Министерство образования Республики Беларусь

Учреждение образования
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИНФОРМАТИКИ И РАДИОЭЛЕКТРОНИКИ

Факультет компьютерных систем и сетей
Кафедра информатики
Дисциплина: Операционные среды и системное программирование

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

ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
к курсовому проекту
на тему

МНОГОПОТОЧНАЯ ПРОГРАММА ДЛЯ ФОНОВОГО КОНТРОЛЯ ИЗМЕНЕНИЙ И ЦЕЛОСТНОСТИ ГРУППЫ ФАЙЛОВ

БГУИР КП 1-40 04 01 009 ПЗ

Студент Демещенко М. В.

Руководитель Гриценко Н. Ю.

Нормоконтролер Гриценко Н. Ю.

Минск 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современные операционные среды и системное программирование предоставляют широкие возможности для разработки многопоточных приложений, которые позволяют эффективно обрабатывать большие объемы данных и выполнять параллельные задачи. Одним из ключевых направлений в данной области является контроль изменений и целостности файлов, что особенно актуально в условиях постоянного увеличения угроз информационной безопасности.

Существующие решения в области мониторинга файловых систем включают антивирусные программы, системы обнаружения вторжений и специализированные утилиты для контроля изменений файлов. Однако большинство из них либо обладают избыточной функциональностью, что делает их ресурсоемкими, либо ориентированы на работу в определенных операционных системах, что ограничивает их применение в многоплатформенной среде. В связи с этим разработка легковесного многопоточного приложения для фонового мониторинга изменений группы файлов является актуальной задачей.

Целью курсовой работы является создание многопоточной программы, обеспечивающей автоматизированный контроль изменений и целостности файлов в фоновом режиме. Данное программное решение должно эффективно обрабатывать информацию, минимизируя нагрузку на систему, а также обеспечивать удобные средства уведомления пользователя о зафиксированных изменениях.

Разработка такой программы требует изучения принципов многопоточного программирования, особенностей работы файловых систем в различных операционных средах, а также методов контроля целостности данных. В рамках данного проекта предполагается реализация механизма отслеживания изменений файлов с возможностью настройки параметров мониторинга, что обеспечит гибкость и адаптивность разработанного программного обеспечения.

1 АРХИТЕКТУРА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

1.1 Структура и архитектура вычислительной системы

Вычислительная система, на базе которой будет работать разрабатываемое программное обеспечение, включает в себя аппаратные и программные компоненты, обеспечивающие многозадачность и возможность эффективного управления ресурсами. Основное внимание в этом разделе уделяется характеристикам операционной системы, механизмам взаимодействия программного обеспечения с аппаратной платформой и особенностям работы с файловой системой.

Типичная конфигурация системы предполагает наличие современного процессора архитектуры *x*86_64 или *ARM*, поддерживающего многопоточность. Количество ядер и потоков играет важную роль, так как программа будет использовать параллельное выполнение задач для минимизации задержек при обработке изменений файлов. Оперативная память определяет объем данных, которые могут одновременно обрабатываться без обращения к дисковым устройствам, что особенно важно при работе с большими группами файлов. Хранение данных осуществляется на жестком диске или *SSD*-накопителе, причем твердотельные накопители обеспечивают меньшие задержки при доступе к файлам, что положительно сказывается на быстродействии программы.

В качестве операционной системы используется *Linux* [1], поскольку данная система предоставляет широкий набор инструментов для системного программирования и мониторинга изменений файлов. Ключевыми возможностями *Linux*, которые используются в рамках данного проекта, являются встроенные механизмы *inotify* и *fanotify* [2], позволяющие эффективно отслеживать изменения в файловой системе. Также важную роль играет поддержка *POSIX Threads* (*pthread*) [3], обеспечивающая удобное управление потоками в многозадачной среде.

Программа будет работать в фоновом режиме, выполняя непрерывное слежение за состоянием файлов и проверку их целостности. Архитектура программного обеспечения предусматривает разделение на несколько взаимосвязанных компонентов. Основной поток выполняет инициализацию программы, настройку параметров мониторинга и обработку команд от пользователя. Рабочие потоки занимаются отслеживанием изменений файлов, получая события из *inotify* и записывая их в лог. Фоновый поток отвечает за проверку контрольных сумм файлов, выявляя возможные нарушения целостности данных.

Подобная организация позволяет программе эффективно использовать системные ресурсы и выполнять задачи мониторинга в режиме реального времени. Благодаря многопоточному исполнению нагрузка распределяется между процессорными ядрами, что снижает вероятность задержек при обработке событий файловой системы. Применение встроенных механизмов

Linux обеспечивает надежность работы и минимизирует дополнительные издержки, связанные с мониторингом состояния файлов.

1.2 История, версии и достоинства

Операционная система *Linux*, выбранная в качестве платформы для разработки программного обеспечения, имеет богатую историю и продолжает активно развиваться. Она является одной из самых популярных операционных систем, особенно в области серверных решений, облачных технологий и системного программирования.

История *Linux* начинается в 1991 году, когда финский программист Линус Торвальдс, вдохновленный архитектурой *Unix* [4], разработал первое ядро операционной системы. В отличие от проприетарных решений, таких как *Windows* и *macOS, Linux* с самого начала распространялась с открытым исходным кодом, что позволило тысячам разработчиков по всему миру участвовать в ее развитии. В результате *Linux* быстро эволюционировала и стала основой для множества дистрибутивов, таких как *Debian, Ubuntu, Arch Linux, Fedora, CentOS* и другие. Каждый из них имеет свои особенности, однако все они основаны на общем ядре, что обеспечивает совместимость между различными системами.

Развитие *Linux* шло параллельно с ростом вычислительных мощностей и усложнением программных архитектур. Одной из ключевых особенностей системы стала ее многозадачность и многопоточность, реализованные через ядро с поддержкой вытесняющей многозадачности. В отличие от *Windows*, где потоки управляются через объектную модель *Windows API* [5], в *Linux* потоки представляют собой легковесные процессы, которые могут работать в одном адресном пространстве. Это позволяет минимизировать затраты на переключение контекста и эффективно использовать многопроцессорные системы.

Существуют различные версии ядра *Linux*, которые регулярно обновляются и включают новые функции, улучшения производительности и исправления безопасности. Наиболее известными ветками развития ядра являются *LTS* (*Long-Term Support*) – версии с длительной поддержкой, ориентированные на серверные системы и стабильные окружения, а также *mainline* – основная ветка, в которую вносятся последние изменения. Это позволяет пользователям выбирать между стабильностью и новейшими возможностями.

Одним из ключевых механизмов *Linux*, который используется в разрабатываемом программном обеспечении, является система мониторинга файловой системы. *Linux* предоставляет несколько инструментов для этих целей, в том числе *inotify* и *fanotify*. *Inotify* позволяет отслеживать события файловой системы, такие как создание, изменение или удаление файлов, при этом его работа оптимизирована за счет использования системных вызовов на уровне ядра. *Fanotify*, в свою очередь, предлагает расширенные возможности,

включая проверку доступа к файлам перед их открытием, что делает его полезным для антивирусных программ и систем обнаружения вторжений.

Если сравнивать *Linux* с другими операционными системами, то можно выделить несколько ключевых отличий. В *Windows* аналогичные задачи мониторинга файлов решаются с помощью *API FileSystemWatcher* [6] или инструментов журналирования файловой системы *NTFS*, однако они обладают большей ресурсоемкостью и зависят от особенностей работы файловых систем *Windows*. В *macOS* для слежения за изменениями используется механизм *FSEvents* [7], который предоставляет события на более высоком уровне, но не дает такой гибкости, как *inotify*. *Unix*-подобные системы, такие как *FreeBSD*, также имеют свои методы отслеживания изменений, однако *Linux* остается наиболее гибкой и универсальной платформой для задач системного программирования.

К числу достоинств *Linux* можно отнести стабильность, безопасность, масштабируемость и гибкость. Операционная система отличается высокой устойчивостью к сбоям благодаря модульной архитектуре и эффективному управлению процессами. Встроенные механизмы управления правами пользователей и модели безопасности, такие как *SELinux* и *AppArmor*, обеспечивают надежную защиту данных. Широкая поддержка многопоточности и системных вызовов позволяет эффективно разрабатывать программное обеспечение, требующее параллельного выполнения задач.

Таким образом, выбор *Linux* в качестве вычислительной системы для данного проекта обусловлен не только удобством в работе с многопоточностью и файловыми системами, но и широкими возможностями контроля целостности данных, надежностью и гибкостью операционной среды.

1.3 Обоснование выбора вычислительной системы

Выбор операционной системы и вычислительной платформы играет ключевую роль в разработке многопоточного программного обеспечения, предназначенного для фонового контроля изменений и целостности группы файлов. В данном проекте в качестве вычислительной среды используется операционная система *Linux*, что обусловлено рядом технических и архитектурных преимуществ, делающих ее наиболее подходящей для реализации поставленных задач.

Помимо перечисленных выше системных механизмов для работы с файловой системой и реализации многопоточности, операционная система имеет еще ряд преимуществ:

1 Стабильность и надежность работы также являются ключевыми аргументами в пользу *Linux*. Операционная система обладает высокой отказоустойчивостью, что делает ее оптимальной средой для долгосрочной работы фоновых процессов. В отличие от *Windows*, где системные обновления могут потребовать перезагрузки и привести к временному прекращению работы сервиса, в *Linux* возможны обновления ядра и библиотек без остановки

системы, что особенно ценно для непрерывно работающих программных решений.

2 Широкий выбор файловых систем, оптимизированных для различных задач. Для проекта особенно подходят *ext*4 и *XFS*, так как они обеспечивают высокую скорость работы, стабильность и поддержку механизмов контроля целостности данных. Для более продвинутых решений можно использовать *Btrfs* [8], которая включает встроенные средства проверки целостности файлов, но в данном случае выбор остается за более проверенными и стабильными файловыми системами.

3 Гибкость настройки и открытость операционной системы. В отличие от *Windows*, где доступ к низкоуровневым механизмам часто ограничен, в *Linux* разработчик получает полный контроль над процессами, правами доступа и взаимодействием с файловой системой. Это позволяет адаптировать среду выполнения программы под конкретные задачи, минимизировать потребление ресурсов и повысить производительность.

Таким образом, выбор *Linux* в качестве операционной системы рассматриваемой вычислительной системы обусловлен сочетанием нескольких критически важных факторов: наличием встроенных механизмов мониторинга файлов, удобной моделью многопоточного программирования, высокой стабильностью, гибкостью в настройке и широким выбором файловых систем. Все это делает *Linux* оптимальной платформой для реализации многопоточного фонового приложения, предназначенного для контроля изменений и целостности группы файлов.

