



Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н. Э. Баумана
(национальный исследовательский университет)
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ”DV и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

*«Метод автоматического управления памятью с
гарантированным временем выполнения на основе
подсчёта ссылок»*

Студент

ИУ7-83Б

А. М. Сапожков

(Подпись, дата)

Руководитель

Ю. В. Строганов

(Подпись, дата)

Нормоконтролер

Д. Ю. Мальцева

(Подпись, дата)

2024 г.

РЕФЕРАТ

Расчетно–пояснительная записка 90 с., 17 рис., 6 табл., 59 ист., 1 прил.

Объектом исследования является работа с памятью в языках программирования. Цель работы — спроектировать и реализовать метод автоматического управления памятью с гарантированным временем выполнения на основе подсчёта ссылок.

Реализованный менеджер памяти основан на модели поколений и использует конкурентную и параллельную сборку мусора. Использование разработанного метода позволило снизить время выполнения алгоритмов, относящихся к классам VP и VE, на 0.3-56.5% и 12-20.7% соответственно в зависимости от длины входа.

Применение новых методов автоматического управления памятью может повысить эффективность приложений, в особенности разработанных для длительной непрерывной работы.

СОДЕРЖАНИЕ

РЕФЕРАТ	3
ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ	7
ВВЕДЕНИЕ	9
1 Аналитический раздел	11
1.1. Управление памятью с точки зрения операционной системы	11
1.1.1. Иерархия памяти	11
1.1.2. Функции операционной системы по управлению памятью .	12
1.2. Управление памятью с точки зрения приложений	12
1.2.1. Среда выполнения языка программирования	12
1.2.2. Управление памятью	13
1.2.2.1. Ручное управление памятью	15
1.2.2.2. Автоматическое управление памятью	17
1.2.3. Трассирующая сборка мусора	18
1.2.4. Подсчёт ссылок	20
1.3. Существующие решения в области автоматического управления памятью	21
1.3.1. Python	21
1.3.1.1. Структура объекта	21
1.3.1.2. Сборка мусора	22
1.3.2. Java	23
1.3.2.1. Управление памятью	23
1.3.2.2. Сборка мусора	24
1.3.3. Javascript	26
1.3.4. C#	27
1.3.4.1. Управление памятью	27
1.3.4.2. Сборка мусора	28
1.3.5. Golang	30
1.3.5.1. Структура кучи	30
1.3.5.2. Сборка мусора	31
1.3.6. Классификация существующих решений	33

1.4. Классификация алгоритмов по требованиям к дополнительной памяти	35
1.5. Постановка задачи	38
2 Конструкторский раздел	40
2.1. Основные концепции менеджера памяти	40
2.2. Проектирование алгоритмов распределения памяти	42
2.2.1. Представление дескриптора объекта	42
2.2.2. Алгоритм выделения памяти	43
2.2.3. Алгоритм сбора циклических ссылок	45
2.2.3.1. Разметка	45
2.2.3.2. Очистка	50
2.2.3.3. Перераспределение объектов между поколениями	52
2.3. Выбор структуры данных для хранения дескрипторов объектов	53
2.4. Выбор подхода к реализации метода	54
3 Технологический раздел	56
3.1. Выбор языка программирования	56
3.2. Аренды памяти	57
3.3. Компоненты программного обеспечения	57
3.3.1. Выделение памяти в куче	59
3.3.2. Выделение памяти в поколении	60
3.3.3. Сбор мусора в поколении	62
3.3.3.1. Разметка	62
3.3.3.2. Очистка	66
3.3.3.3. Перераспределение объектов между поколениями	68
3.4. Оценка трудоёмкости алгоритмов	68
3.4.1. Модель вычислений	69
3.4.2. Трудоёмкость реализации алгоритма выделения памяти	70
3.4.3. Трудоёмкость реализации алгоритма обращения к объекту	70
3.5. Руководство пользователя	71

4 Исследовательский раздел	74
4.1. Цель исследования	74
4.2. Описание исследования	74
4.3. Технические характеристики оборудования	74
4.4. Результаты исследования	75
4.4.1. Алгоритм конвейерной обработки данных	75
4.4.2. Алгоритм сортировки слиянием	76
4.4.3. Алгоритм нахождения расстояния Левенштейна	78
4.4.4. Алгоритм умножения матриц по Винограду	79
4.4.5. Алгоритм сортировки подсчётом	81
ЗАКЛЮЧЕНИЕ	83
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	89
ПРИЛОЖЕНИЕ А	90

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

В настоящей расчетно-пояснительной записке применяют следующие термины с соответствующими определениями.

- 1) Стек** — область адресного пространства процесса, предназначенная для хранения параметров функций, локальных переменных и адреса возврата после вызова функции.
- 2) Куча** — область адресного пространства процесса, предназначенная для выделения памяти, динамически запрашиваемой программой.
- 3) Внутренняя фрагментация** — явление, при котором аллокатор выделяет при каждом запросе больше памяти, чем фактически запрошено.
- 4) Внешняя фрагментация** — явление, при котором свободная память разделена на множество мелких блоков, ни один из которых нельзя использовать для обслуживания запроса на выделение памяти.
- 5) Управление памятью** — процесс координации и контроля использования памяти в вычислительной системе.
- 6) Автоматическое управление памятью** — служба, которая автоматически перерабатывает память, которую программа не собирается использовать в дальнейшем
- 7) Мусор** — объект, о котором достоверно известно, что он не используется программой, так как является недостижимым¹ из других её объектов.
- 8) Сбор мусора** — автоматическая переработка динамически выделяемой памяти.
- 9) Сборщик мусора** — автоматический менеджер памяти, реализующий некоторый алгоритм сборки мусора.
- 10) Мутатор** — пользовательская программа, которая изменяет граф объектов.

¹Термин «недостижимость» может иметь различные интерпретации в разных языках программирования

11) Барьер — функция, которая принимает указатель на объект в памяти, анализирует его статус и в зависимости от него выполняет какие-либо действия с этим указателем или даже с объектом, на который он ссылается, после чего возвращает актуальное значение указателя, которое можно использовать для доступа к объекту.

12) Поколение — группа объектов со схожим временем жизни

13) Финализатор объекта — функция, которая запускается, когда сборщик мусора определяет, что рассматриваемый объект недоступен в программе.

14) Финализация объекта — процесс выполнения финализаторов объекта.

ВВЕДЕНИЕ

Память является одним из основных ресурсов любой вычислительной системы, требующих тщательного управления. Под памятью (memory) в работе будет подразумеваться оперативная память компьютера. Особая роль памяти объясняется тем, что процессор может выполнять инструкции программы только в том случае, если они находятся в памяти. Память распределяется между операционной системой компьютера и прикладными программами. [1]

Для выполнения вычислений в языках программирования используются объекты, которые могут быть представлены как простым типом данных (целые числа, символы, логические значения и т.д.) так и агрегированным (массивы, списки, деревья и т.д.). Значения объектов программ хранятся в памяти для быстрого доступа. Во многих языках программирования переменная в программном коде — это адрес объекта в памяти. [2] [3] [4] Когда переменная используется в программе, процесс считывает значение из памяти и обрабатывает его.

Большинство современных языков программирования использует динамическое распределение памяти, при котором выделение объектов осуществляется во время выполнения программы. Динамическое управление памятью вводит два основных примитива — функции выделения и освобождения памяти, за которые отвечает **аллокатор**.

Существует два способа управления динамической памятью — ручное и автоматическое. При ручном управлении памятью программист должен следить за освобождением выделенной памяти, что приводит к возможности возникновения ошибок. Более того, в некоторых ситуациях (например, при программировании на функциональных языках или в многопоточной среде) время жизни объекта не всегда очевидно для разработчика. [5] Автоматическое управление памятью избавляет программиста от необходимости вручную освобождать выделенную память, устранивая тем самым целый класс возможных ошибок и увеличивая безопасность разрабатываемых программ. Сборка мусора (garbage collection) за последние два десятилетия стала стандартом в области автоматического управления памятью, хотя её использование накладывает дополнительные расходы памяти и времени исполнения. На сегодняшний день среди времени выполнения (language runtime) многих популярных языков программирования, таких как Java, C#, Python и другие, активно используют сбор-

ку мусора.

Современные приложения, в особенности разработанные для длительной непрерывной работы, подвержены влиянию следующих факторов:

- 1) утечки памяти;
- 2) фрагментация памяти;
- 3) длительные паузы на сборку мусора.

Перечисленные факторы могут негативно сказываться на производительности и отказоустойчивости приложений. Снизить влияние данных факторов на работу программ и повысить эффективность работы с памятью можно на основе новых качеств методов автоматического управления памятью.

Целью данной работы является разработка метода автоматического управления памятью с гарантированным временем выполнения на основе подсчёта ссылок. Для достижения поставленной цели необходимо решить следующие задачи.

1. Проанализировать существующие методы распределения памяти в языках программирования с автоматической сборкой мусора.
2. Спроектировать метод автоматического управления памятью.
3. Реализовать спроектированный метод в виде подключаемой библиотеки.
4. Определить области применения реализованного метода.

1 Аналитический раздел

Управление памятью вычислительной системы выполняется на трёх уровнях.

1. Аппаратное обеспечение управления памятью (MMU, ОЗУ и т. д.).
2. Управление памятью операционной системы (виртуальная память, защищена).
3. Управление памятью приложения (выделение и освобождение памяти, сборка мусора).

Аппаратное обеспечение управления памятью состоит из электронных устройств и связанных с ними схем, которые хранят состояние вычислительной системы. К этим устройствам относятся регистры процессора, кэш, ОЗУ, MMU (Memory Management Unit, блок управления памятью) и вторичная (дисковая) память. Конструкция запоминающих устройств имеет решающее значение для производительности современных вычислительных систем. Фактически пропускная способность памяти является основным фактором, влияющим на производительность системы. [6]

Далее будет рассмотрено подробно управление памятью с точки зрения операционной системы и приложений.

1.1. Управление памятью с точки зрения операционной системы

1.1.1. Иерархия памяти

В процессе развития аппаратного обеспечения была разработана концепция **иерархии памяти**, согласно которой компьютеры обладают несколькими мегабайтами очень быстродействующей, дорогой и энергозависимой кэш-памяти, несколькими гигабайтами памяти, средней как по скорости, так и по цене, а также некоторыми терабайтами памяти на довольно медленных, сравнительно дешевых дисковых накопителях, не говоря уже о сменных накопителях, таких как DVD и флеш-устройства USB. Превратить эту иерархию в абстракцию, то есть в модель, а затем управлять этой абстракцией — и есть задача операционной системы. Та часть операционной системы, которая управляет иерархией памяти (или ее частью), называется **менеджером**, или **диспетчером**,

памяти [1]. Он должен следить за тем, какие части памяти используются, выделять память процессам, которые в ней нуждаются, и освобождать память, когда процессы завершат свою работу. Выбор, совершаемый менеджером памяти на этом этапе, может оказать существенное влияние на будущую эффективность программы, так как до 40% (в среднем 17%) времени её выполнения затрачивается на выделение и освобождение памяти. [7]

1.1.2. Функции операционной системы по управлению памятью

Помимо первоначального выделения памяти процессам при их создании операционная система должна также заниматься динамическим распределением памяти, то есть обслуживать запросы приложений на выделение им дополнительной памяти во время выполнения. После того, как приложение перестаёт нуждаться в дополнительной памяти, оно может вернуть её системе. Выделение случайного объёма памяти в случайные моменты времени из общего пула приводит к фрагментации памяти и, вследствие этого, к неэффективному её использованию. Дефрагментация памяти также является функцией операционной системы.

Таким образом, можно сформулировать следующие функции ОС по управлению памятью в мультипрограммных системах.

1. Отслеживание свободной и занятой памяти.
2. Управление виртуальными адресными пространствами процессов, в том числе отображение виртуального адресного пространства процесса на физическую память.
3. Динамическое распределение памяти.
4. Дефрагментация памяти.

1.2. Управление памятью с точки зрения приложений

1.2.1. Среда выполнения языка программирования

Менеджер памяти операционной системы с точки зрения выполняющихся на ней приложений обладает двумя серьёзными недостатками. Во-первых,

он является недетерминированным: процессы не могут напрямую влиять на решения, принимаемые операционной системой по управлению их адресными пространствами. Во-вторых, менеджер памяти операционной системы не реализует те функции управления памятью, которые требуются конечному пользователю. Поэтому языки программирования реализуют собственную **среду выполнения** (language runtime), в которой реализуют необходимые пользователю функции для работы с памятью и создают собственные абстракции памяти.

Среда выполнения языка программирования может выполнять следующие функции: [8]

- 1) управление памятью;
- 2) обработка исключений;
- 3) сборка мусора;
- 4) контроль типов;
- 5) обеспечение безопасности;
- 6) управление потоками.

Управление потоками в среде выполнения языка подразумевает, что язык программирования организует многопоточное выполнение программы, используя возможности операционной системы, а также реализует некоторую **модель памяти**, которая для императивных языков программирования, поддерживающих многопоточное выполнение, определяет, какие записи в разделяемые переменные могут быть видны при чтениях, выполняемых другими потоками. [9]

1.2.2. Управление памятью

Менеджер памяти приложения должен учитывать следующие ограничения. [10]

1. **Нагрузка на процессор** — дополнительное время, затрачиваемое диспетчером памяти во время работы программы.
2. **Блокировки** — время, необходимое диспетчеру памяти для завершения операции и возврата управления программе. Это влияет на способность

программы оперативно реагировать на интерактивные события, а также на любое асинхронное событие, например связанное с сетевым взаимодействием.

3. **Накладные расходы памяти** — дополнительный объём памяти, затрачиваемый на администрирование, а также накладные расходы, связанные с внутренней и внешней фрагментацией.

Основная проблема управления памятью состоит в том, чтобы понять, когда следует сохранить содержащиеся в ней данные, а когда их следует уничтожить, чтобы память можно было использовать повторно. К сожалению, существует множество способов, которыми плохая практика управления памятью может повлиять на надежность и быстродействие программ, как при ручном, так и при автоматическом управлении памятью.

Ниже перечислены наиболее частые проблемы управления памятью. [10]

1. **Преждевременное освобождение памяти** (premature free). Когда программы освобождают память, но позже пытаются получить к ней доступ, их поведение становится неопределенным. Сохранившаяся ссылка на освобожденную память называется **висячим указателем** (dangling pointer).
2. **Утечки памяти** (memory leaks). Когда программы интенсивно выделяют память, не освобождая её, в конечном итоге память заканчивается.
3. **Внешняя фрагментация** (external fragmentation). Работа аллокатора по выделению и освобождению памяти может привести к тому, что он больше не сможет выделять достаточно большие области памяти, несмотря на наличие достаточного общего количества свободной памяти. Это связано с тем, что свободная память может быть разделена на множество небольших областей, разделённых используемыми данными. Данная проблема особенно критична в серверных приложениях, работающих в течение длительного времени.
4. **Неэффективное расположение ссылок** (poor locality of reference). Эта проблема связана с тем, как современные менеджеры памяти аппаратного обеспечения и операционной системы обращаются с памятью: после-

довательные обращения к памяти выполняются быстрее, если они производятся к соседним областям памяти. Если менеджер памяти разместит данные, которые программа будет использовать вместе, не в соседних областях памяти, то быстродействие программы уменьшится.

5. **Негибкий дизайн** (*inflexible design*). Менеджеры памяти могут вызвать серьезные проблемы с производительностью, если они были разработаны с одной целью, а используются для другой. Эти проблемы вызваны тем, что любое решение по управлению памятью опирается на предположения о том, как программа будет использовать выделенную память. Например, на стандартные размеры областей памяти, шаблоны ссылок или время жизни объектов. Если эти предположения неверны, то диспетчер памяти может работать с памятью менее эффективно.
6. **Межмодульное взаимодействие** (*interface complexity*). Если объекты передаются между модулями программы, то при проектировании интерфейсов модулей необходимо учитывать управление их памятью.

Управление памятью приложения объединяет две взаимосвязанные задачи: выделение памяти (*allocation*) и её переиспользование (*recycling*), когда она больше не требуется. За выделение памяти отвечает **аллокатор** [11]. Его необходимость обусловлена тем, что процессы, как правило, не могут заранее предсказать, сколько памяти им потребуется, поэтому они нуждаются в реализации дополнительной логики обслуживания изменяющихся запросов к памяти. Решение об освобождении и переиспользовании выделенной аллокатором памяти, которая больше не используется приложением, может быть принято либо программистом, либо средой выполнения языка. Соответственно, в зависимости от этого управление памятью в языке программирования может считаться либо ручным, либо автоматическим.

1.2.2.1. Ручное управление памятью

При ручном управлении памятью программист имеет прямой контроль над временем жизни объектов программы. Как правило, он осуществляется либо явными вызовами функций управления кучей (например, `malloc` и `free` в C), либо языковыми конструкциями, влияющими на стек управления (например,

объявлениями локальных переменных). Ключевой особенностью ручного менеджера памяти является то, что он даёт возможность явно указать, что заданная область памяти может быть освобождена и переиспользована. [10]

Преимущества ручного управления памятью:

- явное выделение и освобождение памяти делает программы более прозрачными для разработчика;
- ручные менеджеры памяти, как правило, используют память более экономно, так как программист может минимизировать время между моментом, когда выделенная память перестаёт использоваться, и её фактическим освобождением.

Недостатки ручного управления памятью:

- увеличение исходного кода программ за счёт того, что управление памятью, как правило, составляет значительную часть интерфейса любого модуля;
- повышение дублирования кода за счёт использования однотипных инструкций управления памятью;
- увеличение числа ошибок управления памятью из-за человеческого фактора.

К языкам с ручным управлением памятью относятся C, C++, Zig и другие. На таких языках программисты могут писать код, дублирующий поведение менеджера памяти либо путем выделения больших областей и их разделения для использования, либо путем внутреннего переиспользования этих блоков. Такой код называется **субаллокатором** (suballocator) [6], так как он работает поверх другого аллокатора. Субаллокаторы могут использовать как преимущество специальные знания о поведении программы, но в целом они менее эффективны, чем использование базового аллокатора. Также стоит отметить, что субаллокаторы могут быть неэффективными или ненадежными, тем самым создавая новый источник ошибок. [11]

1.2.2.2. Автоматическое управление памятью

Автоматическое управление памятью, как правило, работает либо как часть среды выполнения языка, либо как расширение. Автоматические менеджеры памяти обычно перерабатывают области памяти, которые недоступны из переменных программы (то есть области, к которым невозможно получить доступ с помощью указателей). Задачи автоматического управления памятью: [10]

- выделение памяти под новые объекты;
- идентификация используемых объектов;
- освобождение памяти, занятой неиспользуемыми объектами.

Преимущества автоматического управления памятью:

- освобождение разработчика от задачи управления памятью в своих программах;
- уменьшение объёма исходного программ за счёт отсутствия необходимости явно работать с памятью;
- уменьшение числа ошибок управления памятью;
- автоматическое управление памятью может быть более эффективным, чем ручное, за счёт применения соответствующих алгоритмов.

Недостатки автоматического управления памятью:

- время жизни объектов программы становится непрозрачным для разработчика, так как часть логики работы с памятью реализована на уровне языка;
- память, как правило, используется менее экономно, так как существует некоторая задержка между моментом, когда память перестаёт использоваться, и моментом её освобождения; такие задержки определяются алгоритмами автоматической переработки памяти.

Автоматическое управление памятью используется в большинстве современных языков программирования, среди которых C#, Golang, Haskell, Java, JavaScript, Python, Swift, частично Rust, C++ при использовании умных указателей и другие.

