

РЕФЕРАТ

Расчетно–пояснительная записка 87 с., 18 рис., 1 табл., 77 ист., 1 прил.

УПРАВЛЕНИЕ ПАМЯТЬЮ, СБОРКА МУСОРА, JAVA, PYTHON

Цель работы: изучение алгоритмов распределения памяти в языках программирования с автоматической сборкой мусора.

В данной работе проводится изучение и сравнение алгоритмов автоматического управления памятью в языках программирования Python, Java, JavaScript, C# и Golang.

Среди рассмотренных автоматических менеджеров памяти были выделены менеджеры памяти языков программирования Python и Java как наиболее масштабируемые и оптимизированные для различных сценариев использования.

СОДЕРЖАНИЕ

РЕФЕРАТ	3
ВВЕДЕНИЕ	7
1 Анализ предметной области	9
1.1. Основные понятия	9
1.2. Управление памятью с точки зрения операционной системы	9
1.2.1. Иерархия памяти	9
1.2.2. Адресное пространство процесса	10
1.2.3. Виртуальная память	10
1.2.4. Функции операционной системы по управлению памятью	11
1.3. Управление памятью с точки зрения приложений	12
1.3.1. Среда выполнения языка программирования	12
1.3.2. Управление памятью	13
1.3.2.1. Ручное управление памятью	15
1.3.2.2. Автоматическое управление памятью	16
1.3.3. Трассирующая сборка мусора	18
1.3.3.1. Трёхцветная абстракция	18
1.3.3.2. Алгоритм mark-sweep	20
1.3.3.3. Алгоритм mark-compact	24
1.3.3.4. Копирующая сборка мусора	29
1.3.3.5. Алгоритм поколений	36
1.3.4. Подсчёт ссылок	36
1.3.4.1. Отложенный подсчёт ссылок	38
1.3.4.2. Взвешенный подсчёт ссылок	38
1.3.4.3. Использование счётчика ссылок с ограничен- ным полем	39
1.3.4.4. Подсчёт ссылок с флагом	39
1.3.4.5. Подсчёт циклических ссылок	40
2 Описание существующих решений	42
2.1. Python	42
2.1.1. Выделение памяти	42

2.1.2.	Структура объекта	43
2.1.3.	Сборка мусора	44
2.1.4.	Оптимизации	47
2.2.	Java	48
2.2.1.	Виртуальная машина Java	48
2.2.2.	Управление памятью	49
2.2.3.	Сборка мусора	51
2.2.3.1.	Serial Collector	51
2.2.3.2.	Parallel Collector	51
2.2.3.3.	Garbage-First Collector	52
2.2.3.4.	Z Garbage Collector	53
2.2.3.5.	Выбор сборщика мусора	55
2.3.	JavaScript	55
2.3.1.	Цикл событий	56
2.3.2.	Сборка мусора	57
2.3.3.	Вспомогательные структуры данных	57
2.4.	C#	59
2.4.1.	Платформа .NET	59
2.4.2.	Управление памятью	60
2.4.3.	Сборка мусора	62
2.5.	Golang	66
2.5.1.	Выделение памяти	66
2.5.2.	Структура кучи	69
2.5.3.	Сборка мусора	70
2.5.4.	Настройка сборщика мусора	72
2.5.5.	Аrenы памяти	74
3	Классификация существующих решений	76
3.1.	Критерии сравнения алгоритмов распределения памяти	76
3.2.	Сравнительный анализ алгоритмов распределения памяти	77
3.3.	Вывод	78

ЗАКЛЮЧЕНИЕ	79
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	86
ПРИЛОЖЕНИЕ А	87

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 активно 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.

Целью данной работы является изучение алгоритмов распределения памяти в языках программирования с автоматической сборкой мусора. Для достижения поставленной цели необходимо решить следующие задачи.

1. Проанализировать предметную область работы с памятью в языках программирования с автоматической сборкой мусора.
2. Рассмотреть существующие принципы организации работы с памятью в языках программирования с автоматической сборкой мусора на примере Python, Java, JavaScript, C# и Golang.
3. Описать алгоритмы сборки мусора в рассматриваемых языках.
4. Сформулировать критерии сравнения и оценки описанных алгоритмов.
5. Сравнить существующие решения по сформулированным критериям.

1 Анализ предметной области

1.1. Основные понятия

Управление памятью [6] — это процесс координации и контроля использования памяти в вычислительной системе. Управление памятью выполняется на трёх уровнях.

1. Аппаратное обеспечение управления памятью (MMU, ОЗУ и т.д.).
2. Управление памятью операционной системы (виртуальная память, защищата).
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.

1.2. Управление памятью с точки зрения операционной системы

1.2.1. 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 аб-

стракцию, 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]

1.2.2. Адресное пространство процесса

Чтобы допустить одновременное размещение в памяти нескольких приложений без создания взаимных помех, нужно решить две проблемы, относящиеся к защите и перемещению. Так была разработана новая абстракция операционной системы — адресное пространство. Так же, как понятие процесса создает своеобразный абстрактный центральный процессор для запуска программ, понятие адресного пространства создаёт своеобразную абстрактную память, в которой существуют программы. **Адресное пространство** [1] — это набор адресов, который может быть использован процессом для обращения к памяти. У каждого процесса имеется собственное адресное пространство, независимое от адресных пространств других процессов (за исключением тех случаев, когда процессам требуется совместное использование их адресных пространств).

В каждой операционной системе может по-разному задаваться структура адресного пространства процесса, но, как правило, она содержит две концептуальные области: стек и кучу. **Стек** [8] — область адресного пространства процесса, предназначенная для хранения параметров функций, локальных переменных и адреса возврата после вызова функции. **Куча** [9] — область адресного пространства процесса, предназначенная для выделения памяти, динамически запрашиваемой программой.

1.2.3. Виртуальная память

Для обеспечения одновременного содержания в памяти множества процессов были выработаны два основных подхода. Самый простой из них, называемый **свопингом**, заключается в размещении в памяти всего процесса целиком,

его запуске на некоторое время, а затем сбросе на диск. Бездействующие процессы большую часть времени хранятся на диске и в нерабочем состоянии не занимают пространство оперативной памяти (хотя некоторые из них периодически активизируются, чтобы проделать свою работу, после чего опять приостанавливаются). Второй подход называется **виртуальной памятью**, он позволяет программам запускаться даже в том случае, если они находятся в оперативной памяти лишь частично.

Виртуальная память — метод управления оперативной (внутренней) памятью компьютера, позволяющий выполнять программы, требующие больше оперативной памяти, чем имеется в компьютере, путём автоматического перемещения частей программы между оперативной памятью и вторичным хранилищем (файл на диске или раздел подкачки). В основе виртуальной памяти лежит идея о том, что у каждой программы имеется собственное адресное пространство, которое разбивается на участки, называемые **страницами** [1]. Каждая страница представляет собой непрерывный диапазон адресов. Эти страницы отображаются на физическую память, но для запуска программы одновременное присутствие в памяти всех страниц необязательно.

Если программа требует больше памяти, чем есть в распоряжении компьютера, то операционная система в соответствии с **алгоритмом замещения страниц** [1] выбирает страницы, которые выгружаются из оперативной памяти. Освобождённая оперативная память отдаётся другой программе, которая её запросила. В дальнейшем, если к выгруженной странице произойдёт обращение, операционная система выберет страницу для выгрузки из памяти, чтобы освободить место для загружаемой страницы.

Подмножество виртуального адресного пространства процесса, находящееся в физической памяти, называется **рабочим набором**. [8]

1.2.4. 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. Выделе-

ние 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.

1.3. Управление памятью с точки зрения приложений

1.3.1. 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: [10]

- 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. [11]

1.3.2. Управление памятью

Менеджер памяти приложения должен учитывать следующие ограничения. [12]

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. **Внутренняя фрагментация** [6] — явление, при котором аллокатор выделяет при каждом запросе больше памяти, чем фактически запрошено. **Внешняя фрагментация** [6] — явление, при котором свободная память разделена на множество мелких блоков, ни один из которых нельзя использовать для обслуживания запроса за выделение памяти.

Основная проблема управления памятью состоит в том, чтобы понять, когда следует сохранить содержащиеся в ней данные, а когда их следует уничтожить.

жить, 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. [12]

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, разделенных 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: последовательные 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. На-

пример, 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** [13]. 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.

1.3.2.1. 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. [12]

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. [13]

1.3.2.2. Автоматическое управление памятью

Автоматическое управление памятью [12] — это служба, которая автоматически перерабатывает память, которую программа не собирается использовать в дальнейшем. Эта служба, 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). Автоматическая переработка динамически выделяемой памяти называется **сборкой мусора**. Термин «мусор» используется для обозначения объектов, о которых достоверно известно, что они не используются программой, так являются недостижимыми из других её объектов. Термин «недостижимость» может иметь различные интерпретации в разных языках программирования. [6]

Задачи автоматического управления памятью:

- 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. Как правило, в таких языках автоматическая сборка мусора реализуется либо методом подсчёта ссылок, либо с использованием отслеживающего сборщика мусора. [14] Стоит отметить, что для автоматического управления памятью также можно использовать гибридные методы. [15] [16]

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. Такие действия называются **финализаторами**, а их выполнение — **финализацией** (finalization, termination). [6]

Как правило, финализация 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. [6]

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]

1.3.3. Трассирующая сборка мусора

Сборщик мусора [6] — автоматический менеджер памяти, реализующий некоторый алгоритм сборки мусора.

Трассирующий сборщик мусора [14] — автоматический менеджер памяти, который следует указателям, чтобы определить, какие области памяти доступны из переменных программы, называемых **корневым набором**).

Целью идеального сборщика мусора является освобождение пространства, используемого каждым объектом, как только он перестаёт использоваться программой.

Далее будут подробно рассмотрены основные 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. [17]

1.3.3.1. Трёхцветная абстракция

Для разработки и описания алгоритмов сборки мусора важно иметь краткий способ описания состояния объектов во время сборки (были ли они помечены, есть ли они в рабочем списке и так далее). Трёхцветная абстракция позволя-

ет трассирующим сборщикам мусора судить о корректности сборки в терминах инвариантов, которые сборщики должны сохранять. В рамках трехцветной абстракции трассирующий сборщик мусора разбивает граф объектов 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 черным. Концептуально объект является черным, если сборщик завершил его обработку, и серым, если сборщик знает о нем, но еще не закончил его обработку (или должен обработать его снова). По аналогии с цветом объекта, полям также может быть присвоен цвет: серый, когда сборщик впервые сталкивается с ними, и черный, когда он обработал их. Трассировка выполняется по всей куче путём перемещения волнового фронта сборщика мусора (серые объекты), отделяющего чёрные объекты от белых до тех пор, пока все доступные объекты не будут окрашены чёрным цветом. [17]

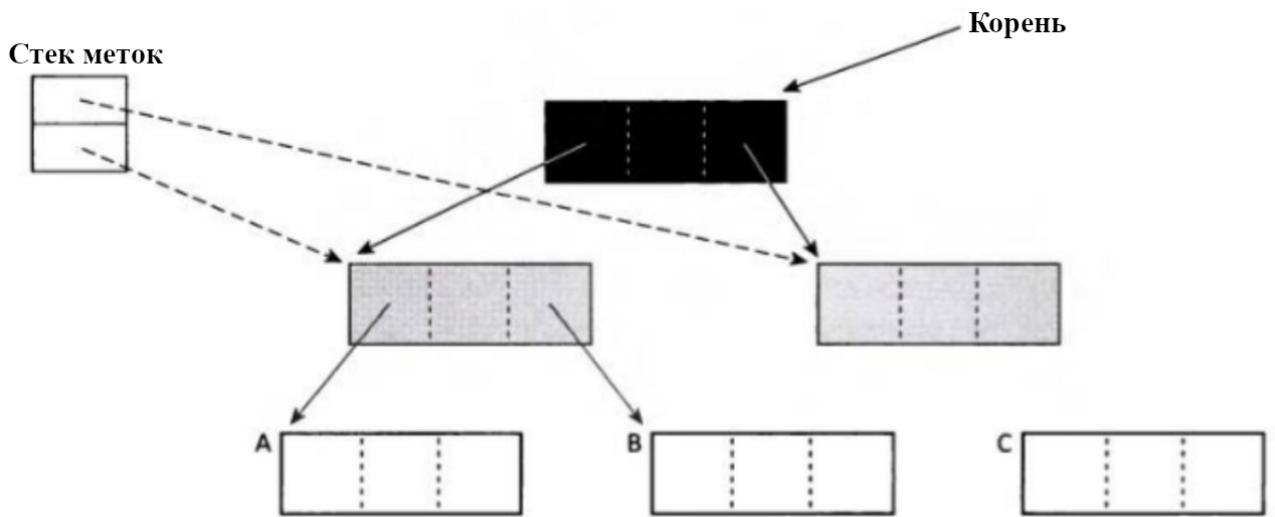


Рис. 1.1 – Пример трёхцветной разметки

На рисунке 1.1 показан пример разметки графа объектов и стек меток, реализующий рабочий список, в процессе работы сборщика мусора. Все объекты, хранящиеся в стеке меток, будут посещены, поэтому они окрашиваются в серый цвет. Любой объект, который был размечен и не находится в стеке, является чёрным (корень графа на рисунке), а все остальные объекты окрашиваются в белый цвет (в настоящее время А, В и С). Однако, как только разметка графа

завершится, стек меток будет пуст (без серых объектов) и только С останется белым (мусор), а все остальные объекты будут окрашены чёрным.

Алгоритм сохраняет важный инвариант: в конце каждой итерации цикла разметки отсутствуют ссылки с чёрных объектов на белые. Таким образом, любой доступный белый объект должен быть доступен из серого объекта. Если бы этот инвариант был нарушен, то используемый потомок чёрного объекта мог быть окрашен белым (и, следовательно, был бы освобожден некорректно), поскольку сборщик не обрабатывает чёрные объекты дальше. Трехцветное представление состояния объектов программы особенно полезно при рассмотрении алгоритмов параллельной сборки мусора, когда она выполняется конкурентно с потоками основной программы. [17]

1.3.3.2. Алгоритм mark-sweep

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. [17]

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. [17]

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»). [17]

На рисунках 1.2-1.4 представлены схемы алгоритма mark-sweep, использу-

зующего битовую карту размеченных областей кучи (bitmap). [17]

