ПРАВИТЕЛЬСТВО РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет компьютерных наук
Образовательная программа «Прикладная математика и информатика»

Отчет об программном проекте на тему:

Сборщик мусора с нуля

Выполнил студент:

группы #БПМИ233, 2 курса Битюков Павел Антонович

Принял руководитель проекта:

Соколов Павел Павлович
Преподаватель
Факультет компьютерных наук НИУ ВШЭ

Москва 2024

Содержание

Аннотация

В данной работе рассматривается разработка собственного сборщика мусора (garbage collector) с нуля. Основное внимание уделяется принципам автоматического управления памятью, анализу существующих алгоритмов сборки мусора и их сравнительному исследованию. В рамках проекта реализуется кастомный сборщик мусора для рантайма языка C.

Проводится анализ эффективности предложенного решения, включая замеры производительности и сравнение с классическими алгоритмами, такими как Mark-Sweep, Mark-Compact и Copying GC. Также рассматриваются вопросы интеграции разработанного сборщика в программные системы и возможные направления его оптимизации.

Ключевые слова

сборщик мусора, управление памятью, автоматическое управление памятью, алгоритмы GC, Mark-Sweep, подсчет ссылок.

1 Введение

Автоматическое управление памятью играет ключевую роль в современных языках программирования, позволяя разработчикам создавать сложные программные системы без необходимости вручную отслеживать выделение и освобождение памяти. Один из наиболее важных компонентов этого процесса — **сборщик мусора** (garbage collector, GC), который автоматически удаляет неиспользуемые объекты, предотвращая утечки памяти и обеспечивая эффективное использование ресурсов.

Сборка мусора особенно актуальна в высокоуровневых языках программирования, таких как Python, Java, C#, Go, где управление памятью скрыто от разработчика. Однако низкоуровневые языки, такие как C и C++, традиционно требуют ручного управления памятью, что увеличивает вероятность ошибок, таких как утечки памяти и использование уже освобождённых объектов.

1.1 Постановка задачи

Цель данного проекта — **изучение актуальных алгоритмов сборки мусора** и **разработка собственного уборщика мусора для рантайма на C**. Для этого необходимо:

1. Разобраться с концепцией автоматического управления памятью, принципами работы сборщиков мусора и особенностями рантаймов языков программирования.

2. Изучить современные алгоритмы сборки мусора, такие как *Reference Counting*, *Mark-Sweep*, *Mark-Compact*, *Copying GC* и *Generational GC*.

3. Реализовать собственный сборщик мусора для рантайма на языке C, выбрав оптимальный алгоритм и протестировав его эффективность.

1.2 Актуальность и значимость

Несмотря на существование развитых систем управления памятью, понимание принципов работы сборщиков мусора имеет важное значение для оптимизации программного кода, повышения производительности и написания безопасных программ. Разработка собственного GC позволяет глубже изучить принципы работы автоматического управления памятью и исследовать возможные улучшения существующих подходов.

В частности, данный проект будет полезен для:

- **Изучения механизмов управления памятью** в современных языках программирования.
- **Проектирования эффективных алгоритмов очистки памяти** с учетом производительности и оптимизации.
- **Разработки инструментов для языков с автоматическим управлением памятью**, что может быть полезно при создании новых языков программирования или расширении существующих.

1.3 Основные результаты работы

В ходе проекта будут изучены теоретические основы автоматического управления памятью и реализован собственный сборщик мусора для рантайма на C. Разработанный GC будет протестирован, а его эффективность сравнена с классическими алгоритмами.

1.4 Структура работы

В первой части работы будет проведён обзор существующих алгоритмов сборки мусора, рассмотрены их преимущества и недостатки. Во второй части будет описан процесс проектирования и реализации собственного GC, включая архитектурные решения и особенности работы с памятью на C. В третьей части будет проведено тестирование и анализ эффективности реализованного решения, а также сравнение с традиционными подходами. В заключении подводятся итоги работы и предлагаются возможные направления дальнейшего развития проекта.

2 Обзор литературы

2.1 Основные идеи сборки мусора

Существуют несколько фундаментальных идей, лежащих в основе современных сборщиков мусора. Эти идеи определяют, **как именно** GC определяет «ненужные» объекты и освобождает память. Идеи ниже были подробнее описаны в книге [6]

2.1.1 Подсчет ссылок (Reference Counting, RC)

Одна из простейших идей управления памятью — **подсчет ссылок (Reference Counting, RC)**. У каждого объекта хранится счётчик активных ссылок. Когда создается новая ссылка, счетчик увеличивается, а при удалении ссылки — уменьшается. Если счетчик достигает нуля, объект удаляется, ресурсы освобождаются

Пример использования Reference Counting можно найти в **C++**, где умные указатели `shared_ptr` реализуют этот подход на уровне стандартной библиотеки, а также в **Rust**, где есть `Rc`, и потокобезопасный `Arc`.

