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: Шведов А.И.

Группа: 8О-210Б

Преподаватель: А.М. Романенков

Москва, 2025

Содержание

Введение

Системное программирование — это ключевая область информатики, сосредоточенная на создании программ, которые взаимодействуют с аппаратным обеспечением и операционными системами. Актуальность этой дисциплины объясняется необходимостью эффективного управления памятью, реализации многопоточных приложений, синхронизации процессов и оптимального использования системных ресурсов. В ходе курсовой работы были выполнены лабораторные задания, затрагивающие основные аспекты системного программирования: разработку сложных структур данных, создание аллокаторов памяти, решение типичных задач синхронизации и проектирование клиент-серверных приложений с применением механизмов межпроцессного взаимодействия (IPC).

Выполненные задачи способствовали углублению знаний в области работы с динамической памятью, многопоточностью, файловыми системами и сетевыми протоколами, а также подчеркнули важность правильной обработки ошибок и проверки данных для создания надежного программного обеспечения.

Теоретическая часть

Примитивная оболочка для командной строки

Были реализованы следующие возможности:

- Time - запрос текущего времени в стандартном формате чч:мм:сс;
- Date - запрос текущей даты в стандартном формате дд:мм:гггг;
- Howmuch flag - запрос прошедшего времени с указанной даты в параметре , параметр flag определяет тип представления результата (-s в секундах, -m в минутах, -h в часах, -y в годах)
- Logout - выйти в меню авторизации
- Sanctions username - команда позволяет ввести ограничения на работу с оболочкой для пользователя username, а именно данный пользователь не может в одном сеансе выполнить более запросов. Для подтверждения ограничений после ввода команды необходимо ввести значение 12345.

Для работы со временем использовалась структура struct tm и функции из библиотеки time.h. Для обработки ошибок использовался enum, обработка команд происходила в цикле.

Консольное приложение для обработки файлов

Реализованная система представляет собой набор консольных утилит, выполняющих операции над файлами. Архитектура системы разделена на два ключевых модуля:

1. Модуль побитовых операций

- Для каждого файла вычисляется XOR-сумма блоков заданного размера .
- Подсчитывает количество четырёхбайтовых целых чисел в файле, соответствующих заданной шестнадцатеричной маске.

2. Модуль управления файлами

- Создание N копий файла выполняется в отдельных процессах.
- Поиск подстроки в файле реализован через алгоритм Кнута-Морриса-Пратта.

Решение проблемы в задаче обедающих философов

Задача была решена при помощи семафоров. Одновременно могут есть максимум два философа (так как вилок 5). Когда философ начинает есть, он опускает семафор, когда заканчивает есть - поднимает.

Решение задачи про волка, козу и капусту с помощью клиент-серверного комплекса приложений

Задача была решена посредством очереди сообщений. Сервер и клиент подключаются к заранее созданному файлу очереди, и обмениваются сообщениями по существу задачи: ‘take’, ‘put’, ‘move’. Если последовательность команд приводит к нарушению условий задачи, сессия завершается.

Решение задачи про университетскую ванную комнату

Задача была решена при помощи мьютекса и двух переменных, одна из которых отвечала за количество человек в комнате, а другая за то, кто в комнате находится (‘EMPTY’, ‘WOMAN_ONLY’, ‘MAN_ONLY’). Мьютекс использовался чтобы избежать data-race при работе с этими переменными.

Клиент-серверная система анализа и группировки файловых путей через сокеты

Задача была решена посредством TCP-сокетов. Сервер слушает на определенном адресе и порту, задающиеся через аргументы командной строки. Клиент отправляет length-prefixed сообщения, содержащие пути к директориям. Сервер отвечает таким же length-prefixed сообщением с листингом. Если директории не существует, отправляется сообщение об ошибки.

Собственная версия программы ls

Реализованная программа представляет собой консольную утилиту, аналогичную команде ls в Unix-подобных системах, с расширенным выводом метаданных файлов. Её основная задача — предоставление структурированной информации о содержимом директорий, включая технические детали файлов и каталогов.

Программа состоит из двух ключевых модулей:

- Модуль обхода директорий: Отвечает за открытие, чтение и закрытие каталогов.

- Модуль сбора метаданных: Извлекает и форматирует информацию о каждом файле.

Серверный логгер

Понятие и назначение

Серверный логгер — это программный компонент, который централизованно фиксирует события, происходящие в распределённых системах или серверных приложениях.

Способы логирования

Серверный логгер поддерживает разнообразные каналы вывода, адаптированные под высоконагруженные среды:

- Стандартные потоки (stdout/stderr): используются для мгновенного отображения состояния сервера в консоли, что удобно при отладке.
- Файловые системы: логи сохраняются на диск с поддержкой ротации (например, по времени или размеру) и архивирования для долгосрочного хранения.
- Сетевые протоколы: интеграция с централизованными системами через syslog, HTTP, gRPC или специализированные сервисы (Elasticsearch, Grafana Loki).

Уровни серьёзности

Для фильтрации потока данных серверный логгер использует иерархию уровней:

- trace: детальная техническая информация (например, состояние потоков или тайминги запросов).
- debug: данные для отладки (параметры запросов, промежуточные результаты).
- info: штатные события (успешные подключения, завершение задач).
- warning: нештатные, но некритичные ситуации (повторные попытки запросов, частичные сбои).
- error: ошибки, нарушающие работу компонентов (отказ БД, недоступность сервиса).

- fatal: критические сбои, требующие немедленного вмешательства (аварийное завершение сервера).

Каналы вывода настраиваются на прием сообщений от определенного уровня.

Global Heap Allocator

Является оберткой над стандартным аллокатором, доступным через *::operator new* и *::operator delete*. Реализация делегирует все вызовы в стандартную библиотеку.

Sorted List Allocator

Свободные блоки хранятся в односвязном списке. В метаданных свободного блока хранится размер и указатель на следующий блок. В метаданных занятого блока хранится размер и указатель на начало аллокатора (_trusted_memory), предназначенный для проверки при освобождении, принадлежит ли блок аллокатору. Когда программа запрашивает выделение памяти, аллокатор извлекает блок из списка свободных блоков и возвращает его в качестве доступной памяти (первый подходящий, наиболее подходящий, наименее подходящий). Блок делится на два, один из которых остается свободным и возвращается в список, а другой становится занятым. Если при делении оставшейся памяти слишком мало, то разделения блоков не происходит. При освобождении освобожденный блок помещается в список свободных блоков. Аллокатор может сливать или объединять соседние свободные блоки в один более крупный блок.

Одной из проблем с использованием списка свободных блоков является фрагментация. Если программа часто запрашивает блоки разного размера, то по мере времени список свободных блоков может заполниться маленькими фрагментами, что затруднит выделение больших блоков памяти.

Boundary Tag Allocator

Аллокатор рассматривает доверенную область памяти как череду занятых блоков и свободных промежутков-«дыр». В самом начале области хранится лишь один указатель на первый занятый блок; остальные занятые блоки связаны между собой двусторонними ссылками, образуя последовательность, через которую можно проходить от меньших адресов к большим и обратно.

Свободный промежуток определяется не собственными метаданными, а только расстоянием между концом одного занятого блока и началом следующего.

Когда поступает запрос на выделение, алгоритм линейным обходом проходит по занятым блокам и вычисляет размеры встречающихся промежутков. Первый, лучший или худший подходящий промежуток выбирается в зависимости от выбранной стратегии поиска. Если подходящий интервал найден, в его начале размещается новый заголовок, формируется занятый блок, а ссылки соседних блоков перестраиваются так, чтобы включить его в цепочку. Если хвост интервала после размещения оказался слишком мал, он полностью присоединяется к новому блоку, чтобы не оставлять неиспользуемых крошек.

Освобождение блока сводится к тому, что его заголовок удаляется из цепочки: предыдущий и следующий занятые блоки перенастраивают свои ссылки так, будто освобождённого блока не существовало. Поскольку свободные промежутки никак не описываются, два соседних освобождённых участка автоматически образуют единый более крупный промежуток, и дополнительная операция слияния не нужна.

В результате алгоритм тратит постоянное время на освобождение и линейное время на поиск места для нового блока, при этом не расходует память на заголовки свободных участков и устраняет фрагментацию метаданных. Внешняя фрагментация остаётся возможной, но исчезает сразу после того, как оба соседних блока окажутся освобождены.

Buddies System Allocator

Алгоритм двойников начинается с единого участка памяти размером 2^K байт. Любой выделенный или свободный фрагмент рассматривается как блок степени двойки. В его заголовке хранятся два поля: признак занятости и степень блока.

При запросе памяти требуемый объем вместе с заголовком округляется вверх до ближайшей степени двойки, достаточной для размещения. Последовательность блоков просматривается линейным проходом, пока не встретится свободный участок подходящего размера в соответствии с выбранной стратегией («первый подходящий», «наиболее подходящий» или «наименее подходящий»). Если найденный блок оказывается слишком велик, он делится пополам; степень новых блоков уменьшается на единицу, и деление продолжается, пока меньший блок не удовлетворит запрос. Один из полученных двойников помечается как занятый и возвращается вызывающему коду, а второй остается свободным и воспринимается алгоритмом как самостоятельный блок более низкого порядка.