1.4 Анализ выбранной вычислительной системы

В качестве вычислительной среды для разработки многопоточного фонового приложения контроля изменений и целостности файлов была выбрана операционная система *Linux*. Этот выбор обусловлен рядом факторов, таких как развитые механизмы работы с файловой системой, эффективная поддержка многопоточности, высокая стабильность, гибкость настройки и широкие возможности интеграции с системным программированием. Анализ операционной системы включает рассмотрение ее архитектурных особенностей, механизмов работы с процессами и потоками, доступных инструментов для мониторинга файлов и ключевых преимуществ перед альтернативными платформами.

Linux относится к классу *Unix*-подобных операционных систем и обладает монолитным ядром, в котором реализована большая часть функциональности, включая управление процессами, памятью, файловыми системами и взаимодействие между компонентами системы. В отличие от микроядерных архитектур, таких как *QNX* или *Minix*, в *Linux* взаимодействие с аппаратным обеспечением и основными подсистемами выполняется в пространстве ядра, что обеспечивает высокую производительность и минимальные накладные расходы. Ядро *Linux* поддерживает вытесняющую многозадачность, благодаря чему процессы и потоки выполняются в

зависимости от приоритетов и доступности ресурсов. Используемый планировщик задач *CFS* (*Completely Fair Scheduler*) [9] эффективно распределяет процессорное время между потоками, что особенно важно при разработке многопоточных приложений, работающих в фоне.

Система управления памятью в *Linux* основана на механизмах виртуальной памяти и отображения файлов в память (*mmap*) [10], что позволяет эффективно работать с большими объемами данных и минимизировать задержки при доступе к файлам. Поддержка *copy-on-write* (*COW*) [11] снижает нагрузку на ресурсы при порождении процессов, что особенно полезно при использовании многопоточности и разделяемых ресурсов. *Linux* предоставляет два основных механизма параллельного выполнения задач: процессы и потоки. Процессы (*fork*) создаются с помощью системного вызова *fork*() [12], который порождает новый процесс с отдельным адресным пространством. Этот механизм активно используется в серверных приложениях, но имеет накладные расходы на создание и переключение контекста.

Более эффективным решением для многопоточных программ является использование *POSIX Threads* (*pthread*) – стандартизированного *API*, позволяющего создавать и управлять потоками в пределах одного процесса. В отличие от *Windows*, где потоки создаются через объектную модель *Windows API*, в *Linux* потоки представляют собой легковесные процессы (*LWP – Lightweight Process*) и выполняются в одном адресном пространстве, что ускоряет их переключение и упрощает совместное использование памяти. Механизмы синхронизации, такие как мьютексы (*mutexes*), семафоры и условные переменные (*condition variables*), позволяют управлять доступом к разделяемым ресурсам и предотвращать гонки потоков. В сочетании с механизмами управления планированием потоков (например, *sched_setaffinity*(), *nice*(), *pthread_setschedparam*()) это обеспечивает гибкость в организации многопоточной обработки данных.

Операционная система *Linux* предоставляет несколько встроенных инструментов для отслеживания изменений файлов и проверки их целостности. Среди этих инструментов можно выделить *inotify* и *fanotify*, а также:

1 Методы вычисления хеш-сумм, такие как *SHA*-256, *MD*5 и *CRC*32 [13], используются для выявления изменений в содержимом файлов, и встроенные команды *sha*256*sum*, *md*5*sum* и *crc*32 позволяют проверять целостность данных без необходимости чтения файла в память целиком.

2 Аудит файловой системы (*auditd*) является подсистемой *Linux*, которая фиксирует события, связанные с изменением файлов, доступом к ним и операциями с правами пользователей.

При сравнении с другими операционными системами *Linux* обладает рядом преимуществ. Одним из них является гибкость и открытость: в отличие от *Windows*, *Linux* предоставляет полный доступ к низкоуровневым системным вызовам, что позволяет разрабатывать эффективные многопоточные и системные приложения. Также *Linux* отличается высокой

производительностью, поскольку отсутствие лишних фоновых сервисов и оптимизированное ядро обеспечивают более высокую скорость выполнения фоновых задач по сравнению с *Windows*. Стабильность и надежность *Linux* также являются важными преимуществами: возможность постоянной работы системы без перезагрузки, а также механизмы самовосстановления процессов делают *Linux* предпочтительной средой для долгосрочной работы фоновых приложений. Масштабируемость *Linux* позволяет использовать систему как на встроенных системах с ограниченными ресурсами, так и на мощных серверных кластерах, что делает ее универсальным решением. Кроме того, поддержка журналируемых файловых систем, таких как *ext*4, *XFS* и *Btrfs*, позволяет эффективно управлять данными и снижать вероятность их повреждения.

В целом, анализ операционной системы *Linux* показывает, что она обладает всеми необходимыми характеристиками для реализации многопоточного приложения мониторинга изменений файлов. Архитектурные особенности ядра, развитые механизмы управления процессами и потоками, эффективные инструменты работы с файловой системой и высокая стабильность делают *Linux* оптимальным выбором для данной задачи.

1.5 Анализ аналогов разрабатываемой программы

Перед разработкой многопоточного фонового приложения для контроля изменений и целостности группы файлов необходимо рассмотреть существующие решения, выполняющие схожие задачи. Анализ аналогов позволяет определить сильные и слабые стороны уже реализованных механизмов, а также выбрать оптимальные подходы для проектирования собственной системы.

Существует несколько классов программного обеспечения, обеспечивающего мониторинг изменений файлов и контроль их целостности. Среди них можно выделить специализированные инструменты для слежения за изменениями в файловой системе, системы обнаружения вторжений (*IDS*) и резервного копирования, а также встроенные механизмы самих операционных систем.

Одним из наиболее известных инструментов является *incron* – утилита для *Linux*, основанная на механизме *inotify*. Она позволяет запускать пользовательские скрипты в ответ на изменения файлов, что делает ее удобной для автоматизации задач администрирования. Однако *incron* не предоставляет встроенных средств проверки целостности файлов и требует дополнительной настройки для работы в многопоточном режиме.

Другим аналогом является *auditd* – подсистема аудита, встроенная в ядро *Linux*. Она отслеживает доступ к файлам и изменения их атрибутов, предоставляя детализированные журналы событий. *Auditd* хорошо интегрируется с системами безопасности, такими как *SELinux*, но его основное предназначение – контроль за действиями пользователей, а не полноценный анализ изменений содержимого файлов.

Для контроля целостности файлов широко используются системы обнаружения вторжений, такие как *AIDE* (*Advanced Intrusion Detection Environment*) и *Tripwire*. Эти инструменты создают контрольные суммы файлов и периодически проверяют их на изменения. Они обеспечивают высокий уровень безопасности, но не работают в режиме реального времени и требуют регулярного запуска проверки, что может создавать дополнительную нагрузку на систему.

Еще одним подходом является использование файловых систем с встроенными механизмами контроля целостности, например, *Btrfs* и *ZFS*. Они позволяют автоматически отслеживать изменения данных на уровне блоков и предотвращать повреждение файлов. Однако такие файловые системы требуют особой настройки и могут быть избыточными для задач простого мониторинга изменений. Структура файловой системы *Btrfs* изображена на рисунке 1.1.

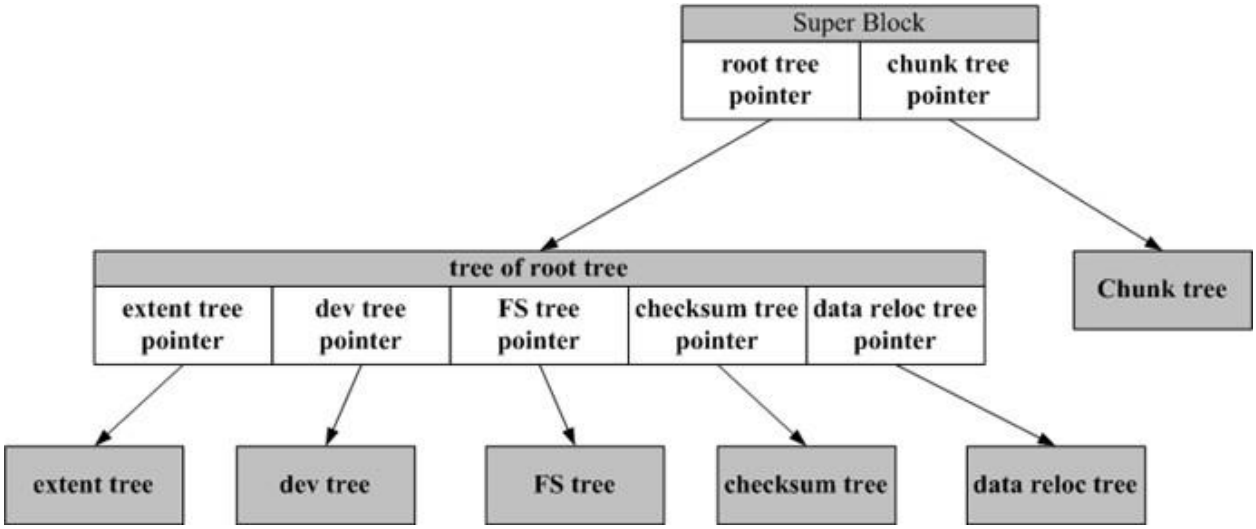


Рисунок 1.1 – Структура файловой системы Btrfs

Если сравнивать с аналогами, разрабатываемое программное обеспечение отличается реализацией мониторинга в реальном времени, поддержкой многопоточности и возможностью не только фиксировать изменения, но и проверять контрольные суммы файлов без значительных накладных расходов. По сравнению с *AIDE* и *Tripwire*, предлагаемый подход снижает задержки между изменением файла и обнаружением проблемы. В отличие от *auditd*, разрабатываемая система ориентирована не на контроль доступа, а на целостность файловых данных.

Таким образом, существующие решения выполняют схожие функции, но либо имеют узкую специализацию, либо не предназначены для работы в режиме реального времени. Разрабатываемая система объединяет лучшие стороны аналогов, предлагая многопоточный подход к мониторингу изменений файлов и их целостности с учетом высокой производительности и эффективности.

2 ПЛАТФОРМА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

2.1 Структура и архитектура платформы

Разрабатываемое программное обеспечение представляет собой многопоточное фоновое приложение, предназначенное для контроля изменений и целостности группы файлов в операционной системе *Linux*. С учетом специфики работы операционной системы, необходимости эффективного управления потоками и взаимодействия с системными вызовами было принято решение использовать язык программирования *C++* [14]. Данный язык обладает высокой производительностью, обеспечивает низкоуровневый доступ к системным ресурсам и поддерживает удобные механизмы многопоточного программирования.

Среда разработки представляет собой *Visual Studio Code* [15], так как она предоставляет удобные инструменты для работы с кодом, поддержку расширений и интеграцию с компилятором. В качестве компилятора используется *GCC* (*GNU Compiler Collection*), а точнее его компонент *g++* [16], который компилирует код на *C++*. Данный компилятор, по сути, является стандартом в среде *Linux* и предоставляет широкие возможности оптимизации, а также поддержку стандартов *C++* различной версии.

Важной частью архитектуры программного обеспечения является взаимодействие с файловой системой и механизмами мониторинга изменений. Для реализации данной функциональности используются системные вызовы *inotify* и библиотека *fanotify*.