Преимущества:

1. Простая реализация
2. Освобождение памяти происходит **немедленно**, как только объект становится ненужным.
3. Подходит для сценариев с предсказуемым управлением памятью.
4. Может быть использован в средах с ограниченными ресурсами, так как не требует сложных глобальных обходов памяти.

Недостатки:

1. Управление счетчиком требует **дополнительных затрат CPU**, которые становятся значительными, если в программе много объектов, происходит много операций присваивания переменных, а также большой оверхед на короткоживущих объектах. Любые Read, Write операции с объектами требуют модификаций Rc. Кроме того, происходит косвенное замедление других операций (так как например нужные значения уходят из кэшей процессора, меняясь на адреса в памяти Rc).
2. Не может автоматически очистить **циклические зависимости**, когда два объекта ссылаются друг на друга.

3. Для эффективной работы, а также, чтобы сохранять исходную структуру пользовательских типов данных требует **глубокой интеграции с компилятором/интерпретатором**

4. В многопоточной среде требуется синхронизация доступа к счетчику, что может приводить к задержкам (например, **использование атомарных операций или мьютексов**).

5. **Не освобождает недостижимые, но еще используемые объекты**: в RC объект освобождается только когда его счетчик ссылок становится равным нулю. Однако объект может оставаться доступным (например, в коллекциях или кэше), даже если он больше не нужен.

6. Размер каждого объекта должен быть увеличен на **8 байтов** для хранения счётчика

2.1.2 Почему Reference Counting не заменяет Tracing GC?

Хотя в некоторых языках (например, C++) можно использовать слабые ссылки (*weak references*) для разрыва циклических зависимостей, **это не делает Reference Counting полноценной заменой Tracing GC**. Кроме недостатков, описанных выше: **Не освобождает сложные структуры данных автоматически**: в RC программист должен сам следить за корректным управлением памятью. Tracing GC автоматически очищает даже сложные графы объектов. То есть управление памяти перестаёт быть полностью автоматическим. Таким образом, **Reference Counting** может быть лишь частью GC.

2.1.3 Отслеживание достижимости (Tracing Garbage Collection) и Mark-Sweep

Вместо явного подсчета ссылок можно использовать другую идею: **не важно, сколько у объекта ссылок, важно — достижим ли он от корневых объектов (GC roots)**. Эта концепция лежит в основе **Tracing GC**.

Основной механизм, использующий эту идею, — это **Mark-Sweep**:

1. На первом этапе GC проходит по всем объектам, начиная от «корневых» (глобальные переменные, стек вызовов, регистры CPU) и, проходя рекурсивно по графу связей объектов, **помечает** все достижимые объекты (**стадия Mark**). Проход по графу объекту - называется Tracing, отсюда и Tracing GC.

2. Затем все непомеченные объекты удаляются (**стадия Sweep**).

Преимущества:

1. Автоматически очищает **циклы** (в отличие от Reference Counting).
2. Не перемещает объекты, **указатели не инвалидируются**
3. Нет оверхеда на операции Read, Write
4. Маленький оверхед по памяти по сравнению с RC. Если вспомнить, что в современных системах используются только первые 48 бит из 64 в указателе, то mark bit можно зашивать в оставшиеся 16. Тогда вообще нет оверхеда.
5. Не требует никакой информации о типах и структуре объектов, то есть не должен работать вместе с компилятором/интерпретатором, может быть отдельным модулем.

Недостатки:

1. Остановка программы на время очистки (*Stop-the-World*).
2. В многопоточном случае проблема Stop-the-World усугубляется.
3. Надо буквально парсить Heap, чтобы находить живущие объекты. А значит время работы пропорционально размеру Heap.
4. Из-за того что не перемещает объекты, то вероятность фрагментации памяти повышается, поэтому вместе с таким GC должен работать продвинутый аллокатор.

2.1.4 Mark-Compact

Чтобы решить проблему фрагментации памяти после Mark-Sweep, используется идея **Mark-Compact**. Вместо простого удаления объектов GC **перемещает** оставшиеся объекты, чтобы устранить пробелы в памяти.