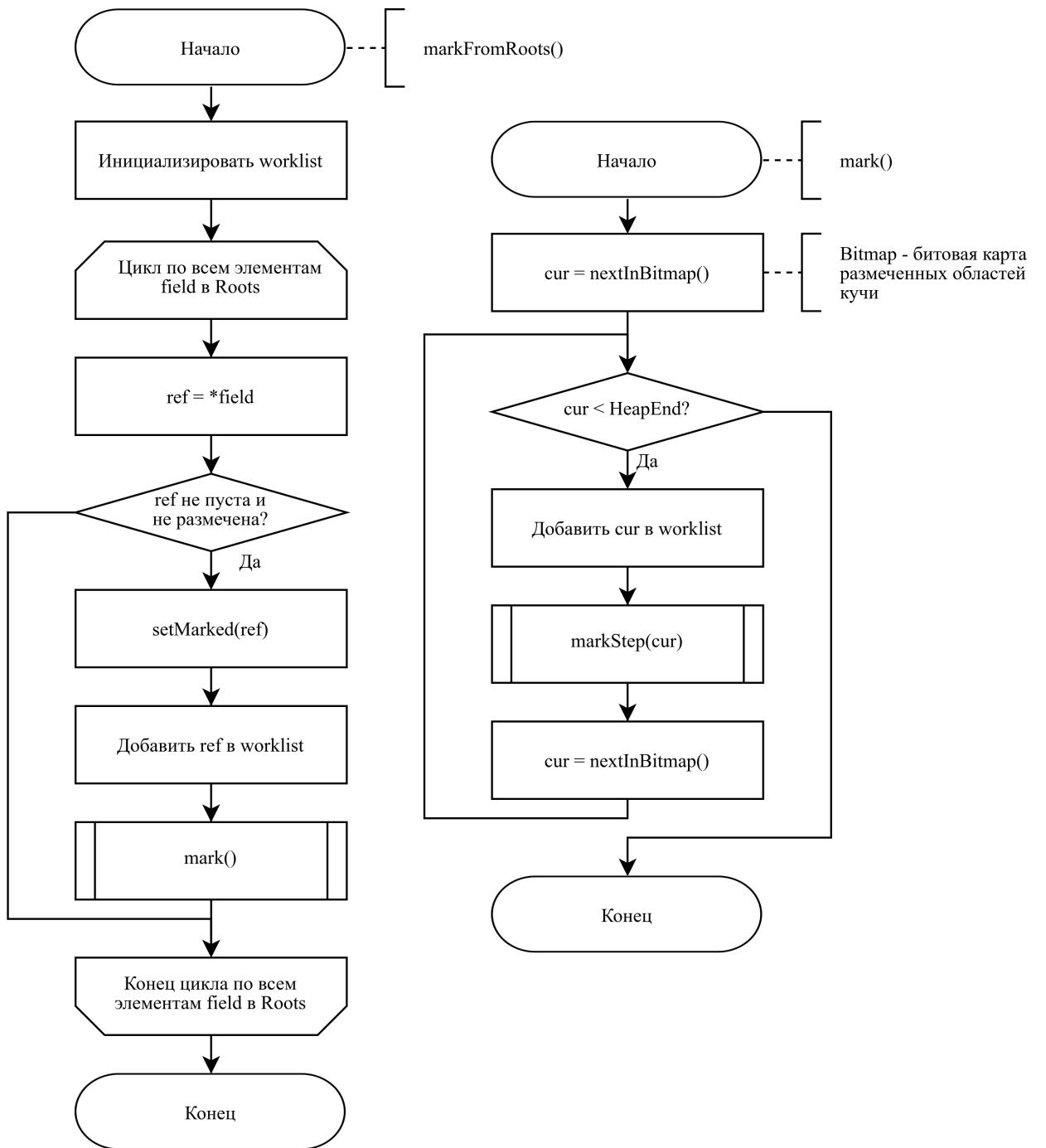


Рис. 1.2 – Функции разметки объектов из корневого набора и рабочего списка

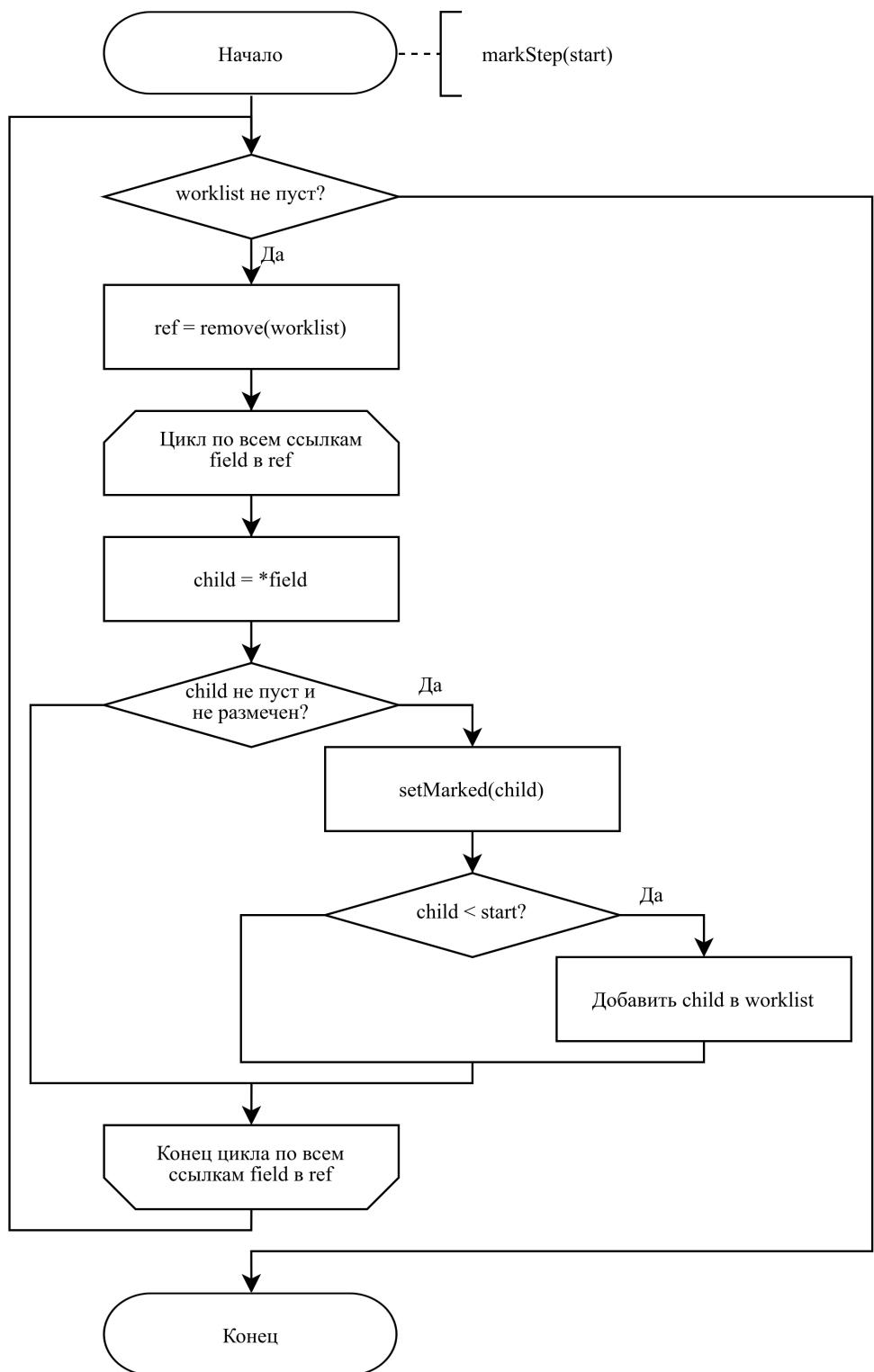


Рис. 1.3 – Функция разметки одного объекта

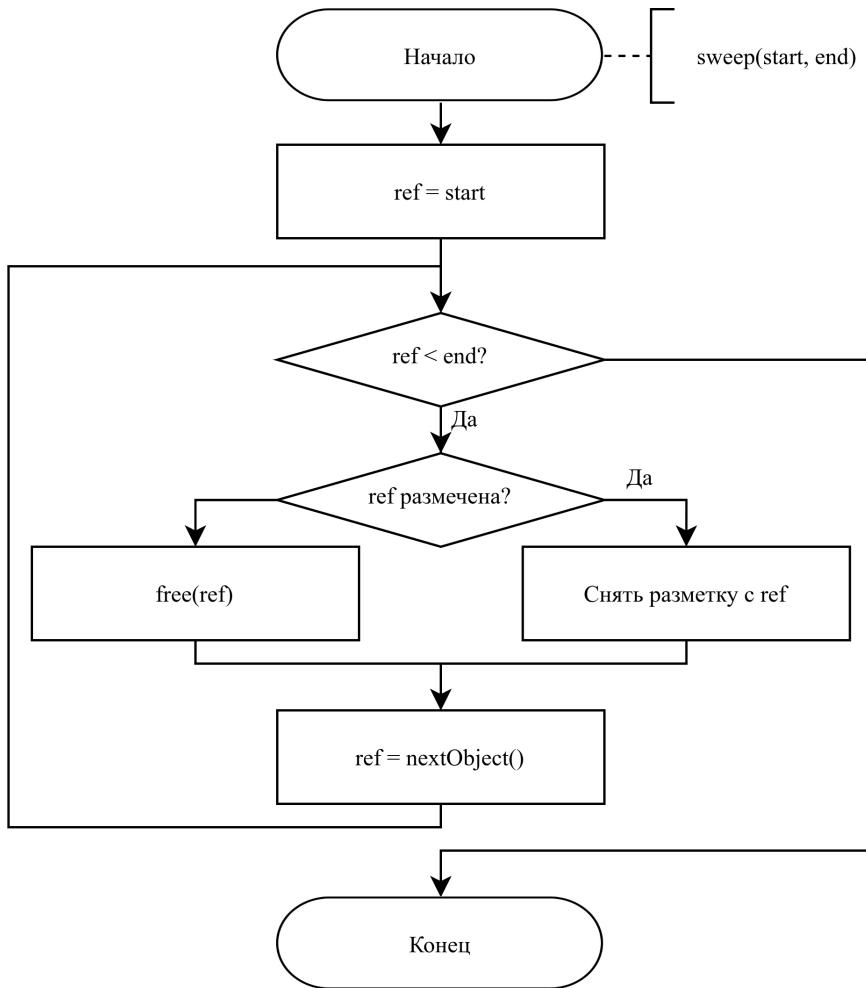


Рис. 1.4 – Функция очистки

Как и другие алгоритмы трассировки, алгоритм mark-sweep должен идентифицировать все используемые объекты в пространстве, прежде чем сможет освободить память, используемую любыми мусорными объектами. Это относительно дорогостоящая операция, поэтому её следует проводить как можно реже. Это означает, что трассирующим сборщикам мусора необходимо предоставить некоторый запас памяти для работы в куче. Если используемые объекты занимают слишком большую часть кучи, а аллокаторы выделяют память слишком быстро, то сборщик мусора, работающий по алгоритму mark-sweep, будет вызываться слишком часто и замедлять основную программу. Для обеспечения пропускной способности, сравнимой с той, что обеспечивается ручным управлением памятью, запас памяти в куче должен составлять не менее 20%. [17]

Как правило, в длительно работающих приложениях куча имеет тенденцию к фрагментации. Для неперемещающихся аллокаторов требуется пространство, в $O(\log(\max/\min))$ большее минимально возможного, где \min и \max — наименьший и наибольший возможный размеры объекта. Таким образом, возмож-

но, придется вызывать неуплотняющий сборщик мусора чаще, чем тот, который уплотняет используемые объекты. [17]

Чтобы избежать снижения производительности из-за чрезмерной фрагментации, многие сборщики, использующие алгоритм mark-sweep для управления кучей, также периодически используют другой алгоритм, такой как mark-compact, для её дефрагментации. Такой подход особенно актуален, если приложение не поддерживает достаточно постоянные соотношения размеров объектов или выделяет много относительно больших объектов. Понизить склонность кучи к фрагментации также может помочь использование того факта, что объекты склонны к выделению и освобождению группами. Размещение объектов группами также может уменьшить необходимость в уплотнении кучи. [17]

1.3.3.3. Алгоритм mark-compact

Фрагментация может быть проблемой для неперемещающихся сборщиков мусора. Несмотря на то, что в куче может быть свободное пространство, либо может отсутствовать непрерывная область, достаточно большая для удовлетворения запроса на выделение памяти, либо время, затрачиваемое на выделение, может стать чрезмерным, поскольку менеджеру памяти приходится искать подходящее свободное пространство. Для борьбы с фрагментацией аллокаторы могут хранить небольшие объекты одинакового размера в смежных областях памяти. Это особенно актуально для приложений, которые не выделяют много относительно больших объектов. [17]

Однако многие длительно работающие приложения, управляемые неперемещающимися сборщиками, будут 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. [17]

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. Сборщик может перемещать объекты в куче одним из трех способов. [17]

- произвольный (arbitrary): объекты перемещаются независимо от их первоначального порядка или от того, указывают ли они друг на друга;
- линеаризованный (linearising): объекты перемещаются таким образом, чтобы они находились рядом со связанными объектами, с которыми они связаны ссылками или образуют одну структуру данных;
- скользящий (sliding): объекты перемещаются в один конец кучи, вытесняя мусор, тем самым изменяя их первоначальный порядок размещения в куче.

Большинство уплотняющих сборщиков использует произвольное или скользящее перемещение. Уплотнение в произвольном порядке приводит к плохой пространственной локализации объектов программы, поскольку связанные объекты могут быть распределены по разным страницам виртуальной памяти. Все современные реализации алгоритма mark-compact основаны на скользящем перемещении, которое не изменяет относительный порядок размещения объектов в памяти и, следовательно, не влияет на их локальность. [17]

На рисунках 1.5-1.8 представлены схемы алгоритма mark-compact, использующиеся сборщиком мусора Lisp 2. [17]

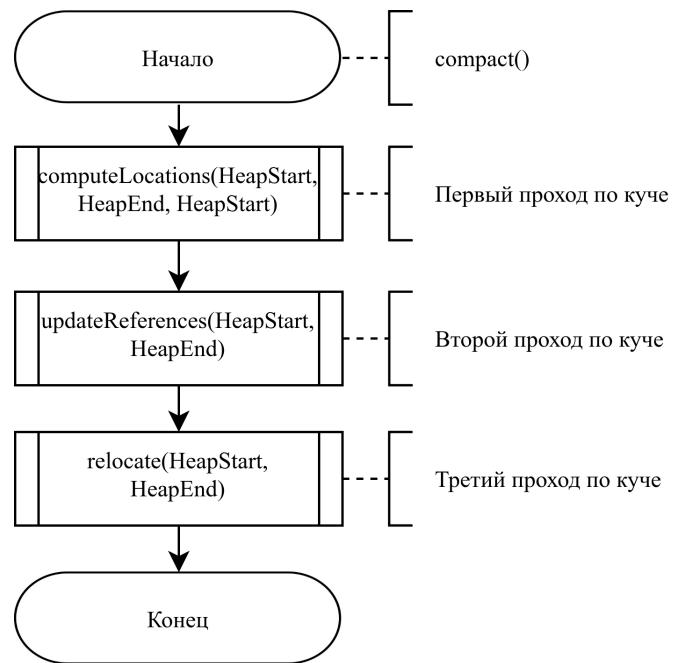


Рис. 1.5 – Функция уплотнения объектов в куче

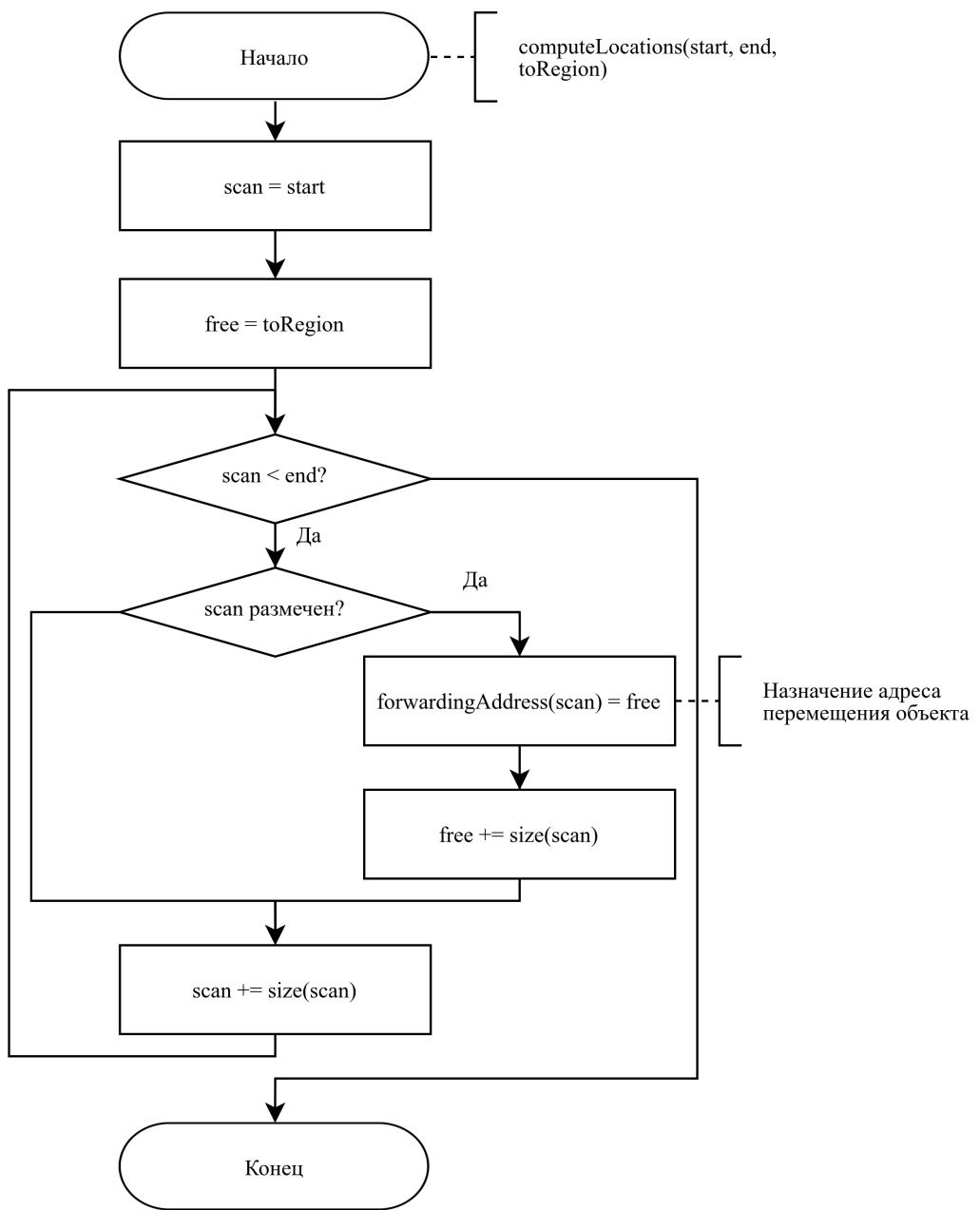


Рис. 1.6 – Первый проход по куче: вычисление адресов перемещения используемых объектов

