Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей

Кафедра информатики

Дисциплина: Операционные системы и среды

К защите допустить:
И.О. Заведующего кафедрой информатики
_____ С. И. Сиротко

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

НИЗКОУРОВНЕВЫЙ РЕДАКТОР БЛОЧНОГО УСТРОЙСТВА УРОВНЯ СЕКТОРОВ (NCURCES)

БГУИР КП 1-40 04 01 026 ПЗ

Студент К. Ю. Фроленко

Руководитель Н. Ю. Гриценко

Нормоконтролёр Н. Ю. Гриценко

Минск 2025

СОДЕРЖАНИЕ

Введение 5
1 Архитектура вычислительной системы 7
1.1 Структура и архитектура вычислительной системы 7
1.2 История, версии и достоинства 8
1.3 Обоснование выбора вычислительной системы 9
1.4 Анализ выбранной вычислительной системы 10
1.5 Заключение 11
2 Платформа программного обеспечения 11
2.1 Структура и архитектура платформы 12
2.2 История, версии и достоинства 12
2.3 Обоснование выбора платформы 13
2.4 Анализ программного обеспечения для написания программы 13
2.5 Заключение 14
3 Теоретическое обоснование разработки программного продукта 15
3.1 Обоснование необходимости разработки 15
3.2 Использованные технологии программирования 16
3.3 Взаимосвязь архитектуры системы с программным обеспечением 17
3.4 Заключение 18
4 Проектирование функциональных возможностей программы 19
4.1 Введение в функциональные возможности 19
4.2 Описание основных функций программного обеспечения 19
4.3 Структура программного обеспечения 21
4.4 Обеспечение эффективности и оптимизации 22
4.5 Возможности дальнейшего расширения функциональности 22
4.6 Заключение 23
5 Архитектура разрабатываемой программы 24
5.1 Общая структура программы 24
5.2 Описание функциональной схемы программы 24
5.3 Описание блок схемы алгоритма программы 25
5.4 Обработка неопределенных результатов 26
5.5 Заключение 27
Заключение 28
Список использованных источников 29
Приложение А (обязательное) Справка о проверке на заимствования 31
Приложение Б (обязательное) Листинг программного кода 32
Приложение В (обязательное) Функциональная схема алгоритма, реализующего программное средство 41
Приложение Г (обязательное) Блок схема алгоритма, реализующего программное средство 42
Приложение Д (обязательное) Графический интерфейс пользователя 43
Приложение Е (обязательное) Ведомость курсового проекта 44

ВВЕДЕНИЕ

В современном мире управление информационными потоками и обеспечение надежного хранения данных являются одними из ключевых задач для любой вычислительной системы. Эффективное управление информацией требует не только работы с высокоуровневыми программными средствами, но и понимания принципов функционирования устройств хранения данных на низком уровне. Блочные устройства, такие как жёсткие диски, *SSD* и другие носители, представляют собой основу для формирования файловых систем, и их корректная работа критически важна для обеспечения стабильности и безопасности информационных систем. Низкоуровневые операции с этими устройствами, такие как чтение и запись данных на уровне секторов, позволяют получить прямой доступ к «сырым» данным, что даёт возможность проводить детальную диагностику, восстановление данных и анализ внутренних структур хранения.

Особое значение данный подход приобретает в условиях, когда традиционные методы работы с файловыми системами оказываются недостаточными для решения сложных задач, связанных с анализом ошибок, восстановлением информации после сбоев или атаками злоумышленников. Прямой доступ к блочным устройствам позволяет специалистам не только выявлять скрытые неисправности аппаратных средств, но и оптимизировать алгоритмы работы систем резервного копирования, шифрования и восстановления данных. В последние годы наблюдается значительный рост объёмов обрабатываемой информации, что требует разработки новых методов анализа и управления данными. В этом контексте разработка специализированных утилит для работы с блочными устройствами становится актуальной не только для научных исследований, но и для практических задач в области системного администрирования и информационной безопасности.

Использование библиотеки *ncurses* для создания текстового интерфейса является логичным выбором для реализации подобных утилит. *Ncurses*, разработанная для *Unix*-подобных систем, обеспечивает высокую переносимость и гибкость при работе в терминальном режиме. Благодаря ей можно создать интуитивно понятный и легковесный интерфейс, который позволит специалистам быстро взаимодействовать с программой даже на системах с ограниченными ресурсами. Этот подход снижает требования к оборудованию и повышает оперативность работы с утилитой, что особенно важно при проведении диагностики в режиме реального времени.

Целью курсового проекта является разработка низкоуровневого редактора блочного устройства, позволяющего осуществлять просмотр и редактирование данных на уровне секторов с использованием текстового интерфейса на базе *ncurses*. Для достижения поставленной цели необходимо решить следующие задачи:

1 Анализ архитектуры вычислительной системы и особенностей работы блочных устройств. Изучить структуру современных вычислительных систем, особенности функционирования блочных устройств в *Linux* и методы доступа

к «сырым» данным, а также рассмотреть проблемы безопасности и устойчивости при работе с низкоуровневыми операциями.

2 Исследование возможностей библиотеки *ncurses*. Оценить функциональные возможности и ограничения *ncurses* для создания удобного и интуитивно понятного текстового интерфейса, рассмотреть способы оптимизации работы терминала при большом объёме отображаемых данных, а также изучить методы обработки пользовательского ввода в условиях ограниченных ресурсов.

3 Разработка алгоритмов работы с блочным устройством. Спроектировать и описать алгоритмы для чтения, редактирования и записи данных на уровне секторов, обеспечить возможность тестирования с использованием файлов-образов, а также проанализировать потенциальные риски и предложить методы их минимизации.

4 Проектирование архитектуры разрабатываемой программы. Определить основные модули программного обеспечения, описать их взаимодействие и структуру, разработать функциональные и блок-схемы алгоритмов, а также обосновать выбор структуры программы с точки зрения масштабируемости, модульности и удобства дальнейшей поддержки.

5 Реализация программного продукта. Написать программный код на языке *C/C++* в среде *Visual Studio Code*, провести комплексное тестирование и отладку созданного редактора, обеспечить документирование кода и подготовку его к дальнейшему расширению функционала, а также провести сравнительный анализ эффективности разработанных алгоритмов.

Реализация данного проекта позволит не только продемонстрировать практические навыки работы с низкоуровневыми операциями ввода-вывода, но и углубленно изучить вопросы взаимодействия программного обеспечения с аппаратными средствами. Успешное выполнение проекта даст возможность оценить эффективность выбранных технологий и архитектурных решений, а также подчеркнуть значимость специализированных инструментов для анализа, диагностики и восстановления данных. Кроме того, разработанный редактор может стать основой для дальнейших исследований и разработки более сложных систем, способных работать в режиме реального времени, что особенно важно в условиях постоянно растущих объёмов обрабатываемой информации и усложнения современных вычислительных систем.

1 АРХИТЕКТУРА ВЫЧИСЛИТЕЛЬНОЙ СИСТЕМЫ

1.1 Структура и архитектура вычислительной системы

В данном проекте в качестве вычислительной системы выбрана операционная система *Linux*, дистрибутивом которой является *Ubuntu*. *Ubuntu* считается одним из самых стабильных и широко используемых дистрибутивов, обеспечивающих открытый доступ к аппаратным ресурсам через файловую систему [1]. Такой подход позволяет осуществлять прямое взаимодействие с блочными устройствами посредством специальных файлов, расположенных в системном каталоге. Возможность выполнять операции чтения и записи данных на уровне секторов даёт разработчику высокий контроль над операциями ввода-вывода, что является критически важным для разработки программ, работающих с «сырыми» данными носителя.

Модульная архитектура *Ubuntu* предусматривает разделение ядра системы, драйверов и прикладного программного обеспечения, что позволяет гибко настраивать систему и оптимизировать её работу под специфические задачи проекта. В рамках данной работы реализуется текстовый интерфейс на основе библиотеки *ncurses*, которая нативно поддерживается в *Linux* и позволяет создавать лёгкие и интуитивно понятные консольные приложения [2]. Для наглядного иллюстрирования принципов взаимодействия компонентов можно использовать готовые схемы из открытых источников. Например, блок-схема, демонстрирующая взаимодействие приложения, файловой системы и блочного устройства, представлена на рисунке 1.1. Такие схемы помогают понять, как данные проходят от пользовательского ввода до непосредственного доступа к физическим устройствам, а также как осуществляется обработка информации на низком уровне.

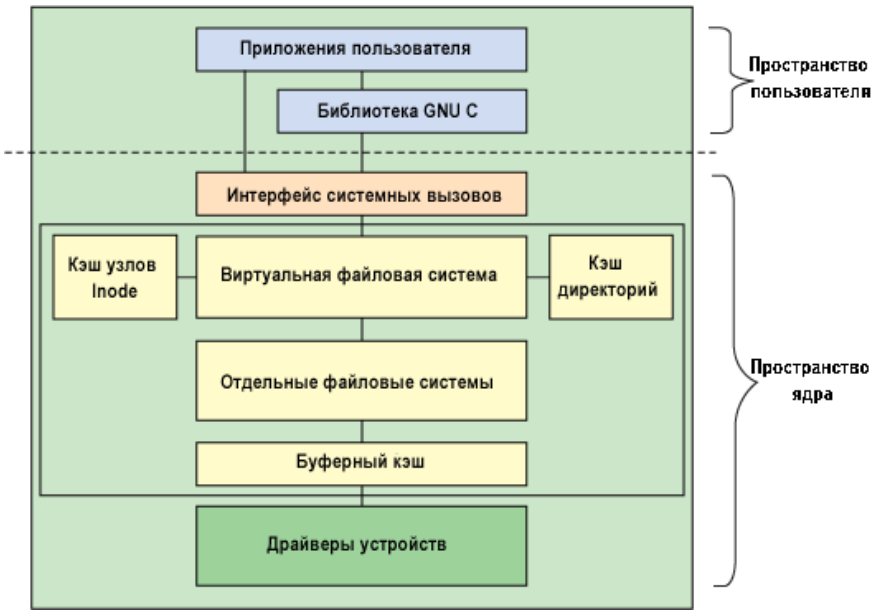

Рисунок 1.1 – Блок схема взаимодействия приложения, файловой системы и блочного устройства

В данном разделе подробно описываются как теоретические основы работы с блочными устройствами, так и практические аспекты реализации доступа к данным, что позволяет создать надёжный фундамент для последующей разработки редактора. При этом особое внимание уделяется принципам взаимодействия на уровне секторов, методам кэширования и корректной обработке ошибок, а также вопросам безопасного тестирования с использованием псевдо-образов, позволяющих оттачивать навыки работы с «сырыми» данными без риска повреждения реальных носителей. Такой подход обеспечивает глубину понимания всех процессов, лежащих в основе низкоуровневого программирования, и формирует необходимую базу для дальнейшей реализации функционала редактора.

