# РЕФЕРАТ

Расчетно–пояснительная записка 35 с., 1 рис., 0 табл., 14 ист, 1 прил.

## СОДЕРЖАНИЕ

# 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. 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.

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. Автоматическое управление памятью избавляет программиста от необходимости вручную освобождать выделенную память, устраняя 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 может накладывать дополнительные расходы по памяти и времени исполнения. На сегодняшний день среды времени выполнения (language runtime) многих популярных языков программирования, таких как Java, C#, Python и другие, активно используют сборку 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 C#, Python, Java, Golang и Haskell.
3. Описать алгоритмы сборки мусора в рассматриваемых языках.
4. Сформулировать критерии сравнения и оценки рассмотренных алгоритмов.
5. Сравнить существующие решения по выделенным критериям.

# 1 Анализ предметной области

## 1.1. Основные понятия

**Управление памятью** [2] — 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, защита).
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. [2]

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 несколькими 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 программы затрачивают на выделение и освобождение памяти. [3]

### 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 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** [4] — область адресного пространства процесса, предназначенная для хранения параметров функций, локальных переменных и адреса возврата после вызова функции. **Куча** [5] — 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, 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 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**. [4]

### 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 тоже 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: [6]

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 параллельного императивного языка программирования определяет, какие записи в разделяемые переменные могут быть видны при чтениях, выполняемых другими потоками. [7]

### 1.3.2. Управление памятью

Менеджер памяти приложения должен учитывать следующие ограничения. [8]

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** [2] — явление, при котором аллокатор выделяет при каждом запросе больше памяти, чем фактически запрошено. **Внешняя фрагментация** [2] — явление, при котором свободная память разделена на множество мелких блоков, ни один из которых нельзя использовать для обслуживания запроса ан выделение памяти.

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

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. [8]

1. **Преждевременное освобождение памяти** (premature free). Когда программы освобождают память, но позже пытаются получить к ней доступ, их поведение становится неопределённым. Сохранившаяся ссылка на освобождённую память называется **висячим указателем** (dangling pointer).

2. **Утечки памяти** (memory leaks). Когда программы интенсивно выделяют память, не освобождая её, в конечном итоге память заканчивается.

3. **Внешняя фрагментация** (external fragmentation). Работа аллокатора по выделению и освобождению памяти может привести к тому, что он больше не сможет выделять достаточно большие области памяти, несмотря на наличие достаточного общего количества свободной памяти. Это связано с тем, что свободная память может быть разделена на множество небольших областей, разделённых используемыми данными. Данная проблема особенно критична в серверных приложениях, работающих в течение длительного времени.

4. **Неэффективное расположение ссылок** (poor locality of reference). Эта проблема связана с тем, как современные менеджеры памяти аппаратного обеспечения и операционной системы обращаются с памятью: последовательные обращения к памяти выполняются быстрее, если они производятся к соседним областям памяти. Если менеджер памяти разместит данные, которые программа будет использовать вместе, далеко друг от друга, 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** [9]. 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, это 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 дает возможность явно указать, что заданная область памяти может быть освобождена и переиспользована. [8]

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) [2], 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. [9]

#### 1.3.2.2. Автоматическое управление памятью

Автоматическое управление памятью [8] — 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. [2]

Задачи автоматического управления памятью:

— выделение памяти под новые объекты;

— идентификация используемых объектов;

— освобождение памяти, занятой неиспользуемыми объектами.