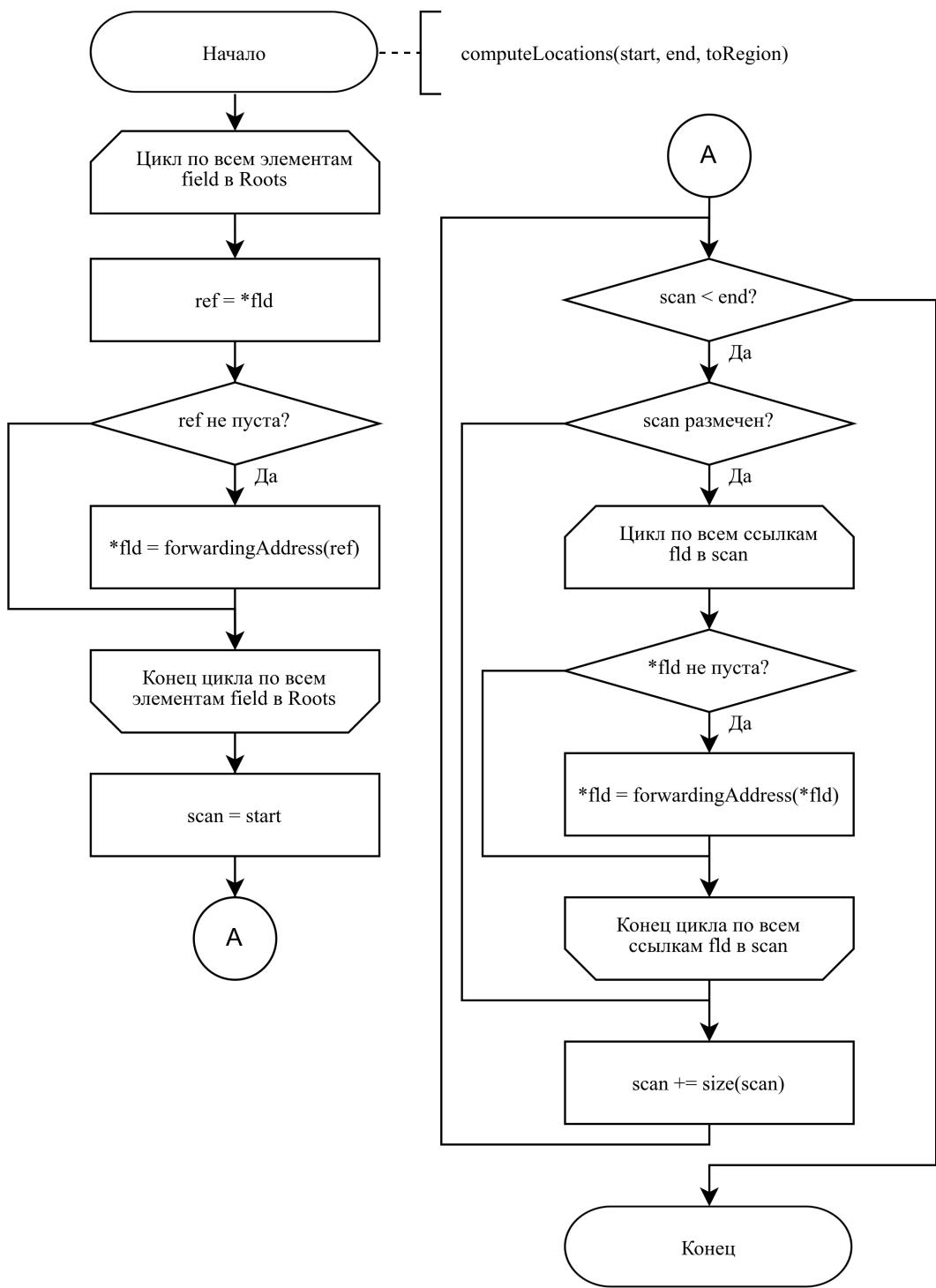


Рис. 1.7 – Второй проход по куче: обновление ссылок в корневом наборе и в размеченных объектах

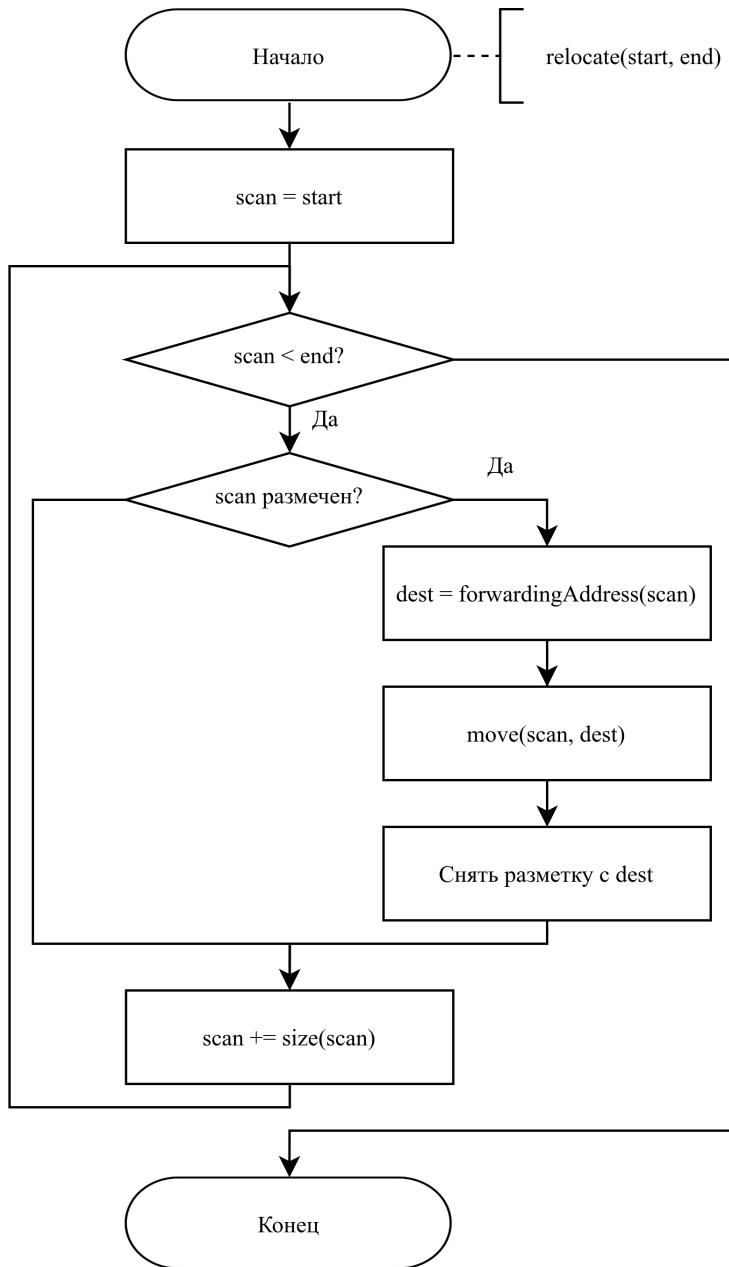


Рис. 1.8 – Третий проход по куче: перемещение используемых объектов

Для сокращения количества проходов сборщика мусора по куче до двух (один для разметки и один для перемещения объектов) и снижения накладных расходов при многопоточной сборке можно хранить адреса перемещения в дополнительной таблице, которая сохраняется во время уплотнения. Данный подход реализуют **однопроходные алгоритмы**.

1.3.3.4. Копирующая сборка мусора

Третий метод трассирующей сборки мусора — полупространственное копирование (semispace copying). Уплотнение кучи в данном подходе обеспечивает быстрое распределение и требует только одного прохода по используемым

объектам в куче. Его главным недостатком является то, что он уменьшает размер доступной кучи вдвое. [17]

Как правило, сборщики копирования делят кучу на два полупространства равного размера, называемых **старым** (old space, fromspace) и **новым пространством** (new space, tospace). Создаваемые объекты размещаются в «новом пространстве», если имеется достаточное количество памяти, иначе запускается сборка мусора. На абстрактном уровне всё, что делает копирующий сборщик мусора, — начинает с корневого набора и обходит все доступные объекты, выделенные в памяти, копируя их из одного полупространства в другое. При копировании объектов происходит их уплотнение таким образом, чтобы они занимали непрерывную область памяти. После копирования объектов «старое пространство» освобождается. Такой подход устраниет «дыры» в памяти, занимаемые неиспользуемыми объектами. После копирования и уплотнения получается сжатая копия данных в «новом пространстве» и, возможно, увеличивается непрерывная область памяти в «старом пространстве», в котором можно размещать новые объекты. На следующем цикле сборки мусора «старое» и «новое» пространства меняются ролями. [17] [18]

На рисунках 1.9-1.11 представлены схемы алгоритма копирующей сборки мусора, использующего полупространственное копирование. [17]

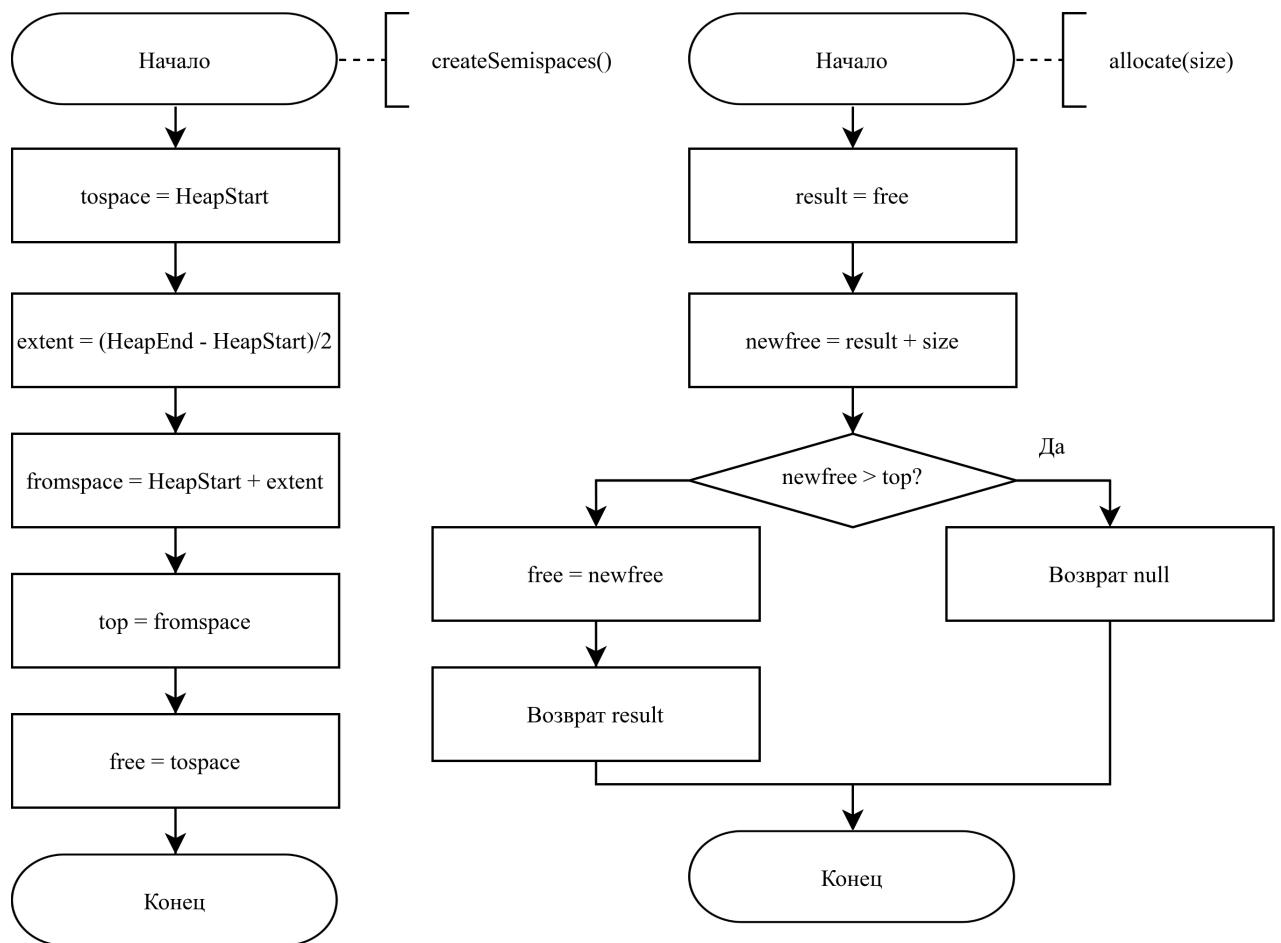


Рис. 1.9 – Функции для создания полупространств кучи и выделения памяти в них

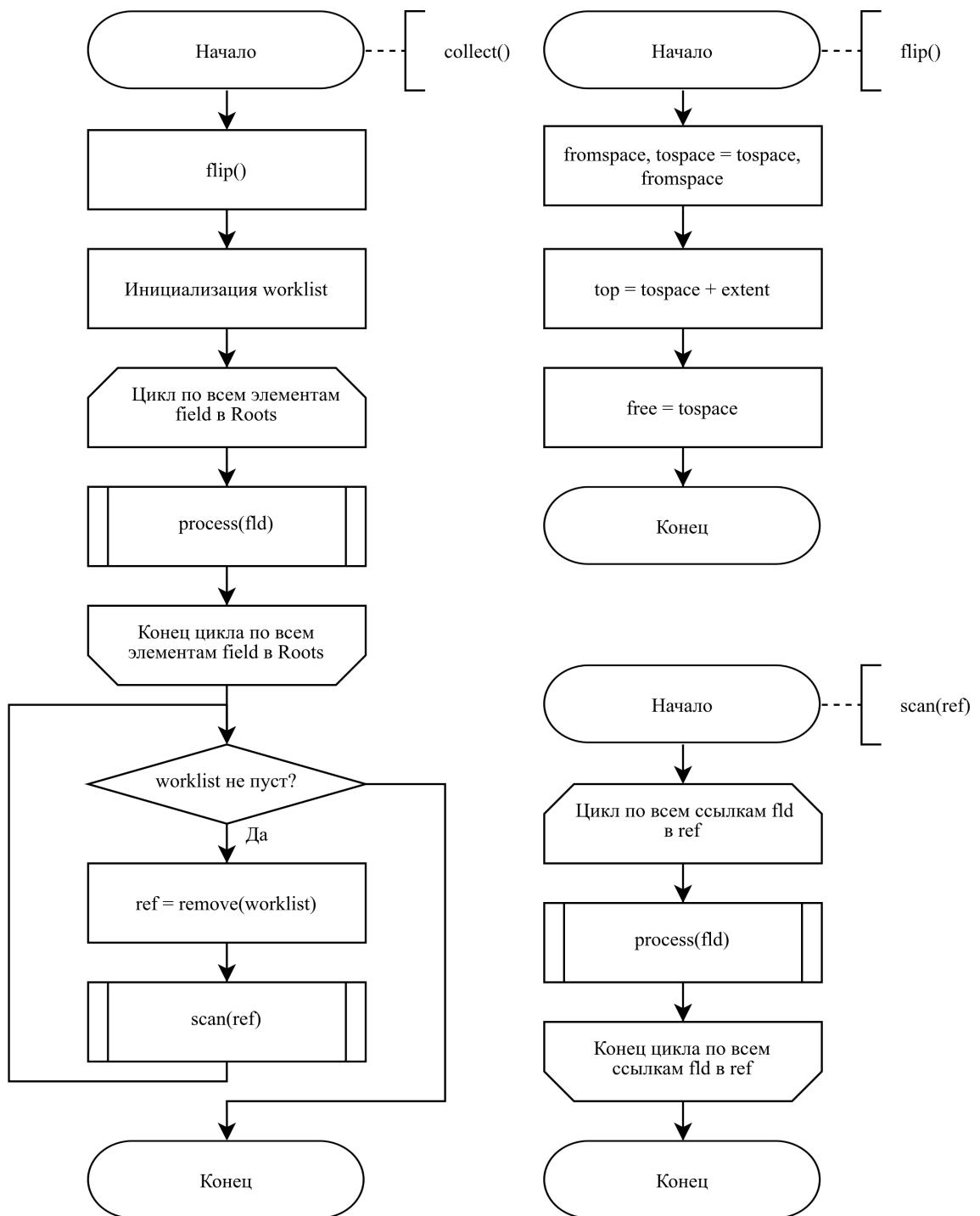


Рис. 1.10 – Функции для сбора мусора, обмена полупространств местами и сканирования ссылки на объект

