МОСКОВСКИЙ АВИАЦИОННЫЙ ИНСТИТУТ
(НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ)

Институт №8 «Компьютерные науки и прикладная математика»

Кафедра №806 «Вычислительная математика и программирование»

Курсовой проект

по курсу «Системное программирование»

Разработка алгоритмов системы хранения и управления данными на основе динамических структур данных

Выполнил: Колесник Д.С.

Группа: 8О-213Б

Преподаватель: А.М. Романенков

Москва, 2025

Содержание

Введение

Развитие современных вычислительных систем сопровождается ростом требований к программному обеспечению системного уровня. Ключевыми аспектами становятся производительность, устойчивость к сбоям и безопасность исполнения в условиях многопоточности, высокой нагрузки и потенциальных аппаратных сбоев. В связи с этим особое значение приобретают программные решения, обеспечивающие эффективное взаимодействие с аппаратными ресурсами, операционными системами и сетевыми компонентами.

Системное программирование играет фундаментальную роль в создании программных компонентов, управляющих памятью, потоками, сетевыми соединениями и механизмами логгирования. Актуальность темы обусловлена необходимостью разработки отказоустойчивых и производительных компонентов, предназначенных для работы в критичных к ресурсам и времени отклика системах: от серверных приложений и встроенных платформ до систем управления базами данных и телекоммуникационных решений.

Целью данной работы является разработка модульной библиотеки системного уровня на языке программирования C++20, включающей компоненты клиентского логгера, серверной архитектуры, аллокаторов памяти и структур данных на основе B-деревьев. Библиотека ориентирована на высокую производительность, надёжность и соответствие принципам строгого управления ресурсами и модульности.

В качестве методологической базы использовались принципы объектно-ориентированного и модульного программирования, современные механизмы C++20 (RAII, семантика перемещения, умные указатели), а также средства тестирования, профилирования и отладки программных компонентов.

Практическая значимость работы заключается в возможности интеграции полученной библиотеки в состав прикладных и системных решений, требующих высокой надёжности и производительности. Разработанное программное обеспечение может быть использовано в реальных проектах, а также в образовательных целях для демонстрации современных подходов к разработке системного ПО.

Теоретическая часть

Современные вычислительные системы требуют от алгоритмов и структур данных высокой производительности, надежности и способности к масштабированию. Основу таких решений составляют базовые принципы информатики: оптимизация вычислительных процессов, эффективное управление ресурсами и создание абстракций, способных адаптироваться к изменениям. В рамках данной работы рассматриваются основные задачи, иллюстрирующие применение классических алгоритмов и шаблонов проектирования с использованием языка программирования C++ (в соответствии со стандартом C++20 и выше).

Серверный логгер

Понятие и назначение

Серверный логгер представляет собой специализированный программный компонент, предназначенный для централизованного сбора и обработки событий, возникающих в серверных приложениях и распределённых системах. Он играет ключевую роль в обеспечении надёжной работы таких систем, выполняя следующие основные функции:

- Безопасность и аудит: Логгер отслеживает действия пользователей и системные события, обеспечивая их прозрачность. Это важно как для соответствия требованиям безопасности, так и для своевременного обнаружения потенциально опасной активности.
- Диагностика и отладка: Логгер регистрирует события и сопутствующую информацию, позволяя разработчикам и системным администраторам выявлять и анализировать ошибки, а также устранять неполадки в серверной инфраструктуре.
- Анализ сбоев: При возникновении критических ошибок логгер предоставляет детализированные журналы, содержащие хронологию событий, что помогает восстановить цепочку причин, устранить неисправности и предотвратить их повторное возникновение.

Серверный логгер особенно важен в распределённых системах, где централизованное логирование позволяет агрегировать данные с множества узлов. Это обеспечивает целостное представление о работе системы и поддерживает предиктивную аналитику для выявления аномалий, прогнозирования сбоев и оптимизации производительности.

Способы логирования

Серверный логгер поддерживает различные каналы вывода логов, каждый из которых адаптирован под конкретные сценарии использования и этапы эксплуатации системы:

- Сетевые каналы
- Вывод в стандартные потоки (stdout/stderr)
- Интеграция с системными службами
- Логирование в файлы

Выбор канала логирования зависит от требований к производительности, надёжности и удобству анализа. Например, в системах реального времени предпочтение отдаётся памяти, в корпоративных средах — централизованным сетевым хранилищам с поддержкой отказоустойчивости и аналитики.

Уровни серьезности

Для управления потоком логов и фильтрации сообщений в зависимости от их важности серверный логгер использует уровни серьезности (severity levels). Это позволяет разработчикам и администраторам сосредотачиваться на наиболее значимых событиях. Основные уровни включают:

- Trace – используются в основном для диагностики;
- Debug – отладочная информация, локализация ошибок;
- Info – сообщения о нормальной работе, общий мониторинг;
- Warning – потенциальные проблемы, которые не нарушают работу системы, но требуют внимания;
- Error – ошибки в отдельных компонентах системы;
- Fatal/Critical – критические сбои всей системы.

Каждому каналу вывода (консоль, файл, сеть) можно назначить минимальный уровень серьезности. Например, консоли – warning, файлам все уровни логов, а сетевым каналам info

Структурированное логирование

Серверный логгер реализует поддержку структурированного логирования, при котором информация о событиях сохраняется в формате, удобном для автоматической обработки, например в JSON. Такой подход позволяет включать в каждую запись дополнительные структурированные данные: временные метки,

уровень важности сообщения, идентификаторы процессов, запросов или пользователей, а также системные метаданные — например, IP-адрес сервера или версию используемого программного обеспечения. Преимущества структурированного логирования включают: автоматизацию анализа, трассировку запросов, интеграцию с аналитикой и так далее.

Структурированное логирование особенно важно в микросервисных архитектурах, где централизованный сбор и анализ логов с множества узлов позволяет быстро выявлять и устранять проблемы. Например, сообщения в формате JSON могут содержать поля, такие как pid, severity, message, timestamp, и дополнительные метаданные, что делает их удобными для обработки в реальном времени и долгосрочного хранения.

Global heap allocator

Понятие и назначение

Аллокатор глобальной кучи (allocator_global_heap) представляет собой программный компонент, реализующий управление динамической памятью на основе глобального механизма выделения, предоставляемого операционной системой через стандартные средства языка C++ — операторы new и delete. Данный аллокатор реализует интерфейс std::pmr::memory_resource, что обеспечивает унифицированный подход к управлению памятью в приложениях, предъявляющих высокие требования к надёжности и эффективности.

Ключевые функции данного модуля включают:

- Диагностика и мониторинг: Интеграция с логгером позволяет фиксировать все операции выделения и освобождения памяти, что упрощает отладку, выявление утечек памяти и анализ производительности.
- Управление памятью: Аллокатор отвечает за выделение и освобождение блоков памяти заданного размера, обеспечивая их корректное использование в рамках приложения.
- Обработка ошибок: Аллокатор предоставляет механизмы обработки исключений, таких как std::bad_alloc, для обеспечения устойчивости системы при нехватке памяти или других сбоях.
- Совместимость и стандартизация: Реализация на основе std::pmr::memory_resource обеспечивает совместимость с полиморфными аллокаторами C++ и позволяет использовать модуль в различных контекстах, включая контейнеры стандартной библиотеки.

Способы выделения и освобождения памяти

Метод do_allocate_sm вызывает ::operator new для резервирования блока памяти заданного размера. При успешном выделении возвращается указатель на выделенную область, а информация об операции фиксируется в логе. В случае нехватки памяти или других ошибок генерируется соответствующее исключение с записью в лог.

Освобождение памяти: Метод do_deallocate_sm использует ::operator delete для освобождения памяти по указанному адресу. Если передан нулевой указатель, операция игнорируется с соответствующей записью в лог, что предотвращает некорректное поведение.

Каждая операция сопровождается логированием, что обеспечивает прозрачность и возможность отслеживания всех действий с памятью, включая адреса выделенных блоков и их размеры.

Уровни логирования

Аллокатор интегрирован с системой логирования, которая фиксирует события на различных уровнях серьезности, чтобы обеспечить эффективный мониторинг и отладку:

- Trace – создание, копирование, перемещение или уничтожение объектов аллокатора;
- Debug – операции по выделению или освобождению памяти;
- Error – регистрация ошибок (например, нехватка памяти).

Интеграция и расширяемость

Аллокатор глобальной кучи спроектирован с учётом современных стандартов C++20, что обеспечивает его совместимость с другими компонентами стандартной библиотеки, использующими полиморфные аллокаторы (std::pmr). Основные особенности включают:

- RAII и управление ресурсами: Аллокатор использует принципы RAII для корректного управления логгером и другими ресурсами, предотвращая утечки памяти.
- Логирование: Интеграция с логгером позволяет фиксировать все операции, что упрощает диагностику и мониторинг. Логгер передаётся в конструкторе и используется для записи событий на всех этапах работы аллокатора.

- Простота и надёжность: Использование глобальной кучи через стандартные операторы new и delete минимизирует накладные расходы и обеспечивает предсказуемое поведение, что особенно важно для систем с высокими требованиями к производительности.
- Поддержка копирования и перемещения: Реализованы конструкторы копирования и перемещения, а также операторы присваивания, что делает аллокатор безопасным для использования в различных сценариях, включая контейнеры и сложные структуры данных.

Boundary tags allocator

Понятие и назначение

allocator_boundary_tags — это специализированная реализация средства управления памятью, созданная на C++20 и ориентированная на использование схемы граничных меток (boundary tags). Данный метод позволяет эффективно отслеживать и перераспределять блоки памяти, минимизируя фрагментацию и ускоряя операции выделения и освобождения.

Этот аллокатор реализует интерфейс std::pmr::memory_resource, что обеспечивает его интеграцию с полиморфной системой управления памятью, предложенной стандартной библиотекой C++. Он функционирует в пределах заранее выделенного непрерывного участка памяти, полученного от родительского аллокатора, и предназначен для систем, где критична предсказуемость поведения и контроль за использованием ресурсов.

Внутренние механизмы аллокатора поддерживают три классические стратегии размещения: first fit (первый подходящий блок), best fit (лучший по размеру), wors fit (самый крупный блок среди доступных).

Аллокатор управляет фиксированным пулом памяти, разделяя его на занятые и свободные блоки, с использованием метаданных (границ) для отслеживания их состояния и связей. Поддержка различных стратегий выделения памяти позволяет оптимизировать использование ресурсов в зависимости от требований приложения, минимизируя фрагментацию.

Интеграция с логгером фиксирует все операции выделения, освобождения и изменения состояния памяти, что упрощает отладку, анализ производительности и выявление ошибок. Ну а использование мьютекса обеспечивает безопасное выполнение операций в многопоточной среде.

Наконец, аллокатор включает механизмы проверки входных данных и обработки исключений, таких как std::bad_alloc или попытки освобождения некорректных указателей, для повышения надёжности.

Способы выделения и освобождения памяти

Аллокатор использует технику граничных меток, при которой каждый блок памяти (занятый или свободный) содержит метаданные, включая размер блока и указатели на соседние блоки (предыдущий и следующий). Это позволяет эффективно управлять памятью и поддерживать различные стратегии размещения:

- Выделение памяти:
    - Аллокатор поддерживает три стратегии:
        - First Fit: Выбирает первый свободный блок, размер которого достаточен для запрошенного объёма.
        - Best Fit: Выбирает свободный блок, наиболее близкий по размеру к запрошенному, минимизируя фрагментацию.
        - Worst Fit: Выбирает самый большой свободный блок, что может быть полезно для распределения оставшегося пространства.
    - Если подходящий блок найден, он разделяется (при необходимости) на занятый блок и оставшийся свободный блок. Метаданные обновляются, включая указатели на соседние блоки и родительский пул.
    - Логирование фиксирует размер выделяемого блока, стратегию и оставшееся свободное пространство.
    - При отсутствии подходящего блока выбрасывается исключение std::bad_alloc.
- Освобождение памяти:
    - Проверяется, принадлежит ли указатель пулу памяти аллокатора. Если указатель недействителен или нулевой, выбрасывается исключение std::invalid_argument или std::logic_error.
    - Освобождённый блок объединяется с соседними свободными блоками (если они есть), чтобы минимизировать фрагментацию.
    - Метаданные соседних блоков обновляются, а информация об освобождении записывается в лог.
- Проверка эквивалентности (do_is_equal):

    - Метод проверяет, является ли другой аллокатор экземпляром allocator_boundary_tags, что важно для совместимости с контейнерами и другими компонентами.
- Итерация по блокам:
    - Аллокатор предоставляет итератор (boundary_iterator), который позволяет обходить блоки памяти, возвращая информацию о их размере и состоянии (занятый или свободный). Это полезно для анализа структуры памяти и отладки.

Уровни логирования

Аллокатор интегрирован с системой логирования для мониторинга операций и диагностики проблем:

- Trace: Фиксирует события жизненного цикла аллокатора, такие как создание, перемещение и уничтожение объектов.
- Debug: Регистрирует детали операций выделения и освобождения, включая адреса блоков, их размеры и структуру памяти (например, список блоков).
- Information: Сообщает о состоянии памяти после операций, например, об общем объёме свободной памяти.
- Warning: Используется при корректировке размера выделяемого блока, если доступное пространство меньше запрошенного.
- Error: Фиксирует ошибки, такие как попытка выделения памяти без подходящего блока, освобождение некорректного указателя или доступ к чужому блоку памяти.

