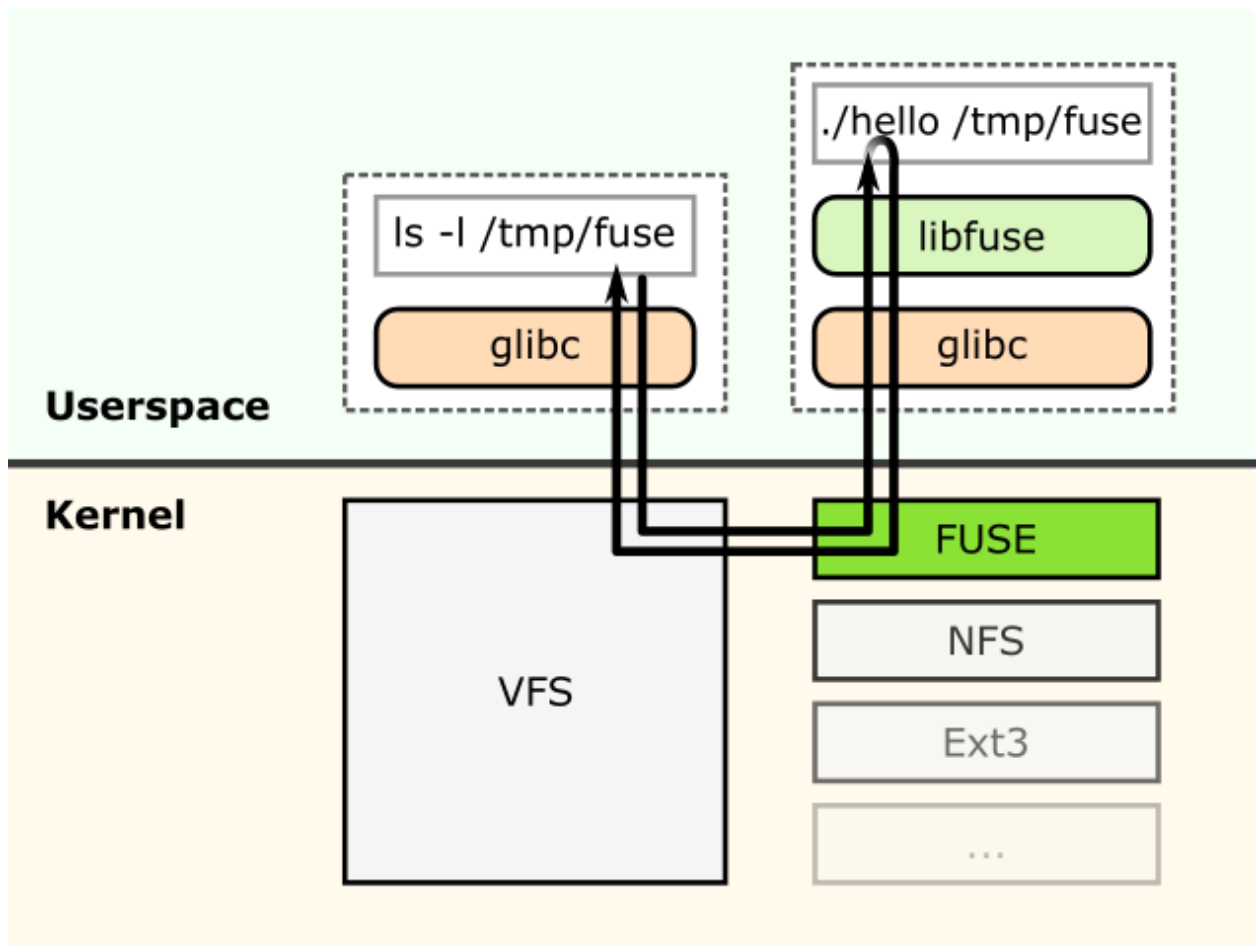


Рисунок 1.1 – Схема работы фреймворка FUSE

FUSE-модуль ядра ОС и драйвер ФС общаются по *RPC*-протоколу. Драйвер устанавливает соединение с *FUSE*-модулем ядра ОС в процессе монтирования, соединения устанавливается через устройство `/dev/fuse`. По *FUSE*-соединению ходят пакеты в режиме запрос-ответ. Порядок ответов драйвера ФС может отличаться от порядка, в котором были отправлены запросы. Каждый ответ ссылается на запрос по *ID*, что позволяет драйверу ФС обрабатывать запросы параллельно.

Рассмотренные аспекты архитектуры Linux, файловых систем, VFS и FUSE демонстрируют ключевые принципы, лежащие в основе гибкости, надежности и универсальности этой операционной системы. Архитектура Linux, сочетающая модульность, стандартизацию и открытость, создает фундамент для инноваций. Файловые системы, VFS и FUSE иллюстрируют, как гибкие абстракции и сообщество разработчиков превращают Linux в универсальную платформу, способную решать задачи любой сложности.

2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Язык программирования C++

Язык программирования для реализации данного курсового проекта был выбран C++.

2.1.1 История

C – процедурный язык программирования, разработанный в 1969–1973 годах сотрудниками *Bell Labs* Кеном Томпсоном и Деннисом Макалистэйр Ритчи как развитие языка Би. C был создан для использования в операционной системе *UNIX*, ключевым разработчиком которой являлся Деннис Ритчи. С тех пор он был перенесён на многие другие операционные системы и стал одним из самых используемых языков программирования.

Деннис Ритчи (9 сентября 1941 – 12 октября 2011) также известен как соавтор классической книги «Язык программирования C», обычно сокращаемой как «K&R» (авторы Керниган и Ритчи).

C++ – объектно-ориентированное расширение языка Си, создан в начале 1980-х годов Бьерном Страуструпом, также сотрудником *Bell Labs*. Вначале он добавил к языку Си возможность работы с классами и объектами, а в 1983 году после добавления большого количества синтаксических конструкций получившийся язык был переименован из Си с классами в «C++».

В 1985 году вышло первое издание «Языка программирования C++», обеспечивающее первое описание этого языка, что было чрезвычайно важно из-за отсутствия официального стандарта. В 1989 году состоялся выход C++ версии 2.0. Его новые возможности включали множественное наследование, абстрактные классы, статические функции-члены, функции-константы и защищённые члены. В 1990 году вышло «Комментированное справочное руководство по C++», положенное впоследствии в основу стандарта. Последние обновления включали шаблоны, исключения, пространства имён, новые способы приведения типов и булевский тип. В качестве основы для хранения и доступа к обобщённым алгоритмам была выбрана Стандартная библиотека шаблонов (*STL*), разработанная Александром Степановым и Менг Ли.

Стандартная библиотека C++ также развивалась вместе с ним. Первым добавлением к стандартной библиотеке C++ стали потоки ввода-вывода, обеспечивающие средства для замены традиционных функций C `printf` и `scanf`. Позднее самым значительным развитием стандартной библиотеки стало включение в неё Стандартной библиотеки шаблонов.