Преимущества автоматического управления памятью:

— разработчик освобождается 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. [10] Стоит отметить, что для автоматического управления памятью также можно использовать гибридные методы. [11] [12]

### 1.3.3. Трассирующая сборка мусора

**Сборщик мусора** [2] — автоматический менеджер памяти, реализующий некоторый алгоритм сборки мусора.

**Трассирующий сборщик мусора** [10] — 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 ка заполненность и фрагментация. [13]

#### 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 — полезная характеристика трассирующих сборщиков мусора, которая позволяет судить о корректности сборки в терминах инвариантов, которые сборщик должен сохранять. В рамках трехцветной абстракции коллекция трассировки разбивает граф объектов на черные (предположительно используемые) и белые (возможно, неиспользуемые) объекты. Изначально каждый узел 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 отслеживает 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. [13]

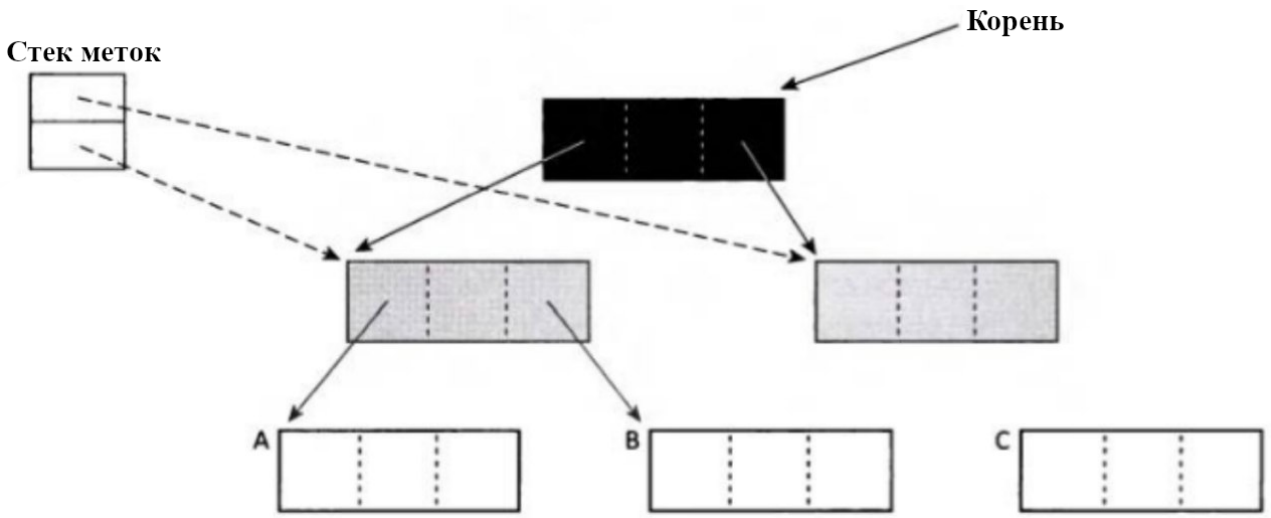


Рис. 1.1 – Пример трёхцветной разметки

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 снова, и поэтому они будут серыми. 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 A, B и C). Однако, как только разметка графа завершится, стек меток будет пуст (без серых объектов) и только C останется белым (мусор), а все остальные объекты будут окрашены чёрным.

Алгоритм сохраняет важный инвариант: в конце каждой итерации цикла разметки отсутствуют ссылки с черных объектов на белые. Таким образом, любой доступный белый объект должен быть доступен из серого объекта. Если бы этот инвариант был нарушен, то используемый потомок черного объекта мог быть окрашен белым (и, следовательно, был бы освобожден некорректно), поскольку сборщик не обрабатывает черные объекты дальше. Трехцветное представление состояния объектов программы особенно полезно при рассмотрении алгоритмов параллельной сборки мусора, когда сборка мусора выполняется параллельно с потоками основной программы. [13]

### 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. [13]

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. [13]

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 конкуррентно с основной программой. Такой режим работы сборщика называют «остановкой мира» (stop-the-world). [13]

**СХЕМЫ АЛГОРИТМОВ**

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 ее 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%. [13]

Как правило, в длительно работающих приложениях куча имеет тенденцию к фрагментации. Для неподвижных распределителей памяти требуется пространство, в O(log(max/min)) большее минимально возможного, где min и max — наименьший и наибольший возможный размеры объекта. Таким образом, возможно, придется вызвать неуплотняющий сборщик мусора чаще, чем

тот, который уплотняет используемые объекты. [13]