1.2 История, версии и достоинства

История развития операционных систем на базе *Linux* начинается в начале 1990-х годов, и за это время платформа претерпела значительные изменения, обеспечив себе репутацию высокостабильного, безопасного и производительного решения [3]. Важным фактором стала массовая поддержка сообщества разработчиков, которое активно вносило вклад в улучшение ядра и вспомогательных инструментов, благодаря чему *Linux* со временем вышла за рамки сугубо экспериментальной платформы и получила широкое признание в корпоративном сегменте, сфере высокопроизводительных вычислений и встраиваемых системах. Среди множества дистрибутивов особое место занимает *Ubuntu*, которая благодаря удобству установки, регулярным обновлениям и активному сообществу разработчиков получила широкое распространение в мире открытого программного обеспечения.

Преимущества *Ubuntu* заключаются в её открытости: доступ к исходному коду позволяет детально анализировать внутреннюю архитектуру системы, вносить необходимые модификации и оптимизировать её под конкретные задачи, что особенно важно при работе с низкоуровневым программированием [4]. Кроме того, *Ubuntu* обладает широким набором инструментов для диагностики и управления блочными устройствами. Использование таких утилит, как *fdisk*, *dd* и *losetup*, позволяет разработчику оперативно решать задачи по восстановлению и анализу данных. Подобная гибкость даёт возможность эффективно адаптировать систему как для исследовательских целей, так и для промышленной эксплуатации, обеспечивая надёжную платформу для разработчиков любых уровней.

Нативная поддержка текстовых интерфейсов посредством библиотеки *ncurses*, интегрированной в *Ubuntu*, даёт возможность создавать эффективные консольные приложения без необходимости обращения к сторонним библиотекам, что часто встречается в других операционных системах. Для иллюстрации эволюции платформы и преимуществ открытого программного обеспечения в *Linux* на рисунке 1.2 приведена диаграмма, отражающая

ключевые этапы развития системы и демонстрирующая, как постепенно формировались её основные достоинства.

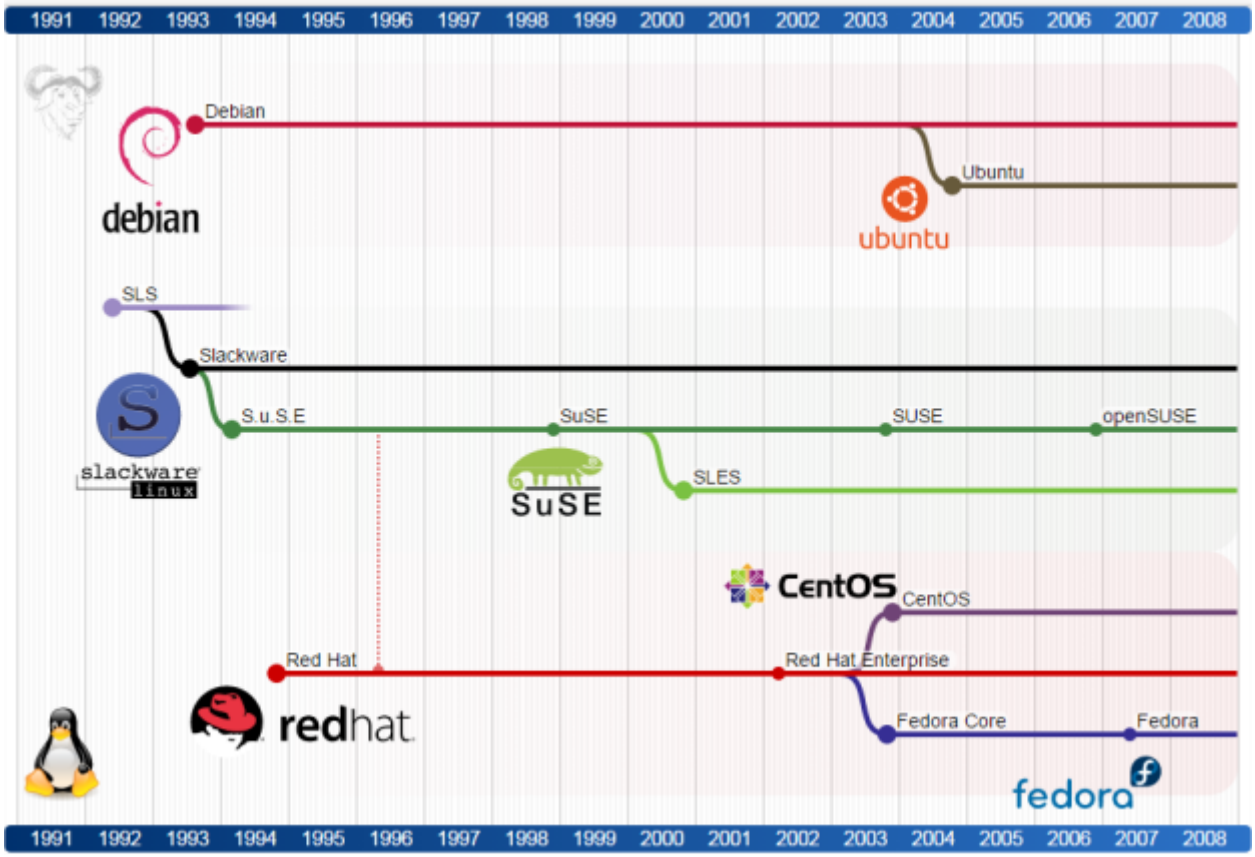

Рисунок 1.2 – Диаграмма развития *Linux* и ключевые этапы эволюции открытых операционных систем

Таким образом, богатая история *Ubuntu* и её технические преимущества делают эту систему надёжной базой для разработки проектов, требующих прямого доступа к аппаратным ресурсам и точного контроля над операциями ввода-вывода.

1.3 Обоснование выбора вычислительной системы

Выбор операционной системы *Linux*, дистрибутива *Ubuntu*, для реализации проекта обусловлен несколькими ключевыми факторами. Прежде всего, прямой доступ к блочным устройствам через файловую систему позволяет работать с данными на уровне секторов без лишних абстрактных слоёв, что существенно повышает точность и скорость выполнения операций. Кроме того, в *Ubuntu* доступны отлаженные средства разработки и отладки, такие как стандартные компиляторы и отладчики, что значительно упрощает процесс создания и оптимизации низкоуровневых приложений.

Нативная поддержка библиотеки *ncurses* является важным преимуществом при создании текстового интерфейса. В отличие от других операционных систем, где для обеспечения аналогичной функциональности

приходится прибегать к дополнительным решениям, *Linux* обеспечивает стабильную и эффективную работу консольных приложений «из коробки». Это позволяет сосредоточиться на реализации основной функциональности редактора без дополнительных затрат времени на настройку среды.

Открытая архитектура *Ubuntu* также играет существенную роль – возможность анализа и модификации исходного кода системы даёт разработчику гибкость в настройке платформы под специфические задачи проекта. Сравнительный анализ характеристик *Linux* и *Windows* представлен в таблице 1.1, что наглядно д абсемонстрирует преимущества выбранной системы для работы с блочными устройствами.

Таблица 1.1 – Сравнение *Linux* и *Windows* для работы с блочными устройствами

Параметр	*Linux*	*Windows*
Доступ к «сырым» устройствам	Прямой доступ через файловую систему	Доступ возможен, но требует дополнительных средств и прав
Набор системных вызовов	Полный, стандартизированный	Ограниченный, часто зависит от проприетарных решений
Поддержка текстовых интерфейсов	Нативная (*ncurses*)	Реализуется через дополнительные библиотеки (например, *PDCurses*)
Сообщество и документация	Обширное и активное	Меньше информации для задач низкоуровневой работы

Функциональные возможности, высокая степень контроля и удобство разработки в *Ubuntu* делают её оптимальным выбором для реализации данного проекта.

1.4 Анализ выбранной вычислительной системы

Подробный анализ показал, что *Ubuntu* обладает всеми необходимыми характеристиками для разработки низкоуровневого редактора блочного устройства. Система позволяет работать как с реальными блочными устройствами, так и с виртуальными образами, что обеспечивает возможность безопасного тестирования приложения без риска повреждения данных. Для обеспечения дополнительной безопасности и упрощения отладки в рамках данного проекта предусмотрено использование именно псевдо-образов (например, файлов или **.iso*), а не реальных устройств [5]. Разработка на языке *C/C++* в *Ubuntu*, с использованием стандартных компиляторов и библиотек,

способствует эффективному созданию и оптимизации программного кода. Надёжность и стабильность платформы гарантируют корректное выполнение низкоуровневых операций, а интеграция с *ncurses* обеспечивает создание удобного текстового интерфейса для конечного пользователя.

В таблице 1.2 представлен сравнительный анализ режимов работы с блочными устройствами, позволяющий увидеть преимущества использования виртуальных образов для тестирования по сравнению с работой с реальными устройствами.

Таблица 1.2 – Сравнительный анализ режимов работы с блочными устройствами

Режим работы	Преимущества	Недостатки
Работа с файловыми образами	Безопасность и возможность тестирования без риска повреждения данных	Может не учитывать все специфические характеристики реальных устройств
Работа с реальными устройствами	Полноценное тестирование и отражение реальных условий	Требует повышенных привилегий и несёт риск возникновения ошибок

Таким образом, *Ubuntu* является высокоэффективной и гибкой платформой, которая удовлетворяет всем требованиям проекта. Это позволяет создать надёжный и масштабируемый редактор, работающий как в условиях лабораторного тестирования, так и при эксплуатации с реальными аппаратными носителями.

1.5 Заключение

Подробный анализ архитектуры вычислительной системы демонстрирует, что операционная система *Linux*, дистрибутивом которой является *Ubuntu*, является оптимальной платформой для разработки низкоуровневого редактора блочного устройства. Прямой доступ к аппаратным ресурсам через файловую систему, широкий набор системных вызовов и наличие отлаженных средств разработки создают благоприятные условия для реализации программного обеспечения, способного работать с «сырыми» данными. Использование языка *C/C++* в сочетании с библиотекой *ncurses* даёт возможность создать компактное, высокопроизводительное и удобное приложение с текстовым интерфейсом.

Для повышения надёжности и упрощения отладки в проекте предусмотрена работа с псевдо-образами (файлами или **.iso*) вместо реальных устройств, что помогает избежать потенциальных ошибок и повреждения данных. Проведённый анализ подчёркивает высокую эффективность и перспективность выбранной вычислительной системы для дальнейшей разработки и оптимизации программного продукта.