Освобождение выполняется симметрично. Блок помечается как свободный, после чего по XOR-правилу смещений вычисляется адрес его двойника той же степени за O(1). Если двойник также свободен, оба блока сливаются в блок со степенью на единицу выше. Процесс повторяется, пока не встретится занятый двойник или не восстановится исходный корневой блок.

Поиск подходящего места выполняется за O(n), где n пропорционально числу блоков и ограничен логарифмом общего объёма памяти. Разделение при выделении и последовательное объединение при освобождении выполняются за O(log M), где M — общий объем памяти. Внутренняя фрагментация не превышает половину размера любого выданного блока, поскольку объем всегда округляется вверх до ближайшей степени двойки, а внешняя фрагментация исчезает сразу после первого же успешного объединения соседних свободных участков.

Red-Black Tree Allocator

КЧ-аллокатор хранит все занятые и свободные блоки в одном двусвязном списке. Свободные блоки дополнительно помещаются в красно-чёрное дерево, где ключом служит их текущий размер. В метаданных аллокатора есть корень дерева, стратегия поиска (первый, лучший или худший подходящий) и мьютекс, который защищает все изменения. Каждый свободный блок знает своих левый и правый детей, цвет и родителя; занятый блок вместо родителя хранит ссылку на доверенную область, что позволяет проверить принадлежность при освобождении.

Когда приходит запрос на выделение, к требуемому объёму прибавляется размер заголовка, после чего дерево просматривается согласно заданной стратегии. При стратегии “наиболее подходящий” алгоритм спускается по левым ветвям, пока не найдёт наименьший свободный блок, достаточный для запроса; при “наименее подходящем” он идёт вправо, выбирая самый крупный свободный блок; при “первом подходящем” дерево обходится в естественном порядке, и берётся первый встретившийся подходящий участок. Найденный узел удаляется из дерева за логарифмическое время, помечается занятым и при необходимости делится: если после заголовка и полезных данных остается место хотя бы под новый заголовок, хвост превращается в свободный блок и вставляется обратно в дерево с восстановлением инвариантов красно-чёрного баланса. Все изменения размера отражаются только в структуре дерева, поэтому сам заголовок свободного участка не перемещается.

Освобождение начинается с проверки «своего» указателя на доверенную область; после этого блок помечается свободным и пытается слиться с соседями в

цепочке. Если предыдущий или следующий блок уже свободен, они удаляются из дерева, звенья цепочки объединяются, а размеры суммируются. Итоговый расширенный участок вставляется в дерево снова, и процедура балансировки восстанавливает цвета и высоту за O(log n). Поскольку объединение выполняется сразу, внешняя фрагментация исчезает, как только освободятся прилегающие блоки.

Поиск свободного участка и вставка–удаление в дереве выполняются за O(log n), где n — число свободных блоков; операции на цепочке требуют постоянного времени. Внутренняя фрагментация ограничена хвостом, меньшим минимального заголовка, а распределение по размеру через дерево позволяет уменьшить вероятность крупных «дыр» без необходимости линейного обхода памяти.

Реализация класса *B*-дерева

B-дерево — это абсолютно сбалансированное дерево поиска, которое используется для хранения и управления большими объемами данных на внешней памяти (например, на жестких дисках или SSD). Оно позволяет эффективно выполнять операции вставки, удаления и поиска данных. Основным его преимуществом перед бинарным деревом поиска является сокращение количества обращений к диску до $O(\log_t n)$, в связи с чем эта структура данных активно используется в различных СУБД.

Каждый узел B-дерева может содержать несколько ключей и несколько дочерних узлов (в отличие от бинарного дерева, где каждый узел имеет не более двух дочерних). Количество ключей в каждом узле ограничено заранее заданным порядком t, который определяет максимальное количество ключей в узле. Каждый узел может содержать от $t-1$ до $2t-1$ ключей.

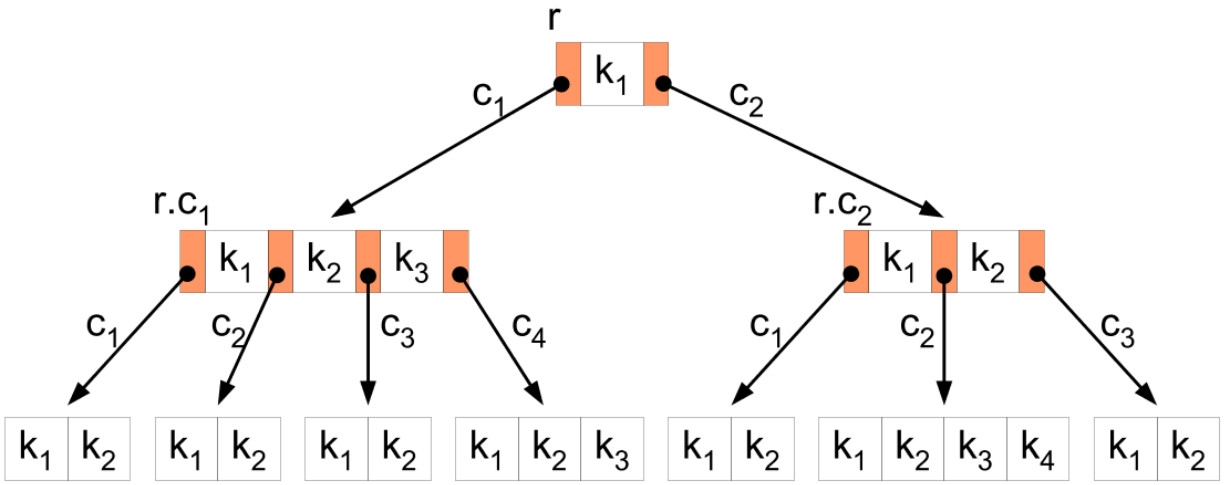

Рисунок 1. Пример B-дерева.

Операции в B-дереве

Поиск. Процесс поиска в B-дереве аналогичен поиску в бинарном дереве поиска, но с учетом того, что каждый узел может иметь несколько ключей.На каждом уровне дерева происходит бинарный поиск по ключам, после чего осуществляется переход к соответствующему дочернему узлу.

Асимптотическая сложность:

Лучший случай: O(1)

Средний случай: $O(\log_t n)$

Худший случай: $O(\log_t n)$

Вставка:

1. Если узел, в который вставляется новый ключ, не переполнен, просто добавляется новый ключ в соответствующий узел.
2. Если узел переполнен (содержит 2t−12t-12t−1 ключей), происходит разделение узла: ключи разделяются между родительским узлом и дочерними узлами. Ключ в “середине” узла перемещается наверх, в родителя, а ключи слева и справа от него становятся двумя новыми узлами.
3. Вставка также может вызвать переполнение в родительском узле, поэтому балансировка продолжается вплоть до корня (проверка количества узлов и разделение при необходимости.

Асимптотическая сложность:

Лучший случай: O(1)

Средний случай: $O(t\log_t n)$

Худший случай: $O(t\log_t n)$

Удаление:

1. Ключ удаляется из узла.
2. Если в узле осталось достаточно ключей (свойства B-дерева не нарушены), то операция завершается, иначе переходим к следующему шагу.
3. Если у одного из братьев количество ключей строго больше, чем t-1, то из этого брата ключ перемещается в родителя, а из родителя ключ перемещается на место удаленного.
4. Если нету братьев, подходящих под предыдущий пункт, то происходит слияние двух братьев. Ключ из родителя “опускается” вниз, после чего ключи двух узлов объединяются в один узел. Так как после этого свойства B-дерева могут нарушиться в родителе, то балансировка продолжается до корня (проверка на выполнение свойств B-дерева и слияние при необходимости).

Асимптотическая сложность:

Лучший случай: O(1)

Средний случай: $O(t\log_t n)$

Худший случай: $O(t\log_t n)$

Реализация класса *B* +-дерева

B+-дерево — это модификация B-дерева, которая также представляет собой абсолютно сбалансированную структуру данных для эффективного хранения и поиска больших объемов данных. Основные изменения в B+-дереве касаются организации хранения данных, что делает его более удобным для использования в системах управления базами данных и других приложениях, где важна эффективность поиска и последовательного обхода данных.

Основные отличия B+-дерева от обычного B-дерева:

1. Хранение данных: В B+-дереве все данные хранятся только в листьях дерева. Внутренние узлы содержат только ключи, которые используются для навигации по дереву, но не для хранения самих данных. В связи с этим может происходить дублирование ключей, когда в правом сыне узла содержится такой же ключ, как и в родителе.