Inotify – это встроенный в ядро *Linux* механизм, который позволяет отслеживать изменения в файловой системе в реальном времени. Он предоставляет *API*, с помощью которого можно подписываться на события, такие как создание, удаление, изменение или перемещение файлов и каталогов. Программа использует *inotify_add_watch*() [17], чтобы отслеживать целевые файлы и реагировать на их изменения. *Inotify* работает асинхронно и генерирует события, которые приложение может обрабатывать через файловый дескриптор, считывая их при помощи *read*().

Fanotify является более продвинутой версией *inotify*, предоставляя дополнительные возможности, такие как блокировка доступа к файлам до завершения их проверки. В отличие от *inotify*, который работает с конкретными файлами и директориями, *fanotify* может работать на уровне всей файловой системы, позволяя отслеживать события для любого файла. Использование *fanotify_init*() и *fanotify_mark*() позволяет программе не только получать уведомления о событиях, но и при необходимости предотвращать выполнение нежелательных операций над файлами.

Помимо механизмов мониторинга файлов, в приложении используются механизмы хеширования для проверки целостности данных. Для вычисления контрольных сумм применяется библиотека *OpenSSL* [18], предоставляющая высокопроизводительные реализации алгоритмов хеширования, таких как *SHA*-256. Это позволяет программе выявлять изменения не только на уровне

метаданных, но и непосредственно в содержимом файлов, сравнивая их контрольные суммы с эталонными значениями.

Для организации многопоточной работы используется стандартная библиотека *C++* (*std*::*thread*, *std*::*mutex*, *std*::*condition_variable*) [19], которая предоставляет удобные средства для управления потоками и синхронизации доступа к общим ресурсам. Это позволяет обеспечивать параллельное отслеживание изменений в нескольких директориях без излишних затрат на управление потоками вручную.

Таким образом, программная платформа включает в себя язык программирования *C++*, среду разработки *Visual Studio Code*, компилятор *GCC* и ряд системных инструментов, включая *inotify*, *fanotify* и *OpenSSL*. Архитектура приложения строится вокруг механизмов асинхронного мониторинга файлов и многопоточной обработки событий, что позволяет минимизировать задержки и обеспечивать высокую эффективность работы в режиме реального времени.

2.2 История, версии и достоинства

Выбранная программная платформа, основанная на языке программирования *C++*, компиляторе *GCC* и системных механизмах мониторинга файлов в *Linux*, прошла значительное развитие, что делает ее оптимальным выбором для создания многопоточного фонового приложения. Анализ ее истории, версионных изменений и ключевых достоинств позволяет более глубоко понять принципы ее работы, а также обосновать выбор именно данной технологической базы.

2.2.1 Язык программирования *C++* был создан Бьёрном Страуструпом в 1983 году как расширение языка *C*, получившее поддержку объектно-ориентированного программирования и более высокоуровневые механизмы управления памятью. На протяжении своей истории *C++* развивался в сторону повышения эффективности, безопасности и удобства использования, при этом оставаясь языком с возможностью низкоуровневого доступа к системным ресурсам. Современные стандарты, начиная с *C++*11, внесли значительные улучшения, включая поддержку многопоточности через *std*::*thread*, а также добавление умных указателей (*std*::*unique_ptr*, *std*::*shared_ptr*), что значительно упростило управление памятью и синхронизацию потоков.

Компилятор *GCC* (*GNU Compiler Collection*) был разработан в 1987 году как часть проекта *GNU* и стал стандартным инструментом для компиляции программного обеспечения в *Linux*. Его постоянное развитие привело к улучшению оптимизации кода, расширению поддержки современных стандартов *C++* и внедрению механизмов статического анализа. В актуальных версиях компилятора *GCC* (например, *GCC* 11+) появилась улучшенная поддержка многопоточности, более точное управление ресурсами процессора и расширенные возможности оптимизации.

Среда мониторинга файлов в *Linux* также эволюционировала. Первоначально для слежения за изменениями файлов применялись более

простые механизмы, такие как системные вызовы *stat*() и *poll*(), которые требовали периодического опроса файловой системы. В 2005 году был разработан *inotify*, который стал стандартным механизмом событийного мониторинга, позволяя программам эффективно отслеживать изменения файлов без постоянного опроса. В 2010 году появился *fanotify*, который предоставил расширенные возможности, включая возможность вмешательства в операции файловой системы, что сделало его более мощным инструментом для контроля целостности данных.

2.2.2 Используемая программная платформа обладает рядом значительных преимуществ, которые делают ее наиболее подходящей для создания многопоточного приложения контроля изменений файлов.

Во-первых, использование языка *C++* позволяет достичь высокой производительности и эффективности работы с системными ресурсами. В отличие от интерпретируемых языков, таких как *Python*, *C++* выполняется напрямую в машинном коде, что минимизирует накладные расходы на обработку файловых событий. Кроме того, наличие стандартной библиотеки многопоточности делает возможной эффективную реализацию параллельной обработки событий без использования сторонних зависимостей.

Во-вторых, компилятор *GCC* предоставляет продвинутые инструменты оптимизации, что позволяет создавать высокопроизводительные программы. Возможности компилятора включают генерацию оптимизированного машинного кода, автоматическое векторизирование циклов и использование расширенных инструкций процессора, что особенно важно для обработки больших объемов данных в реальном времени.

В-третьих, встроенные механизмы *inotify* и *fanotify* обеспечивают событийно-ориентированную модель мониторинга файлов, позволяя системе мгновенно реагировать на изменения в файловой системе. Это значительно эффективнее, чем альтернативные решения, основанные на периодическом опросе состояния файлов. *Fanotify* дополнительно предоставляет возможность вмешиваться в операции с файлами, что позволяет приостанавливать доступ к ним до завершения проверки их целостности.

Четвертым важным преимуществом является гибкость и расширяемость программной платформы. Используемые технологии позволяют легко добавлять новые методы анализа изменений файлов, интегрировать дополнительные алгоритмы хеширования для проверки целостности данных, а также адаптировать систему под различные сценарии использования.

Таким образом, программная платформа, основанная на языке *C++*, компиляторе *GCC* и механизмах *inotify*/*fanotify*, является наиболее эффективным решением для создания многопоточного фонового приложения контроля изменений файлов. Ее эволюция на протяжении нескольких десятилетий привела к высокой степени оптимизации и адаптации под современные требования, а встроенные механизмы мониторинга файлов обеспечивают минимальные задержки и высокую производительность работы приложения.

2.3 Обоснование выбора платформы

Разработка многопоточного фонового приложения для контроля изменений и целостности группы файлов требует использования программной платформы, обладающей высокой производительностью, стабильностью и возможностью эффективного взаимодействия с системными ресурсами. Выбор платформы напрямую влияет на производительность, масштабируемость и надежность конечного решения. В качестве основы программной платформы был выбран язык программирования *C++*, компилятор *GCC*, среда разработки *Visual Studio Code*, а также механизмы мониторинга файлов *inotify* и *fanotify*. Обоснование данного выбора базируется на анализе ключевых характеристик программного окружения, его преимуществ перед альтернативными решениями и соответствия требованиям проекта.

Основным критерием выбора платформы является обеспечение высокой скорости работы приложения при минимальных затратах на системные ресурсы. Язык *C++* является компилируемым, что позволяет генерировать машинный код, исполняемый непосредственно процессором, без необходимости интерпретации или промежуточного выполнения. Это дает значительное преимущество перед языками с динамической типизацией, такими как *Python* [20], где накладные расходы на обработку кода и управление памятью заметно выше.

Использование *GCC* в качестве компилятора позволяет применять оптимизации, такие как автоматическая векторизация, устранение мертвого кода и предсказание ветвлений, что увеличивает скорость выполнения кода. Кроме того, *GCC* поддерживает новейшие стандарты *C++*, включая *C++*20, который предоставляет расширенные возможности работы с потоками и синхронизации.

Для контроля изменений файлов и проверки их целостности приложение должно напрямую взаимодействовать с файловой системой на низком уровне. Язык *C++* и компилятор *GCC* позволяют работать с системными вызовами *Linux*, такими как *inotify_add_watch*(), *fanotify_init*() и *read*(), обеспечивая максимально быстрый и эффективный доступ к системным ресурсам. Это исключает необходимость использования сторонних библиотек и абстракций, что повышает надежность и контроль над процессами.

Выбранная программная платформа поддерживает механизмы событийного мониторинга файловой системы, что значительно снижает нагрузку на процессор и диск по сравнению с методами, основанными на периодическом опросе (*stat*() или *poll*()). Механизмы *inotify* и *fanotify*, встроенные в ядро *Linux*, позволяют мгновенно реагировать на изменения файлов, что критически важно для обеспечения целостности данных.

Программная платформа, основанная на *C++* и системных вызовах *Linux*, обладает высокой степенью масштабируемости. Использование объектно-ориентированного программирования (ООП) в *C++* позволяет легко расширять функциональность приложения, добавлять новые методы анализа

файлов и модули обработки событий. Кроме того, многопоточная архитектура позволяет эффективно распределять нагрузку, что делает систему пригодной для работы как на персональных компьютерах, так и на серверных решениях.

При выборе программной платформы были рассмотрены альтернативные варианты, включая *Python* и *C#*.

Python, несмотря на свою простоту и богатую экосистему библиотек, имеет недостаточную производительность для работы в реальном времени, особенно при интенсивной обработке файловых событий. Его интерпретируемая природа и механизм глобальной блокировки интерпретатора (*GIL*) делают многопоточные решения менее эффективными по сравнению с *C++*.

C# ориентирован на платформу *Windows* и требует *.NET Runtime*, что делает его менее предпочтительным для работы в среде *Linux*, где требуется непосредственное взаимодействие с ядром операционной системы.

Таким образом, выбор программной платформы основан на необходимости обеспечения высокой производительности, эффективной многопоточности и гибкости взаимодействия с ядром *Linux*. Язык *C++* предоставляет возможность работы с системными вызовами на низком уровне, что критично для мониторинга файловой системы в реальном времени. Компилятор *GCC* обеспечивает оптимизацию кода и поддержку последних стандартов языка. Средства мониторинга файлов *inotify* и *fanotify* позволяют минимизировать нагрузку на систему, обеспечивая мгновенную реакцию на изменения данных. В сравнении с альтернативными решениями, такими как *Python* и *C#*, программная платформа на основе *C++* и *GCC* является наиболее эффективным вариантом для поставленной задачи, обеспечивая баланс между производительностью, надежностью и возможностью дальнейшего расширения функциональности.

2.4 Анализ операционной системы (или другого программного обеспечения) для написания программы

Выбранной операционной системой для разработки и выполнения многопоточного фонового приложения контроля изменений и целостности файлов является *Linux*. Данная ОС была выбрана из-за ее развитой системы управления процессами и потоками, встроенных механизмов мониторинга файлов и высокой гибкости в управлении ресурсами. В данном разделе будет рассмотрена архитектура *Linux*, ее ключевые особенности, а также программные интерфейсы, которые обеспечивают эффективное выполнение поставленных задач.

