Правительство Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования
«Санкт-Петербургский государственный университет»

Кафедра Системного Программирования

Самофалов Александр Владимирович

Библиотека почти точной копирующей сборки мусора для C++

Дипломная работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
к. ф.-м. н. Булычев Д. Ю.

Рецензент:
к. ф.-м. н. Плисс О.А.

Санкт-Петербург
2015

SAINT-PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Aleksandr Samofalov

Mostly precise copying garbage collection library for C++

Graduation Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
Dmitri Boulytchev

Reviewer:
Oleg Pliss

Saint-Petersburg
2015

Оглавление

Введение

Память является одним из самых критических ресурсов приложения: неправильная работа с ней может привести к значительной потере производительности или даже к ошибкам в работе программы. Для объектов, размер которых неизвестен во время компиляции, нельзя выделить память статически. Поэтому существует специальная область памяти, называемая кучей (*heap*), в которой возможно выделить память во время исполнения программы, обратившись к специальному менеджеру.

Так как память, доступная приложению, ограничена, то полученные области памяти должны быть освобождены и возвращены обратно куче. В самом простом случае используется ручное управление памятью, при котором обязанность по выделению и освобождению памяти ложится на программиста. Но, во-первых, ручное управление памятью не всегда удобно. Например, при использовании некоторых функциональных языков программирования очень сложно определить время жизни объекта, а значит, и момент, когда нужно освобождать занимаемую им память. Во-вторых, программисты допускают ошибки в управлении памятью, которые могут привести к неправильной работе программы.

Одна из самых распространённых проблем в управлении памятью — это “утечки памяти” (*memory leak*), которые возникают, если какая-нибудь область памяти после использования не была освобождена и стала не доступной для приложения. При управлении памятью также встречается проблема фрагментации, возникающая при невозможности выделить непрерывный блок нужного размера, несмотря на то, что суммарное количество свободной памяти превосходит этот размер. Еще одной распространенной проблемой являются “висячие ссылки” (*dangling pointers*) — указатели на освобожденные области памяти.

Некоторые проблемы ручного управления памятью решает автоматическое управление памятью, или сборка мусора (*garbage collection*). Сборщик мусора обходит память приложения, определяет недоступные

для программы объекты в куче и освобождает их. Недоступными считаются объекты, на которые нет указателей из корневого множества, под которым понимаются объекты на стеке и глобальные переменные. Сборка мусора является точной, если она правильно обнаруживает все недоступные объекты. Точная сборка мусора возможна при соблюдении следующих условий:

1. Сборщик мусора должен правильно строить корневое множество.
2. Сборщик мусора должен правильно определять все указатели внутри объектов.

Если хотя бы одно из условий не выполняется, то возможна только консервативная сборка мусора. В этом случае используются некоторые эвристики для обнаружения указателей на объекты.

Использование сборки мусора имеет недостатки. Главным из этих недостатков являются высокие накладные расходы: например, многие сборщики мусора приостанавливают приложение на время своей работы, что может существенно ухудшить производительность данного приложения. Тем не менее, для некоторых приложений может быть критична не производительность, а другие свойства. Поэтому существует множество различных алгоритмов сборки мусора. Среди них можно выделить копирующие сборщики мусора, которые позволяют уменьшить фрагментацию памяти за счёт того, что после цикла своей работы копируют все выжившие объекты в один непрерывный блок памяти.

C++ — популярный язык общего назначения, широко используемый для разработки приложений, для которых производительность является критичной, а также для различных низкоуровневых приложений. К сожалению, спецификация *C++* не предусматривает использования сборки мусора. Тем не менее, реализации сборщика мусора для этого языка существуют[2, 6]. Большинство из них являются консервативными, хотя хотелось бы иметь точный сборщик мусора для *C++*.

В рамках проекта лаборатории JetBrains[1] на математико-

[1]https://www.jetbrains.com

механическом факультете СПбГУ была реализована библиотека сборки мусора для языка C++[3]. Реализованный сборщик мусора является неконсервативным, не требует поддержки от компилятора, не запрещает использовать какие-либо конструкции языка, но накладывает некоторые ограничения на программу.

К сожалению, у этой библиотеки имеются недостатки. Целью данной работы является устранение недостатков данной библиотеки. Для этого были поставлены следующие задачи:

- исправление ошибок в работе с объектами;
- добавление поддержки многопоточности;
- реализация сжимающего алгоритма сборки мусора;
- тестирование и апробация полученной реализации.

1. Существующие методы автоматического управления памятью в C++

Сборка мусора была впервые применена для языка Lisp[8] в 60-х годах XX века. На сегодняшний день технология сборки мусора широко используется в таких популярных языках, как Java[2], OCaml[3], Python[4].

Однако, как уже отмечалось ранее, несмотря на популярность языка C++, его стандарт не предусматривает использование сборки мусора. Тем не менее, было сделано несколько попыток реализации сборщика мусора для C++. Различные подходы к автоматическому управлению памятью будут описаны далее.