Интеграция и расширяемость

Аллокатор с граничными метками разработан с учётом современных стандартов C++20, что обеспечивает его гибкость и совместимость:

- RAII и управление ресурсами: Аллокатор использует принципы RAII для управления мьютексом, логгером и родительским аллокатором, предотвращая утечки ресурсов.
- Потокобезопасность: Мьютекс гарантирует безопасное выполнение операций выделения и освобождения в многопоточных приложениях.
- Поддержка стратегий размещения: Возможность выбора между first fit, best fit и worst fit делает аллокатор адаптивным к различным сценариям использования.

- Интеграция с логгером: Логгер фиксирует все ключевые события, упрощая отладку и мониторинг. Логгер передаётся через конструктор и используется для записи событий на всех этапах работы.
- Итераторы: Поддержка итерации по блокам памяти позволяет анализировать структуру пула, что полезно для тестирования и оптимизации.
- Совместимость: Реализация интерфейса std::pmr::memory_resource позволяет использовать аллокатор в стандартных контейнерах и других компонентах, поддерживающих полиморфные аллокаторы.

Аллокатор с граничными метками идеально подходит для систем, где требуется точное управление ограниченным пулом памяти, таких как серверные приложения, встроенные системы или сложные структуры данных (например, B-деревья).

Buddy system allocator

Понятие и назначение

allocator_buddies_system — это специализированный модуль управления памятью, реализованный в C++20, использующий алгоритм "buddy system" (система двойников) для эффективного выделения и освобождения памяти. Он реализует интерфейс std::pmr::memory_resource, обеспечивая совместимость с полиморфными аллокаторами стандартной библиотеки C++. Аллокатор управляет фиксированным пулом памяти, разделяя его на блоки, размеры которых являются степенями двойки, что минимизирует фрагментацию и упрощает объединение свободных блоков. Поддерживаются представленные выше три стратегии размещения. Основные функции аллокатора включают:

- Потокобезопасность: Использование мьютекса обеспечивает безопасное выполнение операций в многопоточной среде.
- Управление памятью: Аллокатор разделяет память на блоки, размеры которых являются степенями двойки, и поддерживает эффективное выделение и освобождение памяти с помощью механизма "двойников".
- Гибкость размещения: Различные стратегии выделения позволяют оптимизировать использование памяти в зависимости от требований приложения, балансируя между скоростью и фрагментацией.
- Обработка ошибок: Аллокатор включает проверки входных данных и обработку исключений, таких как std::bad_alloc или попытки освобождения некорректных указателей, для обеспечения надёжности.

- Диагностика и мониторинг: Интеграция с логгером фиксирует все операции выделения и освобождения, а также изменения состояния памяти, что упрощает отладку и анализ производительности.

Аллокатор двойников особенно подходит для систем, где требуется эффективное управление памятью с минимальной фрагментацией, например, в серверных приложениях, встроенных системах или структурах данных, таких как B-деревья, где важна предсказуемость и производительность.

Способы выделения и освобождения памяти

Аллокатор использует алгоритм двойников, при котором пул памяти делится на блоки, размеры которых являются степенями двойки. Каждый блок имеет "брата" — соседний блок того же размера, который можно объединить при освобождении для создания блока большего размера. Это обеспечивает эффективное управление памятью и минимизацию фрагментации:

- Выделение памяти:
  - Аллокатор поддерживает три стратегии:
    - First Fit: Выбирает первый свободный блок, размер которого не меньше запрошенного.
    - Best Fit: Выбирает свободный блок, наиболее близкий по размеру к запрошенному, минимизируя фрагментацию.
    - Worst Fit: Выбирает самый большой свободный блок, что может быть полезно для распределения оставшегося пространства.
  - Если подходящий блок слишком велик, он рекурсивно делится на два блока меньшего размера (степени двойки), пока не будет достигнут подходящий размер. Метаданные блока обновляются, включая флаг занятости и указатель на родительский пул.
  - Логирование фиксирует размер выделяемого блока, стратегию и оставшееся свободное пространство. При отсутствии подходящего блока выбрасывается исключение std::bad_alloc.
- Освобождение памяти:
  - Проверяется, принадлежит ли указатель пулу памяти аллокатора. Если указатель недействителен или нулевой, выбрасывается исключение std::logic_error.
  - Освобождённый блок помечается как свободный, после чего проверяется его брат. Если брат также свободен и имеет тот же размер, блоки объединяются в блок большего размера. Этот процесс

повторяется, пока не будет достигнут максимальный возможный размер блока.
    - Логирование фиксирует адрес освобождаемого блока, его размер и состояние памяти после операции.
- Проверка эквивалентности:
    - Метод проверяет, является ли другой аллокатор экземпляром allocator_buddies_system, что важно для совместимости с контейнерами и другими компонентами.
- Итерация по блокам:
    - Аллокатор предоставляет итератор (buddy_iterator), который позволяет обходить блоки памяти, возвращая информацию о их размере и состоянии (занятый или свободный). Это полезно для анализа структуры памяти и отладки.

Уровни логирования

Аллокатор интегрирован с системой логирования для мониторинга операций и диагностики проблем:

- Trace: Фиксирует события жизненного цикла аллокатора, такие как создание, перемещение и уничтожение объектов.
- Debug: Регистрирует детали операций выделения и освобождения, включая адреса блоков, их размеры и структуру памяти (например, список блоков).
- Information: Сообщает о состоянии памяти после операций, например, об общем объёме свободной памяти.
- Warning: Используется при корректировке размера выделяемого блока, если доступное пространство меньше запрошенного.
- Error: Фиксирует ошибки, такие как попытка выделения памяти без подходящего блока или освобождение некорректного указателя.

Интеграция и расширяемость

Аллокатор двойников разработан с учётом стандартов C++20, что обеспечивает его гибкость и совместимость:

- RAII и управление ресурсами: Аллокатор использует принципы RAII для управления мьютексом, логгером и родительским аллокатором, предотвращая утечки ресурсов.
- Потокобезопасность: Мьютекс гарантирует безопасное выполнение операций выделения и освобождения в многопоточных приложениях.

- Поддержка стратегий размещения: Возможность выбора между first fit, best fit и worst fit делает аллокатор адаптивным к различным сценариям использования.
- Интеграция с логгером: Логгер фиксирует все ключевые события, упрощая диагностику и мониторинг. Логгер передаётся через конструктор и используется для записи событий на всех этапах работы.
- Итераторы: Поддержка итерации по блокам памяти позволяет анализировать структуру пула, что полезно для тестирования и оптимизации.
- Совместимость: Реализация интерфейса std::pmr::memory_resource позволяет использовать аллокатор в стандартных контейнерах и других компонентах, поддерживающих полиморфные аллокаторы.

Аллокатор двойников идеально подходит для систем, где требуется эффективное управление памятью с минимальной фрагментацией, таких как серверные приложения, встроенные системы или сложные структуры данных (например, B-деревья). Он также может служить основой для создания специализированных аллокаторов с дополнительными функциями, такими как оптимизация под специфические паттерны использования памяти или кэширование.

B-дерево

Понятие о B-дереве, структура

B-дерево представляет собой сбалансированную древовидную структуру данных, разработанную специально для работы с большими объемами информации в условиях, где критически важна эффективность операций ввода-вывода — например, при взаимодействии с жёсткими дисками в базах данных и файловых системах. Эта структура была разработана Рудольфом Байером и Эдвардом МакКрейтом в 1972 году как расширение концепции бинарного дерева поиска, позволяющее хранить множество ключей в каждом узле. Такая организация обеспечивает равномерное распределение данных и сводит к минимуму глубину дерева, что в свою очередь сокращает количество обращений к медленной внешней памяти при поиске, вставке и удалении элементов.

Основные характеристики B-дерева:

- Многоуровневая структура: B-дерево состоит из узлов, каждый из которых может содержать несколько ключей и ссылок на дочерние узлы.

- Параметр t (порядок дерева): Определяет минимальное и максимальное количество ключей в узле. Для порядка t:
    - Каждый узел (кроме корня) содержит от t-1 до 2t-1 ключей.
    - Каждый узел имеет от t до 2t дочерних указателей.
- Сбалансированность: Все листовые узлы находятся на одном уровне, что обеспечивает логарифмическую высоту дерева.
- Упорядоченность: Ключи в узле отсортированы, и для каждого ключа k в узле все ключи в левом поддереве меньше k, а в правом — больше.

Структура узла B-дерева:

- Массив ключей (пары ключ-значение).
- Массив указателей на дочерние узлы.
- Флаг, указывающий, является ли узел листом (не имеет дочерних узлов).
- Размер узла (количество ключей).

B-дерево оптимизировано для систем с большими данными, так как оно минимизирует количество дисковых операций за счет хранения множества ключей в одном узле, что позволяет эффективно использовать блоки памяти.

Асимптотическая сложность операций:

- Поиск, вставка, удаление:
    - Средний случай: O(log n)
    - Худший случай: O(log n)
    - Лучший случай: O(log n) (для поиска корневого узла — O(1))

Операции над B-деревом

Операция поиска

Поиск в B-дереве начинается с корня и использует бинарный поиск (или линейный поиск) внутри узла для определения, в какое поддерево следует спуститься:

1. В текущем узле ищется ключ k с помощью бинарного поиска.
2. Если ключ найден, возвращается соответствующее значение.
3. Если ключ не найден, переход осуществляется в соответствующее поддерево (между двумя ключами или в крайнее поддерево).
4. Если достигнут лист и ключ не найден, возвращается "не найдено".

Асимптотическая сложность: O(log n), так как высота дерева логарифмическая.

Операция вставки

Вставка в B-дерево выполняется следующим образом:

1. Находится лист, в который следует вставить новый ключ, с помощью алгоритма поиска.
2. Ключ добавляется в лист в отсортированном порядке.
3. Если после вставки узел превышает максимальное количество ключей (2t-1), он разделяется:
    - Средний ключ (на позиции t) поднимается в родительский узел.
    - Узел разделяется на два новых узла, каждый из которых содержит t-1 ключей.
    - Указатели на дочерние узлы также распределяются между новыми узлами.
4. Если родительский узел становится переполненным, процесс разделения повторяется рекурсивно до корня.
5. Если корень разделяется, создается новый корень, увеличивая высоту дерева.

Асимптотическая сложность: O(log n), так как количество операций пропорционально высоте дерева.

Операция удаления

Удаление ключа из B-дерева сложнее, так как требует поддержания минимального числа ключей в узлах:

- Находится узел, содержащий ключ, с помощью поиска.
- Случай 1: Удаление из листа:
    - Ключ просто удаляется, если после удаления в узле остается не менее t-1 ключей.
    - Если узел становится "недостаточным" (менее t-1 ключей), выполняется ребалансировка:
        - Заимствование: Если соседний узел имеет более t-1 ключей, ключ заимствуется через родительский узел.
        - Слияние: Если заимствование невозможно, узел сливается с соседом через родительский ключ.
- Случай 2: Удаление из внутреннего узла:
    - Находится предшественник (максимальный ключ в левом поддереве) или преемник (минимальный ключ в правом поддереве).

    - Значение предшественника/преемника копируется в удаляемый узел, а затем предшественник/преемник удаляется рекурсивно (это всегда лист).
- Если корень становится пустым после слияния, его дочерний узел становится новым корнем, уменьшая высоту дерева.

Асимптотическая сложность: O(log n), так как все операции ограничены высотой дерева.

B-дерево на диске

Понятие о B-дереве, структура

Дисковое B-дерево представляет собой модифицированную версию классического B-дерева, специально приспособленную для хранения данных на внешних носителях, таких как жёсткие диски или твердотельные накопители (SSD). Ключевая особенность заключается в том, что каждый узел дерева размещается в отдельной области дисковой памяти (например, в блоках или страницах файловой системы), а все операции над деревом — чтение, вставка, удаление — выполняются с учётом специфики медленного дискового ввода-вывода.

Такая архитектура позволяет существенно снизить количество обращений к внешней памяти за счёт высокой степени ветвления дерева, что минимизирует глубину и, соответственно, число загрузок узлов в оперативную память. Благодаря этим качествам дисковые B-деревья стали стандартным решением для индексирования в системах управления базами данных, а также в реализации иерархий в современных файловых системах.

Особенности B-дерева на диске:

- Хранение на диске: Узлы хранятся в двух файлах:
    - Файл индекса (.tree): содержит метаданные узлов (размер, флаг листа, позиция на диске, указатели на дочерние узлы) и ссылки на ключи.
    - Файл данных (.data): хранит пары ключ-значение, сериализованные в бинарном формате.
- Сериализация и десериализация: Ключи и значения должны поддерживать сериализацию (запись в поток) и десериализацию (чтение из потока) для сохранения и восстановления данных.

- Оптимизация ввода-вывода: Размер узла (блока) обычно соответствует размеру дискового блока (например, 4 КБ или 8 КБ), чтобы минимизировать количество операций чтения/записи.
- Управление памятью: Используется специальный аллокатор (disk_allocator), который управляет выделением и освобождением памяти для узлов.
- Обработка ошибок: Реализованы исключения для обработки ошибок файлового ввода-вывода, некорректных узлов и поврежденных данных.

Структура узла B-дерева на диске:

- size: Количество ключей в узле.
- is_leaf: Флаг, указывающий, является ли узел листом.
- position_in_disk: Уникальный идентификатор узла (номер блока).
- keys: Вектор пар ключ-значение (сериализуемых типов tkey, tvalue).
- pointers: Вектор позиций дочерних узлов на диске.

Метаданные файла:

- _count_of_node: Общее количество узлов в дереве.
- _position_root: Позиция корневого узла на диске.
- _node_block_size: Размер блока для хранения узла (динамически адаптируется).