Linux является многозадачной и многопользовательской операционной системой, основанной на монолитном ядре. В отличие от микрокернельных ОС, таких как *Minix*, ядро *Linux* включает в себя все основные подсистемы, что обеспечивает высокую производительность при работе с системными вызовами. Оно предоставляет широкий набор инструментов для работы с

файловой системой, управления процессами и потоками, а также взаимодействия с аппаратными ресурсами.

ОС поддерживает различные файловые системы, такие как *ext*4, *XFS*, *Btrfs*, обеспечивающие высокую скорость работы и механизмы журналирования для защиты данных. Программа контроля файлов использует преимущества событийно-ориентированного мониторинга, который встроен в ядро *Linux*, а именно *inotify* и *fanotify*.

Одной из ключевых особенностей *Linux* является гибкость управления процессами и потоками. ОС поддерживает модель легковесных процессов, где потоки одного процесса разделяют общие ресурсы, но могут исполняться независимо друг от друга. Это позволяет реализовать многопоточное приложение, эффективно распределяющее нагрузку между ядрами процессора.

Linux предоставляет *POSIX Threads* (*pthread*) – стандартный механизм управления потоками, который позволяет создавать, синхронизировать и завершать потоки внутри одного процесса. Использование *std*::*thread* в *C++* является оберткой над *pthread*, что делает работу с многопоточностью более удобной и безопасной.

Кроме того, *Linux* поддерживает асинхронные сигналы и механизм *epoll*, позволяющий эффективно отслеживать события ввода-вывода, что особенно полезно при мониторинге файлов без лишней загрузки процессора.

Linux предоставляет механизмы контроля доступа (*ACL*, *SELinux*, *AppArmor*), которые позволяют ограничивать операции с файлами. Это может быть использовано для дополнительной защиты контролируемых файлов, предотвращая их несанкционированное изменение.

Кроме того, механизм *capabilities* позволяет программе получать привилегии для работы с системой без необходимости запуска от имени суперпользователя, что снижает потенциальные риски безопасности.

Linux является оптимальной операционной системой для реализации многопоточного фонового приложения контроля файлов. Она предоставляет мощные инструменты для управления процессами, встроенные механизмы мониторинга изменений и высокую производительность за счет монолитного ядра и событийно-ориентированного ввода-вывода. Благодаря развитым системным вызовам и поддержке многопоточности, разработка эффективного и быстродействующего приложения становится возможной без избыточных затрат ресурсов.

3 ТЕОРЕТИЧЕСКОЕ ОБОСНОВАНИЕ РАЗРАБОТКИ ПРОГРАММНОГО ПРОДУКТА

3.1 Обоснование необходимости разработки

В условиях современной цифровой среды одной из наиболее актуальных проблем является обеспечение целостности и безопасности данных. С ростом объемов информации, хранящейся в компьютерных системах, возрастает вероятность их случайного или преднамеренного изменения, что может привести к серьезным последствиям. Файлы и данные могут подвергаться повреждениям из-за аппаратных сбоев, ошибок программного обеспечения, несанкционированного доступа, вредоносных программ и действий злоумышленников. В связи с этим контроль изменений и целостности файлов становится одной из важнейших задач в области информационной безопасности и администрирования вычислительных систем.

Существующие решения для мониторинга целостности файловой системы включают в себя как встроенные механизмы операционной системы, так и сторонние программные продукты. Однако большинство таких решений либо требуют значительных вычислительных ресурсов, либо не обеспечивают должного уровня гибкости и удобства использования. Например, классические антивирусные системы, выполняющие сканирование файлов на предмет изменений, зачастую работают в фоновом режиме, но при этом могут создавать значительную нагрузку на систему из-за постоянного обращения к диску и необходимости обработки больших объемов данных. Программы, использующие методы периодического опроса файловой системы, также не являются оптимальными, поскольку проверка файлов в фиксированные моменты времени может привести к тому, что определенные изменения останутся незамеченными в промежутках между сканированиями.

Разрабатываемое программное обеспечение нацелено на решение этих проблем за счет использования многопоточного подхода и низкоуровневых механизмов взаимодействия с операционной системой. Применение многопоточности позволяет эффективно распределять вычислительную нагрузку между ядрами процессора, что особенно важно при обработке большого количества файловых событий в реальном времени. В отличие от традиционных методов сканирования, предложенная программа будет использовать событийно-ориентированную модель мониторинга, основанную на встроенных в ядро *Linux* механизмах *inotify* и *fanotify*. Эти системные инструменты позволяют регистрировать изменения файлов немедленно, без необходимости постоянного сканирования всей файловой системы, что значительно снижает нагрузку на дисковую подсистему и увеличивает производительность.

Кроме того, в современных информационных системах важную роль играет не только обнаружение изменений, но и их анализ с возможностью последующего восстановления данных или предотвращения их модификации. В разрабатываемой программе планируется внедрение механизмов

логирования и анализа событий, что позволит администраторам оперативно реагировать на подозрительную активность и минимизировать потенциальные угрозы. В сочетании с возможностью гибкой настройки контролируемых файлов и каталогов это сделает программу удобным инструментом для системных администраторов и специалистов по информационной безопасности.

Таким образом, необходимость разработки данного программного продукта обусловлена актуальностью проблемы обеспечения целостности данных, недостатками существующих решений, а также возможностью повышения эффективности мониторинга файловой системы за счет использования многопоточного подхода и встроенных механизмов *Linux*. Разработанное программное обеспечение будет сочетать в себе высокую скорость работы, низкие накладные расходы и возможность адаптации под различные сценарии использования, что делает его востребованным инструментом в сфере защиты данных и администрирования информационных систем.

3.2 Технологии программирования, используемые для решения поставленных задач

Для разработки многопоточной программы, предназначенной для фонового контроля изменений и целостности группы файлов, был выбран ряд технологий, обеспечивающих высокую производительность, стабильность и минимальное использование системных ресурсов. Ключевыми факторами, которые повлияли на выбор этих технологий, стали необходимость эффективного взаимодействия с операционной системой *Linux*, поддержка многозадачности, а также возможность мониторинга файловой системы с минимальной нагрузкой на систему.

В первую очередь, для разработки приложения был выбран язык программирования *C++*, который предоставляет мощные инструменты для работы с системными ресурсами и высокой производительностью. *C++* отличается низкоуровневым доступом к памяти и процессам, что позволяет разрабатывать приложения с минимальными накладными расходами и повышенной эффективностью. В рамках задачи контроля целостности файлов, где требуется постоянная проверка и реагирование на изменения в файловой системе, такой подход является оптимальным. Использование *C++* также дает возможность активно применять объектно-ориентированное программирование, что способствует более четкой структуре кода и облегчает его поддержку и расширение в будущем.

Кроме того, *C++* идеально подходит для решения задач многозадачности. В программировании с использованием многопоточности важно эффективно управлять ресурсами системы и минимизировать время отклика на изменения, происходящие в файловой системе. Для реализации многозадачности в программе будет использоваться стандартная библиотека *C++*, которая предоставляет мощные средства для создания потоков, а также

механизмы синхронизации, такие как мьютексы и условные переменные. Эти инструменты обеспечат корректную работу программы в многозадачной среде, позволяя обрабатывать несколько файловых событий одновременно, не нарушая целостности данных и не создавая излишней нагрузки на процессор.

Кроме того, для создания и разработки программы был выбран *Visual Studio Code* как интегрированная среда разработки, которая поддерживает работу с языком *C++* и предоставляет удобные инструменты для отладки, автодополнения кода и управления проектами. Среда разработки обладает множеством расширений, которые делают процесс написания кода более удобным и эффективным. Для сборки проекта будет использован инструмент *CMake*.

Таким образом, для разработки многопоточной программы контроля изменений и целостности группы файлов были использованы эффективные и современные технологии. Язык программирования *C++* предоставляет необходимые средства для управления многозадачностью и работы с низкоуровневыми системными вызовами. В сочетании с современными средствами разработки, такими как *Visual Studio Code* и *CMake*, это решение обеспечит высокую производительность, гибкость и устойчивость программы при минимальных затратах ресурсов.

3.3 Связь архитектуры вычислительной системы с разрабатываемым программным обеспечением

Одним из важнейших элементов, необходимых для реализации задуманного приложения, является операционная система. В данном случае была выбрана *Linux*, поскольку она предоставляет мощные средства для работы с файловыми системами, а также поддерживает низкоуровневые системные вызовы, которые идеально подходят для мониторинга изменений файлов в реальном времени. *Linux*, как открытая и гибкая система, позволяет эффективно взаимодействовать с файловой системой, управлять процессами и потоками, что является ключевым аспектом при разработке программного обеспечения для таких специфичных задач, как мониторинг целостности данных и изменение состояния файлов.

В частности, операционная система *Linux* включает механизмы, такие как *inotify* и *fanotify*, которые позволяют подписываться на события, происходящие с файлами и каталогами. Эти механизмы предоставляют возможности для получения уведомлений о событиях, таких как создание, удаление, изменение или перемещение файлов, что дает возможность программе оперативно реагировать на изменения без необходимости постоянного сканирования всей файловой системы. В отличие от традиционных методов, таких как периодический опрос файловой системы, использование *inotify* и *fanotify* значительно снижает нагрузку на систему, поскольку они позволяют системе уведомить приложение только о произошедших изменениях. Это не только повышает производительность, но

и существенно экономит ресурсы, так как отсутствует необходимость в постоянном циклическом опросе.

Для взаимодействия с операционной системой, а также для работы с файлами и управления процессами в разработке программы будут активно использоваться системные вызовы *Linux*, такие как *open*(), *read*(), *write*() и *mmap*(), которые позволяют оперативно и эффективно работать с файловыми дескрипторами и памятью. Эти системные вызовы являются неотъемлемой частью операционной системы *Linux*, предоставляя программам низкоуровневый доступ к системным ресурсам и обеспечивая их эффективное использование. Например, вызовы *mmap*() позволяют работать с файлами напрямую в память, что дает возможность значительно ускорить обработку данных, особенно при работе с большими объемами информации. В свою очередь, системные вызовы *read*() и *write*() будут использованы для прямого взаимодействия с файлами, что позволяет добиться высокой производительности при обработке изменений.

Кроме того, системные механизмы многозадачности и синхронизации процессов, реализованные в *Linux*, играют ключевую роль в обеспечении эффективной работы программы. *Linux* предоставляет широкий спектр инструментов для организации многозадачности, включая управление потоками, синхронизацию потоков и процессы, что позволяет распределять нагрузку между несколькими ядрами процессора. Это особенно важно для задачи мониторинга файловой системы в реальном времени, поскольку многопоточность позволяет программе одновременно обрабатывать несколько изменений в файловой системе, не блокируя выполнение других операций. Возможности операционной системы для организации взаимодействия между потоками и их синхронизации позволяют минимизировать время отклика на события, что критически важно для приложений, где задержки в обработке изменений могут привести к потерям данных или нарушению целостности системы.