Официальная стандартизация языка началась в 1998 году, когда был опубликован стандарт языка *ISO/IEC 14882:1998* (известный как C++98), разработанный комитетом по стандартизации C++ (*ISO/IEC JTC1/SC22/WG21 working group*). Стандарт C++ не описывал способов именования объектов, некоторых деталей обработки исключений и других

возможностей, связанных с деталями реализации, что делает несовместимым объектный код, созданный различными компиляторами. Однако для этого третьими лицами создано множество стандартов для конкретных архитектур и операционных систем.

В 2003 году опубликован стандарт *C++ ISO/IEC 14882:2003*, где исправлены выявленные ошибки и недочёты предыдущей версии стандарта. В 2005 году опубликован отчёт *Library Technical Report 1* (кратко называемый *TR1*). Не являясь официально частью стандарта, отчёт описывает расширения стандартной библиотеки, которые, по мнению авторов, должны были быть включены в следующую версию стандарта. Степень поддержки *TR1* улучшается почти во всех поддерживаемых компиляторах языка C++.

С 2009 года велась работа по обновлению предыдущего стандарта. Предварительная версия называлась *C++09*, в следующем году её переименовали в *C++0x*. Стандарт был опубликован в 2011 году под названием *C++11*. В него включены дополнения в ядре языка и расширение стандартной библиотеки, в том числе большая часть *TR1*.

Следующая версия стандарта, *C++14*, вышла в августе 2014 года. Она содержит в основном уточнения и исправления ошибок предыдущей версии.

Стандарт *C++17*, опубликованный в декабре 2017 года, включил в стандартную библиотеку параллельные версии стандартных алгоритмов и удалил некоторые устаревшие и крайне редко используемые элементы.

Последняя стабильная на текущий момент действующая версия стандарта – *C++20*. Помимо прочего, он содержит принципиальное новшество – поддержку модулей. Стандарт *C++23* на данный момент активно обсуждается комитетом ISO.

C++ продолжает развиваться, чтобы отвечать современным требованиям. Одна из групп, разрабатывающих язык C++ и направляющих комитету по стандартизации C++ предложения по его улучшению – это Boost, которая занимается, в том числе, совершенствованием возможностей языка путём добавления в него особенностей метапрограммирования.

Никто не обладает правами на язык C++, он является свободным. Однако сам документ стандарта языка (за исключением черновиков) не доступен бесплатно[10]. В рамках процесса стандартизации ISO выпускает несколько видов изданий. В частности, технические доклады и технические характеристики публикуются, когда «видно будущее, но нет немедленной возможности соглашения для публикации международного стандарта.» До 2011 года было опубликовано три технических отчёта по C++: *TR 19768: 2007* (также известный как *C++*, Технический отчёт 1) для расширений библиотеки в основном интегрирован в *C++11*, *TR 29124: 2010* для специальных математических функций, и *TR 24733: 2011* для десятичной арифметики с плавающей точкой. Техническая спецификация *DTS 18822: 2014* (по файловой системе) была утверждена в начале 2015 года, и остальные технические характеристики находятся в стадии разработки и ожидают одобрения.

В марте 2016 года в России была создана рабочая группа РГ21 C++. Группа была организована для сбора предложений к стандарту C++, отправки их в комитет и защиты на общих собраниях Международной организации по стандартизации (ISO)

2.1.3 Достоинства

Статическая типизация. На примере языка со статической типизацией проще понять, что такое тип данных, зачем он нужен и от чего зависит. Видно, что собой представляет объявление, определение и инициализация. Использование языка C++ даёт это явно увидеть, что способствует дальнейшему пониманию того, как работают эти механизмы в других языках. Помимо этого можно на реальных примерах понять, чем беззнаковые целые отличаются от целых со знаком, чем отличаются числа двойной и одинарной точности, чем отличается символ от строки и т.д.

Высокоуровневые и низкоуровневые средства. Использование таких средств, как указатели и динамическое выделение памяти, позволяет понять (или в дальнейшем способствует пониманию), что такое стэк, куча, стэк вызовов, раскрутка стэка и т.д. Помимо этого, на практике закрепляется понимание концепции адресов и адресной арифметики. На примерах демонстрируется, что память надо выделять, освобождать, потому что она не бесконечная, что существуют утечки памяти. В будущем, при изучении языков с GC проще будет понять, что же это такое.

Отдельно стоит отметить простой механизм передачи значений по ссылке, значению, указателю и перенос объекта. Что такое изменяемые и не изменяемые параметры. В дальнейшем данные концепции могут быть использованы и при изучении других языков. Студент будет понимать, например, что объект в языке N передаётся по ссылке, и если его значение изменить в функции-члене, то оно изменится везде.

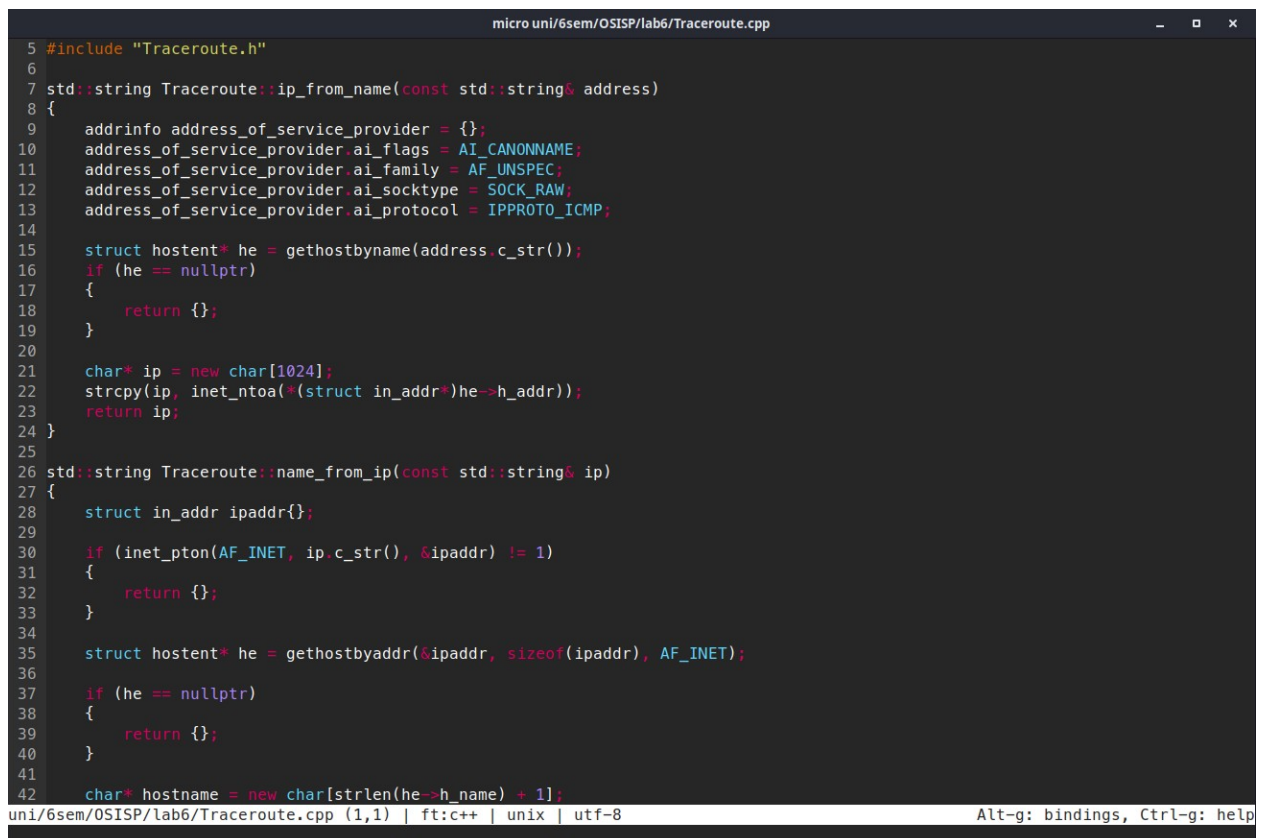
Реализация ООП. Это относительно чистая реализация ООП без всякого синтаксического сахара (относительно некоторых других языков). Чётко разграниченные уровни доступа к членам класса, возможность множественного наследования и динамический полиморфизм дают возможность быстро усвоить основные концепции ООП (абстракция, наследование, инкапсуляция и полиморфизм). Указатели и динамическое выделение памяти позволяют наглядно понять такие важные механизмы, как upcasting и downcasting. В дальнейшем, основываясь на этих знаниях, легко можно понять весь синтаксический сахар в других языках. Необходимость контроля ресурсов (в том числе и «правило трёх» или уже «правило пяти», с учётом C++11), захват их в конструкторе и освобождение в деструкторе также способствуют более глубокому пониманию ООП.

