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 План дальнейшей работы

3.1 Этап 1: Разработка базового GC (Mark-Sweep)

- Проектирование архитектуры библиотеки GC.
- Определение API библиотеки:
  - `gc_init` – инициализация GC и передача GC roots (глобальные переменные, стеки).
  - `gc_malloc` – выделение памяти под объекты.
  - `gc_collect` – сборка мусора с использованием алгоритма Mark-Sweep.
- Реализация механизма отслеживания GC roots:
  - Определение структуры данных для хранения GC roots.
  - Реализация механизма регистрации и удаления GC roots.
- Реализация алгоритма **Mark-Sweep**:
  - Фаза **Mark**: обход объектов от GC roots и пометка достижимых.
  - Фаза **Sweep**: освобождение неотмеченных объектов.
- Разработка базовых тестов для проверки работы GC.
- Оптимизация и исправление ошибок.

3.2 Этап 2: Добавление триггеров для автоматического запуска GC

- Определение критериев для автоматического запуска `gc_collect`:
  - Заполнение выделенной памяти (например, если выделено больше X байт).
  - Временные интервалы (например, сборка каждые N мс).
  - Количество аллокаций (`gc_malloc` вызывается определённое число раз).
- Реализация триггеров.
- Оптимизация частоты вызова GC (чтобы не снижалась производительность программы).
- Написание тестов для проверки работы триггеров.

3.3 Этап 3: Поддержка многопоточности

- Анализ возможных проблем многопоточного доступа
- Использование мьютексов или написание lock free версии
- Реализация потокобезопасной версии `gc_malloc` и `gc_collect`.
- Тестирование многопоточной работы GC.

3.4 Этап 4: Дальнейшее развитие GC

Вариант 1: Встраивание GC в собственный интерпретатор или компилятор.

- Разработка простого интерпретатора (или интеграция с существующим).
- Адаптация GC под работу с интерпретатором.

Вариант 2: Реализация более сложного алгоритма GC:

- **Mark-Compact**: реализация алгоритма с дефрагментацией памяти.
- **Generational GC**: реализация поколенческого GC с разделением объектов на молодое и старое поколения.

Вариант 3: Оптимизация производительности:

- Улучшение стратегии выделения памяти.
- Оптимизация стадий Mark, Sweep
- Тестирование других алгоритмов

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.