Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра электронных вычислительных машин
Дисциплина: Операционные системы и системное программирование

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

«УТИЛИТА ФОРМАТИРОВАНИЯ И ПРОВЕРКИ ФАЙЛОВОЙ СИСТЕМЫ SFS»

БГУИР КП 1-40 02 01 113 ПЗ

Студент В. Н. Коваль

Руководитель Л. П. Поденок

Минск 2025

Учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

УТВЕРЖДАЮ
Заведующий кафедрой
(подпись)
_____ 2025 г.

ЗАДАНИЕ
по курсовому проектированию

Студенту Ковалю Вадиму Николаевичу

1. Тема проекта: *Утилита форматирования и проверки файловой системы SFS*

2. Срок сдачи студентом законченного проекта: *15 мая 2025 г.*

3. Исходные данные к проекту: *язык программирования C*

4. Содержание расчетно-пояснительной записки (перечень вопросов, которые подлежат разработке):

Введение. 1. Обзор методов и алгоритмов решения поставленной задачи. 2. Обоснование выбранных методов и алгоритмов. 3. Описание программы для программиста. 4. Описание алгоритмов решения задачи. 5. Руководство пользователя. Заключение. Список использованных источников. Приложения.

5. Перечень графического материала (с точными обозначениями обязательных чертежей и графиков):

1. *Схема алгоритма работы функции*
2. *Скриншоты работы программы*
3. *Ведомость документов*

6. Консультант по проекту (с обозначением разделов проекта) *Поденок Л. П.*

7. Календарный график работы над проектом на весь период проектирования (с обозначением сроков выполнения и трудоемкости отдельных этапов):

раздел 1 к 01.03. – 15%;

раздел 2, 3 к 01.04. – 50%;

раздел 4, 5 к 01.05. – 80%;

оформление пояснительной записки и графического материала к 15.05.2025 – 100%

Защита курсового проекта с 29.05 по 09.06.

Руководитель Л. П. Поденок

Задание принял к исполнению В. Н. Коваль

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Файловые системы играют ключевую роль в организации и управлении данными в компьютерных системах. Разработка собственной файловой системы требует не только понимания принципов хранения, доступа и защиты данных, но и создания вспомогательных утилит, обеспечивающих корректную работу с её структурой. Одними из таких инструментов являются утилиты форматирования и проверки целостности, которые необходимы как на этапе начальной инициализации файловой системы, так и при её эксплуатации.

В рамках курсового проекта была реализована простая файловая система `SFS` (`Simple File System`) и разработаны две служебные утилиты:

1) `mkfs.sfs` — утилита форматирования, создающая новый образ файловой системы с инициализированными структурами (суперблок, иноды, битовые карты, таблицы размещения данных и др.);

2) `fsch` — утилита проверки, осуществляющая валидацию целостности образа файловой системы, обнаружение логических ошибок и повреждений в структуре метаданных.

Создание указанных утилит выполнено на языке программирования `C` с использованием стандартных средств работы с файлами и структур данных. Утилиты ориентированы на работу с виртуальной файловой системой, состояние которой хранится на двух отдельных бинарных файлах.

Основные задачи, решаемые с помощью разработанных утилит:

1) инициализация образа `SFS` с зафиксированной структурой данных;

2) контроль корректности и согласованности данных в метаструктурах;

3) обнаружение типовых ошибок: битые иноды, несоответствия в битмапах, лишние или отсутствующие ссылки на блоки данных.

Разработка и использование таких инструментов критически важно при построении низкоуровневого хранилища данных, особенно в контексте обучения системному программированию, операционным системам и архитектуре файловых систем.