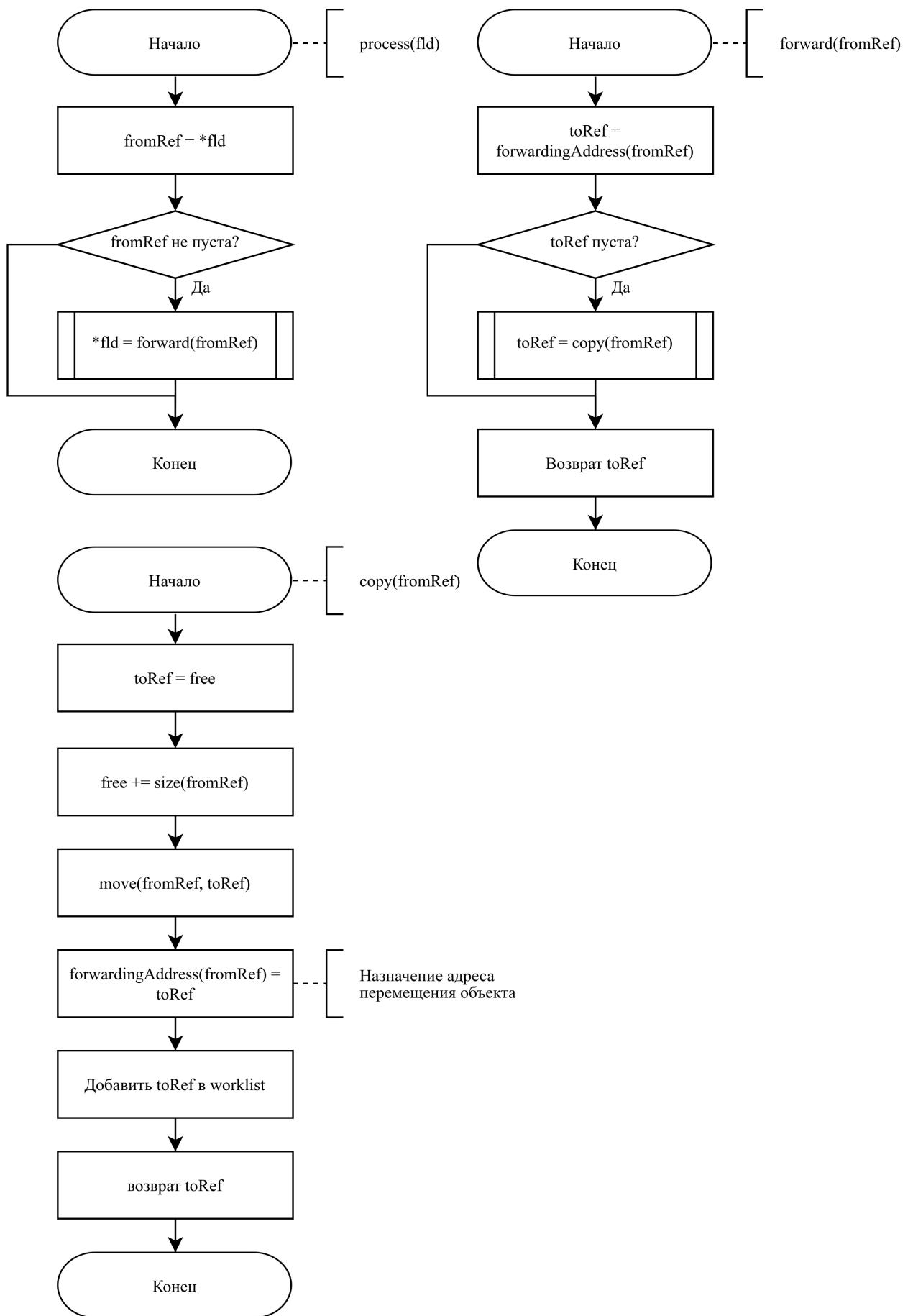


Рис. 1.11 – Функции для перемещения и обновления ссылки на объект

Важным недостатком полупространственного копирования является необходимость хранения второго полупространства, иногда называемого **резервом копирования** (copy reserve), без возможности выделять память в нём. При ограниченном объёме доступной памяти и игнорировании структур данных, необходимых сборщику мусора, такой подход к распределению памяти обеспечивает возможность работы только с половиной пространства кучи, что существенно меньше того объёма памяти, который предоставляют другие менеджеры памяти. Следовательно, копирующие сборщики мусора будут выполнять больше циклов сборки, чем другие сборщики. Приведёт ли это к изменению производительности, зависит от характеристик прикладной программы и объёма доступного пространства в куче. [17]

Анализ асимптотической сложности может свидетельствовать в пользу копирующей сборки мусора. Пусть M — общий размер кучи, а L — объём используемых данных. Сборщики полупространств должны копировать, сканировать и обновлять указатели в L . Сборщики, работающие по алгоритму mark-sweep, должны аналогичным образом отслеживать все используемые объекты и затем зачищать всю кучу. Временные сложности алгоритмов mark-sweep и semispace copying можно определить следующим образом: [17]

$$t_{MS} = mL + sM, \quad (1.1)$$

$$t_{Copy} = cL, \quad (1.2)$$

где c , m и s — некоторые коэффициенты, не зависящие от M и L .

Объём памяти, освобождаемый каждым сборщиком, равен

$$m_{MS} = M - L, \quad (1.3)$$

$$m_{Copy} = M/2 - L. \quad (1.4)$$

Пусть $r = L/M$ — размер используемых данных в куче, которые мы предполагаем постоянными. Так, эффективность алгоритма может быть описана количеством работы, выполненной сборщиком мусора при выделении одного объекта (соотношение «mark/cons» [17]).

$$e_{MS} = \frac{mr + s}{1 - r}, \quad (1.5)$$

$$e_{Copy} = \frac{2cr}{1 - 2r}. \quad (1.6)$$

На рисунке 1.12 представлен график зависимости эффективности алгоритма сборки мусора (e) от доли используемых объектов кучи (live ratio). [17]

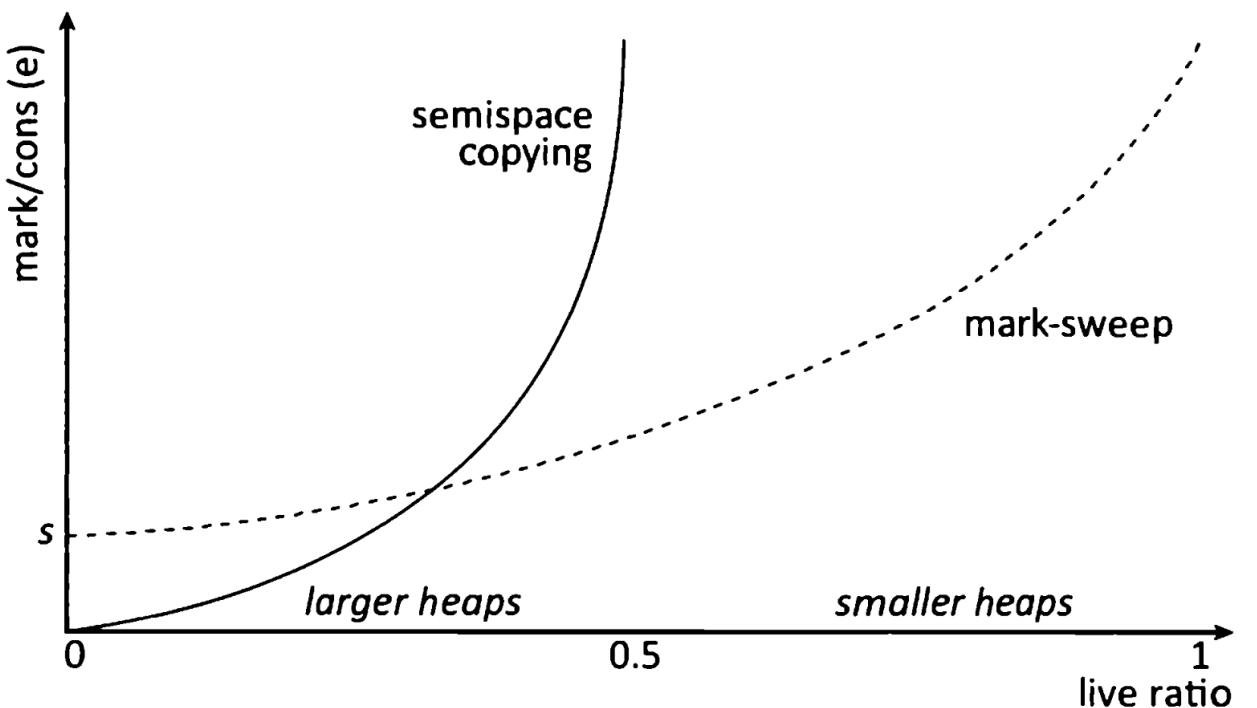


Рис. 1.12 – Соотношение «mark/cons» для алгоритмов mark-sweep и semispace copying (чем меньше, тем алгоритм эффективнее)

Как показал анализ асимптотической сложности, алгоритм semispace copying может быть более эффективным, чем mark-sweep, при условии, что куча достаточно велика, а коэффициент r достаточно мал. Стоит заметить, что такой анализ игнорирует некоторые аспекты реализации алгоритмов. Современные сборщики, основанные на алгоритме mark-sweep, зачастую используют **ленивую зачистку** (lazy sweep, очистка памяти при выделении), тем самым уменьшая константу s и снижая соотношение e . Также анализ сложности игнорирует преимущество локализации объектов при последовательном распределении, когда объекты, к которым доступ осуществляется одновременно, размещаются в соседних областях памяти (см. п. 1.3.2.).

1.3.3.5. Алгоритм поколений

Одна из основных проблем копирующего сборщика мусора заключается в том, что ему приходится просматривать всю кучу при каждом запуске сборки мусора. Объекты в памяти обладают важным свойством временной устойчивости, которое можно сформулировать в виде следующей гипотезы. [18]

Гипотеза поколений (generational hypothesis, infant mortality) [6] — это наблюдение, согласно которому в большинстве случаев «молодые» объекты (те, которые создаются позже), с гораздо большей вероятностью перестают использоваться («умирают») раньше, чем «старые» объекты. Строго говоря, гипотеза состоит в том, что вероятность смерти объекта как функция его возраста (времени жизни) убывает гиперэкспоненциально.

Сборка мусора по алгоритму поколений [6] — это трассирующая сборка мусора, в которой используется гипотеза поколений. **Поколением** называется группа объектов со схожим временем жизни. Новые объекты выделяются в самом молодом поколении и передаются в старшие поколения, если они выживают. Объекты более старых поколений обрабатываются реже, что экономит время процессора.

Как правило, объекты одного поколения имеют мало ссылок на объекты более молодых поколений. Это означает, что сканирование старых поколений в процессе сбора более молодых поколений можно осуществлять реже, отдельно запоминая ссылки на объекты более молодых поколений. [18]

Использование гипотезы поколений позволяет модифицировать алгоритм копирующей сборки мусора. Так, первоначально созданные объекты будут размещены в области памяти, называемой первым поколением. Когда оно заполняется, производится копирование используемых объектов в другую область памяти, называемую вторым поколением, и освобождается первое поколение. Такая последовательность действий повторяется для новых поколений. [18]

1.3.4. 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). [14]

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; [17]
- устойчивость к высоким нагрузкам, так как потенциально подсчёт ссылок может переработать объект как только он перестаёт использоваться; [17]
- 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; [17]
- может быть реализован без помощи среды выполнения языка и без её ведома. [17]

Недостатки подсчёта ссылок:

- требуются накладные расходы на операции чтения и записи для управления числом ссылок;
- операции с числом ссылок на объекты должны быть атомарными; [17]
- случай циклических ссылок требует отдельного рассмотрения; [17]
- подсчёт ссылок может вызывать паузы, например при освобождении ссылочных структур данных. [17]

По сравнению со сборкой мусора алгоритмы управления памятью, основанные на подсчёте ссылок, отличаются детерминированностью, предсказуемостью и меньшей вычислительной сложностью. [15]

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. [14]

Далее будут подробно описаны некоторые подходы к реализации подсчёта ссылок. [14]

1.3.4.1. Отложенный подсчёт ссылок

Отложенный подсчёт ссылок позволяет уменьшить накладные расходы на поддержание счётчика ссылок в актуальном состоянии за счёт отсутствия корректировок при сохранении ссылки в стеке. [6]

Как правило, большинство сохранений ссылок производится в локальные переменные, которые хранятся в стеке. Отложенный подсчёт ссылок позволяет обойтись без них и считать только ссылки, хранящиеся в объектах кучи. Это требует поддержки компилятора, но может привести к значительному повышению производительности сборки мусора. [6]

Объекты не могут быть возвращены, как только счетчик ссылок на них станет равным нулю, поскольку на них еще могут быть ссылки из переменных на стеке. Такие объекты добавляются в «таблицу нулевого счёта» (Zero Count Table, ZCT). Если в куче сохраняется ссылка на объект с нулевым счётчиком, то этот объект удаляется из ZCT. Периодически производится сканирование стека, и все объекты в ZCT, на которые не было ссылок из стека, освобождаются. [6]

Стоит заметить, что отложенный подсчёт ссылок не позволяет корректно обрабатывать объекты с циклическими ссылками, поэтому его часто используют вместе с трассирующим сборщиком мусора. [14]

1.3.4.2. Взвешенный подсчёт ссылок

Распределенная сборка мусора [6] — это сборка мусора в системе, в которой объекты могут храниться в разных адресных пространствах или на разных машинах.

Распределенная сборка мусора влечёт за собой накладные расходы на синхронизацию и связь между процессами. Эти расходы особенно высоки при использовании трассирующего сборщика мусора, поэтому вместо него обычно используются другие методы, например взвешенный подсчёт ссылок. [6]

Взвешенный подсчёт ссылок [6] – техника подсчёта ссылок, которая ши-

роко используется для распределенной сборки мусора из-за низкого уровня межпроцессного взаимодействия.

Межпроцессные ссылки на объекты подсчитываются, но вместо простого подсчёта количества ссылок каждой ссылке присваивается некоторый вес. При создании объекта начальному указателю на него присваивается вес, который, как правило, кратен двум. В объекте записывается сумма весов всех его ссылок. При копировании ссылки её вес делится поровну между новой и оригинальной ссылками. Так как при этой операции сохраняется взвешенная сумма ссылок, то связь с объектом в этот момент не требуется. При удалении ссылки взвешенная сумма ссылки уменьшается на ее вес. Об этом сообщается объекту путем посылки сообщения. Когда счётчик ссылок объекта становится равным нулю, он может быть освобождён. Алгоритм устойчив к протоколам связи, не гарантирующим порядок поступления сообщений об удалении. [6]

1.3.4.3. Использование счётчика ссылок с ограниченным полем

При данном подходе к подсчёту ссылок поле, используемое для хранения числа ссылок на объект, имеет ограниченный размер. В частности, число ссылок на объект может быть настолько большим, что не может быть сохранено в этом поле. [6]

Исходя из того, что на большинство объектов, как правило, не ссылаются большое количество раз, некоторые системы, использующие подсчёт ссылок, хранят точное количество ссылок только до определенного максимального значения. Если объект имеет больше ссылок, чем это значение, то счетчик фиксируется на максимальном значении и никогда не декрементируется. Предполагается, что такие объекты встречаются редко, но их память никогда не может быть освобождена с помощью подсчёта ссылок. Для освобождения такой памяти часто используется отдельный трассирующий сборщик мусора. [6]

1.3.4.4. Подсчёт ссылок с флагом

Подсчёт ссылок с флагом (one-bit reference counting) [6] — это эвристический механизм, позволяющий с минимальными накладными расходами памяти проверить, используется ли объект в программе.

Однобитовый счетчик ссылок является частным случаем счетчика ссы-

лок с ограниченным полем (limited-field reference count). Один бит в объекте, называемый MRB (Multiple Reference Bit), очищается при выделении объекта. Всякий раз, когда создаётся новая ссылка на объект, этот бит устанавливается. Таким образом, $MRB=0$ означает, что на объект имеется ровно одна ссылка, а $MRB=1$ — что на объект может быть более одной ссылки. [6]

MRB может храниться в ссылке, а не в объекте. Это уменьшает количество обращений к памяти, связанных с проверкой и установкой MRB. При копировании ссылки устанавливается MRB так же копируется. Если MRB оригинальной ссылки равен 0, то его также необходимо установить. Установка MRB оригинальной ссылки требует, что её местоположение известно и на момент установки не было перезаписано другими данными. [6]

Подсчёт ссылок с флагом может использоваться компиляторами для анализа времени жизни объекта. Когда анализ во время компиляции предсказывает, что конкретный объект может быть освобождён (обычно из-за того, что переменная, ссылающаяся на объект, уничтожена), компилятор может сгенерировать код, который будет проверять MRB объекта во время выполнения. Если MRB равен 0, то объект можно освободить. [6]

Использование подсчёта ссылок с флагом имеет свои издержки: MRB занимает дополнительную память и его необходимо устанавливать каждый раз, когда количество ссылок на объект увеличивается. Однако эти накладные расходы меньше, чем при использовании других способов подсчёта ссылок, так как MRB занимает всего один бит и счётчик ссылок не корректируется при уничтожении ссылок на объект. [6]

1.3.4.5. Подсчёт циклических ссылок

Поскольку число ссылок на объекты, составляющие циклическую структуру данных, не меньше единицы, подсчёт ссылок сам по себе не может восстановить такие структуры. Однако такие циклы повсеместно используются в программах, например, при создании многосвязных списков и циклических буферов. [17]

Наиболее широко распространенные механизмы обработки циклов посредством подсчёта ссылок используют метод, называемый **пробным удалением**. Его ключевой особенностью является то, что трассирующему сборщику мусора нет необходимости просматривать весь граф используемых объектов.

Вместо этого можно рассматривать только те части графа, в которых удаление указателя могло бы привести к циклу сборки мусора. Стоит заметить, что в любой структуре мусорных указателей все ссылки должны быть связаны с внутренними указателями, то есть с указателями между объектами внутри структуры. Циклы сборки мусора могут возникать только при удалении указателя, в результате которого число ссылок на объект становится равным нулю. [17]

Алгоритмы **частичной трассировки** отслеживают подграф, корнем которого является объект, подозреваемый в том, что он является мусором. Эти алгоритмы пробно удаляют каждую встречающуюся ссылку, временно уменьшая число ссылок, фактически устранивая вклад этих внутренних указателей. Если число ссылок на какой-либо объект остаётся ненулевым, то это означает, что существует указатель на объект не из вершин подграфа, и, следовательно, ни сам объект, ни те объекты, на которые он ссылается, не являются мусором. [17]

Алгоритм сборки циклических структур данных работает в три этапа. [17]

1. **Разметка.** Сначала сборщик отслеживает подграфы, начиная с объектов, идентифицированных как возможные элементы мусорных циклов, уменьшая число ссылок из-за внутренних указателей. Посещённые объекты окрашиваются в серый цвет (см. п. 1.3.3.1.).
2. **Сканирование.** Проверяется число ссылок для каждого объекта в этих подграфах: если оно не равно нулю, то рассматриваемый объект должен быть используемым из-за ссылки, являющейся внешней по отношению к отслеживаемому подграфу, и поэтому результат первого этапа должен быть отменён, перекрашивая живые серые объекты в чёрный цвет. Другие серые объекты окрашиваются в белый цвет.
3. **Очистка.** После завершения сканирования все элементы подграфа, которые являются белыми, могут быть освобождены.