Таким образом, выбор операционной системы *Linux* обоснован ее мощными возможностями для работы с файловыми системами, эффективным использованием системных вызовов и механизма многозадачности. Эти особенности позволяют разрабатывать высокопроизводительные и гибкие приложения для контроля изменений файлов, что соответствует требованиям, предъявляемым к проекту. Связь архитектуры вычислительной системы и разрабатываемого программного обеспечения заключается в том, что операционная система предоставляет необходимые низкоуровневые механизмы для взаимодействия с файловой системой, эффективного мониторинга изменений и управления процессами, что обеспечит стабильную работу программы при минимальной нагрузке на систему.

4 ПРОЕКТИРОВАНИЕ ФУНКЦИОНАЛЬНЫХ ВОЗМОЖНОСТЕЙ ПРОГРАММЫ

4.1 Описание функций программного обеспечения

Разработанная программа будет предоставлять несколько ключевых функциональных возможностей, которые обеспечат эффективное управление группами отслеживаемых файлов, выполнение резервного копирования, логирование изменений, обработку сигналов операционной системы и формирование отчетов. Все эти функции направлены на выполнение задачи контроля изменений и целостности файлов, обеспечивая при этом высокую производительность и гибкость в использовании.

Одной из главных функций программы является управление группами отслеживаемых файлов. В рамках этой функции будет реализованы механизмы создания, чтения, обновления и удаления групп отслеживаемых файлов. Это позволит пользователю или системному администратору легко управлять набором файлов, которые должны подвергаться мониторингу на изменения. Пользователь будет иметь возможность динамически добавлять новые файлы или папки в группу отслеживаемых объектов, а также удалять их, если они больше не нуждаются в мониторинге.

Для удобства работы с отслеживаемыми объектами и обеспечения надежности работы системы, предусмотрена возможность резервного копирования данных. При изменении данных в отслеживаемых файлах или папках, программа будет автоматически создавать их резервные копии. Это позволит не только обеспечить сохранность данных, но и иметь возможность восстановления исходного состояния файлов в случае их повреждения или потери.

Логирование изменений является важной составляющей функциональности программы. Каждый факт изменения в отслеживаемых файлах будет фиксироваться в лог-файле с указанием даты, времени и типа изменения, например, создание, изменение или удаление. Логирование изменений позволяет обеспечить прозрачность работы программы, а также даст возможность отслеживать все действия, происходящие с файлами. Это особенно важно для мониторинга целостности данных и контроля за возможными несанкционированными изменениями. В случае ошибок или сбоев в работе программы, логи могут служить полезным инструментом для диагностики и устранения проблемы.

Обработка сигналов операционной системы является еще одной важной функцией программы. Программа будет настроена на обработку сигналов, таких как *SIGHUP*, *SIGINT*, *SIGTERM* и других. Это позволит программе корректно завершить свою работу или выполнить нужные действия при возникновении определенных событий, таких как запрос на завершение работы или сигнал об изменении конфигурации. Обработка сигналов позволит повысить надежность программы, а также обеспечить гибкость в управлении её состоянием.

Для того чтобы пользователи могли отслеживать состояние программы и получать подробную информацию о выполненных действиях, будет предусмотрено формирование отчетов. Поскольку программа является фоновым процессом, отчеты будут записываться в файл, что обеспечит удобство их хранения и дальнейшего анализа. Формирование отчетов будет происходить в формате *JSON* или *XML*, что обеспечит гибкость в их представлении и позволяет легко адаптировать вывод отчета под различные требования. Данные отчеты могут содержать информацию о произведенных изменениях в файловой системе, а также о ходе работы программы и возникших ошибках.

Таким образом, функциональность программы будет направлена на обеспечение надежного и эффективного мониторинга изменений файлов, удобного управления группами отслеживаемых объектов и прозрачного логирования всех действий. Возможности резервного копирования и обработки сигналов операционной системы повысят надежность программы и обеспечат её стабильную работу в различных условиях.

4.2 Описание пользовательского интерфейса программного продукта

Разрабатываемое программное обеспечение представляет собой фоновый процесс, функционирующий в режиме демона и выполняющий задачи мониторинга изменений файлов, контроля их целостности, ведения логов и резервного копирования без необходимости постоянного взаимодействия с пользователем. В связи с этим программный продукт не требует активного пользовательского интерфейса в традиционном понимании, поскольку его основная задача заключается в обеспечении непрерывного контроля файловой системы на уровне операционной системы. Однако для конфигурирования параметров работы, изменения списка отслеживаемых файлов, просмотра отчетов и логов, а также управления процессом мониторинга необходимо предоставить средства взаимодействия с программой, которые могут быть реализованы различными способами.

Архитектура взаимодействия между пользователем и программным обеспечением построена по принципу клиент-серверной модели, при которой основной процесс, работающий в фоновом режиме, выполняет роль сервера, а средства управления конфигурацией представляют собой клиентскую часть. Такой подход позволяет реализовать гибкую систему, в которой пользовательский интерфейс не оказывает непосредственного влияния на работу системы мониторинга, а лишь предоставляет удобный способ управления ее параметрами. Данная модель взаимодействия схожа с подходом, применяемым в системах, подобных *Docker*, где основной демон выполняет все вычислительные операции, а пользовательский интерфейс представлен в виде командных утилит или веб-интерфейса, обеспечивающих взаимодействие с серверной частью.

На данном этапе разработки конкретный вариант пользовательского интерфейса не определен, поскольку его реализация не является ключевой целью данной работы. Возможные варианты реализации могут включать как консольные инструменты для управления программой посредством командной строки, так и графический интерфейс или веб-панель, позволяющие выполнять основные операции через удобный пользовательский интерфейс. Важной особенностью проектируемого решения является то, что средства управления конфигурацией и просмотра отчетов разрабатываются отдельно от основной логики мониторинга, что позволяет обеспечить независимость фонового процесса от способа взаимодействия с пользователем и обеспечивает возможность последующего расширения функциональности без необходимости внесения изменений в основную архитектуру системы.

Поскольку данная курсовая работа посвящена изучению системного программирования, ключевое внимание уделяется разработке фонового процесса, взаимодействию с операционной системой и использованию системных вызовов для работы с файлами и процессами. В этом контексте пользовательский интерфейс рассматривается как вспомогательный компонент, необходимый для обеспечения удобства работы с программой, но не являющийся центральным элементом проектирования. В дальнейшем возможна разработка различных интерфейсов, включая командные утилиты, графические оболочки или веб-приложения, однако в рамках данной работы основное внимание сосредоточено на функциональности самого фонового процесса и его интеграции с механизмами операционной системы.

5 АРХИТЕКТУРА РАЗРАБАТЫВАЕМОЙ ПРОГРАММЫ

5.1 Общая структура программы

Разрабатываемое программное обеспечение реализовано в соответствии с архитектурой клиент-сервер, где серверная часть представляет собой фоновый процесс (демон), выполняющий непрерывное отслеживание изменений в файловой системе, а клиентская часть предназначена для управления конфигурацией, просмотра отчетов и взаимодействия с пользователем. Основная работа программы сосредоточена в серверной части, которая управляет процессами мониторинга, координирует потоки и обрабатывает события, поступающие от операционной системы.

Внутренняя структура программы основана на многопоточном подходе, что позволяет эффективно распределять вычислительную нагрузку и минимизировать задержки при обработке событий. Основной поток программы выполняет роль диспетчера, отвечающего за инициализацию системы мониторинга, управление списком отслеживаемых файлов и создание рабочих потоков. Рабочие потоки обрабатывают поступающие от операционной системы события, такие как создание, удаление, изменение файлов и каталогов.

Для получения информации об изменениях файловой системы программа использует механизм *inotify*, который предоставляет возможность подписки на события и позволяет мгновенно реагировать на любые изменения в указанных директориях. После получения события основной поток передает его в очередь задач, откуда оно распределяется между рабочими потоками. Эти потоки выполняют необходимые операции, такие как логирование изменений, резервное копирование или проверка целостности файлов. Такой подход обеспечивает высокую производительность и позволяет избежать избыточного опроса файловой системы, снижая нагрузку на систему и ресурсы процессора.

Дополнительно программа поддерживает обработку системных сигналов, что позволяет корректно завершать работу, обновлять конфигурацию в реальном времени и перезапускать мониторинг без остановки всей системы. Клиентская часть может взаимодействовать с сервером с использованием локального межпроцессного взаимодействия, например, через сокеты, файлы конфигурации или специализированные *IPC*-механизмы, такие как *UNIX*-сокеты или именованные каналы.

Данная архитектура обеспечивает модульность и гибкость программы, позволяя легко расширять ее функциональность без изменения базовых механизмов мониторинга. Многопоточное исполнение и использование событийно-ориентированной модели обработки данных позволяют программе работать эффективно даже при высокой интенсивности изменений в файловой системе, что делает ее надежным инструментом для контроля целостности данных.

5.2 Описание функциональной схемы программы

После запуска программа выполняет этап инициализации, в ходе которого загружается конфигурация, содержащая перечень отслеживаемых директорий и файлов, параметры логирования, настройки резервного копирования и контрольных проверок. На основе этих данных создаются структуры данных, необходимые для работы механизма мониторинга. Затем основной поток регистрирует файлы и каталоги в *inotify*, подписываясь на события, связанные с их изменением.

При поступлении сигнала об изменении состояния объекта файловой системы основной поток анализирует полученные данные и передает их в очередь событий. В этот момент выполняется фильтрация событий для устранения дублирующих или незначительных изменений. Рабочие потоки, взаимодействуя с очередью, извлекают события и выполняют соответствующие операции. Если событие связано с модификацией файла, поток считывает его новое состояние, формирует контрольную сумму и сравнивает с ранее сохраненным значением для выявления изменений. В случае обнаружения модификаций выполняются заранее заданные действия, такие как резервное копирование файла, сохранение новой контрольной суммы или запись информации в журнал логов.

Особое внимание уделяется синхронизации потоков, так как одновременно может обрабатываться сразу несколько событий. Для предотвращения состояний гонки используются механизмы блокировок или атомарные операции. Например, при работе с лог-файлом или структурой данных, содержащей контрольные суммы, потоки координируют доступ друг к другу с помощью мьютексов или других примитивов синхронизации.

Дополнительно предусмотрен механизм обработки команд от пользователя, поступающих через клиентский интерфейс. Запросы могут включать изменение списка отслеживаемых файлов, принудительный запуск проверки целостности или формирование отчета. В этом случае информация передается серверному процессу, который обновляет настройки без остановки работы мониторинга.

Таким образом, функциональная схема программы представляет собой последовательность взаимодействий между основными модулями: механизм мониторинга файловой системы генерирует события, основной поток диспетчеризирует их и передает в очередь, рабочие потоки выполняют обработку событий и применяют соответствующие действия, а клиентский интерфейс обеспечивает управление настройками и просмотр отчетов. Такой подход позволяет эффективно организовать работу системы, минимизируя нагрузку на вычислительные ресурсы и обеспечивая высокую скорость реакции на изменения в файловой системе.