2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Структура и архитектура платформы

Программная платформа, в рамках которой создаётся низкоуровневый редактор блочного устройства, включает в себя операционную систему *Ubuntu*, набор инструментов для сборки и отладки (*GCC*, *GDB*, *Make* или *CMake*), библиотеку *ncurses*, а также различные системные утилиты. Подобная конфигурация позволяет эффективно разрабатывать программы, взаимодействующие с аппаратными ресурсами напрямую, и обеспечивает доступ ко всем необходимым функциям ядра *Linux*. Благодаря модульной архитектуре, каждая часть платформы (компиляторы, библиотеки, системные вызовы) может быть обновлена или заменена без кардинальных изменений в остальных компонентах. Этот подход даёт разработчику высокую степень гибкости, что особенно важно при работе с «сырыми» данными, требующими тонкого контроля над операциями ввода-вывода [6].

Важной особенностью структуры является тесная интеграция *ncurses* и стандартных системных библиотек, позволяющая программно управлять терминалом. Приложение, написанное на *C/C++* и использующее *ncurses*, может выводить сложное текстовое меню, динамически обновлять части экрана, а также корректно обрабатывать ввод с клавиатуры. При этом взаимодействие с блочными устройствами осуществляется через *dev*, что означает прямой доступ к секторам, минуя высокоуровневые файловые *API* [7]. В результате разработчик может создавать одновременно удобные для пользования консольные интерфейсы и манипулировать физической структурой носителя, используя одну и ту же платформу.

2.2 История, версии и достоинства

Развитие программных инструментов для *Linux* тесно переплетается с историей свободных проектов *GNU* и идёт с 1980-х годов, когда появились первые версии *GCC*, *GDB* и *Make*. Со временем эти средства стали де-факто стандартом для *Unix*-подобных систем, так как позволяли компилировать и отлаживать программы на низком уровне без необходимости в проприетарных решениях [8]. Библиотека *ncurses*, эволюционировавшая из *curses*, заняла особое место, дав возможность создавать полноценные текстовые интерфейсы, не зависящие от графической среды. *Ubuntu*, появившаяся в 2004 году, упростила распространение этих инструментов, объединив их в системных репозиториях и предоставив удобный менеджер пакетов *apt* [9].

Достоинством выбранной платформы является стабильность и открытость, унаследованные от классических *Unix*-подходов. Компиляторы (*GCC, G++*), входящие в *GNU Toolchain*, обеспечивают поддержку множества языков и флагов оптимизации, а *GDB* позволяет пошагово отлаживать работу программы, вплоть до уровня регистров, что особенно ценно при возникновении критических ошибок на уровне «сырого» ввода-вывода [10].

Ncurses предоставляет богатый функционал для организации окон и обработки пользовательского ввода, при этом программы на её основе могут работать даже в режиме минимального окружения без графического сервера. Такое сочетание гибкости и простоты делает *Ubuntu* оптимальным выбором для проектирования низкоуровневых утилит [11].

2.3 Обоснование выбора платформы

Основанием для выбора именно *Ubuntu* и *GNU*-инструментов служит высокая доступность всех необходимых компонентов «из коробки». Разработчику не требуется искать сторонние библиотеки или вручную собирать компилятор, поскольку всё базовое окружение можно установить за несколько минут через стандартные репозитории [12]. В дополнение к этому, открытость кода, характерная для *Linux*, даёт возможность при необходимости анализировать или даже модифицировать внутренние механизмы ядра, что бывает нужно при отладке низкоуровневых операций.

Дополнительным аргументом выступает удобство тестирования. Работа с псевдо-образами (файлами, которые могут монтироваться как блочные устройства) не только повышает безопасность, но и позволяет легко имитировать различные сценарии использования, меняя параметры образа (размер, структуру, наличие разметки и т. д.) [13]. Разработанная утилита может последовательно читать и писать сектора, в то время как система воспринимает файл как полноценный диск. Это существенно ускоряет цикл «написание кода – тестирование – отладка», позволяя выявлять и исправлять ошибки без риска порчи реальных накопителей.

Нельзя не отметить и богатую документацию. Официальные руководства, руководства для программистов, страницы *man* и сайты сообществ подробно описывают все аспекты работы с системными вызовами, форматами бинарных файлов, а также сложные случаи настройки окружения [14]. Всё это делает платформу максимально прозрачной и дружелюбной к разработчику.

2.4 Анализ программного обеспечения для написания программы

Процесс написания низкоуровневого редактора блочного устройства включает несколько этапов. Сначала создаётся структура проекта (директории для исходных файлов, заголовков, скриптов сборки), а затем выбирается система сборки – *Make* или *CMake* – в зависимости от предпочтений команды. Исходный код, содержащий вызовы системных функций чтения и записи на блочном устройстве, компилируется *GCC*, который поддерживает целый ряд ключей оптимизации и предупреждений (например, *-O2, -Wall*), позволяющих на ранней стадии выявлять потенциальные ошибки.

Наиболее сложная часть разработки обычно связана с отладкой. Для этого используется *GDB*, дающий возможность останавливать выполнение программы, просматривать содержимое памяти и регистров, пошагово

анализировать логику. Если требуется посмотреть, какие именно системные вызовы совершает редактор, в ход идёт *strace* или *ltrace*. Первый отображает обращения к ядру *Linux* (например, чтение сектора), а второй помогает отслеживать вызовы библиотечных функций. Параллельно возможно использование инструментов монтирования псевдо-образов, чтобы проверять корректность работы с виртуальными дисками в безопасном режиме. В результате выстраивается чёткая методика: небольшой участок кода – сборка – тест на псевдо-образе – отладка; это ускоряет выявление логических ошибок и упрощает внесение правок.

Применение библиотеки *ncurses* придаёт редактору более удобный вид, чем традиционные консольные приложения с минимальным интерфейсом. Программа может выводить на экран текущее состояние загруженного сектора, цветом выделять изменённые байты, а также реагировать на команды пользователя без перезапуска (обновляя окно в реальном времени). Подобный подход облегчает ориентирование в массиве двоичных данных, что существенно экономит время при анализе и редактировании «сырой» структуры.

2.5 Заключение

Подводя итоги, можно сказать, что выбранная программная платформа на базе *Ubuntu*, *GNU*-инструментов и библиотеки *ncurses* представляет собой оптимальное решение для разработки низкоуровневого редактора блочного устройства. Платформа обеспечивает прямой доступ к аппаратным ресурсам, гибкость настройки, высокую стабильность и богатый набор средств для сборки и отладки. Использование псевдо-образов позволяет безопасно тестировать операции с секторами, а интерактивный текстовый интерфейс, реализованный с помощью *ncurses*, делает работу с приложением удобной и эффективной.

Кроме того, данная платформа предоставляет широкие возможности для дальнейшей интеграции с современными инструментами автоматизации и мониторинга. Например, разработчик может дополнительно настроить систему для динамического отображения логов и системных показателей в режиме реального времени, что позволяет не только ускорить отладку, но и повысить общую прозрачность работы приложения. Такой комплексный подход гарантирует, что итоговый продукт будет не только функциональным, но и легко масштабируемым, что крайне важно для проектов, связанных с критически важными операциями ввода-вывода.

3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

3.1 Обоснование необходимости разработки

Современные вычислительные системы широко используют высокоуровневые инструменты для работы с данными, скрывающие от пользователя детали внутренней структуры хранения. Операционные системы и прикладное программное обеспечение абстрагируют физические операции чтения/записи секторов, представляя их в виде понятных объектов – файлов и директорий. Такой подход облегчает повседневную работу, но одновременно ограничивает возможности тонкого контроля. В случаях, когда требуется глубокая диагностика сбойных участков диска, анализ структуры нераспределённых областей или исправление повреждённых системных секторов, эти «прозрачные» абстракции могут оказаться препятствием.

Потребность в низкоуровневом редакторе возрастает при возникновении ситуаций, выходящих за рамки стандартных сценариев. Прежде всего, речь идёт о сбоях в загрузочных областях (*MBR* или *GPT*), от которых напрямую зависит способность системы загружаться. Кроме того, достаточно часто встречаются ситуации, когда повреждение метаданных файловой системы не позволяет использовать классические инструменты восстановления: «умные» программы отказываются работать, поскольку считают диск непригодным, а пользователь не может «дойти» до необходимого байта или сектора. Низкоуровневый редактор позволяет вручную просматривать и редактировать любой сектор, в том числе за пределами размеченных разделов, что бывает актуально при ручном восстановлении таблиц разделов или служебных данных [15].

Не менее важен и аспект информационной безопасности. При расследовании инцидентов специалисты по кибербезопасности часто сталкиваются со сложными методами маскировки вредоносного кода, расположенного вне зоны видимости файловой системы, например, в зарезервированных секторах. Возможность напрямую прочитать и проанализировать содержимое таких областей может стать ключом к пониманию методов злоумышленника. Аналогично, эксперты по цифровой криминалистике при работе с образами дисков нередко нуждаются в точечных правках – например, для восстановления структуры раздела, когда автоинструменты бессильны.

Существующие утилиты вроде *dd*, *hexdump* или специализированные скрипты на базе *fdisk* и *parted* не всегда предоставляют удобство и интерактивность: они требуют ручного ввода многих параметров, не обладают функциями отмены (*undo*) или наглядной визуализации байтов. Ошибка в одном параметре может стоить целого раздела или привести к безвозвратной потере данных. Именно поэтому разработка низкоуровневого редактора, способного показать содержимое сектора в удобном текстовом интерфейсе и

предоставляющего механизмы безопасного редактирования, отмены изменений и копирования целых секторов, становится востребованным решением [16].

В образовательном процессе такой редактор также играет значительную роль, поскольку позволяет студентам и начинающим специалистам увидеть, как устроена физическая структура накопителя, понять, что такое сектора и байты, как именно хранятся загрузочные записи, сигнатуры разделов и где располагаются метаданные файловой системы. Практическое изучение этих аспектов способствует более глубокому пониманию архитектуры операционных систем и принципов работы устройств хранения.

3.2 Использованные технологии программирования

Для реализации низкоуровневого доступа к блочным устройствам в операционных системах семейства *UNIX* традиционно применяются языки *C/C++*. Они позволяют напрямую использовать системные вызовы (среди которых *open, pread, pwrite, ioctl*) и гибко управлять памятью. В данном проекте язык *C/C++* выбран в первую очередь из-за его высокой производительности и близости к системному уровню. Это даёт возможность точно задавать смещения и размеры операций чтения/записи, не теряя ресурсов на ненужные абстракции [17].

Создание пользовательского интерфейса, способного динамически отображать содержимое секторов, назначать горячие клавиши для быстрого перехода между байтами или секторами и выводить подсказки, традиционно реализуется через библиотеку *ncurses*. Данная библиотека предоставляет набор функций, позволяющих работать с текстовым терминалом как с «оконной» системой. С *ncurses* проще сформировать заголовок с информацией о текущем секторе, области для вывода шестнадцатеричных значений, а также строку статуса, где пользователь видит описание доступных команд. Библиотека *ncurses* исторически встроена в большинство дистрибутивов *Linux* и *BSD*, что делает приложение переносимым: для его работы не требуется графический сервер или дополнительные проприетарные компоненты.