Стоит отметить такой важный момент, как не принудительное ООП. То есть данный подход к программированию применяется тогда, когда это удобно, и его можно смешивать, например, с функциональным программированием. Это способствует формированию понимания того, что средства реализации выбираются исходя из задачи.

STL. Сама по себе концепция шаблонов C++, генерации кода и применения широкого спектра алгоритмов к различным контейнерам положительно влияет на процесс обучения. Здесь все на поверхности и понятно, почему можно создать вектор целых чисел и вектор пользовательских объектов на основе одного класса-контейнера. Почему можно применить некоторую операцию к последовательности объектов или как отсортировать объекты, для которых не предусмотрена встроенная операция сравнения. Можно понять, как осуществляется доступ к элементам, и узнать о категориях итераторов. Помимо этого закрепляется понимание обобщённого программирования.

2.2 *Micro*

Micro – современный и интуитивно понятный текстовый редактор на базе терминала. На рисунке ниже представлен интерфейс.



```
5 #include "Traceroute.h"
6
7 std::string Traceroute::ip_from_name(const std::string& address)
8 {
9     addrinfo address_of_service_provider = {};
10    address_of_service_provider.ai_flags = AI_CANONNAME;
11    address_of_service_provider.ai_family = AF_UNSPEC;
12    address_of_service_provider.ai_socktype = SOCK_RAW;
13    address_of_service_provider.ai_protocol = IPPROTO_ICMP;
14
15    struct hostent* he = gethostbyname(address.c_str());
16    if (he == nullptr)
17    {
18        return {};
19    }
20
21    char* ip = new char[1024];
22    strcpy(ip, inet_ntoa(*(struct in_addr*)he->h_addr));
23    return ip;
24 }
25
26 std::string Traceroute::name_from_ip(const std::string& ip)
27 {
28     struct in_addr ipaddr{};
29
30     if (inet_pton(AF_INET, ip.c_str(), &ipaddr) != 1)
31     {
32         return {};
33     }
34
35     struct hostent* he = gethostbyaddr(&ipaddr, sizeof(ipaddr), AF_INET);
36
37     if (he == nullptr)
38     {
39         return {};
40     }
41
42     char* hostname = new char[strlen(he->h_name) + 1];
```

uni/6sem/OSISP/lab6/Traceroute.cpp (1,1) | ft:c++ | unix | utf-8 Alt-g: bindings, Ctrl-g: help

Рисунок 2.2 – Интерфейс *Visual Studio Code*

Он позиционируется как современная альтернатива классическим редакторам вроде *Nano*, предлагая более комфортный опыт для пользователей, которые предпочитают работать в терминале. *Micro* распространяется в виде единого статического бинарного файла, что упрощает установку и не требует дополнительных зависимостей [6].

Главной особенностью *Micro* является простота установки (это просто статический двоичный файл без зависимостей) и простота в использовании. Высокая степень настраиваемости. Можно использовать простой формат `json` для настройки параметров по своему вкусу. При необходимости можно использовать *Lua* для дальнейшей настройки редактора.

Micro поддерживает более 75 языков и имеет 7 цветовых схем по умолчанию на выбор. *Micro* поддерживает темы 16, 256 и *truecolor*. Синтаксические файлы и цветовые схемы также очень просты в создании.

Micro поддерживает несколько курсоров в стиле *Sublime*, что дает вам широкие возможности для редактирования прямо в вашем терминале.

Система плагинов *Micro* поддерживает полноценную систему плагинов. Плагины написаны на *Lua*, и есть менеджер плагинов, который автоматически загружает и устанавливает ваши плагины за вас. Сочетания клавиш *Micro* такие, какие можно ожидать от простого в использовании редактора.

Micro полностью поддерживает работу с мышью. Это означает, что можно щелкнуть мышью и перетащить ее, чтобы выделить текст, дважды щелкнуть мышью по слову и трижды щелкнуть мышью, чтобы выделить строку.

2.3 Arch Linux

Данный курсовой проект будет реализовываться на *Arch Linux* [7] с оболочкой *Cinnamon*. Скриншот рабочего стола представлен на рисунке 2.3.

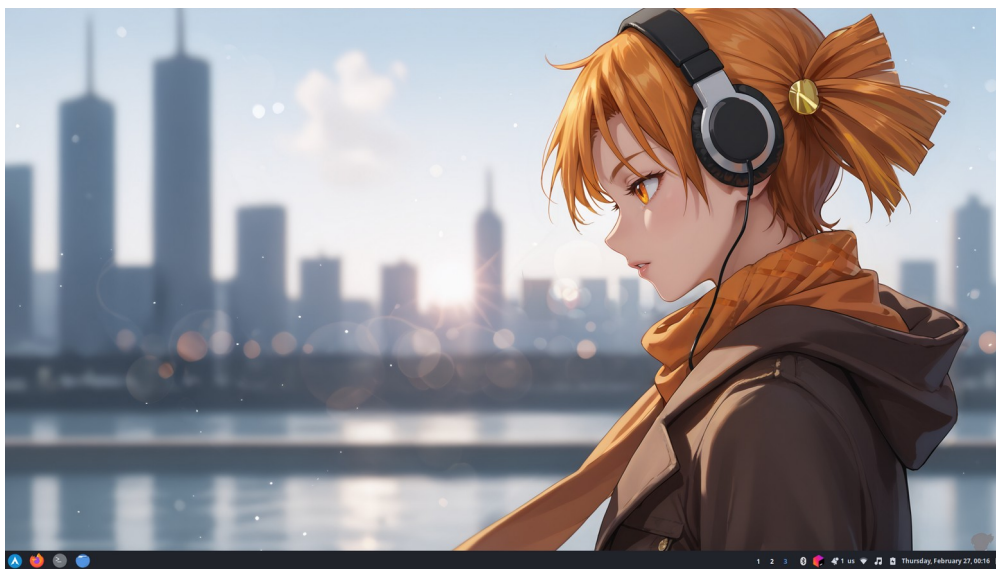


Рисунок 2.3 – Рабочий стол в *Arch Linux Cinnamon*

Arch Linux – это дистрибутив *Linux*, известный своей простотой.