2 Описание существующих решений

2.1. Python

Python [19] — интерпретируемый язык программирования с сильной динамической типизацией. Официально язык был представлен в 1991 году. Существует множество реализаций интерпретатора языка Python [20] [21] [22]. Среди них канонической считается реализация CPython, разработанная на языке C и официально представлена в 1994 году. [23]

Интерпретатор CPython использует механизм **GIL** (Global Interpreter Lock, глобальная блокировка интерпретатора), который обеспечивает однопоточное выполнение байт-кода Python. Этот механизм упрощает реализацию CPython, делая объектную модель (включая встроенные типы, такие как dict) неявно защищённой от конкурентного доступа и упрощая многопоточное выполнение программ. Однако некоторые модули расширения (как стандартные, так и сторонние) [24] освобождают GIL при выполнении ресурсоёмких задач, таких как сжатие или хеширование. Кроме того, GIL всегда освобождается при выполнении ввода-вывода. [25]

2.1.1. Выделение памяти

Все объекты программы на языке Python размещаются в **приватной куче** (private heap), управление которой обеспечивается встроенным в интерпретатор диспетчером памяти. [26]

На самом низком уровне аллокатор необработанной памяти (raw memory allocator) взаимодействует с диспетчером памяти операционной системы и гарантирует, что в приватной куче есть достаточно места для хранения всех данных, используемых программой. Помимо аллокатора необработанной памяти используется несколько объектно-ориентированных аллокаторов, которые реализуют различные политики управления памятью, адаптированные к особенностям каждого типа данных. Диспетчер памяти Python делегирует часть работы объектно-ориентированным аллокаторам, гарантируя, что они работают в пределах приватной кучи. [26]

Для выделения памяти допускается использование как функций API Python/C, так и функций стандартной библиотеки языка C. В большинстве ситуаций рекомендуется выделять память в приватной куче, чтобы интерпретатор

Python имел более точное представление об использовании памяти. [26]

Все функции распределения памяти в языке Python принадлежат одному из трёх **доменов** (PyMemAllocatorDomain). Эти домены представляют разные стратегии распределения и оптимизированы для разных целей. [26]

1. **Raw domain** предназначен для выделения памяти под буферы общего назначения в тех случаях, когда выделение должно передаваться системному аллокатору или когда аллокатор может работать без GIL. Память запрашивается непосредственно у операционной системы.
2. «**Mem» domain» предназначен для выделения памяти для буферов Python и буферов общего назначения, где выделение должно выполняться с удержанием GIL. Память берётся из приватной кучи Python.**
3. **Object domain** предназначен для выделения памяти для объектов Python. Память берётся из приватной кучи Python.

При освобождении памяти, ранее выделенной функциями, принадлежащими какому-либо домену, необходимо использовать соответствующие функции освобождения из того же домена. [26]

2.1.2. Структура объекта

Основной алгоритм сборки мусора, используемый CPython, — подсчет 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 2.1. [27]

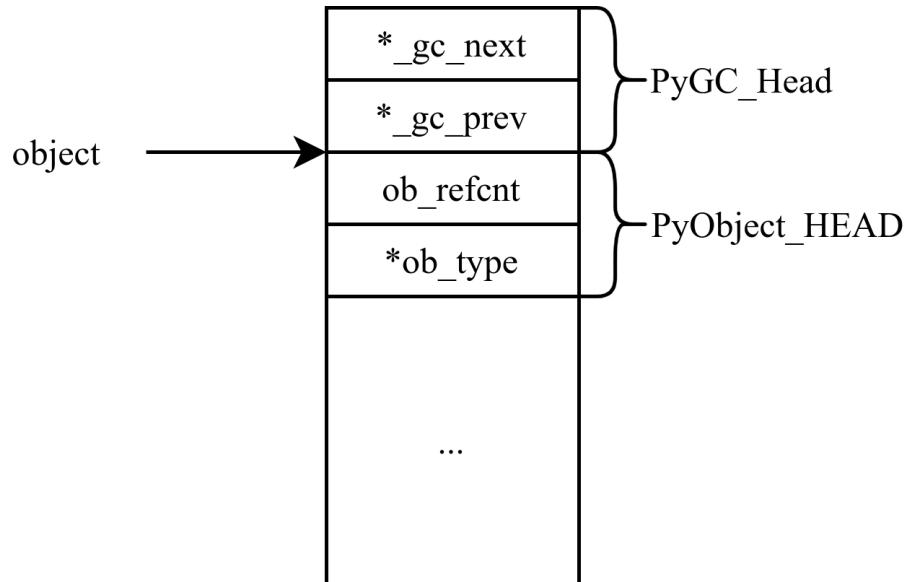


Рис. 2.1 – Структура объекта Python

Когда необходима дополнительная информация, связанная со сборщиком мусора, к полям `PyGC_Head` можно получить доступ с помощью адресной арифметики и приведения типа исходного объекта. [27]

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. [27]

2.1.3. Сборка мусора

Основная проблема подсчета 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. [27]

На рисунках 2.2 и 2.3 представлены схемы алгоритмов сборки мусора, которые обрабатывают циклические ссылки. [27]

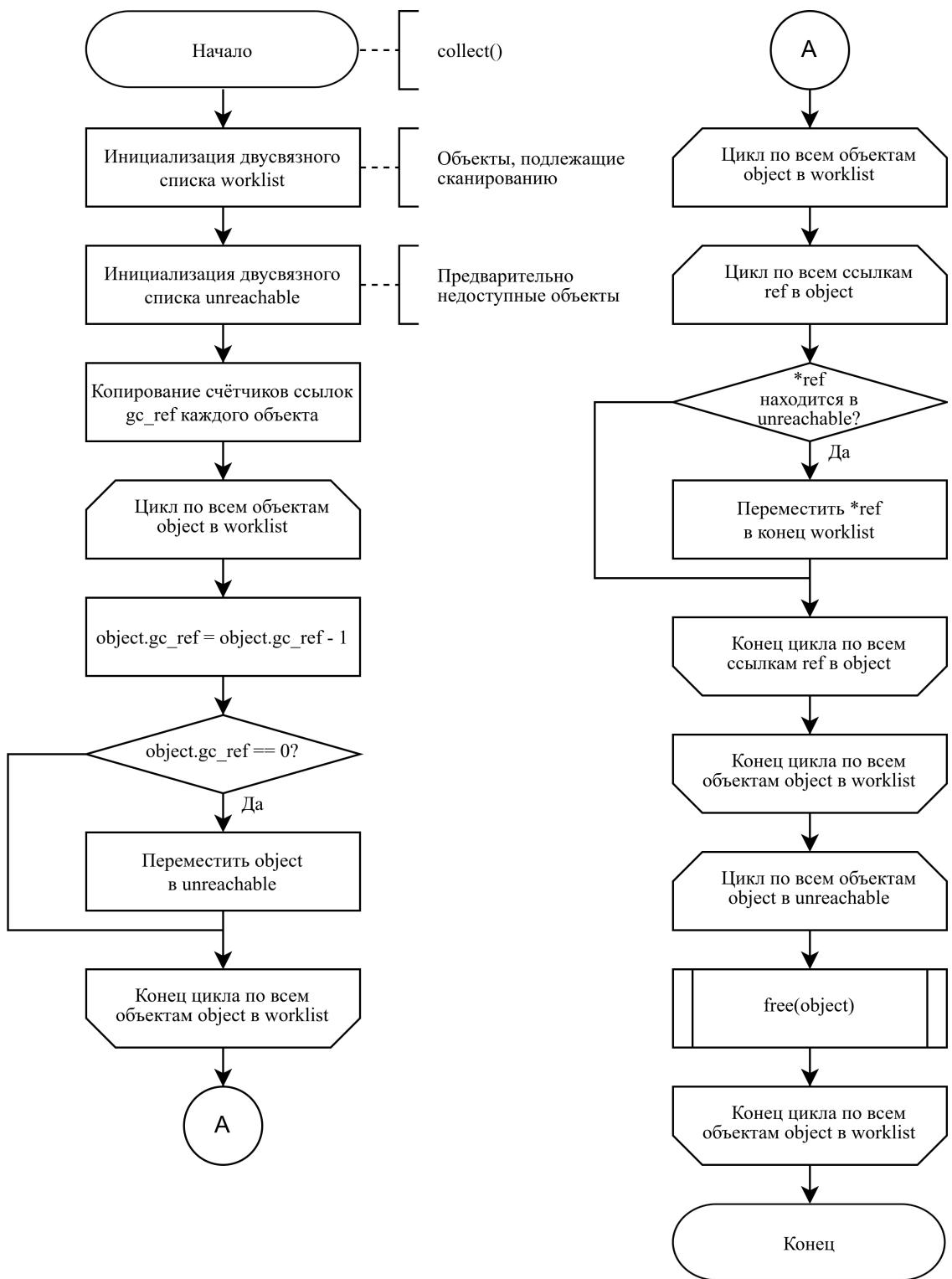


Рис. 2.2 – Схема алгоритма сбора циклических ссылок

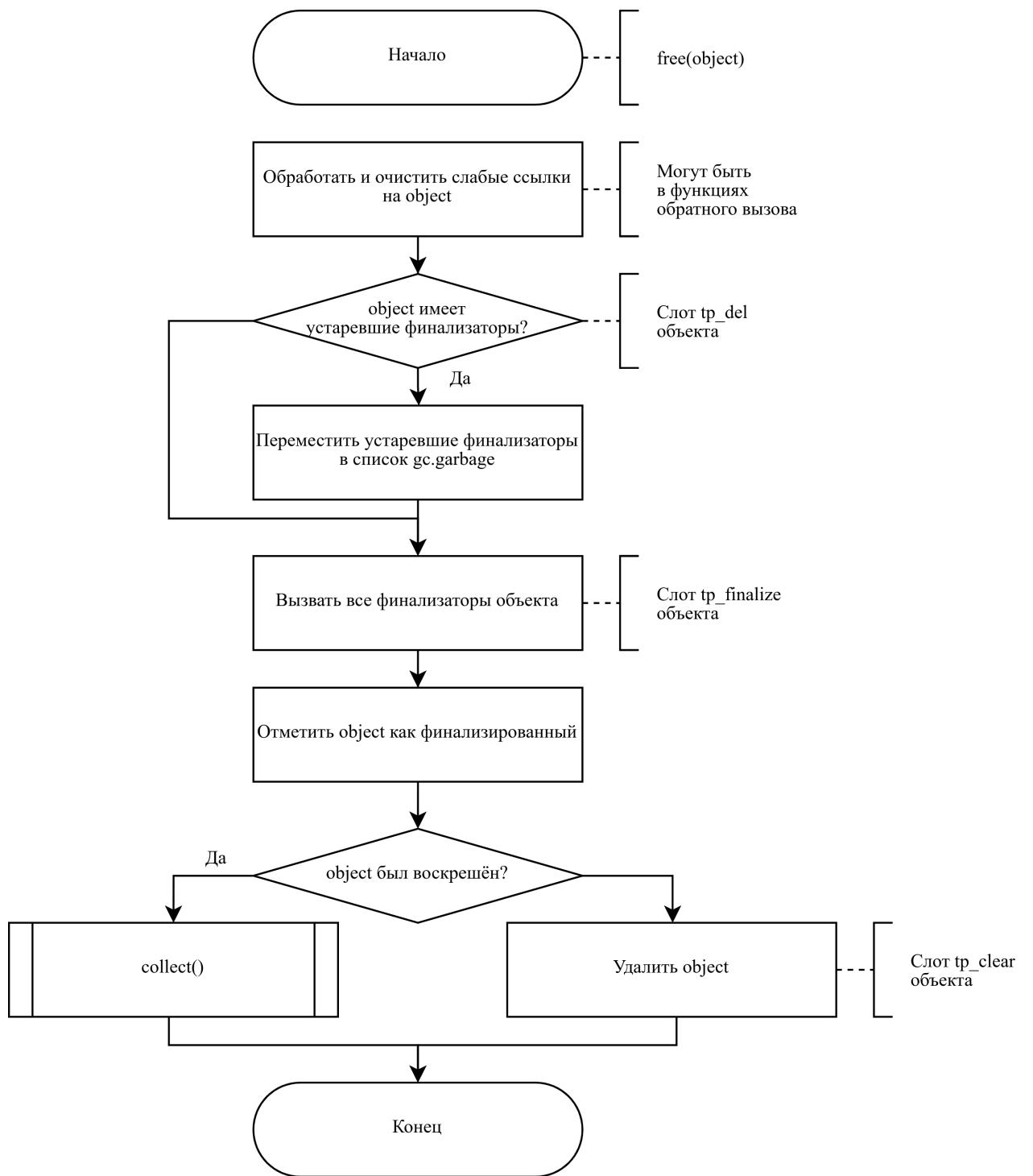


Рис. 2.3 – Схема алгоритма освобождения объекта

Стоит отметить, что такая сборка мусора выполняется конкурентно с основной программой в одном выделенном потоке сборщика, не распределяя работу по сборке на несколько потоков приложения. [28]

2.1.4. Оптимизации

Для снижения накладных расходов на сборку мусора в языке Python используются следующие оптимизации.

1. **Алгоритм поколений** (см. п. 1.3.3.5.). Для его реализации все объекты-контейнеры разделяются на три пространства (поколения). Каждый новый объект помещается в первое поколение. Алгоритм сбора мусора выполняется только над объектами определённого поколения, 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. Чтобы определить момент запуска сбора мусора, сборщик отслеживает количество выделений и освобождений объектов с момента последнего сбора. Когда разность между ними превысит заданный порог (threshold), начнётся сбор. Первоначально исследуется только первое поколение. Если количество его сканирований после последнего анализа первого поколения превысило заданный порог, то также анализируется второе поколение. Последнее поколение сканируется только в том случае, если количество выделений объектов превышает количество освобождений на заданное значение (по умолчанию задано 25%). [27]
2. **Повторное использование полей ссылок.** В целях экономии памяти указатели из поля PyGC_Head каждого объекта используются для нескольких целей. Эта оптимизация носит название «толстые указатели» (fat pointers) или «теговые указатели» (tag pointers). Так, внутрь указателей помещаются дополнительные данные с учётом следующего свойства адресации памяти: большинство архитектур выравнивают размеры типов данных по определённой величине, зачастую кратной машинному слову. Следовательно, несколько младших бит указателя всегда заполнены нулями и их можно использовать для хранения другой информации, чаще всего в виде битового поля. [27]
3. **Задержка отслеживания контейнеров.** Определённые типы контейнеров не могут участвовать в цикле ссылок, поэтому сборщик мусора не

отслеживает их. Существует два случая, когда отслеживание контейнера откладывается:

- когда контейнер создан;
- когда контейнер сканируется сборщиком мусора.

Как правило, экземпляры простых (атомарных) типов данных не отслеживаются, а экземпляры агрегированных типов (контейнеры, определяемые пользователем объекты и т.д.) отслеживаются. Тем не менее, могут быть предусмотрены некоторые оптимизации для конкретных типов, чтобы снизить влияние сборщика мусора на экземпляры простых типов. Так, кортежи (*tuple*) и словари (*dict*), содержащие только неизменяемые объекты (числа, строки, кортежи неизменяемых объектов и т.д.) не отслеживаются сборщиком мусора. [27]

2.2. Java

Java [29] — 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. [29]

2.2.1. Виртуальная машина Java

Виртуальная машина Java (JVM) является основой платформы Java. Это компонент технологии, отвечающий за её независимость от аппаратного обеспечения и операционной системы, минимальный размер скомпилированного исполняемого кода и безопасность. [30]

Виртуальная машина Java [30] — это абстрактная вычислительная машина. Как и реальная вычислительная машина, она имеет набор команд и управляет различными областями памяти во время выполнения.