Важным решением при проектировании редактора является логическая изоляция функционала чтения/записи и интерфейса. Модуль «*Editor Core*» (условно говоря) занимается взаимодействием с устройством, проверяет корректность смещений, выполняет чтение и запись в буфер, обрабатывает ошибки ввода-вывода. В то время как модуль «*UI*», написанный на основе *ncurses*, отвечает за визуальное представление данных и реакцию на пользовательские действия: перемещение курсора, ввод команды на сохранение, редактирование байта и т. п. Подобное разделение даёт программе устойчивость к изменениям: если потребуется доработать логику отображения (например, добавить цветовую подсветку изменённых байтов), это не затронет базовые механизмы чтения/записи.

Применение *C/C++* и *ncurses* оправдано и с точки зрения сложившихся практик системного программирования. По статистике большинства *Unix-*

проектов, языки семейства *C* являются «родными» для низкоуровневых утилит, а библиотеки вроде *ncurses* имеют долгую историю развития и хорошо отточенную функциональность. Их использование позволяет разрабатывать компактный, высокопроизводительный и при этом достаточно наглядный код, понятный другим разработчикам, знакомым с *Unix*-подходом.

3.3 Взаимосвязь архитектуры системы с программным обеспечением

Архитектура операционных систем на базе *Linux* (*Ubuntu, Debian, Fedora* и др.) предусматривает единый способ обращения к устройствам через специальные файлы в каталоге */dev/*. Каждый физический или виртуальный диск, флеш-накопитель, «*loop*»-устройство, а также многие другие блочные устройства представлены там своим дескриптором (к примеру, */dev/sda, /dev/sdb, /dev/loop0*). При открытии файла, соответствующего блочному устройству, приложение получает низкоуровневый доступ к его содержимому, не завися от промежуточных слоёв файловых систем. Это облегчает задачу разработчикам низкоуровневых программ, так как они могут использовать стандартные функции *open, read, write* либо позиционированные вызовы *pread, pwrite* для строгого указания смещения относительно начала носителя.

Подобная модель делает доступ к виртуальным образам (*ISO, IMG*) практически неотличимым от доступа к реальным дискам. Внутри редактора можно переоткрывать разные файлы */dev/loopN*, которые эмулируют блочные устройства, и таким образом безопасно отлаживать логику чтения/записи, не рискуя повредить рабочую систему. Только после полной проверки корректности алгоритмов и интерфейса допускается работа с реальным носителем. Это согласуется с принципами безопасной разработки: сначала тщательное тестирование и отладка в лабораторной среде – затем выверенная эксплуатация в производственных условиях.

С учётом того, что операции с низкоуровневыми данными чреваты серьёзными последствиями (например, одно неверное смещение может стереть важную загрузочную запись), в программном обеспечении сделан упор на дополнительную защиту пользователя от опрометчивых действий. Интерфейс спрашивает подтверждение при особенно опасных операциях (например, «Вставить сектора из буфера?» или «Сохранить изменения на устройство?»), а также ведёт журнал действий для обеспечения возможности отмены (*undo*) или повтора (*redo*) последних модификаций.

В случае автосохранения (при соответствующем флаге запуска) каждое изменение немедленно записывается в устройство, что может быть полезно при ручном восстановлении структуры, когда нужно постоянно проверять результат в сторонней программе (например, перезагрузкой системы или повторным сканированием разделов). Без включённого автосохранения редактор хранит изменения только в памяти до тех пор, пока пользователь явно не сохранит их на диск. Такой подход помогает минимизировать риск

случайной порчи данных: изменения не попадут в накопитель, если пользователь экспериментирует и не нажимает «Сохранить».

3.4 Заключение

Теоретическое обоснование разработки низкоуровневого редактора блочного устройства свидетельствует о необходимости подобных инструментов в сфере системного администрирования, компьютерной криминалистики, отладки и обучения. Активное использование операционных систем с семейством *Linux*, а также высокий уровень автоматизации и абстракции приводят к ситуации, когда доступ к физическим секторам становится «свёрнутым» за механикой высокоуровневых файловых операций. Однако при возникновении нештатных ситуаций, повреждениях или угрозе безопасности столь глубокий и точный контроль над блочным устройством оказывается исключительно важен.

Благодаря опоре на язык *C/C++*, проект получает все преимущества низкоуровневого управления памятью, а библиотека *ncurses* обеспечивает удобный и переносимый интерфейс, не привязанный к графической оболочке. Распределение ответственности между логикой чтения/записи и интерфейсными модулями делает структуру приложения гибкой, а систему журналирования изменений – надёжной с точки зрения защиты от фатальных ошибок оператора. В совокупности такие решения формируют основу для создания надёжного, удобного и многофункционального редактора, который можно рекомендовать как профессионалам, сталкивающимся с проблемами восстановления и анализа, так и студентам, желающим глубоко изучить внутреннее устройство дисковых накопителей.

В дальнейшем данный инструмент может быть расширен дополнительными функциями, позволяющими автоматически распознавать и подсвечивать ключевые области (например, *GPT*, таблицы *FAT*, метаданные *ext4*), предоставлять фильтры для поиска сигнатур или даже интегрироваться с другими системными утилитами для комплексной диагностики. Подобное развитие значительно упростило бы обнаружение потенциально проблемных зон и сократило бы время локализации неисправностей при проведении глубокого аудита носителя. Параллельно можно внедрить механизмы логирования всех произведённых операций, что открывает возможности для применения редактора в судебной экспертизе, где важно детально документировать каждое действие. Таким образом, реализация подобного редактора не только помогает в решении текущих задач анализа и восстановления, но и способствует общему повышению культуры работы с данными, более глубокому пониманию низкоуровневых механизмов операционных систем и развитию комплексных подходов к обеспечению безопасности.

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

4.1 Введение в функциональные возможности

На этапе проектирования функциональных возможностей низкоуровневого редактора блочного устройства необходимо было определить, какие именно операции будут особенно важны для специалистов, работающих с «сырыми» данными носителя. С одной стороны, такой инструмент создаётся для решения реальных практических задач: диагностики нештатных ситуаций, восстановления критических областей диска и тонкой настройки структур разделов. С другой стороны, он должен быть достаточно универсален, чтобы им могли пользоваться студенты, изучающие принципы взаимодействия операционной системы с блочными устройствами.

По мере сбора требований, стало понятно, что ключевой является гибкая работа с отдельными секторами. Пользователь должен иметь возможность быстро находить нужный сектор, считывать из него байты и изменять необходимые участки. При этом одним из центральных моментов является безопасность подобных изменений: любое неверное действие способно вызвать необратимые последствия, вплоть до утраты файловой системы. Поэтому при проектировании функционала сразу предусматривались механизмы «страховки» – такие как журнал изменений (*undo/redo*) и опциональное автосохранение. Немалую роль сыграло и то, что в случае криминалистического анализа каждый шаг редактирования должен быть чётко контролируемым, чтобы минимизировать риск потери доказательств или несогласованных воздействий на носитель.

Особое внимание уделялось удобству интерфейса. Решение работать через библиотеку *ncurses* помогло снять ограничения, связанные с отсутствием графической среды, и сделать программу доступной даже в режиме аварийного восстановления системы. Специалист получает одновременно детальный текстовый вывод и управляемый интерфейс со всплывающими подсказками и возможностью быстрого обновления экрана. Всё это позволяет выполнять сложные операции, используя при этом сравнительно небольшое количество ресурсов.

4.2 Описание основных функций программного обеспечения

В соответствии с поставленными целями и собранными требованиями, в программе реализован набор основных функций, необходимых для детального и безопасного взаимодействия с секторным пространством накопителя:

1 Просмотр содержимого сектора. Приложение отображает текущий сектор в двух форматах: шестнадцатеричном (*HEX*) и символьном (*ASCII*). При этом каждый байт представлен парой *HEX*-цифр, а рядом выводится символ, если его код находится в диапазоне печатаемых символов. Такой

способ просмотра даёт наглядное представление о структуре данных и позволяет выделить «читаемые» фрагменты. Важным преимуществом является то, что пользователь может оперативно переключаться по строкам и столбцам, осматривая содержимое по 16 байтов на строку.

2 Редактирование отдельных байтов. Программа даёт возможность изменять выбранный байт прямо на месте: после ввода двух шестнадцатеричных цифр новое значение фиксируется в буфере, и на экране обновляется соответствующая ячейка. Благодаря такому механизму легко проводить точечные правки, например, восстанавливать испорченные загрузочные записи или корректировать содержимое таблицы разделов. Визуальная подсветка курсора снижает вероятность ошибки: пользователь видит, к какому конкретно байту относится ввод, и может вовремя отменить неверное действие.

3 Переход между секторами. Для перемещения по накопителю редактор поддерживает как пошаговое продвижение к следующему или предыдущему сектору, так и «прыжок» к нужному номеру по запросу пользователя. Это существенно ускоряет навигацию при больших объёмах данных, позволяя быстро перейти к сомнительной области (скажем, к сектору, в котором предположительно находится начала раздела или сигнатура загрузчика).

4 Копирование и вставка целого сектора (*Copy/Paste*). Утилита позволяет сохранить в буфер памяти весь текущий сектор и в дальнейшем «вставить» его поверх другого сектора. Подобный функционал востребован при восстановлении служебных структур, когда нужно быстро продублировать уже «чистые» данные или, наоборот, сделать копию «проблемного» сектора, чтобы продолжить эксперименты. Чтобы исключить риск случайного затирания, перед вставкой программа выводит запрос на подтверждение.

5 Сохранение изменений. В редакторе реализованы два варианта сохранения: пользовательский (ручной) и автоматический (автосохранение). В первом случае внесённые изменения хранятся в оперативной памяти до тех пор, пока оператор не выберет соответствующую команду. Это даёт возможность «передумать» или отменить сделанные правки, не оставляя следа на устройстве. Автосохранение же полезно в ситуациях, когда нужно оперативно фиксировать каждый байт изменения, а объём экспериментов невелик.

6 Отмена и повтор действий (*Undo/Redo*). Данный механизм делает процесс редактирования более гибким и безопасным. Программа ведёт журнал, куда записывает информацию об изменённых значениях и их прежнем состоянии. При желании пользователь может на один (или несколько) шагов вернуться к состоянию, предшествующему неверной правке. «Повтор» (*Redo*) возобновляет отменённую операцию, что полезно при сравнении различных вариантов редактирования.

7 Встроенная справка. Утилита содержит краткие подсказки по основным командам, отображаемые внизу экрана, и более развернутую справку, вызываемую по специальной клавише. Это упрощает работу с

редактором и позволяет быстро освежить в памяти назначение тех или иных команд, не заглядывая во внешнюю документацию.