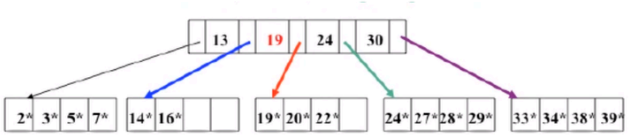

Рисунок 2. B+-дерево

2. В B-дереве листья могут быть связаны между собой, но это не является обязательным. Листья могут быть организованы в произвольном порядке.В B+ дереве листья всегда связаны между собой в связанный список (обычно двусторонний). Это позволяет эффективно выполнять последовательный обход всех элементов в дереве.

Операции в B+-дереве.

Поиск в B+-дереве выполняется так же, как и в обычном B-дереве, с учетом того, что данные находятся только в листьях.

Диапазонный запрос. В B+ дереве благодаря связанным листьям можно легко пройти по всему диапазону данных, используя простой последовательный обход листьев, что делает диапазонный поиск очень эффективным.

Вставка в B+ дерево происходит аналогично вставке в B-дерево, но с дополнительной особенностью: данные вставляются только в листья. Помимо этого, после вставки нужно обновить связанный список листов.

Удаление в B+ дереве аналогично удалению в B-дереве, но также с учетом того, что данные находятся только в листьях.

Преимущества B+ дерева:

1. Эффективность диапазонных запросов: Связанные листья позволяют легко и быстро выполнять диапазонные запросы (например, выборку данных в пределах определенного диапазона).

2. Лучше использует кэш: Поскольку данные находятся только в листьях и они организованы в связанный список, это улучшает работу с кэшированием и последовательным доступом.
3. Более эффективная индексация: В СУБД и файловых системах, где индексы часто используются для быстрого поиска, B+ дерево обеспечивает большую гибкость за счет упрощенной структуры хранения данных.

Реализация класса *B*-дерева, сохраняющее данные в файловом хранилище

Операции реализуются так же, как и в обычном B-дереве. В структуре узла дерева появляются дополнительные поля, необходимые для корректной работы с диском. Помимо этого, само B-дерево может существовать независимо от того выполняется программа или нет, то есть при вызове деструктора B-дерево сохраняется на диск и может быть загружено при необходимости. Более подробно отличия реализации B-дереве на диске от реализации обычного B-дерева рассмотрены в практической части.

Практическая часть

Примитивная оболочка для командной строки

Реализация задачи представляет собой простую систему для управления пользователями с функциями регистрации, логина, санкций и управления временем. Она использует структуры данных, такие как вектор, для хранения информации о пользователях, и работает с файлами для загрузки и сохранения данных. Основные компоненты перечислены ниже.

Классы и структуры:

- user: структура, которая хранит данные о пользователе (логин, пин-код, санкции).
- vector: абстракция динамического массива, реализующая хранение и управление пользователями.

Основные функции:

- Registrate: регистрирует нового пользователя, проверяет уникальность логина и валидность пин-кода.
- Login: процесс логина с проверкой пин-кода.
- Sanctions: накладывает санкции на пользователя, если введен правильный код подтверждения. Функция представлена в приложении А, листинг А.1.
- Time/Date/HowMuch: операции для получения времени, даты или вычисления прошедшего времени с указанного момента.
- ProccessCommands: обработка команд пользователя в интерактивном режиме.

Использованные средства языка:

- Функции работы с строками и файлами (например, fscanf, fopen, fclose).
- Проверка ошибок с использованием кодов ошибок и выводом сообщений об ошибках.
- Использование динамической памяти для обработки строк и временных данных.
- Библиотека “time.h” для работы со временем (функции mktime, difftime и т.д.)

При обработке любой команды ввод разбирается на аргументы и валидируется, при неверном вводе пользователю выводится соответствующее сообщение об ошибке. Проверка времени производится с помощью стандартной библиотеки для работы с датой и временем (например, time.h).

Программа работает в интерактивном режиме, позволяя пользователю зарегистрироваться, войти в систему, наложить санкции или проверить время, а также загружать и сохранять информацию о пользователях в файл.

Фрагмент алгоритма приведен в приложении А, листинг А.1.

Консольное приложение для обработки файлов

Программа реализует консольную утилиту для обработки файлов с поддержкой многозадачности и валидации входных данных. Ниже приведено детальное описание её работы.

Основные функции:

- main: Обработка аргументов (см. приложение Б, листинг Б.1), открытие файлов, выбор операции.
- xor_blocks, count_masked_numbers, copy_file, find_string: Реализация логики операций.

Взаимодействие функций:

- main: получает информацию из аргументов командной строки и в зависимости от флага передает управление программы нужной функции, которая выполняет основную логику работы.
- xor_blocks, count_masked_numbers, copy_file, find_string: каждая функция выполняет свою работу с переданными в аргументы командной строки адресами и выводит в консоль отформатированный ввод, если это необходимо.

Решение проблемы в задаче обедающих философов

Программа моделирует поведение философов, которые поочередно берут вилки (ресурсы), едят и затем возвращают их.

Программа использует системные семафоры для синхронизации действий философов. Каждый философ должен взять два соседних ресурса (вилки), прежде чем начать есть. Семафоры защищают эти ресурсы от одновременного использования несколькими философами.

В коде создаются шесть семафоров. Семафор 0 (для общего контроля): ограничивает количество философов, которые могут быть активными одновременно. Семафоры 1–5: каждый из этих семафоров управляет состоянием

вилки (от 0 до 5). Изначально каждый семафор установлен в значение 1, что означает, что все вилки доступны.

В главной функции создается несколько процессов с использованием fork(). Каждый процесс представляет собой философа. Философы пытаются взять соседние вилки (с помощью операций semop для изменения значений семафоров), едят, а затем возвращают вилки, освобождая их для других философов. Семафор 0 управляет количеством активных философов, позволяя не более двух философов действовать одновременно. Каждый философ выполняет цикл из 3 попыток взять вилки, поесть и вернуть вилки, с использованием задержки (sleep(1)), чтобы смоделировать время еды.

Алгоритм взаимодействия философов:

1. Каждый философ пытается взять вилки, расположенные слева и справа.
2. Если оба соседа не используют вилки, философ может начать есть.
3. После еды философ возвращает вилки на место, чтобы другие философы могли их использовать.
4. Семафор 0 контролирует количество философов (максимум двое), которые могут есть одновременно, избегая проблем с дедлоками.

Фрагмент алгоритма приведен в приложении В, листинг В.1.

Решение задачи про волка, козу и капусту с помощью клиент-серверного комплекса приложений

В решении задействованы два независимых исполняемых модуля — «клиент» и «сервер» — оба написаны на C и обмениваются данными через очереди сообщений System V. Каждый модуль подключает стандартные заголовки stdbool.h, stdio.h, stdlib.h, string.h и sys/ipc.h / sys/msg.h; сервер дополнительно использует unistd.h.

Клиент объявляет макросы SEND и RECV — обёртки вокруг msgsnd и msgrcv с проверкой кода ошибки. Командный файл читается функцией getline; строки разбираются strtok-ом, после чего формируется динамический буфер message с malloc / realloc. Для каждой команды клиент отправляет подготовленную структуру на очередь сервера, ждёт ответ и выводит сообщения об ошибках через perror или fprintf. Очередь для обратной связи создается вызовом msgget с IPC_PRIVATE, а идентификатор сервера вычисляется через ftok

и повторный msgget. По завершении работы клиент закрывает IPC-ресурс через msgctl с IPC_RMID и освобождает выделенную память.

Сервер содержит макрос LOG для вывода отладочной информации с номером клиентской очереди и описывает структуру session — односвязный список, хранящий идентификатор очереди и текущее положение каждого объекта. Управление списком реализовано функциями register_client, unregister_client и destroy_sessions; память под узлы выделяется и освобождается вручную. Обёртки send_error, send_ok, send_all_moved используют msgsnd для унифицированной отправки ответов. Логика обработки запросов вынесена в handle–функции, каждая из которых получает сообщение, проверяет корректность состояния, модифицирует поля session и отсылает подтверждение или ошибку. Главный цикл блокирующим msgrcv принимает сообщения, затем переключателем switch вызывает нужный обработчик; после завершения приложения очередь удаляется вызовом msgctl с IPC_RMID.

Фрагмент алгоритма приведен в приложении Г, листинг Г.1.

Решение задачи про университетскую ванную комнату

Для моделирования задачи используется многозадачность и синхронизация с помощью мьютекса и атомарных переменных.

Основные компоненты:

- pthread_mutex_t mutex — мьютекс для защиты общего состояния.
- _Atomic(int) cur_state — текущее состояние комнаты. (EMPTY, MAN_ONLY, WOMAN_ONLY)
- _Atomic(int) current_number — количество людей в комнате.

Основные функции:

- woman_wants_to_enter и man_wants_to_enter — проверяют возможность входа, если комната пуста или в ней только люди одного пола и есть свободные места.
- woman_leave и man_leave — управляют выходом, сбрасывая состояние комнаты, если в ней больше никого нет.
- manage_gender — потоки обрабатывают вход и выход в зависимости от пола, а затем "спят" случайное время.