Преимущества:

1. Исключает фрагментацию памяти. Сильно ускоряется аллокации.
2. Обеспечивает лучшее кеширование данных CPU, так как данные расположены рядом

Недостатки:

1. Дополнительные затраты на перемещение объектов.
2. Хуже throughput, чем у mark-sweep, так как требует несколько проходов по heap

3. Требуется интеграция с компилятором/интерпретатором/аллокатором, так как надо знать много доп. информации про перемещаемые участки памяти, надо обновлять указатели, которые перестали быть действительными, надо знать когда выгодно запускать.

4. Долгоживущие объекты являются bottleneck этого алгоритма, так как тратятся ресурсы на постоянное копирование таких объектов

5. Чтобы сделать время работы приемлемым, то может не выполняться стадия compact для мелких частей, тем самым фрагментацию нельзя избежать полностью

6. Сохраняются недостатки 1-3 mark-sweep

Хорошим вариантом комбинации двух идей, описанных выше, является запускать почти всегда mark-sweep, а когда аллокатор начинает тормозить, то есть фрагментация памяти становится ощутимой, запускать mark-compact. Ключевая идея здесь в том, что mark-compact не имеет смысла запускать через небольшие промежутки времени, так как память не успевает фрагментироваться.

2.1.5 Копирующий сборщик мусора (Copying GC)

Этот сборщик мусора также спроектирован для решения проблемы фрагментации памяти. В начале вся Heap делится на две равные части:

1. From-space. Область памяти в которой находятся все объекты.

2. To-space. Пустая область, в которую будут копироваться объекты.

Аллокации происходят очень быстро: в to-space на последнем байте установлен free pointer, с помощью сдвига free pointer на нужное кол-во байт происходит аллокация, если места не хватает, то части меняются местами и происходит копирование живых объектов из from-space в to-space. Притом после такого копирования решается проблема фрагментации.

Преимущества

1. Сохраняются преимущества mark-compact, но аллокация быстрее.

2. Collect быстрее чем у mark-compact, так как требует 1 проход по Heap.

3. этот GC удобнее применять в связке с Generational GC, Parallel GC, чем mark-compact

4. Фрагментации получается избегать полностью.

Недостатки:

1. Главный недостаток – уменьшение размера Heap в два раза.
2. Сохраняются недостатки 1, 3, 4, 6 mark-compact

2.1.6 Поколенческий сборщик мусора (Generational GC)

Было замечено, что **большинство объектов в программах «живут» недолго**. На основе этого наблюдения появилась идея **поколенческого GC**.

Этот GC делит память на **несколько зон (поколений)**:

1. **Молодое поколение (Young Generation)** — недавно созданные объекты. Они очищаются чаще всего.
2. **Среднее поколение (Survivor Generation)** — объекты, пережившие одну или несколько сборок.
3. **Старое поколение (Old Generation)** — долгоживущие объекты, которые очищаются реже.

Преимущества:

1. Более эффективная очистка памяти: короткоживущие объекты удаляются быстро.
2. Уменьшается время пауз GC.

Недостатки:

1. Надо поддерживать время жизни объектов, а также выполнять их перераспределение между поколениями

2.1.7 Конкурентный сборщик мусора (Concurrent GC)

Традиционные GC требуют остановки программы на время работы, но в многопоточных приложениях такие паузы недопустимы. Решением стала идея **конкурентного GC (Concurrent GC)**, который выполняет сборку мусора **параллельно с выполнением кода**.

Преимущества:

1. Минимальные задержки (low latency).

2. Подходит для высоконагруженных систем, абсолютно необходимое решение для real-time систем.

Недостатки:

1. Тяжелее реализация и поддержка.
2. Больше расходуется CPU time.

2.2 Сборка мусора в популярных языках программирования

2.2.1 CPython (Python)

- Использует **Reference Counting** как основной механизм.
- Для циклических ссылок применяется **Generational GC**, разделяющий объекты на три поколения. [1]

2.2.2 C# (.NET)

- Основной механизм — **Generational GC** с Mark-Compact.
- Поддерживает **конкурентные режимы** GC для серверных приложений. [4]

2.2.3 Java (JVM)

- Использует комбинацию **Generational GC** и разных алгоритмов (Serial, Parallel, G1 GC, ZGC [5]).
- Поддерживает low latency GC (Shenandoah [3], ZGC) для минимизации пауз.