1 ОБЗОР МЕТОДОВ И АЛГОРИТМОВ РЕШЕНИЯ ПОСТАВЛЕННОЙ ЗАДАЧИ

1.1 Методы работы с пользовательскими файловыми системами

Реализация собственной пользовательской файловой системы требует разработки набора утилит, обеспечивающих базовую и вспомогательную функциональность: форматирование, проверку целостности, чтение/запись данных, управление метаинформацией и т.д. В рамках поставленной задачи взаимодействие с ядром операционной системы или использование модуля `FUSE` не предполагается. Вместо этого, реализация основана на работе с бинарными файлами, которые эмулируют поведение и структуру реального блочного устройства. Такой подход обеспечивает независимость от операционной системы, позволяет точно контролировать структуру данных и облегчает отладку.

Типичный подход к построению пользовательской файловой системы в таких условиях включает следующие ключевые методы:

1) на этапе инициализации файловой системы создаётся специальная структура метаданных — суперблок, содержащая базовую информацию о конфигурации файловой системы. В данной реализации суперблок выделяется в отдельный бинарный файл (`super`), что упрощает его изоляцию от пользовательских данных и снижает риск повреждения.

2) второй бинарный файл (`file_structure`) используется для хранения основной информации, формирующей «тело» файловой системы. Он включает в себя:

1) битмап блоков данных — показывает, какие блоки заняты, а какие свободны;

2) битмап `inodes` — указывает, какие `inodes` используются;

3) таблицу `inodes` — массив структур, каждая из которых описывает файл или директорию: размер, тип, временные метки, ссылки на блоки данных и т.д.;

4) блоки пользовательских данных — непосредственно содержимое файлов, расположенное по заранее рассчитанным смещениям и размеченное по блочной системе.

Раздельное хранение позволяет гибко управлять компонентами и уменьшает связанность между утилитами. Например, проверка суперблока может выполняться независимо от анализа данных.

1.2 Методы форматирования и инициализации

Утилита `mkfs.sfs` отвечает за первичную инициализацию файловой системы, создавая структуру, пригодную для дальнейшего использования. Форматирование осуществляется напрямую на уровне бинарного файла, представляющего виртуальный диск, с точным соблюдением смещений и порядка размещения данных.

Основные этапы форматирования:

1) инициализация суперблока. Суперблок содержит ключевые параметры: общее количество блоков, размер блока, число `inodes`, смещения для каждой области (битмапы, таблица `inodes`, данные) и сигнатуру. Эта информация записывается в начало файла и служит ориентиром для всех операций с файловой системой.

2) создание битмапов блоков и `inodes`. Формируются два отдельных битмапа: один — для отслеживания использования блоков данных, второй — для `inodes`. Изначально все позиции помечаются как свободные, за исключением служебных, которые резервируются.

3) очистка блоков данных. Вся область данных заполняется нулями или другим шаблоном. Это исключает наличие мусорной информации и имитирует «пустой» диск. Это достигается простой операцией: `lseek` до начала `data region` и многократной записью блоков нулей (`write(zero_block, block_size)` в цикле). В результате получается полностью предсказуемое и упорядоченное состояние всех данных.

4) формирование таблицы `inodes`. Таблица `inodes` создаётся в виде массива структур фиксированного размера. Каждый `inode` содержит метаданные файла: тип, размер, указатели на блоки и статус. На этапе форматирования все `inodes` инициализируются значениями по умолчанию.

5) резервирование служебных ресурсов. Выделяется `inode` под корневой каталог, а также соответствующий блок данных. Эти элементы сразу помечаются как занятые в битмапах и могут содержать минимальную инициализированную структуру (например, "." и "..").

Вся логика построена на строгом следовании соглашениям о размещении и фиксированных смещениях, что обеспечивает предсказуемость и совместимость между утилитами и структурой образа.

1.3 Методы проверки целостности файловой системы

Утилита `fsch` реализует методы логической проверки образа пользовательской файловой системы `SFS`. Целью является выявление нарушений структурной целостности и несоответствий между метаданными и фактическими данными на диске. Работа утилиты основана на анализе бинарного файла `file_structure` и суперблока `super`.