Реализация перечисленных функций позволила сформировать «костяк» приложения, охватывающий базовые сценарии восстановления данных, исследования повреждённых участков и экспериментов с низкоуровневыми структурами. Каждый пункт решает практические задачи и может быть задействован как по отдельности, так и в комплексе при сложных операциях.

4.3 Структура программного обеспечения

Разработка структуры программного обеспечения велась с учётом необходимости долгосрочной поддержки и расширяемости. Принято решение выделить несколько ключевых модулей, каждый из которых отвечает за свой аспект работы редактора. В частности, базовая логика, связанная с чтением и записью секторов, вынесена в отдельный блок, в котором сосредоточены функции, работающие напрямую с системными вызовами *pread* и *pwrite*. Такой подход упрощает отладку, поскольку ошибки, связанные с неверными смещениями или некорректной обработкой файловых дескрипторов, изолированы и не затрагивают интерфейс.

Слой, ответственный за пользовательский интерфейс, построен на базе библиотеки *ncurses* и обеспечивает взаимодействие с клавиатурой, динамическую перерисовку экранных областей и организацию «окон». В этом же слое обрабатываются основные команды (например, переходы по секторам и редактирование байтов). При этом модуль интерфейса сам не занимается операциями чтения/записи: он передаёт запросы на загрузку сектора и запись правок модулю логики и лишь выводит результат.

Для обеспечения функции «отмены/повтора» изменений (*undo/redo*) в программу интегрирован вспомогательный компонент, который отслеживает внесённые правки. Он хранит в своём буфере предыдущие значения, возвращаться к которым можно при помощи специальных команд. Если пользователь редактировал один-единственный байт, в журнал заносится пара старое/новое значение, если же был скопирован и вставлен целый сектор – сохраняется информация о прежнем и новом состоянии целого блока. Такая гибкая организация позволяет оперативно реагировать на любые изменения, не потребляя избыточных системных ресурсов.

Кроме того, в структуру заложен механизм конфигурации – например, пользователь может выбрать размер сектора (в случае нестандартных устройств или образов) и режим сохранения (автосохранение или ручной). Эти параметры считываются при запуске приложения и передаются модулям логики и интерфейса, что обеспечивает гибкость при работе с различными блочными устройствами, включая псевдо-образы.

В совокупности эти блоки – модуль ввода-вывода, модуль интерфейса и механизм журналирования – образуют целостную систему, которая способна надёжно и наглядно работать с блочными устройствами в режиме реального времени. При желании в программу можно добавить дополнительные уровни

(например, логику автоматического распознавания файловых структур), не нарушая уже существующей модульной организации.

4.4 Обеспечение эффективности и оптимизации

Работа с «сырыми» данными напрямую предполагает высокие требования к корректности и производительности операций ввода-вывода. В первую очередь, в проекте реализована стратегия «один сектор – один буфер». Программа в любой момент оперирует лишь текущим сектором, который хранит в памяти и отражает на экране. При переключении на другой сектор старый буфер сохраняется (если надо) или сбрасывается, а затем считываются байты по новому смещению. Это даёт возможность ограничить использование оперативной памяти и избежать путаницы с частичными данными.

Позиционированные вызовы *pread* и *pwrite* позволяют исключить промежуточное использование *lseek*, сокращая системные расходы. Такой подход особенно полезен при работе с большими объёмами, когда каждая операция ввода-вывода может занимать заметное время, а частые переключения контекста ядра нежелательны.

Ncurses оптимизирован для частичного обновления экрана. Редактор обновляет лишь те участки, которые действительно изменились: например, если пользователь вёл курсором по строкам, перерисовываются лишь строки, где курсор меняется, а не весь экран. При редактировании одного байта меняется только соответствующее двумерное «поле» в текстовом буфере. Это даёт выигрыш в производительности, особенно при удалённом доступе (через *SSH* или аналогичные инструменты), где лишние перерисовки экрана могут приводить к задержкам и «заморозкам» интерфейса.

Механизм *undo/redo* также реализован с учётом оптимизации. При одиночной правке хранится минимальный набор данных: индекс байта, старое и новое значения. Только при операциях копирования и вставки сектора в журнал сохраняется целый блок, который может занимать 512 байт или более, если сектор имеет нетипичный размер. Таким образом, программа не расходует ресурсы на хранение полной истории всех секторов, а лишь фиксирует ключевые отличия между состояниями.

4.5 Возможности дальнейшего расширения функциональности

При создании редактора учитывалось, что область применения подобных утилит может значительно расширяться с течением времени. Уже сейчас потенциальные улучшения могут включать интеграцию механизма анализа и подсветки типовых структур: например, при обнаружении сигнатур *MBR, GPT* или суперглобальных записей файловой системы редактор мог бы выводить дополнительную информацию об их содержимом и значении.

Важной опцией представляется организация поиска по сигнатурам (бинарным последовательностям). Это необходимо в ситуациях, когда нужно найти конкретное смещение, где могут храниться критические данные или

вредоносный код. Редактор мог бы автоматически сканировать носитель и останавливать просмотр на первых совпадениях, что сильно упростило бы работу при больших объёмах.

Помимо этого, возможно добавление более продвинутой справочной системы, включающей примеры редактирования реальных структур (к примеру, таблицы разделов или загрузочных записей). Это могло бы послужить хорошим пособием для студентов, желающих «потрогать руками» низкоуровневые аспекты дисковой архитектуры, и для специалистов, которые ищут сэкономить время при решении типовых проблем.

Наконец, стоит упомянуть о перспективах подключения сторонних модулей или скриптов, позволяющих автоматизировать определённые задачи. Если редактор предоставит *API* для внешних программ, то он сможет становиться частью комплексных решений по автоматическому восстановлению, сбору доказательной базы при расследованиях и анализе нестандартных файловых систем.

4.6 Заключение

Таким образом, при проектировании функциональных возможностей низкоуровневого редактора ставилась задача сочетать наглядность и удобство интерфейса с точностью и гибкостью низкоуровневых операций. Основные функции приложения, включающие просмотр и редактирование байтов, копирование секторов, переходы по заданным смещениям, а также механизмы сохранения и логирования изменений, формируют прочную основу для решения широкого круга задач, связанных с диагностикой, восстановлением и анализом данных.

Структура программы, основанная на модульности и чёткости разграничения ответственности между вводно-выводным слоем, интерфейсом и системой журналирования, делает её гибкой и способной к дальнейшему развитию. Применённые оптимизационные приёмы, такие как позиционированные вызовы ввода-вывода и частичная перерисовка экрана, гарантируют высокую производительность и отзывчивость даже в режиме интенсивного редактирования.

Предложенные направления расширения, среди которых автоматическая идентификация файловых систем и возможности поиска по сигнатурам, помогут в будущем ещё больше повысить ценность и универсальность редактора. В итоге проект, представленный в этой главе, можно рассматривать не только как учебный пример, но и как задел для профессионального инструмента, способного успешно применяться в различных сферах – от восстановления критических систем до криминалистических исследований и образовательных курсов по операционным системам и средам

5 АРХИТЕКТУРА РАЗРАБАТЫВАЕМОЙ ПРОГРАММЫ

5.1 Общая структура программы

При создании низкоуровневого редактора блочного устройства уровень секторов (*NCURSES*) была выбрана модульная структура, позволяющая чётко разграничивать различные аспекты работы приложения. Подобный подход способствует удобству сопровождения кода и облегчает добавление новых функций. В общем виде программу можно представить как совокупность четырёх основных блоков. Первый отвечает за взаимодействие с носителем (реальным или псевдо-образом), используя системные вызовы для точного позиционированного чтения и записи. Второй обеспечивает логику редактирования и ведёт журнал изменений (*undo/redo*), что помогает восстанавливать прежнее состояние секторов при ошибках. Третий блок реализует пользовательский интерфейс на базе *ncurses*, визуализирует содержимое секторов и обрабатывает нажатия клавиш. Наконец, четвёртый блок концентрируется на обработке возможных сбоев и некорректных действий, перехватывая системные ошибки и уведомляя пользователя.

Подобная структура даёт возможность разделять ответственность: если потребуется изменить, к примеру, алгоритм копирования секторов или расширить механизмы редактирования, изменения будут внесены только в один блок, не затрагивая остальные. Модули взаимодействуют через ясно определённые интерфейсы и обмениваются лишь необходимой для решения своих задач информацией. За счёт этого вся система сохраняет целостность и предсказуемость работы, оставаясь при этом достаточно гибкой, чтобы в будущем адаптироваться к новым требованиям.

В листинге исходного кода, приведённом в приложении Б, детально отражено, каким образом каждый модуль реализован в контексте языка *C/C++* и библиотеки *ncurses*. Организация данных в виде отдельных файлов (например, *editor.cpp* для логики редактирования и *main.cpp* для запуска приложения) и заголовков облегчает навигацию по проекту, позволяет эффективно проводить тестирование и искать потенциальные ошибки.

5.2 Описание функциональной схемы программы

Общая логика взаимодействия и переходов между ключевыми блоками редактора представлена на функциональной схеме, которая помещена в приложении В. Эта схема даёт представление о том, как пользовательские запросы преобразуются в системные вызовы чтения или записи, а также каким образом результаты обновляются на экране.

Пользователь запускает редактор с указанием целевого устройства или файла-образа, а также, при необходимости, дополнительных флагов (автосохранение, размер сектора, номер начального сектора). Система проверяет корректность параметров и открывает соответствующий дескриптор устройства. Далее загружается указанный сектор (или нулевой по

умолчанию), после чего выводится информация в двойном формате: шестнадцатеричном и *ASCII*-представлении. Пользователь, взаимодействуя с интерфейсом *ncurses*, перемещает курсор, редактирует байты, копирует и вставляет сектора, а также выполняет сохранение. Все эти действия отражаются в модуле логики редактирования, который поддерживает журнал изменений, что даёт возможность при необходимости отменить или повторить серию правок. В случае включённого автосохранения каждый шаг немедленно записывается на носитель; в противном случае запись происходит при явном выборе команды «сохранить».

Подобная функциональная схема демонстрирует, как пользовательские действия последовательно проходят сквозь интерфейс, обрабатываются в логике редактора, при необходимости вызывают операции ввода-вывода и в итоге возвращаются обратно к интерфейсу для отображения обновлённого состояния. Ключевым элементом этой схемы является журнал изменений, гарантирующий, что даже в случае ошибки со стороны пользователя или сбоя со стороны оборудования (о чём будет сказано далее) оператор не потеряет возможность вернуться к прежнему состоянию данных.

5.3 Описание блок схемы алгоритма программы

Более детальный алгоритм, описывающий пошаговую логику работы редактора от запуска до закрытия, приведён в приложении Г. Данную блок схему условно можно разбить на несколько этапов:

1 Инициализация. На этом этапе идёт проверка переданных при запуске параметров и установка режима работы (например, определение флага автосохранения). Редактор пытается открыть файл-образ или блочное устройство и, при успехе, подготавливает среду *ncurses*, чтобы приступить к взаимодействию с терминалом.