2.2.4 Go

- Concurrent Mark-Sweep [2]

3 Архитектура и реализация сборщика мусора

3.1 Общая архитектура

Разрабатываемый сборщик мусора представляет собой систему автоматического управления памятью, реализованную как самостоятельная библиотека на языке C++ с внешним C-интерфейсом. Система предназначена для интеграции в рантайм C-программ, где традиционно отсутствует автоматическое управление памятью.

Архитектура библиотеки построена с разделением на три независимых компонента:

- **GCImpl** — отвечает за внутреннюю реализацию сборки мусора, включая аллокацию, управление корнями, фазу маркировки и удаления.
- **GCScheduler** — реализует автоматический запуск сборки на основе внешних параметров.
- **GCPacer** — модуль анализа интенсивности выделения памяти, принимает решение о необходимости сборки.

Такое разбиение реализовано с целью масштабируемости и дальнейшей расширяемости: каждая из частей может быть модифицирована или заменена независимо. Например, можно легко внедрить другой алгоритм (например, Generational GC) без необходимости переписывать логику планирования или сбора статистики. Благодаря модульной архитектуре возможно проводить точечные оптимизации, не нарушая остальную логику.

3.2 Выбор стратегии сборки: Mark-Sweep

В качестве основного алгоритма сборки выбран классический **Mark-Sweep**. Основные причины этого выбора:

- **Простота реализации.** Алгоритм не требует перемещения объектов, не нуждается в обновлении указателей, не зависит от структуры типов и может быть реализован полностью на уровне пользовательского кода без поддержки со стороны компилятора.
- **Эффективность в реальных сценариях.** Для большинства программ, особенно в ручных системах управления памятью, Mark-Sweep демонстрирует предсказуемое поведение и малый оверхед в паузах.

- **Поддержка модификаций.** На основе Mark-Sweep легко реализовать улучшенные версии, включая Mark-Compact, Incremental GC, Generational GC и др. Выбор данного алгоритма позволяет сохранить гибкость дальнейшего развития проекта.
- **Совместимость с языком C++.** Использование C++ позволило реализовать безопасную и быструю работу с памятью, используя RAII и STL-контейнеры, при этом оставаясь независимым от высокоуровневых механизмов языков с JIT или встроенным GC.

3.3 Жизненный цикл аллокации

Пользователь взаимодействует с системой через функции:

- `gc_malloc`, `gc_calloc`, `gc_realloc`, `gc_free` — аналоги стандартных функций.
- Все объекты автоматически регистрируются в GC с помощью метода `CreateAllocation`, где сохраняется указатель, размер, финализатор и внутренний «временной маркер» `timer_`.

Удаление объектов возможно как вручную (через `gc_free`), так и автоматически в ходе сборки мусора.

3.4 Стратегия сборки: Stop-the-World

Реализация использует модель **Stop-the-World** при сборке мусора. Это означает, что во время выполнения фазы сборки (маркировки и удаления) все пользовательские потоки временно приостанавливаются. Данный подход выбран по следующим причинам:

- **Суммарная экономия CPU ресурсов.** В отличие от конкурентных GC, которые требуют введения write barriers (барьеров записи) и постоянного мониторинга изменений в памяти, Stop-the-World позволяет проводить сборку полностью в одном потоке без дополнительных оверхедов. Это объясняется тем, что большая часть действий с памятью в реальных программах – чтение и запись, а не аллокация, деаллокация.
- **Простота реализации и отладки.** Модель с полной остановкой потоков упрощает синхронизацию и избегает сложных состояний гонки, характерных для параллельных и инкрементальных GC.

- **Надёжность.** Все изменения в памяти происходят в контролируемом режиме, что делает систему более устойчивой и предсказуемой, а также более удобной для последующих изменений.

На практике, несмотря на паузы, Stop-the-World GC часто оказывается быстрее и стабильнее в типичных сценариях, особенно при наличии короткоживущих объектов и небольших heap'ов.

3.5 Управление корнями

GC работает на основе понятия **GC-Root** — это участок памяти, в котором могут находиться указатели на управляемые объекты. Пользователь обязан явно регистрировать такие области через функции `gc_add_root` и `gc_delete_root` либо при инициализации `gc_init`. Это позволяет сборщику точно знать, где искать активные ссылки.

3.6 Автоматическая сборка и автонастройка

В системе реализован адаптивный механизм триггера сборки мусора. Компонент `GCPacer` отслеживает:

- количество выделенных байтов и число аллокаций;
- скорость выделения памяти (мгновенное и сглаженное значения).