Чтобы избежать снижения производительности из-за чрезмерной фрагментации, многие рабочие 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. [13]

#### 1.3.3.3. Алгоритм mark-compact

Фрагментация может быть проблемой для неподвижных коллекторов. 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 блоках. Это особенно актуально для приложений, которые не выделяют много относительно больших объектов примерно одинакового размера. [13]

Однако многие длительно работающие приложения, управляемые неперемещающими 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 одного региона в другой. 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. [13]

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. [13]

— 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 строкам кэша или страницам виртуальной памяти. Все современные реализации алгоритма mark-compact реализуют скользящее перемещение, которое не влияет на локальность объектов путем изменения относительного порядка их размещения. [13]

**СХЕМЫ АЛГОРИТМОВ**

### 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. [13]

Как правило, сборщики копирования делят кучу на два полупространства равного размера, называемых **старым** и **новым пространством**. 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 устраняет «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 роли старого и нового пространства поменяются местами. [13] [14]

**СХЕМЫ АЛГОРИТМОВ**

**ГРАФИК**

### 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. [14]

**Гипотеза поколений** (generational hypothesis, infant mortality) [2] — 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 как функция возраста (времени жизни) объекта убывает гиперэкспоненциально.

**Сборка мусора по алгоритму поколений** [2] — это трассирующая сборка мусора, в которой используется гипотеза поколений. Объекты собираются вместе в поколениях. Новые объекты выделяются в самом молодом или детском поколении и передаются в старшие поколения, если они выживают. Объекты более старых поколений осуждаются реже, что экономит время процессора.

Как правило, объект редко ссылается на более молодой объект. Следовательно, объекты одного поколения обычно имеют мало ссылок на объекты более молодых поколений. Это означает, что сканирование старых поколений в процессе сбора более молодых поколений можно осуществлять более эффективно, отдельно запоминая ссылки на объекты более молодых поколений. [14]

Использование гипотезы поколений позволяет модифицировать копирующий алгоритм сборки мусора. Так, первоначально объекты со схожим временем жизни, называемые **поколением**, 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. [14]

### 1.3.4. Подсчёт ссылок

Подсчёт ссылок предполагает отслеживание количества указателей на определенные области памяти из других областей. Он используется в качестве основы для некоторых методов автоматической переработки, которые не требуют отслеживания (трассировки). [10]

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. [2]

Преимущества подсчёта ссылок:

— накладные расходы по управлению памятью распределяются по всему времени работы программы; [13]

— устойчивость к высоким нагрузкам, так как потенциально подсчёт ссылок может переработать объект как только он перестаёт использоваться; [13]

— может быть реализован без помощи среды выполнения языка и без её ведома. [13]

Недостатки подсчёта ссылок:

— требуются накладные расходы на операции чтения и записи для управления числом ссылок;

— операции с числом ссылок на объекты должны быть атомарными; [13]

— случай циклических ссылок требует отдельного рассмотрения; [13]

— подсчёт ссылок может вызывать паузы, например при освобождении ссылочных структур данных. [13]

По сравнению со сборкой мусора алгоритмы управления памятью, основанные на подсчёте ссылок, отличаются детерминированностью, предсказуемостью и меньшей вычислительной сложностью. [11]

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. [10]

Далее будут подробно описаны некоторые подходы к реализации подсчёта ссылок. [10]

#### 1.3.4.1. Отложенный подсчёт ссылок

Отложенный подсчёт ссылок позволяет уменьшить накладные расходы на поддержание счётчика ссылок в актуальном состоянии за счет отсутствия корректировок при сохранении ссылки в стеке. [2]

Как правило, большинство сохранений 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. [2]

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 счета (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. [2]

Стоит заметить, что отложенный подсчёт ссылок не позволяет корректно обрабатывать объекты с циклическими ссылками, поэтому его часто используют вместе с трассирующим сборщиком мусора. [10]