5.3 Описание блок-схемы алгоритма программы

Разрабатываемая программа представляет собой фоновый многопоточный сервис, выполняющий контроль изменений в группе файлов с возможностью логирования, резервного копирования и формирования отчетов. Для реализации этих функций используется взаимодействие между основным потоком, который управляет общим процессом мониторинга, и рабочими потоками, занимающимися обработкой событий, поступающих от операционной системы. Алгоритм работы программы включает несколько ключевых этапов, каждый из которых выполняет строго определенные задачи, обеспечивая согласованность работы системы в целом.

На первом этапе после запуска программы выполняется ее инициализация. В ходе этого процесса загружается конфигурация, содержащая перечень отслеживаемых директорий и файлов, параметры логирования, настройки резервного копирования, а также механизмы контроля целостности данных. После загрузки конфигурации выполняется проверка существования всех указанных файлов и каталогов, а также при необходимости создаются недостающие структуры. Далее происходит инициализация механизмов взаимодействия с операционной системой, включая установку *inotify* для подписки на события файловой системы, а также запуск обработчиков системных сигналов, позволяющих корректно завершать выполнение программы или обновлять ее настройки без необходимости полной перезагрузки. Если программа должна работать в фоновом режиме, на этом этапе создается процесс-демон, отделяющийся от терминала и перенаправляющий потоки ввода-вывода.

После успешного завершения инициализации основной поток программы переходит в режим ожидания событий от файловой системы. При возникновении изменений *inotify* передает уведомление, содержащее информацию о типе события, имени файла и его местоположении в файловой системе. Полученные данные подвергаются первичному анализу, в ходе которого определяется, является ли событие значимым. Если, например, событие связано с временными файлами или системными операциями, оно может быть проигнорировано. В противном случае информация о нем передается в очередь событий, откуда она будет обработана рабочими потоками.

Рабочий поток, получив событие на обработку, анализирует текущее состояние соответствующего файла. В первую очередь считываются его атрибуты, включая размер, время последнего изменения и контрольную сумму. Далее полученные данные сравниваются с ранее сохраненными значениями. Если контрольная сумма изменилась, это свидетельствует о модификации содержимого файла. В этом случае программа выполняет необходимые действия, такие как создание резервной копии измененной версии или запись информации в журнал логов.

Помимо обработки изменений файлов, программа также поддерживает взаимодействие с клиентской частью, обеспечивая возможность управления

конфигурацией и получения отчетов о работе системы. Для этого реализован механизм межпроцессного взаимодействия, который может быть основан на *UNIX*-сокетах или другом протоколе передачи данных. Когда пользователь отправляет запрос, например, на добавление новой директории в список мониторинга или на получение актуального отчета, основной процесс принимает команду, анализирует ее и выполняет соответствующие действия. Если запрос требует изменения конфигурации, вносятся соответствующие корректировки, а при необходимости пересоздаются структуры данных для учета новых параметров без остановки работы основного процесса.

Корректное завершение работы программы также является важным этапом алгоритма, поскольку оно предполагает освобождение всех занятых ресурсов и предотвращение потери данных. При получении сигнала завершения работы программа сначала завершает выполнение всех активных потоков, обрабатывая оставшиеся задачи в очереди. После этого производится деинициализация механизмов мониторинга файловой системы, закрытие всех открытых дескрипторов и сохранение финального состояния системы в лог. Завершающим этапом является завершение работы серверного процесса, что приводит к остановке всей системы мониторинга.

Таким образом, блок-схема алгоритма программы представляет собой последовательность взаимодействий между основным потоком, рабочими потоками и клиентскими запросами. Основной поток управляет подпиской на события и распределением задач, рабочие потоки выполняют обработку файловых изменений, а клиентская часть обеспечивает возможность настройки конфигурации и просмотра отчетов. Такой подход позволяет реализовать эффективную и отказоустойчивую систему мониторинга, минимизируя нагрузку на вычислительные ресурсы и обеспечивая высокую скорость реагирования на изменения в файловой системе.

ЗАКЛЮЧЕНИЕ

В процессе выполнения курсового проекта, была спроектирована и реализована многопоточная программа, предназначенная для фонового контроля изменений и проверки целостности группы файлов в операционной системе Linux. В рамках работы был выполнен полный цикл предварительного проектирования: проведён анализ вычислительной и программной платформы, выбраны технологии реализации, описана архитектура приложения и разработаны алгоритмы основных компонентов системы.

Программа, проектируемая в ходе данного исследования, основывается на механизмах взаимодействия с ядром Linux, в частности на использовании подсистем inotify и системных вызовов, обеспечивающих эффективный мониторинг файловой системы в режиме реального времени. Также была реализована базовая логика работы фонового демона, способного обрабатывать сигналы операционной системы, управлять потоками и вести логирование событий, что приближает программу к уровню промышленного программного обеспечения системного уровня.

Практическая значимость разработанного решения заключается в возможности его использования для контроля за изменениями в критически важных директориях, что особенно актуально в контексте обеспечения безопасности и предотвращения несанкционированного изменения данных. Кроме того, выбранный подход демонстрирует ключевые приёмы системного программирования, включая управление потоками, взаимодействие с ядром операционной системы и работу с файловыми дескрипторами.

Таким образом, поставленные в рамках курсового проектирования задачи были успешно решены, а проведённая работа позволила на практике закрепить знания в области операционных систем, системных вызовов, многопоточности и архитектуры прикладного программного обеспечения в контексте Linux.

СПИСОК ЛИТЕРАТУРНЫХ ИСТОЧНИКОВ

[1] Документация к дистрибутиву Ubuntu [Электронный ресурс]. – Режим доступа: https://help.ubuntu.com/. Дата доступа: 10.02.2025.

[2] Документация к утилите inotify [Электронный ресурс]. – Режим доступа: https://man7.org/linux/man-pages/man7/inotify.7.html. Дата доступа: 10.02.2025.

[3] Документация к pthreads [Электронный ресурс]. – Режим доступа: https://man7.org/linux/man-pages/man7/pthreads.7.html. Дата доступа: 10.02.2025.

[4] Бах, М. Дж. The Design of the UNIX Operating System / М. Дж. Бах. – М.: Вильямс, 2003. – 336 с.

[5] Документация Windows API [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/windows/win32/api/. Дата доступа: 17.03.2025.

[6] Документация к FileSystemWatcher [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/dotnet/api/system.io.filesystemwatcher ?view=net-9.0. Дата доступа: 17.03.2025.

[7] Докуменация к FSEvents [Электронный ресурс]. – Режим доступа: https://developer.apple.com/documentation/coreservices/file_system_events. Дата доступа: 17.03.2025.

[8] Документация к файловой системе btrfs [Электронный ресурс]. – Режим доступа: https://btrfs.readthedocs.io/en/latest/. Дата доступа: 17.03.2025.

[9] Документация к CFS [Электронный ресурс]. – Режим доступа: https://docs.kernel.org/scheduler/sched-design-CFS.html. Дата доступа: 17.03.2025.

[10] Документация к mmap [Электронный ресурс]. – Режим доступа: https://man7.org/linux/man-pages/man2/mmap.2.html. Дата доступа: 17.03.2025.

[11] Таненбаум, Э. С. Современные операционные системы / Э. С. Таненбаум, Х. Бос ; пер. с англ. – 4-е изд. – СПб.: Питер, 2020. – 1136 с.

[12] Документация к fork [Электронный ресурс]. – Режим доступа: https://man7.org/linux/man-pages/man2/fork.2.html. Дата доступа: 17.04.2025.

[13] Столлингс, У. Криптография и сетовая безопасность: принципы и практика / У. Столлингс ; пер. с англ. – 7-е изд. – М.: Вильямс, 2017. – 944 с.

[14] Документация Microsoft C++ [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/cpp/?view=msvc-170. Дата доступа: 17.04.2025.

[15] Документация Visual Studio Code [Электронный ресурс]. – Режим доступа: https://code.visualstudio.com/docs. Дата доступа: 17.04.2025.

[16] Документация к компилятору GCC [Электронный ресурс]. – Режим доступа: https://gcc.gnu.org/onlinedocs/. Дата доступа: 17.04.2025.

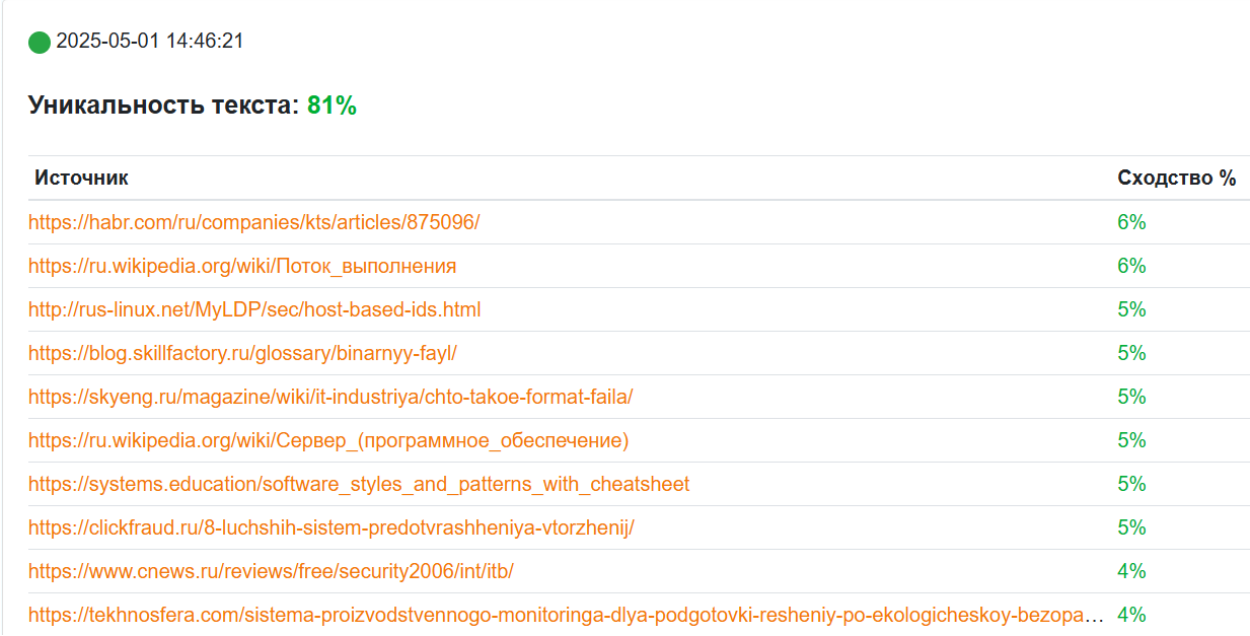
[17] Документация к inotify_add_watch [Электронный ресурс]. – Режим доступа: https://man7.org/linux/man-pages/man2/inotify_add_watch.2.html. Дата доступа: 17.04.2025.