2.3.1 История

Arch Linux – это дистрибутив *Linux*, который был создан в 2002 году Джастином Шлайтом (*Justin Schlytter*). Основной концепцией *Arch* стала простота, гибкость и минимализм, что позволяет пользователям самостоятельно настраивать систему под свои нужды.

Первые версии *Arch Linux* основывались на принципах *KISS* («*Keep It Simple and Stupid*»), или «Делай просто и без излишеств») и философии UNIX, где каждое приложение выполняет одну задачу, но делает это хорошо. Это обеспечивало лёгкость и быструю работу системы, а также открывало широкие возможности для кастомизации.

Одной из ключевых особенностей *Arch Linux* является его система управления пакетами *Pacman*, которая обеспечивает простой и удобный способ установки, обновления и удаления программ. Она стала одним из факторов популярности дистрибутива среди опытных пользователей, которые ценят возможность тонкой настройки своей системы.

В 2013 году разработка *Arch Linux* перешла к новому руководству после того, как основатель проекта Джастин Шлайт отошёл от активной работы над системой. Несмотря на смену руководства, основные принципы и философия *Arch Linux* остались неизменными.

Arch Linux активно развивается и поддерживает последние версии программного обеспечения, предлагая пользователям возможность быть на передовой технологий. Дистрибутив завоевал популярность среди разработчиков, энтузиастов и тех, кто предпочитает иметь полный контроль над своей операционной системой. Благодаря обширной документации и активному сообществу *Arch Linux* остаётся одним из наиболее предпочитаемых дистрибутивов для тех, кто ищет гибкость и контроль в своей работе.

2.3.2 Версии

Arch Linux не имеет традиционных версий или номеров релизов, как некоторые другие дистрибутивы. Вместо этого он следует модели непрерывного развития (*rolling release*), что означает постоянное обновление пакетов в репозиториях.

Система всегда находится в состоянии готовности к обновлению до последних версий программного обеспечения, что позволяет пользователям иметь актуальные компоненты без необходимости перехода на новую версию дистрибутива.

2.3.3 Достоинства

1 Гибкость и настраиваемость. *Arch Linux* предоставляет пользователям максимальную свободу в настройке системы под свои нужды. Благодаря минималистичной базовой установке и обширным возможностям конфигурирования, пользователи могут адаптировать систему под конкретные требования и предпочтения.

2 Модель непрерывного развития (*rolling release*). Эта модель позволяет пользователям всегда иметь актуальные версии программного

обеспечения без необходимости перехода на новую версию дистрибутива. Обновления выпускаются регулярно, что обеспечивает доступ к последним функциям и улучшениям.

3 Обширная документация и активное сообщество. *Arch Linux* славится своей подробной и качественной документацией, которая помогает пользователям решать возникающие вопросы и задачи. Кроме того, активное и дружелюбное сообщество предоставляет поддержку и делится опытом, что особенно полезно для новичков.

4 Высокая производительность. Благодаря минималистичному подходу и возможности тонкой настройки системы, *Arch Linux* может обеспечить высокую производительность на различных аппаратных платформах.

Выбор инструментов для реализации курсового проекта — языка *C++*, редактора *Micro* и дистрибутива *Arch Linux* — обусловлен их уникальными преимуществами, которые в совокупности формируют эффективную среду разработки. Использование *Arch Linux* с графической оболочкой *Cinnamon* обеспечивает баланс между производительностью и удобством, *Micro* — быстрое взаимодействие с кодом, а *C++* — реализацию высокоэффективных алгоритмов. Такой набор инструментов не только ускоряет разработку, но и способствует глубокому пониманию внутренних механизмов ОС, что соответствует целям проектирования файловых систем на базе *FUSE*.

3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

3.1 Обоснование необходимости разработки

В современных условиях информационных технологий эффективное управление файловыми ресурсами является одной из ключевых задач при организации работы операционных систем. Традиционные файловые системы ядра обеспечивают высокую производительность, однако их разработка и модификация требует глубокого понимания работы ядра ОС и сопряжена с повышенными рисками сбоев из-за ошибок в критически важном коде. Использование *FUSE (Filesystem in Userspace)* позволяет реализовывать файловые системы в пространстве пользователя, что снижает возможность возникновения системных сбоев и упрощает процесс разработки и тестирования.

Файловая система *SFS* призвана обеспечить эмуляцию стандартного функционала операционных систем на базе *Linux*, позволяя пользователям выполнять операции чтения, записи, создания, удаления, а также управления атрибутами файлов через набор специализированных функций (таких как *xmp_init*, *xmp_getattr*, *xmp_readdir*, *xmp_mknod*, *xmp_mkdir*, *xmp_unlink*, *xmp_rmdir*, *xmp_read* и *xmp_write*). Такой подход позволяет:

- 1 Повысить надежность: реализация ключевых операций в пользовательском пространстве минимизирует риск выхода из строя ядра при возникновении ошибки;

- 2 Обеспечить гибкость: модульная архитектура позволяет легко добавлять новые функции или изменять поведение файловой системы без необходимости пересборки ядра;

- 3 Упростить тестирование и отладку: ошибки можно диагностировать и исправлять на уровне пользовательских процессов, что существенно сокращает время разработки.

Таким образом, разработка *SFS* является актуальным решением для образовательных и экспериментальных проектов, направленных на исследование особенностей реализации файловых систем и применение стандартных системных вызовов в окружении *FUSE*.