Алгоритм работы:

1. Поток пытается войти в комнату, если в комнате пусто или только его пол и есть свободные места.
2. После входа, программа выводит сообщение ("Woman entered" или "Man entered").
3. Человек через случайное время покидает комнату, выводя сообщение о выходе ("Woman leaved" или "Man leaved").
4. Когда в комнате больше никого нет, состояние сбрасывается в EMPTY.
5. Потоки синхронизируются с помощью мьютекса, атомарные операции обеспечивают безопасное обновление данных.

   Фрагмент алгоритма синхронизации приведен в приложении Д, листинг Д.1.

Клиент-серверная система анализа и группировки файловых путей через сокеты

Код состоит из двух программ на C: серверной и клиентской. Обе используют POSIX API для имитации команды ls по сети.

Сервер создаёт TCP-сокет, настраивает SO_REUSEADDR через setsockopt, связывает его с заданным адресом и портом вызовами bind и listen, а затем в цикле принимает соединения функцией accept. После установления соединения он читает сообщения фиксированного префикса: сначала восемь сетевых байт с длиной пути, потом сам путь через recv с флагом MSG_WAITALL. Путь проверяется realpath и stat; если это абсолютный каталог, содержимое каталога перечисляется функциями opendir, readdir, closedir, формируется динамический буфер через realloc и отправляется клиенту. Отправка идёт двумя write: сначала длина ответа переводится в сетевой порядок htobe64, затем передаётся сам текст. Все операции завершаются выводом диагностик через макрос LOG и perror. Связь с клиентом закрывается вызовом close; основной сокет удаляется тем же вызовом в обработчике ошибок или при завершении.

Клиент после создания TCP-сокета вызывает connect, затем в цикле читает строки из stdin с getline. Перед отправкой каждая строка очищается от символа новой строки, её длина кодируется htobe64 и передаётся двумя send (заголовок и данные). Клиент получает ответ аналогичным двуходовым протоколом: сначала восемь байт размера be64toh, потом сам текст, который при необходимости дописывается в буфер realloc, и выводит его на stdout. По окончании работы

клиент корректно завершает соединение через close и освобождает всю динамическую память.

В программе активно используются макросы-обёртки LOG и cleanup, статические буферы со sprintf/snprintf, функции преобразования порядка байт из endian.h, системные вызовы socket, setsockopt, bind, listen, accept, connect, send, recv, write, close и файловые вызовы opendir, readdir, closedir, stat. Такой минимальный набор средств языка C и системных вызовов демонстрирует классическую реализацию клиент-серверной утилиты с обработкой длинных путей, переводом сетевых заголовков и динамическим управлением памятью.

Фрагмент алгоритма приведен в приложении Е, листинг Е.1.

Собственная версия программы ls

Данная программа представляет собой упрощённый аналог команды ls с выводом расширенных метаданных файлов. Ниже приведено описание её реализации.

Структура программы:

Программа состоит из двух основных функций:

- print_file_info: Форматирует и выводит метаданные файла(см. приложение Ж, листинг Ж.1).
- list_directory: Обрабатывает содержимое директории.

Взаимодействие функций:

1. main принимает аргументы командной строки (имена директорий) или использует текущую директорию.
2. Для каждой директории вызывается list_directory, которая открывает её и передаёт управление print_file_info для каждого файла.
3. Функция print_file_info получает и обрабатывает всю информацию о файле и выводит ее в консоль в отформатированном виде.

Серверный логгер

В данной лабораторной работе реализована клиентская часть серверного логгера, который отправляет логи на удалённый сервер через HTTP-запросы. Логгер поддерживает гибкую настройку через паттерн "Строитель" (Builder) и интеграцию с JSON-конфигурацией.

1. Архитектура

- Класс server_logger_builder:
  Отвечает за конфигурацию логгера:
  - Добавление файловых и консольных потоков вывода для каждого уровня серьёзности (add_file_stream, add_console_stream).
  - Загрузка настроек из JSON-файла (transform_with_configuration).
  - Сброс конфигурации (clear).
- Класс server_logger:
  Реализует отправку логов на сервер:
  - Форматирование сообщений с временными метками и уровнем серьёзности.
  - Отправка HTTP-запросов на сервер через библиотеку httplib (см. приложение З, листинг З.1).

2. Основные функции

2.1. Настройка логгера

- Добавление потоков:
  Для каждого уровня серьёзности (trace, debug, info и т.д.) можно задать:
  - Путь к файлу для записи логов.
  - Флаг вывода в консоль.
- JSON-конфигурация: Метод transform_with_configuration парсит JSON и применяет настройки.

2.2. Отправка логов

- Форматирование сообщения:
  Сообщение включает временную метку, уровень серьёзности и текст:
- HTTP-запросы:
  Логгер отправляет GET-запросы на эндпоинты сервера:
  - /init — инициализация логгера с указанием настроек.
  - /log — отправка сообщения.

- /destroy — завершение работы логгера.

Данная реализация демонстрирует принципы работы с сетевыми запросами, гибкой настройкой компонентов и кроссплатформенным кодом.

Global Heap Allocator

Класс allocator_global_heap объявлен в заголовке как final и унаследован от четырёх базовых компонентов: smart_mem_resource задаёт интерфейс памяти, allocator_dbg_helper и logger_guardant добавляют вспомогательные макросы трассировки, typename_holder возвращает строковое имя типа. Внутри хранится только один указатель на logger; никаких других структур управления памятью нет, поэтому аллокатор служит тонкой обёрткой над глобальными операторами new / delete. В той же секции объявлены стандартные специальные функции: конструкторы копирования и перемещения, операторы присваивания и деструктор, а также приватный метод get_typename — всё это подтягивается через inline-определения и не требует дополнительного хранилища.

Функция do_allocate_sm получает размер, выводит диагностическое сообщение через debug_with_guard, затем вызывает оператор new из глобального пространства. Если оператор бросает std::bad_alloc, error_with_guard регистрирует ошибку и исключение передаётся вызывающему коду. Успешное выделение логируется с адресом и числом байт; после чего указатель выдаётся наружу. do_deallocate_sm проверяет аргумент на ненулевой адрес, пишет запись о высвобождении и вызывает оператор delete, не обрабатывая возможных исключений, что соответствует требованиям стандартного ресурса памяти. Ни выравнивание, ни отслеживание размера при удалении не реализованы, так как управление полностью передано механизмам оператора delete.

Деструктор тривиален, потому что объект не владеет никакими ресурсами кроме логгера, а специальная функция do_is_equal сравнивает this с адресом другого ресурса, объявляя два экземпляра эквивалентными только если они идентичны. Конструкторы и операторы с перемещением копируют или перемещают указатель логгера и обёрнуты trace_with_guard для детальной трассировки жизненного цикла. В совокупности код демонстрирует минималистичный адаптер поверх глобальной кучи с расширенным логированием, сохраняя совместимость с интерфейсом std::pmr::memory_resource и соблюдая правила C++ про noexcept в методах, где не происходит потенциально опасных операций.

Реализация предоставлена в приложении И, листинг И.1.

Sorted List Allocator

Основные характеристики

Реализация аллокатора включает в себя следующие моменты:

- Поддержка разных стратегий выделения (first-fit, best-fit, worst-fit)
- Потокобезопасность (используется мьютекс)
- Логирование операций
- Возможность работы с родительским аллокатором PMR
- Итераторы для обхода блоков

В метаданных аллокатора хранится:

- Указатель на логгер
- Указатель на родительский аллокатор PMR
- Режим выделения (fit_mode)
- Размер выделенного пространства
- Мьютекс для синхронизации
- Указатель на первый свободный блок

В метаданных блока памяти хранится:

- Каждый блок имеет метаданные:
- Размер блока
- Указатель на следующий свободный блок (если блок свободен), либо указатель на _trusted_memory, если блок занят.

Ключевые методы

Выделение памяти (do_allocate_sm)

1. Блокирует мьютекс
2. Ищет подходящий блок согласно выбранной стратегии
3. При необходимости разделяет блок на занятую часть и остаток
4. Обновляет список свободных блоков
5. Возвращает указатель на выделенную память

Освобождение памяти (do_deallocate_sm)

1. Блокирует мьютекс

2. Проверяет валидность освобождаемого блока
3. Добавляет блок в список свободных
4. Пытается объединить соседние свободные блоки

Фрагмент функции do_deallocate_sm представлен в приложении К, листинг К.1.

Для удобства работы было реализовано два вида итераторов: sorted_iterator - для обхода всех блоков и sorted_free_iterator - только для свободных блоков.