Операции над B-деревом на диске

Операция поиска

Поиск выполняется аналогично B-дереву в памяти, но с учетом дисковых операций:

1. Читается корневой узел с диска (disk_read).
2. В узле выполняется бинарный поиск для нахождения ключа или определения следующего поддерева.
3. Если ключ найден, возвращается значение.
4. Если узел не листовой, читается дочерний узел по указателю (позиция на диске).
5. Если ключ не найден в листе, возвращается std::nullopt.

Асимптотическая сложность: O(log n)

Операция вставки

Вставка в B-дерево на диске включает следующие шаги:

1. Выполняется поиск пути к листу, куда нужно вставить ключ (find_path), с сохранением пути в стеке.
2. Читается целевой листовой узел с диска.
3. Ключ вставляется в массив ключей в отсортированном порядке (insert_array).
4. Узел записывается обратно на диск (disk_write).
5. Если узел переполнен (более 2t-1 ключей), выполняется разделение узла (split_node):
    - Создается новый узел с t-1 ключами из правой части.
    - Средний ключ поднимается в родительский узел.
    - Указатели распределяются между узлами.
    - Новый узел записывается на диск с новым position_in_disk.
6. Если родительский узел переполнен, процесс разделения повторяется.
7. Обновляются метаданные файла (write_metadata).

Асимптотическая сложность: O(log n), с учетом дисковых операций.

Операция удаления

Удаление в B-дереве на диске включает:

1. Поиск узла с ключом (find_path).
2. Если ключ находится в листе:
    - Ключ удаляется (remove_array), узел записывается на диск.
    - Если узел становится недостаточным (менее t-1 ключей), выполняется ребалансировка (rebalance_node):
        - Заимствование: Ключ берется из соседнего узла через родительский.
        - Слияние: Узел объединяется с соседом, родительский ключ опускается.
3. Если ключ находится во внутреннем узле:
    - Находится предшественник (find_max_element в левом поддереве).
    - Значение предшественника копируется в узел, предшественник удаляется.
4. Ребалансировка выполняется по всему пути до корня.
5. Если корень становится пустым, обновляется _position_root.
6. Метаданные записываются на диск.

Асимптотическая сложность: O(log n), с учетом дисковых операций.

Операция итерации

B-дерево на диске поддерживает итерацию по ключам с помощью btree_disk_const_iterator:

- begin(): Находит минимальный ключ, читая узлы от корня к левому листу.
- end(): Пустой итератор, указывающий на конец дерева.
- operator++: Переходит к следующему ключу, спускаясь в правое поддерево или поднимаясь к родителю.
- operator--: Переходит к предыдущему ключу, используя find_max_element для правого поддерева.

Асимптотическая сложность перехода: O(log n) в худшем случае из-за дисковых операций.

Система управления пользователями и санкциями

Понятие и назначение

Система управления пользователями и командами — это консольное приложение, предназначенное для регистрации и аутентификации пользователей, выполнения ограниченного набора команд и управления доступом через механизм санкций. Основные задачи системы:

1. Контроль активности: Ограничение числа выполняемых команд за сессию с помощью санкций, что позволяет регулировать использование ресурсов.
2. Персистентность данных: Сохранение информации о пользователях и санкциях между запусками программы.
3. Регистрация и аутентификация: Создание учетных записей с уникальным логином и PIN-кодом, проверка подлинности для доступа к функционалу.
4. Безопасность: Защита учетных данных через хэширование PIN-кодов и подтверждение изменений санкций кодом.
5. Обработка временных запросов: Предоставление информации о текущем времени, дате и вычисление разницы между заданным и текущим временем.

Способы реализации

Система реализована через следующие подходы:

1. Консольный интерфейс:
    - Взаимодействие через стандартные потоки ввода (scanf) и вывода (printf).

    - Поддерживает меню для регистрации, входа, выхода и выполнения команд (Time, Date, Howmuch, Sanctions, Logout).
    - Простой и удобный для локального использования, но не предназначен для сетевых сценариев.
2. Файловое хранение:
    - Файл users.txt: Сохраняет логины и хэши PIN-кодов.
    - Файл sanctions.txt: Хранит данные о лимитах команд и их использовании.
    - Обеспечивает персистентность, но не поддерживает ротацию или масштабируемость.
3. Локальная память:
    - Данные хранятся в массивах users и sanctions с фиксированным размером (100 записей).
    - Быстрый доступ, но ограниченная масштабируемость.
4. Отсутствие сетевой интеграции:
    - Работает локально, без поддержки сетевых протоколов или внешних систем.

Структурированное представление данных

1. Текущее состояние:
    - Файлы users.txt и sanctions.txt используют простой текстовый формат (поля разделены пробелами), читаемый через fscanf.
    - Консольные сообщения — неструктурированный текст, затрудняющий автоматическую обработку.
    - Данные в памяти организованы в структуры User (логин, хэш PIN) и Sanction (имя, лимит, использование).

Система обработки файлов

Понятие и назначение

Система обработки файлов и команд — это консольное приложение, предназначенное для выполнения операций над файлами, таких как побитовое XOR, поиск по маске, копирование файлов и поиск текста. Основные задачи системы:

1. Обработка файлов: Выполнение операций побитового XOR и поиска значений, соответствующих заданной маске, для анализа содержимого файлов.

2. Копирование файлов: Создание нескольких копий входного файла с использованием процессов для параллельного выполнения.
3. Поиск текста: Обнаружение строк, содержащих заданный текстовый фрагмент, с указанием номеров строк.
4. Параллельная обработка: Использование системных вызовов fork для распределения задач между процессами, что повышает производительность.
5. Обработка ошибок: Обеспечение информативных сообщений об ошибках и корректное управление ресурсами (файлами, памятью, процессами).

Система подходит для задач автоматизации обработки файлов, анализа данных и выполнения массовых операций в локальной среде.

Способы реализации

Система реализована через следующие подходы:

1. Консольный интерфейс:
    - Принимает аргументы командной строки (имена файлов и команду: xor, mask, copy, find) для определения операции.
    - Выводит результаты в стандартный поток (printf) и ошибки в поток ошибок (fprintf(stderr)).
    - Простой и гибкий для локального использования, но ограничен отсутствием интерактивного интерфейса.
2. Файловый ввод-вывод:
    - Чтение и запись файлов в бинарном (rb, wb) или текстовом (r) режиме с использованием функций fopen, fread, fwrite, fgetc.
    - Поддерживает обработку файлов в блоках фиксированного размера (CHUNK_SIZE) для эффективности.
    - Не реализует долговременное хранение результатов, фокусируясь на одноразовой обработке.
3. Параллельная обработка:
    - Использует системный вызов fork для создания дочерних процессов при выполнении операций копирования (copy) и поиска текста (find).
    - Управляет процессами через waitpid для синхронизации и проверки успешности выполнения.
4. Динамическое управление памятью:
    - Хранит промежуточные данные (результаты поиска, позиции совпадений) в динамически выделяемых массивах с автоматическим расширением (realloc).

    - Обеспечивает освобождение ресурсов (free, fclose) для предотвращения утечек памяти.
5. Отсутствие сетевой интеграции:
    - Работает локально, без поддержки сетевых протоколов или интеграции с внешними системами.

Структурированное представление данных

1. Текущее состояние:
    - Данные в файлах обрабатываются как бинарные потоки (для xor, mask) или текст (для find), без сохранения в структурированном формате.
    - Выходные данные представляются в текстовом виде через консоль, например: списки позиций совпадений (mask), номера строк (find) или байты результата XOR.
    - В памяти используются структуры MatchResult (для позиций совпадений по маске) и TextMatchResult (для номеров строк с совпадениями), содержащие счетчик, массив позиций и емкость.

Система отображения содержимого директорий

Понятие и назначение

Данная система представляет собой консольное приложение, реализующее функциональность, аналогичную утилите ls -l в UNIX-подобных системах. Основные задачи:

1. Вывод подробной информации о файлах и каталогах: отображение типа файла, прав доступа, владельца, группы, размера, даты последнего изменения и имени файла.
2. Работа с несколькими каталогами: возможность передать несколько путей в аргументах командной строки.
3. Поддержка различных типов файлов: включая обычные файлы, каталоги, символьные ссылки, устройства, FIFO и сокеты.
4. Безопасная работа с файловой системой: использование безопасных системных вызовов (fstatat, dirfd) для обхода проблем с символическими ссылками и защищённого доступа к файлам.

Способы реализации

Система реализована через следующие подходы:

1. Консольный интерфейс:
    - Программа вызывается из командной строки с указанием одного или нескольких путей (каталогов).
    - При отсутствии аргументов — обрабатывается текущий каталог (".").
    - Результаты выводятся в стандартный поток вывода (printf), ошибки — в стандартный поток ошибок (perror).
2. Обработка каталогов и файлов:
    - Используется opendir, readdir, closedir для обхода содержимого каталогов.
    - Для получения информации о файлах применяется fstatat, что позволяет избежать ошибок при работе с символическими ссылками и не требует изменения текущего каталога.
    - Вывод информации формируется вручную в стиле ls -l.
3. Определение типа файла:
    - Тип файла (обычный, каталог, ссылка и т.д.) определяется через макросы S_ISREG, S_ISDIR и др.
    - Первый символ строки прав (-, d, l, c, b, p, s) отображает тип объекта.
4. Отображение прав доступа:
    - Права доступа отображаются по шаблону rwxrwxrwx, формируемому побитовой проверкой флагов S_IRUSR, S_IWUSR и т.д.
5. Получение владельца и группы:
    - Используются функции getpwuid и getgrgid для получения имён владельца и группы по их ID.
6. Форматирование времени:
    - Время последнего изменения файла форматируется через strftime в человекочитаемый формат ("%b %d %H:%M").
7. Дополнительная информация:
    - Отображается номер inode файла (как "первичный адрес на диске").

Структурированное представление данных

1. Данные о каждом файле представлены в формате, аналогичном ls -l, включая:
    - Права доступа, количество жёстких ссылок, имя владельца и группы, размер файла, дату изменения, имя файла.
    - Дополнительно отображается inode-файл.
2. Не используются структуры для хранения информации — вывод идёт напрямую, данные обрабатываются по мере чтения.

Практическая часть

Серверный логгер

Серверный логгер — это класс, который наследуется от абстрактного logger. Внутри он хранит словарь, где для каждого уровня важности задано два параметра: флаг вывода в консоль и путь к файлу для записи логов. Для отправки сообщений на удалённый сервер логгер использует HTTP-клиент, передавая пути к файлам в JSON-запросе для серверной обработки. Управление ресурсами реализовано через таймауты HTTP-запросов и проверку конфигурации при инициализации, что исключает утечки и обеспечивает стабильную работу с сервером.

Основной метод — log, который принимает сообщение и уровень важности, форматирует сообщение, отправляет его на сервер, а при необходимости выводит в консоль и записывает в файл.

Листинг #1. Функция log

```
logger& server_logger::log(
    const std::string &message,
    const logger::severity severity) &
{
    if (message.empty()) {
        std::cerr << "Warning: Empty log message" << std::endl;
    }
    try {
        std::string formatted = _format;
        std::map<std::string, std::string> replacements = {
            {"%d", get_current_date()},
            {"%t", get_current_time()},
            {"%s", severity_to_string(severity)},
            {"%m", message}
        };
```

```
        for (const auto& [pattern, replacement] : replacements) {
            formatted = std::regex_replace(formatted, std::regex(pattern),
replacement);
        }
        std::string server_severity =
convert_severity_for_server(severity_to_string(severity));
        nlohmann::json payload;
        try {
            payload = {
                {"pid", inner_getpid()},
                {"severity", server_severity},
                {"message", formatted},
                {"streams", nlohmann::json::array()}
            };
        } catch (const nlohmann::json::exception& e) {
            std::cerr << "Error creating JSON payload: " << e.what() <<
std::endl;
            throw;
        }
        if (const auto it = _streams.find(severity); it != _streams.end())
{
            const auto& [path, is_console] = it->second;
            if (is_console) {
                payload["streams"].push_back({{"type", "console"}});
                std::cout << formatted << std::endl;
            }
            if (!path.empty()) {
                payload["streams"].push_back({
                    {"type", "file"},
                    {"path", path}
```

```
                });

            }

        }

        try {

            _client.set_connection_timeout(2);


            if (auto res = _client.Post("/log", payload.dump(),
"application/json"); !res) {

                std::cerr << "HTTP error: " <<
httplib::to_string(res.error()) << std::endl;

            } else if (res->status != 200) {

                std::cerr << "Server error: Status " << res->status << ",
Body: " << res->body << std::endl;

            }

        } catch (const std::exception& e) {

            std::cerr << "Exception during HTTP request: " << e.what() <<
std::endl;

        }

    } catch (const std::exception& e) {

        std::cerr << "Logger error: " << e.what() << std::endl;


    }

    return *this;

}
```

Метод сначала проверяет, не пустое ли сообщение. Затем он форматирует строку, заменяя спецификаторы `%d`, `%t`, `%s`, `%m` на дату, время, уровень важности и текст сообщения соответственно. Форматирование выполняется с использованием регулярных выражений для точной замены. Далее создаётся JSON-payload.

Форматирование выполняется вспомогательными функциями get_current_date и get_current_time, которые получают текущие дату и время в

формате GMT. Эти функции кроссплатформенны, используя gmtime_s на Windows и gmtime_r на Unix.