2 Загрузка сектора. После успешной инициализации редактор считывает содержимое первого (или заданного) сектора и формирует внутренний буфер, отражающий текущие байты. Содержимое буфера выводится на экран: каждое слово состоит из двух *HEX*-цифр и, по возможности, конвертируется в *ASCII*. Пользователь видит, куда можно переместить курсор для редактирования.

3 Основной цикл. Алгоритм переходит в режим ожидания команд. Когда пользователь нажимает клавишу, приложение определяет соответствующее действие: перемещение курсора, изменение байта, переход к другому сектору, копирование или вставку, отмену или повтор правки, сохранение, вызов справки или выход. При необходимости операция чтения или записи (*pread/pwrite*) выполняется через модуль ввода-вывода. Если команда предусматривает обновление журнала, изменения заносятся туда и отражаются на экране.

4 Сохранение и возможные ошибки. По мере редактирования (либо после окончания) производится запись изменённых данных. Если пользователь выбрал автосохранение, запись идёт сразу; при ручном режиме программа сообщает, что изменения не зафиксированы, и предлагает

сохранить их перед выходом. В случае непредвиденных ситуаций (ошибки системы ввода-вывода, невозможности записи или неверного номера сектора) срабатывает механизм обработки ошибок, после чего оператор решает, повторять попытку или отменить операцию.

5 Завершение работы. При выборе команды «выход» программа освобождает ресурсы *ncurses*, закрывает дескриптор устройства и завершает цикл работы. Если пользователь не сохранил последние изменения, редактор предлагает сделать это, чтобы избежать потери данных.

Таким образом, блок схема алгоритма характеризуется гибким механизмом взаимодействия и контролем целостности данных, что особенно важно при работе с «сырыми» байтами в потенциально критически важных областях диска.

5.4 Обработка неопределенных результатов

В процессе работы с низкоуровневым редактором не всегда возможно однозначно интерпретировать результат той или иной операции. Например, если пользователь пытается перейти к сектору, которого фактически не существует на устройстве, редактор сталкивается с ситуацией, когда извлечь данные из заданного смещения не представляется возможным. Аналогично, при попытке записи редактор может столкнуться с ограничениями прав доступа, аппаратным сбоем или неверным номером сектора. В подобных случаях возникает состояние неопределённого результата, которое может поставить под угрозу консистентность буфера или самого устройства.

Чтобы свести к минимуму риски и обеспечить предсказуемое поведение, в программе реализовано несколько принципов. Во-первых, перед любой операцией чтения или записи редактор проверяет диапазон секторов и соответствие смещения реальным границам устройства. Если расчёт показывает, что пользователь находится за пределами доступного пространства, редактор не будет пытаться выполнить недопустимую операцию, а вместо этого выдаст соответствующее предупреждение и вернётся к ожиданию следующей команды. Во-вторых, при обнаружении сбоя системные вызовы (*pread/pwrite*) возвращают ошибочное состояние, которое улавливается модулем обработки ошибок. На экране появляется сообщение о невозможности считать или сохранить данные, и пользователю предлагается альтернативное действие, например выбор другого сектора или прерывание редактирования. В-третьих, если при вводе шестнадцатеричного значения обнаруживается, что пользователь вышел за пределы формата (например, ввёл символы, не соответствующие *A–F* или 0–9), такое изменение не будет применено, а программа сигнализирует об ошибочном вводе. Данные при этом сохраняют прежнее состояние, что защищает от случайных необратимых правок.

В случае если невозможно завершить операцию копирования сектора, потому что во внутреннем буфере редактора не оказалось действительных данных для вставки, программа также сообщает, что действие отменено. Такая

тщательная проверка сценариев помогает избежать скрытых неполадок, способных привести к порче системных областей диска или некорректному отображению информации. Все нештатные ситуации, при которых нельзя получить однозначный результат, переводят редактор в безопасный режим ожидания команд, сохраняя уже накопленную в буфере историю.

5.5 Заключение

Выбранная архитектура обеспечивает эффективное объединение низкоуровневого доступа к данным, логики редактирования и визуального интерфейса в одном приложении. Модульная структура даёт возможность гибко расширять функционал: при необходимости можно легко внести изменения в логику обработки секторов либо дополнить редактор новыми функциями анализа. Важную роль в повышении наглядности играет интерфейс на базе *ncurses*, позволяющий в реальном времени отображать текущие байты, подсвечивать изменения и оперативно реагировать на команды пользователя.

Система хранения истории изменений (*undo/redo*) защищает от случайных ошибок при работе с критическими областями, предоставляя возможность быстро вернуться к предыдущему состоянию. Кроме того, программа надёжно обрабатывает нештатные ситуации, такие как неверные значения при вводе или выход за границы носителя. Все возникающие проблемы сразу становятся видимы пользователю, что даёт шанс избежать фатальных сбоев.

Интерфейс, приведённый в приложении Д, наглядно показывает содержимое сектора в шестнадцатеричном и *ASCII*-формате, а также отображает подсказки по ключевым операциям. Подобная визуализация облегчает анализ структуры и упрощает процесс поиска проблемных байтов. В совокупности описанные механизмы делают редактор удобным и безопасным инструментом как для специалистов по восстановлению и диагностике, так и для студентов, которые только осваивают устройство блочных накопителей и принципы взаимодействия с ними на низком уровне.

В перспективе редактор может дополниться автоматическим выявлением файловых структур, поиском сигнатур или дополнительными уровнями проверки при редактировании жизненно важных областей. Это позволит расширить область его применения – от сценариев аварийного восстановления до криминалистических исследований и детального изучения архитектуры современных систем хранения.

ЗАКЛЮЧЕНИЕ

В ходе курсового проекта «Низкоуровневый редактор блочного устройства уровня секторов (*ncurses*)» была проведена работа по анализу, проектированию и реализации средства для глубокого доступа к «сырым» данным на накопителях. Исследованная архитектура системы *Linux*, позволяющая обращаться к блочным устройствам через специальные файлы */dev/*, даёт существенные преимущества для низкоуровневых утилит, упрощая позиционированные операции и делая возможным использование виртуальных образов.

Особое внимание уделялось выбору программной платформы. Изучение *Ubuntu*, *GNU Toolchain* и *ncurses* обосновало оптимальное сочетание технологий: *C/C++* предоставляет высокий контроль над системными вызовами (*pread*, *pwrite*) и памятью, а *ncurses* позволяет создавать отзывчивый консольный интерфейс без графической среды. Это даёт высокоточную работу с *I/O* и удобную визуализацию секторов даже при ограниченных ресурсах.

Теоретическое обоснование разработки низкоуровневого редактора подтвердило актуальность подобных программ в современных условиях. Традиционные высокоуровневые инструменты не всегда способны корректно работать с повреждёнными разделами, скрытыми зонами или нестандартными структурными метаданными. В этом случае именно низкоуровневый редактор, оперирующий сырыми байтами, даёт возможность детального анализа и ручного исправления проблемных участков. Такой подход востребован в сфере компьютерной криминалистики, где важно прямое исследование носителей, а также при глубоком восстановлении данных, когда полуавтоматические утилиты уже не справляются. Помимо этого, рассмотренные в ходе работы инструменты могут быть полезны и при обучении азам операционных систем – студенты получают реальную практику взаимодействия с физическим устройством, видят, как устроены сектора, и учатся безопасно прchоводить эксперименты в низкоуровневом пространстве.

Проектирование функциональных возможностей редактора и выбор архитектуры стали ключевыми этапами разработки. В результате была создана программа, предлагающая набор базовых, но крайне важных функций: просмотр содержимого секторов с параллельным отображением *HEX* и *ASCII*-представления, редактирование отдельных байтов по двузначному шестнадцатеричному коду, копирование и вставка всего сектора, переход по номеру сектора, а также механизмы сохранения изменений (автоматического или по запросу) и журналирования действий (*undo/redo*). Последний пункт является одним из наиболее значимых, поскольку позволяет избежать необратимых ошибок при работе со сложными и критическими для системы участками диска. Помимо этого, в случае сбоя операционной системы или аппаратной поломки пользователь всегда может выбрать безопасный путь отмены последнего изменения.

В ходе пятой главы осуществлён обзор архитектуры разрабатываемого редактора, включая детализацию модулей (*I/O Core, Logic & History, UI* на базе *ncurses*) и описание функциональных и блок схем алгоритма работы приложения. Показано, как команды, поступающие от пользователя, обрабатываются в редакторе: часть из них требует операции чтения или записи на устройство, другая затрагивает логику копирования, вставки и управления историей, а третья служит для работы с интерфейсом (справка, подсветка, смена сектора). Отдельно рассмотрена обработка неопределённых результатов, когда по каким-то причинам – неправильному смещению, сбою чтения или неверному *HEX*-вводу – редактор не может корректно продолжить операцию. В таких сценариях программа оповещает о проблеме и предлагает пользователю вернуться к предыдущему состоянию или сменить действие.

Значимым итогом является то, что созданный редактор успешно выполняет свои функции в условиях, когда другие утилиты не позволяют получить нужную детализацию работы с носителем. Это означает, что достигнута основная цель проекта – предоставить безопасный и прозрачный механизм прямого редактирования «сырых» данных. Практическая ценность решения заключается в возможности использовать его как для глубокой диагностики системы, так и в образовательном процессе, где важно показать, как именно операционная система управляет физическими блоками памяти. К тому же наличие журнала изменений делает программу подходящим инструментом для экспериментальной среды, где допускаются рискованные операции по модификации загрузочных записей и структур разделов.

В целом можно констатировать, что проект достиг поставленных целей. Реализованная программа сочетает в себе удобство текстового интерфейса, высокую производительность языка *C/C++* и гибкость базовой архитектуры Linux, дающей прямой доступ к устройствам. Полученный опыт может быть расширен за счёт внедрения новых возможностей вроде автоматического распознавания файловых систем, контроля за критическими секторами (запросов на подтверждение при редактировании системных областей) и подключения модулей аналитики или криминалистики. Это позволит ещё полнее раскрыть потенциал низкоуровневого редактора и позиционировать его как универсальный инструмент для администрирования, обучения и профессионального анализа данных.

Таким образом, выполненная работа подтвердила целесообразность создания низкоуровневого редактора и показала, что при соблюдении принципов модульности, аккуратном использовании системных вызовов и интеграции ncurses возможно получить удобное и надёжное приложение, способное глубоко взаимодействовать с блочными устройствами на уровне секторов. Результаты проекта могут служить базой для дальнейших исследований и разработок, направленных на совершенствование инструментов точечного доступа к структурам хранения данных, а также на повышение качества обучения принципам организации и взаимодействия вычислительных систем.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

[1] Ubuntu. Официальная документация [Электронный ресурс]. – Режим доступа: https://ubuntu.com/tutorials. – Дата доступа: 20.02.2025.