Ключевые методы проверки:

1) проверка согласованности суперблока. Считываются параметры суперблока (размер блоков, общее число блоков, смещения битмапов и `inode`-таблицы) и сопоставляются с фактическими размерами файлов. Обнаружение несоответствий сигнализирует о возможной порче или ошибке при форматировании.

2) анализ битмапов**.** Производится проход по битмапам блоков и `inodes`: проверяется, не помечены ли используемые блоки как свободные и наоборот. Ведётся подсчёт числа используемых записей, сравниваемый с фактическими `inode`-записями и связанными блоками.

3) проверка связности данных. Все активные `inodes` анализируются на предмет допустимости указанных блоков данных. Проверяется, что блоки не выходят за границы файловой системы, не пересекаются между различными `inodes` и соответствуют логической структуре хранения.

4) проверка ссылочной целостности. Для каждого блока данных и `inode` отслеживаются ссылки. В процессе обхода формируются таблицы использованных ресурсов. Затем выявляются блоки и `inodes`, на которые никто не ссылается (потенциальные утечки), или наоборот — те, которые используются несколькими структурами (что может указывать на повреждение).

Таким образом, утилита `fsch` выступает аналогом системной `fsck`, но адаптирована под специфику структуры и ограничений файловой системы `SFS`. Она обеспечивает базовый уровень контроля целостности данных, особенно важный при прямой работе с бинарным образом без слоя абстракции.

2 ОБОСНОВАНИЕ ВЫБРАННЫХ МЕТОДОВ И АЛГОРИТМОВ

2.1 Выбор подхода к форматированию

Для реализации утилиты форматирования SFS (Simple File System) был выбран низкоуровневый подход, основанный на прямом управлении структурой дискового образа через стандартные функции языка C (fopen, fread, fwrite, fseek). Такой подход обеспечивает:

1) прямой контроль над размещением данных — форматирование производится на уровне байтов и секторов, что исключает промежуточные абстракции и даёт полный контроль над структурой файловой системы.

2) независимость от платформенных библиотек — работа осуществляется без внешних зависимостей и драйверов, что делает код переносимым и пригодным для отладки и монтирования с помощью FUSE.

3) высокую воспроизводимость — все метаданные инициализируются по заранее определённой схеме, что гарантирует предсказуемость результата.

Форматирование начинается с инициализации суперблока, битовых карт (data/inode bitmap) и таблицы инодов. В коде инициализация суперблока осуществляется функцией superblock_init(), а корневой каталог создаётся с помощью root_dir_init().

В процессе инициализации корневого каталога (/) происходит:

1) выделение памяти под структуру filetype и inode.

2) установка базовых атрибутов (path, name, permissions, timestamps, UID/GID и т.д.).

3) установка inode_bitmap[1] = 1, что резервирует второй инод под корень (нулевой инод может быть зарезервирован или неиспользуем).

4) назначение числа жёстких ссылок (num_links = 2) как у типичного корневого каталога.

5) назначение уникального номера инода через find_free_inode().

6) инициализация всех временных меток (ctime, atime, mtime, btime) через time(NULL).

После инициализации данные сериализуются функцией save_system_state().

При наличии ранее созданного образа система предлагает пользователю форматировать диск повторно или загрузить существующую файловую структуру. Это реализовано через:

1) проверку наличия файлов с помощью `access()`.

2) запрос подтверждения у пользователя.

3) условное выполнение форматирования или восстановления (`restore_file_system()`).

Такой подход позволяет безопасно переформатировать диск по желанию пользователя и, при необходимости, восстановить состояние из бинарных образов.

Выбранный метод позволяет обеспечить полную гибкость при разработке собственной файловой системы: от ручного управления метаданными до точной отладки процессов сериализации и восстановления.

2.2 Выбор алгоритма проверки целостности

Проверка целостности является критически важной задачей при работе с файловыми системами. Основной целью данной проверки является выявление логических и структурных ошибок, которые могут привести к потере данных или некорректной работе файловой системы. При разработке утилиты проверки файловой системы `SFS` был выбран программный подход, основанный на последовательной десериализации структур и выполнении набора логических проверок над ними.

