МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Московский Авиационный Институт

(Национальный исследовательский университет)

Институт №8

«Компьютерные науки и прикладная математика»

Кафедра 806

«Вычислительная математика и программирование»

Курсовая работа по дисциплине «Операционные системы и архитектура компьютеров»

Студент:

Группа: М8О-211Б-22

Преподаватель: Романенков А. М.

Оценка:

Дата:

Москва 2024

Содержание

Введение

В данной курсовой работе было реализовано приложение, позволяющие выполнять операции над коллекциями данных типа <tkey, tvalue> и контекстами их хранения.

Коллекция данных описывается набором строковых параметров:

Листинг 1. <Набор строковых параметров>

- название пула схем данных, хранящего схемы данных;
- название схемы данных, хранящей коллекции данных;
- название коллекции данных.

Коллекции данных, схемы данных и пулы схем данных представляют собой ассоциативный контейнер вида B-tree, в котором каждый объект данных соответствует некоторому уникальному ключу. Для ассоциативного контейнера отдельно вынесена интерфейсное часть. Взаимодействие с коллекцией объектов происходит посредством выполнения одной из операций над ней:

Листинг 2. <Операции над коллекцией объектов>

- добавление новой записи по ключу;
- чтение записи по ее ключу;
- чтение набора записей с ключами из диапазона; [*minbound*... *maxbound*]
- обновление данных для записи по ключу;
- удаление существующей записи по ключу.

Во время работы приложения возможно выполнение также следующих операций:

Листинг 3. <Дополнительные операции над коллекцией объектов>

- добавление/удаление пулов данных;
- добавление/удаление схем данных для заданного пула данных;
- добавление/удаление коллекций данных для заданной схемы данных заданного пула данных.

Поток команд, выполняемых в рамках работы приложения, поступает из файла, путь к которому подаётся в качестве аргумента командной строки.

Приложение должно работать в двух режимах:

Листинг 4. <Режимы работы приложения>

- структуры и данные размещаются в оперативной памяти приложения (in-memory cache);
- структуры и данные размещаются в файловой системе.

Дополнительно был реализован функционал:

Листинг 5. <Дополнительный функционал>

1. Интерактивный диалог с пользователем, выполнение команд адресованные пользователем. Ввод реализован с помощью консоли и файлов с потоками команд.
2. Механизм персистентности, позволяющий выполнять запросы к данным в рамках коллекции данных на заданный момент времени (дата и время, для которых нужно вернуть актуальную версию данных, передается как параметр).
3. Хранение объектов строк, размещенных в объектах данных, обеспечено на основе структурного паттерна проектирования “Приспособленец”.
4. Механизмы сохранения состояния системы хранения данных в файловую систему и восстановления состояния системы хранения данных из файловой системы для режима in-memory cache.
5. Возможность кастомизации аллокаторов для размещения объектов данных: глобальные операторы new и delete; первый + лучший + худший подходящий + освобождение в отсортированном списке; первый + лучший + худший подходящий + освобождение с дескрипторами границ, первый + лучший + худший подходящий + система двойников; первый + лучший + худший подходящий + аллокатор на красно-черном дереве.
6. Приложения в виде сервера, запросы на который поступают из клиентских приложений. Взаимодействие клиентских приложений с серверным реализовано посредством средств сетевого взаимодействия.
7. Механизм сердцебиения (heartbeat) для проверки работоспособности сервера. При отсутствии ответа на heartbeat-запрос, целевой сервер подвергнут перезапуску.
8. Механизм асинхронной обработки запросов (формат GUID v4).
9. Серверное приложение, собирающее логи клиентской (и, если есть, серверной) части приложения в файловые потоки вывода. Конфигурирование JSON.

Вид структуры проекта:

Рисунок 1. <Структура проекта>

Основная часть

Пункт [0]

Алгоритм решения задания

Листинг 6. <Шаги алгоритма>

1. Определить структуры данных для представления пулов данных, схем данных и коллекций данных.
2. Реализовать абстрактный класс для ассоциативного контейнера в виде B-tree и его интерфейсные методы для добавления, чтения, обновления и удаления записей.
3. Создать классы для пулов данных, схем данных и коллекций данных, использующие ассоциативные контейнеры.
4. Реализовать операции добавления/удаления пулов данных, схем данных и коллекций данных.
5. Написать парсер для чтения команд из файла и выполнения соответствующих операций.
6. Разработать два режима работы приложения: в оперативной памяти и в файловой системе.