Современные реализации Oracle JVM эмулируют работу виртуальной машины Java на мобильных, настольных и серверных устройствах, но виртуальная машина Java не предполагает какой-либо конкретной технологии реализации, аппаратного обеспечения или операционной системы компьютера. JVM по своей сути не является интерпретатором, но может быть реализована путем компиляции её набора команд в набор команд процессора. [30]

JVM ничего не знает о языке программирования Java и работает только с файлами классов в двоичном формате. Файл класса содержит инструкции JVM (байт-код) и вспомогательную информацию. [30] В целях безопасности виртуальная машина Java накладывает строгие синтаксические и структурные ограничения на код в файле класса. Однако любой язык, функциональность которого может быть выражена в виде допустимого файла класса, может размещаться на виртуальной машине Java. По этой причине разработчики могут использовать JVM в качестве машинно-независимой платформы для запуска приложений, разработанных на языках программирования, совместимых с JVM. [31]

Существует множество реализаций JVM. [32] [33] [34] Среди них наиболее распространённой является HotSpot JVM, представленная в 1999 году компанией Sun Microsystems. [35]

2.2.2. Управление памятью

Частью JVM является **менеджер хранилища** (storage manager), который отвечает за управление жизненным циклом объектов Java: выделение новых объектов, сбор недоступных объектов и отправка уведомлений о недоступности по запросу. Виртуальная машина HotSpot предоставляет несколько менеджеров хранилища для удовлетворения потребностей различных типов приложений, обеспечивая короткие паузы в работе приложения и высокую пропускную способность. [36]

Основной структурой данных HotSpot JVM является **иерархия классов** (klass hierarchy), которая описывает объекты в хранилище виртуальной машины и обеспечивает операции над этими объектами. Иерархия классов поддерживает объектно-ориентированную парадигму языка Java, предоставляя механизм для методов объектов (в том числе виртуальных). [36]

Из соображений масштабируемости, каждый поток JVM имеет собственную область выделения памяти — **буфер выделения потока** (Thread-Local

Allocation Buffers, TLAB). Каждый поток может выделять ресурсы из своей TLAB без координации с другими потоками, за исключением случаев, когда ему требуется новая TLAB. [36]

Было замечено, что большинство программ на языке Java следует гипотезе поколений (см. п. 1.3.3.5.). 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. [36]

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. [36]

Идентификация достижимости в молодом поколении осуществляется без просмотра всего графа объектов. Решение данной задачи основывается на гипотезе поколений, согласно которой имеется мало ссылок от старых объектов к молодым. JVM сохраняет «запоминаемый набор» (remembered set) ссылок от старого поколения к молодому и использует его для обхода объектов молодого поколения. [36]

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. Данная операция повышает накладные расходы на сборку мусора, поскольку необходимо перемещать объекты и обновлять все ссылки на объект, чтобы они указывали на новое местоположение объекта. С другой стороны, копирование объектов означает, что можно аккумулировать освобождённое пространство в один большой регион, из которого выделение происходит быстрее (что ускоряет очистку молодого поколения), а также появляется возможность вернуть избыточную память операционной системе. [36] [37]

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**. Дан-

ное название носит исторический характер и на самом деле объекты в нём не существуют на протяжении всего времени выполнения программы. Например, информация о загруженных классах хранится в постоянном поколении и уничтожается, когда эти классы больше не доступны из приложения. [36]

Начиная с Java 8 постоянное поколение было заменено **метапространством** (MetaSpace), предназначенным для хранения классов и метаданных JVM. [38]

2.2.3. Сборка мусора

HotSpot JVM предоставляет несколько сборщиков мусора, оптимизированных для различных типов приложений. Далее будут рассмотрены сборщики мусора, доступные в Java 21. [39]

2.2.3.1. Serial Collector

Serial Collector работает по алгоритму mark-compact (см. п. 1.3.3.3.) 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. [39]

Он лучше всего подходит для однопроцессорных вычислительных машин, поскольку не может использовать преимущества многопроцессорного оборудования, хотя может быть полезен на многопроцессорных системах для приложений с небольшими наборами данных (приблизительно до 100 МБ). [39]

Также Serial Collector используется в том случае, когда на одной машине работает большое количество JVM, которое может превышать количество доступных процессоров. В таких средах рекомендуется использовать только один процессор для сборки мусора, чтобы минимизировать влияние виртуальных машин друг на друга, жертвуя увеличением времени сборки мусора. [29]

2.2.3.2. Parallel Collector

Parallel Collector отличается от Serial Collector использованием нескольких потоков для сборки мусора, количество которых может регулироваться с помощью аргументов командной строки. [39] По умолчанию количество используемых потоков равно количеству доступных процессоров. [29] Во время работы Parallel Collector все потоки основной программы приостанавливаются

(«stop the world», см. п. 1.3.3.2.). [40]

На однопроцессорной машине используется сборщик мусора по умолчанию, даже если был выбран Parallel Collector. На машине с двумя процессорами он сопоставим по производительности со сборщиком мусора по умолчанию. На вычислительных машинах с большим количеством процессоров можно ожидать сокращения времени паузы на сборку мусора. [29]

Parallel Collector 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. [29] Рассматриваемый сборщик мусора следует использовать на многопроцессорном оборудовании, когда необходимо выполнить относительно большой объём работы и допустимы длинные паузы. Например, при пакетной обработке (batch processing), такой как печать документов или выполнение большого количества запросов к базе данных. [39]

2.2.3.3. Garbage-First Collector

Garbage-First Collector (также G1) — это сборщик мусора, работающий по алгоритму mark-sweep (см. п. 1.3.3.2.), 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. [41]

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. [41]

Garbage-First Collector достигает высокой производительности при снижении длительности пауз за счет применения следующих методов. [41]

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. 1.3.3.) и в конце при обнаружении изменений, произошедших за время сборки мусора. [40]

2. Куча разделяется на набор непрерывных областей одинакового размера, называемых регионами. Garbage-First Collector выполняет конкурентную фазу глобальной разметки для обнаружения используемых объектов в куче, после чего 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. Garbage-First Collector концентрирует свою деятельность по сбору и уплотнению на тех областях кучи, которые, предположительно, будут заполнены неиспользуемыми объектами.
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.

Важно отметить, что Garbage-First Collector не является сборщиком мусора в режиме реального времени. Вероятность соответствия заданному целевому времени паузы высока, но не равна единице. На основе данных о предыдущих циклах сборки мусора Garbage-First Collector оценивает, сколько регионов можно освободить в течение заданного пользователем целевого времени с помощью модели определения накладных расходов на обработку каждого региона. [41]

2.2.3.4. Z Garbage Collector

Z Garbage Collector (также ZGC) — это сборщик мусора, работающий по алгоритму mark-compact (см. п. 1.3.3.3.) и нацеленный на минимизацию пауз.

ZGC работает конкурентно с основной программой, не останавливая выполнение потоков приложения более чем на 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. [42]

Первоначально ZGC был представлен как экспериментальная опция в Java 11 и стал полноценной частью JVM в Java 15. [42] Начиная с Java 21 поддерживает опцию использования алгоритма поколений. [39]

Ниже представлены основные характеристики Z Garbage Collector. [42]

1. Сборка мусора выполняется конкурентно с основной программой.
2. Куча разделяется на регионы для дальнейшего уплотнения, как при работе Garbage-First Collector.
3. Используются **цветные указатели** (colored pointers): ZGC поддерживает только 64-битные указатели и хранит в них не только адрес объекта в памяти, но и дополнительные метаданные, определяющие текущий статус указателя:
 - Marked0 и Marked1 — фаза сборки мусора;
 - Remapped — адрес в указателе является окончательным и не должен модифицироваться в рамках текущего цикла сборки;
 - Finalizable — объект достижим только из финализатора.
4. Во время конкурентных фаз сборки мусора используются **барьеры** (barriers) — функции, которые принимают указатель на объект в памяти, анализируют его статус, в зависимости от него выполняют какие-либо действия с этим указателем или даже с объектом, на который он ссылается, после чего возвращают актуальное значение указателя, которое можно использовать для доступа к объекту.

Важной особенностью барьера является то, что он выполняется в том числе в потоках основной программы. То есть сборкой мусора занимаются не только выделенные потоки сборщика, но и само приложение.

2.2.3.5. Выбор сборщика мусора

Сборщик мусора для JVM следует выбирать для каждого приложения отдельно, поскольку производительность сбора мусора зависит от размера кучи, объема используемых данных, а также от количества и характеристик доступных процессоров. Однако можно сформировать некоторые рекомендации, которые послужат отправной точкой для выбора сборщика. [43]

1. Если приложение не имеет довольно строгих требований к времени паузы, сначала запустите приложение и позвольте виртуальной машине выбрать сборщик.
2. Если приложение работает с относительно небольшим набором данных (приблизительно до 100 Мб), следует выбрать Serial Collector.
3. Если приложение будет запускаться на одном процессоре и нет требований к времени паузы, следует выбрать Serial Collector.
4. Если пиковая производительность приложения является главным приоритетом и нет требований к времени паузы, то есть паузы длительностью более одной секунды приемлемы, тогда следует выбрать сборщик по умолчанию или Parallel Collector.
5. Если время ответа на запрос выделения памяти более важно, чем общая пропускная способность, а паузы для сбора мусора должны быть короче, следует выбрать Garbage-First Collector.
6. Если время ответа имеет высокий приоритет, следует выбрать Z Garbage Collector.

2.3. JavaScript

JavaScript [44] — интерпретируемый язык программирования со слабой динамической типизацией, представленный в 1995 году. Стандартом языка является ECMAScript. [45]

Наиболее широкое применение JavaScript получил в качестве языка сценариев веб-страниц, но также используется и в других программных продуктах, таких как node.js [46] и Apache CouchDB [47].

2.3.1. Цикл событий

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). Далее будет описана теоретическая модель цикла событий, которую реализуют современные движки JavaScript, при этом, как правило, оптимизируя её семантику. [48]

Вызов любой функции создаёт контекст выполнения (Execution Context). При вызове вложенной функции создаётся новый контекст, а старый сохраняется в специальной структуре данных — стеке вызовов (Call Stack). Объекты размещаются в куче. Также среда выполнения JavaScript содержит очередь задач, подлежащих обработке. Каждая задача ассоциируется с некоторым обработчиком (функцией). Обработка задачи заканчивается, когда стек снова становится пустым. Следующая задача извлекается из очереди и начинается её обработка. Каждая задача выполняется полностью, прежде чем начнёт выполняться следующая. Благодаря этому можно гарантировать, что выполнение функции не может быть приостановлено и будет завершено до начала выполнения другого кода (который может изменить данные, с которыми работает текущая функция). У данного подхода есть недостатки: если задача занимает слишком много времени, то веб-приложение не может обрабатывать действия пользователя в это время. В таких случаях современные браузеры выводят сообщение «скрипт выполняется слишком долго» («a script is taking too long to run») и предлагает остановить его. Хорошей практикой считается создание задач, которые исполняются относительно быстро и, если возможно, разбиение одной задачи на несколько подзадач. [48]

В браузерах события добавляются в очередь в любое время, если событие произошло, а так же если у него есть обработчик. В том случае, если у события нет обработчика, оно теряется. [48]

Для запуска скриптов в фоновом потоке предназначен **Web Worker** [49]. Поток Web Worker может выполнять задачи без вмешательства в пользовательский интерфейс, а также осуществлять ввод-вывод. Web Worker имеет свой собственный стек, кучу и очередь событий. Потоки событий можно связать друг с другом через отправку сообщений, добавляя их в соответствующие очереди. [48]

Поток выполнения цикла событий в JavaScript никогда не блокируется.

Выполнение ввода-вывода обычно осуществляется асинхронно с помощью событий и функций обратного вызова, поэтому, например, когда приложение ожидает ответа на запрос по сети, оно может обрабатывать другие задачи, например пользовательский ввод. [48]

2.3.2. Сборка мусора

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. 1.3.3.2.). 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. [50]

2.3.3. Вспомогательные структуры данных

Хотя JavaScript не предоставляет напрямую API сборщика мусора, язык предлагает несколько структур данных, которые косвенно наблюдают за сборкой мусора и могут использоваться для управления использованием памяти. [50]

WeakMap и **WeakSet** («слабые» аналоги Map и Set) позволяют поддерживать коллекцию пар ключ-значение и коллекцию уникальных значений соответственно. Эти структуры данных получили своё название от концепции слабо удерживаемых значений. Объект X слабо удерживается объектом Y, если имеется возможность получить доступ к значению X через Y, при этом алгоритм сборки мусора не будет считать X достижимым, если он не удерживается никакими другими объектами. Большинство структур данных, за исключением WeakMap, WeakSet и WeakRef, строго привязаны к вложенным объектам, поэтому их можно получить в любой момент. Ключи WeakMap и WeakSet могут быть очищены сборщиком мусора (для объектов WeakMap значения также будут подлежать сбору мусора), если ничто другое в программе не ссылается на

них. Это обеспечивается следующими характеристиками. [50]

- WeakMap и WeakSet могут хранить только объекты или символы, так как сборке мусора подлежат только они.
- WeakMap и WeakSet не являются итерируемыми (iterable).

Стоит отметить, что в программах на JavaScript допускается случай, когда значение ссылается на ключ. Так будет создана циклическая ссылка, которая сделает невозможным сбор мусора как для ключа, так и для значения, даже если ничто другое не ссылается на ключ. Чтобы исправить это, записи в WeakMap и WeakSet являются не реальными ссылками, а **эфемеронами** (ephemeros), что является усовершенствованием алгоритма mark-sweep. Эфемероны [51] — это уточнение слабых пар, в которых ни ключ, ни значение не могут быть классифицированы как слабые или сильные. Связность ключа определяет связность значения, но связность значения не влияет на связность ключа. Если сборка мусора предлагает поддержку эфемеронов, то она проходит в три этапа вместо двух (разметка и очистка). [50]

Все переменные со значением объекта являются ссылками на этот объект. Такие ссылки являются сильными, т. е. их существование не позволит сборщику мусора отметить объект как неиспользуемый. **WeakRef** — это слабая ссылка на объект, которая позволяет сборщику мусора освободить его, сохраняя при этом возможность чтения содержимого объекта в течение его жизни. Одним из вариантов использования WeakRef является система кеширования, которая сопоставляет строковые URL-адреса с объектами. [50]

FinalizationRegistry предоставляет ещё более надёжный механизм наблюдения за сборкой мусора, который позволяет регистрировать объекты и получать уведомления, когда они освобождаются сборщиком мусора. Однако из соображений производительности и безопасности нет никакой гарантии, когда будет вызван обработчик такого уведомления и будет ли он вызван вообще. Существуют и другие способы более детерминированного управления ресурсами. Например, синтаксическая конструкция **try...finally**, которая всегда выполняет блок **finally**. WeakRef и FinalizationRegistry существуют исключительно для оптимизации использования памяти в программах, которые предполагают выполнение в течение долгого времени. [50]

2.4. C#

C# [52] — 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. CLI является основой для создания сред выполнения и разработки приложений, в которых языки и библиотеки прозрачно работают друг с другом. [52]

При выполнении программы C# сборка загружается в среду CLR. Исходный код, написанный на языке C# компилируется в **промежуточный язык** (Intermediate Language, IL), который соответствует спецификациям CLI. Среда CLR выполняет JIT-компиляцию из кода на языке IL в набор машинных инструкций, ориентированных на конкретную платформу. Среда CLR также осуществляет автоматическую сборку мусора, обработку исключений и управление ресурсами. [52]

Обеспечение взаимодействия между языками является ключевой функцией .NET. Программа на языке IL, созданная из кода на C#, может взаимодействовать с программами, созданными платформой .NET из языков, соответствующих **спецификации общих типов** (Common Type System, CTS). К таким языкам относятся F#, Visual Basic, C++ и другие. [53]

2.4.1. Платформа .NET

.NET [53] — это кроссплатформенная среда выполнения приложений с открытым исходным кодом, предназначенная для разработки различных типов приложений, среди которых серверное программное обеспечение, мобильные приложения, игры, а также решения в области машинного обучения и интернета вещей. Ниже перечислены основные функции платформы .NET. [54]

- статический анализ кода;
- вывод типов данных для языков C#, F# и Visual Basic;

- обработка исключений;
- сборка мусора;
- поддержка асинхронного выполнения кода;
- обработка событий;
- поддержка технологии LINQ (Language-Integrated Query, интегрированный язык запросов);
- выполнение «небезопасного» кода. [55]

Основой всех приложений, работающих на платформе .NET, является среда CLR. Её основные функции: [54] [56]

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

Код, выполняемый средой CLR, также называют **«управляемым кодом»** (managed code). Он называется управляемым в первую очередь потому, что он использует сборщик мусора для управления памятью и обеспечивает безопасность памяти и типов. Среда CLR абстрагирует различные понятия операционной системы и аппаратного обеспечения, такие как память, потоки и исключения. [57]

2.4.2. Управление памятью

Сборщик мусора среды Common Language Runtime управляет выделением и освобождением памяти для приложения. Ниже перечислены основные концепции управления памятью в CLR. [58]

1. Каждый процесс имеет своё собственное виртуальное адресное пространство. Все процессы на одном компьютере используют одну и ту же физическую память и файл подкачки, если он есть.

2. По умолчанию на 32-разрядных компьютерах каждый процесс имеет виртуальное адресное пространство размером 2 ГБ.
3. Разработчики приложений работают только с виртуальным адресным пространством и никогда не управляют физической памятью напрямую.
4. Если при разработке приложения используются функции Windows для работы с виртуальным адресным пространством, то они выделяют и освобождают память в собственных кучах.
5. Виртуальная память может находиться в одном из следующих состояний.
 - **Свободна** (Free): область памяти не имеет ссылок на него и доступен для выделения.
 - **Зарезервирована** (Reserved): область памяти доступна для использования и не может быть использована для какого-либо другого запроса на выделение. Однако запрещается хранить данные в этой области памяти до тех пор, пока она не будет зафиксирована.
 - **Зафиксирована** (Committed): область памяти закреплена за физическим хранилищем.
6. Виртуальное адресное пространство может быть фрагментированным, а это означает, что в адресном пространстве существуют свободные блоки, называемые **дырами** (holes).
7. В случае нехватки памяти используется файл подкачки. Стоит отметить, что он используется, даже если нехватка памяти относительно низкая. В первый раз, когда она становится высокой, операционная система должна освободить память и выполнить резервное копирование данных из памяти в файл подкачки. Данные не выгружаются до тех пор, пока они не перестанут использоваться в программе.