Суперблок десериализуется из бинарного файла `super.bin`. После загрузки выполняются следующие проверки:

1) соответствие размеров битмап и блока данных ожидаемым значениям;

2) отсутствие явно некорректных значений (например, `0xFF` во всех байтах блока данных);

3) проверка допустимости размеров и структуры.

Древовидная структура файлов (`filetype`) загружается из файла `file_structure.bin`. Для каждой вершины дерева выполняются:

1) проверка корректности количества дочерних элементов;

2) отсутствие `NULL`-указателей в массивах дочерних узлов;

3) проверка правильности связей родитель-потомок.

Для каждого файла и директории (если задан `inode`) осуществляется валидация следующего:

1) корректность номера `inode`;

2) допустимое количество привязанных блоков данных;

3) допустимость временных меток (a_time, m_time, c_time, b_time);

4) проверка значений полей доступа (permissions, user_id, group_id);

5) корректность номеров блоков данных (в пределах общего объема блоков).

Проверки проводятся итеративно и рекурсивно, что позволяет анализировать всю файловую систему целиком. Такой подход не требует внедрения хэш-функций или контрольных сумм, но обеспечивает детальную диагностику логических ошибок.

Выбор именно этого подхода обусловлен следующими причинами:

1) полный контроль над структурой данных (так как файловая система разрабатывается самостоятельно);

2) возможность отладки и трассировки ошибок;

3) простота реализации и отсутствие зависимости от внешних библиотек;

4) возможность последующего расширения (например, добавления контрольных сумм, CRC или хэширования inode'ов и блоков).

Таким образом, был реализован универсальный и расширяемый механизм программной проверки целостности, способный выявлять как очевидные, так и структурные ошибки.

2.3 Выбор файловой архитектуры

В рамках разработки утилиты форматирования и проверки файловой системы SFS была выбрана модульная архитектура, включающая три основные структурные компоненты: суперблок, inod’ы и абстракции файловой иерархии (структура filetype). Такая архитектура обеспечивает как простоту реализации, так и возможность расширения в будущем.

Общая структура образа.

Файловая система SFS представляет собой монолитный бинарный образ, состоящий из следующих сегментов:

1) суперблок, содержащий служебную информацию и битмапы занятости;

2) таблица inod’ов, описывающих файлы и каталоги;

3) непрерывная область блоков данных фиксированного размера;

4) связанная структура filetype, обеспечивающая иерархию путей,

метаданные и связи между файлами.

Структура суперблока.

Суперблок определён в `superblock.h` и содержит:

1) `data_blocks[102400]` — область для хранения пользовательских данных (100 блоков по 1024 байта);

2) `data_bitmap[105]` — битовая карта занятости блоков данных;

3) `inode_bitmap[105]` — битовая карта занятости `inod'ов`.

Эта информация используется при форматировании и проверке целостности, а также для поиска свободных ресурсов.

Иерархия файлов и каталогов: структура `filetype`.

Для поддержки логической структуры файловой системы введена структура `filetype (filetype.h)`, содержащая:

1) абсолютный путь (`path`) и имя (`name`);

2) указатель на соответствующий `inode` (`inum`);

3) указатель на родителя (`parent`) и массив дочерних элементов (`children`);

4) тип объекта (`type`) — например, `"file"` или `"directory"`;

5) счётчик ссылок (`num_links`) и дочерних элементов (`num_children`);

6) флаг `valid`, определяющий корректность элемента.

Таким образом, `filetype` реализует логическую файловую модель и позволяет выполнять операции поиска, создания, удаления и проверки путей.

Данная архитектура была выбрана благодаря своей логической структурированности: данные и метаданные чётко разделены по модулям, что упрощает отладку и последующую модификацию; `inod'ы` обеспечивают низкоуровневое представление файловых ресурсов, в то время как структура `filetype` реализует высокоуровневую модель, позволяющую строить иерархию каталогов, аналогичную используемой в современных файловых системах. Кроме того, такая архитектура легко масштабируется и допускает дальнейшее расширение — например, добавление поддержки символьных ссылок, механизмов журналирования или кэширования.