Boundary Tag Allocator

Класс allocator_boundary_tags объявлен как final и выводится из smart_mem_resource, allocator_test_utils и allocator_with_fit_mode, а вспомогательные механизмы трассировки и получения имени типа подключаются миксинами logger_guardant и typename_holder. Единственным полем-членом остаётся _trusted_memory, указывающий на выделенный в конструкторе буфер. Внутри доверенной области сначала размещается структура allocator_metadata с указателем на первый занятый блок, флагом режима поиска, логгером, мьютексом std::mutex и ссылкой на родительский ресурс; следом располагаются пользовательские блоки, каждый из которых описывается структурой block_metadata с полями размера, двойными ссылками, а также обратной ссылкой на trusted-область для проверки принадлежности при освобождении .

Функция do_allocate_sm использует std::lock_guard для защиты от гонок данных, выводит диагностику через macros debug_with_guard / information_with_guard, подбирает подходящий участок при помощи вспомогательных get_block_* функций, а затем, при необходимости, кор-размер корректирует warning-сообщением. Конструкция std::construct_at создаёт мьютекс внутри доверенной области памяти, std::format формирует строки лога, std::bad_alloc выбрасывается из-за нехватки памяти, а std::exchange и std::swap реализуют перемещающий конструктор и оператор. do_deallocate_sm зеркально проверяет tm_ptr_, удаляет блок из двусвязного списка, пишет дамп содержимого, фиксирует освобождение и снова логирует свободный объём; при ошибке принадлежности выбрасывается std::logic_error. Деструктор вручную вызывает деструктор мьютекса и возвращает всю область памяти через parent->deallocate, подчёркивая контроль над ресурсом на уровне самого аллокатора .

Дополнительный функционал включает двусторонний boundary_iterator с категорией bidirectional_iterator_tag; он ходит по занятым блокам и виртуальным «дыркам», вычисляя их размер без хранения заголовков свободных участков.

Операции begin и end создают прокси-итераторы поверх trusted-указателя. Метод set_fit_mode переводит enum в строку и атомарно обновляет поле fit_mode_, get_blocks_info обходит область памяти и собирает срез в std::vector; обе функции используют std::lock_guard для потокобезопасности. do_is_equal возвращает true только при сравнении с тем же объектом, а noexcept-методы get_typename и get_logger отдают строковый идентификатор и логгер без лишних вычислений.

Некоторые функции аллокатора предоставлены в листинге Л.1 приложения Л.

Buddies System Allocator

Класс allocator_buddies_system определён как final и собирается из пяти миксинов: smart_mem_resource обеспечивает интерфейс памяти, allocator_test_utils отдаёт сведения о блоках, allocator_with_fit_mode хранит выбранный алгоритм поиска, logger_guardant подключает журналирование, typename_holder возвращает строковое имя типа. Внутри оставлен единственный указатель _trusted_memory, который указывает на целиком выделенную арену. В самом начале арены располагается структура allocator_metadata с полями logger*, базовый std::pmr::memory_resource*, выбранный fit_mode, степень двойки size_k и std::mutex; сразу за ней лежат пользовательские блоки, каждый описывается битовым struct block_metadata, где один бит отвечает за занятость, а семь — за степень двойки самого блока. Статические константы occupied_block_metadata_size, free_block_metadata_size и min_k вычисляют минимальный ранг, а вложенный buddy_iterator реализует однопроходный forward-итератор по всем блокам, предоставляя size, occupied и адрес текущего блока.

Конструктор принимает требуемую степень двойки, опциональный родительский аллокатор, логгер и режим поиска. Он повышает запрошенное значение до ближайшей степени двойки через вспомогательный nearest_greater_k_of_2, затем запрашивает из parent_allocator единую область sizeof(allocator_metadata)+arena и инициализирует заголовок: заполняет ссылки, выставляет fit_mode, записывает size_k, конструирует mutex placement-new, а следом формирует первый свободный блок, задав ему occupied = false и size_k = k−min_k . Перемещающий конструктор и оператор обмениваются _trusted_memory с помощью std::exchange и std::swap, копирующие операции запрещены. Деструктор раскрывает mutex через destroy_at, а затем отдаёт всю память родителю, что подчёркивает отсутствие дополнительных ресурсов . Вспомогательные inline-методы get_logger, get_typename, begin, end и

get_blocks_info оборачивают доступ к метаданным под мьютексом и выводят удобные сервисные сведения.

do_allocate_sm блокирует мьютекс, прибавляет к пользовательскому размеру величину служебных данных, выбирает свободный блок через один из трёх get_block_* методов, а если память закончилась, формирует сообщение error_with_guard и бросает std::bad_alloc. Пока найденный блок способен делиться, его size_k уменьшается, для новой половинки создаётся twin-metadata, после чего возможное несоответствие округлённого размера фиксируется предупреждением. Заголовок блока помечается occupied, сразу за ним сохраняется контрольный указатель на trusted-область, и наружу возвращается адрес за пределами служебных байт; успешная операция журналируется с количеством свободных байт и дампом текущих блоков . do_deallocate_sm выполняет обратные действия: проверяет контрольный указатель, сбрасывает occupied, через get_buddy ищет соседа с тем же рангом и пока оба свободны объединяет их, увеличивая size_k; по завершении пишет сводку в лог . Все обращения к данным синхронизированы std::lock_guard, а вся навигация по арене строится из расчётов смещений и XOR-правила для buddies, что делает код независимым от выделения дополнительных вспомогательных структур.

Некоторые функции аллокатора предоставлены в листинге М.1 приложения М.

Red-Black Tree Allocator

Класс allocator_red_black_tree наследует интерфейс smart_mem_resource и несколько служебных миксинов, а единственным собственным полем держит указатель _trusted_memory на арену. В начале этой арены размещается структура allocator_metadata — в ней хранятся ссылки на логгер и родительский memory_resource, текущий fit_mode, размер арены, мьютекс std::mutex и корень дерева свободных блоков. Сами блоки описываются двумя уплотнёнными структурами: block_metadata с битовым полем «занят / цвет» плюс указатели back и forward_, и производным free_block_metadata с полями left_ и right_ для красно-чёрного дерева; если блок занят, поле parent_ ссылается на доверенную область, иначе — на родителя в дереве.

В конструкторе запрашивается непрерывный буфер sizeof(allocator_metadata)+size у переданного или глобального аллокатора, заполняется заголовок, конструируется мьютекс и в начало пользовательской памяти помещается первый свободный чёрный блок, ставший корнем дерева . do_allocate_sm берёт замок, выбирает свободный узел через одну из трёх

вспомогательных функций поиска, удаляет его из дерева, помечает занятым и, если позволяет размер, разрезает, создавая новый свободный узел и вставляя его обратно через rb_tree_insert; адрес полезной части возвращается вызывающему коду с подробным логом. do_deallocate_sm проверяет принадлежность блока, переводит его в свободное состояние, по двусвязной цепочке пытается слить с левым и правым соседями, предварительно удалив их из дерева, и затем фиксирует итоговый участок повторной вставкой.

Поддержку инвариантов дерева обеспечивают rb_tree_insert и rb_tree_remove — полноценные реализации красно-чёрного алгоритма с малыми поворотами rb_small_left_rotation и rb_small_right_rotation, написанными как inline-функции . Класс-итератор rb_iterator обходит всю память, двигаясь по полю forward_, и используется тестовой утилитой get_blocks_info. Перемещающие операции реализованы через std::exchange и std::swap, копирование запрещено, а деструктор снимает мьютекс и отдаёт арену родительскому ресурсу. В коде активно применяются возможности C++17: std::byte для арифметики адресов, std::construct_at, std::format, лямбда-выражения для внутренних хелперов, битовые поля и #pragma pack для экономии служебных байт, а вся публичная деятельность сопровождается макросами debug_with_guard / error_with_guard для детального журналирования.

Некоторые функции аллокатора предоставлены в листинге Н.1 приложения Н.

Реализация класса *B*-дерева

Узел B-дерева имеет следующую структуру:

- Вектор пар ключ-значение
- Вектор указателей на дочерние узлы

Используется static_vector из Boost для хранения элементов с фиксированным максимальным размером. Вектор _pointers всегда содержит на 1 элемент больше, чем _keys (для n ключей - n+1 указателей). В листовых узлах все указатели _pointers равны nullptr.

В классе помимо указателя на корень дерева также содержатся поля, отвечающие за логгер, аллокатор и количество элементов в дереве.

Операции с деревом

Во всех операциях для работы с B-деревом используется стек, который хранит путь до узла (в стеке хранятся родители и индексы указателей, по которым можно прийти в узел).

Поиск реализован в различных вариантах, соответствующих функциям из STL (find, contains, lower_bound, upper_bound). Алгоритм выполнения поиска описан в теоретической части.

При вставке при переполнении узла вызывается отдельная функция split. Операция вставки поддерживает move семантику, также возможна вставка через emplace.

При удалении используется (при необходимости) отдельная функция merge. Возможно удаление по итератору и по ключу. Исходный код функции merge представлен в приложении О, листинг О.1.

Реализованы 4 типа итераторов:

1. btree_iterator - прямой итератор

2. btree_const_iterator - константный прямой итератор

3. btree_reverse_iterator - обратный итератор

4. btree_const_reverse_iterator - константный обратный итератор

Итераторы двунаправленные (поддерживают ++ и --), они используют стек для хранения пути до текущего элемента.