[18] Документация OpenSSL [Электронный ресурс]. – Режим доступа: https://docs.openssl.org/master/. Дата доступа: 17.04.2025.

[19] Документация к стандартной библиотеке C++ [Электронный ресурс]. – Режим доступа: https://learn.microsoft.com/en-us/cpp/standard-library/cpp-standard-library-reference?view=msvc-170. Дата доступа: 17.04.2025.

[20] Документация Python [Электронный ресурс]. – Режим доступа: https://docs.python.org/3/. Дата доступа: 17.04.2025.

ПРИЛОЖЕНИЕ А
(обязательное)
Справка о проверке на заимствования



2025-05-01 14:46:21

Уникальность текста: 81%

Источник	Сходство %
https://habr.com/ru/companies/kts/articles/875096/	6%
https://ru.wikipedia.org/wiki/Поток_выполнения	6%
http://rus-linux.net/MyLDP/sec/host-based-ids.html	5%
https://blog.skillfactory.ru/glossary/binarnyy-fayl/	5%
https://skyeng.ru/magazine/wiki/it-industriya/chto-takoe-format-faila/	5%
https://ru.wikipedia.org/wiki/Сервер_(программное_обеспечение)	5%
https://systems.education/software_styles_and_patterns_with_cheatsheet	5%
https://clickfraud.ru/8-luchshih-sistem-predotvrashheniya-vtorzhenij/	5%
https://www.cnews.ru/reviews/free/security2006/int/itb/	4%
https://tekhnosfera.com/sistema-proizvodstvennogo-monitoringa-dlya-podgotovki-resheniy-po-ekologicheskoy-bezopa…	4%

Рисунок 1 – Справка о проверке на заимствования

ПРИЛОЖЕНИЕ Б
(обязательное)
Листинг программного кода

Листинг Б.1 – Содержимое файла main.cpp

```
#include <filesystem>
#include <iostream>
#include <vector>
#include <memory>
#include "ConfigLoader.hpp"
#include "VaultService.hpp"
#include "ChecksumService.hpp"
#include "InitializationService.hpp"
#include "TrackingFile.hpp"
#include "StatePersistenceService.hpp"

void processDirectory(const std::filesystem::path& dirPath,
InitializationService& initializer, std::vector<TrackingFile>& trackedFiles,
bool recursive) {
    if (!std::filesystem::exists(dirPath) ||
!std::filesystem::is_directory(dirPath)) {
        std::cerr << "Путь не является директорией: " << dirPath <<
std::endl;
        return;
    }

    try {
        if (recursive) {
            for (const auto& entry :
std::filesystem::recursive_directory_iterator(dirPath)) {
                if (std::filesystem::is_regular_file(entry)) {
                    try {
                        TrackingFile tf =
initializer.initialize(entry.path().string());
                        trackedFiles.push_back(tf);
                        std::cout << "  → Инициализирован файл: " <<
entry.path() << std::endl;
                    } catch (const std::exception& ex) {
                        std::cerr << "  ⚠ Ошибка инициализации файла " <<
entry.path()
                                  << ": " << ex.what() << std::endl;
                    }
                }
            }
        } else {
            for (const auto& entry :
std::filesystem::directory_iterator(dirPath)) {
                if (std::filesystem::is_regular_file(entry)) {
                    try {
                        TrackingFile tf =
initializer.initialize(entry.path().string());
                        trackedFiles.push_back(tf);
                        std::cout << "  → Инициализирован файл: " <<
entry.path() << std::endl;
                    } catch (const std::exception& ex) {
                        std::cerr << "  ⚠ Ошибка инициализации файла " <<
entry.path()
                                  << ": " << ex.what() << std::endl;
                    }
                }
            }
```

```
        }
    } catch (const std::filesystem::filesystem_error& fe) {
        std::cerr << "  ⚠ Ошибка обхода директории " << dirPath << ": " <<
fe.what() << std::endl;
    }
}

int main() {
    const std::string configPath = "config.json";

    ConfigLoader loader(configPath);
    if (!loader.load()) {
        std::cerr << "Ошибка загрузки конфигурации!" << std::endl;
        return 1;
    }

    const auto& groups = loader.getMonitoringGroups();
    std::cout << "Загружено групп: " << groups.size() << std::endl;

    VaultService vault(".filevault");
    ChecksumService checksum;
    InitializationService initializer(vault, checksum);

    std::vector<TrackingFile> trackedFiles;

    for (const auto& group : groups) {
        std::cout << "Группа ID: " << group.id << std::endl;
        std::cout << "Описание: " << group.description << std::endl;

        for (const auto& path : group.paths) {
            if (std::filesystem::is_regular_file(path.path)) {
                // Если путь — файл
                try {
                    TrackingFile tf = initializer.initialize(path.path);
                    trackedFiles.push_back(tf);
                    std::cout << "  → Инициализирован файл: " << path.path <<
std::endl;
                } catch (const std::exception& ex) {
                    std::cerr << "  ⚠ Ошибка инициализации файла " <<
path.path
                              << ": " << ex.what() << std::endl;
                }
            }
            else if (std::filesystem::is_directory(path.path)) {
                // Если путь — директория и нужно обрабатывать рекурсивно
                std::cout << "  → Инициализация директории: " << path.path <<
std::endl;
                processDirectory(path.path, initializer, trackedFiles,
path.recursive);
            }
        }

        std::cout << "------------------------------" << std::endl;
    }

    std::cout << "Отслеживаемых файлов: " << trackedFiles.size() <<
std::endl;

    StatePersistenceService dbService("tracking.db");
    dbService.initializeSchema();

    for (auto& file : trackedFiles)
    {
```

```
        dbService.createTrackingFile(file);
    }

    return 0;
}
```

Листинг Б.2 – Загрузчик конфигураций из json формата

```
#pragma once

#include <string>
#include <vector>
#include <memory>
#include <nlohmann/json.hpp>

struct PathConfig {
    std::string path;
    bool recursive;
};

struct ChecksumConfig {
    bool enabled;
    std::string algorithm;
};

struct MonitoringGroup {
    std::string id;
    std::string description;
    std::vector<PathConfig> paths;
    std::vector<std::string> events;
    ChecksumConfig checksum;
};

class ConfigLoader {
public:
    explicit ConfigLoader(const std::string& configPath);

    bool load(); // Загрузка конфига
    const std::vector<MonitoringGroup>& getMonitoringGroups() const;

private:
    std::string m_configPath;
    std::vector<MonitoringGroup> m_monitoringGroups;

    void parse(const nlohmann::json& root); // Разбор JSON
};

#include "ConfigLoader.hpp"
#include <fstream>
#include "nlohmann/json.hpp"

ConfigLoader::ConfigLoader(const std::string& configPath)
    : m_configPath(configPath) {}

bool ConfigLoader::load() {
    std::ifstream file(m_configPath);
    if (!file.is_open()) {
        return false;
    }

    nlohmann::json root;
    try {
        file >> root;
        parse(root);
```

```
    } catch (const nlohmann::json::exception& ex) {
        // Можно логировать ошибку
        return false;
    }

    return true;
}

void ConfigLoader::parse(const nlohmann::json& root) {
    m_monitoringGroups.clear();
    auto groups = root.at("monitoring").at("groups");

    for (const auto& group : groups) {
        MonitoringGroup mg;
        mg.id = group.at("id").get<std::string>();
        mg.description = group.at("description").get<std::string>();

        for (const auto& pathObj : group.at("paths")) {
            PathConfig pc;
            pc.path = pathObj.at("path").get<std::string>();
            pc.recursive = pathObj.at("recursive").get<bool>();
            mg.paths.push_back(pc);
        }

        mg.events = group.at("events").get<std::vector<std::string>>();

        auto checksumObj = group.at("checksum");
        mg.checksum.enabled = checksumObj.at("enabled").get<bool>();
        if (mg.checksum.enabled) {
            mg.checksum.algorithm =
checksumObj.at("algorithm").get<std::string>();
        }

        m_monitoringGroups.push_back(mg);
    }
}

const std::vector<MonitoringGroup>& ConfigLoader::getMonitoringGroups() const
{
    return m_monitoringGroups;
}
```

Листинг Б.3 – Модель данных предметной области

```
#pragma once

#include <string>
#include <list>
#include <chrono>
#include <uuid/uuid.h>

struct FileChange
{
    std::chrono::system_clock::time_point timestamp;
    std::string changeType;
    std::string checksum;
    std::string savedVersionId;
    std::string user;
    std::string additionalInfo;
};

struct FileHistory
{
    std::list<FileChange> changes;
```

```
};

struct TrackingFile
{
    std::string filePath;
    std::string fileId;
    FileHistory history;
    std::string lastChecksum;
    bool isMissing = false;
};
```

Листинг Б.4 – Сервис создания резервных копий отслеживаемых файлов

```
#pragma once

#include <string>
#include <filesystem>

class VaultService {
public:
    VaultService(std::filesystem::path vaultRoot);

    std::string save(const std::filesystem::path& filePath); // returns
versionId
    bool restore(const std::string& versionId, const std::filesystem::path&
destination);
    bool exists(const std::string& versionId) const;

private:
    std::filesystem::path vaultDir;
};

#include "VaultService.hpp"
#include <filesystem>
#include <fstream>
#include <iostream>
#include <random>
#include <sstream>
#include <iomanip>

VaultService::VaultService(std::filesystem::path vaultRoot)
    : vaultDir(vaultRoot) {
    if (!std::filesystem::exists(vaultDir)) {
        std::filesystem::create_directory(vaultDir);
    }
}

std::string VaultService::save(const std::filesystem::path& filePath) {
    std::random_device rd;
    std::uniform_int_distribution<int> dist(0, 15);
    std::ostringstream versionId;

    // Генерация случайного versionId (UUID-like)
    for (int i = 0; i < 8; ++i) {
        versionId << std::hex << dist(rd);
    }

    std::filesystem::path destination = vaultDir / versionId.str();
    std::filesystem::copy(filePath, destination,
std::filesystem::copy_options::overwrite_existing);

    return versionId.str();
}
```

```
bool VaultService::restore(const std::string& versionId, const
std::filesystem::path& destination) {
    std::filesystem::path source = vaultDir / versionId;
    if (std::filesystem::exists(source)) {
        std::filesystem::copy(source, destination,
std::filesystem::copy_options::overwrite_existing);
        return true;
    }
    return false;
}

bool VaultService::exists(const std::string& versionId) const {
    std::filesystem::path versionPath = vaultDir / versionId;
    return std::filesystem::exists(versionPath);
}
```