Использованные средства языка C++

Листинг 7. <Средства языка>

- Классы и наследование для организации структур данных.
- Шаблоны классов для абстрактного класса ассоциативного контейнера.
- Умные указатели (например, std::unique_ptr) для управления памятью.
- Файловый ввод/вывод для работы с командами из файла.
- STL контейнеры (например, std::map) для реализации ассоциативных контейнеров.

Алгоритмы и структуры данных

Листинг 8. <Алгоритмы и структуры данных>

- B-дерево для хранения данных пулов, схем и коллекций.
- Алгоритмы добавления, чтения, обновления и удаления записей в B-дереве.
- Парсинг команд из файла для выполнения соответствующих операций.

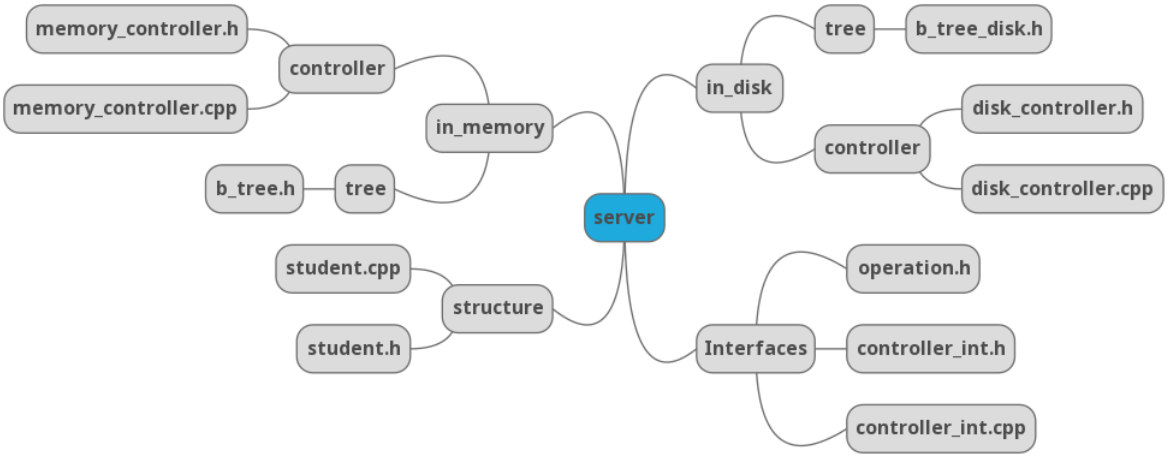

Рисунок 2. <Организация B-tree, интерфейса, структуры данных и контроллера памяти>

Пункт [1]

Алгоритм решения задания

Для реализации интерактивного режима работы программы и обработки команд, вводимых пользователем, а также выполнения команд из файла, необходимо выполнить следующие шаги:

Листинг 9. <Шаги алгоритма>

1. Разработка интерфейса для ввода команд:
    a. Функция, которая будет обрабатывать команды, введенные пользователем.
    b. Чтение команд из файла и их выполнение.
2. Обработка и выполнение команд:
    a. Реализовать обработку различных команд.
    b. В зависимости от введенной команды выполнять соответствующие действия.
3. Вывод результат выполнения команд:
    a. Выводить результат выполнения на экран.

Использованные средства языка C++

Листинг 10. <Средства языка>

- **Регулярные выражения (`std::regex`)** для распознавания команд.
- **Библиотека для HTTP-запросов (`httplib`)** для взаимодействия с сервером.
- **Обработка JSON (`nlohmann::json`)** для работы с данными.
- **Потоки ввода-вывода (`iostream`)** для взаимодействия с пользователем и файлами.
- **Опциональные значения (`std::optional`)** для работы с возможными ответами от сервера.

Алгоритмы и структуры данных

Листинг 11. <Алгоритмы и структуры данных>

- **Строки (`std::string`)** для хранения команд и аргументов.
- **Контейнеры STL** (`std::vector`, `std::map`) для хранения и управления данными.
- **Функции обратного вызова** для асинхронной обработки ответов от сервера.
- **Паттерн проектирования "Приспособленец"** для эффективного хранения строковых объектов.