На основе этих данных `GCScheduler` принимает решение о запуске сборки. Сборка может быть вызвана, если:

- превышен порог по аллокациям/байтам;
- обнаружен пик активности (например, резкое увеличение аллокаций).

Все пороги можно узнать и поменять с помощью предоставленного api, также можно отключить автоматическую сборку и очищать память только посредством вызова `gc_collect`

3.7 Параллельность и многопоточность

Сборщик реализует безопасную поддержку многопоточности. Для этого:

- Каждый поток, использующий GC, должен быть явно зарегистрирован с помощью функции
  `gc_register_thread`, а при завершении — удалён через `gc_deregister_thread`.

- Во время сборки выполняется механизм **Stop-the-World**, при котором активные потоки приостанавливаются с помощью условной переменной, и GC может безопасно выполнять обход и очистку памяти.
- Потоки обязаны периодически вызывать функцию `gc_safepoint()` — специальную контрольную точку. Она может вызываться в произвольных местах кода. Если поток ни разу не вызовет safepoint, сборщик не сможет приостановить выполнение и, соответственно, не сможет начать сборку мусора.
- Используются мьютексы и атомарные переменные для синхронизации потоков и корректного завершения фазы сборки.

Также реализована маркировка в виде параллельного обхода объектов с помощью очередей со стилем work-stealing — это позволяет эффективно масштабироваться при большом количестве потоков. Но данный механизм маркировки не выбран основным, так как имеет большую деградацию в производительности на большом количестве gc roots, на маленьком количестве потоков, а также на маленьком количестве памяти.

3.8 Интерфейс и интеграция

Библиотека предоставляет полноценный C-интерфейс, не требующий использования классов. Это позволяет легко интегрировать сборщик в низкоуровневые проекты на C и C++. Кроме того, предусмотрены функции управления поведением GC в рантайме: включение/отключение автоматической сборки, установка порогов и интервалов.

3.9 Объёмные характеристики

- Объём кода: $\sim$2000 строк.
- Число тестов: более 20 модульных и многопоточных тестов.
- Присутствуют 5 сценариев бенчмарк тестирования, один из которых - симуляция работы с памятью, одновременно и стресс тестирование.
- Поддерживаемые режимы: ручной GC, автоматический GC с автонастройкой, многопоточный обход.

4 Экспериментальный анализ и тестирование

4.1 Цели тестирования

Целью экспериментального анализа является:

- Проверка корректности работы сборщика мусора в различных сценариях: однопоточные и многопоточные аллокации, работа с корнями, циклические ссылки, realloc и пр.
- Измерение производительности: скорость аллокации, время сборки, эффективность автоматической и ручной сборки.
- Анализ влияния различных эвристик и параметров (байтовые и количественные пороги, частота обновления, всплески аллокаций).
- Демонстрация масштабируемости и многопоточности.

4.2 Подход к тестированию

Тестирование разделено на два направления:

1. **Юнит-тесты**, реализованные на базе `Google Test`. Охватывают:
   - простую аллокацию и освобождение;
   - сохранение объектов, достижимых через корни;
   - корректную работу при realloc и массивных выделениях;
   - сборку циклических ссылок;
   - автоматическую и ручную сборку в многопоточной среде;
   - изменение параметров GC во время работы.
2. **Бенчмарки**, реализованные через `Google Benchmark`, направленные на измерение времени аллокации, сборки и общего throughput.

4.3 Результаты производительности

Ниже приведены результаты запусков различных тестов и нагрузочных сценариев.

Realloc-тест

- 100 последовательных `realloc()` на одном объекте (64–128 байт).
- Время на итерацию: 43.4 μs (CPU time), 82.3 μs (реальное).
- Скорость: $2.3 \cdot 10^6$ аллокаций/с.

Drop 5%

- 10 000 объектов, каждая сборка «теряет» 5% указателей.
- Время сборки: 28.6 μs CPU, 35.0 μs wall-time.
- Скорость обработки: ∼350 млн объектов/сек.

DropProbability Sweep (от 0% до 100%)

Таблица 4.1: Результаты теста GC с различной вероятностью утери ссылок (DropProbability). Замерено время полной сборки мусора и количество обработанных элементов в секунду.