В языках с автоматическим управлением памятью часто необходимо выполнять действия над некоторыми объектами после того, как они перестали использоваться, и до того, как память, которую они занимают, может быть переиспользована. Для этого используется механизм финализации объектов, который, как правило, используется для освобождения ресурсов, когда работающий с ними объект перестаёт использоваться. Например, открытый файл может быть представлен объектом потока ввода-вывода. Когда менеджер памяти подтверждает, что этот объект больше не используется в программе, то можно быть гарантировать, что файл больше не используется программой и его нужно закрыть до повторного использования потока. Стоит отметить, что момент финализации объекта программы, как правило, не фиксирован. Этот факт может стать проблемой, если финализация используется для управления ограниченными ресурсами операционной системы, такими как файловые дескрипторы. [6]

Как правило, автоматическая сборка мусора реализуется либо методом подсчёта ссылок, либо с использованием трассирующего сборщика мусора. [12] Целью идеального сборщика мусора является освобождение пространства, используемого каждым объектом, как только он перестаёт использоваться программой. Стоит отметить, что для автоматического управления памятью также можно использовать гибридные методы. [13] [14]

1.2.3. Трассирующая сборка мусора

Трассирующий сборщик мусора [12] — автоматический менеджер памяти, который следует указателям, чтобы определить, какие области памяти доступны из переменных программы, называемых **корневым набором**). Далее будут описаны наиболее популярные алгоритмы сборки мусора. Стоит отметить, что алгоритмы могут сочетаться и заменяться во время выполнения программы в зависимости от параметров кучи, таких как заполненность и фрагментация. [15]

Алгоритм **mark-sweep** реализует рекурсивное определение достижимости объекта по указателям. Сборка мусора осуществляется в два этапа. Сначала

сборщик обходит граф объектов, начиная с «корней» (регистры, стеки потоков, глобальные переменные), через которые программа могла бы немедленно получить доступ к объектам, а затем, следуя указателям, сканирует каждый найденный объект. Такой обход называется **трассировкой**. На втором этапе, этапе очистки (sweep), сборщик проверяет каждый объект в куче: любой неразмеченный объект считается мусором и освобождается. [15]

Mark-sweep — это алгоритм косвенного сбора данных. В отличие от косвенных методов, прямые алгоритмы, такие как подсчёт ссылок, определяют достижимость объекта исключительно по самому объекту, без обращения к трассировке. [15]

Рассматриваемый алгоритм не обнаруживает мусор как таковой, а скорее идентифицирует все используемые объекты, а затем приходит к выводу, что всё остальное должно быть мусором. Стоит заметить, что сборщику необходимо размечать используемые объекты при каждом вызове. Для обеспечения согласованности при работе с памятью сборщик мусора, реализующий алгоритм mark-sweep, не должен работать параллельно или конкурентно с основной программой. Такой режим работы сборщика называют **«остановкой мира»** («stop the world»). [15]

Длительно работающие приложения, управляемые неперемещающими сборщиками мусора (например, работающими по алгоритму mark-sweep), могут фрагментировать кучу, что отрицательно скажется на производительности приложений. Для устранения внешней фрагментации предлагается две стратегии. Первая, которой придерживается алгоритм mark-compact, заключается в уплотнении используемых объектов кучи, вторая — в перемещении объектов из одной области памяти в другую. Основное преимущество уплотнения кучи заключается в том, что она обеспечивает относительно быстрое последовательное распределение, просто проверяя ограничение кучи и находя свободный указатель, соответствующий запросу на выделение. [15]

Алгоритм **mark-compact** работает в несколько этапов. Первая фаза — фаза разметки. Затем дальнейшие этапы уплотнения сжимают используемые данные путем перемещения объектов и обновления значений указателей всех ссылок на объекты, которые были перемещены. Количество обходов кучи, порядок, в котором они выполняются, и способ перемещения объектов могут зависеть от реализации. Порядок уплотнения влияет на местоположение данных. [15]

1.2.4. Подсчёт ссылок

Подсчёт ссылок предполагает отслеживание количества указателей на определённые области памяти из других областей. Он используется в качестве основы для некоторых методов автоматической переработки, которые не требуют отслеживания (трассировки). [12]

Системы подсчёта ссылок выполняют автоматическое управление памятью, сохраняя в каждом объекте, обычно в заголовке, число ссылок на данный объект. Объекты, на которые нет ссылок (счётчик ссылок равен нулю), не могут быть доступны вызывающей стороне. Следовательно, они не используются и могут быть переработаны. [6]

Преимущества подсчёта ссылок:

- накладные расходы по управлению памятью распределяются по всему времени работы программы; [15]
- устойчивость к высоким нагрузкам, так как потенциально подсчёт ссылок может переработать объект как только он перестаёт использоваться; [15]
- масштабируемость по объёму кучи, так как накладные расходы, как правило, зависят только от количества выполняемых операций с указателями на объекты, а не от объема хранимых данных; [15]
- может быть реализован без помощи среды выполнения языка и без её ведома. [15]

Недостатки подсчёта ссылок:

- требуются накладные расходы на операции чтения и записи для управления числом ссылок;
- операции с числом ссылок на объекты должны быть атомарными; [15]
- случай циклических ссылок требует отдельного рассмотрения; [15]
- подсчёт ссылок может вызывать паузы, например при освобождении ссылочных структур данных. [15]

По сравнению с трассирующей сборкой мусора алгоритмы управления памятью, основанные на подсчёте ссылок, отличаются детерминированностью, предсказуемостью и меньшей вычислительной сложностью. [13]

Подсчёт ссылок наиболее полезен в ситуациях, когда можно гарантировать отсутствие циклических ссылок и сравнительно редкие модификации ссылочных структур данных. Такие условия могут иметь место в некоторых типах структур баз данных и некоторых файловых системах. Подсчёт ссылок также может быть полезен, если важно, чтобы объекты утилизировались своевременно, например, в системах с жёсткими ограничениями памяти. [12]

1.3. Существующие решения в области автоматического управления памятью

1.3.1. Python

Python [16] — интерпретируемый язык программирования с сильной динамической типизацией. Официально язык был представлен в 1991 году. Существует множество реализаций интерпретатора языка Python [17] [18] [19]. Среди них канонической считается реализация CPython, разработанная на языке С и официально представленная в 1994 году. [20]

1.3.1.1. Структура объекта

Все объекты программы на языке Python размещаются в **приватной куче** (private heap), управление которой обеспечивается встроенным в интерпретатор диспетчером памяти. [21]

Основной алгоритм сборки мусора, используемый CPython, — подсчёт ссылок. Для его реализации в структуре объекта Python предусмотрены поля, хранящие число ссылок и данные о типе объекта (**PyObject_HEAD**), а также указатели на элементы двусвязного списка объектов, отслеживаемых сборщиком мусора (**PyGC_Head**). Структура объекта Python представлена на рисунке 1.1. [22]

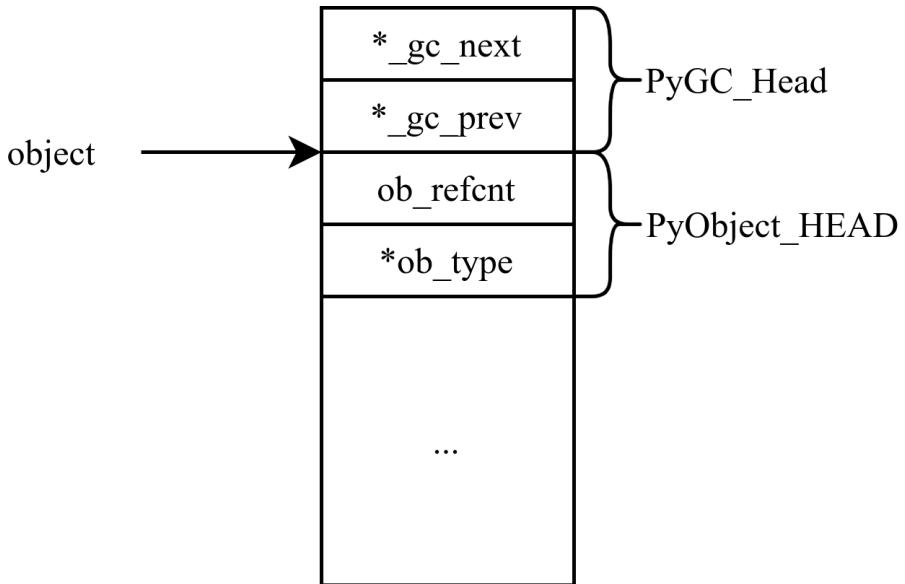


Рис. 1.1 – Структура объекта Python

Когда необходима дополнительная информация, связанная со сборщиком мусора, к полям PyGC_Head можно получить доступ с помощью адресной арифметики и приведения типа исходного объекта. [22]

Двусвязные списки используются по той причине, что они эффективно поддерживают операции, наиболее часто используемые при выполнении алгоритма сбора мусора, такие как перемещение объекта из одного раздела в другой, добавление нового объекта, полное удаление объекта, а также разбиение и объединение списков. [22]

1.3.1.2. Сборка мусора

Основная проблема подсчёта ссылок заключается в том, что он не обрабатывает циклические ссылки. Для её решения используется отдельный сборщик мусора, который занимается только очисткой объектов-контейнеров, то есть объектов, которые могут содержать ссылки на другие объекты. [22]

Алгоритм сборки мусора, который обрабатывает циклические ссылки, состоит в обходе двусвязного списка объектов и использовании метода пробного удаления [15]: объекты, счётчик ссылок на которые не превышает 1, помещаются в список условно недостижимых. Объекты, которые остались в этом списке после завершения обхода графа объектов, могут быть освобождены. [22]

Стоит отметить, что такая сборка мусора выполняется конкурентно с мутатором в одном выделенном потоке сборщика, не распределяя работу по сборке на несколько потоков приложения. [23]

1.3.2. Java

Java [24] — компилируемый язык программирования с сильной статической типизацией. Официально был представлен в 1995 году компанией Sun Microsystems.

Среда выполнения языка Java (Java Runtime Environment, JRE) состоит из **виртуальной машины Java** (Java Virtual Machine, JVM), основных классов и вспомогательных библиотек платформы Java. Рассматриваемый язык является кроссплатформенным: приложения, разработанные на Java, компилируются в **байт-код**, который сохраняется в файлах классов и запускается в JVM. Поэтому программы на языке Java можно запустить на всех аппаратных plataформах и операционных системах, для которых реализована JVM. [24]

Существует множество реализаций JVM. [25] [26] [27] Среди них наиболее распространённой является HotSpot JVM, представленная в 1999 году компанией Sun Microsystems. [28]

1.3.2.1. Управление памятью

Частью JVM является **менеджер хранилища** (storage manager), который отвечает за управление жизненным циклом объектов Java: выделение новых объектов, сбор недоступных объектов и отправку уведомлений о недоступности по запросу. [29] Из соображений масштабируемости, каждый поток JVM имеет собственную область выделения памяти — **буфер выделения потока** (Thread-Local Allocation Buffer, TLAB). Каждый поток может выделять ресурсы из своей TLAB без координации с другими потоками, за исключением случаев, когда ему требуется новая TLAB. [29]

Было замечено, что большинство программ на языке Java следует **гипотезе поколений** [15], согласно которой в большинстве случаев «молодые» объекты (которые создаются позже), с гораздо большей вероятностью перестают использоваться раньше, чем «старые» объекты.. Чтобы учесть это свойство программ при распределении памяти, объекты Java разделяются на три поколения: молодое (young), старое (old) и постоянное (permanent). Управление поколениями может осуществляться по различным алгоритмам. Предполагается, что в молодом поколении будет больше объектов, чем в старом, а также будет выше плотность недоступных объектов. [29]

Молодое поколение должно поддерживать быстрое распределение, при этом ожидается, что большинство этих объектов относительно быстро станет недоступным. При сборе мусора в молодом поколении выявляются все достижимые объекты и копируются в старое поколение. Далее объекты, оставшиеся в молодом поколении, освобождаются. [29]

Объекты в **старом поколении** могут анализироваться сборщиком мусора реже, чем объекты в молодом поколении. Также сборщик мусора в зависимости от выбранной стратегии выполнять копирование и уплотнение объектов. [29] [30]

Помимо объектов, созданных программой на языке Java, существуют объекты, созданные и используемые JVM. Чтобы не путать их с объектами выполняемой программы, такие объекты выделяют в **постоянное поколение**. Данное название носит исторический характер и на самом деле объекты в нём не существуют на протяжении всего времени выполнения программы. [29] Начиная с Java 8 постоянное поколение было заменено **метапространством** (MetaSpace), предназначенным для хранения классов и метаданных JVM. [31]

1.3.2.2. Сборка мусора

HotSpot JVM предоставляет несколько сборщиков мусора, оптимизированных для различных типов приложений. Далее будут рассмотрены сборщики мусора, доступные в Java 21. [32]

Serial Collector работает по алгоритму mark-compact (см. п. 1.2.3.) и использует один поток для выполнения основной и второстепенной работы по сбору мусора, что может повысить его эффективность за счёт отсутствия накладных расходов на взаимодействие параллельных потоков. [32]

Parallel Collector отличается от Serial Collector использованием нескольких потоков для сборки мусора, количество которых может регулироваться с помощью аргументов командной строки. [32] Во время работы Parallel Collector все потоки мутатора приостанавливаются («stop the world», см. п. 1.2.3.). [33] Данный сборщик мусора также называют Throughput Collector, так как он может использовать несколько процессоров для ускорения сборки мусора и повышения пропускной способности при работе с памятью, что повышает производительность приложений. [24]

Garbage-First Collector (также G1) — это сборщик мусора, работающий

по алгоритму mark-sweep (см. п. 1.2.3.), предназначенный для многопроцессорных машин с большим объемом памяти. Он с высокой вероятностью соответствует целевому времени паузы для сборки мусора, обеспечивая при этом относительно высокую пропускную способность. Операции над всей кучей, такие как глобальная разметка объектов, выполняются конкурентно с потоками приложения. Это предотвращает паузы в работе приложений, пропорциональные размеру кучи и используемых данных. Начиная с версии Java 9, Garbage-First Collector выбирается по умолчанию в большинстве конфигураций оборудования и операционной системы или может быть явно включён с помощью параметров командной строки. Garbage-First Collector достигает высокой производительности при снижении длительности пауз за счет применения следующих методов. [34]

1. В отличие от Parallel Collector работа потоков приостанавливается только на некоторые этапы сборки мусора: в начале при разметке объектов из корневого набора (см. п.1.2.3.) и в конце при обнаружении изменений, произошедших за время сборки мусора. [33]
2. Куча разделяется на набор непрерывных областей одинакового размера, называемых регионами. После обнаружения используемых объектов в куче Garbage-First Collector определяет, какие регионы «почти пусты» (mostly empty). Именно в этих регионах в первую очередь будет выполняться сбор мусора и уплотнение неиспользуемых объектов, чтобы как можно раньше дать аллокатору непрерывные свободные области памяти наибольшего размера.
3. Используется **модель прогнозирования паузы** (pause prediction model) для достижения заданного пользователем целевого времени паузы на сборку мусора и на его основе выбирает количество регионов для обработки.
4. В отличие от Parallel Collector используемые объекты выбираются для копирования и уплотнения не из всей кучи, а только из некоторых её областей. Эта операция выполняется параллельно на нескольких процессорах, чтобы уменьшить время паузы на сборку мусора и увеличить пропускную способность. Таким образом, при каждой сборке мусора G1 непрерывно

работает над уменьшением фрагментации кучи, работая в течение заданного пользователем времени паузы.

Z Garbage Collector (также ZGC) — это сборщик мусора, работающий по алгоритму mark-compact (см. п. 1.2.3.) и нацеленный на минимизацию пауз. ZGC работает конкурентно с мутатором, не останавливая выполнение потоков приложения более чем на одну миллисекунду, при этом жертвуя пропускной способностью. Он предназначен для приложений, которым требуется низкая задержка на сборку мусора. Время пауз не зависит от размера используемых данных в куче. ZGC оптимизирован для работы с кучей размером от нескольких сотен мегабайт до 16 Тб. [35]

1.3.3. Javascript

JavaScript [36] — интерпретируемый язык программирования со слабой динамической типизацией, представленный в 1995 году. Стандартом языка является ECMAScript. [37] Наиболее широкое применение JavaScript получил в качестве языка сценариев веб-страниц, но также используется и в других программных продуктах, таких как node.js [38] и Apache CouchDB [39].

Модель времени выполнения (runtime model) в JavaScript основана на **цикле событий** (event loop), который отвечает за сбор и обработку событий, а также за выполнение подзадач из очереди (queued sub-tasks). [40]

Изначально интерпретаторы JavaScript использовали сборку мусора на основе подсчёта ссылок, не обрабатывая циклические ссылки, что могло приводить к систематическим утечкам памяти. По этой причине, начиная с 2012 года, все современные веб-браузеры оснащаются сборщиками мусора, работающими исключительно по алгоритму mark-sweep (см. п. 1.2.3.). Все усовершенствования алгоритма сборки мусора в интерпретаторах JavaScript (инкрементальная, конкурентная и параллельная сборка мусора, а также применение алгоритма поколений) за последние несколько лет представляют собой усовершенствования данного алгоритма, не являясь новыми алгоритмами сборки мусора, поскольку дальнейшее сужение понятия «объект более не используется» не представляется возможным. [41]

1.3.4. C#

C# [42] — компилируемый язык программирования с сильной статической типизацией. Официально был представлен в 2001 году компанией Microsoft.

Программы на языке C# предназначены для запуска на виртуальной среде выполнения .NET, работающей с **общязыковой средой выполнения** (Common Language Runtime, CLR) и набором библиотек классов. Среда CLR — это реализация **общязыковой инфраструктуры языка** (Common Language Infrastructure, CLI), являющейся международным стандартом компании Microsoft. [42] Основные функции среды CLR: [43] [44]

- сборка мусора;
- обеспечение безопасности памяти и типов данных;
- поддержка языков программирования высокого уровня: C#, F# и Visual Basic;
- изоляция программ.

1.3.4.1. Управление памятью

Сборщик мусора среды Common Language Runtime управляет выделением и освобождением памяти для приложения. При инициализации нового процесса среда выполнения резервирует для него непрерывную область адресного пространства, называемого **управляемой кучей** (managed heap), которая является частью кучи адресного пространства процесса и предназначена для размещения всех ссылочных типов данных. Выделение памяти из управляемой кучи происходит быстрее, чем неуправляемое выделение памяти, так как среда выполнения выделяет память путём изменения указателя на следующий объект в куче, что аналогично выделению памяти на стеке. Кроме того, последовательно хранение объектов позволяет ускорить доступ к ним. Также стоит отметить, что все потоки процесса выделяют память в одной куче. [45]

Для повышения производительности среда выполнения выделяет память для больших объектов (размером не менее 85 000 байт) в отдельной куче, называемой **кучей больших объектов** (Large Object heap, LOH). Также во избе-

жение перемещения больших объектов в памяти куча больших объектов, как правило, не подлежит уплотнению. [46]

1.3.4.2. Сборка мусора

Сборщик мусора определяет оптимальное время для выполнения сбора на основе выделяемых ресурсов. Перед началом сборки мусора все управляемые потоки приостанавливаются, за исключением потока, который инициировал сборку. Период времени, в течение которого сборщик мусора активен, называется его **задержкой** (delay). Сборщик мусора работает по алгоритму mark-compact (см. п. 1.2.3.), разделяя сбор мусора на три фазы. [47]

1. **Фаза разметки:** определение неиспользуемых объектов путём исследования корневого набора программы (см. п. ??), каждый элемент которого либо ссылается на объект в управляемой куче, либо имеет значение null. Корневой набор включает статические поля, локальные переменные и параметры в стеке потока, а также регистры процессора. Сборщик мусора имеет доступ к списку активных «корней», который поддерживается JIT-компилятором и средой выполнения. С помощью этого списка исследуется корневой набор и строится граф доступных объектов.
2. **Фаза перемещения:** корректировка указателей, чтобы после сжатия используемых данных элементы корневого набора ссылались на новое местоположение объектов.
3. **Фаза уплотнения:** освобождение объектов, которые не попали в граф на этапе разметки, считаются недоступными и освобождаются. На данном этапе сборщик мусора проверяет управляемую кучу в поисках областей памяти, занятых недоступными объектами. При обнаружении каждого недостижимого объекта он использует функцию копирования для уплотнения доступных объектов, тем самым освобождая области памяти, выделенные для недоступных объектов. Он также устанавливает указатель управляемой кучи за последним доступным объектом.

Сбор мусора гарантированно происходит при выполнении одного из следующих условий. [47]

1. В системе обнаружена нехватка физической памяти. Данная ситуация определяется либо по уведомлению от операционной системы, либо по приближении к пределу объёма физической памяти, доступной на вычислительной машине.
2. Объём памяти, используемый объектами в управляемой куче, превысил допустимое пороговое значение, которое может корректироваться во время выполнения программы.
3. Сборщик мусора вызван явно с помощью метода GC.Collect.

Работа сборщика мусора основана на следующих соображениях. [47]

- Уплотнение для части управляемой кучи выполняется быстрее, чем для всей управляемой кучи.
- Новые объекты, как правило, обладают меньшим временем жизни по сравнению со старыми объектами.
- Новые объекты, как правило, связаны друг с другом и используются приложением примерно в одно и то же время.