2.4 Обоснование дополнительных проектных решений

В процессе разработки утилиты проверки и форматирования `SFS` были приняты ряд технических решений, направленных на повышение надёжно-

сти, расширяемости и удобства эксплуатации программного обеспечения.

1) реализована строгая сериализация и десериализация всех ключевых структур файловой системы (`superblock`, `inode`, `filetype`) с чётким контролем порядка записи и считывания, что позволяет гарантировать восстановление корректного состояния даже при наличии рекурсивных связей в структуре каталогов. В целях упрощения отладки и анализа данные разделены между двумя бинарными файлами: метаданные (суперблок и `inode`) и файловая иерархия (`filetype`) хранятся отдельно. Это повышает читаемость и модульность проекта, позволяя независимо анализировать состояние блоков и логическую структуру дерева.

2) предусмотрена поддержка вложенной файловой иерархии с восстановлением связей между родительскими и дочерними элементами, что позволяет корректно моделировать реальную структуру файловой системы. Это решение особенно важно для последующей проверки связности дерева каталогов и детектирования «висячих» `inod`.

3) реализована возможность масштабирования: структура `superblock` и способ хранения `inod` позволяют легко адаптировать систему под различные объёмы — увеличивая количество блоков или добавляя новые типы дескрипторов без кардинальной переработки архитектуры.

Такой подход делает разработанную утилиту не просто инструментом для начальной инициализации файловой системы, но и полноценным средством для анализа, контроля и восстановления её состояния. Архитектура проекта ориентирована на расширение: в дальнейшем может быть реализована поддержка журналирования, учёта прав доступа, квотирования и интеграции с пользовательскими пространствами, что делает систему гибкой основой для дальнейших исследований и разработок в области системного программного обеспечения.

3 ОПИСАНИЕ ПРОГРАММЫ ДЛЯ ПРОГРАММИСТА

Основная реализация утилит форматирования и проверки файловой системы выполнена в двух ключевых компонентах:

1) `fsch.c` — реализует утилиту проверки целостности пользовательской файловой системы `SFS`. Он загружает `superblock` и файловую структуру из бинарных файлов, проводит валидацию метаданных (битмап, размеры, блоки, `inode`), а также проверяет целостность дерева каталогов;

2) `mkfs_sfs.c` — содержит минимальную обёртку для запуска восстановления файловой системы: он вызывает функцию `restore_file_system()`, которая переинициализирует структуру `SFS`, например, при создании новой файловой системы;

`Makefile` — обеспечивает процесс сборки программы.

Файл `filetype.c` содержит функцию `filetype_from_path`, которая по абсолютному пути находит соответствующий узел (файл или директорию) в структуре `filetype`, начиная от корня `root`.

Файл `fs_init.c` — модуль инициализации и восстановления файловой системы.

Содержит функции для:

1) `root_dir_init()` — создания корневого каталога;

2) `save_system_state()` — сохранения состояния;

3) `restore_file_system()` — восстановления существующей или форматирования новой файловой системы;

4) `cleanup_filesystem()` — очистки ресурсов.

Файл inode.c — вспомогательный модуль для работы с inode и структурой каталогов.

Содержит:

1) `find_free_inode()` — поиск первого свободного `inode` в битовой карте суперблока (начиная с 2, так как 0 и 1 — зарезервированы);

2) `add_child()` — добавление дочернего узла (файла или каталога) в родительскую директорию с реаллокацией массива детей;

Файл `superblok.c` — модуль инициализации и управления суперблоком файловой системы.

Содержит:

1) глобальную переменную `s_block` — сам суперблок;

2) `superblock_init()` — обнуляет битовые карты `inodes` и `data blocks`;

3) `find_free_db()` — ищет первый свободный блок данных в `data bitmap` и помечает его как занятый.