Drop, %	**Время (мкс)**	**Пропускная способность (items/sec)**
0%	1854	5.52M
5%	2411	4.18M
15%	2377	4.25M
30%	2129	4.63M
50%	1922	5.34M
70%	1556	6.69M
100%	420	23.55M

Вывод: по мере увеличения вероятности удаления мусора, время сборки уменьшается, а throughput увеличивается — это ожидаемо, так как остаётся меньше достижимых объектов, и проход по heap упрощается.

Сложные действия с памятью (SimulateActions)

- Смоделированы 200 случайных действий над 10 000 объектами.
- Время: 2581 μs CPU, 2599 μs wall-time.
- Тест охватывает чтение, запись, удаление, перезапись указателей.

Таблица 4.2: Производительность авто-GC при различных размерах и количестве объектов. Замерено общее время на цикл и число итераций в секунду.

Размер задачи	**Время (мкс)**	**Итераций в секунду**
1 000 000 × 32–64 байт	24 632	29
1 000 000 × 64–512 байт	102 282	5
500 000 × 1–4 КБ	56 447	11
100 000 × 1–8 КБ	42 225	14

Автоматическая сборка на больших массивах

Тестирование сборки при включённом авто-GC:

Таблица 4.2 демонстрирует, что сборщик мусора адекватно масштабируется при росте как объёма, так и числа объектов. Более крупные объекты увеличивают суммарное время, но не приводят к деградации системы или зависаниям.

4.4 Многопоточность и безопасность

- Проверены сценарии с 8 потоками, одновременно выполняющими аллокации и сборки.
- Поддерживается регистрация/удаление потоков во время работы.
- Потоки корректно реагируют на `StopTheWorld` через `gc_safepoint()`.

Все тесты завершились успешно, без утечек памяти и гонок. Тестирование было проведено с разными санитайзерами (`address, thread, undefined, valgrind`)

4.5 Выводы

Эксперименты показали:

- Высокую производительность и масштабируемость при росте числа объектов.
- Эффективную работу авто-GC при разных сценариях.
- Корректность и стабильность даже при интенсивной многопоточной нагрузке.
- Полное покрытие типичных и сложных ситуаций: циклические ссылки, массивы, realloc, потоковые гонки.

Таким образом, реализация сборщика мусора показывает достойные характеристики как в тестах на корректность, так и в производительных нагрузках.

5 Оптимизации и улучшения производительности

5.1 Профилирование и выявление узких мест

Для анализа производительности сборщика мусора в процессе прохождения бенчмарков был использован инструмент низкоуровневого профилирования — **Perforator**, который позволяет визуализировать вызовы функций в виде флеймграфа (flamegraph). Данный подход позволил локализовать ключевые узкие места в работе GC.

Главным боттлнеком оказалась функция `FindAllocation` — она отвечает за поиск объекта в множестве аллокаций по произвольному указателю. Эта операция выполняется многократно в процессе маркировки (`MarkRoots`, `MarkHeapAllocs`) и критична к производительности.

5.2 Проблема точного поиска

Первоначально для хранения аллокаций использовалась структура `std::map`, предоставляющая логарифмическое время доступа. Однако классический поиск по ключу невозможен в данной задаче: пользовательский указатель может указывать не на начало объекта, а на произвольную внутреннюю часть (например, после `ptr + k`). Таким образом, GC должен искать вхождение по диапазону: найти такую аллокацию, что `alloc.ptr <= ptr < alloc.ptr + alloc.size`.

В силу этого нельзя было использовать хеш-таблицу, а `std::map` оказалась узким местом по времени и кэш-локальности.

5.3 Переход на вектор с сортировкой

Была реализована следующая замена:

- Вместо `std::map` используется `std::vector<Allocation>`,
  в котором перед каждой сборкой аллокации сортируются по `ptr`.
- Для поиска применяется бинарный поиск с модификацией: после нахождения ближайшей левой границы проверяется, входит ли указатель внутрь диапазона аллокации.

Несмотря на одинаковую асимптотику поиска ($O(\log n)$), на практике такой подход дал более чем **двухкратное ускорение** только этой функции за счёт лучшей кэш-локальности и компактного представления. Также ускорились все остальные стадии сборки мусора и функции GC, так как операции с `std::vector` намного быстрее, чем с std::map.

5.4 Эвристики для ускорения поиска

Для ускорения `FindAllocation` были реализованы две эвристики:

Фильтрация по heap-диапазону Поскольку корневые области (`GCRoots`) часто представляют собой стек или неинициализированную память, они содержат множество произвольных значений, не являющихся валидными указателями. Эти значения, как правило, значительно меньше чем адреса в heap.