Для повышения производительности сборщика мусора и снижения среднего времени паузы на сборку, управляемая куча разделена на три поколения, обрабатываемые отдельно. Поскольку сжать часть управляемой кучи быстрее, чем всю кучу, эта схема позволяет сборщику мусора освобождать память только в определенном поколении, а не во всей управляемой куче при каждом вызове. [47] [45]

Поколение 0 содержит самые новые и, как правило, недолговечные объекты, такие как локальные переменные. Сбор мусора происходит чаще всего именно в этом поколении, при этом зачастую освобождается достаточно памяти, чтобы приложение могло продолжать выделять память. Объекты, оставшиеся после сборки мусора поколения 0, переводятся в поколение 1.

Поколение 1 служит буфером между новыми и старыми объектами. Если сбор мусора в поколении 0 не освобождает достаточно памяти для того, чтобы приложение могло создать новый объект, сборщик мусора может выполнить сборку в поколении 1, а затем в поколении 2. Объекты, оставшиеся после сборки мусора поколения 1, переводятся в поколение 2.

Поколение 2 содержит старые объекты, такие как статические данные, используемые на протяжении всего времени выполнения программы. Объекты, попавшие в поколение 2, остаются в нём до тех пор, пока они не перестанут использоваться в программе. Также при сборе мусора в поколении 2 выполняется сбор объектов в куче больших объектов (иногда называется поколением 3).

1.3.5. Golang

Golang (также Go) [4] — компилируемый язык программирования с сильной статической типизацией, разработанный компанией Google. Официально язык был представлен в ноябре 2009 года.

В любой программе на языке Golang присутствует **среда выполнения** (runtime) [48], то есть код, который выполняется без ведома программиста. Среда выполнения состоит из следующих компонентов.

1. **Планировщик** (Scheduler) [49], отвечающий за управление работой горутин. **Горутина** — это легковесный поток, управляемый средой выполнения языка Golang. [50]
2. **Аллокатор памяти.**
3. **Сборщик мусора.**

1.3.5.1. Структура кучи

Куча состоит из набора областей, называемых **аренами** (Arena), размер которых составляет 64 Мб в 64-разрядной версии и 4 Мб в 32-разрядной (heapArenaBytes). Начальный адрес каждой арены также выровнен по размеру арены. С каждой арендой связан объект heapArena [51], который хранит метаданные для этой аренды: битовую карту кучи для всех слов (word) в арене и карту span для всех страниц в ней. Объекты heapArena выделяются вне кучи. Структура mheap содержит **карту аренд**, которая охватывает всё доступное адресное пространство программы, поэтому его можно рассматривать как набор аренд. Аллокатор старается сохранять аренды смежными, чтобы большие span (и, следовательно, большие объекты) могли занимать одновременно несколько аренд. [52]

1.3.5.2. Сборка мусора

В языке Golang для сборки мусора используется алгоритм **Concurrent Mark-Sweep**, являющийся модификацией алгоритма mark-sweep (см. п. 1.2.3.), предназначенного для конкурентной сборки мусора. Сборщик мусора запускается параллельно с мутатором и позволяет нескольким потокам сборки мусора выполнять параллельно. Алгоритм сборки мусора не является уплотняющим и не использует поколения (см. п. 1.3.2.1.). [53]

Сборщик мусора языка Golang использует трёхцветную абстракцию, в рамках которой граф объектов разбивается на черные (предположительно используемые) и белые (возможно, неиспользуемые) объекты. Изначально каждый объект белый. Когда объект впервые обнаруживается во время трассировки, он окрашивается в серый цвет. Когда объект был отсканирован и были идентифицированы его дочерние элементы, он закрашивается чёрным. [15]

Для конкурентной разметки и очистки используются **барьеры записи** (write barrier), позволяющий избежать ситуации, когда чёрные объекты указывают на белые. Такое может произойти, например, при перемещении указателя в программе до того, как сборщик мусора успел его разметить. В такой ситуации мутатор скрывает объект от сборщика мусора. [53] Для реализации барьера записи используются операции «затенения» (shade), перемещающие указатели программы. Барьер записи «затеняет» как записываемый указатель, так и записываемое значение при любой записи по указателю [54].

Ниже представлено описание основных шагов алгоритма сборки мусора. [53]

1. Фаза завершения очистки (sweep termination).
 - 1.1. «Остановка мира» («stop the world», см. п. 1.2.3.), приводящая к тому, что все потоки достигают **безопасной точки** (GC safe-point).
 - 1.2. Очистка всех мусорных mspan. Неочищенные mspan будут только в том случае, если цикл сборки мусора был запущен раньше ожидаемого времени.
2. Фаза разметки (mark).
 - 2.1. Включение барьера записи, постановка в очередь заданий по разметке объектов из корневого набора (см. п. 1.2.3.) и включение ассистен-

тов (assistants), выполняющих свою работу во время выделения памяти. Никакие объекты не могут быть просканированы до тех пор, пока все потоки не включат барьер записи, что достигается с помощью «остановки мира».

- 2.2. «Запуск мира» («start the world»). С этого момента работа сборщика мусора выполняется обработчиками разметки (mark workers), запущенными планировщиком, и ассистентами. Выделяемые объекты отмечаются чёрным цветом.
 - 2.3. Разметка объектов из корневого набора, включающая в себя сканирование стеков всех горутин, «затенение» всех глобальных переменных и любых указателей кучи в структурах данных среды выполнения вне кучи. При сканировании стека горутина останавливается, «затеняются» все указатели, найденные в её стеке, а затем горутина продолжает выполняться.
 - 2.4. Очистка рабочей очереди от серых объектов. Каждый серый объект сканируется и отмечается чёрным, «затеняя» все указатели, найденные в объекте, что, в свою очередь, может привести к добавлению этих указателей в рабочую очередь.
 - 2.5. Выполнение алгоритма распределенного завершения (distributed termination algorithm), чтобы определить, когда закончится выполнение заданий разметки объектов из корневого набора или серых объектов.
3. Фаза завершения разметки (mark termination).
 - 3.1. «Остановка мира».
 - 3.2. Отключение обработчиков и ассистентов.
 - 3.3. Очистка, включающая сброс кешей mcache.
 4. Фаза очистки (sweep).
 - 4.1. Отключение барьера записи.
 - 4.2. «Запуск мира». С этого момента выделяемые объекты становятся белыми. Также при необходимости аллокатор может очищать msan перед использованием.

4.3. Выполнение параллельной очистки в фоновом режиме и во время выделения памяти.

1.3.6. Классификация существующих решений

Ниже приведены критерии классификации алгоритмов распределения памяти.

1. Разделение объектов на поколения.

Для оптимизации сборки мусора и уменьшения времени пауз в языках программирования может использоваться алгоритм поколений (см. п. 1.3.2.1.). В таблице сравнения будут использоваться обозначения «+» и «-» для определения использования алгоритма поколений менеджером памяти рассматриваемого языка программирования.

2. Отсутствие хранения вспомогательных данных в объектах.

Для реализации алгоритмов распределения памяти может понадобиться хранение дополнительных данных в выделяемых объектах, что создаёт зависимость объёма памяти, необходимого сборщику мусора, от количества отслеживаемых объектов. В таблице сравнения для обозначения факта хранения вспомогательных данных в объектах будет использоваться «-», иначе — «+».

3. Использование конкурентной сборки мусора.

Для снижения времени пауз на сборку мусора менеджеры памяти некоторых языков программирования могут выполнять её конкурентно с потоками мутатора. В таблице сравнения для обозначения факта использования конкурентной сборки мусора будет использоваться «+», иначе — «-».

4. Использование параллельной сборки мусора.

Менеджеры памяти некоторых языков программирования могут выполнять сборку мусора параллельно в нескольких потоках приложения, задействуя больше вычислительных ресурсов процессора для её ускорения. В таблице сравнения для обозначения факта использования параллельной сборки мусора будет использоваться «+», иначе — «-».

5. Отсутствие остановки потоков мутатора на весь цикл сборки мусора.

Некоторые алгоритмы сборки мусора оптимизированы таким образом, чтобы останавливать работу потоков основной программы только на некоторых этапах сбора для минимизации времени пауз. В таблице сравнения для обозначения этого факта будет использоваться «+», иначе — «-».

6. Количество остановок потоков мутатора за один цикл сборки мусора.

Описанные выше характеристики алгоритмов распределения памяти, как правило, декларируются в документациях к соответствующим языкам программирования. Результаты сравнения алгоритмов распределения памяти приведены в таблице 1.1.

Таблица 1.1 – Сравнение алгоритмов распределения памяти

Язык программирования		Сборщик мусора					
		Разделение объектов на поколения	Отсутствие хранения вспомогательных данных в объектах	Использование конкурентной сборки мусора	Использование параллельной сборки мусора	Отсутствие остановки потоков мутатора на весь цикл сборки мусора	Количество останововок потоков мутатора за один цикл сборки мусора
Python	По умолчанию	+	-	+	-	+	1
Java	Serial	+	+	+	-	-	1
	Parallel	+	+	+	+	-	1
	Garbage-First	+	+	+	+	+	2
	ZGC	+	+	+	+	+	1
JavaScript	По умолчанию	-	+	-	-	-	1
C#	По умолчанию	+	+	+	+	-	1
Golang	По умолчанию	-	+	+	+	+	2

Среди рассмотренных интерпретируемых языков программирования наи-

более оптимизированным для различных сценариев использования можно считать менеджер памяти языка Python за счёт применения алгоритма поколений и подсчёта ссылок.

Среди рассмотренных компилируемых языков программирования наиболее универсальным и масштабируемым можно считать менеджер памяти языка Java за счёт предоставления пользователю возможности выбора сборщика мусора для выполнения каждой программы, а также оптимизации времени пауз на сборку мусора.

1.4. Классификация алгоритмов по требованиям к дополнительной памяти

Теоретическое исследование ресурсных характеристик компьютерных алгоритмов предполагает введение соответствующих классификаций, отражающих различные ресурсные требования алгоритма и его программной реализации. [55]

Далее будет рассмотрена классификация компьютерных алгоритмов, в основу которой положен требуемый алгоритмом объём дополнительной памяти в принятой модели вычислений. В данном случае рассматривается только объём оперативной памяти, так как затраты памяти на внешних носителях требуют отдельного рассмотрения и выходят за рамки работы. Пусть D — конкретный вход алгоритма A , причем $|D| = n$, где n — длина или некоторая мера длины входа. Под объёмом памяти, требуемым алгоритмом A для входа, заданного множеством D , будем понимать максимальный объём памяти модели вычислений, задействованный в ходе выполнения алгоритма. Функцию объёма памяти алгоритма для входа D будем обозначать через $V_A(D)$. Значение функции $V_A(D)$ есть целое положительное число. [55]

Ресурсные требования алгоритма к объёму памяти определяются памятью входа, выхода и дополнительной памятью, задействованной алгоритмом в ходе его выполнения. В соответствии с этим будем различать следующие компоненты функции объема памяти: память входа $V_I(D)$, память выхода $V_R(D)$ и дополнительная память $V_t(D)$. По объёму памяти входа (выхода) алгоритмы можно разделить на потоковые, обрабатывающие входные (выходные) данные поэлементно, и непотоковые, хранящие входные (выходные) данные в памяти целиком. [55]

Поскольку ресурсные компоненты $V_I(D)$ и $V_R(D)$ во многом определяются особенностями задачи, а для различных алгоритмов ее решения, обладающих статическими входом и выходом, фактически совпадают, то именно компонент $V_t(D)$ различает алгоритмы по ресурсным затратам памяти в рамках статического или потокового типа. Очевидно, что для современных компьютеров ограничение решаемых задач по ресурсу оперативной памяти не столь существенно, как ограничения по временной эффективности. Тем не менее известная дилемма «память-время» приводит к тому, что попытка улучшения временных характеристик приводит к ощутимым затратам памяти, даже при современных объемах. Актуальные размерности современных сложных задач также приводят к тому, что ресурс памяти становится значимым в комплексной оценке ресурсной эффективности алгоритма. Однако при сравнении алгоритмов, относящихся или к статическому, или к потоковому типу, основную роль играет именно дополнительная память, затраты которой обусловлены спецификой данного алгоритма. В связи с этим предлагается следующая классификация алгоритмов, основанная на оценке дополнительной памяти. [55]

Класс V0 — алгоритмы с нулевой дополнительной памятью.

$$\forall n > 0, \forall D \in D_n \quad V_t(D) = 0. \quad (1.1)$$

Алгоритмы этого класса либо вообще не требуют дополнительной памяти, либо используют ресурсы памяти входа и/или выхода в качестве необходимой дополнительной памяти по мере обработки элементов входа или по мере заполнения памяти результата. [55]

Класс VC — алгоритмы с фиксированной дополнительной памятью.

$$\forall n > 0, \forall D \in D_n \quad V_t(D) = const \neq 0. \quad (1.2)$$

Алгоритмы класса VC используют постоянный, не зависящий от длины и особенностей входа и длины выхода, объем дополнительной оперативной памяти. В реальных алгоритмах — это, как правило, память для хранения счетчиков циклов, промежуточных результатов вычислений и указателей на структуры. [55]

Класс VL — алгоритмы, дополнительная память которых линейно зависит от длины входа. Введем в рассмотрение функцию дополнительной памяти в худшем случае для всех допустимых входов с мерой n , обозначив её через

$V_t^*(n)$:

$$V_t^*(n) = \max_{D \in D_n} V_t(D). \quad (1.3)$$

В этих обозначениях алгоритм принадлежит к типу с линейной дополнительной памятью, если

$$V_t^*(n) = \Theta(n), \quad (1.4)$$

где обозначение Θ есть стандартное обозначение класса функций в асимптотическом анализе. [55]

Класс VQ — алгоритмы, дополнительная память которых квадратично зависит от длины входа. Для этого типа алгоритмов объём дополнительной памяти в худшем случае для входов размерности n пропорционален по порядку квадрату меры размерности:

$$V_t^*(n) = \Theta(n^2). \quad (1.5)$$

Примером может служить стандартный алгоритм умножения длинных целых чисел, заданных побитно массивами длины n . [55]

Класс VP — алгоритмы, дополнительная память которых имеет полиномиальную надквадратичную зависимость. Это алгоритмы, требующие ресурса дополнительной памяти, по порядку большего, чем квадрат меры длины входа, но полиномиально зависящего от этой меры:

$$\exists k > 2 : V_t^*(n) = \Theta(n^k). \quad (1.6)$$

К этому классу относятся, например, рекурсивные алгоритмы, порождающие дерево рекурсии с оценкой глубины $\Theta(n^k)$, $k > 1$, и требующие хранения в каждой вершине дерева дополнительного массива, имеющего длину порядка $\Theta(n)$. [55]

Класс VE — алгоритмы, дополнительная память которых экспоненциально зависит от длины входа. Формально это тип алгоритмов, требующих, в худшем случае, ресурса дополнительной памяти, по порядку экспоненциально зависящего от меры длины входа:

$$\exists \lambda > 0 : V_t^*(n) = \Theta(e^{\lambda n}). \quad (1.7)$$

Реально это алгоритмы, использующие дополнительные массивы или более сложные структуры данных, размер которых определяется значениями элементов входа. Характерным примером могут служить алгоритм сортировки методом индексов или табличные алгоритмы, реализующие метод динамического программирования. [55]

1.5. Постановка задачи

Алгоритмы автоматического управления памятью в каждом из языков программирования обладают различными недостатками, среди которых однопоточная или неконкурентная сборка мусора, необходимость остановки потоков мутатора для её выполнения, а также отсутствие разделения объектов на поколения. Поэтому разрабатываемый метод должен быть ориентирован на устранение недостатков существующих решений, возникающих при реализации алгоритмов, относящихся к классам VL, VQ, VP, VE, а также VC при условии, что дополнительная память для работы такого алгоритма будет выделяться в куче, а не на стеке (например, во избежание переполнения стека).

На рисунке 1.2 представлена диаграмма IDEF0 нулевого уровня решаемой задачи.

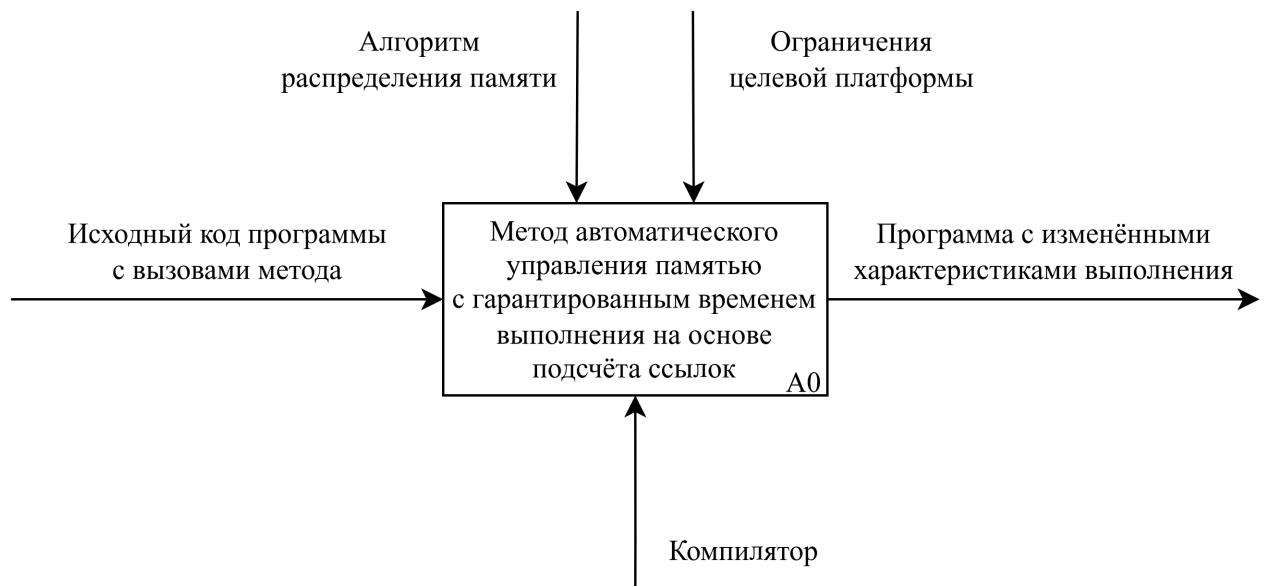


Рис. 1.2 – Диаграмма IDEF0 нулевого уровня

Под временем выполнения метода понимается время, которое пользовательская программа затрачивает на вызовы реализации метода, то есть на выде-

ление памяти и обращение к выделенным объектам. Гарантия времени выполнения метода подразумевает, что это время будет иметь некоторое ограничение, выражаемое асимптотической оценкой сложности алгоритмов выделения памяти и обращения к ней.

Подсчёт ссылок позволяет при освобождении памяти распределять накладные расходы по всему времени работы программы. При этом сбор циклических ссылок должен осуществляться конкурентно с основной программой, чтобы снизить накладные расходы мутатора на работу с объектами.

Выводы из аналитического раздела

В данном разделе был проведён анализ предметной области автоматического управления памятью. Были проведены анализ и классификация существующих решений. На основе классификации методов была сформулирована концепция комбинированного метода и обозначена область его применения. Постановка задачи была формализована в виде функциональной схемы метода в нотации IDEF0.

2 Конструкторский раздел

2.1. Основные концепции менеджера памяти

В общем случае, эффективность менеджера памяти можно представить в виде следующей регрессионной модели:

$$y = \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + \beta_4 x_4 + \beta_5 x_5 + \beta_6 x_6, \quad (2.1)$$

где $\beta_1 - \beta_6$ — коэффициенты регрессии, $x_1 - x_6$ — факторы модели:

1. **Задержка мутатора при выполнении операций выделения памяти и получения доступа к объектам.** При выполнении этих операций основная программа блокируется в ожидании обслуживания запроса к менеджеру памяти. Поэтому одной из задач является минимизация времени блокировки мутатора.
2. **Доля процессорного времени, затрачиваемая на сбор мусора.** Если сборщик мусора работает конкурентно с основной программой, то он отнимает процессорное время, выделенное программе операционной системой. Задачей сборщика является избежание ситуаций, при которых частая сборка мусора приводит к перегрузкам, замедляющим основную программу.
3. **Накладные расходы памяти на хранение данных об обрабатываемых объектах.** Их минимизация повышает эффективность выделения памяти, однако, возможно, в ущерб трудоёмкости алгоритма сборки мусора.
4. **Использование конкурентной сборки мусора.** Для снижения среднего времени паузы мутатора сборщик мусора может выполнять её конкурентно с потоками основной программы.
5. **Использование параллельной сборки мусора.** Задействование сборщиком сразу нескольких потоков программы может ускорить его работу.
6. **Использование алгоритма поколений.** Следование гипотезе поколений (см. п. 1.3.2.1.) позволяет снизить время одного цикла сборки мусора без потери её эффективности. Эффективность цикла сборки мусора можно оценить объёмом памяти, освобождённым за единицу времени.