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. [59]

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 избежание перемещения больших объектов в памяти куча больших объектов, как правило, не подлежит уплотнению. [60]

Время жизни большинства объектов, создаваемых в программе, управляется сборщиком мусора автоматически. Однако существуют так называемые **неуправляемые ресурсы**, требующие явного освобождения. Наиболее распространённым примером неуправляемого ресурса является объект, который управляет ресурсом операционной системы, таким как дескриптор файла или сетевое соединение. Хотя сборщик мусора может отслеживать существование управляемого объекта, инкапсулирующего неуправляемый ресурс, он не обладает конкретными знаниями о том, как его освободить. При создании объекта, инкапсулирующего неуправляемый ресурс, рекомендуется предоставить необходимый код для очистки неуправляемого ресурса. Это позволяет явно освобождать память, занимаемую объектом, после его использования. [58]

2.4.3. 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. 1.3.3.3.), разделяя сбор мусора на три фазы. [58]

1. **Фаза разметки:** определение неиспользуемых объектов путём исследования корневого набора программы (см. п. 1.3.3.), 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.

Стоит отметить, что уплотнение выполняется только в том случае, если сборщик обнаруживает относительно большое количество недоступных объектов. Если все объекты в управляемой куче являются доступными, уплотнение не требуется. [58]

Сбор мусора гарантированно происходит при выполнении одного из следующих условий. [58]

1. В системе обнаружена нехватка физической памяти. Данная ситуация определяется либо по уведомлению от операционной системы, либо по приближении к пределу объема физической памяти, доступной на вычислительной машине.
2. Объем 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. [58]

- 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. [58] [59]

- **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).

Сбор мусора в каждом поколении происходит при выполнении определённых условий. Под сборкой мусора в поколении понимается сбор объектов

этого поколения и всех более младших поколений. Поэтому сборку мусора в поколении 2 также называют **полной сборкой мусора**. [58]

Поскольку объекты поколений 0 и 1 имеют относительно небольшое время жизни, эти поколения называются **эфемерными**. Выделение памяти для них осуществляется в сегменте памяти, известном как **эфемерный сегмент**. Каждый новый сегмент, полученный сборщиком мусора, становится новым эфемерным сегментом и содержит объекты, оставшиеся после сборки мусора в поколении 0. Старый эфемерный сегмент становится сегментом нового поколения 2. Размер эфемерного сегмента зависит разрядности операционной системы, а также от конфигурации сборщика мусора. [58]

Пользователь может указать тип сборки мусора в зависимости от характеристик выполняемой программы. CLR предоставляет следующие типы сборки мусора. [61]

- **Сборка мусора рабочей станции** (Workstation garbage collection), предназначенная для клиентских приложений и установленная по умолчанию для автономных приложений (standalone). Сбор мусора рабочей станции может быть параллельной или непараллельной. Параллельная (или **фоновая**) сборка мусора позволяет потокам программы продолжать выполнение во время сборки мусора. Сбор происходит в пользовательском потоке, который инициировал сборку мусора, конкурентно с другими пользовательскими потоками. Если вычислительная машина, выполняющая программу, является однопроцессорной, то на ней всегда выполняется сборка мусора рабочей станции, независимо от заданной пользователем конфигурации.
- **Серверная сборка мусора** (Server garbage collection), предназначенная для серверных приложений, которым требуется высокая пропускная способность и масштабируемость. Сбор происходит в нескольких выделенных потоках. В ОС Windows эти потоки выполняются на более высоком уровне приоритета, чем пользовательские потоки, что позволяет выполнить серверную сборку мусора быстрее, чем сборку мусора рабочей станции, при одинаковом размере кучи. Для каждого логического процессора вычислительной машины предусмотрена куча и выделенный поток для выполнения сборки мусора, причём кучи собираются одновременно. Объекты в разных кучах могут ссылаться друг на друга.

Если предполагается использование большого количества экземпляров приложения на одной машине, рекомендуется использовать сборку мусора рабочей станции в непараллельном режиме. Это приведёт к снижению числа переключений между потоками и может повысить производительность. [61]

Существуют ситуации, в которых полная сборка мусора средой CLR может отрицательно повлиять на производительность. Это может быть проблемой, особенно для серверов, обрабатывающих большие объемы запросов. В этом случае длительная сборка мусора может привести к истечению времени ожидания запроса. Чтобы предотвратить полную сборку мусора в критический период, приложение может получить уведомление о приближении полной сборки мусора, чтобы перенаправить рабочую нагрузку на другой экземпляр сервера. Также сбор мусора можно вызвать явно в тот момент, когда экземпляр сервера не обрабатывает запросы. [62]

2.5. 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) [63], то есть код, который выполняется без ведома программиста. Среда выполнения состоит из следующих компонентов.

1. **Планировщик** (Scheduler) [64], отвечающий за управление работой горутин. **Горутина** — это легковесный поток, управляемый средой выполнения языка Golang. [65]
2. **Аллокатор памяти.**
3. **Сборщик мусора.**

2.5.1. Выделение памяти

Первоначально аллокатор памяти языка Golang был основан на **tcmalloc** [66], но со временем стал сильно отличаться от него. [67]

Аллокатор работает с наборами страниц (runs of pages). Если размер выделяемого объекта не превышает 32 Кб, то он округляется до одного из 67 классов

размеров (size classes) [68], каждый из которых имеет свой собственный набор свободных объектов соответствующего размера. Любая свободная страница памяти может быть разделена на набор объектов одного класса размера, которыми затем можно управлять с помощью битовой карты свободных областей (free bitmap). [67]

Ниже перечислены основные структуры данных, с которыми работает аллокатор памяти в языке Golang. [67]

1. **fixalloc** [69] — аллокатор для объектов фиксированного размера, размещаемых за пределами кучи. Используется для управления хранилищем, которое использует аллокатор.
2. **mheap** — куча, поделённая на страницы размером 8 Кб.
3. **mspan** — набор используемых страниц, управляемых mheap.
4. **mcentral** — хранилище всех mspan заданного класса размера.
5. **mcache** — кеш для каждого ядра процессора, содержащий набор mspan со свободным пространством.
6. **mstats** — статистика распределения памяти.

На рисунке 2.4 показана схема алгоритма выделения памяти.

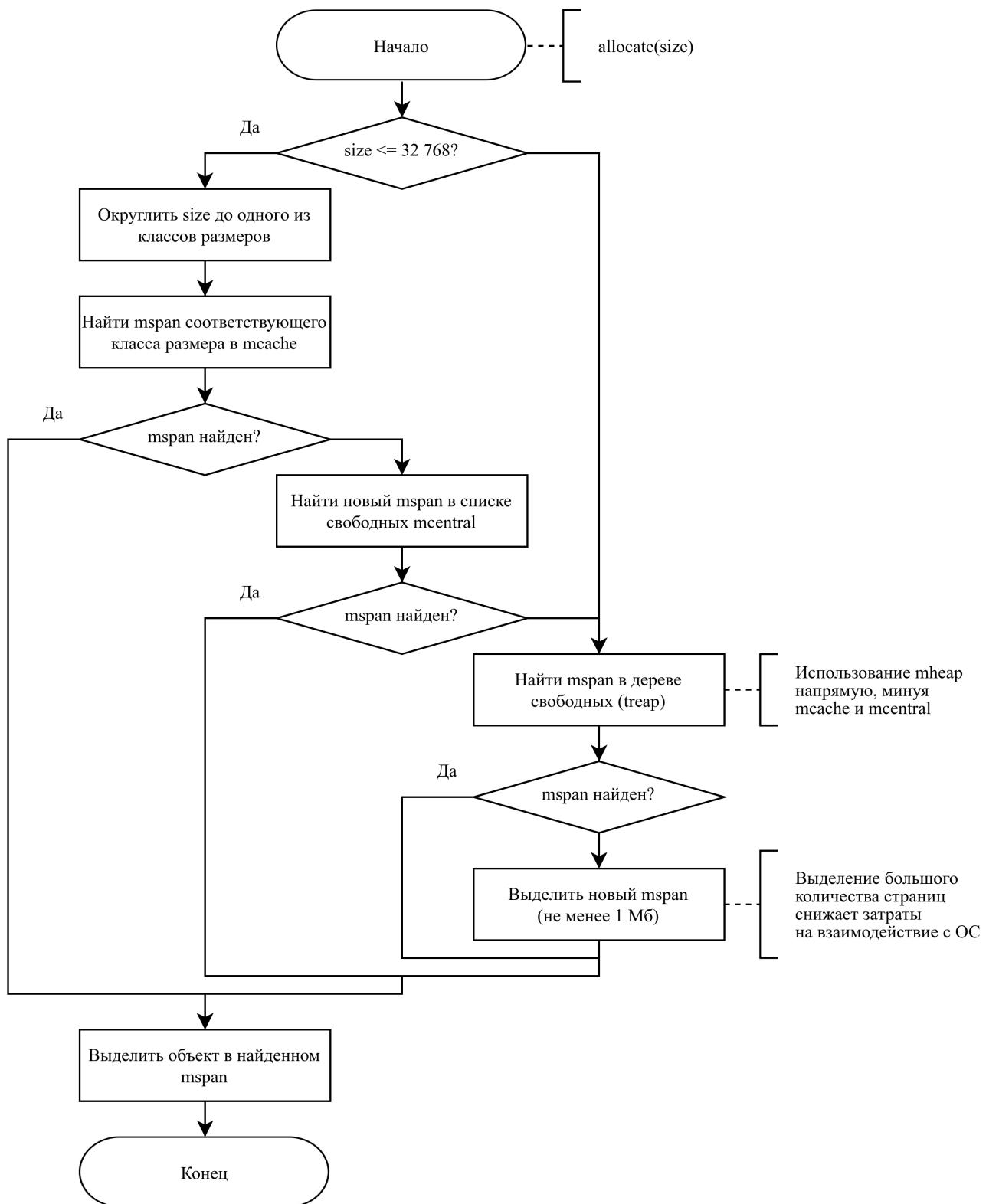


Рис. 2.4 – Выделение памяти в куче

На рисунке 2.5 показана схема алгоритма освобождения памяти.

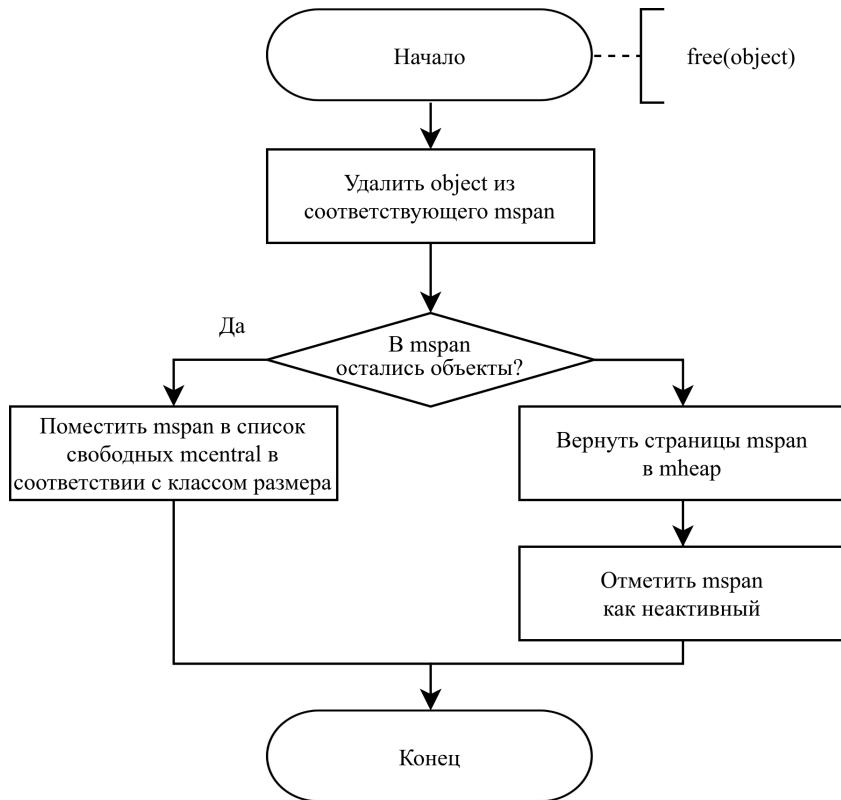


Рис. 2.5 – Освобождение памяти в куче

2.5.2. 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. [67]

С каждой арендой связан объект heapArena [70], 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. [67]

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. [67]

Концептуальная схема, отражающая структуру кучи в программе на языке Golang, представлена на рисунке 2.6.

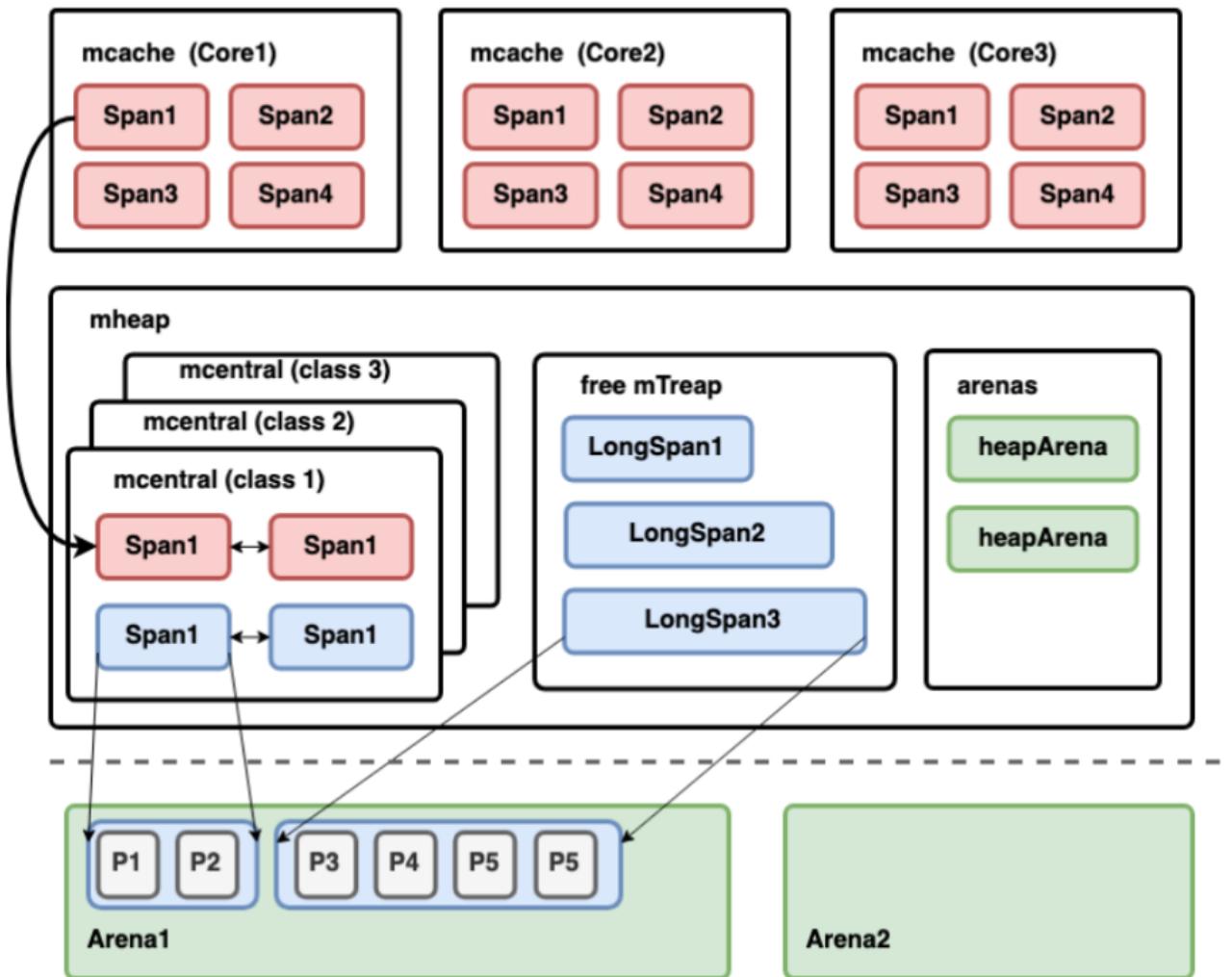


Рис. 2.6 – Структура кучи в Golang

2.5.3. Сборка мусора

В языке Golang для сборки мусора используется алгоритм **Concurrent Mark-Sweep**, являющийся модификацией алгоритма mark-sweep (см. п. 1.3.3.2.), предназначенного для сборки мусора конкурентно с основной программой. [71]

Сборщик мусора запускается параллельно с потоками основной программы, обладает точностью типов (type accurate, type precise), позволяет нескольким потокам сборки мусора выполняться параллельно. Алгоритм сборки мусора не является уплотняющим и не использует поколения (см. п. 1.3.3.3. и 1.3.3.5.). [71]

Для конкурентной разметки и очистки используются **барьеры записи** (write barrier), позволяющий избежать ситуации, когда чёрные объекты указы-

вают на белые. Такое может произойти, например, при перемещении указателя в программе до того, как сборщик мусора успел его разметить. В такой ситуации основная программа скрывает объект от сборщика мусора. [71] 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 [72].