Листинг Б.5 – Сервис вычисления контрольной суммы

```
#pragma once

#include <string>
#include <filesystem>

class ChecksumService {
public:
    static std::string compute(const std::filesystem::path& filePath);
};

// ChecksumService.cpp
#include "ChecksumService.hpp"
#include <openssl/sha.h>
#include <fstream>
#include <sstream>
#include <iomanip>

std::string ChecksumService::compute(const std::filesystem::path& filePath) {
    std::ifstream file(filePath, std::ios::binary);
    if (!file) {
        throw std::runtime_error("Не удалось открыть файл для вычисления
контрольной суммы");
    }

    SHA256_CTX sha256Context;
    SHA256_Init(&sha256Context);

    char buffer[1024];
    while (file.read(buffer, sizeof(buffer))) {
        SHA256_Update(&sha256Context, buffer, file.gcount());
    }
    SHA256_Update(&sha256Context, buffer, file.gcount());  // Для последней
части

    unsigned char hash[SHA256_DIGEST_LENGTH];
    SHA256_Final(hash, &sha256Context);

    std::ostringstream result;
    for (int i = 0; i < SHA256_DIGEST_LENGTH; i++) {
        result << std::hex << std::setw(2) << std::setfill('0') <<
(int)hash[i];
    }

    return result.str();
}
```

Листинг Б.6 – Сервис сохранения событий в базу данных

```
#pragma once

#include <string>
#include <vector>
#include <sqlite3.h>
#include "TrackingFile.hpp"

class StatePersistenceService {
public:
    StatePersistenceService(const std::string& dbPath);

    void initializeSchema(); // Создание таблиц при первом запуске
    void createTrackingFile(TrackingFile& file);
    void saveTrackingFile(const TrackingFile& file);
    void saveFileChange(const std::string& fileId, const FileChange& change);

    std::vector<TrackingFile> loadTrackedFiles(); // Восстановление состояния

    ~StatePersistenceService();

private:
    std::string toIsoString(const std::chrono::system_clock::time_point& tp);
    std::chrono::system_clock::time_point fromIsoString(const std::string&
str);
    void execute(const std::string& sql);

    sqlite3* db;
};

#include "StatePersistenceService.hpp"
#include <iostream>
#include <sstream>
#include <iomanip>
#include <chrono>
#include <sqlite3.h>

StatePersistenceService::StatePersistenceService(const std::string& dbPath) {
    if (sqlite3_open(dbPath.c_str(), &db) != SQLITE_OK) {
        throw std::runtime_error("Не удалось открыть БД: " +
std::string(sqlite3_errmsg(db)));
    }
}

StatePersistenceService::~StatePersistenceService() {
    if (db) sqlite3_close(db);
}

void StatePersistenceService::initializeSchema() {
    const std::string drop = R"SQL(
        drop table file_changes;
        drop table tracking_files;
    )SQL";
    execute(drop); //DEBUG DROP ONLY, SHOULD BE DELETED

    const std::string sql = R"SQL(
        CREATE TABLE IF NOT EXISTS tracking_files (
            file_id integer PRIMARY KEY AUTOINCREMENT,
            file_path TEXT NOT NULL,
            last_checksum TEXT,
            is_missing INTEGER NOT NULL
        );
```

```
        CREATE TABLE IF NOT EXISTS file_changes (
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            file_id integer NOT NULL,
            timestamp TEXT NOT NULL,
            change_type TEXT NOT NULL,
            checksum TEXT,
            saved_version_id TEXT,
            user TEXT,
            additional_info TEXT,
            FOREIGN KEY (file_id) REFERENCES tracking_files(file_id)
        );
    )SQL";
    execute(sql);
}

void StatePersistenceService::execute(const std::string& sql) {
    char* errMsg = nullptr;
    if (sqlite3_exec(db, sql.c_str(), nullptr, nullptr, &errMsg) !=
SQLITE_OK) {
        std::string msg = errMsg ? errMsg : "Unknown error";
        sqlite3_free(errMsg);
        throw std::runtime_error("SQL error: " + msg);
    }
}

void StatePersistenceService::createTrackingFile(TrackingFile& file) {
    const std::string sql = "INSERT INTO tracking_files (file_path,
last_checksum, is_missing) VALUES (?, ?, ?);";
    sqlite3_stmt* stmt;
    sqlite3_prepare_v2(db, sql.c_str(), -1, &stmt, nullptr);
    sqlite3_bind_text(stmt, 1, file.filePath.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 2, file.lastChecksum.c_str(), -1,
SQLITE_TRANSIENT);
    sqlite3_bind_int(stmt, 3, file.isMissing ? 1 : 0);
    sqlite3_step(stmt);
    sqlite3_finalize(stmt);

    // Получаем автоинкрементный ID
    sqlite3_int64 rowId = sqlite3_last_insert_rowid(db);
    file.fileId = std::to_string(rowId);  // сохраняем в структуру для
дальнейшего использования

    for (const auto& change : file.history.changes) {
        saveFileChange(file.fileId, change);
    }
}

void StatePersistenceService::saveTrackingFile(const TrackingFile& file) {
    const std::string sql = "INSERT OR REPLACE INTO tracking_files (file_id,
file_path, last_checksum, is_missing) VALUES (?, ?, ?, ?);";
    sqlite3_stmt* stmt;
    sqlite3_prepare_v2(db, sql.c_str(), -1, &stmt, nullptr);
    sqlite3_bind_text(stmt, 1, file.fileId.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 2, file.filePath.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 3, file.lastChecksum.c_str(), -1,
SQLITE_TRANSIENT);
    sqlite3_bind_int(stmt, 4, file.isMissing ? 1 : 0);
    sqlite3_step(stmt);
    sqlite3_finalize(stmt);

    for (const auto& change : file.history.changes) {
        saveFileChange(file.fileId, change);
    }
}
```

```
void StatePersistenceService::saveFileChange(const std::string& fileId, const
FileChange& change) {
    const std::string sql = "INSERT INTO file_changes (file_id, timestamp,
change_type, checksum, saved_version_id, user, additional_info) VALUES (?, ?,
?, ?, ?, ?, ?);";
    sqlite3_stmt* stmt;
    sqlite3_prepare_v2(db, sql.c_str(), -1, &stmt, nullptr);
    sqlite3_bind_text(stmt, 1, fileId.c_str(), -1, SQLITE_TRANSIENT);
    std::string ts = toIsoString(change.timestamp);
    sqlite3_bind_text(stmt, 2, ts.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 3, change.changeType.c_str(), -1,
SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 4, change.checksum.c_str(), -1,
SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 5, change.savedVersionId.c_str(), -1,
SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 6, change.user.c_str(), -1, SQLITE_TRANSIENT);
    sqlite3_bind_text(stmt, 7, change.additionalInfo.c_str(), -1,
SQLITE_TRANSIENT);
    sqlite3_step(stmt);
    sqlite3_finalize(stmt);
}

std::string StatePersistenceService::toIsoString(const
std::chrono::system_clock::time_point& tp) {
    auto t = std::chrono::system_clock::to_time_t(tp);
    std::stringstream ss;
    ss << std::put_time(std::gmtime(&t), "%FT%TZ");
    return ss.str();
}

std::chrono::system_clock::time_point
StatePersistenceService::fromIsoString(const std::string& str) {
    std::tm tm{};
    std::istringstream ss(str);
    ss >> std::get_time(&tm, "%Y-%m-%dT%H:%M:%SZ");
    return std::chrono::system_clock::from_time_t(std::mktime(&tm));
}

// Метод loadTrackedFiles будет добавлен по запросу
std::vector<TrackingFile> StatePersistenceService::loadTrackedFiles() {
    std::vector<TrackingFile> files;

    const std::string sql = "SELECT file_id, file_path, last_checksum,
is_missing FROM tracking_files;";
    sqlite3_stmt* stmt;
    if (sqlite3_prepare_v2(db, sql.c_str(), -1, &stmt, nullptr) != SQLITE_OK)
{
        throw std::runtime_error("Ошибка подготовки запроса к
tracking_files");
    }

    while (sqlite3_step(stmt) == SQLITE_ROW) {
        TrackingFile file;
        file.fileId = reinterpret_cast<const char*>(sqlite3_column_text(stmt,
0));
        file.filePath = reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 1));
        file.lastChecksum = reinterpret_cast<const
char*>(sqlite3_column_text(stmt, 2));
        file.isMissing = sqlite3_column_int(stmt, 3) != 0;

        // Загружаем изменения для файла
```

```
        const std::string changesSql = "SELECT timestamp, change_type,
checksum, saved_version_id, user, additional_info FROM file_changes WHERE
file_id = ? ORDER BY timestamp ASC;";
        sqlite3_stmt* changesStmt;
        sqlite3_prepare_v2(db, changesSql.c_str(), -1, &changesStmt,
nullptr);
        sqlite3_bind_text(changesStmt, 1, file.fileId.c_str(), -1,
SQLITE_TRANSIENT);

        while (sqlite3_step(changesStmt) == SQLITE_ROW) {
            FileChange change;
            change.timestamp = fromIsoString(reinterpret_cast<const
char*>(sqlite3_column_text(changesStmt, 0)));
            change.changeType = reinterpret_cast<const
char*>(sqlite3_column_text(changesStmt, 1));
            change.checksum = reinterpret_cast<const
char*>(sqlite3_column_text(changesStmt, 2));
            change.savedVersionId = reinterpret_cast<const
char*>(sqlite3_column_text(changesStmt, 3));
            change.user = reinterpret_cast<const
char*>(sqlite3_column_text(changesStmt, 4));
            change.additionalInfo = reinterpret_cast<const
char*>(sqlite3_column_text(changesStmt, 5));
            file.history.changes.push_back(change);
        }

        sqlite3_finalize(changesStmt);
        files.push_back(file);
    }

    sqlite3_finalize(stmt);
    return files;
}
```

Листинг Б.7 – Файл конфигураций

```
{
  "monitoring": {
    "groups": [
      {
        "id": "test_group_1",
        "description": "Тестовая группа 1 — простые файлы",
        "paths": [
          {
            "path": "/home/maksd/course_wrok/test_storage/group1",
            "recursive": false
          }
        ],
        "events": ["MODIFY", "DELETE"],
        "checksum": {
          "enabled": true,
          "algorithm": "sha256"
        }
      },
      {
        "id": "test_group_2",
        "description": "Тестовая группа 2 — с рекурсивным обходом",
        "paths": [
          {
            "path": "/home/maksd/course_wrok/test_storage/group2",
            "recursive": true
          }
        ],
        "events": ["MODIFY", "CREATE", "DELETE"],
```

```
        "checksum": {
          "enabled": true,
          "algorithm": "sha256"
        }
      },
      {
        "id": "test_group_3",
        "description": "Тестовая группа 3 — без контрольных сумм",
        "paths": [
          {
            "path": "/home/maksd/course_wrok/test_storage/group3",
            "recursive": false
          }
        ],
        "events": ["MODIFY", "MOVE"],
        "checksum": {
          "enabled": false
        }
      }
    ]
  }
}
```

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

ПРИЛОЖЕНИЕ Г
(обязательное)
Блок схема алгоритма

ПРИЛОЖЕНИЕ Д
(обязательное)
Графический интерфейс пользователя

ПРИЛОЖЕНИЕ Е
(обязательное)
Ведомость документов