Основные функции

Листинг 12. <Основные функции>

1. **std::optional<std::string> add_pool(const std::string &pool_name)**
    a. Отправляет запрос на сервер для добавления нового пула данных с указанным именем.
2. **std::optional<std::string> remove_pool(const std::string &pool_name)**
    a. Отправляет запрос на сервер для удаления пула данных с указанным именем.
3. **std::optional<std::string> add_scheme(const std::string &pool_name, const std::string &scheme_name)**
    a. Отправляет запрос на сервер для добавления новой схемы данных в указанный пул.
4. **std::optional<std::string> remove_scheme(const std::string &pool_name, const std::string &scheme_name)**
    a. Отправляет запрос на сервер для удаления схемы данных из указанного пула.
5. **std::optional<std::string> add_collection(const std::string &pool_name, const std::string &scheme_name, const std::string &collection_name)**
    a. Отправляет запрос на сервер для добавления новой коллекции данных в указанную схему.
6. **std::optional<std::string> remove_collection(const std::string &pool_name, const std::string &scheme_name, const std::string &collection_name)**
    a. Отправляет запрос на сервер для удаления коллекции данных из указанной схемы.
7. **std::optional<std::string> insert(const std::string &pool_name, const std::string &scheme_name, const std::string &collection_name, const nlohmann::json &student)**
    a. Отправляет запрос на сервер для вставки нового элемента данных (например, информации о студенте) в указанную коллекцию.
8. **std::optional<std::string> read_value(const std::string &pool_name, const std::string &scheme_name, const std::string &collection_name, const std::string &key, bool need_persist, std::time_t time)**
    a. Отправляет запрос на сервер для чтения значения данных из указанной коллекции по ключу.
9. **std::optional<std::string> read_range(const std::string &pool_name, const std::string &scheme_name, const std::string &collection_name, const std::string &key1, const std::string &key2, bool need_persist, std::time_t time)**
    a. Отправляет запрос на сервер для чтения диапазона значений данных из указанной коллекции между двумя ключами.

10. **void start_heartbeat()**
    a. Запускает поток для периодической проверки состояния сервера с помощью "heartbeat".
11. **void stop_heartbeat()**
    a. Останавливает поток "heartbeat".
12. **void execute_command(const std::string &command)**
    a. Выполняет команду, введенную пользователем, распознает команду и вызывает соответствующий метод клиента.
13. **void run()**
    a. Запускает интерактивный режим работы с клиентом, принимая команды от пользователя через стандартный ввод.
14. **void run_from_file(const std::string &filename)**
    a. Запускает клиент и выполняет команды, считанные из файла.

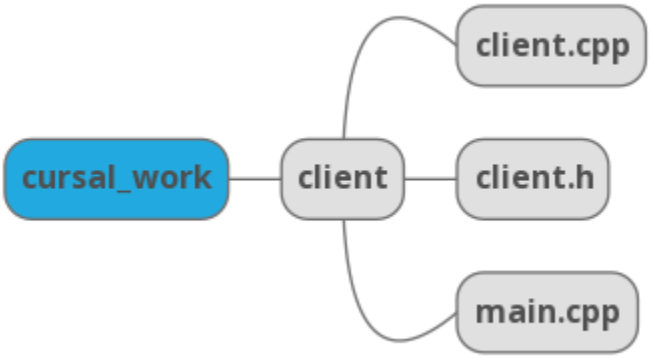

Рисунок 3. <Структура реализации работы с пользователем>

Пункт [2]

Алгоритм решения задания

Механизм персистентности, позволяющий выполнять запросы к данным в рамках коллекции данных на заданный момент времени (дата и время, для которых нужно вернуть актуальную версию данных, передается как параметр).

Листинг 13. <Шаги алгоритма>

1. **Реализация механизма персистентности:**
    a. Механизм персистентности реализован с использованием двух методов: `redo_all()` и `revert_to()`.
    b. Метод `redo_all()` выполняет все операции, которые были помечены как отмененные.
    c. Метод `revert_to()` отменяет все операции, выполненные после указанного момента времени.
2. **Использование поведенческих паттернов:**
    a. Паттерн "Команда" используется для реализации операций, которые изменяют состояние данных.
    b. Паттерн "Цепочка обязанностей" используется в методах `read_value()` и `read_range()` для обработки запросов к данным.

Использованные средства языка C++

Листинг 14. <Средства языка>

- Шаблоны классов (`template`)
- Стандартные контейнеры и асинхронное выполнение задач с помощью библиотеки `Boost`.

Алгоритмы и структуры данных

Листинг 15. <Алгоритмы и структуры данных>

- `B-tree`
- Асинхронное выполнение задач из библиотеки `boost`

Основные функции

Листинг 16. <Основные функции>

- `redo_all()`: Перевыполняет все операции, которые были помечены как отмененные.
- `revert_to()`: Отменяет все операции, выполненные после указанного момента времени.
- `read_value()`: Читает значение данных по ключу из указанной коллекции данных на заданный момент времени.
- `read_range()`: Читает диапазон данных из указанной коллекции данных на заданный момент времени.