Листинг #2. Функция get_current_date

```
std::string get_current_date() {
    try {
        const auto now = std::chrono::system_clock::now();
        auto in_time = std::chrono::system_clock::to_time_t(now);
        std::tm tm_buf{};
        #ifdef _WIN32
        gmtime_s(&tm_buf, &in_time);
        #else
        if (gmtime_r(&in_time, &tm_buf) == nullptr) {
            throw std::runtime_error("Failed to convert time to GMT");
        }
        #endif

        std::stringstream ss;
        ss << std::put_time(&tm_buf, "%Y-%m-%d");
        return ss.str();
    } catch (const std::exception& e) {
        std::cerr << "Error getting date: " << e.what() << std::endl;
        return "[DATE ERROR]";
    }
}
```

Функция получает текущее время, конвертирует его в структуру std::tm и форматирует в строку вида YYYY-MM-DD. При ошибке возвращается строка "[DATE ERROR]", а диагностика выводится в std::cerr.

Класс server_logger_builder отвечает за конфигурацию логгера. Метод build создаёт экземпляр server_logger, проверяя валидность настроек. Методы add_file_stream и add_console_stream добавляют потоки для указанных уровней важности, а transform_with_configuration позволяет загрузить настройки из JSON-файла.

Листинг #3. Метод transform_with_configuration

```
logger_builder& server_logger_builder::transform_with_configuration(
    std::string const &configuration_file_path,
    std::string const &configuration_path) &
{
    try {
        if (!std::filesystem::exists(configuration_file_path)) {
            throw std::runtime_error("Configuration file not found: " +
configuration_file_path);
        }

        nlohmann::json config;
        std::ifstream file(configuration_file_path);
        file >> config;

        nlohmann::json* section = &config;
        if (!configuration_path.empty()) {
            section =
&config.at(nlohmann::json::json_pointer(configuration_path));
        }

        if (section->contains("destination")) {
            auto dest = section->at("destination").get<std::string>();
            set_destination(dest);
        }
```

```
if (section->contains("format")) {
    std::string format = section->at("format");
    set_format(format);
}

if (section->contains("streams")) {
    const nlohmann::json& streams = section->at("streams");
    for (const auto& stream : streams) {
        const std::string type = stream.at("type");
        std::vector<logger::severity> severities;
        for (const auto& sev_str : stream.at("severities")) {
            severities.push_back(string_to_severity(sev_str));
        }

        if (type == "file") {
            const std::string path = stream.at("path");
            for (auto sev : severities) {
                add_file_stream(path, sev);
            }
        } else if (type == "console") {
            for (auto sev : severities) {
                add_console_stream(sev);
            }
        }
    }
}
return *this;
```

```
    } catch (const std::exception& e) {

        std::cerr << "Error in transform_with_configuration: " << e.what()
<< std::endl;

        return *this;

    }

}
```

Метод читает JSON-конфигурацию, извлекая адрес сервера (destination), шаблон формата (format) и список потоков (streams). Для каждого потока определяется тип — файл или консоль — и вызывается соответствующий метод добавления (add_file_stream или add_console_stream). Все ошибки обрабатываются для поддержания устойчивости конфигурации.

Серверный логгер не работает с файлами напрямую, а передаёт пути к ним в JSON-payload, позволяя серверу самостоятельно записывать логи. Это снижает нагрузку на клиент и упрощает управление ресурсами. HTTP-клиент (_client) настроен с таймаутами (2 секунды на соединение, 5 секунд на чтение) для предотвращения зависаний.

Для передачи настроек без лишних затрат реализованы операции копирования и перемещения логгера. Класс server_logger_builder контролирует доступность путей к файлам при добавлении потоков, предотвращая ошибки на этапе конфигурации. Валидация URL и формата JSON обеспечивает надёжность работы.

В итоге серверный логгер сочетает гибкость конфигурации через паттерн «Строитель», надёжную отправку логов на сервер и локальный вывод в консоль и файлы. Он устойчив к ошибкам, кроссплатформенен и оптимизирован для распределённых систем, где важно централизованное логирование. Благодаря JSON-payload и HTTP, логгер легко интегрируется с современными системами мониторинга, например Elasticsearch.

Global heap allocator

Аллокатор глобальной кучи (allocator_global_heap) представляет собой класс, наследующий интерфейс std::pmr::memory_resource, и предназначен для выделения и освобождения памяти с помощью глобальных операторов new и delete. Он интегрирован с системой логирования, что позволяет отслеживать операции выделения и освобождения памяти, а также фиксировать возникающие ошибки.

Внутри класс хранит указатель на объект логгера, через который записываются сообщения о выполнении операций с памятью и жизненном цикле аллокатора. Реализованы стандартные методы интерфейса: do_allocate_sm для выделения памяти заданного размера с использованием глобального оператора new, с логированием начала и успешного завершения операции и обработкой исключений; do_deallocate_sm для освобождения памяти и do_is_equal для сравнения с другими ресурсами памяти.

Листинг #4. Метод do_allocate_sm

```
void* allocator_global_heap::do_allocate_sm(const size_t size)

{

    debug_with_guard("Starting allocation of size " +
std::to_string(size));

    try

    {

        void* ptr = ::operator new(size);

        std::ostringstream oss;

        oss << "0x" << std::hex << reinterpret_cast<std::uintptr_t>(ptr);

        debug_with_guard("Successfully allocated memory at " + oss.str() +
" of size " + std::to_string(size));

        return ptr;

    }

    catch (const std::bad_alloc& e)

    {

        error_with_guard("Failed to allocate memory of size " +
std::to_string(size) + ": " + std::string(e.what()));

        throw;

    }

    catch (const std::exception& e)

    {

        error_with_guard("Unexpected exception during memory allocation: "
+ std::string(e.what()));
```

```
        throw;
    }
}
```

Метод принимает размер памяти (size) и выполняет следующие шаги:

1. Логирует начало операции выделения на уровне debug.
2. Вызывает ::operator new для выделения памяти.
3. При успехе форматирует адрес выделенной памяти в шестнадцатеричном виде и логирует его вместе с размером.
4. При возникновении исключения std::bad_alloc или других ошибок логирует их на уровне error и пробрасывает исключение дальше.

Форматирование адреса памяти выполняется с использованием std::ostringstream, что обеспечивает безопасное преобразование указателя в строку.

Метод do_deallocate_sm освобождает память, ранее выделенную аллокатором, с использованием глобального оператора delete. Он также логирует операцию и проверяет указатель на nullptr.

Листинг #5. Метод do_deallocate_sm

```
void allocator_global_heap::do_deallocate_sm(void* at)
{
    if (at == nullptr)
    {
        debug_with_guard("Attempted to deallocate NULL pointer - 
ignoring");
        return;
    }

    std::ostringstream oss;
    oss << "0x" << std::hex << reinterpret_cast<std::uintptr_t>(at);
    debug_with_guard("Starting deallocation of memory at " + oss.str());
    ::operator delete(at);
    debug_with_guard("Successfully deallocated memory at " + oss.str());
```

```
}
```

Метод выполняет следующие действия:

1. Проверяет, не является ли указатель nullptr. Если это так, логирует попытку освобождения и завершает выполнение.
2. Форматирует адрес памяти в шестнадцатеричном виде.
3. Логирует начало освобождения на уровне debug.
4. Вызывает ::operator delete для освобождения памяти.
5. Логирует успешное завершение операции.

Обработка nullptr предотвращает неопределённое поведение, а логирование помогает отслеживать все операции освобождения.

Аллокатор поддерживает конструкторы и операторы присваивания для корректного управления указателем на логгер. Конструктор принимает указатель на логгер, который используется для всех операций логирования.

Листинг #6. Конструктор и деструктор

```
allocator_global_heap::allocator_global_heap(logger *logger)
    : _logger(logger)
{
    trace_with_guard("allocator_global_heap constructor started");
    trace_with_guard("allocator_global_heap constructor finished");
}
allocator_global_heap::~allocator_global_heap()
{
    trace_with_guard("allocator_global_heap destructor started");
    trace_with_guard("allocator_global_heap destructor finished");
}
```

Конструктор инициализирует поле _logger и логирует начало и конец своей работы на уровне trace. Деструктор логирует своё выполнение, но не освобождает логгер, так как он не владеет им. Копирование и перемещение также логируются, обеспечивая полный контроль над жизненным циклом объекта.

Аллокатор глобальной кучи не управляет памятью напрямую, а делегирует эту задачу глобальным операторам new и delete. Указатель на логгер передаётся по ссылке и не копируется, что предотвращает утечки ресурсов.

Операции копирования и перемещения реализованы стандартно:

- Копирование: Копируются только указатель на логгер, без дублирования самого логгера.
- Перемещение: Указатель на логгер переносится без дополнительных затрат.
- Сравнение: Метод do_is_equal проверяет, является ли другой ресурс экземпляром allocator_global_heap, что позволяет корректно сравнивать аллокаторы.

Аллокатор глобальной кучи (allocator_global_heap) представляет собой простую, но надёжную реализацию std::pmr::memory_resource, интегрированную с системой логирования. Он использует глобальные операторы new и delete для управления памятью, минимизируя накладные расходы.

Boundary tags allocator

Аллокатор с пограничными метками (allocator_boundary_tags) реализован как класс, наследующий интерфейс std::pmr::memory_resource, и предназначен для управления памятью с использованием структуры блоков, разделённых пограничными метками. Он поддерживает три стратегии выделения памяти (first fit, best fit, worst fit), интегрирован с системой логирования и обеспечивает потокобезопасность через мьютекс. Аллокатор подходит для приложений, требующих гибкого управления памятью с учётом фрагментации и отладки.

Метод do_allocate_sm выделяет память заданного размера, выбирая подходящий свободный блок в зависимости от текущей стратегии выделения. Операция защищена мьютексом для потокобезопасности.

Листинг #7. Метод do_allocate_sm

```
[[nodiscard]] void* allocator_boundary_tags::do_allocate_sm(size_t size) {

    std::lock_guard<std::mutex> lock(get_mutex());

    const size_t required_size = size + occupied_block_metadata_size;

    debug_with_guard("Starting allocation for " +
      std::to_string(required_size) + " bytes");

    void* suitable_block = nullptr;
```

```
switch (get_fit_mode()) {
    case allocator_with_fit_mode::fit_mode::first_fit:
        suitable_block = get_first_fit(required_size);
        break;
    case allocator_with_fit_mode::fit_mode::the_best_fit:
        suitable_block = get_best_fit(required_size);
        break;
    case allocator_with_fit_mode::fit_mode::the_worst_fit:
        suitable_block = get_worst_fit(required_size);
        break;
}
if (!suitable_block) {
    error_with_guard("Allocation failed: no suitable block found");
    throw std::bad_alloc();
}
size_t available_space = 0;
if (suitable_block == _trusted_memory) {
    if (*get_first_block_ptr(_trusted_memory) == nullptr) {
        available_space = get_size(_trusted_memory);
    } else {
        available_space =
 reinterpret_cast<std::byte*>(*get_first_block_ptr(_trusted_memory)) -
 reinterpret_cast<std::byte*>(_trusted_memory) -
 allocator_metadata_size;
    }
} else {
    void* next_block = get_next_block(suitable_block);
    if (next_block == nullptr) {
```

```
        available_space =
  (reinterpret_cast<std::byte*>(_trusted_memory) +
  allocator_metadata_size + get_size(_trusted_memory)) -
  (reinterpret_cast<std::byte*>(suitable_block) +
  get_occupied_block_size(suitable_block) +
  occupied_block_metadata_size);

    } else {

        available_space = reinterpret_cast<std::byte*>(next_block) -
  (reinterpret_cast<std::byte*>(suitable_block) +
  get_occupied_block_size(suitable_block) +
  occupied_block_metadata_size);

    }

}

const size_t actual_size = (available_space < required_size +
  occupied_block_metadata_size) ? available_space : required_size;

if (actual_size != required_size) {

    warning_with_guard("Reduced allocation size to " +
  std::to_string(actual_size));

}

void* block_start = nullptr;

if (suitable_block == _trusted_memory) {

    block_start = reinterpret_cast<std::byte*>(_trusted_memory) +
  allocator_metadata_size;

} else {

    block_start = reinterpret_cast<std::byte*>(suitable_block) +
  get_occupied_block_size(suitable_block) +
  occupied_block_metadata_size;

}

auto* size_ptr = reinterpret_cast<size_t*>(block_start);

*size_ptr = size;

auto* back_ptr = reinterpret_cast<void**>(size_ptr + 1);

*back_ptr = suitable_block;

auto* forward_ptr = reinterpret_cast<void**>(back_ptr + 1);

*forward_ptr = (suitable_block == _trusted_memory) ?
  *get_first_block_ptr(_trusted_memory) :
  get_next_block(suitable_block);
```

```
    auto* parent_ptr = reinterpret_cast<void**>(forward_ptr + 1);

    *parent_ptr = _trusted_memory;

    void* next_block = nullptr;

    if (suitable_block == _trusted_memory) {

        next_block = *get_first_block_ptr(_trusted_memory);

    } else {

        next_block = get_next_block(suitable_block);

    }

    if (next_block != nullptr) {

        auto* next_prev_ptr =
     reinterpret_cast<void**>(reinterpret_cast<std::byte*>(next_block) +
     sizeof(size_t));

        *next_prev_ptr = block_start;

    }

    if (suitable_block == _trusted_memory) {

        *get_first_block_ptr(_trusted_memory) = block_start;

    } else {

        auto* suitable_next_ptr =
     reinterpret_cast<void**>(reinterpret_cast<std::byte*>(suitable_block)
     + sizeof(size_t) + sizeof(void*));

        *suitable_next_ptr = block_start;

    }

    information_with_guard("Remaining free space: " +
     std::to_string(get_free_size()));

    debug_with_guard(print_blocks());

    return reinterpret_cast<std::byte*>(block_start) +
     occupied_block_metadata_size;

}
```

Метод выполняет следующие шаги:

1. Блокирует мьютекс для потокобезопасности.
2. Вычисляет требуемый размер блока, включая метаданные (occupied_block_metadata_size).
3. Логирует начало выделения на уровне debug.