Файл `utilities.c` — вспомогочный модуль, реализующий сериализацию и десериализацию структур данных файловой системы.

Содержит:

1) функции сохранения и восстановления суперблока (`serialize_superblock_to_file`, `deserialize_superblock_from_file`) `inode'ов` и узлов типа `filetype`;

2) рекурсивные функции сериализации и десериализации дерева каталогов (`serialize_filetype_to_file`, `deserialize_filetype_from_file`);

3) функции для разбора путей: получения имени файла (`get_file_name`) и директории (`get_file_path`).

`restore_file_system()` — восстанавливает состояние файловой системы из сохранённых файлов или инициализирует новую файловую систему, если данные отсутствуют.

Поведение:

1) проверяет наличие файлов суперблока и структуры файловой системы.

2) если они существуют, программа запрашивает у пользователя подтверждение на форматирование.

3) если файлы отсутствуют — создаёт новую файловую систему (инициализация суперблока, корневого каталога, сохранение состояния).

`check_superblock_integrity()` — проверяет целостность суперблока файловой системы, включая размеры основных структур: блока данных и битмапов.

Возвращаемое значение:

1) `true` — если суперблок соответствует всем ожидаемым параметрам;

2) `false` — если есть несоответствия в размере данных или битмапов.

`check_file_structure_integrity()` — проверяет корректность корневого узла (`root`) и запускает рекурсивную проверку всей файловой иерархии.

Зависимости:

1) `print_debug()` — для отладочного вывода;

2) `check_filetype_node()` — реализует основную проверку структуры дерева.

`root_dir_init()` — инициализирует корневую директорию файловой системы. Это ключевой этап при создании новой файловой структуры, так как именно с корня начинается вся иерархия. Функция выделяет память под структуру `filetype` для корня и ассоциированный с ней `inode`, настраивает права доступа, владельца, временные метки, и другие параметры. Далее корню присваивается уникальный номер `inode`, выделенный через `find_free_inode()`. В конце сохраняется состояние файловой системы на диск. Если на любом этапе происходит ошибка (например, нехватка памяти или невозможность получить свободный `inode`), происходит корректная очистка и завершение инициализации.

Основные действия:

1) резервирование `inode` с индексом 1;

2) выделение и обнуление памяти под `root`;

3) установка основных атрибутов `root`;

4) создание и настройка `inode` для корня;

5) получение свободного номера `inode`;

6) финальные параметры `inode`;

7) сохранение состояния файловой системы;

Основные заголовочные файлы:

1) файл `inode.h` — содержит определение структуры `inode`, описывающей метаданные файлов и директорий (права, размеры, блоки, временные метки и т.д.), а также объявления функций для работы с инодами и деревом директорий.

2) `filetype.h` — описывает структуру `filetype`, представляющую файл или директорию как узел в дереве. Также содержит прототип функции поиска по пути и внешние ссылки на глобальные объекты.

3) `superblock.h` — определяет структуру `superblock`, которая хранит глобальную информацию о состоянии файловой системы: блоки данных, битмапы инодов и блоков. Также содержит константу размера блока и функции инициализации суперблока и поиска свободного блока.

4 ОПИСАНИЕ АЛГОРИТМОВ РЕШЕНИЯ ЗАДАЧ

В данном разделе рассмотрены алгоритмы работы четырёх функций:

1) `save_system_state()`;

2) `load_file_structure()`;

3) `check_all_inodes()`;

4) `root_dir_init()`;

4.1 Разработка схем алгоритмов

Схема алгоритма функции, которая сохраняет текущее состояние файловой системы, `save_system_state()` приведена в приложении А.

Схема алгоритма функции, которая восстанавливает состояние файловой системы из сохранённых данных, `load_file_structure()` приведена в приложении Б.