### 1.3.4.2. Взвешенный подсчёт ссылок

Распределенная сборка мусора [2] — 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 затраты особенно высоки при использовании трассирующего сборщика мусора, поэтому вместо него обычно используются другие методы, включая взвешенный подсчёт ссылок. [2]

Взвешенный подсчёт ссылок [2] – техника подсчёта ссылок, которая широко используется для распределенной сборке 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. [2]

### 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. [2]

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 фикси-

руется 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. [2]

#### 1.3.4.4. Подсчёт ссылок с флагом

Подсчёт ссылок с флагом (one-bit reference counting) [2] — это эвристический механизм, позволяющий с минимальными накладными расходами памяти проверить, используется ли объект в программе.

Однобитовый счетчик ссылок является частным случаем счетчика ссылок 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. [2]

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. [2]

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. [2]

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. [2]

### 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. [13]

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. При этом стоит заметить, что в любой структуре мусорных указателей все ссылки должны быть связаны с внутренними указателями, то есть с указателями между объектами внутри структуры. Также отметим, что циклы сборки мусора могут возникать только при удалении указателя, в результате которого число ссылок на объект становится равным нулю. [13]

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 устраняя 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 объект, ни его транзитивное замыкание не являются мусором. [13]

Алгоритм сборки циклических структур данных работает в три этапа. [13]

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.

2. **Сканирование**. Проверяется число ссылок для каждого узла в этих подграфах: если оно не равно нулю, то объект должен быть используемым из-за ссылки, являющейся внешней по отношению к отслеживаемому подграфу, и поэтому результат первого этапа должен быть отменён, перекра-

шивая живые серые объекты в чёрный цвет. Другие серые объекты окрашиваются в белый цвет.

3. **Очистка**. Наконец, все элементы подграфа, которые на данном этапе являются белыми, могут быть освобождены.

## 2 Описание существующих решений

### 2.1. Swift / C#

#### 2.1.1. Управление памятью

#### 2.1.2. Алгоритм работы сборщика мусора

### 2.2. Python

#### 2.2.1. Управление памятью

#### 2.2.2. Алгоритм работы сборщика мусора

### 2.3. Java

#### 2.3.1. Управление памятью

#### 2.3.2. Алгоритм работы сборщика мусора

### 2.4. Golang

#### 2.4.1. Управление памятью

#### 2.4.2. Алгоритм работы сборщика мусора

### 2.5. Haskell

#### 2.5.1. Управление памятью

#### 2.5.2. Алгоритм работы сборщика мусора

# 3 Классификация существующих решений

## 3.1. Критерии сравнения алгоритмов распределения памяти

GC паузы... Например, может ли gc перейти определённый порог оверхеда
Реализован ли алгоритм поколений в GC
Из gc handbook:
Safety
Throughput
Completeness and promptness
Pause time
Space overhead
Optimizations for specific languages
Scalability and portability

## 3.2. Сравнительный анализ алгоритмов распределения памяти

## 3.3. Вывод

# ЗАКЛЮЧЕНИЕ

Большинство современных языков программирования строятся на одной из трех ссылочных моделей:

Первая категория это языки с ручным управлением временем жизни объектов. Примеры — C/C++/Zig. В этих языках объекты аллоцируются и освобождаются вручную, а указатель — это просто адрес памяти, никого ни к чему не обязывающий.

Во вторую категорию попадают языки с подсчетом ссылок. Это Objective-C, Swift, Частично Rust, C++ при использовании умных указателей и некоторые другие. Эти языки позволяют автоматизировать до некоторой степени удаление ненужных объектов. Но это имеет свою цену. В многопоточный среде такие счетчики ссылок должны быть атомарными, а это дорого. К тому же, подсчет ссылок не может освободить все виды мусора. Когда объект А ссылается на объект Б а объект Б обратно ссылается на объект А такая закольцованная иерархия не может быть удалена подсчетом ссылок. Такие языки как Rust, Swift вводят дополнительные не владеющие ссылки которые решают проблему закольцовок ценой усложнения объектной модели и синтаксиса.