4. Выбирает подходящий свободный блок с помощью метода get_first_fit, get_best_fit или get_worst_fit, в зависимости от режима (fit_mode).
5. Если блок не найден, логирует ошибку и выбрасывает std::bad_alloc.
6. Определяет доступное пространство в выбранном блоке, учитывая его положение относительно _trusted_memory и следующего блока.
7. Если доступное пространство меньше требуемого, выделяет меньший блок и логирует предупреждение.
8. Создаёт новый занятый блок, записывая метаданные: размер, указатели на предыдущий, следующий и родительский блоки.
9. Обновляет связи между блоками, корректируя указатели в соседних блоках.
10. Логирует оставшееся свободное пространство и состояние блоков.
11. Возвращает указатель на начало пользовательской области блока.

Метод do_deallocate_sm освобождает память, ранее выделенную аллокатором, удаляя блок из цепочки и обновляя связи между соседними блоками.

Листинг #8. Метод do_deallocate_sm

```
void allocator_boundary_tags::do_deallocate_sm(void* at) {

    std::lock_guard<std::mutex> lock(get_mutex());

    if (at == nullptr) {

        error_with_guard("Attempt to deallocate null pointer");

        throw std::invalid_argument("Cannot deallocate null pointer");

    }

    std::byte* block_start = reinterpret_cast<std::byte*>(at) -
     occupied_block_metadata_size;

    if (get_parent_block(block_start) != _trusted_memory) {

       error_with_guard("Deallocation attempt for foreign block at " +
     std::to_string(reinterpret_cast<uintptr_t>(at)));

       throw std::logic_error("Attempt to deallocate memory not owned by
     this allocator");

    }

    const size_t block_size = get_occupied_block_size(block_start);

    debug_with_guard("Deallocating block at " +
     std::to_string(reinterpret_cast<uintptr_t>(at)) + ", size: " +
     std::to_string(block_size) + " bytes");

    void* prev_block = get_prev_block(block_start);
```

```
    void* next_block = get_next_block(block_start);

    std::byte* prev_block_next_ptr = (prev_block == _trusted_memory) ?
      reinterpret_cast<std::byte*>(get_first_block_ptr(_trusted_memory)) :
      reinterpret_cast<std::byte*>(prev_block) + sizeof(size_t) +
      sizeof(void*);

    *reinterpret_cast<void**>(prev_block_next_ptr) = next_block;

    if (next_block != nullptr) {

        std::byte* next_block_prev_ptr =
      reinterpret_cast<std::byte*>(next_block) + sizeof(size_t);

        *reinterpret_cast<void**>(next_block_prev_ptr) = prev_block;

    }

    information_with_guard("Memory state after deallocation:");

    information_with_guard("  Total free memory: " +
      std::to_string(get_free_size()) + " bytes");

    debug_with_guard(print_blocks());

}
```

Метод выполняет следующие действия:

1. Блокирует мьютекс для потокобезопасности.
2. Проверяет, не является ли указатель nullptr, и выбрасывает исключение при попытке освободить нулевой указатель.
3. Вычисляет начало блока, смещая указатель на размер метаданных.
4. Проверяет, принадлежит ли блок текущему аллокатору, сравнивая родительский блок с _trusted_memory.
5. Логирует начало освобождения, включая адрес и размер блока.
6. Обновляет указатели в предыдущем и следующем блоках, исключая освобождаемый блок из цепочки.
7. Логирует состояние памяти после освобождения, включая общее количество свободной памяти и структуру блоков.

Аллокатор предоставляет итератор boundary_iterator, который позволяет обходить блоки памяти и получать информацию об их размере и состоянии (занятый или свободный).

Листинг #9. Оператор инкремента итератора

```
allocator_boundary_tags::boundary_iterator&
     allocator_boundary_tags::boundary_iterator::operator++() & noexcept {

    if (!_occupied) {

        _occupied = true;

        _occupied_ptr = (_occupied_ptr == _trusted_memory) ?
      *get_first_block_ptr(_trusted_memory) :
      get_next_block(_occupied_ptr);

        return *this;

    }

    const size_t current_size = get_occupied_block_size(_occupied_ptr);

    const std::byte* current_end =
      reinterpret_cast<std::byte*>(_occupied_ptr) + current_size +
      occupied_block_metadata_size;

    void* next_block = get_next_block(_occupied_ptr);

    bool is_contiguous = (next_block == current_end);

    bool is_last_block = (next_block == nullptr) && (current_end ==
      reinterpret_cast<std::byte*>(_trusted_memory) +
      get_size(_trusted_memory) + allocator_metadata_size);

    if (is_contiguous || is_last_block) {

        _occupied_ptr = next_block;

    } else {

        _occupied = false;

    }

    return *this;

}
```

Итератор поддерживает переход между занятыми и свободными блоками:

1. Если текущий блок свободный (_occupied == false), итератор переходит к следующему занятому блоку.
2. Если текущий блок занятый, итератор проверяет, примыкает ли следующий занятый блок непосредственно к концу текущего. Если нет, итератор переходит в свободный режим.
3. Если достигнут конец памяти, итератор указывает на nullptr.

4. Метод size() возвращает размер текущего блока, а occupied() — его состояние.

Итератор используется в методах get_blocks_info и get_free_size для анализа структуры памяти и отладки.

Buddy system allocator

Аллокатор двойников (allocator_buddies_system) реализован как класс, наследующий интерфейс std::pmr::memory_resource, предназначенный для эффективного управления памятью с использованием алгоритма "buddy system" (система друзей). Этот алгоритм оптимизирован для выделения и освобождения блоков памяти, размеры которых являются степенями двойки, что минимизирует фрагментацию. Аллокатор поддерживает три стратегии выделения памяти (first fit, best fit, worst fit), синхронизацию через мьютекс и интеграцию с системой логирования для отслеживания операций.

Метод do_allocate_sm выполняет выделение блока памяти заданного размера, выбирая подходящий свободный блок в зависимости от текущей стратегии (fit_mode). Он защищён мьютексом для потокобезопасности и логирует все этапы операции.

Листинг #10. Метод do_allocate_sm

```
void *allocator_buddies_system::do_allocate_sm(size_t size)

{

    std::lock_guard lock(get_mutex());


    const size_t real_size = size + occupied_block_metadata_size;

    debug_with_guard("Allocator buddies system started allocating " +
      std::to_string(real_size) + " bytes");


    void* free_block = [this, real_size]() -> void* {

        switch (get_fit_mode()) {

            case allocator_with_fit_mode::fit_mode::first_fit:  return
      get_first_fit(real_size);

            case allocator_with_fit_mode::fit_mode::the_best_fit: return
      get_best_fit(real_size);
```

```
        case allocator_with_fit_mode::fit_mode::the_worst_fit: return
  get_worst_fit(real_size);

        default: return nullptr;

    }

}();


if (free_block == nullptr) {

    const auto error_msg = "Allocator boundary tags throwing bad_alloc
  while trying to allocate " +

                           std::to_string(real_size) + " bytes";

    error_with_guard(error_msg);

    throw std::bad_alloc();

}


while (get_occupied_block_size(free_block) >= real_size * 2) {

    auto* metadata = static_cast<block_metadata*>(free_block);

    --metadata->size;


    auto* buddy_metadata =
  static_cast<block_metadata*>(get_buddy(free_block));

    buddy_metadata->occupied = false;

    buddy_metadata->size = metadata->size;

}


if (get_occupied_block_size(free_block) != real_size) {

    warning_with_guard("Allocator buddies system changed allocating
  block size to " +


  std::to_string(get_occupied_block_size(free_block)));

}


auto* metadata = static_cast<block_metadata*>(free_block);
```

```
    metadata->occupied = true;

    *reinterpret_cast<void**>(metadata + 1) = _trusted_memory;


    information_with_guard(std::to_string(get_free_size()));

    debug_with_guard(print_blocks());


    return static_cast<std::byte*>(free_block) +
      occupied_block_metadata_size;

}
```

Метод выполняет следующие шаги:

1. Захватывает мьютекс для потокобезопасности.
2. Вычисляет реальный размер блока, включая метаданные (occupied_block_metadata_size).
3. Логирует начало выделения на уровне debug.
4. Выбирает свободный блок, используя одну из стратегий:
    - get_first_fit: Возвращает первый подходящий блок.
    - get_best_fit: Выбирает наименьший подходящий блок.
    - get_worst_fit: Выбирает наибольший подходящий блок.
5. Если блок не найден, логирует ошибку и выбрасывает std::bad_alloc.
6. Если найденный блок слишком велик, он рекурсивно делится на два брата, пока размер не станет минимально достаточным.
7. Устанавливает флаг занятости и сохраняет указатель на _trusted_memory в метаданных блока.
8. Логирует оставшееся свободное место и состояние блоков.

Метод do_deallocate_sm освобождает ранее выделенный блок памяти и пытается объединить его с "другом", если тот свободен, чтобы минимизировать фрагментацию. Операция также защищена мьютексом.

Листинг #11. Метод do_deallocate_sm

```
void allocator_buddies_system::do_deallocate_sm(void *at)

{

    std::lock_guard lock(get_mutex());
```

```
void* block_start = reinterpret_cast<std::byte*>(at) -
 occupied_block_metadata_size;


if (get_parent_block(block_start) != _trusted_memory)

{

    error_with_guard("Incorrect deallocation object");

    throw std::logic_error("Incorrect deallocation object");

}


size_t block_size = get_occupied_block_size(block_start) -
 occupied_block_metadata_size;


debug_with_guard(get_dump((char*)at, block_size));


reinterpret_cast<block_metadata*>(block_start)->occupied = false;


void* buddy = get_buddy(block_start);


while(get_occupied_block_size(block_start) < get_size(_trusted_memory)
 && get_occupied_block_size(block_start) ==
 get_occupied_block_size(buddy) && !is_occupied(buddy))

{

    void* i_ptr = block_start < buddy ? block_start : buddy;


    auto metadata = reinterpret_cast<block_metadata*>(i_ptr);

    ++metadata->size;


    block_start = i_ptr;

    buddy = get_buddy(block_start);

}


information_with_guard(std::to_string(get_free_size()));
```

```
    debug_with_guard(print_blocks());
}
```

Метод выполняет следующие действия:

1. Захватывает мьютекс для синхронизации.
2. Вычисляет начало блока, вычитая размер метаданных.
3. Проверяет, принадлежит ли блок аллокатору, сравнивая указатель в метаданных с _trusted_memory. Если нет, выбрасывает исключение.
4. Логирует содержимое освобождаемого блока на уровне debug.
5. Устанавливает флаг occupied в false.
6. Проверяет брата (get_buddy). Если брат свободен и имеет тот же размер, блоки объединяются, увеличивая размер метаданных.
7. Повторяет объединение, пока возможно.
8. Логирует оставшееся свободное место и состояние блоков.

Конструктор аллокатора инициализирует блок памяти, выделенный родительским аллокатором, и настраивает метаданные для работы двойников.

Листинг #12. Конструктор

```
allocator_buddies_system::allocator_buddies_system(
    size_t space_size,
    std::pmr::memory_resource *parent_allocator,
    logger *logger,
    allocator_with_fit_mode::fit_mode allocate_fit_mode)
{
    if (space_size < allocator_metadata_size) {
        throw std::logic_error("Small size");
    }
    if (parent_allocator == nullptr) {
        parent_allocator = std::pmr::get_default_resource();
    }

    size_t k = __detail::nearest_greater_k_of_2(space_size);
```

```
    size_t size = __detail::power_of_2(k) + allocator_metadata_size;


    _trusted_memory = parent_allocator->allocate(size);


    auto logger_ptr = reinterpret_cast<class logger**>(_trusted_memory);

    *logger_ptr = logger;

    auto parent_allocator_ptr =
      reinterpret_cast<std::pmr::memory_resource**>(logger_ptr + 1);

    *parent_allocator_ptr = parent_allocator;

    auto fit_mode_ptr =
      reinterpret_cast<allocator_with_fit_mode::fit_mode*>(parent_allocator
      _ptr + 1);

    *fit_mode_ptr = allocate_fit_mode;

    auto size_ptr = reinterpret_cast<unsigned char*>(fit_mode_ptr + 1);

    *size_ptr = k;

    auto mutex_ptr = reinterpret_cast<std::mutex*>(size_ptr + 1);

    new (mutex_ptr) std::mutex();


    auto block_start =
      reinterpret_cast<block_metadata*>(reinterpret_cast<std::byte*>(_trust
      ed_memory) + allocator_metadata_size);


    block_start->occupied = false;

    block_start->size = k - min_k;

}
```

B-дерево

Основные компоненты реализации

Узел B-дерева хранит массив пар ключ-значение и указатели на дочерние узлы. Используется boost::container::static_vector для фиксированного размера хранения, что оптимизирует использование памяти. Узел поддерживает универсальное конструирование через variadic templates.

Листинг #13. Структура узла B-дерева

```
struct btree_node {

public:

    boost::container::static_vector<tree_data_type, maximum_keys_in_node + 1> _keys;

    boost::container::static_vector<btree_node*, maximum_keys_in_node + 2> _pointers;

    template<class ...Args>

    explicit btree_node(Args&& ...args);

    virtual ~btree_node() = default;

};
```

Основной класс

Класс B_tree реализует основную логику B-дерева. Он является шаблонным и поддерживает настройку типов ключей, значений, компаратора и степени дерева.