4.2 Разработка алгоритмов

Функция `root_dir_init()`:

1) начало;

2) входные данные отсутствуют;

3) выходные данные:

функция `void`;

4) установка `s_block.inode_bitmap[1]` в `1`;

5) выделение памяти для `root` с помощью `malloc()`. Если память не выделена, переход на шаг 6. Иначе переход на шаг 7;

6) вывод сообщения `Failed to allocate memory for root`. Завершение функции. Переход на шаг 18;

7) очистка выделенной памяти для `root` с помощью `memset()`;

8) копирование пути `/` и имени `/` в `root->path и root→name;`

9) выделение памяти для `root->inum` с помощью `malloc()`. Если память не выделена, переход на шаг 10. Иначе переход на шаг 11;

10) вывод сообщения `Failed to allocate memory for inode`. Освобождение памяти для `root` и завершение функции. Переход на шаг 18;

11) очистка выделенной памяти для root->inum с помощью memset();

12) инициализация полей структуры inode;

13) инициализация полей root;

14) вызов find_free_inode() для поиска свободного inode. Если find_free_inode() возвращает -1, переход на шаг 15;

15) вывод сообщения Failed to find a free inode. Очистка файловой системы с помощью cleanup_filesystem(). Освобождение памяти root->inum и root. Переход на шаг 18;

16) установка root->inum->number в найденный индекс inode;

17) вызов save_system_state() для сохранения текущего состояния;

19) конец.

Функция check_all_inodes():

1) начало;

2) входные данные:

filetype *node — указатель на узел файловой системы;

3) выходные данные:

true — все inodes корректны;

false — обнаружена ошибка в одном из inodes;

4) если node == NULL, то возврат true и переход на шаг 10;

5) если node→inum != NULL, вызов check_inode_integrity(node→inum). Если check_inode_integrity() возвращает false, переход на шаг 9;

6) проход по всем дочерним узлам node->children[i] (от i = 0 до node->num_children – 1);

7) для каждого дочернего узла:

Вызов check_all_inodes(node→children[i]);

Если check_all_inodes() возвращает false, переход на шаг 9;

8) если все проверки пройдены успешно, возврат true и переход на шаг 10;

9) возврат false — обнаружена ошибка в одном из inodes;

10) конец.

5 РУКОВОДСТВО ПОЛЬЗОВАТЕЛЯ

5.1 Инструкция по использованию программы

Программа использует систему сборки на основе `Makefile`, что упрощает процесс компиляции, настройки и установки. В `Makefile` определены следующие основные правила:

1) компиляция программы:

Выполняется с помощью команды `make`, которая создает два исполняемых файла в каталоге `bin`:

`mkfs.sfs` — утилита для создания файловой системы;

`fsch` — утилита для проверки файловой системы.

2) отладочная сборка:

Выполняется командой `make debug`, которая добавляет флаги `-g -O0` для включения отладочной информации;

3) релизная сборка:

Выполняется командой `make release`, которая добавляет флаги `-O2` для оптимизированной сборки;

4) очистка проекта:

Выполняется командой `make clean`, которая удаляет директории `build` и `bin`, включая все объектные и исполняемые файлы.

5.2 Результаты работы программы

Запуск утилиты форматирования:

```
$ ./mkfs.sfs ../Simple-File-System-main/build/release
Filesystem already exists. Format it? (y/N): y
Formatting filesystem...
Filesystem formatted successfully.
```

Запуск утилиты проверки целостности файловой системы:

```
$ ./fsch ../Simple-File-System-main/build/release

Starting filesystem check...
Checking inode integrity...
```

```
Inode integrity check passed.

Filesystem is healthy!
```