3.2 Технологии программирования, используемые при разработке

Основой программного продукта является библиотека *FUSE*, которая предоставляет интерфейс для создания файловых систем в пространстве пользователя. В выборе данной технологии отражены следующие ключевые аспекты:

- 1 Безопасность разработки: Реализация системы в пространстве пользователя позволяет существенно снизить критичность ошибок, поскольку сбой в пользовательском процессе не приводит к аварийному завершению работы операционной системы;

2 Соответствие *POSIX*-стандарту: Использование стандартных системных вызовов (таких как *lstat*, *opendir*, *readdir*, *pread*, *pwrite*, *unlink*, *rmdir* и др.) гарантирует совместимость *SFS* с существующими приложениями и инструментами *Linux*, что важно для интеграции в существующие инфраструктуры;

3 Язык программирования *C*: Выбор языка *C* обусловлен высокой производительностью, прямым доступом к системным вызовам и возможностью тонкой настройки работы с ресурсами. *C* является де-факто стандартом для разработки системного ПО, обеспечивая минимальные накладные расходы и высокую эффективность;

4 Модульность архитектуры: Разработка *SFS* структурирована в виде отдельных модулей для инициализации, работы с атрибутами, чтения содержимого каталогов, создания и удаления файлов, а также обработки операций чтения/записи и создания ссылок. Такой подход облегчает сопровождение, тестирование и потенциальную модернизацию продукта;

Комбинация *FUSE*, языка *C* и стандартов *POSIX* позволяет создать производительное, надежное и расширяемое программное обеспечение для работы с файловыми системами в условиях многозадачности современных операционных систем.

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

4.1 Обоснование и описание функциональных возможностей

Целью курсового проекта является разработка простой файловой системы (*SFS*), работающей в пространстве пользователя на базе *FUSE* (*Filesystem in Userspace*). В данном разделе описаны основные функциональные возможности создаваемой файловой системы.

4.1.1 Инициализация файловой системы

При запуске программы необходимо выполнить начальные настройки файловой системы, включая инициализацию структур данных, необходимых для работы файловой системы.

При запуске файловой системы происходит вызов функции `xmp_init` из библиотеки *FUSE*. Эта функция отвечает за настройку параметров работы файловой системы, таких как кэширование, параллельная запись и обработка `inode`. В рамках инициализации создаются необходимые структуры данных (например, таблица открытых файлов) и подготавливается пользовательское пространство для работы с файлами.

4.1.2 Работа с атрибутами файлов

Для корректной работы файловой системы необходимо реализовать возможность получения информации о файлах и директориях, таких как размер, права доступа, время последнего изменения и другие атрибуты.

Реализована функция `xmp_getattr`, которая использует системный вызов `lstat` для получения атрибутов файлов. Эта функция возвращает информацию, необходимую для работы с файлами и отображения структуры файловой системы.

4.1.3 Чтение содержимого директорий

Для работы с файловой системой пользователю нужно видеть содержимое директорий, а также иметь возможность навигации по файловой структуре.

Реализована функция `xmp_readdir`, которая использует системный вызов `opendir` и `readdir` для получения списка файлов и поддиректорий. Список возвращается в формате, который поддерживается *FUSE*, что позволяет интегрировать эту функцию в пользовательские приложения.

4.1.4 Создание новых файлов и директорий

Одним из ключевых функционалов любой файловой системы является возможность создания новых файлов и директорий.

Для реализации данного функционала используются функции `xmp_mknod` и `xmp_mkdir`. Первая отвечает за создание файлов с заданными атрибутами, вторая – за создание директорий с заданными правами доступа.

4.1.5 Удаление файлов и директорий

Пользователь должен иметь возможность удалять ненужные файлы и директории.

Реализованы функции *xmp_unlink* и *xmp_rmdir*, которые используют системные вызовы *unlink* и *rmdir* для удаления файлов и директорий соответственно. Эти функции проверяют существование файла или директории перед удалением.

4.1.6 Чтение и запись в файлы

Основная задача файловой системы – предоставлять возможность чтения и записи файлов для хранения данных.

Для чтения данных используется функция *xmp_read*, которая вызывает системный вызов *pread* для чтения данных из файла. Для записи данных используется функция *xmp_write*, которая вызывает *pwrite* для записи данных в определённое место файла.

4.1.7 Символьные и жёсткие ссылки

Символьные и жёсткие ссылки широко применяются для удобства работы с файлами и реализации сложных файловых структур.

Реализованы функции *xmp_symlink* и *xmp_link*, которые используют системные вызовы *symlink* и *link* для создания символических и жёстких ссылок соответственно.

4.1.8 Завершение работы файловой системы

Для предотвращения утечек памяти и некорректного завершения работы файловой системы необходимо обеспечить корректное завершение работы.

Завершение работы происходит при вызове функции *xmp_release*, которая закрывает все открытые дескрипторы файлов и освобождает занятые ресурсы.

В данном разделе были подробно рассмотрены основные функциональные возможности реализуемой файловой системы *SFS* на базе *FUSE*. Были описаны ключевые этапы работы системы, начиная с инициализации, когда происходит настройка всех необходимых структур данных, и заканчивая корректным завершением работы с освобождением ресурсов. Реализованные функции для работы с атрибутами файлов, чтением содержимого директорий, созданием и удалением файлов и директорий, чтением и записью данных, а также поддержки символьных и жёстких ссылок, демонстрируют комплексный подход к обеспечению функциональности файловой системы через стандартные системные вызовы *Linux*. Такой подход обеспечивает совместимость с пользовательскими приложениями, надёжность и эффективность при выполнении операций, что является залогом успешной работы и дальнейшего развития проекта.

5 АРХИТЕКТУРА РАЗРАБАТЫВАЕМОЙ ПРОГРАММЫ

5.1 Общая структура программы

Архитектура файловой системы SFS построена на модульном принципе, что позволяет разделить функциональность на независимые компоненты. Это упрощает разработку, тестирование и сопровождение программы. Основные модули программы:

1 Модуль инициализации (*xmp_init*): Выполняет настройку файловой системы при её монтировании;

2 Модуль работы с атрибутами (*xmp_getattr*): Отвечает за получение информации о файлах и директориях;

3 Модуль чтения содержимого директорий (*xmp_readdir*): Реализует функциональность отображения списка файлов и поддиректорий;

4 Модуль создания файлов и директорий (*xmp_mknod*, *xmp_mkdir*): Обеспечивает возможность создания новых элементов файловой системы;

5 Модуль удаления файлов и директорий (*xmp_unlink*, *xmp_rmdir*): Отвечает за удаление объектов из файловой системы;

6 Модуль чтения и записи (*xmp_read*, *xmp_write*): Реализует операции ввода-вывода с файлами;

7 Модуль ссылок (*xmp_symlink*, *xmp_link*): Обеспечивает создание символических и жёстких ссылок;

8 Модуль завершения работы (*xmp_release*): Выполняет очистку ресурсов и завершение работы.

5.2 Функциональная схема программы

Функциональная схема программы описывает последовательность взаимодействий между модулями:

При запуске программы происходит первичная инициализация, когда вызывается функция *xmp_init* для настройки параметров файловой системы и подготовки необходимых структур данных для работы с файлами и директориями. После инициализации система готова обрабатывать запросы, возникающие от пользователя. Для получения атрибутов файлов используется функция *xmp_getattr*, которая позволяет корректно определять свойства объектов файловой системы, а реализация чтения содержимого директорий через *xmp_readdir* обеспечивает полноценную навигацию по структуре системы.