Листинг #14. Фрагмент класса B-дерева

```
template<typename tkey, typename tvalue, comparator<tkey> compare = std::less<tkey>, size_t t = 2>

class B_tree : private compare {

protected:

    btree_node* _root;

    logger* _logger;

    size_t _size;
```

```
    pp_allocator<tree_data_type> _allocator;
};
```

Итераторы

Реализация включает итераторы для обхода ключей в отсортированном порядке

Итераторы соответствуют концепции std::bidirectional_iterator и содержат поле _backup для хранения предыдущего узла, что позволяет корректно обрабатывать декремент от итератора end().

Вспомогательный класс

Класс btree_impl инкапсулирует базовую логику работы с B-деревом, реализуя шаблон проектирования «template method».

Листинг #15. Вспомогательный класс btree_impl

```
namespace __detail {
    template<typename tkey, typename tvalue, typename compare, size_t t>
    class btree_impl {
    public:
        template<class ...Args>
        static btree_node* create_node(B_tree &cont, Args &&...args);
        static void delete_node(B_tree &cont, btree_node* n);
        static void erase(B_tree &cont, btree_node** node_ptr);
        static void swap(B_tree &lhs, B_tree &rhs) noexcept;
    };
}
```

Вставка элементов

Вставка элемента выполняется по стандартному алгоритму B-дерева. Сначала находится подходящий листовой узел для вставки. Если узел переполняется, он разделяется, а медианный ключ поднимается в родительский узел.

Листинг #16. Цикл вставки нового ключа

```
if (node->_keys.size() < 2 * t - 1) {

    node->_keys.push_back(new_data);

    std::sort(node->_keys.begin(), node->_keys.end(), compare);

} else {

    split_node(node, new_data);

}

++_size;

if (_logger) {

    _logger->log("Inserted new key with move semantics",
logger::severity::debug);

}

__detail::btree_impl<tkey, tvalue, compare, t>::post_insert(*this, &node);

return infix_iterator(node, new_data);
```

Удаление элементов

Удаление рассматривает случаи, когда узел становится менее чем наполовину заполненным. В таких случаях происходит заимствование ключей у соседних узлов или слияние узлов. Если удаляется ключ из внутреннего узла, он заменяется предшественником или преемником.

Листинг #17. Функция удаления элемента erase

```
typename B_tree<tkey, tvalue, compare, t>::btree_iterator

B_tree<tkey, tvalue, compare, t>::erase(const tkey& key) {

    if (!_root) return end();

    std::function<bool(btree_node*, const tkey&)> remove_key;

    remove_key = [this, &remove_key](btree_node* node, const tkey& k) ->
bool {

        size_t idx = 0;

        while (idx < node->_keys.size() &&
compare_keys(node->_keys[idx].first, k)) ++idx;

        if (idx < node->_keys.size() && !compare_keys(k,
node->_keys[idx].first) && !compare_keys(node->_keys[idx].first, k)) {
```

```
            if (node->_pointers.empty()) {
                node->_keys.erase(node->_keys.begin() + idx);
                --_size;
                return node->_keys.size() < minimum_keys_in_node;
            } else {
                btree_node* pred = node->_pointers[idx];
                while (!pred->_pointers.empty()) pred =
pred->_pointers.back();
                auto pred_pair = pred->_keys.back();
                node->_keys[idx] = pred_pair;
                bool underflow = remove_key(node->_pointers[idx],
pred_pair.first);
                if (underflow) {
                    node->_pointers[idx]->_keys.push_back({});}
                return false;}
        }
        if (node->_pointers.empty()) {
            return node->_keys.size() < minimum_keys_in_node;
        }
        bool underflow = remove_key(node->_pointers[idx], k);
        if (underflow) {
        }
        return false;
    };
    remove_key(_root, key);
    if (_root->_keys.empty()) {
        btree_node* old = _root;
        if (!_root->_pointers.empty()) {
            _root = _root->_pointers[0];
```

```
        } else {_root = nullptr;}
        delete old;}
    return find(key);}
```

Поиск элементов

Поиск выполняется путем обхода дерева от корня к листьям, сравнивая ключи с искомым значением.

Листинг #18. Функция поиска элемента find

```
btree_node* current = _root;
while (current != nullptr) {
    for (size_t i = 0; i < current->_keys.size(); ++i) {
        if (compare::operator()(key, current->_keys[i].first)) {
            current = current->_pointers[i];
            break;
        } else if (compare::operator()(current->_keys[i].first, key)) {
            if (i == current->_keys.size() - 1) {
                current = current->_pointers[i + 1];
                break;
            }
        } else {
            return infix_iterator(current, i);
        }
    }
}
return end();
```

B-дерево на диске

Основные компоненты реализации

Структура узла B-дерева на диске

Узел B-дерева хранит массив пар ключ-значение, массив указателей на дочерние узлы и метаинформацию (размер, флаг листа, позиция на диске). Структура реализована с учетом сериализации для хранения на диске, обеспечивая эффективное использование пространства.

Листинг #19. Структура узла B-дерева

```
struct btree_disk_node {

    size_t size;

    bool _is_leaf;

    size_t position_in_disk;

    std::vector<tree_data_type> keys;

    std::vector<size_t> pointers;




    void serialize(std::fstream& stream, std::fstream& stream_for_data)
const;

    static btree_disk_node deserialize(std::fstream& stream, std::fstream&
stream_for_data);

    explicit btree_disk_node(bool is_leaf);

    btree_disk_node();

    size_t calculate_block_size() const;

};
```

Основной класс B-дерева на диске

Класс `B_tree_disk` является шаблонным и поддерживает операции с B-деревом, хранимым на диске. Он использует два файла: один для структуры дерева (`.tree`), другой для хранения данных ключ-значение (`.data`).

Листинг #20. Фрагмент класса B-дерева на диске

```
template<serializable tkey, serializable tvalue, compator<tkey> compare =
std::less<tkey>, std::size_t t = 2>

class B_tree_disk final : private compare {

protected:

    std::fstream _file_for_tree;

    std::fstream _file_for_key_value;

    allocator_type _allocator;

    size_t _node_block_size;

    size_t _position_root;

    btree_disk_node _current_node;

    static size_t _count_of_node;

};
```

Класс поддерживает семантику перемещения, но копирование запрещено из-за работы с файлами. Основные операции (`insert`, `erase`, `update`, `at`) реализованы с учетом дискового хранения и обеспечивают корректное управление файлами.

Итераторы

Реализован константный двунаправленный итератор `btree_disk_const_iterator`, соответствующий стандарту STL и концепции std::bidirectional_iterator. Итератор использует стек пар `{позиция_узла, индекс_ключа}` для навигации по дереву в инфиксном порядке (отсортированный обход).

Листинг #21. Итератор B-дерева

```
class btree_disk_const_iterator {

    std::stack<std::pair<size_t, size_t>> _path; // Стек пути (позиция
узла, индекс ключа)

    size_t _index;                                  // Текущий индекс

    B_tree_disk<tkey, tvalue, compare, t>& _tree; // Ссылка на дерево
```

```
public:
    using value_type = tree_data_type_const;
    using reference = value_type;
    using pointer = value_type*;
    using iterator_category = std::bidirectional_iterator_tag;
    using difference_type = ptrdiff_t;

    value_type operator*() const noexcept;
    self& operator++();
    self operator++(int);
    self& operator--();
    self operator--(int);
    bool operator==(const self& other) const noexcept;
    bool operator!=(const self& other) const noexcept;
};
```

Итератор поддерживает инкремент и декремент для обхода ключей в отсортированном порядке, а также методы сравнения. Поле `_path` хранит путь от корня до текущего узла, что позволяет эффективно перемещаться по дереву, минимизируя дисковые операции.

Вспомогательные методы

Основная логика операций с деревом инкапсулирована в методах класса `B_tree_disk`. Ключевые методы включают:

- disk_read: чтение узла с диска по заданной позиции.

- disk_write: запись узла на диск с учетом сериализации.

- find_path: поиск пути к ключу с возвратом стека позиций и информации о нахождении.

- split_node: разделение узла при превышении максимального количества ключей.

- rebalance_node: ребалансировка узла после удаления для соблюдения свойств B-дерева.

- insert_array и remove_array: вспомогательные функции для вставки и удаления элементов в массивы узла.

Эти методы обеспечивают корректное управление структурой дерева и данными на диске, минимизируя количество операций ввода-вывода.

Основные операции

Вставка элементов

Вставка элемента выполняется следующим образом:

1. Поиск подходящего узла с помощью `find_path`.
2. Если ключ уже существует, вставка не выполняется.
3. Вставка данных в массив ключей узла с помощью `insert_array`.
4. Если узел переполняется (более 2t-1 ключей), вызывается `split_node` для разделения узла.
5. Метаданные (количество узлов, позиция корня) обновляются с помощью `write_metadata`.

Листинг #22. Операция вставки

```
bool insert(const tree_data_type& data) {
    try {
        if (_file_for_tree.good() && _file_for_key_value.good()) {
            auto [path, info] = find_path(data.first);
            if (info.second) return false; // Ключ уже существует
            if (path.empty() || _position_root == 0) {
                btree_disk_node root(true);
                root.keys.push_back(data);
                root.size = 1;
                _count_of_node++;
                root.position_in_disk = _count_of_node;
                _position_root = root.position_in_disk;
```

```
                disk_write(root);
                write_metadata();
                return true;
            }
            auto [node_pos, idx] = path.top();
            auto node = disk_read(node_pos);
            insert_array(node, 0, data, idx);
            disk_write(node);
            if (node.size > maximum_keys_in_node) {
                auto split_path = path;
                split_node(split_path);
            }
            write_metadata();
            return true;
        }
        return false;
    } catch (const exception& e) {
        std::cerr << "Error in insert: " << e.what() << std::endl;
        return false;
    }
}
```

Удаление элементов

Удаление элемента включает следующие шаги:

1. Поиск узла с ключом с помощью `find_path`.
2. Если ключ найден в листовом узле, он удаляется с помощью `remove_array`.
3. Если ключ находится во внутреннем узле, он заменяется предшественником (максимальным ключом в левом поддереве), после чего предшественник удаляется.

4. После удаления вызывается `rebalance_node` для восстановления свойств B-дерева (например, объединение или перераспределение ключей между соседями).
5. Обновляются метаданные.

Листинг #23. Операция удаления

```
bool erase(const tkey& key) {
    try {
        auto [path, info] = find_path(key);
        size_t index = info.first;
        bool found = info.second;
        if (!found) return false;
        auto [node_pos, node_idx] = path.top();
        auto node = disk_read(node_pos);
        if (node._is_leaf) {
            auto new_node = remove_array(node, index, false);
            disk_write(new_node);
        } else {
            size_t child_pos = node.pointers[index];
            auto pred = disk_read(child_pos);
            while (!pred._is_leaf) {
                child_pos = pred.pointers[pred.size];
                pred = disk_read(child_pos);
            }
            auto pred_pair = pred.keys[pred.size - 1];
            node.keys[index] = pred_pair;
            disk_write(node);
            pred = remove_array(pred, pred.size - 1, false);
            disk_write(pred);
```

```
            path.push({child_pos, pred.size});
        }
        while (!path.empty()) {
            auto [pos, pos_index] = path.top();
            path.pop();
            auto curr = disk_read(pos);
            rebalance_node(path, curr, pos_index);
            disk_write(curr);
        }
        auto root = disk_read(_position_root);
        if (!root._is_leaf && root.size == 0) {
            _position_root = root.pointers[0];
        }
        disk_write(root);
        write_metadata();
        return true;
    } catch (const exception& e) {
        std::cerr << "Error in erase: " << e.what() << std::endl;
        return false;
    }
}
```

Поиск элементов

Поиск элемента выполняется с помощью `find_path`, возвращающего путь к узлу, содержащему ключ, или место, куда он мог бы быть вставлен. Если ключ найден, возвращается соответствующее значение, иначе — std::nullopt.

Листинг #24. Операция поиска

```
std::optional<tvalue> at(const tkey& key) {
    try {
```

```
        if (!_file_for_tree.is_open() || !_file_for_key_value.is_open()) {
            throw file_error("Files not open for reading");
        }
        auto [path, index_info] = find_path(key);
        auto [index, found] = index_info;
        if (!found || path.empty()) {
            return std::nullopt;
        }
        auto [pos, idx] = path.top();
        auto node = disk_read(pos);
        if (idx < node.keys.size()) {
            const auto& node_key = node.keys[idx].first;
            if (!compare_keys(key, node_key) && !compare_keys(node_key,
key)) {
                return node.keys[idx].second;
            }
        }
        return std::nullopt;
    } catch (const exception& e) {
        std::cerr << "Error in at(): " << e.what() << std::endl;
        return std::nullopt;
    }
}
```

Балансировка

B-дерево использует операции разделения (split_node) и ребалансировки (rebalance_node) для поддержания свойств сбалансированного дерева. Разделение узла происходит, когда он превышает максимальное количество ключей (2t-1), а ребалансировка — при удалении, если узел становится меньше минимального количества ключей (t-1). Эти операции минимизируют высоту дерева и количество дисковых операций.