В третью категорию попадают большинство современных языков программирования. Это языки с автоматической сборкой мусора: Java, JavaScript, Kotlin, Python, Lua... В этих языках ненужные объекты удаляются автоматически, но есть нюанс. Сборщик мусора потребляет очень много памяти и процессорного времени. Он включается в случайные моменты времени и ставит основную программу на паузу. Иногда полностью — на все свое время работы, иногда частично. Сборки мусора без пауз не существует. Гарантию сборки всего мусора может дать только алгоритм который просматривает всю память и останавливает приложение на все свое время работы. В реальной жизни такие сборщики давно не используются ввиду своей неэффективности. В современных системах некоторые мусорные объекты не удаляются вообще.

Кроме того, само определение ненужного объекта нуждается в уточнении. Если, например, у нас есть GUI-приложение, и вы убираете с формы какой-то управляющий элемент, подписанный на события таймера, он не может быть удален просто так потому что где-то в объекте таймера хранится ссылка на этот объект, и сборщик мусора не будет считать такой объект мусором.

Как уже говорилось выше, каждая из трех ссылочных моделей имеет свои

недостатки. В первом случае имеем дыры в memory safety и утечки памяти, во втором случае мы имеем медленную работу в многопоточной среде и утечки памяти из-за закольцовок, в третьем получаем спорадические остановки программы сильное потребление памяти, процессора и необходимость ручного разрыва ссылок когда объект становится не нужным. К тому же система с подсчетом ссылок и сборкой мусора не позволяют управлять временем жизни других ресурсов — таких как открытые файловые дескрипторы, идентификаторы окон, процессов, шрифтов и так далее. Эти методы рассчитаны только на память. Есть еще одна проблема систем со сборкой мусора — виртуальная память. В условиях, когда программная система накапливает мусор, а затем сканирует память для его освобождения, вытеснение части адресного пространства на внешний носитель может полностью убить производительность приложения. Поэтому сборка мусора не совместима с виртуальной памятью.

То есть проблемы есть и текущие методы их решения имеют изъяны.

cornell2:

В документе представлена эта структура, показано, что трассировка и подсчет ссылок на самом деле двойственны, показано, как оптимизированные сборщики мусора представляют собой гибриды трассировки и подсчета ссылок, а также представлен анализ затрат для определения компромисса между временем и пространством сборщиков в этом пространстве проектирования.

# СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ


1. *Таненбаум Э.*, *Бос Х.* Современные операционные системы. — 4-е изд. — Питер, 2015.
2. Memory Management Glossary [Электронный ресурс]. — Режим доступа, URL: https://www.memorymanagement.org/glossary/ (дата обращения: 12.10.2023).
3. *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).
4. Внутреннее устройство Windows / Р. М. [и др.]. — 7-е изд. — Питер, 2018.
5. *Bovet D. P.*, *Cesati M.* Understanding the LINUX KERNEL. — 3-е изд. — O'Reilly, 2005.
6. Common Language Runtime (CLR) overview [Электронный ресурс]. — Режим доступа, URL: https://learn.microsoft.com/en-us/dotnet/standard/clr (дата обращения: 12.10.2023).
7. 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.
8. Overview [Электронный ресурс]. — Режим доступа, URL: https://www.memorymanagement.org/mmref/begin.html (дата обращения: 15.10.2023).
9. Allocation techniques [Электронный ресурс]. — Режим доступа, URL: https://www.memorymanagement.org/mmref/alloc.html#mmref-alloc (дата обращения: 12.10.2023).
10. Recycling techniques [Электронный ресурс]. — Режим доступа, URL: https://www.memorymanagement.org/mmref/recycle.html (дата обращения: 15.10.2023).
11. *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).

12. *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.

13. *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.

14. Copying Garbage Collection [Электронный ресурс]. — Режим доступа, URL: https://www.cs.cornell.edu/courses/cs312/2003fa/lectures/sec24.htm (дата обращения: 21.10.2023).

# ПРИЛОЖЕНИЕ А