Пункт [4]

Алгоритм решения задания

Хранение объектов строк, размещенных в объектах данных, на основе структурного паттерна проектирования “Приспособленец”.

Листинг 17. <Шаги алгоритма>

1. **Интерфейс**
    a. В строковом пуле реализовать метод добавления новой строки и поиска уже существующей.
    b. В классе строк реализовать методы для сериализации и десериализации строк.
2. **Сериализация**
    a. Реализовать метод `serialize` для записи строк в файл без дублирования.
3. **Десериализация**
    a. Реализовать метод для чтения строк из файла и их восстановления.

Использованные средства языка C++

Листинг 18. <Средства языка>

- **Классы и объекты:** для реализации структур данных и паттернов проектирования.
- **STL контейнеры:** для хранения уникальных строк в строковом пуле.
- **Указатели и ссылки:** для управления памятью.
- **Многопоточность:** для обеспечения потокобезопасности при работе со строковым пулом.
- **Файловый ввод/вывод:** для сериализации и десериализации данных.

Алгоритмы и структуры данных

Листинг 19. <Алгоритмы и структуры данных>

- **Паттерн "Приспособленец":** для хранения уникальных строк без дублирования.
- **Паттерн "Одиночка":** для создания единственного экземпляра строкового пула.
- **Алгоритмы поиска и вставки:** для работы со строковым пулом.
- **Мьютексы:** для обеспечения потокобезопасности.

Основные функции

Листинг 20. <Основные функции>

1. **Сериализация строки:**
    a. `serialize(std::fstream &stream)`: записывает строку в файл, если она еще не была записана.
2. **Десериализация строки:**
    a. `deserialize(std::fstream &stream)`: считывает строку из файла.

Пункт [5]

Алгоритм решения задания

Для режима in-memory cache реализуйте механизмы сохранения состояния системы хранения данных в файловую систему и восстановления состояния системы хранения данных из файловой системы.

Листинг 21. <Шаги алгоритма>

1. **Сохранение состояния в файловую систему**:
    a. При завершении работы или по расписанию создается резервная копия текущего состояния данных.
    b. Все данные сериализуются и сохраняются в файл на диске.
2. **Восстановление состояния из файловой системы**:
    a. При запуске системы проверяется наличие файла с резервной копией.
    b. Если файл существует, данные из него десериализуются и восстанавливаются в память.

Использованные средства языка C++

Листинг 22. <Средства языка>

- **STL (Standard Template Library)** для структур данных и управления памятью.
- **Boost Library** для работы с асинхронными задачами и будущими (futures).
- **nlohmann::json** для сериализации и десериализации данных.
- **std::filesystem** для работы с файловой системой (в случае если доступен стандарт C++17 и выше).

Алгоритмы и структуры данных

Листинг 23. <Алгоритмы и структуры данных>

- **B-дерево** для хранения данных, позволяющее эффективно управлять большими объемами информации.
- **Асинхронная обработка** с использованием Boost Futures для выполнения операций ввода-вывода без блокировки основного потока.
- **Мьютексы и блокировки** для синхронизации доступа к данным и обеспечения безопасности потоков.

Основные функции

Листинг 24. <Основные функции>

- **`save_state_to_file`** - Сериализует текущее состояние данных и сохраняет его в файл.
- **`load_state_from_file`** - Загружает данные из файла и восстанавливает состояние системы.

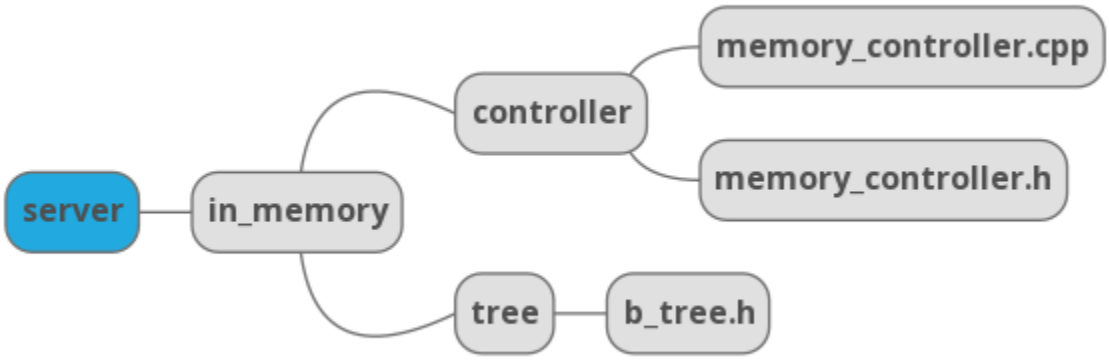

Рисунок 4. <Структура реализации работы с пользователем>

Пункт [7]