Целью разрабатываемого метода распределения памяти является минимизация нагрузки на мутатор за счёт использование параллельной и конкурентной сборки мусора.

Для организации объектов, обрабатываемых менеджером памяти, предлагаются использовать модель поколений (см. п. 1.3.2.1.), предполагающей наличие молодого, промежуточного и старого поколения. Промежуточное поколение необходимо для того, чтобы замедлить продвижение объектов по поколениям. Если объект пережил один сбор мусора, то это не значит, что он будет жить дольше других объектов. А вот если он пережил два сбора мусора, то есть его возраст больше промежутка времени между сборами мусора, то вероятность его дальнейшего выживания оценивается выше.

Для реализации модели поколений куча разделяется на три пространства (поколения). Каждый новый объект помещается в первое поколение. Алгоритм сбора мусора выполняется только над объектами определённых поколений, и если объект не уничтожается во время обработки своего поколения, он будет перемещён в следующее поколение, где его будут анализировать реже. Если тот же объект не уничтожается после ещё одного цикла сборки мусора, он будет перемещен в последнее поколение, где его будут анализировать реже всего. Первоначально сборка мусора выполняется только в первом поколении. Такой подход позволяет сборщику мусора избежать сканирования всей кучи на каждом цикле сборки за счёт их разбиения на более короткие. В таком случае сборщик будет выполнять свою работу инкрементально, с каждым новым циклом увеличивая количество обрабатываемых объектов и охватывая более старые объекты.

Также в описанную модель поколений предлагается добавить поколение «больших» объектов, являющееся аналогом кучи больших объектов в C# (см. п. 1.3.4.1.). Это позволит реже всего анализировать «большие» объекты, так как чаще всего они представляются агрегированными типами данных и их анализ является намного более трудоёмкой процедурой, чем анализ объектов, представленных простым типом данных. Минимальный размер объекта, подлежащего попаданию в поколение больших объектов, предлагается выбрать равным 85000 байт, так как это значение уже было выведено для языка C#.

Для контроля фрагментации кучи предлагается разделить пространство каждого поколения на **арены памяти** фиксированного размера, определяемого особенностями языка реализации. Предполагается, что именно арене будет яв-

ляться единицей перераспределения памяти между поколениями. Это означает, что после циклов сборки мусора объекты будут перемещаться между аренами не по одному, а множеством. Такой подход позволит в некоторой степени сохранить эффективное расположение ссылок (см. п. 1.2.2.).

2.2. Проектирование алгоритмов распределения памяти

2.2.1. Представление дескриптора объекта

Для управления объектами программы разрабатываемый менеджер памяти должен хранить следующие данные о них:

- адрес для обеспечения доступа к объекту;
- метаданные о типе объекта для анализа его ссылок на другие объекты программы;
- аrena памяти, в которой был выделен объект, для анализа загруженности арен;
- число ссылок на объект;
- данные о том, были ли финализированы все ссылки на объект;
- данные о том, участвует ли объект в цикле ссылок;
- идентификатор последнего цикла сборки мусора, в котором участвовал объект, для определения того, был ли размечен объект на текущем цикле сборки.

Также для избежания гонок данных при работе с дескриптором объекта он должен содержать примитив блокировки для осуществления атомарных операций над ним.

Для определения, является ли объект мусором, предлагается проверять следующие условия: все ссылки на объект были финализированы и объект участвует в цикле ссылок и количество ссылок на него не превышает 1. Первое условие не является достаточным для определения недостижимости объекта в основной программе, поскольку он может участвовать в цикле ссылок между

мусорными объектами. В таком случае предлагается использование дизъюнкции первого и второго условий для определения достижимости объекта. Первое условие соответствует состоянию счётчика ссылок на объект, а второе предполагает осуществление трассирующей сборки мусора для определения циклических ссылок между объектами программы. Стоит отметить, что ограничение по количеству ссылок на объект во втором условии не даёт возможность идентифицировать мусорными те объекты, которые участвуют одновременно в нескольких циклах ссылок. Это означает, что сборщик мусора не сможет за один цикл сборки освободить все объекты, участвующие в пересекающихся циклах ссылок, опираясь на предположение о том, что подобные случаи будут встречаться в программах относительно редко. К тому же, для корректной идентификации объектов, участвующих в нескольких циклах ссылок, пришлось бы использовать алгоритмы обнаружения компонент сильной связности в орграфах, которые образуют объекты с помощью отношения ссылки. Примерами таких алгоритмов могут послужить алгоритмы Тарьяна и Косарайю [56]. Их использование может существенно увеличить трудоёмкость алгоритмов сборки мусора.

2.2.2. Алгоритм выделения памяти

На рисунке 2.1 представлен алгоритм выделения памяти.

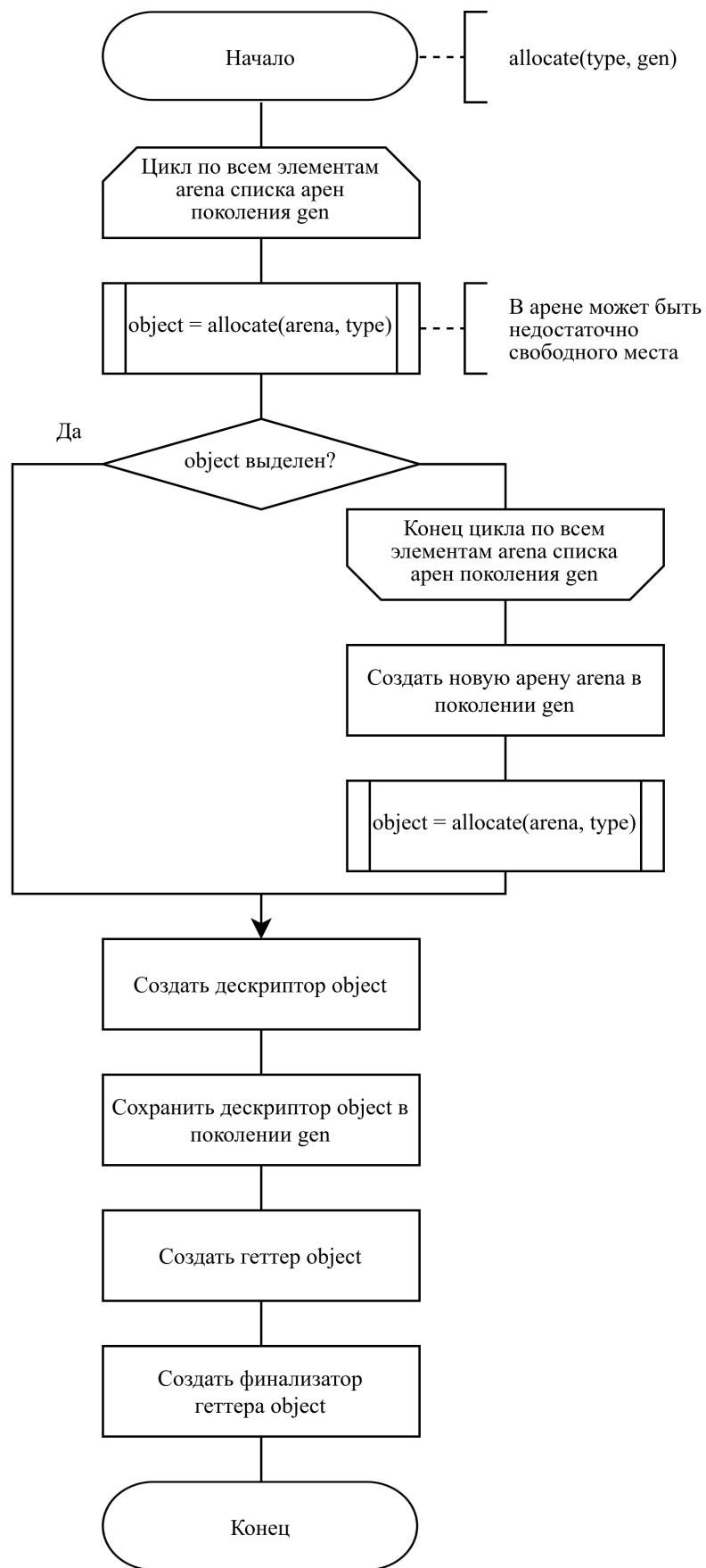


Рис. 2.1 – Алгоритм выделения памяти

Предполагается, что геттер объекта будет использоваться мутатором как

ссылка на него, поскольку геттер предоставляет единственный вариант получения доступа к объекту из основной программы. Геттер является барьером между основной программой и менеджером памяти, предотвращающим гонки данных между ними.

2.2.3. Алгоритм сбора циклических ссылок

Для сбора циклических ссылок между объектами программы предлагается использование трассирующей сборки мусора.

Каждая следующая сборка мусора выполняется после того, как был выделен дополнительный объём памяти, пропорциональный тому, который был зафиксирован при предыдущем запуске сборки мусора. Данная пропорция должна конфигурироваться с помощью переменной среды выполнения программы, чтобы пользователь мог настраивать частоту сборки мусора. Например, если переменная среды равна 100 и программа использует 4 Мб памяти, следующий цикл сборки мусора запустится только тогда, когда объём используемой памяти достигнет 8 Мб. Такая настройка позволяет сохранить линейную зависимость накладных расходов на сборку мусора от накладных расходов на выделение памяти. Переменная среды просто изменяет линейную константу.

Также сам пользователь должен иметь возможность явно вызывать сборку мусора. Это может понадобиться для того, чтобы предотвратить сбор мусора в критический период, например, при большой нагрузке на сервер. В таком случае пользователь может определить момент, когда сервер не обрабатывает запросы, и вызвать сборку мусора.

Сбор мусора предлагается осуществлять по алгоритму **mark-sweep** (см. п. 1.2.3.), который не предполагает уплотнения кучи. Такой подход позволяет избежать накладных расходов на копирование объектов, жертвуя их непрерывным последовательным распределением.

Сбор циклических ссылок будет выполняться в три фазы: разметка, очистка и перераспределение объектов между поколениями.

2.2.3.1. Разметка

На рисунке 2.2 представлен алгоритм разметки объектов нового поколения. Для ускорения разметки предлагается выполнять её в нескольких потоках программы. Параллелизм может быть ограничен путём использования пула по-

токов. Стоит заметить, что перед началом разметки в счётчике ссылок каждого объекта устанавливается значение -1, так как каждый объект будет обработан не менее одного раза, независимо от того, есть на него ссылки или нет.

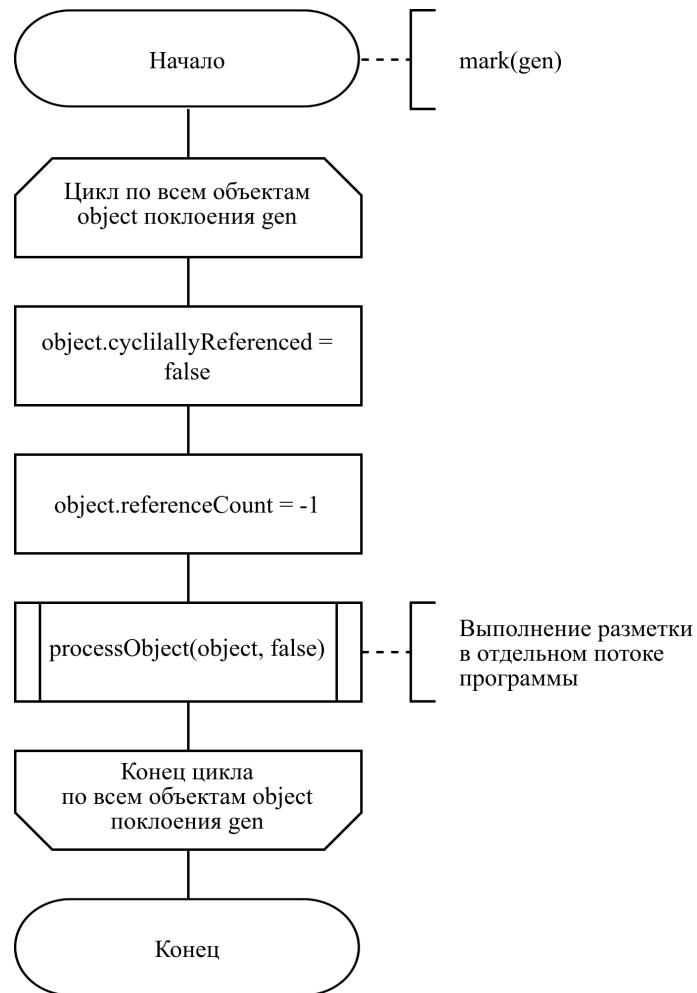


Рис. 2.2 – Алгоритм разметки объектов поколения

На рисунках 2.3 и 2.4 представлен алгоритм обработки одного объекта, включающий в себя его разметку и анализ ссылок на другие объекты программы, если данный объект ранее не был проанализирован. Также анализ объекта может быть пропущен, если были финализированы все ссылки на него. Выполнение этого условия гарантирует, что рассматриваемый объект является мусором.

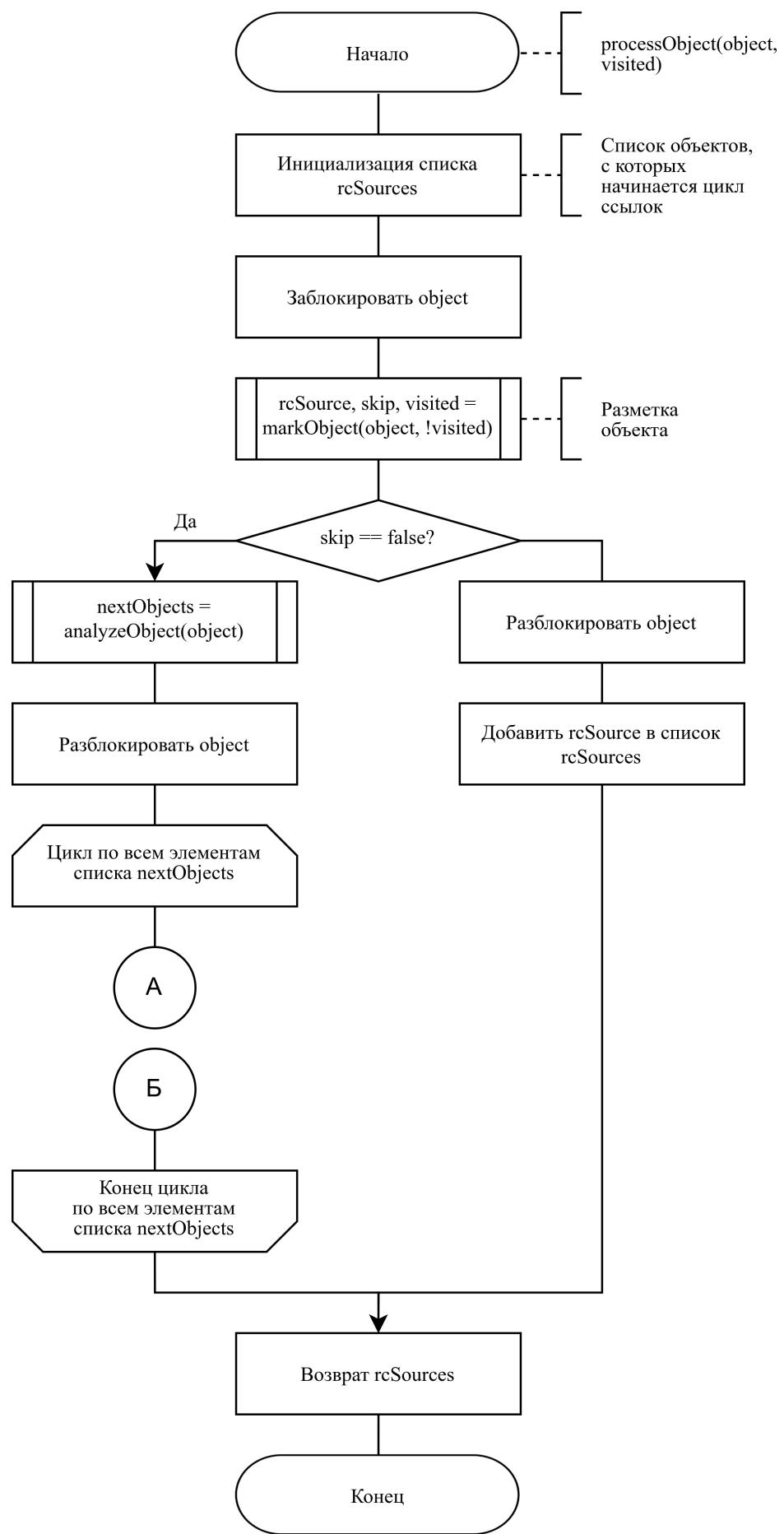


Рис. 2.3 – Алгоритм обработки одного объекта

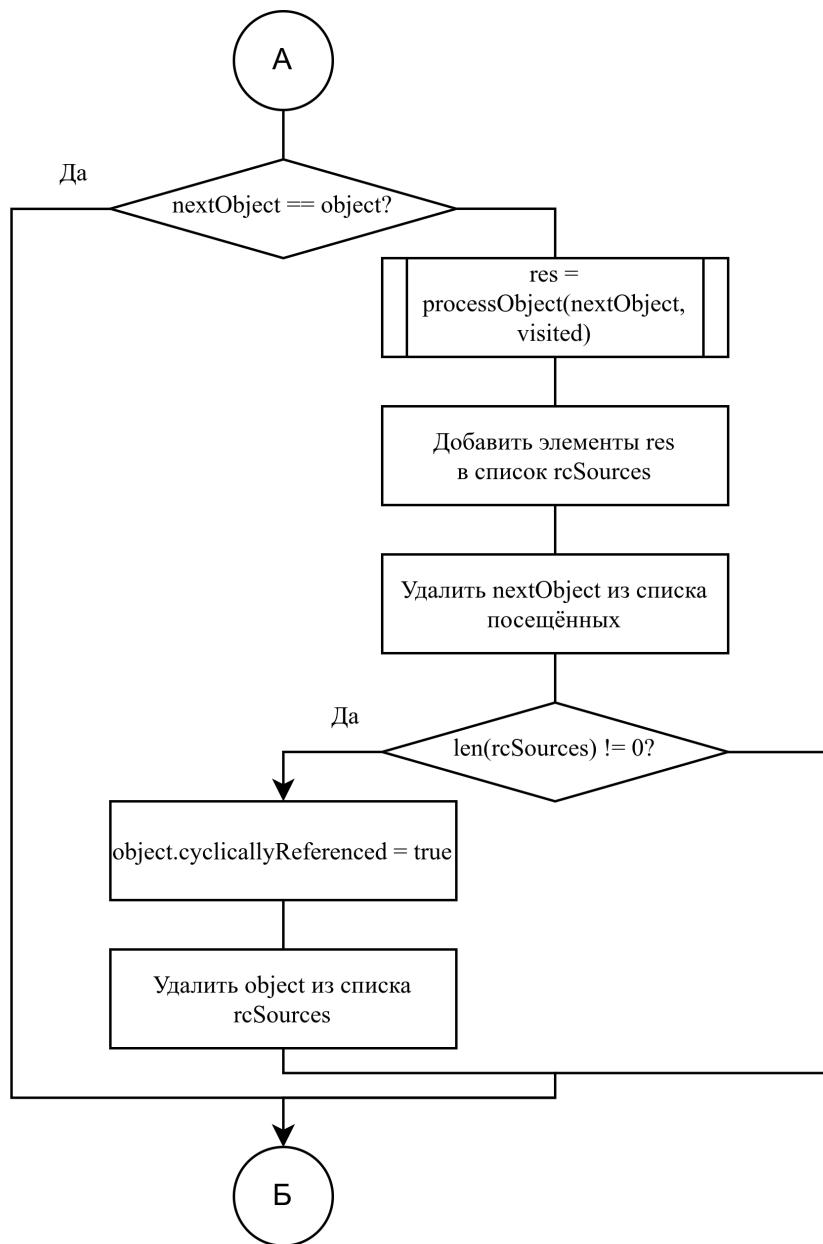


Рис. 2.4 – Алгоритм обработки одной ссылки объекта

На рисунке 2.5 представлен алгоритм разметки объекта, цель которого состоит в том, чтобы заполнить или актуализировать его дескриптор, а также сделать вывод о том, следует ли анализировать его ссылки на другие объекты.