Листинг #25. Операция разделения узла

```
void split_node(std::stack<std::pair<size_t, size_t>>& path) {
    if (path.empty()) {
        throw node_error("Cannot split node: path is empty");
    }
    auto [pos, index] = path.top();
    path.pop();
    auto node = disk_read(pos);
    if (node.keys.size() < 2*t-1) {
        throw node_error("Cannot split node: too few keys");
    }
    btree_disk_node new_node(node._is_leaf);
    new_node.size = t - 1;
    auto median_key = node.keys[t-1];
    new_node.keys.clear();
    for (size_t i = t; i < node.keys.size(); ++i) {
        new_node.keys.push_back(node.keys[i]);
    }
    if (!node._is_leaf) {
        new_node.pointers.clear();
        for (size_t i = t; i <= node.keys.size() && i <
node.pointers.size(); ++i) {
            new_node.pointers.push_back(node.pointers[i]);
```

```
    }
}
node.keys.resize(t - 1);
if (!node._is_leaf) {
    node.pointers.resize(t);
}
node.size = t - 1;
_count_of_node++;
new_node.position_in_disk = _count_of_node;
disk_write(node);
disk_write(new_node);
if (path.empty()) {
    btree_disk_node root_node(false);
    root_node.size = 1;
    root_node.keys.push_back(median_key);
    root_node.pointers.push_back(node.position_in_disk);
    root_node.pointers.push_back(new_node.position_in_disk);
    _count_of_node++;
    root_node.position_in_disk = _count_of_node;
    _position_root = root_node.position_in_disk;
    disk_write(root_node);
    write_metadata();
} else {
    auto [ppos, pindex] = path.top();
    auto parent = disk_read(ppos);
    if (pindex > parent.keys.size()) {
        pindex = parent.keys.size();
    }
```

```
        insert_array(parent, new_node.position_in_disk, median_key,
pindex);
        disk_write(parent);
    }
}
```

Система управления пользователями и санкциями

Основные компоненты реализации

Структуры данных

Система использует две основные структуры данных для хранения информации о пользователях и санкциях. Структура User хранит логин пользователя (до 6 символов) и хеш PIN-кода. Структура Sanction хранит имя пользователя, лимит команд и количество использованных команд в текущей сессии.

Листинг #26. Структуры User и Sanction

```
typedef struct {
    char login[7];          // Логин пользователя (до 6 символов)
    unsigned long pin_hash; // Хеш PIN-кода
} User;
typedef struct {
    char username[7];       // Имя пользователя
    int limit;              // Лимит команд
    int used;               // Количество использованных команд
} Sanction;
```

Система использует две основные структуры данных для хранения информации о пользователях и санкциях. Структура User хранит логин пользователя (до 6 символов) и хеш PIN-кода, созданный с использованием алгоритма FNV-1a. Структура Sanction хранит имя пользователя, лимит команд и количество использованных команд в текущей сессии.

Основной функционал программы

Программа реализует систему управления пользователями с поддержкой регистрации, авторизации, выполнения команд и наложения санкций (ограничений на количество команд). Данные хранятся в двух файлах: users.txt для пользователей и sanctions.txt для санкций. Основные операции включают работу с файлами, валидацию ввода, хеширование PIN-кода и управление лимитами команд.

Листинг #27. Основная структура программы

```
int main() {
    load_users();
    load_sanctions();
    main_menu();
    return 0;
}
```

Программа реализует систему управления пользователями с поддержкой регистрации, авторизации, выполнения команд и наложения санкций (ограничений на количество команд). Данные хранятся в двух файлах: users.txt для пользователей и sanctions.txt для санкций. Основные операции включают работу с файлами, валидацию ввода, хеширование PIN-кода и управление лимитами команд.

Программа использует глобальные массивы users и sanctions для хранения до 100 пользователей и 100 записей о санкциях соответственно. Глобальные переменные current_user и current_user_limit отслеживают текущего авторизованного пользователя и его лимит команд.

Для обеспечения корректности ввода реализованы функции валидации логина, PIN-кода и формата времени. Логин должен быть длиной от 1 до 6 символов и содержать только буквенно-цифровые символы. PIN-код должен быть в диапазоне от 0 до 1,000,000. Формат времени проверяется для соответствия шаблону DD:MM:YYYY_HH:MM:SS с учетом високосных годов.

Листинг #28. Функции валидации

```
int validate_login(const char *login) {
    size_t len = strlen(login);
    if (len == 0 || len > 6) return 0;
```

```
    for (int i = 0; i < len; i++) {
        if (!isalnum(login[i])) return 0;
    }
    return 1;
}

int validate_pin(long pin) {
    return (pin >= 0 && pin <= 1000000);
}

int validate_time_format(const char *time_str) {
    int day, month, year, hour, minute, second = 60;
    if (sscanf(time_str, "%d:%d:%d_%d:%d:%d", &day, &month, &year, &hour,
&minute, &second) != 6) {
        return 0;
    }
    if (day < 1 || day > 31 || month < 1 || month > 12 || year < 0 ||
        hour < 0 || hour > 23 || minute < 0 || minute > 59 || second < 0 ||
second > 59) {
        return 0;
    }
    int days_in_month[] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
    if (month == 2 && is_leap_year(year)) {
        days_in_month[1] = 29;
    }
    if (day > days_in_month[month - 1]) {
        return 0;
    }
    return 1;
```

```
}
```

Основные операции

Регистрация и авторизация

Регистрация пользователя (register_user) запрашивает логин и PIN-код, проверяет их валидность и сохраняет хеш PIN-кода в массиве users. Авторизация (authorize) проверяет соответствие введенного PIN-кода сохраненному хешу и устанавливает текущего пользователя.

Листинг #30. Операции регистрации и авторизации

```
void register_user() {
    char login[100];
    long pin;
    printf("Enter new login (up to 6 characters, letters and digits only): 
");
    scanf("%s", login);
    if (!validate_login(login)) {
        printf("Invalid login format! Must be 1-6 alphanumeric 
characters.\n");
        return;
    }
    if (find_user(login) != -1) {
        printf("User with this login already exists!\n");
        return;
    }
    printf("Enter PIN (0 to 1000000): ");
    if (scanf("%ld", &pin) != 1 || !validate_pin(pin)) {
        printf("Invalid PIN format!\n");
        return;
    }
    if (user_count >= MAX_USERS) {
```

```
        printf("User limit reached!\n");
        return;
    }
    strcpy(users[user_count].login, login);
    users[user_count].pin_hash = hash_pin(pin);
    user_count++;
    save_users();
    printf("User %s registered successfully!\n", login);
}

int authorize() {
    char login[100];
    long pin;
    printf("Enter login (up to 6 characters, letters and digits only): ");
    scanf("%s", login);
    if (!validate_login(login)) {
        printf("Invalid login format! Must be 1-6 alphanumeric
characters.\n");
        return 0;
    }
    int user_idx = find_user(login);
    if (user_idx == -1) {
        printf("User not found! Please register first.\n");
        return 0;
    }
    printf("Enter PIN (0 to 1000000): ");
    if (scanf("%ld", &pin) != 1 || !validate_pin(pin)) {
        printf("Invalid PIN format!\n");
```

```
        return 0;
    }
    if (users[user_idx].pin_hash != hash_pin(pin)) {
        printf("Incorrect PIN!\n");
        return 0;
    }
    strcpy(current_user, login);
    int idx = find_sanction(current_user);
    current_user_limit = (idx != -1) ? sanctions[idx].limit : -1;
    if (idx != -1) sanctions[idx].used = 0;
    printf("User %s logged in successfully!\n", current_user);
    return 1;
}
```

Команды времени и даты

Команды Time и Date выводят текущее время и дату соответственно, используя функции стандартной библиотеки C (time, localtime). Команда Howmuch вычисляет разницу между текущим временем и введенной датой в формате DD:MM:YYYY_HH:MM:SS, поддерживая флаги -s, -m, -h, -y для вывода в секундах, минутах, часах или годах.

Листинг #31. Команды Time, Date и Howmuch

```
void cmd_time() {
    time_t now = time(NULL);
    struct tm *tm = localtime(&now);
    printf("Current time: %02d:%02d:%02d\n", tm->tm_hour, tm->tm_min,
tm->tm_sec);
}


void cmd_date() {
    time_t now = time(NULL);
```

```
    struct tm *tm = localtime(&now);

    printf("Current date: %02d:%02d:%04d\n", tm->tm_mday, tm->tm_mon + 1,
tm->tm_year + 1900);

}


void cmd_howmuch(const char *time_str, const char *flag) {

    if (!validate_time_format(time_str)) {

        printf("Invalid time format! Use DD:MM:YYYY_HH:MM:SS\n");

        return;

    }

    if (!validate_flag(flag)) {

        printf("Invalid flag! Use -s, -m, -h, or -y\n");

        return;

    }

    int day, month, year, hour, minute, second;

    sscanf(time_str, "%d:%d:%d_%d:%d:%d", &day, &month, &year, &hour,
&minute, &second);

    struct tm input_time = {0};

    input_time.tm_mday = day;

    input_time.tm_mon = month - 1;

    input_time.tm_year = year - 1900;

    input_time.tm_hour = hour;

    input_time.tm_min = minute;

    input_time.tm_sec = second;

    time_t input = mktime(&input_time);

    time_t now = time(NULL);

    double diff = difftime(now, input);

    if (diff < 0) {

        printf("This is Future\n");
```

```
    } else if (strcmp(flag, "-s") == 0) {
        printf("Time difference: %.0f seconds\n", diff);
    } else if (strcmp(flag, "-m") == 0) {
        printf("Time difference: %.0f minutes\n", diff / 60);
    } else if (strcmp(flag, "-h") == 0) {
        printf("Time difference: %.0f hours\n", diff / 3600);
    } else if (strcmp(flag, "-y") == 0) {
        printf("Time difference: %.0f years\n", diff / (3600 * 24 * 365));
    }
}
```

Наложение санкций

Команда Sanctions позволяет администратору (с кодом подтверждения 12345) устанавливать лимит команд для пользователя. Лимит сохраняется в массиве sanctions и применяется к текущему пользователю, если он совпадает с указанным.

Листинг #32. Команда Sanctions

```
void cmd_sanctions(const char *username, int number) {
    if (!validate_login(username)) {
        printf("Invalid username format!\n");
        return;
    }
    if (find_user(username) == -1) {
        printf("User %s is not registered! Cannot apply sanctions.\n",
username);
        return;
    }
    if (number < 0) {
        printf("Number must be non-negative!\n");
        return;
```

```
    }
    printf("Enter confirmation code (12345): ");
    int code;
    if (scanf("%d", &code) != 1 || code != 12345) {
        printf("Invalid confirmation code!\n");
        return;
    }
    int idx = find_sanction(username);
    if (idx == -1) {
        if (sanction_count >= MAX_SANCTIONS) {
            printf("Sanction list is full!\n");
            return;
        }
        strcpy(sanctions[sanction_count].username, username);
        sanctions[sanction_count].limit = number;
        sanctions[sanction_count].used = 0;
        sanction_count++;
    } else {
        sanctions[idx].limit = number;
        sanctions[idx].used = 0;
    }
    if (strcmp(username, current_user) == 0) {
        current_user_limit = number;
    }
    save_sanctions();
    printf("Sanction applied successfully!\n");
}
```

Обработка команд

Функция process_commands реализует цикл обработки пользовательских команд (Time, Date, Howmuch, Sanctions, Logout). Она проверяет лимит команд для текущего пользователя и вызывает соответствующие функции для выполнения команд.

Листинг #33. Обработка команд

```
void process_commands() {
    char command[100];
    while (1) {
        if (current_user_limit != -1) {
            int idx = find_sanction(current_user);
            if (idx != -1) {
                sanctions[idx].used++;
                if (sanctions[idx].used > current_user_limit) {
                    printf("Command limit exceeded for this session!\n");
                    cmd_logout();
                    return;
                }
            }
        }
        printf("Enter command (Time, Date, Howmuch, Logout, Sanctions): ");
        scanf("%s", command);
        if (strcmp(command, "Time") == 0) {
            cmd_time();
        } else if (strcmp(command, "Date") == 0) {
            cmd_date();
        } else if (strcmp(command, "Howmuch") == 0) {
            char time_str[20], flag[3];
            scanf("%s %s", time_str, flag);
```

```
            cmd_howmuch(time_str, flag);
        } else if (strcmp(command, "Logout") == 0) {
            cmd_logout();
            return;
        } else if (strcmp(command, "Sanctions") == 0) {
            char username[7];
            int number;
            scanf("%s %d", username, &number);
            cmd_sanctions(username, number);
        } else {
            printf("Unknown command!\n");
        }
    }
}
```

Система обработки файлов

Основные компоненты реализации

Структура данных для результатов поиска

Система использует структуры `MatchResult` и `TextMatchResult` для хранения результатов операций поиска. `MatchResult` хранит позиции 32-битных значений, соответствующих заданной маске, а `TextMatchResult` хранит номера строк, содержащих искомый текстовый шаблон. Обе структуры используют динамические массивы с автоматическим расширением емкости.

Листинг #34. Структуры MatchResult и TextMatchResult

```
typedef struct {
    ll count;
    ll* positions;
    ll capacity;
} MatchResult;
```

```
typedef struct {
    ll count;
    ll* lines;
    ll capacity;
} TextMatchResult;
```

Основной функционал программы

Программа реализует консольную утилиту для обработки файлов с поддержкой четырех команд: `xor` (побитовое исключающее ИЛИ), `mask` (поиск по битовой маске), `copy` (создание копий файла) и `find` (поиск текста). Программа принимает список входных файлов и команду через аргументы командной строки, обрабатывая файлы с использованием процессов (fork) для параллельного выполнения задач.

Листинг #35. Основная структура программы

```
int main(int argc, char** argv) {
    return parsing(argc, argv);
}
```

Программа использует динамическое выделение памяти для массивов файловых потоков и результатов, а также функции работы с файлами и процессами из стандартной библиотеки C. Константы `CHUNK_SIZE` (2048 байт) и `INITIAL_CAP` (256) определяют размеры буферов и начальную емкость массивов.

Обработка побитового исключающего ИЛИ (XOR)

Команда `xorN` (где N от 2 до 6) выполняет побитовое исключающее ИЛИ над блоками данных указанного размера (2^N байт) из входного файла. Результат выводится в шестнадцатеричном формате. Если блок данных короче требуемого размера, он дополняется значением 0x16.