Также они поддерживают методы для диагностики:

- depth() - глубина текущего узла
- current_node_keys_count() - количество ключей в текущем узле
- is_terminate_node() - проверка на листовой узел

Для аллокации используется специальный аллокатор pp_allocator. Все узлы создаются через _allocator.template new_object<btree_node>(), а удаление через _allocator.template delete_object<btree_node>().

Эта реализация предоставляет интерфейс, во многом аналогичный контейнерам STL, с поддержкой range-based for, initializer_list и других современных возможностей C++.

Реализация класса *B* +-дерева

Как уже было сказано в теоретической части, реализация B+-дерева во многом схожа с реализацией обычного B-дерева. Рассмотрим лишь основные отличия. Во-первых, были созданы два класса для узлов, для внутренних и листьев (bptree_node_term и bptree_node_middle), которые наследуются от класса bptree_node_base. В классе листа содержится вектор с ключами и данными, а в классе внутреннего узла вектор хранит только ключи. Исходный код этих классов представлен в приложении П, листинг П.1.

При выполнении операций следует поддерживать односвязный список, в который объединены узлы. Также следует учитывать тот факт, что данные хранятся только в листьях.

Реализация итераторов сильно упростилась благодаря тому, что все листья объединены в связный список. Было принято решение остановиться на реализации константной и не константной версиях итераторов, а также сделать их однонаправленными (только operator++).

Реализация класса *B*-дерева, сохраняющее данные в файловом хранилище

Реализация B-дерева на диске во многом схожа с реализацией обычного B-дерева, однако есть несколько отличий. Во-первых, вместо указателей используются поля size_t, которые указывают на позицию узла в файле с метаданными. Так как размер метаданных узла фиксирован, позицию нужного узла можно найти в файле по формуле <порядковый номер узла > * <размер метаданных одного узла>. С ключом и значениями все сложнее, так как их размер не фиксирован, поэтому для каждого ключа в узле хранится позиция, по которой этот ключ можно найти в соответствующем файле. Операции чтения и записи ключа должны быть реализованы в самом типе данных (за их проверку отвечает специальный concept). Полный исходный код структуры узла представлен в приложении Р, листинг Р.1.

Для работы с диском используются функции btree_disk_node disk_read(size_t node_position) и void disk_write(btree_disk_node& node). Функция чтения проверяет валидность позиции, перемещается к нужному месту в файле, загружает узел с указанной позиции в файле, десериализует данные в объект узла. Функция записи перемещается к позиции узла в файле, сериализует данные узла, обновляет позиции в файле ключей-значений.

Вывод

В процессе выполнения курсовой работы были успешно реализованы различные компоненты системного программирования, соответствующие стандартам C++20 и C99. Каждое задание требовало глубокого изучения теоретических основ, таких как принципы работы B-деревьев или алгоритмы синхронизации, а также их практической реализации — от проектирования классов с соблюдением принципов SOLID до отладки многопоточных приложений.

Проблемы, возникшие в ходе работы, например, обеспечение атомарности операций в аллокаторах или устранение ошибок в межпроцессном взаимодействии, были решены благодаря изучению документации, тщательному тестированию и применению инструментов профилирования.

Полученные результаты имеют значительную практическую ценность: приобретённые знания и навыки применимы в разработке операционных систем, драйверов, высоконагруженных серверов и других решений, ориентированных на системное программирование.

Список использованных источников

1. 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.

Приложения

Приложение А. Примитивная оболочка для командной строки

Листинг А.1. *Команда Sanctions.*

```
kErrors Sanctions(const char* user_string, const char* count, const vector* user_vector)
{
   if (strlen(count) > 8) {
      return INC_INPUT;
   }
   char* endptr;
   const ll num = strtol(count, &endptr, 10);
   // is there at least one digit in the number?
   if (endptr == count) {
      return INC_INPUT;
   }
   while (*endptr != '\0') {
      if (!isspace((unsigned char)*endptr)) {
         return INC_INPUT;
      }
      endptr++;
   }
   if (num < 0 || num > INT_MAX) {
      return INC_INPUT;
   }
   user* man = vector_search(user_vector, user_string);
   if (man == NULL) {
      return NO_SUCH_USER;
   }
   printf("Confirm\n");

   char check;
   char command[100] = "";

   scanf("%5s", command);
   scanf("%c", &check);
   if (check != '\n') {
      flush();
      return INC_INPUT;
   }
   if (strcmp(command, "12345") != 0) {
      printf("Confirm failed\n");
   } else {
      printf("Confirm succeed\n");
      man->sanctions = num;
      man->sanctions_current = num + 1;
   }
   return OK;
}
```

Приложение Б. Консольное приложение для обработки файлов

Листинг Б.1 — Парсинг флага из командной строки

```
char* flag = argv[argc - 1];
      if (strncmp(flag, "xor", 3) == 0) {
            …
      } else if (strncmp(flag, "mask", 4) == 0) {
            …
      } else if (strncmp(flag, "copy", 4) == 0) {
            …
      } else if (strncmp(flag, "find", 4) == 0) {
            …
      } else {
            fprintf(stderr, "Unexpected flag: %s\n", flag);
            close_files(descriptors, number_of_files);
            free(descriptors);
            return 1;}
```

Приложение В. Решение проблемы в задаче обедающих философов

Листинг В.1. *Фрагмент алгоритма*

```
if (id == 0) {
   philosopher p = {false, false, i};
   for (int l = 0; l < 3; l++) {
      status = semop(sem_id, &down0, 1);CHECK(status);
      status = take_fork(p.id - 1, p.id);CHECK(status);// sleep(1);
      status = take_fork(p.id + 1, p.id);
      CHECK(status);
      printf("Philosopher %d is eating\n", p.id);
      sleep(1);
      status = return_fork(p.id - 1);
      CHECK(status);
      status = return_fork(p.id + 1);
      CHECK(status);
      status = semop(sem_id, &up0, 1);
      CHECK(status);
   }
```

Приложение Г. Решение задачи про волка, козу и капусту с помощью клиент-серверного комплекса приложений

Листинг Г.1. Некоторые обработчики сообщений

```
static void handle_put(const message msg) {
      const int queue_id = msg.msg_text.queue_id;
      LOG(queue_id, "[>] request: put");
      session* session = get_session(queue_id);
      if (session == NULL) {
      send_error(queue_id, "Not registered");
      return;}
      if (session->obj_in_boat == OBJ_NONE) {
      send_error(queue_id, "No object in boat");
      return;}
      session->obj_in_boat = OBJ_NONE;
      // Check if all objects were moved to the right shore.
      bool all_objects_moved = session->wolf_on_right &&
session->goat_on_right && session->cabbage_on_right &&
session->boat_on_right;
      if (all_objects_moved) {
      send_all_moved(queue_id);
      return;}
      send_ok(queue_id);}
static void handle_move(const message msg) {
      const int queue_id = msg.msg_text.queue_id;
      LOG(queue_id, "[>] request: move");
      session* session = get_session(queue_id);
      if (session == NULL) {
      send_error(queue_id, "Not registered");
      return;}
```

```
session->boat_on_right = !session->boat_on_right;
// Update object's position on shore.
switch (session->obj_in_boat) {
case OBJ_WOLF:
      session->wolf_on_right = session->boat_on_right;
      break;
case OBJ_GOAT:
      session->goat_on_right = session->boat_on_right;
      break;
case OBJ_CABBAGE:
      session->cabbage_on_right = session->boat_on_right;
      break;
case OBJ_NONE:
      break;}
// Check if the wolf and goat are left unsupervised.
if (session->wolf_on_right == session->goat_on_right &&
session->wolf_on_right != session->boat_on_right) {
send_error(queue_id, "Goat eaten");
return;
}
// Check if the goat and cabbage are left unsupervised.
if (session->goat_on_right == session->cabbage_on_right &&
session->goat_on_right != session->boat_on_right) {
send_error(queue_id, "Cabbage eaten");
return;}
send_ok(queue_id);}
```

Приложение Д. Решение задачи про университетскую ванную комнату

Листинг Д.1. *Алгоритм синхронизации для мужчин.*

```
int man_wants_to_enter(_Atomic(int) *cur_state) {
   while (true) {
      pthread_mutex_lock(&mutex);
      if (*cur_state == EMPTY || *cur_state == MAN_ONLY) {
         *cur_state = MAN_ONLY;
         if (current_number < max_in_room) {
            current_number++;
            printf("Man entered\n");
            pthread_mutex_unlock(&mutex);
            break;
         }}
      pthread_mutex_unlock(&mutex);
   }}
```

Приложение Е. Клиент-серверная система анализа и группировки файловых путей через сокеты