[2] GNU Ncurses. Ncurses Library [Электронный ресурс]. – Режим доступа: https://invisible-island.net/ncurses/announce.html. – Дата доступа: 20.02.2025.

[3] Linux Kernel Archives. История и развитие ядра Linux [Электронный ресурс]. – Режим доступа: https://www.kernel.org/category/about.html. – Дата доступа: 21.02.2025.

[4] The Linux Documentation Project. Руководства по Linux [Электронный ресурс]. – Режим доступа: https://www.tldp.org/. – Дата доступа: 21.02.2025.

[5] IBM Developer. Introduction to Linux block devices [Электронный ресурс]. – Режим доступа: https://developer.ibm.com/tutorials/. – Дата доступа: 21.02.2025.

[6] Freed. Linux from Scratch [Электронный ресурс]. – Режим доступа: https://www.linuxfromscratch.org/. – Дата доступа: 23.02.2025.

[7] man7.org. Linux man pages [Электронный ресурс]. – Режим доступа: https://man7.org/linux/man-pages/. – Дата доступа: 23.02.2025.

[8] Stack Exchange. Unix & Linux Q&A [Электронный ресурс]. – Режим доступа: https://unix.stackexchange.com/. – Дата доступа: 24.02.2025.

[9] Stallman R. The GNU Project [Электронный ресурс]. – Режим доступа: https://www.gnu.org/gnu/gnu-history.en.html. – Дата доступа: 24.02.2025.

[10] Open Source Initiative. About the Open Source Initiative [Электронный ресурс]. – Режим доступа: https://opensource.org/about. – Дата доступа: 24.02.2025.

[11] Visual Studio Code. Official Documentation [Электронный ресурс]. – Режим доступа: https://code.visualstudio.com/docs. – Дата доступа: 24.02.2025.

[12] Oracle. Linux System Calls Overview [Электронный ресурс]. – Режим доступа: https://docs.oracle.com/en/operating-systems/linux/. – Дата доступа: 23.02.2025.

[13] CMake. Official Documentation [Электронный ресурс]. – Режим доступа: https://cmake.org/documentation/. – Дата доступа: 24.02.2025.

[14] FHS. Filesystem Hierarchy Standard [Электронный ресурс]. – Режим доступа: https://refspecs.linuxfoundation.org/fhs.shtml. – Дата доступа: 24.02.2025.

[15] Freedman M. Low-level Disk Editing: Practical Techniques and Analysis [Электронный ресурс]. – Режим доступа: https://example.com/low-level-disk-editing – Дата доступа: 10.03.2025.

[16] Brown A. Advanced Forensic Methods for Boot Sector Security [Электронный ресурс]. – Режим доступа: https://example.com/forensic-boot-sector – Дата доступа: 11.03.2025.

[17] Davis R. Efficient Usage of System Calls in C/C++ [Электронный ресурс]. – Режим доступа: https://example.com/system-calls-usage – Дата доступа: 12.03.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Справка о проверке на заимствования