Алгоритм решения задания

Возможность кастомизации (для заданной коллекции данных) аллокаторов для размещения объектов данных: глобальные операторы new и delete; первый + лучший + худший подходящий + освобождение в отсортированном списке; первый + лучший + худший подходящий + освобождение с дескрипторами границ, первый + лучший + худший подходящий + система двойников; первый + лучший + худший подходящий + аллокатор на красно-черном дереве.

Листинг 25. <Шаги алгоритма>

1. **Глобальные операторы `new` и `delete`:**
2. Определить перегруженные глобальные операторы `new` и `delete` для выделения и освобождения памяти для объектов коллекции данных.
3. **Рассортированный список:**
4. Реализовать класс `sorted_list_allocator`, который будет управлять выделением и освобождением памяти с использованием рассортированного списка.
5. **Освобождение с дескрипторами границ:**
6. Создать класс `boundary_tags_allocator`, который будет использовать дескрипторы границ для управления памятью.
7. **Система двойников:**
8. Реализовать класс `buddies_system_allocator`, который будет осуществлять выделение и освобождение памяти с помощью системы двойников.
9. **Аллокатор на красно-черном дереве:**
10. Создать класс `red_black_tree_allocator`, который будет использовать красно-чёрное дерево для управления памятью.

Использованные средства языка C++

Листинг 26. <Средства языка>

- Шаблоны классов для обобщенной реализации аллокаторов.
- Перегрузка операторов `new` и `delete` для глобальных аллокаторов.
- Стандартные контейнеры для реализации отсортированного списка, дескрипторов границ и красно-чёрного дерева.
- Указатели для работы с динамической памятью.

Алгоритмы и структуры данных

Листинг 27. <Алгоритмы и структуры данных>

- Рассортированный список.

- Дескрипторы границ.
- Система двойников.
- Красно-чёрное дерево.
- Глобальная-куча

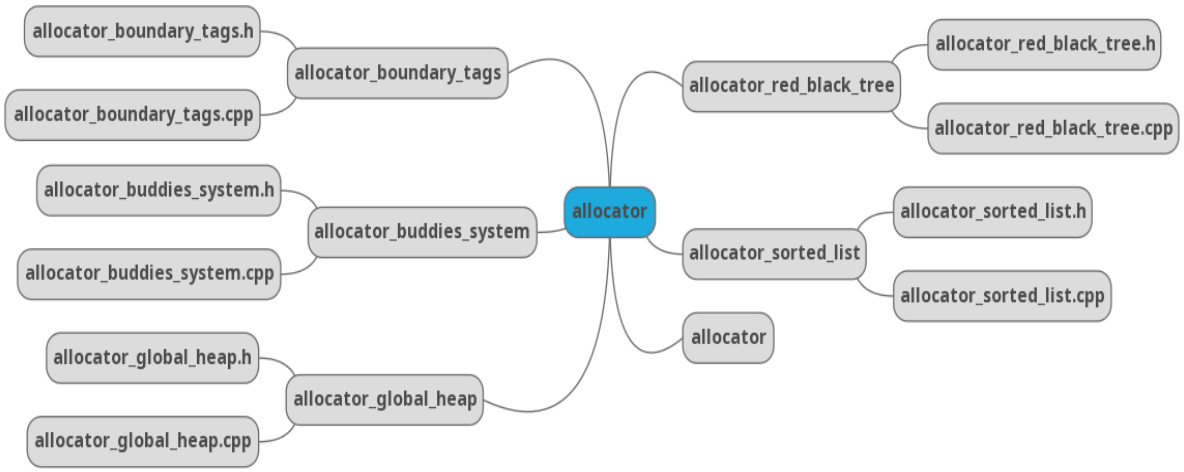

Рисунок 5. <Семейство аллокаторов>

Пункт [8]

Алгоритм решения задания

Для реализации сервера, который обрабатывает запросы от клиентских приложений, используются средства сетевого взаимодействия, такие как библиотека Crow для создания веб-сервера и библиотека nlohmann::json для работы с JSON.