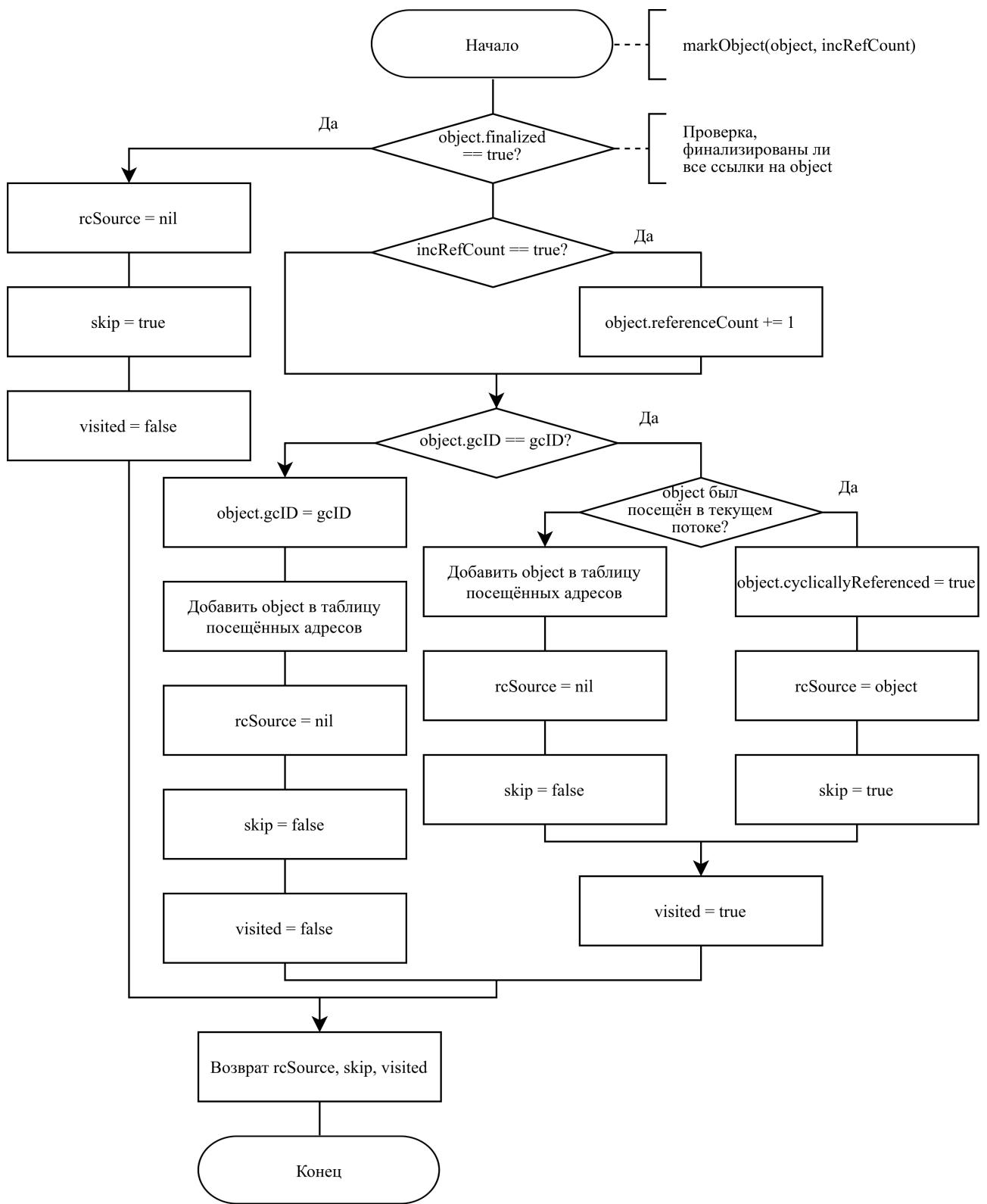


Рис. 2.5 – Алгоритм разметки объекта

На рисунке 2.6 представлен алгоритм анализа ссылок объекта на другие объекты программы. Ссылка на объект анализируется только в том случае, если она принадлежит множеству объектов, анализируемых на данном цикле сборки

мусора. С каждым последующим циклом сбора данное множество расширяется за счёт совокупного анализа всё большего количества поколений, обеспечивая инкрементальное выполнение сборки мусора.

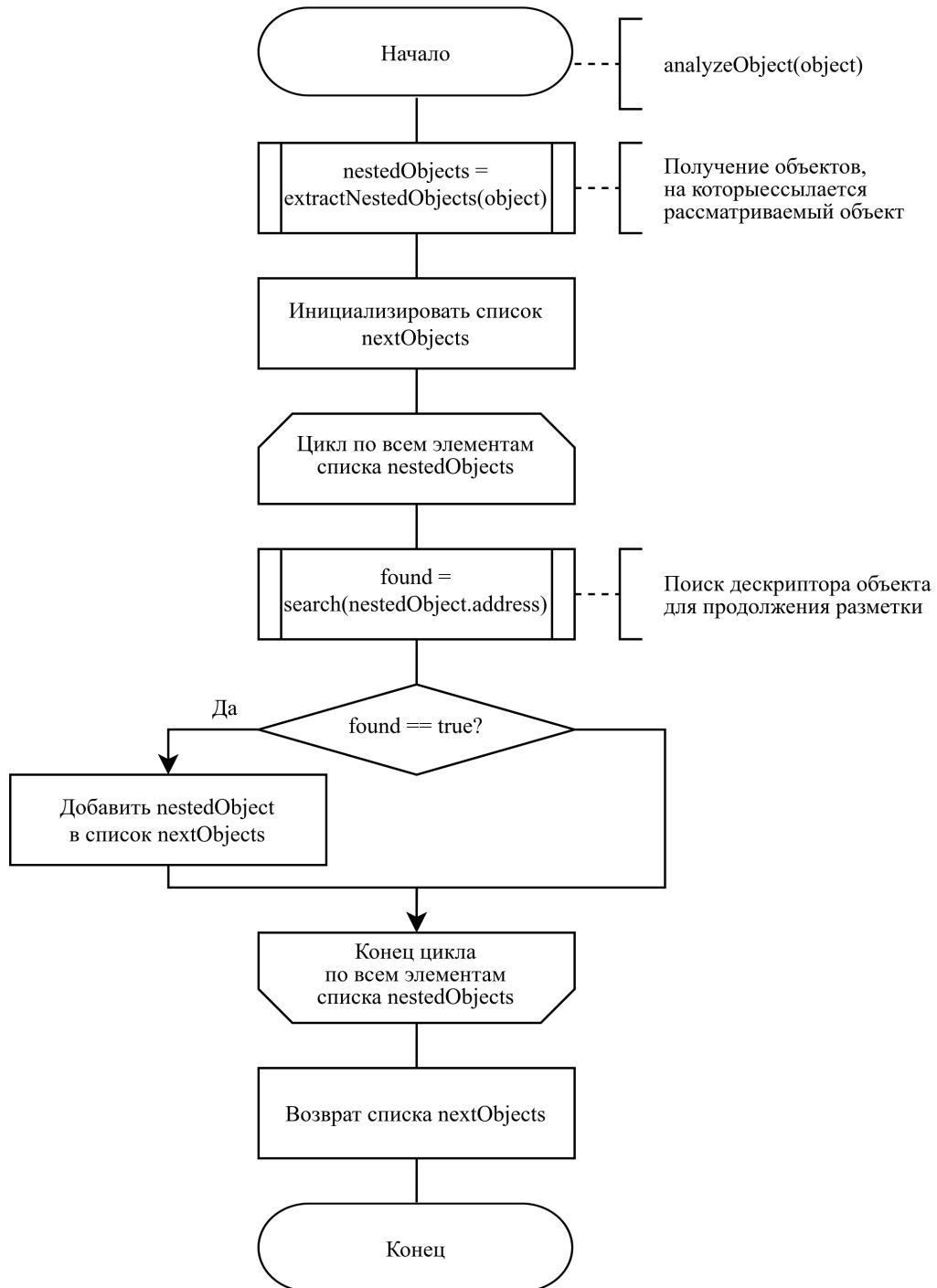


Рис. 2.6 – Алгоритм анализа ссылок объекта

2.2.3.2. Очистка

На рисунке 2.7 представлен алгоритм очистки поколения от мусорных объектов. Для дальнейшего принятия решения о перераспределении объектов

между поколениями, рассматриваемый алгоритм подсчитывает размер поколения до и после очистки. В данном контексте размер измеряется в аренах.

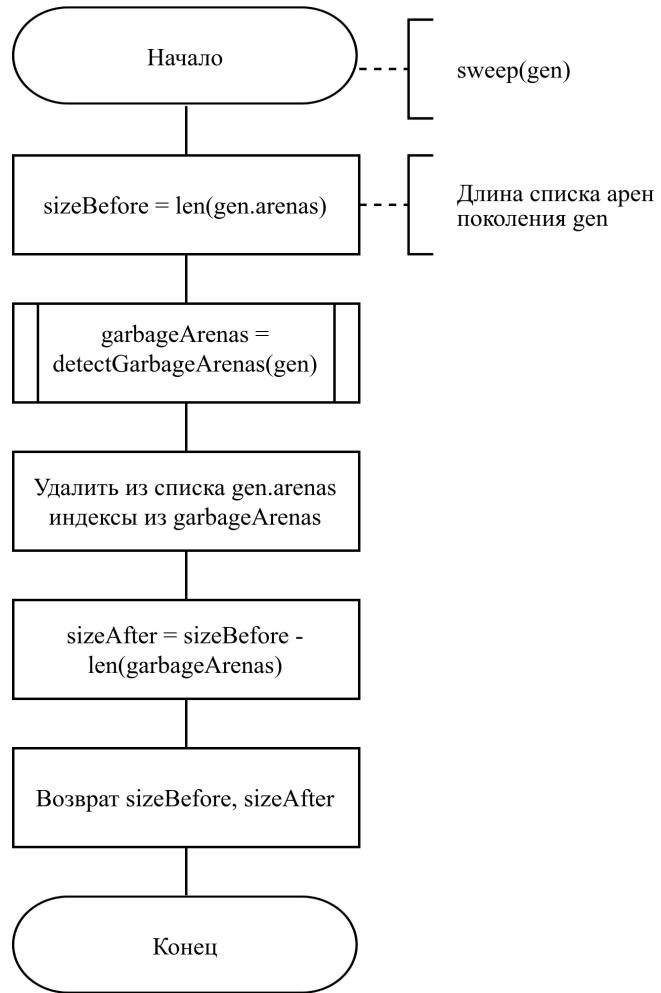


Рис. 2.7 – Алгоритм очистки поколения от мусорных объектов

На рисунке 2.8 представлен алгоритм обнаружения мусорных арен для их последующего освобождения. Арен считается мусорной, если все объекты, которые в ней были выделены на момент анализа, являются мусором. Такое определение мусорной арены может приводить к ситуациям, когда арена не может быть освобождена из-за наличия в ней живых объектов, суммарный размер которых много меньше размера арены. Данная проблема может решаться по-разному в зависимости от реализации, но в общем случае можно предложить выбор оптимального размера арены памяти, при котором частота возникновения описанных ситуаций пренебрежимо мала.

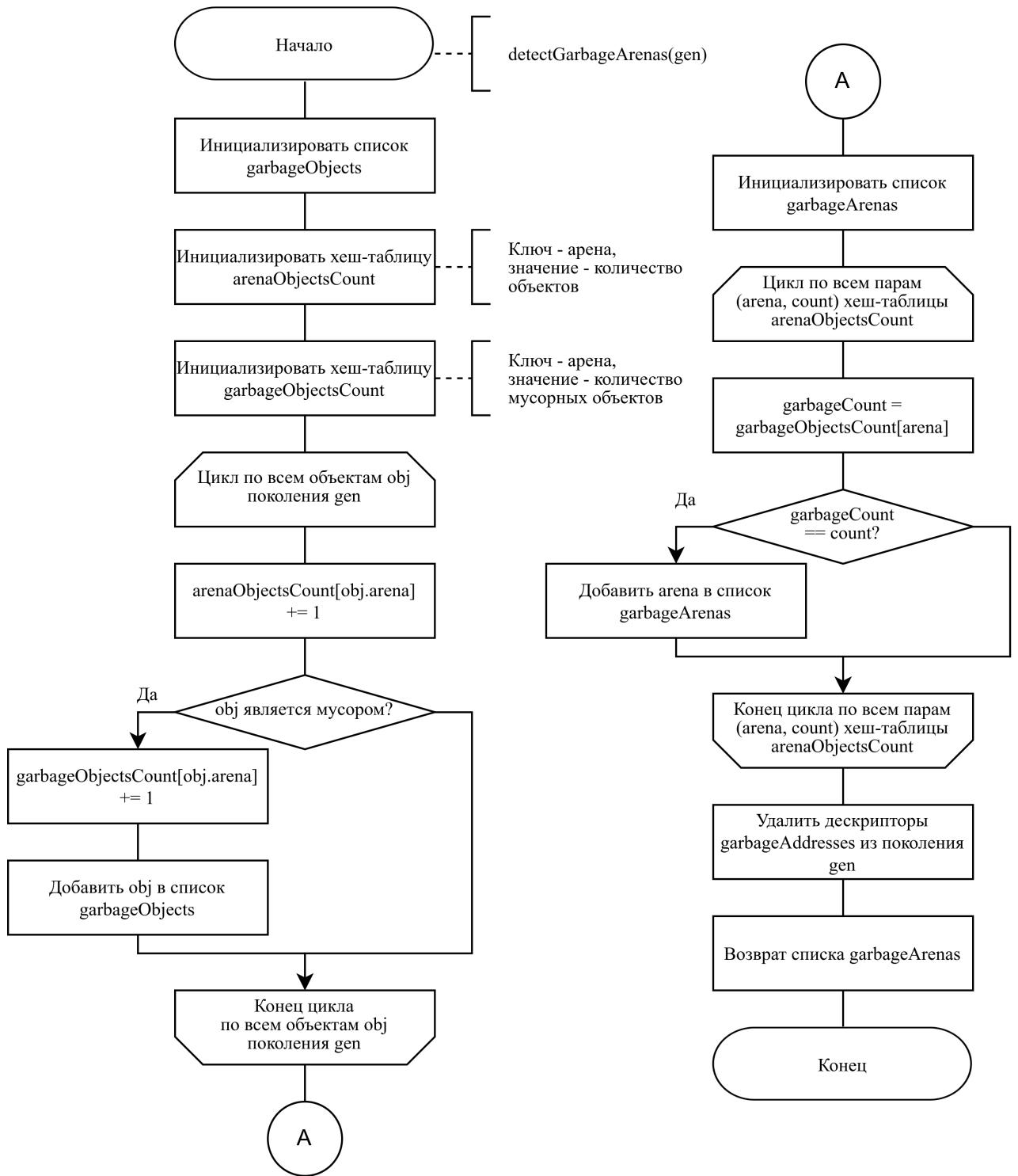


Рис. 2.8 – Алгоритм обнаружения мусорных арен

2.2.3.3. Перераспределение объектов между поколениями

На рисунке 2.9 представлен алгоритм перераспределения объектов между поколениями. Порог перемещения поколения используется для ограничения скорости продвижения объектов по поколениям.

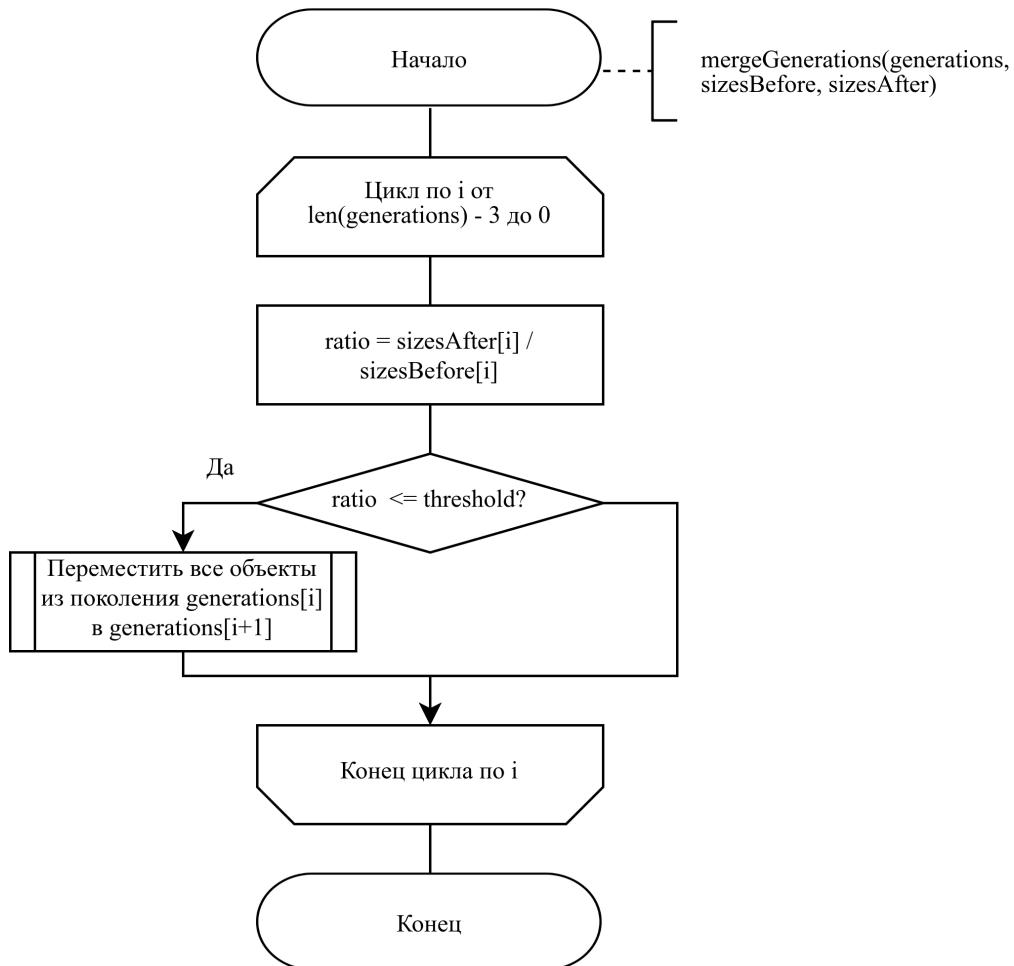


Рис. 2.9 – Алгоритм перераспределения объектов между поколениями

2.3. Выбор структуры данных для хранения дескрипторов объектов

Алгоритмы выделения памяти и сборки мусора предполагают наличие контейнера дескрипторов объектов. Основываясь на особенностях разработанных алгоритмов, можно сделать вывод о том, что операция поиска элемента будет выполняться гораздо чаще, чем операции вставки и удаления. Следовательно, главным критерием выбора структуры данных для хранения дескрипторов объектов будет являться времененная трудоёмкость выполнения операции поиска.

Среди различных типов структур данных самыми эффективными с точки зрения поиска элементов являются

- хеш-таблицы (сложность поиска $O(1)$ в лучшем случае, $O(N)$ в худшем);
- деревья ($O(\log N)$ в лучшем и худшем случае, если рассматривать, например, красно-чёрные деревья);

- отсортированные массивы ($O(\log N)$ в лучшем и худшем случае при использовании алгоритма бинарного поиска).

Все перечисленные структуры данных требуют затрат памяти, линейно зависящих от количества элементов.

Отсортированный массив не подходит для хранения дескрипторов объектов, поскольку хранит данные в памяти последовательно, и с увеличением количества объектов затраты времени и памяти на перевыделение массива будут расти, приводя к задержкам мутатора при выделении новых объектов.

Между хеш-таблицами и деревьями рекомендуется выбрать хеш-таблицу по причине меньшей временной сложности поиска элемента в лучшем случае. Для случаев, в которых сложность поиска элемента в хеш-таблице составляет не $O(1)$, а $O(1) - O(N)$, зависит от выбора хеш-функции для ключей.

Стоит отметить, что оптимальность выбора структуры данных для хранения дескрипторов объектов может зависеть от особенностей языка реализации.

2.4. Выбор подхода к реализации метода

Разработанные алгоритмы распределения памяти можно реализовать либо в виде модификации компилятора или интерпретатора языка, либо в виде подключаемой библиотеки.