Далее программа осуществляет создание и удаление объектов. Добавление новых файлов и директорий реализовано через вызовы функций *xmp_mknod* и *xmp_mkdir*, что позволяет динамически формировать файловую структуру. При этом удаление ненужных объектов производится посредством вызовов функций *xmp_unlink* и *xmp_rmdir*, что гарантирует безопасное удаление файлов и каталогов, при условии предварительной проверки их существования.

Работа с файлами осуществляется через функции *xmp_read* и *xmp_write*, предназначенные для чтения и записи данных соответственно. Эти функции обеспечивают оперативное взаимодействие с содержимым файлов, позволяя извлекать или сохранять данные в любой момент работы файловой системы. Завершение работы системы происходит через вызов функции *xmp_release*, которая отвечает за очистку задействованных ресурсов, закрытие открытых дескрипторов и освобождение памяти.

5.3 Блок-схема алгоритма

Программа начинается с функции *main*, которая устанавливает начальные параметры – в частности, с вызова *umask(0)* для установки маски прав доступа. Затем происходит перебор аргументов командной строки. Если встречается опция "*--plus*", устанавливается специальный флаг (*fill_dir_plus*), влияющий на способ заполнения списка директорий в функции *xmp_readdir*. После этого формируется новый список аргументов, который передается в функцию *fuse_main* вместе со структурой *xmp_oper*, содержащей указатели на функции-обработчики файловых операций.

Вызов *fuse_main* запускает главный цикл обработки запросов от файловой системы. При монтировании вызывается функция *xmp_init*, которая инициализирует файловую систему, настраивая поля структуры *fuse_config*. Например, здесь включается использование *inode* (*use_ino* = 1) и поддержка параллельных прямых операций записи (*parallel_direct_writes* = 1), а также устанавливаются таймауты кэширования в 0, если отключено автоматическое кэширование.

Далее основной цикл *FUSE* ожидает поступления запросов от операционной системы. В зависимости от типа запроса вызывается соответствующая функция из структуры *xmp_oper*. Например, для получения атрибутов файла или каталога используется *xmp_getattr*, которая выполняет системный вызов *lstat* для извлечения информации об объекте; для проверки доступа – функция *xmp_access* с вызовом *access*; для чтения символических ссылок – функция *xmp_readlink*, которая вызывает *readlink* и добавляет завершающий символ окончания строки.

Для обработки содержимого директорий используется функция *xmp_readdir*. Она открывает каталог с помощью *opendir*, затем перебирает записи с использованием *readdir*. Если установлен флаг *fill_dir_plus*, дополнительно вызывается *fstatat* для получения детальной информации о каждом объекте; если не установлен, информация заполняется вручную (например, номер *inode* и режим файла). Каждая полученная запись добавляется в буфер ответа посредством *callback*-функции *filler*.

Создание новых объектов файловой системы осуществляется через функции *xmp_mknod* и *xmp_mkdir*. Функция *xmp_mknod* вызывает вспомогательную функцию *mknod_wrapper*, которая в зависимости от типа создаваемого объекта (регулярный файл, каталог, символическая ссылка, *FIFO*, или, в *FreeBSD*, сокет) выбирает необходимый системный вызов,

например, *openat* для создания файла, *mkdirat* для создания каталога, *symlinkat* для создания символической ссылки, либо *mkfifoat* для *FIFO*. Для остальных типов объекты создаются посредством *mknodat*.

Удаление файлов и каталогов обрабатывается через функции *xmp_unlink* и *xmp_rmdir*, которые вызывают стандартные системные вызовы *unlink* и *rmdir* соответственно. Операции переименования и создания ссылок также реализованы: *xmp_rename* для переименования, *xmp_link* для создания жестких ссылок, и *xmp_symlink* для создания символических ссылок.

Работа с файлами (чтение, запись, открытие, создание) делегируется функциям *xmp_open*, *xmp_create*, *xmp_read* и *xmp_write*. *xmp_open* открывает файл и сохраняет файловый дескриптор в структуре *fuse_file_info*, обрабатывая флаг *O_DIRECT*, который активирует прямой ввод-вывод. Чтение и запись данных реализованы посредством *pread* и *pwrite*, что позволяет работать с данными по заданному смещению в файле.

Дополнительные операции, такие как изменение прав доступа (*xmp_chmod*), изменение владельца (*xmp_chown*), изменение размера файла (*xmp_truncate*) и обновление временных меток (*xmp_utimens*, если поддерживается), также вызывают соответствующие системные вызовы (*chmod*, *lchown*, *truncate/ftruncate*, *utimensat*).

Дополнительный функционал, такой как получение состояния файловой системы (*xmp_statfs*), синхронизация данных (*xmp_fsync*), резервирование дискового пространства (*xmp_fallocate*), работа с расширенными атрибутами (*xmp_setxattr*, *xmp_getxattr*, *xmp_listxattr*, *xmp_removexattr*), копирование данных между файлами (*xmp_copy_file_range*) и изменение позиции в файле (*xmp_lseek*), реализован в отдельных функциях, каждая из которых вызывает системные вызовы для выполнения своих задач.

По завершении обработки запроса происходит освобождение ресурсов. Файловые дескрипторы закрываются (например, в функции *xmp_release* или локально в функциях, если файл открывался непосредственно для конкретной операции). Результатом каждой операции является статус (0 указывает на успешное выполнение, а в случае ошибки возвращается отрицательное значение, соответствующее коду ошибки *errno*). При завершении работы файловой системы *FUSE* происходит размонтирование и корректное освобождение памяти, что предотвращает утечки ресурсов.

Таким образом, алгоритм программы последовательно выполняет следующие этапы: инициализация и разбор аргументов, настройка параметров файловой системы в *xmp_init*, переход к циклу обработки запросов, где в зависимости от типа запроса вызывается соответствующий обработчик (например, получение атрибутов, работа с содержимым директории, создание или удаление объектов, работа с файлом или управление атрибутами), и, наконец, завершение работы с корректным освобождением ресурсов. Это обеспечивает модульность, гибкость и стабильность работы файловой системы, реализованной с использованием *FUSE*.