Листинг 28. <Шаги алгоритма>

1. **Создание сервера**:
    a. Использовать библиотеку `Crow` для создания веб-сервера.
    b. Определить маршруты (`routes`) для обработки различных запросов.
2. **Обработка запросов**:
    a. Для каждого маршрута реализовать соответствующие функции для обработки запросов (добавление/удаление пулов, схем, коллекций, вставка, чтение, обновление, удаление данных).
    b. Взаимодействие с контроллером для выполнения операций над данными.
3. **Форматирование ответов**:
    a. Формировать ответы в формате `JSON` для возврата клиенту.

Использованные средства языка C++

Листинг 29. <Средства языка>

- **`Crow`**: легковесная библиотека для создания `HTTP`-сервера.
- **`nlohmann::json`**: библиотека для работы с `JSON`.
- **`STL (Standard Template Library)`**: стандартная библиотека `C++` для контейнеров и алгоритмов.
- **`std::chrono`**: библиотека для работы с временем.

Алгоритмы и структуры данных

Листинг 30. <Алгоритмы и структуры данных>

- ***Контейнеры `STL`:*** *используются для хранения и управления данными.*
- ***`JSON`:*** *используется для обмена данными между сервером и клиентами.*
- ***Механизмы многопоточности:*** *для обеспечения конкурентного доступа к серверу.*

Основные функции

Листинг 31. <Основные функции>

- *add_pool: добавляет новый пул.*
- *remove_pool: удаляет пул.*
- *add_scheme: добавляет новую схему в пул.*
- *remove_scheme: удаляет схему из пула.*
- *add_collection: добавляет новую коллекцию в схему.*
- *remove_collection: удаляет коллекцию из схемы.*
- *insert: вставляет данные в коллекцию.*
- *read_value: читает значение по ключу.*
- *read_range: читает диапазон значений.*
- *update: обновляет данные в коллекции.*
- *remove: удаляет данные из коллекции.*
- *get: получает данные по GUID.*

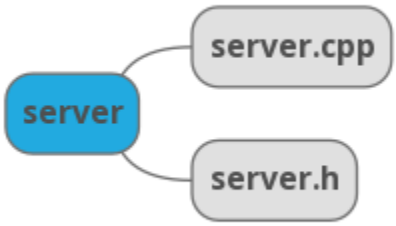

Рисунок 6. <Реализация сервера>

Пункт [10]

Алгоритм решения задания

Механизм сердцебиения (heartbeat) для проверки работоспособности сервера. При отсутствии ответа на heartbeat-запрос, целевой сервер должен быть подвергнут перезапуску.

Листинг 32. <Шаги алгоритма>

1. **Создание механизма сердцебиения (heartbeat):**
    a. Запуск серверного процесса.
    b. Проверка состояния сервера с регулярным интервалом.
    c. Перезапуск сервера при отсутствии ответа на запрос /heart.
2. **Использование многопоточности:**
    a. Запуск мониторинга сервера в отдельном потоке для непрерывного контроля.
3. **Обработка запросов:**
    a. Проверка состояния сервера через HTTP-запрос.
    b. Перезапуск сервера при необходимости.

Использованные средства языка C++

Листинг 33. <Средства языка>

- **Crow:** библиотека для создания HTTP-сервера.
- **nlohmann::json:** библиотека для работы с JSON.
- **std::thread:** стандартная библиотека C++ для многопоточности.
- **httplib:** легковесная HTTP-библиотека для проверки состояния сервера.
- **process:** механизм для запуска и остановки процессов (реализуйте самостоятельно).

Алгоритмы и структуры данных

Листинг 34. <Алгоритмы и структуры данных>

- ***Многопоточность:*** *для непрерывного мониторинга состояния сервера.*
- ***HTTP-запросы:*** *для проверки состояния сервера.*
- ***Обработка процессов:*** *для управления серверным процессом (запуск и остановка).*

Основные функции

Листинг 35. <Основные функции>

- `start()`: запуск мониторинга сервера.
- `stop()`: остановка мониторинга и сервера.
- `monitor_our_server()`: мониторинг состояния сервера.
- `check_server()`: проверка состояния сервера.
- `restart_server()`: перезапуск сервера при отсутствии ответа на `heartbeat`-запрос.

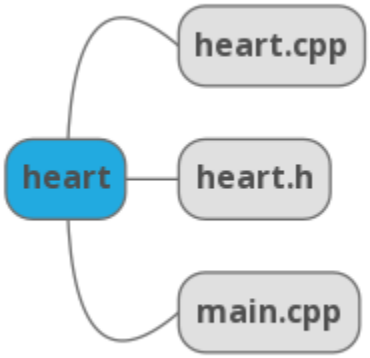

Рисунок 7. <Реализация задания heartbeat>

Пункт [13]

Алгоритм решения задания

Механизм асинхронной обработки запросов (результатом запроса на выполнение операции должен являться идентификатор запроса (формат GUID v4).