Реализация в виде модификации компилятора или интерпретатора языка предоставляет больше возможностей для использования собственных барьеров чтения и записи объектов, позволяет безопасно выполнять копирование и перемещение отдельно взятых объектов программы, а также напрямую влиять на работу основного сборщика мусора, если его наличие предусмотрено языком реализации метода.

Реализация в виде подключаемой библиотеки позволяет не зависеть от метода распределения памяти, используемого языком программирования. Такой подход предполагает добавление новой библиотеки в пользовательский проект вместо модификации исходного языка, что является более гибким решением, так как появляется возможность выбирать методы распределения памяти для каждого модуля приложения и сравнивать их при использовании в одном отдельно взятом модуле.

Оба подхода к реализации предполагают добавление новой зависимости в пользовательских проектах. Описанные ранее алгоритмы работы с памятью

спроектированы таким образом, чтобы их выполнение можно было совмещать с работой сборщика мусора, встроенного в язык программирования реализации. При этом не требуется напрямую влиять на его работу, а также иметь возможность копировать и перемещать объекты, так как сборщики мусора, работающие по алгоритму mark-sweep являются неперемещающими. Также стоит отметить, что модификация компилятора или интерпретатора языка блокирует возможность сравнивать в рамках одного языка разработанный менеджер памяти с существующим. Следовательно, наиболее предпочтительным является вариант реализации метода в виде подключаемой библиотеки.

Выводы из конструкторского раздела

В данном разделе были разработаны алгоритмы основные этапов метода автоматического управления памятью с гарантированным временем выполнения на основе подсчёта ссылок. Был описан подход к реализации метода.

Исходя из количества данных, которые хранит в себе дескриптор объекта, можно выдвинуть предположение о том, что метод будет иметь наибольшую эффективность при его использовании для аллокации объектов программы, которые не содержат циклических ссылок и занимают объём памяти, намного превышающий размер дескриптора объекта. Тогда накладные расходы на хранение дескрипторов объектов и сборку циклических ссылок могут оказаться пренебрежимо малы по сравнению с ресурсными затратами основной программы.

3 Технологический раздел

3.1. Выбор языка программирования

Для реализации разработанного метода должен быть выбран язык, представляющий возможность создавать собственные менеджеры памяти. При этом он должен иметь встроенный сборщик мусора, что позволит провести сравнительный анализ разработанного метода с существующей реализацией в рамках одного языка при использовании разных менеджеров памяти. По этим причинам в качестве языка реализации был выбран Golang [4].

3.2. Аренды памяти

В версии Golang 1.20 появилось экспериментальное решение для управления памятью, которое позволяет совместить безопасное выделение динамической памяти и уменьшение влияния среды выполнения языка, включающей интегрированный менеджер памяти, на производительность приложения. [57]

Пакет arena предоставляет альтернативную возможность выделения памяти в программах на языке Golang. Освобождать такую память необходимо вручную, причем всю одновременно. Цель этой функциональности — повышение эффективности программ: ручное освобождение памяти откладывает следующий цикл сборки мусора, что приводит к снижению частоты циклов и накладных расходов на сборку мусора. [57]

Стоит отметить, что если размер объекта превышает 25% размера пустой аренды, то он будет выделен за пределами аренды и его жизненный цикл будет контролироваться встроенным сборщиком мусора. [58] Поэтому с точки зрения разрабатываемого метода такие объекты считаются **неконтролируемыми**, при этом их ссылки могут быть проанализированы и факт их финализации сборщиком мусора может быть определён.

3.3. Компоненты программного обеспечения

Разработанный метод распределения памяти был реализован в виде подключаемой библиотеки, состоящей из следующих компонентов.

1. **Аренда (arena)**, предназначенная для выделения и хранения объектов программы. Стоит отметить, что аренды из пакета arena языка Golang имеют неявные ограничения по размеру и количеству выделяемых объектов: если размер объекта превышает 25% размера аренды или размер свободной памяти в ней, то он выделяется за пределами аренды, что может свести к нулю выигрыш во времени выделения памяти по сравнению со стандартными средствами языка Golang. [58] По этой причине был реализован обёрточный класс `limited_arena`, реализующий описанное ограничение явно.
2. **Дескриптор объекта (metadata)**, предназначенный для сохранения данных о выделяемых объектах (см. п. 2.2.1.).

3. **Поколение** (generation), предназначенное для хранения дескрипторов выделенных объектов и поддерживающее операции сборки мусора и перемещения объектов между поколениями. Поколение не хранит отдельные объекты напрямую, а использует для этого арены памяти фиксированного размера.
4. **Память** (memory), являющаяся абстракцией, хранящей все поколения и предназначеннной для обработки внешних запросов на выделение памяти в куче.
5. **Гипервизор** (hypervisor), отвечающий за выполнение сборки мусора и перераспределение объектов между поколениями по итогам сбора. Именно гипервизор определяет момент, когда необходимо провести сборку мусора.
6. **Разметчик** (mark worker), предназначенный для разметки объектов, выделенных менеджером памяти. Он работает в одном потоке программы, поэтому поколение для проведения параллельной разметки запускает несколько разметчиков в отдельных потоках программы и ожидает завершения их работы.

Структуры, описывающие основные компоненты реализованной библиотеки, представлены на рисунке 3.1 в виде диаграммы классов.

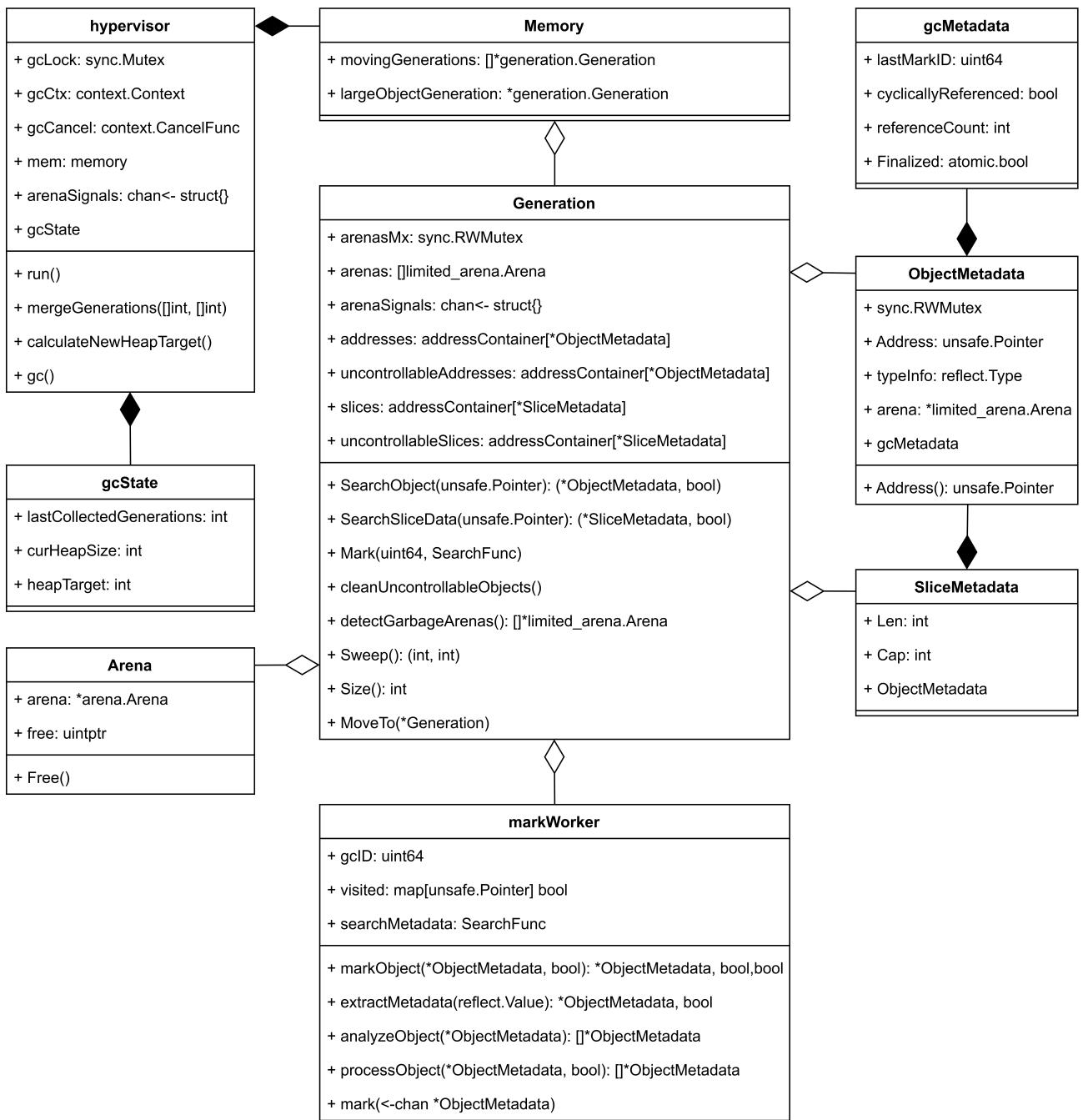


Рис. 3.1 – Диаграмма классов разработанной библиотеки

3.3.1. Выделение памяти в куче

Функции выделения памяти в куче, определённые для объекта `memory`, представлены на листинге 3.1. Все новые объекты выделяются в молодом поколении. Если размер объекта превышает 85000 байт, то он выделяется в специальном поколении больших объектов, чтобы реже всего подвергаться анализу.

Листинг 3.1 – Функции выделения памяти

```

1 const objectSizeThreshold = 85_000
2
3 func allocateObject [T any] (mem memory) (m *generation.ObjectMetadata) {
4     size := unsafe.Sizeof(*new(T)) // no allocation
5     if size > objectSizeThreshold {
6         return generation.AllocateObject [T] (mem.largeObjectGeneration)
7     }
8     return generation.AllocateObject [T] (mem.movingGenerations [0])
9 }
10
11 func allocateSlice [T any] (
12     mem memory, len, cap int,
13 ) *generation.SliceMetadata {
14     size := unsafe.Sizeof(*new(T)) * uintptr(cap) // no allocation
15     if size > objectSizeThreshold {
16         return generation.AllocateSlice [T] (
17             mem.largeObjectGeneration, len, cap,
18         )
19     } else {
20         return generation.AllocateSlice [T] (
21             mem.movingGenerations [0], len, cap,
22         )
23     }
24 }
```

3.3.2. Выделение памяти в поколении

Функции выделения памяти в поколении объектов представлены на листинге 3.2. Аллокация обычных объектов и срезов выполняется по одному алгоритму, однако требует выполнения различных операций по формированию возвращаемых значений, поэтому реализована в отдельных функциях.

Листинг 3.2 – Функции выделения памяти в поколении объектов

```

1 type holder [T any] interface {
2     *T | []T
3 }
4
5 type allocateFunc [H holder [T], T any] func (*limited_arena.Arena) (H, bool)
6
7 type allocatedObject [T any, H holder [T]] struct {
8     container H
9     controllable bool
10    arena *limited_arena.Arena
11 }
12
13 func allocate [T any, H holder [T]] (
```

```

14     gen *Generation, allocateObject allocateFunc[H, T],
15 ) allocatedObject[T, H] {
16     gen.arenasMx.Lock()
17     var object allocatedObject[T, H]
18     for _, arena := range gen.arenas {
19         object.container, object.controllable = allocateObject(&arena)
20         if object.container != nil {
21             object.arena = &arena
22             break
23         }
24     }
25     if object.container == nil {
26         arena := limited_arena.NewLimitedArena()
27         gen.arenaSignals <- struct{}{}
28         object.container, object.controllable = allocateObject(&arena)
29         object.arena = &arena
30         gen.arenas = append(gen.arenas, arena)
31     }
32     gen.arenasMx.Unlock()
33
34     return object
35 }
36
37 func AllocateObject[T any](gen *Generation) *ObjectMetadata {
38     object := allocate[T](gen, limited_arena.New[T])
39     addr := unsafe.Pointer(object.container)
40
41     metadata := ObjectMetadata{
42         Address: addr,
43         typeInfo: reflect.TypeOf(*object.container),
44         arena:    object.arena,
45     }
46
47     if !object.controllable {
48         gen.uncontrollableAddresses.Add(addr, &metadata)
49     } else {
50         gen.addresses.Add(addr, &metadata)
51     }
52
53     return &metadata
54 }
55
56 func AllocateSlice[T any](gen *Generation, len, cap int) *SliceMetadata {
57     object := allocate[T](
58         gen, func(arena *limited_arena.Arena) ([]T, bool) {
59             return limited_arena.MakeSlice[T](arena, len, cap)
60         },
61     )

```

```

62     addr := unsafe.Pointer(&object.container[:1][0])
63
64     metadata := SliceMetadata{
65         ObjectMetadata: ObjectMetadata{
66             Address:   addr,
67             typeInfo: reflect.TypeOf(object.container),
68             arena:     object.arena,
69         },
70         Len: len,
71         Cap: cap,
72     }
73
74     if !object.controllable {
75         gen.uncontrollableSlices.Add(addr, &metadata)
76     } else {
77         gen.slices.Add(addr, &metadata)
78     }
79
80     return &metadata
81 }

```

3.3.3. Сбор мусора в поколении

Сбор мусора проводится для каждого поколения отдельно и состоит из разметки, очистки и перераспределения данных между поколениями. Первые два этапа реализует поколение, а третий — гипервизор, принимающий решение о том, стоит ли переносить объекты из одного поколения в другое после очередного цикла сбора мусора. Также именно гипервизор устанавливает целевой размер кучи, по достижении которого будет осуществлён следующий цикл сбора мусора.

3.3.3.1. Разметка

Реализация алгоритма разметки объектов поколения представлена на листинге 3.3. Для выполнения параллельной разметки запускается несколько разметчиков в отдельных потоках программы. Ограничение количества запущенных разметчиков по умолчанию равно половине от количества процессорных ядер, доступных программе.

Листинг 3.3 – Реализация алгоритма разметки объектов поколения

```

1 var gcMarkConcurrency = (runtime.NumCPU() + 1) / 2
2
3 type SearchFunc func(addr unsafe.Pointer) (

```

```

4     metadata *ObjectMetadata, exist bool,
5 )
6
7 func (gen *Generation) Mark(gcID uint64, searchMetadata SearchFunc) {
8     search := func(addr unsafe.Pointer) (
9         metadata *ObjectMetadata, exist bool,
10    ) {
11        metadata, exist = gen.SearchObject(addr)
12        if !exist {
13            metadata, exist = searchMetadata(addr)
14        }
15        return
16    }
17
18 objects := make(chan *ObjectMetadata)
19 wg := sync.WaitGroup{}
20 for range gcMarkConcurrency {
21     mw := markWorker{
22         gcID:           gcID,
23         visited:       make(map[unsafe.Pointer] bool),
24         searchMetadata: search,
25     }
26     wg.Add(1)
27     go func() {
28         mw.mark(objects)
29         wg.Done()
30     }()
31 }
32
33 gen.addresses.Map(func(metadata *ObjectMetadata) {
34     metadata.cyclicallyReferenced = false
35     metadata.referenceCount--
36     objects <- metadata
37 })
38
39 gen.slices.Map(func(metadata *SliceMetadata) {
40     metadata.cyclicallyReferenced = false
41     metadata.referenceCount--
42     objects <- &metadata.ObjectMetadata
43 })
44
45 close(objects)
46 wg.Wait()
47 }

```

Реализация алгоритма обработки одного объекта представлена на листинге 3.4. При рекурсивном обходе графа объектов разметчик обнаруживает цик-

лические ссылки, исключая ссылки объекта на самого себя.

Листинг 3.4 – Реализация алгоритма обработки одного объекта

```
1 func (mw markWorker) processObject(
2     object *ObjectMetadata, visited bool,
3 ) (rcSrcs []*ObjectMetadata) {
4     object.Lock()
5     rcSrc, skip, visited := mw.markObject(object, !visited)
6     if !skip {
7         nextObjects := mw.analyzeObject(object)
8         object.Unlock()
9         for _, nextObject := range nextObjects {
10             if nextObject == object {
11                 continue
12             }
13             rcSrcs = append(rcSrcs,
14                 mw.processObject(nextObject, visited)...
15             )
16             delete(mw.visited, nextObject.Address)
17             if len(rcSrcs) != 0 {
18                 object.cyclicallyReferenced = true
19                 rcSrcs = slices.DeleteFunc(
20                     rcSrcs, func(metadata *ObjectMetadata) bool {
21                         return metadata == object
22                     }
23                 )
24             }
25         }
26     } else {
27         object.Unlock()
28         rcSrcs = append(rcSrcs, rcSrc)
29     }
30
31     return
32 }
```

Реализация алгоритма разметки одного объекта представлена на листинге 3.5. Разметка объекта включает в себя подсчёт ссылок на объект, определение факта участия рассматриваемого объекта в цикле ссылок и принятие решения о дальнейшей обработке его ссылок на другие объекты программы.

Листинг 3.5 – Реализация алгоритма разметки объекта

```
1 func (mw markWorker) markObject(object *ObjectMetadata, incRefCount bool) (
2     rcSrc *ObjectMetadata, skip, visited bool,
3 ) {
4     if object.Finalized.Load() {
5         return nil, true, false
6     }
7     if !object.cyclicallyReferenced && !visited {
8         object.cyclicallyReferenced = true
9         rcSrc = object
10        if incRefCount {
11            object.refCount++
12        }
13    }
14    if object.refCount == 0 {
15        object.refCount = 1
16        rcSrc = object
17    }
18    if !skip {
19        rcSrcs = append(rcSrcs, rcSrc)
20    }
21    if !visited {
22        mw.visited[object.Address] = true
23    }
24    return nil, false, false
25 }
```

```

6         }
7
8     if incRefCount {
9         object.referenceCount++
10    }
11
12    if object.lastMarkID == mw.gcID {
13        if mw.visited[object.Address] {
14            object.cyclicallyReferenced = true
15            return object, true, true
16        } else {
17            mw.visited[object.Address] = true
18            return nil, false, true
19        }
20    } else {
21        object.lastMarkID = mw.gcID
22        mw.visited[object.Address] = true
23        return nil, false, false
24    }
25 }

```

Реализация алгоритма анализа ссылок объекта представлена на листинге 3.6. Извлечение ссылок объекта производится с помощью библиотеки reflect [59]. После извлечения ссылок на другие объекты производится поиск их дескрипторов в множестве доступных для разметки адресов, которое может изменяться между циклами сборки мусора вместе с количеством поколений, анализируемых в текущем цикле сборки. Чем шире область поиска, тем больше ссылок между объектами можно обнаружить и тем выше вероятность обнаружения цикла ссылок между объектами программы.

Листинг 3.6 – Реализация алгоритма анализа ссылок объекта

```

1 func (mw markWorker) analyzeObject(metadata *ObjectMetadata) (
2     nextObjects []*ObjectMetadata,
3 ) {
4     object := reflect.NewAt(metadata.typeInfo, metadata.Address).Elem()
5     nestedObjects := extractNestedObjects(object)
6     for _, nestedObject := range nestedObjects {
7         nestedMetadata, exist := mw.extractMetadata(nestedObject)
8         if exist {
9             nextObjects = append(nextObjects, nestedMetadata)
10        }
11    }
12
13    return
14 }

```

3.3.3.2. Очистка

Реализация алгоритма очистки поколения от мусорных объектов представлена на листинге 3.7. Очистка производится в одном потоке программы и состоит в удалении не отдельных объектов, а арен, состоящих только из мусорных объектов.

Листинг 3.7 – Реализация алгоритма очистки поколения от мусорных объектов

```
1 func (gen *Generation) Sweep() (int, int) {
2     gen.arenasMx.Lock()
3
4     sizeBefore := len(gen.arenas)
5
6     garbageArenas := gen.detectGarbageArenas()
7     if len(garbageArenas) == 0 {
8         return sizeBefore, sizeBefore
9     }
10
11    for offset, arena := range garbageArenas {
12        index := slices.Index(gen.arenas, *arena)
13        tailIndex := len(gen.arenas) - offset - 1
14        gen.arenas[index] = gen.arenas[tailIndex]
15        gen.arenas[tailIndex].Free()
16        gen.arenas[tailIndex] = limited_arena.Arena{}
17    }
18
19    gen.arenas = gen.arenas[:len(gen.arenas)-len(garbageArenas)]
20
21    gen.arenasMx.Unlock()
22
23    return sizeBefore, sizeBefore - len(garbageArenas)
24 }
```

Реализация алгоритма обнаружения мусорных арен представлена на листинге 3.8. Очистка производится в одном потоке программы и состоит в удалении не отдельных объектов, а арен, состоящих только из мусорных объектов.