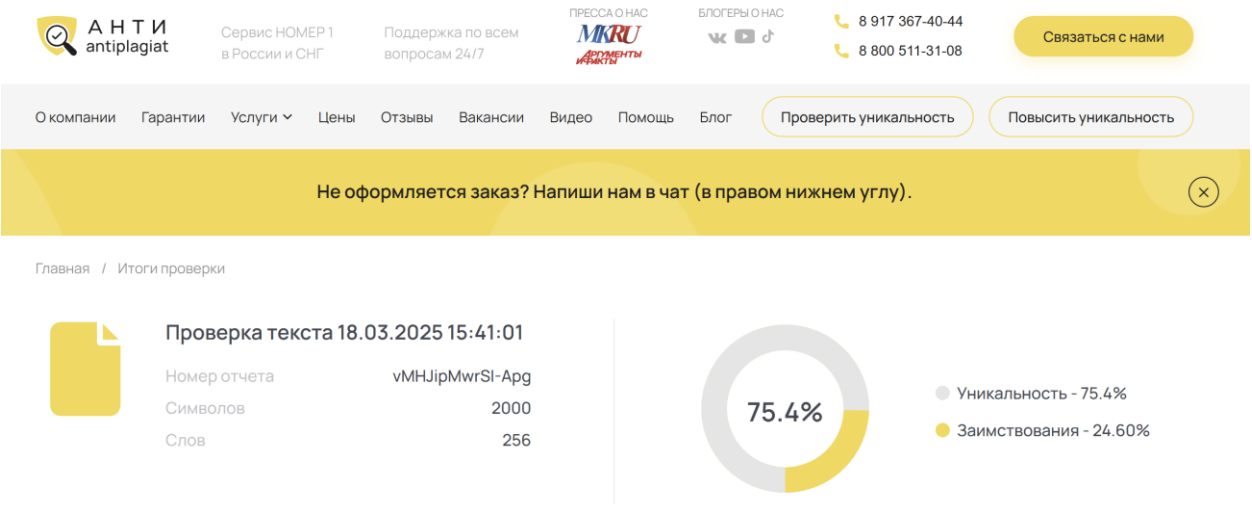

Рисунок А.1 – Скриншот проверки на антиплагиат

ПРИЛОЖЕНИЕ Б
(обязательное)
Листинг программного кода

```
#include "editor.h"
#include <ctype.h>
#include <fcntl.h>
#include <linux/fs.h>
#include <ncurses.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/ioctl.h>
#include <unistd.h>
#include <vector>


#define BYTES_PER_ROW 16

struct Change {
  char op;
  int sector;
  int offset;
  unsigned char oldValue;
  unsigned char newValue;
  std::vector<unsigned char> oldSector;
  std::vector<unsigned char> newSector;
};

static int fd = -1;
static unsigned long long deviceSize;
static int sectorSizeGlobal = 512;
static int numSectors = 0;
static bool autosave = false;

static int currentSector = 0;
static int cursorX = 0, cursorY = 0;

static std::vector<Change> undoStack;
static std::vector<Change> redoStack;

static std::vector<unsigned char> sectorBuffer;
static bool sectorBufferLoaded = false;

static std::vector<unsigned char> copyBuffer;
static bool hasCopy = false;

static inline int getIndexInBuffer(int row, int col) {
  return row * BYTES_PER_ROW + col;
}

static bool loadCurrentSector() {
  if (currentSector < 0 || currentSector >= numSectors)
    return false;
  sectorBufferLoaded = false;
  sectorBuffer.resize(sectorSizeGlobal, 0);

  off_t offset = (off_t)currentSector * sectorSizeGlobal;
  ssize_t r = pread(fd, sectorBuffer.data(), sectorSizeGlobal, offset);
  if (r != (ssize_t)sectorSizeGlobal) {
    return false;
  }
```

```
        sectorBufferLoaded = true;
        return true;
      }

      static bool saveCurrentSector() {
        if (!sectorBufferLoaded)
          return false;
        off_t offset = (off_t)currentSector * sectorSizeGlobal;
        ssize_t w = pwrite(fd, sectorBuffer.data(), sectorSizeGlobal, offset);
        return (w == (ssize_t)sectorSizeGlobal);
      }

      static void drawUI(const char *filename) {
        clear();
        mvprintw(0, 0, "File: %s | Sector: %d/%d | Sector size: %d bytes",
filename,
                 currentSector, numSectors - 1, sectorSizeGlobal);

        if (!sectorBufferLoaded) {
          mvprintw(2, 0, "Error: sector not loaded!");
          refresh();
          return;
        }

        int rows = sectorSizeGlobal / BYTES_PER_ROW;
        for (int row = 0; row < rows; row++) {
          int offset = row * BYTES_PER_ROW;
          mvprintw(row + 1, 0, "%04X: ", offset);
          for (int col = 0; col < BYTES_PER_ROW; col++) {
            int index = offset + col;
            unsigned char byteVal = sectorBuffer[index];
            if (row == cursorY && col == cursorX) {
              attron(A_REVERSE);
              printw("%02X ", byteVal);
              attroff(A_REVERSE);
            } else {
              printw("%02X ", byteVal);
            }
          }
          printw("  ");
          for (int col = 0; col < BYTES_PER_ROW; col++) {
            int index = offset + col;
            unsigned char byteVal = sectorBuffer[index];
            if (isprint(byteVal)) {
              printw("%c", byteVal);
            } else {
              printw(".");
            }
          }
        }
        int bottom = (sectorSizeGlobal / BYTES_PER_ROW) + 2;
        mvprintw(bottom, 0,
                 "Arrows: move cursor | n/p: next/prev sector | j: jump
sector");
        mvprintw(bottom + 1, 0,
                 "e/Enter: edit byte | s: save | c: copy | v: paste | u:undo |
"
                 "r:redo | h:help | q:quit");
        refresh();
      }

      static void moveCursor(int dy, int dx) {
        int rows = sectorSizeGlobal / BYTES_PER_ROW;
        cursorY += dy;
```

```
        cursorX += dx;
        if (cursorY < 0)
          cursorY = 0;
        if (cursorY >= rows)
          cursorY = rows - 1;
        if (cursorX < 0)
          cursorX = 0;
        if (cursorX >= BYTES_PER_ROW)
          cursorX = BYTES_PER_ROW - 1;
      }

      static void showDetailedHelp() {
        clear();
        int row = 0;
        attron(A_BOLD);
        mvprintw(row++, 0, "Low-Level Sector Editor - Detailed Help");
        attroff(A_BOLD);
        row++;
        mvprintw(row++, 0,
                 "This editor allows viewing and editing of raw sectors from a
file "
                 "or block device.");
        mvprintw(row++, 0,
                 "You can navigate inside each sector using the arrow keys.
Each "
                 "sector is displayed");
        mvprintw(row++, 0, "in both hexadecimal and ASCII format, 16 bytes per
row.");
        row++;
        mvprintw(row++, 0, "Controls:");
        mvprintw(row++, 0,
                 "  Arrow keys  : Move the cursor within the current sector "
                 "(up/down/left/right)");
        mvprintw(row++, 0, "  n/p         : Go to the next/previous sector");
        mvprintw(row++, 0,
                 "  j            : Jump to a specific sector by entering its
number");
        mvprintw(row++, 0,
                 "  e or Enter  : Edit the byte under the cursor (enter a 2-
digit "
                 "HEX value)");
        mvprintw(row++, 0,
                 "  s          : Save the current sector to the file/device");
        mvprintw(row++, 0, "  c             : Copy the entire sector into a
buffer");
        mvprintw(row++, 0,
                 "  v            : Paste the copied sector into the current
sector "
                 "(overwrites data)");
        mvprintw(row++, 0,
                 "  u/r        : Undo/redo changes (including single-byte edits
or "
                 "entire paste operations)");
        mvprintw(row++, 0, "  h           : Show this detailed help screen");
        mvprintw(row++, 0, "  q           : Quit the editor");
        row++;
        mvprintw(row++, 0, "Press any key to return...");
        refresh();
        getch();
      }

      static void editByte() {
        if (!sectorBufferLoaded)
          return;
```

```
        int offset = cursorY * BYTES_PER_ROW + cursorX;
        echo();
        curs_set(1);

        mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 4, 0,
                 "Enter new HEX (2 digits) for offset %04X: ", offset);
        clrtoeol();
        char input[3] = {0};
        getnstr(input, 2);

        int newVal;
        if (sscanf(input, "%x", &newVal) == 1 && newVal >= 0 && newVal <= 255)
{
          unsigned char oldVal = sectorBuffer[offset];
          if (oldVal != (unsigned char)newVal) {
            Change ch;
            ch.op = 'E';
            ch.sector = currentSector;
            ch.offset = offset;
            ch.oldValue = oldVal;
            ch.newValue = (unsigned char)newVal;
            undoStack.push_back(ch);
            redoStack.clear();
          }
          sectorBuffer[offset] = (unsigned char)newVal;

          if (autosave) {
            if (!saveCurrentSector()) {
              mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 5, 0,
                       "Error writing sector!");
              getch();
            }
          }
        } else {
          mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 5, 0,
                   "Invalid input! Press any key...");
          getch();
        }
        noecho();
        curs_set(0);
      }

      static void saveSector(bool showMessage) {
        if (!sectorBufferLoaded)
          return;
        if (!saveCurrentSector()) {
          mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 6, 0,
                   "Error writing sector!");
          getch();
        } else {
          if (showMessage) {
            mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 6, 0,
                     "Sector saved. Press any key...");
            getch();
          }
        }
      }

      static void undoChange() {
        if (undoStack.empty())
          return;
        Change ch = undoStack.back();
        undoStack.pop_back();
```

```
  if (ch.sector != currentSector) {
    if (autosave)
      saveCurrentSector();
    currentSector = ch.sector;
    loadCurrentSector();
    cursorX = 0;
    cursorY = 0;
  }

  if (ch.op == 'E') {
    sectorBuffer[ch.offset] = ch.oldValue;
  } else if (ch.op == 'P') {
    memcpy(sectorBuffer.data(), ch.oldSector.data(), sectorSizeGlobal);
  }

  redoStack.push_back(ch);
  if (autosave)
    saveCurrentSector();
  mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 6, 0,
           "Undo done. Press any key...");
  getch();
}

static void redoChange() {
  if (redoStack.empty())
    return;
  Change ch = redoStack.back();
  redoStack.pop_back();

  if (ch.sector != currentSector) {
    if (autosave)
      saveCurrentSector();
    currentSector = ch.sector;
    loadCurrentSector();
    cursorX = 0;
    cursorY = 0;
  }

  if (ch.op == 'E') {
    sectorBuffer[ch.offset] = ch.newValue;
  } else if (ch.op == 'P') {
    memcpy(sectorBuffer.data(), ch.newSector.data(), sectorSizeGlobal);
  }
  undoStack.push_back(ch);
  if (autosave)
    saveCurrentSector();
  mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 6, 0,
           "Redo done. Press any key...");
  getch();
}

static void copySector() {
  if (!sectorBufferLoaded)
    return;
  copyBuffer.resize(sectorSizeGlobal);
  memcpy(copyBuffer.data(), sectorBuffer.data(), sectorSizeGlobal);
  hasCopy = true;
  mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 5, 0,
           "Sector copied. Press any key...");
  getch();
}

static void pasteSector() {
  if (!sectorBufferLoaded)
```

```
          return;
        if (!hasCopy) {
          mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 5, 0,
                   "No sector copied. Press any key...");
          getch();
          return;
        }
        mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 5, 0, "Confirm paste?
(y/n): ");
        int c = getch();
        if (c != 'y' && c != 'Y') {
          mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 6, 0,
                   "Paste cancelled. Press any key...");
          getch();
          return;
        }
        Change ch;
        ch.op = 'P';
        ch.sector = currentSector;
        ch.oldSector.resize(sectorSizeGlobal);
        ch.newSector.resize(sectorSizeGlobal);

        memcpy(ch.oldSector.data(), sectorBuffer.data(), sectorSizeGlobal);
        memcpy(sectorBuffer.data(), copyBuffer.data(), sectorSizeGlobal);
        memcpy(ch.newSector.data(), sectorBuffer.data(), sectorSizeGlobal);

        undoStack.push_back(ch);
        redoStack.clear();

        if (autosave)
          saveCurrentSector();
        mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 6, 0,
                 "Sector pasted. Press any key...");
        getch();
      }

      static void nextSector() {
        if (currentSector < numSectors - 1) {
          if (autosave)
            saveCurrentSector();
          currentSector++;
          loadCurrentSector();
          cursorX = 0;
          cursorY = 0;
        }
      }

      static void prevSector() {
        if (currentSector > 0) {
          if (autosave)
            saveCurrentSector();
          currentSector--;
          loadCurrentSector();
          cursorX = 0;
          cursorY = 0;
        }
      }

      static void jumpSector() {
        echo();
        curs_set(1);
        mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 5, 0,
                 "Enter sector number (0..%d): ", numSectors - 1);
        char buf[32];
```

```
  getnstr(buf, 31);
  noecho();
  curs_set(0);

  int s = atoi(buf);
  if (s >= 0 && s < numSectors) {
    if (autosave)
      saveCurrentSector();
    currentSector = s;
    if (!loadCurrentSector()) {
      mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 6, 0,
               "Error reading sector! Press any key...");
      getch();
    }
    cursorX = 0;
    cursorY = 0;
  } else {
    mvprintw((sectorSizeGlobal / BYTES_PER_ROW) + 6, 0,
             "Invalid sector! Press any key...");
    getch();
  }
}

void runEditor(const char *filename, int secSize, bool autosaveFlag) {
  sectorSizeGlobal = secSize;
  autosave = autosaveFlag;

  fd = open(filename, O_RDWR);
  if (fd < 0) {
    perror("Error opening file/device");
    exit(EXIT_FAILURE);
  }

  unsigned long long size64 = 0;
  if (ioctl(fd, BLKGETSIZE64, &size64) == 0 && size64 > 0) {
    deviceSize = size64;
  } else {
    off_t sz = lseek(fd, 0, SEEK_END);
    if (sz < 0) {
      perror("lseek");
      close(fd);
      exit(EXIT_FAILURE);
    }
    deviceSize = (unsigned long long)sz;
  }

  numSectors = deviceSize / sectorSizeGlobal;
  if (deviceSize % sectorSizeGlobal != 0) {
    numSectors++;
  }

  sectorBuffer.resize(sectorSizeGlobal, 0);
  sectorBufferLoaded = false;

  copyBuffer.resize(sectorSizeGlobal, 0);
  hasCopy = false;

  currentSector = 0;
  if (!loadCurrentSector()) {
    printf("Error reading first sector.\n");
    close(fd);
    return;
  }
```

```
  initscr();
  if (has_colors()) {
    start_color();
    init_pair(1, COLOR_WHITE, COLOR_BLACK);
    init_pair(2, COLOR_RED, COLOR_BLACK);
  }
  noecho();
  cbreak();
  keypad(stdscr, TRUE);
  curs_set(0);

  int ch;
  while (true) {
    drawUI(filename);
    ch = getch();
    if (ch == 'q') {
      break;
    } else if (ch == KEY_UP) {
      moveCursor(-1, 0);
    } else if (ch == KEY_DOWN) {
      moveCursor(1, 0);
    } else if (ch == KEY_LEFT) {
      moveCursor(0, -1);
    } else if (ch == KEY_RIGHT) {
      moveCursor(0, 1);
    } else if (ch == 'n') {
      nextSector();
    } else if (ch == 'p') {
      prevSector();
    } else if (ch == 'j') {
      jumpSector();
    } else if (ch == 'e' || ch == '\n') {
      editByte();
    } else if (ch == 's') {
      saveSector(true);
    } else if (ch == 'c' || ch == 'C') {
      copySector();
    } else if (ch == 'v' || ch == 'V') {
      pasteSector();
    } else if (ch == 'u' || ch == 'U') {
      undoChange();
    } else if (ch == 'r' || ch == 'R') {
      redoChange();
    } else if (ch == 'h') {
      showDetailedHelp();
    }
  }

  endwin();
  close(fd);
}
#ifndef EDITOR_H
#define EDITOR_H

void runEditor(const char *filename, int sectorSize, bool autosave);

#endif
#include "editor.h"
#include <cstdio>
#include <cstdlib>
#include <cstring>

static void printUsage(const char *progName) {
  printf("Usage: %s [--help] [--autosave] <image file> [sector size]\n",
```

```
            progName);
    }

    int main(int argc, char *argv[]) {
      bool autosave = false;
      const char *filename = NULL;
      int sectorSize = 512;

      if (argc < 2) {
        printUsage(argv[0]);
        return EXIT_FAILURE;
      }

      int argIndex = 1;
      while (argIndex < argc && strncmp(argv[argIndex], "--", 2) == 0) {
        if (strcmp(argv[argIndex], "--help") == 0) {
          printUsage(argv[0]);
          printf("\nLow-Level Sector Editor\n");
          printf("Options:\n    --autosave      Automatically  save  after
editing\n");
          return EXIT_SUCCESS;
        } else if (strcmp(argv[argIndex], "--autosave") == 0) {
          autosave = true;
        } else {
          printUsage(argv[0]);
          return EXIT_FAILURE;
        }
        argIndex++;
      }

      if (argIndex < argc) {
        filename = argv[argIndex++];
      } else {
        printUsage(argv[0]);
        return EXIT_FAILURE;
      }

      if (argIndex < argc) {
        sectorSize = atoi(argv[argIndex]);
        if (sectorSize <= 0)
          sectorSize = 512;
      }

      runEditor(filename, sectorSize, autosave);
      return EXIT_SUCCESS;
    }
    CXX      = g++
    CXXFLAGS = -Wall -Wextra -O2 -std=c++17
    LDFLAGS  = -lncursesw
    TARGET   = editor
    SRCS     = main.cpp editor.cpp
    OBJS     = $(SRCS:.cpp=.o)

    .PHONY: all clean

    all: $(TARGET)

    $(TARGET): $(OBJS)
          $(CXX) $(CXXFLAGS) -o $@ $(OBJS) $(LDFLAGS)

    %.o: %.cpp
          $(CXX) $(CXXFLAGS) -c $< -o $@
```

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

ПРИЛОЖЕНИЕ Г
(обязательное)
Блок схема алгоритма, реализующего программное средства

ПРИЛОЖЕНИЕ Д
(обязательное)
Графический интерфейс пользователя

ПРИЛОЖЕНИЕ Е
(обязательное)
Ведомость курсового проекта