Листинг Е.1. Функция обработки клиента

```
static void process_client(const int client_fd) {
	char* msg_buf = NULL;
	size_t msg_buf_size = 0;
	uint64_t msg_len;
	while (recv(client_fd, &msg_len, sizeof(msg_len), MSG_WAITALL) == sizeof(msg_len)) {
	if (msg_len == 0) {
		continue;}
	msg_len = be64toh(msg_len);
	LOG(client_fd, "read message of %zu bytes", msg_len);
	// Resize the buffer to fit the new message.
	if (msg_len > msg_buf_size) {
		char* new_msg_buf = (char*)realloc(msg_buf, msg_len);
		if (new_msg_buf == NULL) {
			fprintf(stderr, "failed to resize msg buffer\n");
			break;}
		msg_buf = new_msg_buf;
		msg_buf_size = msg_len;
	}
	// Read the full message.
	if (recv(client_fd, msg_buf, msg_len, MSG_WAITALL) != msg_len) {
		LOG(client_fd, "failed to read message");
		break;
	}
	LOG(client_fd, "read message: %s", msg_buf);
	// Retrieve absolute path.
	char resolved_path[MAX_PATH];
```

```
if (realpath(msg_buf, resolved_path) == NULL) {
      send_string(client_fd, "no such path exists");
      continue;}
// Validate the path.
if (strcmp(msg_buf, resolved_path) != 0) {
      send_string(client_fd, "not an absolute path");
      continue;}{
      struct stat path_stat;
      stat(resolved_path, &path_stat);
      if (!S_ISDIR(path_stat.st_mode)) {
            send_string(client_fd, "not a directory");
            continue;}}
char* listing = list_files(resolved_path);
if (listing == NULL) {
      send_string(client_fd, "failed to list files");
} else {
      send_string(client_fd, listing);
      free(listing);
}}
free(msg_buf);}
```

Приложение Ж. Собственная версия программы ls

Листинг Ж.1 — Функция file_stat_info

```
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);}
```

Приложение 3. Серверный логгер

Листинг 3.1 — Отправка логов на сервер

```
logger& server_logger::log(const std::string& message,
                           logger::severity severity) & {
    std::stringstream stringstream;
        stringstream << "[" << current_date_to_string() << " " << 
current_time_to_string() <<
        "][" << severity_to_string(severity) << "] " << message;
    httplib::Params par;
    par.emplace("pid", std::to_string(server_logger::inner_getpid()));
    par.emplace("sev", severity_to_string(severity));
    par.emplace("message", stringstream.str());
    check(_client.Get("/log", par, httplib::Headers()));
    return *this;}
```

Приложение И. Global Heap Allocator

Листинг 3.1. Реализация аллокатора

```
[[nodiscard]] void *allocator_global_heap::do_allocate_sm(
      size_t size){
      debug_with_guard(std::format("[*] do_allocate_sm({})", size));
      void* mem;
      try{
      mem = ::operator new(size);
} catch (const std::bad_alloc &e){
      error_with_guard(std::format("[!] allocation failed: {}", e.what()));
      throw;}
      debug_with_guard(std::format("[+] allocated {} bytes at {:p}", size,
mem));
      return mem;}
void allocator_global_heap::do_deallocate_sm(void *at){
      if (at){
      debug_with_guard(std::format("[*] freeing at {:p}", at));
      ::operator delete(at);}}
```

Приложение К. Sorted List Allocator

Листинг К.1. Фрагмент реализации функции do_deallocate_sm

```
if (prev_free_block != nullptr) {
   set_next_ptr(prev_free_block, block_header);}
set_next_ptr(block_header, next_free_block);
if (try_merge(prev_free_block, static_cast<uint8_t *>(block_header))) {
   try_merge(prev_free_block, next_free_block);} else {
   try_merge(block_header, next_free_block);}
void *first_block_ptr = get_first_free_block_pointer();
if (prev_free_block == nullptr && static_cast<uint8_t *>(first_block_ptr) >
block_header ||
   first_block_ptr == nullptr) {
   set_first_free_block_pointer(block_header);}
if (l != nullptr){
   l->information("Memory left: ");
   size_t count = get_free_memory_count();
   l->information(std::to_string(count));
   auto blocks = get_blocks_info();
   for(auto item: blocks){
                    l->debug(std::to_string(item.block_size)   +   "   "   +
std::to_string(item.is_block_occupied) + "\n");}
l->debug("do_deallocate_sm started finished\n");}
```

Листинг К.1. Функции выделения и освобождения

```
[[nodiscard]] void *allocator_boundary_tags::do_allocate_sm(size_t size){
      size_t total_size = size + sizeof(block_metadata);
      debug_with_guard(std::format("[*] allocating {} bytes", total_size));
      auto& metadata = get_allocator_metadata();
      std::lock_guard lock(metadata.mutex_);
      block_metadata* block = nullptr;
      switch (metadata.fit_mode_){
      case fit_mode::first_fit:
      block = get_block_first_fit(total_size);break;
      case fit_mode::the_best_fit:
      block = get_block_best_fit(total_size);break;
      case fit_mode::the_worst_fit:
      block = get_block_worst_fit(total_size);break;}
      if (block == nullptr){
      error_with_guard(std::format(
            "[!] out of memory: requested {} bytes", total_size));
      throw std::bad_alloc();}
      // XXX: 'block' указывает на блок перед свободным блоком.
      const size_t free_block_size = get_next_free_block_size(block);
      if (free_block_size < total_size + sizeof(block_metadata)){
      // Если не получается выделить ещё один блок после этой аллокации,
      // отдаём блоку всю оставшуюся память.
      warning_with_guard(std::format(
            "[*] changing block size to {} bytes", free_block_size));
      total_size = free_block_size;}
      block_metadata* free_block;
      bool iter_begin = block == _trusted_memory;
      if (iter_begin){
```

```
      // Мы в начале итератора, где _occupied_ptr указывает на начало
области памяти.

      // Это и будет свободным блоком.

      free_block = reinterpret_cast<block_metadata*>(

           static_cast<std::byte*>(_trusted_memory) +
sizeof(allocator_metadata));}else{

      free_block = reinterpret_cast<block_metadata*>(block->block_end());}

      free_block->block_size_ = total_size - sizeof(block_metadata);

      free_block->prev_ = block;

      free_block->next_ = iter_begin ? metadata.first_block_ :
block->next_;

      free_block->tm_ptr_ = _trusted_memory;

      // Подвязываем блоки

      if (free_block->next_){

      free_block->next_->prev_ = free_block;}

      if (iter_begin){

      metadata.first_block_ = free_block;}else{

      free_block->prev_->next_ = free_block;}

      debug_with_guard(std::format(

      "[+] allocated {} bytes at {:p}",

      total_size, static_cast<void*>(free_block + 1)));

      information_with_guard(std::format(

      "[*] available memory: {}", get_available_memory()));

      debug_with_guard(print_blocks());

      return free_block + 1;}

void allocator_boundary_tags::do_deallocate_sm(void *at){

      debug_with_guard(std::format("[*] deallocating block {:p}", at));

      auto& metadata = get_allocator_metadata();

      std::lock_guard lock(metadata.mutex_);

      auto block = reinterpret_cast<block_metadata*>(

      static_cast<std::byte*>(at) - sizeof(block_metadata));
```

```
    if (block->tm_ptr_ != _trusted_memory){
    error_with_guard(std::format(
         "[!] block doesn't belong to this allocator: {:p}", at));
    throw std::logic_error("unknown block");}
    debug_with_guard(get_dump(static_cast<char*>(at),
block->block_size_));
    if (block->prev_ == _trusted_memory){
    metadata.first_block_ = block->next_;}else{
    block->prev_->next_ = block->next_;}
    if (block->next_){
    block->next_->prev_ = block->prev_;}
    debug_with_guard("[+] block deallocated successfully");
    information_with_guard(std::format(
    "[*] available memory: {}", get_available_memory()));
    debug_with_guard(print_blocks());}
```

Листинг Л.1. Функции выделения и освобождения

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

debug_with_guard("[>] entering allocator_buddies_system::do_allocate_sm");

auto metadata = reinterpret_cast<allocator_metadata *>(_trusted_memory);

     std::lock_guard<std::mutex> lock(metadata->mutex);     size_t
size_with_metadata = size + sizeof(occupied_block_metadata_size);

     block_metadata *block = nullptr;

     debug_with_guard(std::format("[*] allocating {} bytes",
size_with_metadata));

     switch (metadata->fit_mode){

     case allocator_with_fit_mode::fit_mode::first_fit:

     block = get_block_first_fit(size_with_metadata);break;

     case allocator_with_fit_mode::fit_mode::the_best_fit:

     block = get_block_best_fit(size_with_metadata);break;

     case allocator_with_fit_mode::fit_mode::the_worst_fit:

     block = get_block_worst_fit(size_with_metadata);break;}

     if (block == nullptr){

     error_with_guard(std::format("[!] out of memory: requested {} bytes",
size));throw std::bad_alloc();}

     while (block->size_k > 0 && block->block_size() >= size_with_metadata
* 2){--block->size_k;auto buddy = get_buddy(block);buddy->occupied = false;

     buddy->size_k = block->size_k;}

if (block->block_size() != size_with_metadata){

     warning_with_guard(std::format("[!] changed allocation size to {}
bytes", size_with_metadata));}

     block->occupied = true;

     auto tm_ptr = reinterpret_cast<void **>(block + 1);