Листинг 36. <Шаги алгоритма>

1. **Асинхронные операции**:
    a. Использование библиотеки `Boost` для реализации асинхронных задач с помощью `boost::async`.
    b. Создание асинхронной задачи, которая выполняет операцию (например, добавление пула, чтение данных и т.д.).
    c. Использование идентификатора (`GUID`) для каждой операции, чтобы отслеживать ее состояние и результаты.
2. **Хранение результатов**:
    a. Использование `std::unordered_map` для хранения результатов операций по их идентификаторам.
    b. Обеспечение потокобезопасности с помощью `std::mutex` для управления доступом к карте результатов.
3. **Получение результатов**:
    a. Метод `get`, который по идентификатору возвращает результат операции, если он доступен.

Использованные средства языка C++

Листинг 37. <Средства языка>

- **`Boost.Asio`**: Для реализации асинхронных операций.
- **`std::unordered_map`**: Для хранения результатов операций.
- **`std::mutex`** и **`std::lock_guard`**: Для обеспечения потокобезопасности.
- **`nlohmann::json`**: Для работы с `JSON`-данными.

Алгоритмы и структуры данных

Листинг 38. <Алгоритмы и структуры данных>

- **B-дерево**: Используется для хранения данных в памяти.
- **GUID (UUID v4)**: Для уникальной идентификации операций.
- **Асинхронные задачи**: Для выполнения операций без блокировки основного потока.

Основные функции

Листинг 39. <Основные функции>

- `add_pool`: Асинхронное добавление нового пула.
- `remove_pool`: Асинхронное удаление пула.
- `add_scheme`: Асинхронное добавление новой схемы.
- `remove_scheme`: Асинхронное удаление схемы.
- `add_collection`: Асинхронное добавление новой коллекции.
- `remove_collection`: Асинхронное удаление коллекции.
- `insert`: Асинхронная вставка данных.
- `read_value`: Асинхронное чтение значения по ключу.
- `read_range`: Асинхронное чтение диапазона значений.
- `update`: Асинхронное обновление данных.
- `remove`: Асинхронное удаление данных.
- `get`: Получение результатов операции по идентификатору.

Пункт [15]

Алгоритм решения задания

Серверное приложение, собирающее логи клиентской (и, если есть, серверной) части приложения в файловые потоки вывода. Конфигурирование серверного логгера на основе файла со структурой JSON.

Листинг 40. <Шаги алгоритма>

1. **Создание серверного приложения для сбора логов**:
   a. Разработка механизма сбора логов из клиентской и, если применимо, серверной части приложения.
   b. Конфигурирование логгера на основе файла со структурой JSON.
2. **Использование библиотеки httplib**:
   a. Реализация сервера для обработки HTTP-запросов.
   b. Взаимодействие с клиентскими приложениями через HTTP-запросы.
3. **Обработка запросов**:
   a. Инициализация и отправка логов на сервер.
   b. Перехват и обработка запросов на сервере.
4. **Логгирование в файлы и консоль**:
   a. Конфигурирование серверного логгера для вывода логов в файловые потоки и/или на консоль.

Использованные средства языка C++

Листинг 41. <Средства языка>

- **httplib**: библиотека для создания HTTP-сервера и клиента.
- **nlohmann::json**: библиотека для работы с JSON.
- **std::unordered_map**: контейнер для хранения потоков вывода логов.
- **std::stringstream**: для форматирования логов перед отправкой на сервер.

Алгоритмы и структуры данных

Листинг 42. <Алгоритмы и структуры данных>

- **JSON**: использование структуры JSON для конфигурации логгера.
- **unordered_map**: хранение информации о потоках вывода логов с учетом их уровня серьезности.

Основные функции

Листинг 43. <Алгоритмы и структуры данных>

- `add_file_stream()`: добавление файла для вывода логов определенного уровня серьезности.
- `add_console_stream()`: добавление консольного вывода логов определенного уровня серьезности.
- `transform_with_configuration()`: преобразование конфигурации из JSON-файла в параметры логгера.
- `clear()`: очистка конфигурации логгера.
- `build()`: создание объекта логгера на основе конфигурации.