Листинг #36. Функция обработки XOR

```
int process_xor_block(FILE* stream, size_t bits, uint8_t* result_buffer) {
    size_t bytes_needed = (bits + 7) / 8;
```

```
    uint8_t temp[bytes_needed];
    memset(result_buffer, 0, bytes_needed);
   while (1) {
        ssize_t read_size = fread(temp, 1, bytes_needed, stream);
        if (read_size <= 0) break;
        if (read_size < bytes_needed) {
            memset(temp + read_size, 0x16, bytes_needed - read_size);
            read_size = bytes_needed;
        }
        for (size_t idx = 0; idx < read_size; idx++) {
            result_buffer[idx] ^= temp[idx];
        }
    }
    return feof(stream) ? 0 : -1;
}
```

Поиск по битовой маске

Команда `maskXXXXXXXX` (где XXXXXXXX — 8-значное шестнадцатеричное число) ищет 32-битные значения в файле, соответствующие заданной маске. Позиции совпадений сохраняются в структуре `MatchResult` и выводятся в консоль.

Листинг #37. Функция поиска по маске

```
MatchResult tally_masked_values(FILE* stream, uint32_t pattern) {
    MatchResult result = {0, NULL, 0};
    ll position = 0;
    uint32_t data;

    result.capacity = INITIAL_CAP;
    result.positions = malloc(result.capacity * sizeof(ll));
```

```
    if (!result.positions) return result;


    rewind(stream);
    while (fread(&data, sizeof(uint32_t), 1, stream)) {
        if ((data & pattern) == pattern) {
            if (result.count >= result.capacity) {
                result.capacity *= 2;
                ll* temp = realloc(result.positions, result.capacity *
sizeof(ll));
                if (!temp) {
                    free(result.positions);
                    result.positions = NULL;
                    return result;
                }
                result.positions = temp;
            }
            if (result.count == INT64_MAX) {
                result.count = -1;
                return result;
            }
            result.positions[result.count++] = position;
        }
        position++;
    }
    return result;
}
```

Создание копий файлов

Команда `copyN` (где N — число от 1 до 10) создает N копий каждого входного файла, используя процессы (fork) для параллельного копирования. Имена копий формируются путем добавления суффикса `-N` к имени исходного файла.

Листинг #38. Функция создания копий файлов

```
int spawn_file_duplicates(int num_copies, const char* src_path) {
    char* dot = strrchr(src_path, '.') ? strrchr(src_path, '.') : "";
    char* prefix = strdup(src_path);
    if (!prefix) return 0;
    if (dot) prefix[dot - src_path] = 0;
    pid_t* workers = (pid_t*)malloc(num_copies * sizeof(pid_t));
    if (!workers) {
        free(prefix);
        return 0;
    }
    for (int i = 0; i < num_copies; i++) {
        char new_path[CHUNK_SIZE];
        snprintf(new_path, CHUNK_SIZE, "%s-%d%s", prefix, i + 1, dot);
        pid_t child = fork();
        if (child < 0) {
            free(prefix);
            free(workers);
            return 0;
        }
        if (!child) {
            exit(replicate_file(src_path, new_path) ? 0 : 1);
        }
        workers[i] = child;
```

```
    }
    int all_good = 1;
    for (int i = 0; i < num_copies; i++) {
        int stat;
        waitpid(workers[i], &stat, 0);
        all_good &= WIFEXITED(stat) && !WEXITSTATUS(stat);
    }
    free(prefix);
    free(workers);
    return all_good;
}
```

Поиск текста

Команда `find<шаблон>` выполняет поиск строк, содержащих указанный текстовый шаблон, в каждом входном файле. Поиск выполняется в отдельных процессах для каждого файла, а результаты (номера строк) сохраняются в структуре `TextMatchResult`.

Листинг #39. Функция поиска текста

```
TextMatchResult locate_text(FILE* source, const char* needle) {
    TextMatchResult result = {0, NULL, 0};
    int line_index = 0;
    char* current;

    result.capacity = INITIAL_CAP;
    result.lines = malloc(result.capacity * sizeof(ll));
    if (!result.lines) return result;

    rewind(source);
    while ((current = fetch_line(source))) {
```

```
        line_index++;
        if (strstr(current, needle)) {
            if (result.count >= result.capacity) {
                result.capacity *= 2;
                ll* temp = realloc(result.lines, result.capacity *
sizeof(ll));
                if (!temp) {
                    free(result.lines);
                    result.lines = NULL;
                    free(current);
                    return result;
                }
                result.lines = temp;
            }
            result.lines[result.count++] = line_index;
        }
        free(current);
    }
    return result;
}
```

Обработка аргументов командной строки

Функция `parsing` разбирает аргументы командной строки, открывает входные файлы и вызывает соответствующую функцию обработки в зависимости от команды (`xor`, `mask`, `copy`, `find`). Ошибки ввода-вывода, памяти или синтаксиса обрабатываются с выводом сообщений в stderr.

Листинг #40. Функция разбора аргументов

```
int parsing(int arg_count, char** args) {
    if (arg_count < 3) {
        fprintf(stderr, "Need: %s files... command\n", args[0]);
```

```
    return 1;
}

int stream_count = arg_count - 2;
FILE** streams = (FILE**)malloc(stream_count * sizeof(FILE*));
if (!streams) {
    fprintf(stderr, "Out of memory\n");
    return 1;
}

for (int i = 0; i < stream_count; i++) {
    streams[i] = fopen(args[i + 1], "r");
    if (!streams[i]) {
        fprintf(stderr, "Cannot access %s\n", args[i + 1]);
        shutdown_streams(streams, i);
        free(streams);
        return 1;
    }
}

const char* cmd = args[arg_count - 1];
int result = 0;

if (!strncmp(cmd, "xor", 3)) {
    if (strlen(cmd) != 4 || cmd[3] < '2' || cmd[3] > '6') {
        fprintf(stderr, "XOR size must be 2-6\n");
        shutdown_streams(streams, stream_count);
        free(streams);
```

```
            return 1;
        }
        size_t bit_width = 1 << (cmd[3] - '0');
        for (int i = 0; i < stream_count; i++) {
            uint8_t* xor_result = (uint8_t*)malloc((bit_width + 7) / 8);
            if (!xor_result) {
                fprintf(stderr, "Memory error\n");
                shutdown_streams(streams, stream_count);
                free(streams);
                return 1;
            }
            if (process_xor_block(streams[i], bit_width, xor_result) < 0) {
                fprintf(stderr, "Failed to process %s\n", args[i + 1]);
                result = 1;
            } else {
                printf("%s XOR: ", args[i + 1]);
                for (size_t j = 0; j < (bit_width + 7) / 8; j++) {
                    printf("%02x ", xor_result[j]);
                }
                printf("\n");
            }
            free(xor_result);
        }
    } else if (!strncmp(cmd, "mask", 4)) {
        // Обработка команды mask (см. Листинг #37)
    } else if (!strncmp(cmd, "copy", 4)) {
        // Обработка команды copy (см. Листинг #38)
    } else if (!strncmp(cmd, "find", 4)) {
```

```
        // Обработка команды find (см. Листинг #39)
    } else {
        fprintf(stderr, "Unrecognized command: %s\n", cmd);
        shutdown_streams(streams, stream_count);
        free(streams);
        return 1;
    }
    shutdown_streams(streams, stream_count);
    free(streams);
    return result;
}
```

Эта реализация обеспечивает эффективную обработку файлов с использованием параллельных процессов, динамического управления памятью и поддержки различных операций, включая побитовую обработку, поиск по маске и тексту, а также копирование файлов.

Система отображения содержимого директорий

Основные компоненты реализации

Функция отображения информации о файле

Программа предоставляет подробную информацию о файлах в указанной директории, включая тип файла, права доступа, владельца, группу, размер, время последней модификации и адрес на диске (номер инода). Функция `print_file_info` использует структуру `stat` для получения метаданных файла и форматирует вывод в стиле команды `ls -l`.

Листинг #41. Функция отображения информации о файле

```
void print_file_info(const int dir_fd, const char *filename) {
    struct stat file_stat;
    if (fstatat(dir_fd, filename, &file_stat, AT_SYMLINK_NOFOLLOW) < 0) {
        perror("fstatat");
        return;
```

```
}
char type;
if (S_ISREG(file_stat.st_mode)) type = '-'; // Regular file
else if (S_ISDIR(file_stat.st_mode)) type = 'd'; // Directory
else if (S_ISLNK(file_stat.st_mode)) type = 'l'; // Link
else if (S_ISCHR(file_stat.st_mode)) type = 'c'; // Character device
else if (S_ISBLK(file_stat.st_mode)) type = 'b'; // Block device
else if (S_ISFIFO(file_stat.st_mode)) type = 'p'; // FIFO (named pipe)
else if (S_ISSOCK(file_stat.st_mode)) type = 's'; // Socket
else type = '?';
char permissions[12];
snprintf(permissions, sizeof(permissions), "%c %c%c%c%c%c%c%c%c%c",
    type,
    (file_stat.st_mode & S_IRUSR) ? 'r' : '-',
    (file_stat.st_mode & S_IWUSR) ? 'w' : '-',
    (file_stat.st_mode & S_IXUSR) ? 'x' : '-',
    (file_stat.st_mode & S_IRGRP) ? 'r' : '-',
    (file_stat.st_mode & S_IWGRP) ? 'w' : '-',
    (file_stat.st_mode & S_IXGRP) ? 'x' : '-',
    (file_stat.st_mode & S_IROTH) ? 'r' : '-',
    (file_stat.st_mode & S_IWOTH) ? 'w' : '-',
    (file_stat.st_mode & S_IXOTH) ? 'x' : '-');
const struct passwd *pwd = getpwuid(file_stat.st_uid);
const struct group *grp = getgrgid(file_stat.st_gid);
char time_buf[64];
const struct tm *tm_info = localtime(&file_stat.st_mtime);
strftime(time_buf, sizeof(time_buf), "%b %d %H:%M", tm_info);
printf("%s %2lu %-8s %-8s %8lld %s %s\n",
```

```
        permissions,
        file_stat.st_nlink,
        pwd ? pwd->pw_name : "unknown",
        grp ? grp->gr_name : "unknown",
        (long long)file_stat.st_size,
        time_buf,
        filename);
    printf("  First disk address: %llu\n", (unsigned long
long)file_stat.st_ino);
}
```

Обработка директорий

Функция `list_directory` открывает указанную директорию, читает её содержимое и вызывает `print_file_info` для каждого файла, кроме специальных записей "." и "..". Использование дескриптора директории (`dirfd`) позволяет безопасно получать метаданные файлов с помощью `fstatat`.

Листинг #43. Функция обработки директорий

```
void list_directory(const char *dirname) {
    DIR *dir;
    struct dirent *entry;
    printf("%s:\n", dirname);
    if ((dir = opendir(dirname)) == NULL) {
        perror("opendir");
        return;
    }
    const int dir_fd = dirfd(dir);
    if (dir_fd == -1) {
        perror("dirfd");
        closedir(dir);
        return;
```

```
    }
    while ((entry = readdir(dir)) != NULL) {
        if (strcmp(entry->d_name, ".") == 0 || strcmp(entry->d_name, "..")
== 0)
            continue;
        print_file_info(dir_fd, entry->d_name);
    }
    closedir(dir);
    printf("\n");
}
```

Вывод

В процессе разработки была создана модульная библиотека системного уровня, сочетающая в себе ключевые компоненты, востребованные в современных высоконагруженных и встраиваемых системах. В неё вошли: высокопроизводительный клиентский логгер с настраиваемым уровнем детализации, масштабируемые серверные приложения для обработки параллельных запросов, специализированные аллокаторы памяти с контролем фрагментации и адаптивные B-деревья для быстрого поиска и модификации данных.

Каждый модуль библиотеки ориентирован на минимизацию накладных расходов и строгое управление ресурсами: реализованы принципы RAII, использованы умные указатели и семантика перемещения, что позволило достичь высокой надёжности и предотвратить утечки памяти. Дополнительно проработаны механизмы обработки ошибок и валидации входных данных, что делает библиотеку устойчивой к сбоям и применимой в критических системах.

Практическая значимость проекта заключается в его применимости для реальных задач системного уровня: от разработки серверных решений до встроенных платформ и баз данных. Библиотека демонстрирует современные подходы к проектированию системного ПО и может служить как основой для интеграции в производственные системы, так и учебным примером для изучения принципов безопасной и эффективной разработки на C++20.

Список использованных источников

1. Быстрое преобразование Фурье [Электронный ресурс] // Algorithmica. — URL: https://ru.algorithmica.org/cs/algebra/fft/ (дата обращения: 23.05.2025).

2. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн. Алгоритмы: построение и анализ, 3-е издание = Introduction to Algorithms, Third Edition. — М.: «Вильямс», 2013. — 1328 с. — ISBN 978-5-8459-1794-2.

3. Кнут, Д. Э. Искусство программирования, том 2: Полученные алгоритмы / Д. Э. Кнут. — М.: Вильямс, 2007. — 784 с. — ISBN 978-5-8459-0086-9.

4. Седжвик, Р. Алгоритмы на C++, части 1-4: основы, структуры данных, сортировка, поиск / Р. Седжвик. — 4-е изд. — М.: Диалектика, 2017. — 960 с. — ISBN 978-5-8459-2055-3.

Приложение А Репозиторий с исходным кодом

FIIT_FA_SP [Электронный ресурс]. // github — URL: https://github.com/DmitriyKolesnikM8O/FIIT_FA_SP (дата обращения: 23.05.2025).

SP [Электронный ресурс]. // github — URL: https://github.com/DmitriyKolesnikM8O/SP (дата обращения: 23.05.2025).