     *tm_ptr = _trusted_memory;

     auto allocated_block = static_cast<void *>(

     reinterpret_cast<std::byte *>(block) + occupied_block_metadata_size);
```

```
     information_with_guard(std::format("[+] allocated {} bytes at {},
available memory: {} bytes",size_with_metadata, allocated_block,
available_memory()));

     debug_with_guard(std::format("[*] current blocks: \n{}",
print_blocks()));

     debug_with_guard("[<] leaving
allocator_buddies_system::do_allocate_sm");

     return allocated_block;}

void allocator_buddies_system::do_deallocate_sm(void *at){

     debug_with_guard("[>] entering
allocator_buddies_system::do_deallocate_sm");

     auto metadata = reinterpret_cast<allocator_metadata
*>(_trusted_memory);

     std::lock_guard<std::mutex> lock(metadata->mutex);

     debug_with_guard(std::format("[*] deallocating block at {}", at));

     auto block = reinterpret_cast<block_metadata *>(static_cast<std::byte
*>(at) - occupied_block_metadata_size);{

     auto tm_ptr = reinterpret_cast<void **>(block + 1);

     if (*tm_ptr != _trusted_memory){

          error_with_guard(std::format("[!] block is not allocated by this
allocator"));throw std::logic_error("foreign block");}}

     block->occupied = false;

     auto buddy = get_buddy(block);

     while (block->block_size() < metadata->size() && block->block_size()
== buddy->block_size() && !buddy->occupied){

     // Берём блок, который "выше" в памяти

     if (buddy < block){std::swap(block, buddy);}

     ++block->size_k;

     buddy = get_buddy(block);}

     information_with_guard(std::format("[+] deallocated block at {},
available memory: {} bytes",at, available_memory()));

debug_with_guard(std::format("[*] current blocks: \n{}", print_blocks()));

debug_with_guard("[<] leaving
allocator_buddies_system::do_deallocate_sm");}
```

Приложение H. Red-Black Tree Allocator

Листинг M.1. Функции выделения и освобождения

```
[[nodiscard]] void *allocator_red_black_tree::do_allocate_sm(
      size_t size){
      debug_with_guard("[>] entering
allocator_red_black_tree::do_allocate_sm");
      allocator_metadata* alloc = get_metadata();
      std::lock_guard guard(alloc->mutex_);
      debug_with_guard(std::format("[*] allocating {} bytes", size));
      free_block_metadata* taken_block = nullptr;
      switch (alloc->fit_mode_){
      case fit_mode::first_fit:
      taken_block = get_first_free_block(size); break;
      case fit_mode::the_best_fit:
      taken_block = get_best_free_block(size);break;
      case fit_mode::the_worst_fit:
      taken_block = get_worst_free_block(size);break;}
      if (taken_block == nullptr){
      error_with_guard(std::format("[!] out of memory: requested {} bytes",
size));
      throw std::bad_alloc();}
      rb_tree_remove(taken_block);
      taken_block->occupied = true;
      taken_block->parent_ = _trusted_memory;
      size_t required_size = size + sizeof(block_metadata);
      const bool can_split = taken_block->get_size(_trusted_memory)
      >= required_size + sizeof(free_block_metadata);
      if (can_split){
      auto* new_block = reinterpret_cast<free_block_metadata*>(
            reinterpret_cast<std::byte*>(taken_block) + required_size);
```

```
	// Подвязываем блоки в двусвязном списке
	new_block->forward_ = taken_block->forward_;
	new_block->back_ = taken_block;
	taken_block->forward_ = new_block;
	if (new_block->forward_){new_block->forward_->back_ = new_block;}
	// Вставляем свободный блок в КЧД
	new_block->occupied = false;
	new_block->parent_ = nullptr;
	rb_tree_insert(new_block);}
	auto allocated_block = static_cast<void*>(
		reinterpret_cast<std::byte*>(taken_block) +
sizeof(block_metadata));
	information_with_guard(std::format("[+] allocated {} bytes at {},
available memory: {} bytes",size, allocated_block, available_memory()));
	debug_with_guard(std::format("[*] current blocks: \n{}",
print_blocks()));
	debug_with_guard("[<] leaving
allocator_red_black_tree::do_allocate_sm");return allocated_block;}
void allocator_red_black_tree::do_deallocate_sm(void *at){
	allocator_metadata* alloc = get_metadata();
	std::lock_guard guard(alloc->mutex_);
	debug_with_guard(std::format("[*] deallocating block at {}", at));
	auto* block = reinterpret_cast<block_metadata*>(
	static_cast<std::byte*>(at) - sizeof(block_metadata));
	if (block->parent_ != _trusted_memory){
	error_with_guard(std::format("[!] block is not owned by this
allocator"));
	throw std::logic_error("foreign block");}
	block->occupied = false;
	if (block->back_ && !block->back_->occupied){
	// Сливаем предыдущий блок с текущим: удаляем из дерева свободных
блоков,
```

```
    // подвязывем со следующим текущего, ставим на место текущего.
    auto* back = static_cast<free_block_metadata*>(block->back_);
    rb_tree_remove(back);
    back->forward_ = block->forward_;
    if (back->forward_) back->forward_->back_ = back;
    block = back;}
    if (block->forward_ && !block->forward_->occupied){
    // Аналогично сливаем со следующим.
    auto* fwd = static_cast<free_block_metadata*>(block->forward_);
    rb_tree_remove(fwd);
    block->forward_ = fwd->forward_;
    if (block->forward_) block->forward_->back_ = block;}
    rb_tree_insert(static_cast<free_block_metadata*>(block));
    information_with_guard(std::format("[+] deallocated block at {},
available memory: {} bytes",at, available_memory()));
    debug_with_guard(std::format("[*] current blocks: \n{}",
print_blocks()));
    debug_with_guard("[<] leaving
allocator_red_black_tree::do_deallocate_sm");}
```

Приложение О. Реализация класса *B*-дерева

Листинг Н.1. *Функция merge*

```
template<typename tkey, typename tvalue, compator<tkey> compare, std::size_t t>

void B_tree<tkey, tvalue, compare, t>::merge(B_tree::btree_node * left, B_tree::btree_node * right, B_tree::btree_node * parent, size_t split_key_index) {

          B_tree::btree_node * new_node = _allocator.template new_object<B_tree::btree_node>();

   new_node->_keys = left->_keys;

   new_node->_pointers = left->_pointers;

   new_node->_keys.push_back(parent->_keys[split_key_index]);

   new_node->_pointers.push_back(nullptr);

       new_node->_keys.insert(new_node->_keys.end(), right->_keys.begin(), right->_keys.end());

                      new_node->_pointers.insert(new_node->_pointers.end(), right->_pointers.begin(), right->_pointers.end());

   parent->_keys.erase(parent->_keys.begin() + split_key_index);

   parent->_pointers.erase(parent->_pointers.begin() + split_key_index);

   if (parent->_pointers.size() > split_key_index) {

       parent->_pointers[split_key_index] = new_node;} else {

       parent->_pointers.push_back(new_node);}

   _allocator.delete_object(left);

   _allocator.delete_object(right);}
```

Приложение П. Реализация класса *B* +-дерева

Листинг О.1. *Структуры для узлов.*

```
struct bptree_node_base{

   bool _is_terminate;

   bptree_node_base() noexcept;

   virtual ~bptree_node_base() =default;

   size_t size();};

struct bptree_node_term : public bptree_node_base{

   bptree_node_term* _next;

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

   bptree_node_term() noexcept;};

struct bptree_node_middle : public bptree_node_base{

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

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

   bptree_node_middle() noexcept;};
```

Приложение Р. Реализация класса *B*-дерева, сохраняющее данные в файловом хранилище

Листинг Н.1. *Структура узла*

```
struct btree_disk_node{

   static size_t _last_position_in_key_value_file;

   size_t size; // кол-во заполненных ячеек

   bool _is_leaf;

   size_t position_in_disk;

   std::vector<tree_data_type> keys;

   std::vector<size_t> pointers;

   mutable std::vector<size_t> positions_in_key_value_file;

     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();

   btree_disk_node(size_t);

   static const size_t node_metadata_size = 2 * sizeof(size_t)

                                                    + sizeof(bool)

                                                          + sizeof(size_t) *
maximum_pointers_in_node

                                                          + sizeof(size_t) *
maximum_keys_in_node; //for positions_in_key_value_file};
```

Приложение С Репозиторий с исходным кодом

mai-fa-sp-4-sem [Электронный ресурс]. // github — URL: https://github.com/oryce/mai-fa-sp-4-sem (дата обращения: 20.05.2025).