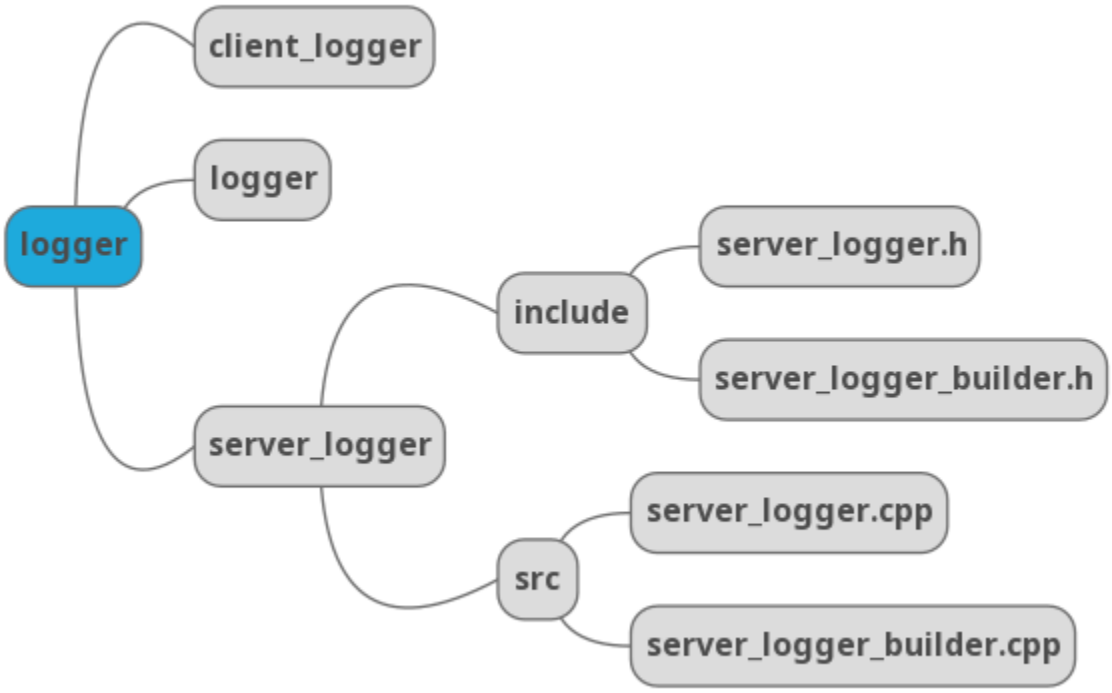

Рисунок 8. <Структура server_logger>

Вывод

В ходе написания программы использовались алгоритмы для реализации B-tree [1], а также алгоритмы для размещения памяти [2]. Например, методы best_fit, worst_fit, для следующих аллокаторов: allocator_red_black_tree, allocator_boundary_tags. Проектирование проекта основывалось на поведенческих паттернах: Chain of Responsibility, Command, порождающих паттернах: Builder, Singleton[3]. Для справочной информации по языку C++ использовался [4]. В качестве основы для написания серверного взаимодействия данные брались из [5].

Ключевые моменты приложения:

Листинг 44. <Ключевые моменты приложения>

```
1. B-tree построенное на диске - b_tree_disk.h
2. Контроллер управления памятью на диске - disk_controller.h
3. Все изученные аллокаторы - dir::allocator
4. Контроллер управления оперативной памятью - memmory_contoller.h
5. B-tree - b_tree.h
6. Сервер и интерфейс - server.cpp, server.h
7. Структра базы данных - student.h
```

В итоге, для решения поставленной задачи использовался весь материал пройденный за оба семестра. На практике было применено множество концепций проектирования, алгоритмов, знаний по межпроцессному и межсерверному взаимодействию, отточен навык написания код на языке C++.

Знания в области программирования были углублены, а навыки в разработке кода значительно усовершенствовались.

Список использованных источников

1. *Кормен*, Томас Х., Лейзерсон, Чарльз И., Ривест, Рональд Л., Штайн,. Клиффорд. В24. *Алгоритмы*: построение и анализ, 4-е издание.
2. Дональд Кнут - «Искусство программирования. Том 1» .
3. Приемы объектно-ориентированного проектирования. Паттерны проектирования (2016) - GOF. Авторы: Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидес.
4. Скотт Мейерс - Эффективный и современный C++.
5. Таненбаум Эндрю С, Бос Херберт - Modern Operating Systems .
6. Андрей Александреску - Современное проектирование на C++.pdf

Приложения

[1]https://github.com/yashelter/Empty_CW/