Была добавлена предварительная проверка: если значение указателя явно вне диапазона heap, поиск не запускается. Такая оптимизация уменьшает количество ложных вызовов `FindAllocation`.

Кэширование последнего результата поиска Вторая эвристика основывается на том, что при обходе памяти множество указателей имеют пространственную локальность. Поэтому вводится поле `prev_find_`, которое запоминает последнюю успешную аллокацию и позволяет ускорить следующий поиск, если он попадает в тот же диапазон. Такой подход дал дополнительное **двукратное ускорение** поиска.

5.5 Оптимизация сортировки аллокаций

После ускорения поиска стало очевидно, что значительное время начала занимать операция `SortAllocations`, выполняемая перед каждой сборкой. Изначально сортировка выполнялась полностью каждый раз, что имело сложность $O(n \log n)$.

Реализована следующая схема:

- Поддерживается индекс `last_size_`, указывающий, до какого места вектор уже отсортирован.
- Новые аллокации добавляются в конец вектора, и сортируется только эта часть.
- Затем применяется алгоритм **in-place merge** из сортировки слиянием, объединяющий отсортированные части в один массив за $O(n)$.

Такой подход дал **троекратное ускорение** операции сортировки на больших объёмах.

5.6 Итоговый эффект

Оптимизации привели к следующим результатам:

- `FindAllocation()` ускорена в 6 раз по сравнению с исходной реализацией на `std::map`. Тогда как общая производительность возросла почти в 10 раз.
- Общая фаза `Mark` стала занимать существенно меньше времени, особенно при большом количестве указателей.
- Время `Collect` стабилизировалось и стало менее чувствительно к росту числа аллокаций.
- Все изменения были протестированы и подтверждены на реальных бенчмарках (см. таблицы 4.1, 4.2).

Эти улучшения обеспечили масштабируемость системы и подготовили основу для будущих расширений.

6 Сравнение с аналогами и обоснование новизны

6.1 Существующие решения

Механизмы сборки мусора присутствуют во многих современных языках программирования и виртуальных машинах. Ниже перечислены основные существующие реализации:

- **CPython (Python)** использует подсчёт ссылок (`Reference Counting`) как основной механизм, дополнительно реализуя поколенческий сборщик мусора для циклических ссылок.
- **Java Virtual Machine (JVM)** поддерживает множество GC-алгоритмов: Serial GC, G1, Shenandoah, ZGC, включая concurrent и generational GC. Реализация тесно связана с байткодом и компилятором.
- **.NET (C#)** использует поколенческий Mark-Compact GC с конкурентными возможностями для серверных приложений.
- **Go** применяет concurrent Mark-Sweep GC, заточенный под low-latency сценарии.
- **Rust / C++** предоставляют только ручное управление памятью, иногда с использованием умных указателей (`shared_ptr`, `Rc`, `Arc`), что не является полноценным GC.

Во всех перечисленных случаях реализация GC встроена в компилятор или рантайм, тесно завязана на типовую информацию, байткод и внутреннюю структуру языка. Это делает их либо недоступными, либо избыточными для внедрения в проекты на C/C++.

6.2 Позиционирование данного проекта

Разработанный сборщик мусора представляет собой самостоятельную, независимую библиотеку GC с ручной интеграцией в проекты на C и C++. Его ключевые отличия:

- **Не требует поддержки со стороны компилятора**. Полностью работает в пространстве пользовательского кода, не нуждается в знании типов и метаинформации.
- **Реализует stop-the-world Mark-Sweep GC** с многопоточной параллельной маркировкой и ручным управлением корнями.
- **Имеет гибкий runtime-интерфейс** для настройки порогов авто-GC и контроля сборки в процессе выполнения программы.

- **Полноценная многопоточность с учётом safepoint'ов и синхронизации**.
- **Полный тестовый и бенчмаркинговый стек**: от юнитов до стрессов, от функциональных проверок до метрик throughput.

6.3 Новизна и вклад

Несмотря на то, что базовый алгоритм Mark-Sweep хорошо известен, представленный проект обладает рядом аспектов, отличающих его от стандартных реализаций:

1. **Форм-фактор:** легко встраиваемая C/C++-библиотека, ориентированная на независимое использование без JVM/CLR.
2. **Архитектура:** модульное разбиение на GCImpl / GCScheduler / GCPacer облегчает расширение и модификации.
3. **Высокоэффективная реализация:**
   - оптимизированный поиск `FindAllocation` с ускорением;
   - эвристики фильтрации и кэширования;
   - частичная сортировка аллокаций с линейным слиянием.
4. **Доказанная эффективность:** бенчмарки демонстрируют сборку 10 000 объектов менее чем за 40 мкс, пропускную способность до 23 млн объектов/сек.
5. **Исследовательский вклад:** проект показывает, как можно реализовать эффективный GC без поддержки типов и без связки с JIT/VM, используя лишь стандартные механизмы языка.