Листинг 3.8 – Реализация алгоритма обнаружения мусорных арен

```
1 func (gen *Generation) cleanUncontrollableObjects() {
2     garbageObjects := make([]unsafe.Pointer, 0)
3     gen.uncontrollableAddresses.Map(func(object *ObjectMetadata) {
4         if object.Finalized.Load() {
5             garbageObjects = append(garbageObjects, object.Address)
6         }
7     })
8     gen.uncontrollableAddresses.Delete(garbageObjects)
```

```

9      garbageSlices := make([] unsafe.Pointer, 0)
10     gen.uncontrollableSlices.Map(func(slice *SliceMetadata) {
11         if slice.Finalized.Load() {
12             garbageSlices = append(garbageSlices, slice.Address)
13         }
14     })
15     gen.uncontrollableSlices.Delete(garbageSlices)
16 }
17 }

18

19 func isGarbage(object *ObjectMetadata) bool {
20     return (object.cyclicallyReferenced && object.referenceCount <= 1) ||
21     object.Finalized.Load()
22 }

23

24 func (gen *Generation) detectGarbageArenas() []* limited_arena.Arena {
25     arenaObjectsCount := make(map[* limited_arena.Arena] int, len(gen.arenas))
26     garbageObjectsCount := make(
27         map[* limited_arena.Arena] int, len(gen.arenas),
28     )

29     garbageAddresses := make([] unsafe.Pointer, 0)
30     garbageSlices := make([] unsafe.Pointer, 0)

31     gen.addresses.Map(func(object *ObjectMetadata) {
32         arenaObjectsCount[object.arena]++
33         if isGarbage(object) {
34             garbageObjectsCount[object.arena]++
35             garbageAddresses = append(garbageAddresses, object.Address)
36         }
37     })
38 }

39     gen.slices.Map(func(object *SliceMetadata) {
40         arenaObjectsCount[object.arena]++
41         if isGarbage(&object.ObjectMetadata) {
42             garbageObjectsCount[object.arena]++
43             garbageSlices = append(garbageSlices, object.Address)
44         }
45     })
46 }

47 }

48

49     garbageArenas := make([]* limited_arena.Arena, 0)
50     for arena, count := range arenaObjectsCount {
51         if garbageObjectsCount[arena] == count && count != 0 {
52             garbageArenas = append(garbageArenas, arena)
53         }
54     }
55

56     gen.addresses.Delete(garbageAddresses)

```

```

57     gen.slices.Delete(garbageSlices)
58     gen.cleanUncontrollableObjects()
59
60     return garbageArenas
61 }

```

3.3.3.3. Перераспределение объектов между поколениями

После завершения очистки мусорных арен в каждом поколении гипервизор на основании результатов сбора принимает решение о продвижении объектов по поколениям. Перемещение каждого объекта происходит до тех пор, пока он не окажется в последнем («постоянном») поколении. Также стоит отметить, что перемещению не подвергаются объекты, выделенные в поколении «больших» объектов (см. п. 2.1.).

Листинг 3.9 – Реализация алгоритма перераспределения объектов между поколениями

```

1 const generationMovementThreshold = 0.9
2
3 func (h *hypervisor) mergeGenerations(sizesBefore, sizesAfter []int) {
4     for i := len(sizesBefore) - 3; i >= 0; i-- {
5         if float64(sizesAfter[i])/float64(sizesBefore[i]) <=
6             generationMovementThreshold {
7             h.mem.movingGenerations[i].MoveTo(h.mem.movingGenerations[i+1])
8         }
9         if sizesBefore[i] != sizesAfter[i] {
10             Debugger.arenasFreed.Add(int64(sizesBefore[i] - sizesAfter[i]))
11         }
12     }
13 }

```

3.4. Оценка трудоёмкости алгоритмов

Для определения гарантии времени исполнения необходимо оценить трудоёмкость выполнения мутатором выделения памяти и обращения к выделенным объектам. Трудоёмкость реализаций алгоритмов будет оцениваться только для худшего случая, так как именно его рассмотрение поможет определить гарантию времени выполнения реализованного метода.

3.4.1. Модель вычислений

Для последующего вычисления трудоемкости необходимо ввести модель вычислений.

1. Операции из списка 3.1 имеют трудоемкость 1.

$$+, -, =, + =, - =, ==, !=, <, >, <=, >=, [], ++, --, \&\&, ||, ! \quad (3.1)$$

2. Операции индексации, разыменования указателя, получения адреса, определения размера типа данных, блокировки и разблокировки мьютекса, создания пустого среза и определения его длины, удаления элемента из среза и отправки данных в канал имеют трудоёмкость 1.
3. Трудоёмкость условного оператора вида «if условие then A else B» определяется по формуле 3.2.

$$f_{if} = f_{\text{условие}} + \begin{cases} f_A, & \text{если условие выполняется,} \\ f_B, & \text{иначе.} \end{cases} \quad (3.2)$$

4. Трудоёмкость цикла for, предполагающего выполнение N итераций, определяется по формуле 3.3.

$$f_{for} = f_{\text{инициализация}} + f_{\text{сравнение}} + N(f_{\text{тело}} + f_{\text{инкремент}} + f_{\text{сравнение}}) \quad (3.3)$$

5. Трудоёмкость цикла for range по коллекции из N элементов определяется по формуле 3.4.

$$f_{for} = f_{\text{инициализация}} + N(f_{\text{тела}} + f_{\text{присваивание}}) \quad (3.4)$$

6. Трудоёмкость вызова функции и преобразования типа данных переменной равна 0.
7. Операция добавления элемента в срез имеет трудоёмкость 7.
8. Трудоёмкость получения ссылок объекта, создания экземпляра reflect.Value [59] по указателю на объект, выделения памяти в арене,

создания и освобождения арены, а также создания горутины равна 25.

3.4.2. Трудоёмкость реализации алгоритма выделения памяти

Худшим случаем для оценки трудоёмкости реализации алгоритма выделения памяти является тот, в котором алгоритм начинает выполнятся одновременно с очисткой поколения, при этом все его объекты должны являться мусорными. Так как данные процедуры не могут выполняться параллельно, для выполнения выделения памяти в поколении необходимо дождаться завершения его очистки от мусорных объектов.

В худшем случае трудоёмкость реализации алгоритма выделения памяти без учёта очистки определяется по формуле 3.5.

$$f_{allocateObject}^{\wedge} = 3 + f_{AllocateObject}^{\wedge} = 19 + f_{allocate}^{\wedge} = 19 + 71 = 90 \quad (3.5)$$

В худшем случае трудоёмкость реализации алгоритма очистки поколения от мусорных объектов определяется по формуле 3.6.

$$\begin{aligned} f_{detectGarbageArenas}^{\wedge} &= 13 + 24c + 14a + o + 12 * u = 13 + 25c + 14a + \\ &+ 13u \\ f_{Sweep}^{\wedge} &= 19 + 43a + a^2 + f_{detectGarbageArenas}^{\wedge} = 32 + 57a + 25c + 13u + \\ &+ a^2, \end{aligned} \quad (3.6)$$

где a — число арен в поколении, c — число контролируемых объектов, u — число неконтролируемых объектов, $o = c + u$ — общее число объектов в поколении.

Таким образом, в худшем случае трудоёмкость реализации алгоритма выделения памяти определяется по формуле 3.7.

$$f_{allocation}^{\wedge} = f_{allocateObject}^{\wedge} + f_{Sweep}^{\wedge} = 122 + 57a + 25c + 13u + a^2 \quad (3.7)$$

3.4.3. Трудоёмкость реализации алгоритма обращения к объекту

Худшим случаем для оценки трудоёмкости реализации алгоритма обращения к объекту может являться тот, в котором алгоритм начинает выполнятся

одновременно с разметкой объекта. Так как данные процедуры не могут выполняться параллельно, для выполнения обращения к объекту необходимо дождаться завершения его разметки.

В худшем случае трудоёмкость реализации алгоритма обращения к объекту без учёта разметки равна 4.

В худшем случае трудоёмкость реализации алгоритма обработки объекта разметчиком определяется по формуле 3.8.

$$\begin{aligned} f_{markObject}^{\wedge} &= 11 \\ f_{analyzeObject}^{\wedge} &= 55 + 13r + 2or \\ f_{processObject}^{\wedge} &= 9 + f_{markObject}^{\wedge} + f_{analyzeObject}^{\wedge} = 75 + 13r + 2or, \end{aligned} \tag{3.8}$$

где r — число ссылок рассматриваемого объекта на другие объекты, o — число объектов, участвующих в текущем цикле сборки.

3.5. Руководство пользователя

Библиотека, реализующая разработанный метод, может использоваться в программах на языке Golang версии 1.22.0 и выше. Для подключения библиотеки в проект необходимо установить её с помощью команды 3.10.

Листинг 3.10 – Команда установки библиотеки

```
1 go get github.com/Inspirate789/alloc
```

Далее её можно использовать в проекте, не изменяя процесс сборки приложений. Примеры эквивалентных с точки зрения пользователя программ, использующих встроенные средства языка Golang и реализованную библиотеку, представлены на листингах 3.11 и 3.12 соответственно.

Листинг 3.11 – Программа, использующая встроенные средства языка Golang

```
1 package main
2
3 func main() {
4     object := new(int)
5     println(*object) // 0
6     *object = 7
7     println(*object) // 7
8 }
```

Листинг 3.12 – Программа, использующая реализованную библиотеку

```

1 package main
2
3 import "github.com/Inspirate789/alloc"
4
5 func main() {
6     object := alloc.New[int]()
7     println(*object.Get()) // 0
8     *object.Get() = 7
9     println(*object.Get()) // 7
10 }

```

Стоит отметить, что для корректной работы сборщика мусора разработанного менеджера памяти не следует при работе с выделенными объектами игнорировать их геттеры, являющиеся барьерами чтения и записи. Так, программа, представленная на листинге 3.13, аналогична 3.11 и 3.12 с точки зрения пользователя, однако может привести к гонкам данных при разметке объектов.

Листинг 3.13 – Программа, использующая реализованную библиотеку

```

1 package main
2
3 import "github.com/Inspirate789/alloc"
4
5 func main() {
6     object := alloc.New[int]().Get()
7     println(*object)
8     *object = 7
9     println(*object)
10 }

```

Выводы из технологического раздела

В данном разделе было приведено обоснование выбора языка программирования для реализации разработанного метода. Было программное обеспечение, реализующее предложенный метод и были изложены его особенности.

Также была дана теоретическая оценка трудоёмкости выполнения вызовов реализации метода, а именно выделения памяти и обращения к выделенным объектам. В худшем случае трудоёмкость выделения памяти линейно зависит от числа объектов в самом «молодом» поколении и квадратично от числа арен в нём, при этом трудоёмкость обращения к объекту линейно зависит от числа ссылок в объекте и общего числа объектов, анализируемых сборщиком. Благодаря использованию модели поколений, сборщик мусора анализирует не все объекты, а только некоторую их часть. Таким образом удалось устраниТЬ зави-

симость накладных расходов мутатора от общего числа объектов во всей куче. Полученную оценку трудоёмкости предлагается считать гарантией времени выполнения метода.

4 Исследовательский раздел

4.1. Цель исследования

Реализованный метод распределения памяти может иметь в различные сценарии использования и каждом из них оказывать различное влияние на характеристики выполнения программ. Предполагается, что изменение характеристик выполнения программы при применении реализованного метода зависит от класса алгоритма, который она реализует (см. п. 1.4.).

Целью исследования является проведение сравнительного анализа разработанного метода с реализацией, существующей программирования Golang.

4.2. Описание исследования

Для проведения сравнительного анализа необходимо установить зависимость между длиной входа алгоритма и временем его выполнения для различных классов алгоритмов и для двух методов распределения памяти: встроенного и реализованного. Для этого были подготовлены тесты производительности для следующих алгоритмов:

- 1) алгоритм конвейерной обработки данных (класс VC);
- 2) алгоритм сортировки слиянием (класс VL);
- 3) алгоритм нахождения расстояния Левенштейна (класс VQ);
- 4) алгоритм умножения матриц по Винограду (класс VP);
- 5) алгоритм сортировки подсчётом (класс VE).

Для каждого алгоритма были разработаны две реализации: одна использует встроенные средства языка Golang, а другая — реализованный метод.

4.3. Технические характеристики оборудования

Ниже приведены технические характеристики устройства, на котором выполнялось тестирование.

- Операционная система: Debian Linux 12 (bookworm) x86_64, версия ядра 6.1.0-17.

- Объём оперативной памяти: 16 Гб.
- Процессор: Intel i5-9300H 2.4 ГГц.

Тестирование проводилось на ноутбуке, включенном в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также непосредственно системой тестирования.

4.4. Результаты исследования

4.4.1. Алгоритм конвейерной обработки данных

Для проведения замеров был разработан конвейер, состоящий из трёх линий: аллокация буфера фиксированного размера, запись данных в него и их чтение. В таблице 4.1 представлены результаты замеров времени выполнения алгоритма при различном количестве заявок.

Таблица 4.1 – Замеры времени выполнения алгоритма конвейерной обработки данных

<i>Число заявок, шт</i>	<i>Время выполнения при использовании реализованного метода, мс</i>	<i>Время выполнения при использовании встроенных средств языка, мс</i>
10	271	321
50	476	458
100	702	678
500	1811	1720
1000	3389	3248
3000	11673	9290
5000	19897	15272
7500	28661	21387
10000	37349	27465

На рисунке 4.1 представлены данные из таблицы 4.1 в виде графика.

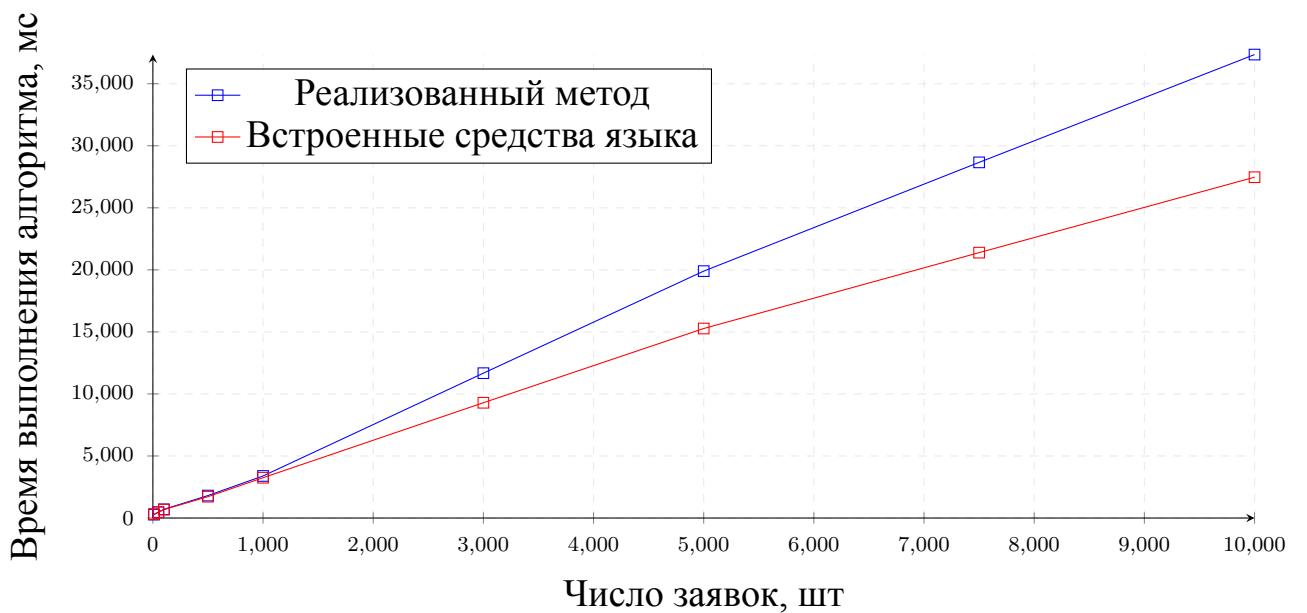


Рис. 4.1 – График зависимости времени выполнения алгоритма конвейерной обработки данных от числа заявок

4.4.2. Алгоритм сортировки слиянием

В таблице 4.2 представлены результаты замеров времени выполнения алгоритма сортировки слиянием при различных размерах входного массива целых чисел.

Таблица 4.2 – Замеры времени выполнения алгоритма сортировки слиянием

<i>Число элементов в массиве, шт</i>	<i>Время выполнения при использовании реализованного метода, мс</i>	<i>Время выполнения при использовании встроенных средств языка, мс</i>
100	91	13
500	694	111
1000	993	241
5000	5138	1566
10000	10037	3855
30000	30771	10163
50000	51519	16459
75000	76830	29381
100000	101879	42221

На рисунке 4.2 представлены данные из таблицы 4.2 в виде графика.

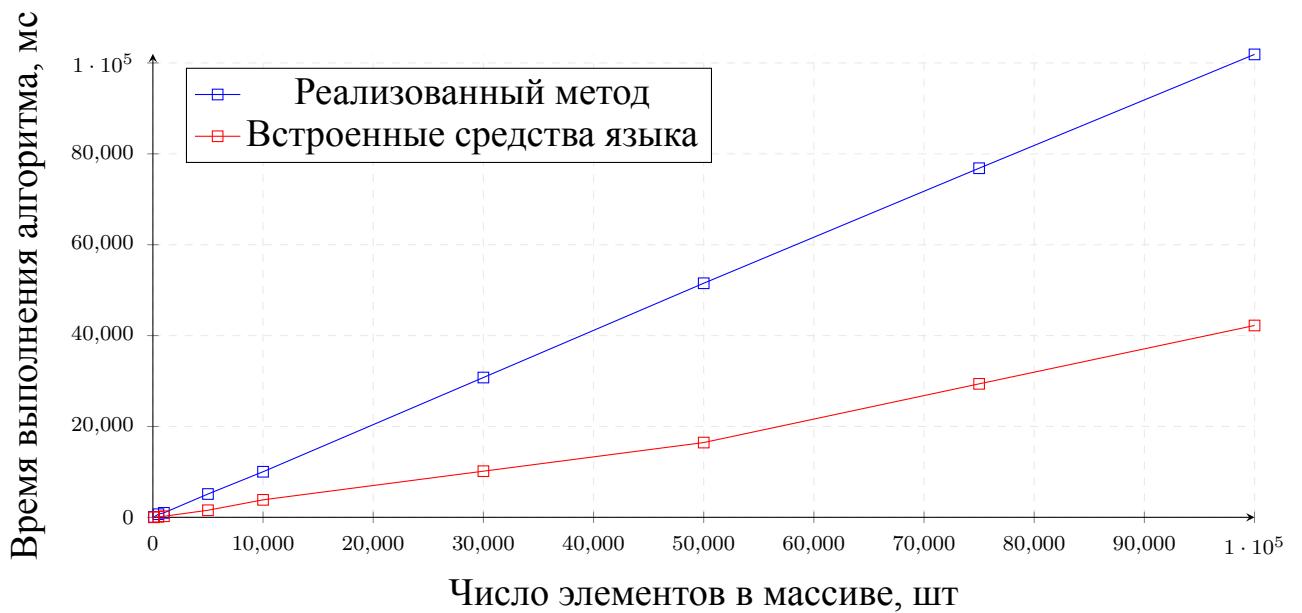


Рис. 4.2 – График зависимости времени выполнения алгоритма сортировки слиянием от числа элементов в массиве

4.4.3. Алгоритм нахождения расстояния Левенштейна

В таблице 4.3 представлены результаты замеров времени выполнения нерекурсивного алгоритма нахождения расстояния Левенштейна между строками равной длины.

Таблица 4.3 – Замеры времени выполнения алгоритма нахождения расстояния Левенштейна

<i>Число символов в строках, шт</i>	<i>Время выполнения при использовании реализованного метода, мс</i>	<i>Время выполнения при использовании встроенных средств языка, мс</i>
100	184	42
300	1780	326
500	5258	5730
750	11621	1818
1000	19222	3483
2000	73641	13963
3000	166382	31616
4000	346885	53427
5000	544065	86802

На рисунке 4.3 представлены данные из таблицы 4.3 в виде графика.