Запуск утилиты проверки после работы в файловой системе в режиме:

```
$ ./fsch ../Simple-File-System-main/build/release –debug

=== DEBUG MODE ENABLED ===

Starting filesystem check...

========= Starting Superblock Integrity Check ===========

[1/3] Checking data blocks size... OK
[2/3] Checking bitmap sizes... OK
[3/3] Verifying bitmap consistency... OK

=== Superblock Check PASSED ===

========== Starting File Structure Integrity Check ======
Checking root directory structure...
Checking node '/' (type: directory, valid: 1)

=== File Structure Check PASSED ===
Checking inode integrity...

============= Checking inode 2 ======

[1/6] Checking inode number... OK (2)
[2/6] Checking block count... OK (0 blocks)
[3/6] Checking timestamps...
- Access time: 1749311704 - OK
- Modification time: 1749244765 - OK
- Change time: 1749244765 - OK
- Birth time: 1749244765 - OK
[4/6] Checking data blocks...
```

```
    [5/6] Checking permissions... OK (40777)
    [6/6] Checking user/group IDs... OK (UID: 1000, GID: 1000)

    === Inode 2 check COMPLETE - VALID ===


    ================== Checking inode 3 ===============

    [1/6] Checking inode number... OK (3)
    [2/6] Checking block count... OK (0 blocks)
    [3/6] Checking timestamps...
    - Access time: 1749311721 - OK
    - Modification time: 1749311721 - OK
    - Change time: 1749311721 - OK
    - Birth time: 1749311721 - OK
    [4/6] Checking data blocks...
    [5/6] Checking permissions... OK (40777)
    [6/6] Checking user/group IDs... OK (UID: 1000, GID: 1000)
    === Inode 3 check COMPLETE - VALID ===

    Inode integrity check passed.\

    === SUMMARY ===
    Superblock: OK
    File structure: OK
    Inodes: OK

    Filesystem is healthy!
```

ЗАКЛЮЧЕНИЕ

В рамках курсового проекта была разработана утилита, реализующая базовую файловую систему с возможностью форматирования и проверки целостности её структур. Основное внимание в проекте было уделено моделированию компонентов файловой системы — суперблока, `inode`-таблицы и дерева директорий — а также созданию инструментов для их сериализации и десериализации. Особое место занимает реализация функций проверки целостности (`check_superblock_integrity()`, `check_file_structure_integrity()`), что позволяет отслеживать и выявлять логические ошибки и повреждения данных при загрузке или работе с системой.

Модульный подход обеспечил чёткое разделение ответственности: `superblock.c`, `inode.c`, `filetype.c`, `utilities.c` и другие модули реализуют отдельные аспекты файловой системы, что упрощает отладку, сопровождение и расширение функционала.

Форматирование файловой системы подразумевает создание новой структуры с корневой директорией и обнулением всех битовых карт и таблиц. При этом сохраняются базовые метаданные, включая временные метки, идентификаторы пользователя и группы, права доступа и структуру дерева директорий.

Для повышения устойчивости и надёжности реализованы механизмы автоматической валидации целостности данных, как при запуске, так и после выполнения операций. Вывод диагностических сообщений и пошаговая проверка структуры корневой директории, метаданных `inode` и размеров битовых карт позволяют оперативно диагностировать сбои и ошибки.

Разработанная утилита может служить основой для более сложной пользовательской файловой системы с поддержкой операций записи, чтения, создания и удаления файлов, а также для системного моделирования или преподавания основ организации данных в операционных системах. Архитектура проекта легко расширяема — в дальнейшем возможно добавление журналирования, кэширования, управления правами пользователей и других функциональных блоков.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Программа управления компиляцией GNU make [Электронный ресурс]. – 2025. – Режим доступа : https://www.opennet.ru/make_compile/

[2] Структура и работа файловых систем UNIX-подобных ОС [Электронный ресурс]. – 2025. – Режим доступа : https://www.opennet.ru/base/dev/fs_unix_structure.txt

[2] Основы POSIX. Работа с системными вызовами в C [Электронный ресурс]. – 2025. – Режим доступа : https://habr.com/ru/articles/440436/

ПРИЛОЖЕНИЕ А

(обязательное)

Схема алгоритма функции `save_system_state()`

ПРИЛОЖЕНИЕ Б

(обязательное)

Схема алгоритма функции `load_file_structure()`

ПРИЛОЖЕНИЕ В

(обязательное)

Ведомость документов