6.4 Возможности повторного использования

Разработанная библиотека может быть применена в:

- Создании интерпретаторов или DSL-языков,
  где требуется простое автоматическое управление памятью.
- Встраиваемых системах или проектах на C++, нуждающихся в безопасной и автоматической очистке памяти без утечек.
- Образовательных целях — как демонстрация реализации базового и расширяемого GC.

- Исследовательских работах, связанных с сравнением GC-стратегий, оптимизацией runtime-алгоритмов и синхронизацией потоков.

7 Заключение

В рамках курсового проекта был реализован полностью функциональный сборщик мусора, предназначенный для использования в проектах на языке C/C++. Работа охватывала как теоретическое исследование существующих алгоритмов, так и практическую реализацию, профилирование, оптимизацию и экспериментальную валидацию решения.

Основные достижения проекта

- Разработана масштабируемая архитектура сборщика мусора, включающая независимые компоненты:
  - `GCImpl` — управление аллокациями и памятью;
  - `GCScheduler` — контроль авто-сборки;
  - `GCPacer` — логика авто-сборки.
- Реализован эффективный Mark-Sweep GC с полной поддержкой многопоточности и ручным управлением корнями.
- Введены **safepoint**-механизмы и stop-the-world-сборка с корректной синхронизацией потоков.
- Проведено масштабное профилирование и оптимизация.
- Поддержан гибкий runtime-интерфейс управления GC (ручной и автоматический режим, установка порогов, интервалы и прочее).
- Разработан полноценный стек тестов: юнит-тесты, многопоточные сценарии, нагрузочные бенчмарки.

Практическая значимость

Проект представляет интерес как для встраивания в прикладные C++ проекты, так и в качестве базы для разработки языков программирования, интерпретаторов и образовательных инструментов. Он демонстрирует, что эффективный сборщик мусора может быть реализован без поддержки со стороны компилятора и без необходимости использования внешней виртуальной машины.

Ограничения и перспективы развития

Несмотря на достигнутые результаты, работа GC ограничена следующими моментами:

- используется только один алгоритм (Mark-Sweep), без поддержки компактизации или поколенческого подхода;
- сборка осуществляется в stop-the-world-режиме;
- управление корнями требует ручной и явной регистрации.

В качестве перспектив возможны следующие направления развития:

- внедрение поколенческого GC;
- добавление поддержки копирующих или компактифицирующих алгоритмов;
- экспериментальная реализация инкрементальной или concurrent-сборки;
- интеграция с компилятором/анализатором для автоматического сбора корней;
- расширение API для поддержки слабых ссылок, финализаторов, внешней трассировки.

Итог

Разработанный сборщик мусора демонстрирует высокую производительность, масштабируемость, корректность и простоту интеграции. Проект может служить как практическим инструментом, так и основой для дальнейших исследований в области автоматического управления памятью.

Список литературы

[1] Python Software Foundation. *gc — Garbage Collector interface*. URL: https://docs.python.org/3/library/gc.html (дата обр. 31.01.2025).

[2] Google. *A Guide to the Go Garbage Collector*. URL: https://tip.golang.org/doc/gc-guide#Additional_notes_on_GOGC (дата обр. 31.01.2025).

[3] Francisco De Melo Junior. *A beginner's guide to the Shenandoah garbage collector*. URL: https://developers.redhat.com/articles/2024/05/28/beginners-guide-shenandoah-garbage-collector# (дата обр. 30.01.2025).

[4] Microsoft Learn. *Garbage collection*. URL: https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/ (дата обр. 31.01.2025).

[5] Oracle. *Java Platform, Standard Edition HotSpot Virtual Machine Garbage Collection Tuning Guide, Release 21*. URL: https://docs.oracle.com/en/java/javase/21/gctuning/index.html#GUID-6C8D4E24-A580-4FEA-82F0-FE610057DD15 (дата обр. 31.01.2025).

[6] Antony Hosking Richard Jones и Eliot Moss. *The Garbage Collection Handbook*. 2nd edition. CRC Press, 2023.