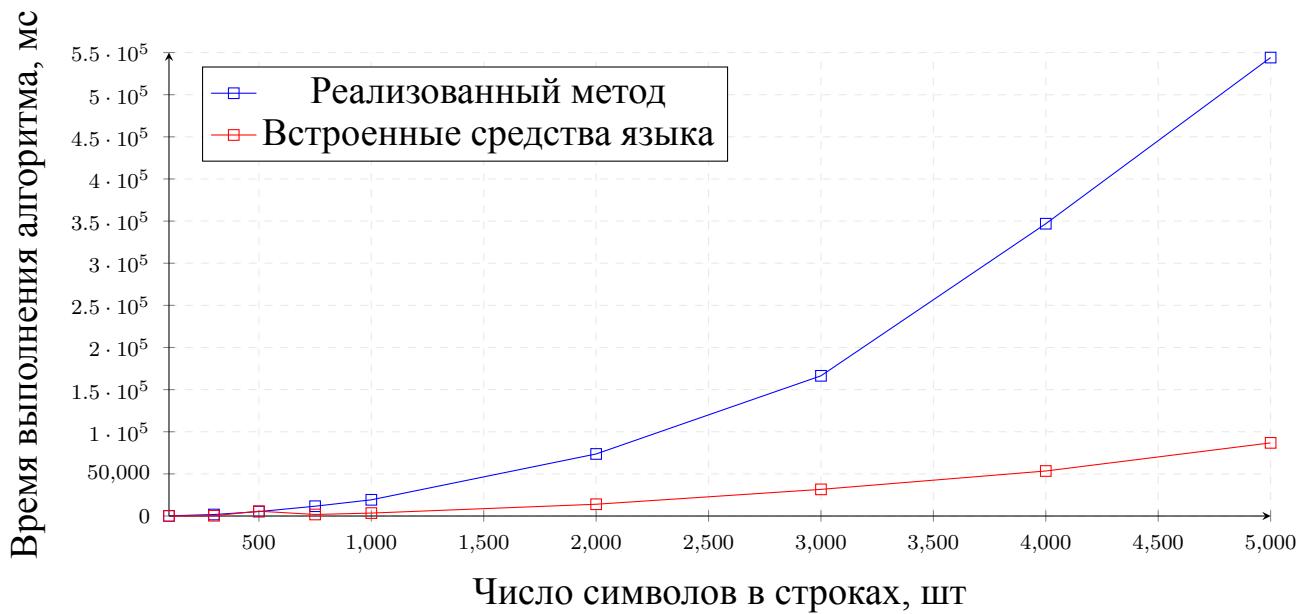


Рис. 4.3 – График зависимости времени выполнения алгоритма нахождения расстояния Левенштейна от числа символов в строках

4.4.4. Алгоритм умножения матриц по Винограду

В таблице 4.4 представлены результаты замеров времени выполнения алгоритма Винограда умножения квадратных матриц равной размерности.

Таблица 4.4 – Замеры времени выполнения алгоритма умножения матриц по Винограду

<i>Размерность матриц</i>	<i>Время выполнения при использовании реализованного метода, мс</i>	<i>Время выполнения при использовании встроенных средств языка, мс</i>
10	8	8
50	222	303
100	1495	2196
300	52627	78902
500	227228	406249
650	462251	839183
750	838280	1401997
900	1508186	3053736
1000	3049303	7017908

На рисунке 4.4 представлены данные из таблицы 4.4 в виде графика.

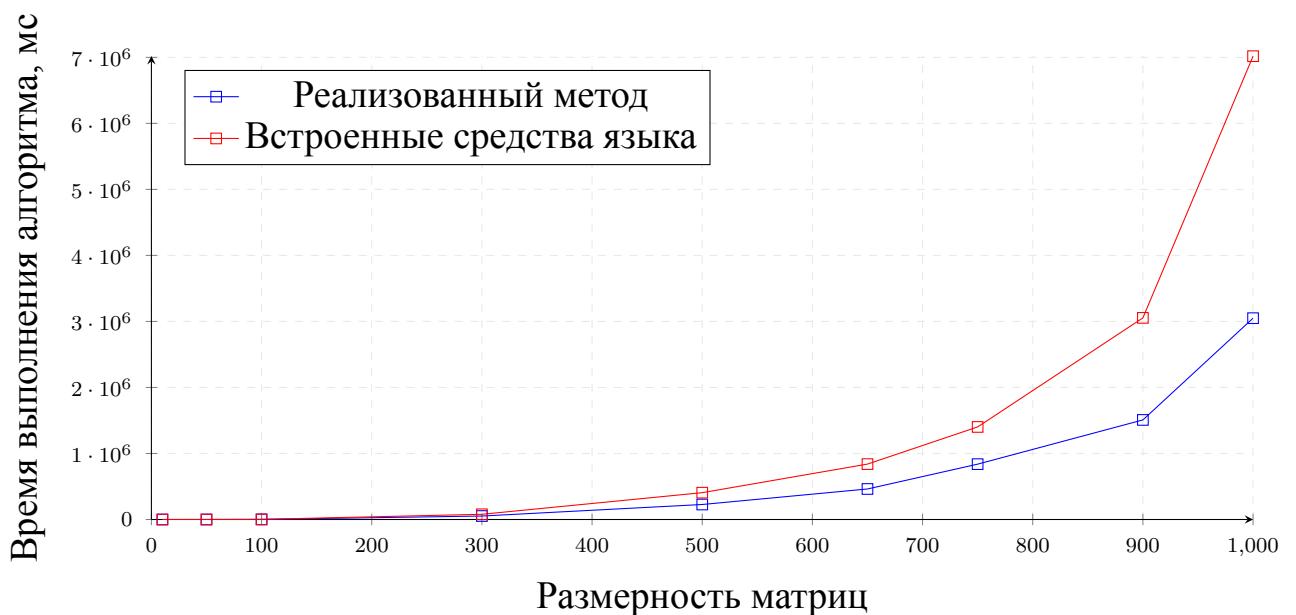


Рис. 4.4 – График зависимости времени выполнения алгоритма умножения матриц по Винограду от их размерности

4.4.5. Алгоритм сортировки подсчётом

В таблице 4.5 представлены результаты замеров времени выполнения алгоритма сортировки подсчётом при различных размерах входного массива целых чисел, находящихся в полуинтервале $[0; 10^7]$.

Таблица 4.5 – Замеры времени выполнения алгоритма сортировки подсчётом

<i>Число элементов в массиве, шт</i>	<i>Время выполнения при использовании реализованного метода, мс</i>	<i>Время выполнения при использовании встроенных средств языка, мс</i>
1000	14687	18527
10000	15220	18667
100000	16556	20360
500000	20712	23525
1000000	28030	32345
3000000	52895	55984
5000000	79864	78069
7500000	110101	109593
10000000	138367	137969

На рисунке 4.5 представлены данные из таблицы 4.5 в виде графика.

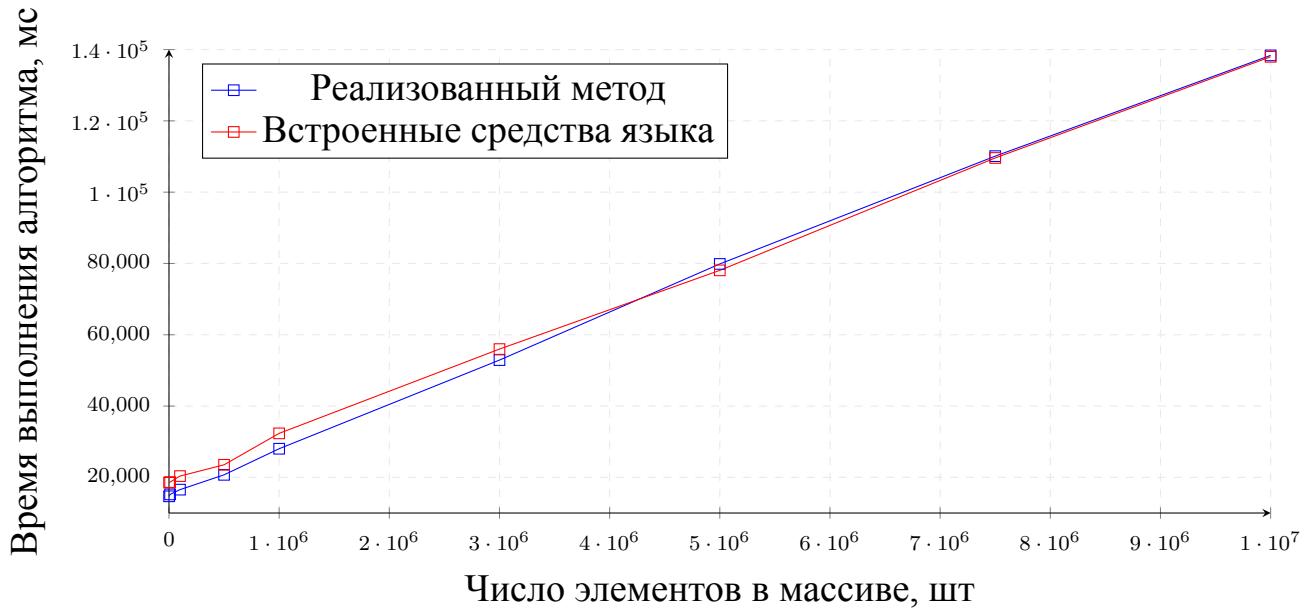


Рис. 4.5 – График зависимости времени выполнения алгоритма сортировки подсчётом от числа элементов в массиве

Выводы из исследовательского раздела

В данном разделе было проведено исследование влияния разработанного метода распределения памяти на время выполнения программ, реализующих различные классы алгоритмов. Сравнительный анализ показал, что для классов алгоритмов VP и VE разработанный метод оказался эффективнее реализации, существующей в языке программирования Golang. Использование разработанного метода позволило снизить время выполнения алгоритма Винограда умножения матриц (класс VP) на 0.3-56.5% в зависимости от размерности матриц, а время выполнения алгоритма сортировки подсчётом (класс VE) на 5.8-20.7% в зависимости от размера входного массива при условии, что он не достигает 5000000.

ЗАКЛЮЧЕНИЕ

В рамках настоящей работы был разработан и реализован метод распределения памяти. Все поставленные задачи были выполнены.

Был проведён анализ предметной области автоматического управления памятью, а также классификация существующих менеджеров памяти языков программирования. На основе их классификации были сформулированы концепции комбинированного метода: минимизация накладных расходов мутатора, времени сбора мусора и объёма памяти для хранения данных об объектах, применение конкурентной и параллельной сборки мусора, а также использование алгоритма поколений.

Была проведена классификация компьютерных алгоритмов по требованиям к дополнительной памяти. Для дальнейшего рассмотрения были выбраны алгоритмы, относящиеся к классам VC, VL, VQ, VP и VE.

Была дана гарантия времени выполнения реализованного метода в виде теоретической оценки трудоёмкости выполнения выделения памяти и обращения к выделенным объектам. Благодаря использованию модели поколений, удалось устранить зависимость накладных расходов мутатора от общего числа объектов во всей куче.

Для классов алгоритмов VP и VE разработанный метод оказался эффективнее реализации, существующей в языке программирования Golang. Использование разработанного метода позволило снизить время выполнения алгоритма Винограда умножения матриц (класс VP) на 0.3-56.5% в зависимости от размерности матриц, а время выполнения алгоритма сортировки подсчётом (класс VE) на 12-20.7% в зависимости от размера входного массива при условии, что он не достигает 5000000.

В качестве дальнейшего развития предлагается внедрение разработанного метода или отдельных его концепций в основной сборщик мусора языка Golang. Также в дальнейшем планируется исследование стабильности разработанного менеджера памяти, а также фрагментации кучи при его работе.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Таненбаум Э., Бос Х.* Современные операционные системы. — 4-е изд. — Питер, 2015.
2. ISO/IEC 9899:TC3 C standard [Электронный ресурс]. — Режим доступа, URL: <https://www.open-std.org/jtc1/sc22/wg14/www/docs/n1256.pdf> (дата обращения: 29.10.2023).
3. Standard C++ [Электронный ресурс]. — Режим доступа, URL: <https://isocpp.org/> (дата обращения: 29.10.2023).
4. The Go Programming Language documentation [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/> (дата обращения: 29.10.2023).
5. The Elixir programming language documentation [Электронный ресурс]. — Режим доступа, URL: <https://elixir-lang.org/> (дата обращения: 29.10.2023).
6. Memory Management Glossary [Электронный ресурс]. — Режим доступа, URL: <https://www.memorymanagement.org/glossary/> (дата обращения: 12.10.2023).
7. Richardson O. Reconsidering Custom Memory Allocation [Электронный ресурс]. — Режим доступа, URL: <https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/custom-alloc/> (дата обращения: 12.10.2023).
8. Common Language Runtime (CLR) overview [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/standard/clr> (дата обращения: 12.10.2023).
9. A theory of memory models / V. Saraswat [и др.] // Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. — 2007. — DOI: 10.1145/1229428.1229469.
10. Memory Management Overview [Электронный ресурс]. — Режим доступа, URL: <https://www.memorymanagement.org/mmref/begin.html> (дата обращения: 15.10.2023).
11. Allocation techniques [Электронный ресурс]. — Режим доступа, URL: <https://www.memorymanagement.org/mmref/alloc.html#mmref-alloc> (дата обращения: 12.10.2023).

12. Recycling techniques [Электронный ресурс]. — Режим доступа, URL: <https://www.memorymanagement.org/mmref/recycle.html> (дата обращения: 15.10.2023).
13. *Sinha O., Maitland O.* A Unified Theory of Garbage Collection [Электронный ресурс]. — Режим доступа, URL: <https://www.cs.cornell.edu/courses/cs6120/2019fa/blog/custom-alloc/> (дата обращения: 12.10.2023).
14. *Blackburn S. M., McKinley K. S.* Ulterior Reference Counting: Fast Garbage Collection without a Long Wait // Proceedings of the 2003 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications, OOPSLA 2003. — 2003. — DOI: 10.1145/949305.949336.
15. *Jones R., Hosking A., Moss E.* THE GARBAGE COLLECTION HANDBOOK. The Art of Automatic Memory Management. — CRC Press, 2012. — ISBN 978-1-4200-8279-1.
16. Python Programming Language documentation [Электронный ресурс]. — Режим доступа, URL: <https://www.python.org/> (дата обращения: 10.11.2023).
17. Jython documentation [Электронный ресурс]. — Режим доступа, URL: <https://www.jython.org/> (дата обращения: 10.11.2023).
18. IronPython: the Python programming language for .NET [Электронный ресурс]. — Режим доступа, URL: <https://ironpython.net/> (дата обращения: 10.11.2023).
19. PyPy documentation [Электронный ресурс]. — Режим доступа, URL: <https://www.python.org/> (дата обращения: 10.11.2023).
20. CPython definition [Электронный ресурс]. — Режим доступа, URL: <https://docs.python.org/3/glossary.html#term-CPython> (дата обращения: 10.11.2023).
21. Memory Management — Python 3.12.0 documentation [Электронный ресурс]. — Режим доступа, URL: <https://docs.python.org/3/c-api/memory.html> (дата обращения: 10.11.2023).
22. Garbage collector design [Электронный ресурс]. — Режим доступа, URL: <https://devguide.python.org/internals/garbage-collector/> (дата обращения: 10.11.2023).

23. PEP 556 — Threaded garbage collection [Электронный ресурс]. — Режим доступа, URL: <https://peps.python.org/pep-0556/> (дата обращения: 26.11.2023).
24. Java Garbage Collection Basics [Электронный ресурс]. — Режим доступа, URL: <https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html> (дата обращения: 16.11.2023).
25. Open J9 documentation [Электронный ресурс]. — Режим доступа, URL: <https://eclipse.dev/openj9/> (дата обращения: 16.11.2023).
26. Codename One: Cross-Platform App Development with Java [Электронный ресурс]. — Режим доступа, URL: <https://www.codenameone.com/> (дата обращения: 16.11.2023).
27. GraalVM documentation [Электронный ресурс]. — Режим доступа, URL: <https://www.graalvm.org/> (дата обращения: 16.11.2023).
28. The HotSpot Group [Электронный ресурс]. — Режим доступа, URL: <https://openjdk.org/groups/hotspot/> (дата обращения: 16.11.2023).
29. HotSpot VM Storage Management [Электронный ресурс]. — Режим доступа, URL: <https://openjdk.org/groups/hotspot/docs/StorageManagement.html> (дата обращения: 16.11.2023).
30. Memory Management in the Java HotSpot Virtual Machine [Электронный ресурс]. — Режим доступа, URL: <https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf> (дата обращения: 16.11.2023).
31. Troubleshooting Memory Issues in Java Applications [Электронный ресурс]. — Режим доступа, URL: https://www.oracle.com/webfolder/technetwork/tutorials/mooc/JVM_Troubleshooting/week1/lesson1.pdf (дата обращения: 16.11.2023).
32. Java SE 21. HotSpot Virtual Machine Garbage Collection Tuning Guide. Available Collectors [Электронный ресурс]. — Режим доступа, URL: <https://docs.oracle.com/en/java/javase/21/gctuning/available-collectors.html> (дата обращения: 16.11.2023).

33. Oracle JRockit Online Documentation Library Release 4.0. Understanding Memory Management [Электронный ресурс]. — Режим доступа, URL: https://docs.oracle.com/cd/E15289_01/JRSDK/garbage_collect.htm (дата обращения: 16.11.2023).
34. The Garbage First Garbage Collector [Электронный ресурс]. — Режим доступа, URL: <https://www.oracle.com/java/technologies/javase/hotspot-garbage-collection.html> (дата обращения: 16.11.2023).
35. The Z Garbage Collector (ZGC) [Электронный ресурс]. — Режим доступа, URL: <https://wiki.openjdk.org/display/zgc/Main> (дата обращения: 16.11.2023).
36. JavaScript documentation [Электронный ресурс]. — Режим доступа, URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (дата обращения: 12.11.2023).
37. ECMAScript 2023 language specification [Электронный ресурс]. — Режим доступа, URL: <https://ecma-international.org/publications-and-standards/standards/ecma-262/> (дата обращения: 12.11.2023).
38. Node.js documentation [Электронный ресурс]. — Режим доступа, URL: <https://nodejs.org> (дата обращения: 12.11.2023).
39. Apache CouchDB documentation [Электронный ресурс]. — Режим доступа, URL: <https://couchdb.apache.org/> (дата обращения: 12.11.2023).
40. The event loop [Электронный ресурс]. — Режим доступа, URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop (дата обращения: 12.11.2023).
41. JavaScript memory management [Электронный ресурс]. — Режим доступа, URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (дата обращения: 12.11.2023).
42. A tour of the C# language [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/> (дата обращения: 17.11.2023).
43. What is .NET? Introduction and overview [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/core/introduction> (дата обращения: 17.11.2023).

44. Introduction to the Common Language Runtime (CLR) [Электронный ресурс]. — Режим доступа, URL: <https://github.com/dotnet/runtime/blob/main/docs/design/coreclr/botr/intro-to-clr.md> (дата обращения: 17.11.2023).
45. Automatic Memory Management [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/standard/automatic-memory-management> (дата обращения: 17.11.2023).
46. The large object heap on Windows systems [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/large-object-heap> (дата обращения: 17.11.2023).
47. Fundamentals of garbage collection [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals> (дата обращения: 17.11.2023).
48. runtime package (Golang) [Электронный ресурс]. — Режим доступа, URL: <https://pkg.go.dev/runtime> (дата обращения: 4.11.2023).
49. Golang source file src/runtime/proc.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/proc.go> (дата обращения: 4.11.2023).
50. A Tour of Go. Goroutines [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/tour/concurrency/1> (дата обращения: 4.11.2023).
51. Golang source file src/runtime/mheap.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/mheap.go> (дата обращения: 4.11.2023).
52. Golang source file src/runtime/malloc.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/malloc.go> (дата обращения: 4.11.2023).
53. Golang source file src/runtime/mgc.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/mgc.go> (дата обращения: 5.11.2023).
54. Golang source file src/runtime/mbarrier.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/mbarrier.go> (дата обращения: 4.11.2023).
55. Ульянов М. В. Ресурсно-эффективные компьютерные алгоритмы. Разработка и анализ. — М.: ФИЗМАТЛИТ, 2008. — ISBN 978-5-9221-0950-5.

56. *Jeff E. Algorithms*. — 2019. — ISBN 978-1-792-64483-2.
57. Golang source file src/arena/arena.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/arena/arena.go> (дата обращения: 4.11.2023).
58. Golang source file src/runtime/arena.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/arena.go> (дата обращения: 21.4.2024).
59. reflect package (Golang) [Электронный ресурс]. — Режим доступа, URL: <https://pkg.go.dev/reflect> (дата обращения: 1.5.2024).

ПРИЛОЖЕНИЕ А

Презентация.