ЗАКЛЮЧЕНИЕ

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

- [1] История Linux [Электронный ресурс]. – Режим доступа: <https://losst.pro/istoriya-komand-linux>. – Дата доступа: 25.02.2025.
- [2] Файловые системы [Электронный ресурс]. – Режим доступа: <https://www.kingston.com/ru/blog/personal-storage/understanding-file-systems> – Дата доступа: 25.02.2025.
- [3] Файловая система Linux [Электронный ресурс] – Режим доступа: <https://timeweb.com/ru/community/articles/struktura-i-tipy-faylovyh-sistem-v-linux>. Дата доступа: 25.02.2025.
- [4] The Virtual File System [Электронный ресурс] – Режим доступа: <http://www.science.unitn.it/~fiorella/guidelinux/tlk/node102.html#SECTION00112000000000000000>. Дата доступа: 15.01.2025.
- [5] FUSE: как написать свою файловую систему [Электронный ресурс] – Режим доступа: <https://habr.com/ru/companies/vk/articles/821905/>. Дата доступа: 25.02.2025.
- [6] Micro a modern and intuitive terminal-based text editor [Электронный ресурс]. – Режим доступа: <https://micro-editor.github.io/> – Дата доступа: 25.02.2025.
- [7] Arch Linux [Электронный ресурс]. – Режим доступа: <https://archlinux.org/> – Дата доступа: 25.02.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Справка о проверке на заимствования

ПРИЛОЖЕНИЕ Б

(обязательное)

Листинг программного кода

```
#define FUSE_USE_VERSION 31

#define _GNU_SOURCE

#ifdef linux
/* For pread()/pwrite()/utimensat() */
#define _XOPEN_SOURCE 700
#endif

#include <fuse.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <dirent.h>
#include <errno.h>
#ifdef __FreeBSD__
#include <sys/socket.h>
#include <sys/un.h>
#endif
#include <sys/time.h>
#ifdef HAVE_SETXATTR
#include <sys/xattr.h>
#endif

static int mknod_wrapper(int dirfd, const char *path, const char *link,
                        int mode, dev_t rdev)
{
    int res;

    if (S_ISREG(mode)) {
        res = openat(dirfd, path, O_CREAT | O_EXCL | O_WRONLY, mode);
        if (res >= 0)
            res = close(res);
    } else if (S_ISDIR(mode)) {
        res = mkdirat(dirfd, path, mode);
    } else if (S_ISLNK(mode) && link != NULL) {
        res = symlinkat(link, dirfd, path);
    } else if (S_ISFIFO(mode)) {
        res = mkfifoat(dirfd, path, mode);
    }
#ifdef __FreeBSD__
    } else if (S_ISSOCK(mode)) {
        struct sockaddr_un su;
        int fd;

        if (strlen(path) >= sizeof(su.sun_path)) {
            errno = ENAMETOOLONG;
            return -1;
        }
        fd = socket(AF_UNIX, SOCK_STREAM, 0);
```



```

        if (fd >= 0) {
            /*
             * We must bind the socket to the underlying file
             * system to create the socket file, even though
             * we'll never listen on this socket.
             */
            su.sun_family = AF_UNIX;
            strncpy(su.sun_path, path, sizeof(su.sun_path));
            res = bindat(dirfd, fd, (struct sockaddr*)&su,
                        sizeof(su));
            if (res == 0)
                close(fd);
        } else {
            res = -1;
        }
    #endif
    } else {
        res = mknodat(dirfd, path, mode, rdev);
    }

    return res;
}

static int fill_dir_plus = 0;

static void *xmp_init(struct fuse_conn_info *conn,
                     struct fuse_config *cfg)
{
    (void) conn;
    cfg->use_ino = 1;

    cfg->parallel_direct_writes = 1;

    if (!cfg->auto_cache) {
        cfg->entry_timeout = 0;
        cfg->attr_timeout = 0;
        cfg->negative_timeout = 0;
    }

    return NULL;
}

static int xmp_getattr(const char *path, struct stat *stbuf,
                      struct fuse_file_info *fi)
{
    (void) fi;
    int res;

    res = lstat(path, stbuf);
    if (res == -1)
        return -errno;

    return 0;
}

```

```

static int xmp_access(const char *path, int mask)
{
    int res;

    res = access(path, mask);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_readlink(const char *path, char *buf, size_t size)
{
    int res;

    res = readlink(path, buf, size - 1);
    if (res == -1)
        return -errno;

    buf[res] = '\0';
    return 0;
}

static int xmp_readdir(const char *path, void *buf, fuse_fill_dir_t filler,
                      off_t offset, struct fuse_file_info *fi,
                      enum fuse_readdir_flags flags)
{
    DIR *dp;
    struct dirent *de;

    (void) offset;
    (void) fi;
    (void) flags;

    dp = opendir(path);
    if (dp == NULL)
        return -errno;

    while ((de = readdir(dp)) != NULL) {
        struct stat st;
        if (fill_dir_plus) {
            fstatat(dirfd(dp), de->d_name, &st,
                    AT_SYMLINK_NOFOLLOW);
        } else {
            memset(&st, 0, sizeof(st));
            st.st_ino = de->d_ino;
            st.st_mode = de->d_type << 12;
        }
        if (filler(buf, de->d_name, &st, 0, fill_dir_plus))
            break;
    }

    closedir(dp);
    return 0;
}

```

```

}

static int xmp_mknod(const char *path, mode_t mode, dev_t rdev)
{
    int res;

    res = mknod_wrapper(AT_FDCWD, path, NULL, mode, rdev);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_mkdir(const char *path, mode_t mode)
{
    int res;

    res = mkdir(path, mode);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_unlink(const char *path)
{
    int res;

    res = unlink(path);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_rmdir(const char *path)
{
    int res;

    res = rmdir(path);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_symlink(const char *from, const char *to)
{
    int res;

    res = symlink(from, to);
    if (res == -1)
        return -errno;

    return 0;
}

```

```

}

static int xmp_rename(const char *from, const char *to, unsigned int flags)
{
    int res;

    if (flags)
        return -EINVAL;

    res = rename(from, to);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_link(const char *from, const char *to)
{
    int res;

    res = link(from, to);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_chmod(const char *path, mode_t mode,
                    struct fuse_file_info *fi)
{
    (void) fi;
    int res;

    res = chmod(path, mode);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_chown(const char *path, uid_t uid, gid_t gid,
                    struct fuse_file_info *fi)
{
    (void) fi;
    int res;

    res = lchown(path, uid, gid);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_truncate(const char *path, off_t size,
                    struct fuse_file_info *fi)

```

```

{
    int res;

    if (fi != NULL)
        res = ftruncate(fi->fh, size);
    else
        res = truncate(path, size);
    if (res == -1)
        return -errno;

    return 0;
}

#ifdef HAVE_UTIMENSAT
static int xmp_utimens(const char *path, const struct timespec ts[2],
                      struct fuse_file_info *fi)
{
    (void) fi;
    int res;

    /* don't use utime/utimes since they follow symlinks */
    res = utimensat(0, path, ts, AT_SYMLINK_NOFOLLOW);
    if (res == -1)
        return -errno;

    return 0;
}
#endif

static int xmp_create(const char *path, mode_t mode,
                     struct fuse_file_info *fi)
{
    int res;

    res = open(path, fi->flags, mode);
    if (res == -1)
        return -errno;

    fi->fh = res;
    return 0;
}

static int xmp_open(const char *path, struct fuse_file_info *fi)
{
    int res;

    res = open(path, fi->flags);
    if (res == -1)
        return -errno;

    /* Enable direct_io when open has flags O_DIRECT to enjoy the feature
    parallel_direct_writes (i.e., to get a shared lock, not exclusive
lock,
    for writes to the same file). */
    if (fi->flags & O_DIRECT) {