Ниже представлено описание основных шагов алгоритма сборки мусора. [71]

1. Фаза завершения очистки (sweep termination).
 - 1.1. «Остановка мира» («stop the world», см. п. 1.3.3.2.), 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. 1.3.3.) и включение ассистентов (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 затенение 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 `mspan` перед использованием.
 - 4.3. Выполнение параллельной очистки в фоновом режиме и во время выделения памяти.

Чтобы предотвратить длительные паузы при сканировании больших объектов и улучшить параллелизм, сборщик мусора разбивает задания сканирования объектов размером более `maxObletBytes` на «облеты», размер которых не превышает `maxObletBytes`. Когда при сканировании обнаруживается начало большого объекта, сборщик сканирует только первый фрагмент, а остальные фрагменты помещает в очередь как новые задания сканирования. [71]

2.5.4. Настройка сборщика мусора

Следующая сборка мусора выполняется после того, как был выделен дополнительный объём памяти, пропорциональный тому который был зафиксирован при предыдущем запуске сборки мусора. Данная пропорция контролируется переменной среды **GOGC** (по умолчанию имеет значение 100). Если

GOGC=100 и программа использует 4 Мб памяти, следующий цикл сборки мусора запустится 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. Настройка GOGC просто изменяет линейную константу. [71]

По сути GOGC определяет компромисс между накладными расходами памяти и процессорного времени. После каждого цикла сборки мусора определяется целевой размер кучи, а также целевое значение общего размера кучи для запуска следующего цикла. Целевой размер кучи определяется следующим образом. [73]

$$\text{Target heap} = \text{Live heap} + (\text{Live heap} + \text{GC roots}) * \text{GOGC}/100 \quad (2.1)$$

Стоит заметить, что данная настройка не учитывает то, что доступная программе память ограничена. В версии Golang 1.19 было добавлено ограничение, которое устанавливает максимальный общий объем памяти (**GOMEMLIMIT**), который может использовать среда выполнения Golang. [74] [75]

Поскольку сборщик мусора имеет явный контроль над тем, сколько памяти кучи используется программой, он устанавливает общий размер кучи на основе этого ограничения памяти и того, сколько другой памяти использует среда выполнения Golang. Когда размер используемой памяти достигает пикового значения, определенного GOGC, сборщик мусора запускается чаще, чтобы поддерживать его в пределах лимита. [73]

Стоить заметить, что даже когда GOGC отключен, ограничение памяти всё равно соблюдается. Это позволяет достичь максимальной экономии ресурсов, поскольку устанавливается минимальная частота запуска сборщика мусора, необходимая для поддержания ограничения памяти.

Несмотря на то, что ограничение памяти, очевидно, является мощным инструментом, его использование не обходится без накладных расходов и, конечно же, не отменяет полезность GOGC. [73]

Такая ситуация, когда программа не может выполняться из-за постоянных циклов сборки мусора, называется **перегрузкой**. Во многих случаях неопределённая остановка хуже, чем нехватка памяти, которая обычно приводит к гораздо более быстрому сбою. По этой причине ограничение памяти в Golang

определяется как **мягкое** (soft memory limit) [75]. Среда выполнения Go не дает никаких гарантий, что она сохранит заданный лимит памяти при любых обстоятельствах, а только обещает некоторое разумное количество усилий. Это ослабление ограничения памяти имеет решающее значение для предотвращения зависаний, поскольку оно позволяет аллокатору превысить предел, чтобы не тратить слишком много времени на сборку мусора. [73]

В языке Golang мягкое ограничение памяти реализовано следующим образом: сборщик мусора устанавливает верхний предел количества процессорного времени, которое он может использовать в течение некоторого временного промежутка (окна, frame). В настоящее время этот предел установлен примерно на уровне 50%. В случае неправильной настройки лимита памяти, когда по ошибке он был установлен слишком низким, программа замедлится максимум в 2 раза, поскольку GC не может отнять у неё более 50% процессорного времени. [73]

2.5.5. Арены 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.

Пакет arena предоставляет возможность выделения памяти в программах на языке Golang и освобождать её вручную, причем безопасно и одновременно. Цель этой функциональности — повышение эффективности программ: освобождение памяти вручную откладывает следующий цикл сборки мусора. В свою очередь, снижение частоты циклов означает уменьшение накладных расходов на сборку мусора. [76]

Большая часть описанной функциональности рассматриваемого пакета реализована в типе Arena, который является абстракцией относительно большого объема памяти, выделенного в куче. Наиболее эффективное использование Arena достигается при хранении в них некоторого множества объектов программы, занимающих около 1 Мб памяти. [76] [77]

Стоит заметить, что при использовании этой ограниченной формы ручного выделения памяти возможны ошибки типа «use-after-free» (использование после освобождения). Пакет Arena ограничивает влияние этих ошибок, предот-

вращая повторное использование освобождённых областей памяти до тех пор, пока сборщик мусора не сможет убедиться в том, что это безопасно. Как правило, ошибка «use-after-free» приводит к сбою и получению сообщения об ошибке, но пакет Arena оставляет за собой право не вызывать сбой программы при использовании освобождённой памяти. Это означает, что реализация данного пакета допускает выделение памяти так, как это обычно делает среда выполнения, оставляя за собой право иногда делать это для некоторых объектов программы. [76]

Повышение производительности программы, использующей выделение памяти на аренах, можно ожидать в тех случаях, когда приложение интенсивно выделяет память (например, при работе с ссылочными структурами данных, такими как двоичные деревья), но при этом предполагается, что выделенные структуры данных являются относительно долгоживущими и существуют до момента освобождения арены целиком (сборщик мусора для арены не применяется и выделенные на ней объекты не освобождаются автоматически).

3 Классификация существующих решений

3.1. Критерии сравнения алгоритмов распределения памяти

Анализ алгоритмов распределения памяти является задачей классификации. Ниже приведены критерии для их сравнения.

1. Разделение объектов на поколения.

Для оптимизации сборки мусора и уменьшения времени пауз в языках программирования может использоваться алгоритм поколений (см. п. 1.3.3.5.). В таблице сравнения будут использоваться обозначения «+» и «-» для определения использования алгоритма поколений менеджером памяти 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 основной программы. 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 основной программы 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 основной программы за один цикл сборки мусора.

Описанные выше характеристики алгоритмов распределения памяти, как правило, декларируются в документациях к соответствующим языкам программирования.

3.2. Сравнительный анализ алгоритмов распределения памяти

Результаты сравнения алгоритмов распределения памяти приведены в таблице 3.1. Для краткости записи в данной таблице используются следующие обозначения описанных критериев:

- К1 — разделение объектов на поколения;
- К2 — отсутствие хранения вспомогательных данных в объектах;
- К3 — использование конкурентной сборки мусора;
- К4 — использование параллельной сборки мусора;
- К5 — отсутствие остановки потоков основной программы на весь цикл сборки мусора;
- К6 — количество остановок потоков основной программы за один цикл сборки мусора.

Таблица 3.1 – Сравнение алгоритмов распределения памяти

Язык программирования	Сборщик мусора	K1	K2	K3	K4	K5	K6
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

3.3. Вывод

Среди рассмотренных интерпретируемых языков программирования наиболее 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.

ЗАКЛЮЧЕНИЕ

В рамках научно-исследовательской работы была проведена классификация алгоритмов распределения памяти.

С течением времени языки программирования интегрируются со средами выполнения, которые предоставляют им низкоуровневые средства для реализации всё более производительных алгоритмов автоматического управления памятью.

В результате сравнения были выделены менеджеры памяти языков программирования Python и Java как наиболее масштабируемые и оптимизированные для различных сценариев использования.

В ходе выполнения данной работы были решены следующие задачи.

1. Проанализирована предметная область работы с памятью в языках программирования с автоматической сборкой мусора.
2. Рассмотрены существующие принципы организации работы с памятью в языках программирования с автоматической сборкой мусора на примере Python, Java, JavaScript, C# и Golang.
3. Описаны алгоритмы сборки мусора в рассмотренных языках.
4. Сформулированы критерии сравнения и оценки описанных алгоритмов.
5. Проведено сравнение существующих решений по сформулированным критериям.

Таким образом, все поставленные задачи были выполнены, поставленная цель достигнута.

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 [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/> (дата обращения: 29.10.2023).
5. The Elixir programming language [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. Внутреннее устройство Windows / М. Руссинович [и др.]. — 7-е изд. — Питер, 2018.
9. Bovet D. P., Cesati M. Understanding the Linux kernel. — 3-е изд. — O'Reilly, 2005.
10. Common Language Runtime (CLR) overview [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/standard/clr> (дата обращения: 12.10.2023).
11. 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.
12. Overview [Электронный ресурс]. — Режим доступа, URL: <https://www.memorymanagement.org/mmref/begin.html> (дата обращения: 15.10.2023).

13. Allocation techniques [Электронный ресурс]. — Режим доступа, URL: <https://www.memorymanagement.org/mmref/alloc.html#mmref-alloc> (дата обращения: 12.10.2023).
14. Recycling techniques [Электронный ресурс]. — Режим доступа, URL: <https://www.memorymanagement.org/mmref/recycle.html> (дата обращения: 15.10.2023).
15. *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).
16. *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.
17. *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.
18. Copying Garbage Collection [Электронный ресурс]. — Режим доступа, URL: <https://www.cs.cornell.edu/courses/cs312/2003fa/lectures/sec24.htm> (дата обращения: 21.10.2023).
19. The official home of the Python Programming Language [Электронный ресурс]. — Режим доступа, URL: <https://www.python.org/> (дата обращения: 10.11.2023).
20. Jython Home [Электронный ресурс]. — Режим доступа, URL: <https://www.jython.org/> (дата обращения: 10.11.2023).
21. IronPython: the Python programming language for .NET [Электронный ресурс]. — Режим доступа, URL: <https://ironpython.net/> (дата обращения: 10.11.2023).
22. PyPy [Электронный ресурс]. — Режим доступа, URL: <https://www.pyton.org/> (дата обращения: 10.11.2023).
23. CPython [Электронный ресурс]. — Режим доступа, URL: <https://docs.python.org/3/glossary.html#term-CPython> (дата обращения: 10.11.2023).

24. Extending Python with C or C++ [Электронный ресурс]. — Режим доступа, URL: <https://www.python.org/> (дата обращения: 10.11.2023).
25. GIL [Электронный ресурс]. — Режим доступа, URL: <https://docs.python.org/3/glossary.html#term-GIL> (дата обращения: 10.11.2023).
26. 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).
27. Garbage collector design [Электронный ресурс]. — Режим доступа, URL: <https://devguide.python.org/internals/garbage-collector/> (дата обращения: 10.11.2023).
28. PEP 556 — Threaded garbage collection [Электронный ресурс]. — Режим доступа, URL: <https://peps.python.org/pep-0556/> (дата обращения: 26.11.2023).
29. 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).
30. The Java Virtual Machine Specification. Java SE 21 Edition. Chapter 1. Introduction [Электронный ресурс]. — Режим доступа, URL: <https://docs.oracle.com/javase/specs/jvms/se21/html/jvms-1.html> (дата обращения: 16.11.2023).
31. Urma R.-G. Alternative Languages for the JVM [Электронный ресурс]. — Режим доступа, URL: <https://www.oracle.com/technical-resources/articles/java/architect-languages.html> (дата обращения: 16.11.2023).
32. OpenJ9 [Электронный ресурс]. — Режим доступа, URL: <https://eclipse.dev/openj9/> (дата обращения: 16.11.2023).
33. Codename One: Cross-Platform App Development with Java [Электронный ресурс]. — Режим доступа, URL: <https://www.codenameone.com/> (дата обращения: 16.11.2023).
34. GraalVM [Электронный ресурс]. — Режим доступа, URL: <https://www.graalvm.org/> (дата обращения: 16.11.2023).
35. The HotSpot Group [Электронный ресурс]. — Режим доступа, URL: <https://openjdk.org/groups/hotspot/> (дата обращения: 16.11.2023).

36. HotSpot VM Storage Management [Электронный ресурс]. — Режим доступа, URL: <https://openjdk.org/groups/hotspot/docs/StorageManagement.html> (дата обращения: 16.11.2023).
37. 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).
38. 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).
39. 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).
40. 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).
41. 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).
42. The Z Garbage Collector (ZGC) [Электронный ресурс]. — Режим доступа, URL: <https://wiki.openjdk.org/display/zgc/Main> (дата обращения: 16.11.2023).
43. Java SE 11. HotSpot Virtual Machine Garbage Collection Tuning Guide. Available Collectors [Электронный ресурс]. — Режим доступа, URL: <https://docs.oracle.com/en/java/javase/11/gctuning/available-collectors.html> (дата обращения: 16.11.2023).
44. JavaScript [Электронный ресурс]. — Режим доступа, URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (дата обращения: 12.11.2023).

45. ECMAScript 2023 language specification [Электронный ресурс]. — Режим доступа, URL: <https://ecma-international.org/publications-and-standards/standards/ecma-262/> (дата обращения: 12.11.2023).
46. Node.js [Электронный ресурс]. — Режим доступа, URL: <https://nodejs.org> (дата обращения: 12.11.2023).
47. Apache CouchDB [Электронный ресурс]. — Режим доступа, URL: <https://couchdb.apache.org/> (дата обращения: 12.11.2023).
48. The event loop [Электронный ресурс]. — Режим доступа, URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Event_loop (дата обращения: 12.11.2023).
49. Using Web Workers [Электронный ресурс]. — Режим доступа, URL: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers (дата обращения: 12.11.2023).
50. Memory management [Электронный ресурс]. — Режим доступа, URL: <https://developer.mozilla.org/en-US/docs/Web/JavaScript> (дата обращения: 12.11.2023).
51. Barros A., Jerusalimschy R. Eliminating Cycles in Weak Tables // Journal of Universal Computer Science. — 2008. — Т. 14, № 21.
52. 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).
53. NET | Build. Test. Deploy [Электронный ресурс]. — Режим доступа, URL: <https://dotnet.microsoft.com/en-us/> (дата обращения: 17.11.2023).
54. 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).
55. Unsafe code, pointer types, and function pointers [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/csharp/language-reference/unsafe-code> (дата обращения: 17.11.2023).
56. 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).

57. What is «managed code»? [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/standard/managed-code> (дата обращения: 17.11.2023).
58. 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).
59. Automatic Memory Management [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/standard/automatic-memory-management> (дата обращения: 17.11.2023).
60. 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).
61. Workstation and server garbage collection [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/workstation-server-gc> (дата обращения: 17.11.2023).
62. Garbage Collection Notifications [Электронный ресурс]. — Режим доступа, URL: <https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/notifications> (дата обращения: 17.11.2023).
63. runtime package [Электронный ресурс]. — Режим доступа, URL: <https://pkg.go.dev/runtime> (дата обращения: 4.11.2023).
64. Source file src/runtime/proc.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/proc.go> (дата обращения: 4.11.2023).
65. A Tour of Go. Goroutines [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/tour/concurrency/1> (дата обращения: 4.11.2023).
66. TCMalloc : Thread-Caching Malloc [Электронный ресурс]. — Режим доступа, URL: <https://goog-perftools.sourceforge.net/doc/tcmalloc.html> (дата обращения: 4.11.2023).
67. Source file src/runtime/malloc.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/malloc.go> (дата обращения: 4.11.2023).
68. Source file src/runtime/sizeclasses.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/sizeclasses.go> (дата обращения: 4.11.2023).

69. Source file src/runtime/mfixalloc.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/mfixalloc.go> (дата обращения: 4.11.2023).
70. Source file src/runtime/mheap.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/mheap.go> (дата обращения: 4.11.2023).
71. Source file src/runtime/mgc.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/mgc.go> (дата обращения: 5.11.2023).
72. Source file src/runtime/mbarrier.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/runtime/mbarrier.go> (дата обращения: 4.11.2023).
73. A Guide to the Go Garbage Collector [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/doc/gc-guide> (дата обращения: 5.11.2023).
74. Go 1.19 Release Notes [Электронный ресурс]. — Режим доступа, URL: <https://tip.golang.org/doc/go1.19> (дата обращения: 5.11.2023).
75. Proposal: Soft memory limit [Электронный ресурс]. — Режим доступа, URL: <https://github.com/golang/proposal/blob/master/design/48409-soft-memory-limit.md> (дата обращения: 5.11.2023).
76. Source file src/arena/arena.go [Электронный ресурс]. — Режим доступа, URL: <https://go.dev/src/arena/arena.go> (дата обращения: 4.11.2023).
77. Proposal: arena: new package providing memory arenas [Электронный ресурс]. — Режим доступа, URL: <https://github.com/golang/go/issues/51317> (дата обращения: 4.11.2023).

ПРИЛОЖЕНИЕ А

Презентация.