```

```

        fi->direct_io = 1;
        fi->parallel_direct_writes = 1;
    }

    fi->fh = res;
    return 0;
}

static int xmp_read(const char *path, char *buf, size_t size, off_t offset,
                   struct fuse_file_info *fi)
{
    int fd;
    int res;

    if(fi == NULL)
        fd = open(path, O_RDONLY);
    else
        fd = fi->fh;

    if (fd == -1)
        return -errno;

    res = pread(fd, buf, size, offset);
    if (res == -1)
        res = -errno;

    if(fi == NULL)
        close(fd);
    return res;
}

static int xmp_write(const char *path, const char *buf, size_t size,
                    off_t offset, struct fuse_file_info *fi)
{
    int fd;
    int res;

    (void) fi;
    if(fi == NULL)
        fd = open(path, O_WRONLY);
    else
        fd = fi->fh;

    if (fd == -1)
        return -errno;

    res = pwrite(fd, buf, size, offset);
    if (res == -1)
        res = -errno;

    if(fi == NULL)
        close(fd);
    return res;
}

```

```

static int xmp_statfs(const char *path, struct statvfs *stbuf)
{
    int res;

    res = statvfs(path, stbuf);
    if (res == -1)
        return -errno;

    return 0;
}

static int xmp_release(const char *path, struct fuse_file_info *fi)
{
    (void) path;
    close(fi->fh);
    return 0;
}

static int xmp_fsync(const char *path, int isdatasync,
                    struct fuse_file_info *fi)
{
    /* Just a stub.    This method is optional and can safely be left
       unimplemented */

    (void) path;
    (void) isdatasync;
    (void) fi;
    return 0;
}

#ifdef HAVE_POSIX_FALLOCATE
static int xmp_fallocate(const char *path, int mode,
                        off_t offset, off_t length, struct fuse_file_info *fi)
{
    int fd;
    int res;

    (void) fi;

    if (mode)
        return -EOPNOTSUPP;

    if (fi == NULL)
        fd = open(path, O_WRONLY);
    else
        fd = fi->fh;

    if (fd == -1)
        return -errno;

    res = -posix_fallocate(fd, offset, length);

    if (fi == NULL)
        close(fd);
    return res;
}

```

```

}
#endif

#ifdef HAVE_SETXATTR
/* xattr operations are optional and can safely be left unimplemented */
static int xmp_setxattr(const char *path, const char *name, const char
*value,
                        size_t size, int flags)
{
    int res = lsetxattr(path, name, value, size, flags);
    if (res == -1)
        return -errno;
    return 0;
}

static int xmp_getxattr(const char *path, const char *name, char *value,
                        size_t size)
{
    int res = lgetxattr(path, name, value, size);
    if (res == -1)
        return -errno;
    return res;
}

static int xmp_listxattr(const char *path, char *list, size_t size)
{
    int res = llistxattr(path, list, size);
    if (res == -1)
        return -errno;
    return res;
}

static int xmp_removexattr(const char *path, const char *name)
{
    int res = lremovexattr(path, name);
    if (res == -1)
        return -errno;
    return 0;
}
#endif /* HAVE_SETXATTR */

#ifdef HAVE_COPY_FILE_RANGE
static ssize_t xmp_copy_file_range(const char *path_in,
                                   struct fuse_file_info *fi_in,
                                   off_t offset_in, const char *path_out,
                                   struct fuse_file_info *fi_out,
                                   off_t offset_out, size_t len, int flags)
{
    int fd_in, fd_out;
    ssize_t res;

    if(fi_in == NULL)
        fd_in = open(path_in, O_RDONLY);
    else
        fd_in = fi_in->fh;

```



```

    if (fd_in == -1)
        return -errno;

    if (fi_out == NULL)
        fd_out = open(path_out, O_WRONLY);
    else
        fd_out = fi_out->fh;

    if (fd_out == -1) {
        close(fd_in);
        return -errno;
    }

    res = copy_file_range(fd_in, &offset_in, fd_out, &offset_out, len,
                          flags);
    if (res == -1)
        res = -errno;

    if (fi_out == NULL)
        close(fd_out);
    if (fi_in == NULL)
        close(fd_in);

    return res;
}
#endif

static off_t xmp_lseek(const char *path, off_t off, int whence, struct
fuse_file_info *fi)
{
    int fd;
    off_t res;

    if (fi == NULL)
        fd = open(path, O_RDONLY);
    else
        fd = fi->fh;

    if (fd == -1)
        return -errno;

    res = lseek(fd, off, whence);
    if (res == -1)
        res = -errno;

    if (fi == NULL)
        close(fd);
    return res;
}

static const struct fuse_operations xmp_oper = {
    .init          = xmp_init,
    .getattr       = xmp_getattr,
    .access        = xmp_access,

```

```

        .readlink    = xmp_readlink,
        .readdir     = xmp_readdir,
        .mknod       = xmp_mknod,
        .mkdir       = xmp_mkdir,
        .symlink     = xmp_symlink,
        .unlink      = xmp_unlink,
        .rmdir       = xmp_rmdir,
        .rename      = xmp_rename,
        .link        = xmp_link,
        .chmod       = xmp_chmod,
        .chown       = xmp_chown,
        .truncate    = xmp_truncate,
#ifdef HAVE_UTIMENSAT
        .utimens     = xmp_utimens,
#endif
        .open        = xmp_open,
        .create      = xmp_create,
        .read        = xmp_read,
        .write       = xmp_write,
        .statfs      = xmp_statfs,
        .release     = xmp_release,
        .fsync       = xmp_fsync,
#ifdef HAVE_POSIX_FALLOCATE
        .fallocate   = xmp_fallocate,
#endif
#ifdef HAVE_SETXATTR
        .setxattr    = xmp_setxattr,
        .getxattr    = xmp_getxattr,
        .listxattr   = xmp_listxattr,
        .removexattr = xmp_removexattr,
#endif
#ifdef HAVE_COPY_FILE_RANGE
        .copy_file_range = xmp_copy_file_range,
#endif
        .lseek       = xmp_lseek,
};

int main(int argc, char *argv[])
{
    enum { MAX_ARGS = 10 };
    int i, new_argc;
    char *new_argv[MAX_ARGS];

    umask(0);
    /* Process the "--plus" option apart */
    for (i=0, new_argc=0; (i<argc) && (new_argc<MAX_ARGS); i++) {
        if (!strcmp(argv[i], "--plus")) {
            fill_dir_plus = FUSE_FILL_DIR_PLUS;
        } else {
            new_argv[new_argc++] = argv[i];
        }
    }
    return fuse_main(new_argc, new_argv, &xmp_oper, NULL);
}

```

ПРИЛОЖЕНИЕ В
(обязательное)
Функциональная схема алгоритма, реализующего
программное средство

ПРИЛОЖЕНИЕ Г
(обязательное)
Блок схема алгоритма

ПРИЛОЖЕНИЕ Д
(обязательное)
Графический интерфейс пользователя

ПРИЛОЖЕНИЕ Е
(обязательное)
Ведомость курсового проекта