1.1. Сборщик мусора Бёма-Демерса-Вайзера

Одним из самых известных сборщиков мусора для C++ является сброщик мусора Бёма-Демерса-Вайзера (Boehm-Demers-Weiser garbage collector)[6, 5, 4]. Этот сборщик мусора является консервативным: машинное слово считается указателем, если оно является адресом внутри какого-нибудь объекта в куче. Этот сборщик мусора работает за счёт того, что заменяет функции выделения и освобождения памяти на свои специальные функции. Эти функции, кроме работы с памятью, сохраняют необходимую для сборки мусора метаинформацию.

Одна из главных целей, которые ставилась перед разработчиками этого сборщика мусора, — это совместимость с существующими программами на языке C++ без необходимости что-либо менять в исходном коде приложения. Этот сборщик мусора накладывает минимальные ограничения на работу программы. Для работы с ним достаточно динамически загрузить библиотеку со сборщиком мусора во время запуска приложения. Также сборщик мусора Бёма-Демерса-Вайзера обладает функциональностью поиска утечек памяти в приложении.

[2]http://java.com

[3]http://ocaml.org

[4]http://www.python.org

Основным недостатком этого сборщика мусора является его консервативность. Из-за консервативности он не всегда способен идентифицировать весь мусор и, соответственно, очистить память из-под него. Следствием этого могут стать задержки в работе менеджера памяти, сборщика мусора или даже исчерпание памяти.

1.2. Почти копирующий сборщик мусора

Другим консервативным сборщиком мусора для языка C++ является почти копирующий сборщик мусора (*Mostly Copying Collector*)[2, 1]. Данный сборщик мусора является копирующим, что имеет некоторые преимущества по сравнению с остальными алгоритмами, например, более быстрое выделение памяти и уменьшение фрагментации кучи. В почти копирующем сборщике мусора вся память, доступная куче, делится на две области — старую (*old*) и новую (*next*). Во время сборки мусора все обнаруженные достижимые объекты копируются из старой области памяти в новую, а во всех указателях адрес этих объектов заменяется на новый. После сборки мусора области меняются местами.

Так как этот сборщик является консервативным, то он не всегда может определить указатели. Если сборщик мусора не уверен, является ли данное машинное слово указателем, оно считается неявным корнем, а объект, на который оно указывает, переносится в новую область без копирования. Для того, чтобы это было возможно, области памяти представлены не непрерывными блоками памяти, как в самом простом mark-and-compact алгоритме[7], а разделены на несвязанные участки памяти. Из вышесказанного следует, что этот сборщик мусора показывает хорошие результаты, только когда он способен точно находить большой процент указателей.

Для того, чтобы приложение работало с данным сборщиком мусора, необходимо его собрать вместе с библиотекой сборщика, а также использовать предоставляемые ей методы для выделения и освобождения памяти. Также для улучшения работы сборщика мусора пользователь может отметить указатели внутри собственных структур данных.

1.3. “Умные указатели”

Под “умными” указателями понимаются классы, которые имитируют работу указателя. За счёт различных механизмов они способны автоматически освобождать память.

Одним из типов умных указателей является указатель единоличного владения (*unique pointer*). Данный указатель реализует идиому “получение ресурса есть инициализация” (*Resource Acquisition Is Initialization*), которая заключается в том, что выделение памяти под объект происходит во время инициализации “умного” указателя. Когда “умный” указатель уничтожается, он в своём деструкторе освобождает выделенную память. Для того, чтобы память из под объекта не была освобождена дважды, для данного указателя запрещено копирование. Владение над объектом можно только передать другому указателю.

К сожалению, данный указатель нельзя использовать, если у объекта есть несколько различных владельцев — например, при использовании одного ресурса из разных потоков. Для решения данной проблемы существует указатель раздельного владения (*shared pointer*). При применении этого умного указателя используется подсчёт ссылок — с каждым объектом ассоциируется дескриптор, в котором хранится число ссылающихся на этот объект указателей. При инициализации указателя счётчик увеличивается, при уничтожении уменьшается. При присваивании указателю нового значения счётчик соответствующий старому значению уменьшается, а для нового значения увеличивается. Если в какой-то момент счётчик стал равным нулю, то память из-под него освобождается, так как это означает, что никто больше не ссылается на этот объект.

Несмотря на то, что подсчёт ссылок является широко распространённой техникой, он имеет существенный недостаток — неспособность работать с циклическими ссылками. Если два объекта ссылаются друг на друга, то их счётчики ссылок содержат, как минимум, по одному объекту, а значит, не станут равными нулю, и память из-под них не будет освобождена. Эта проблема разрешается с помощью невладею-

щих указателей (*weak pointer*), отличающихся от указателей раздельного владения тем, что не влияют на счётчик ссылок объекта, на который они указывают. Однако забота о правильном и своевременном использовании невладеющих указателей является ответственностью программиста.

Существует множество реализаций умных указателей в различных библиотеках, например в boost[5], Qt[6]. Начиная со стандарта C++11 вышеописанные умные указатели были добавлены в стандартную библиотеку языка C++, в которой реализованы на основе библиотеки boost.

1.4. Библиотека неконсервативной сборки мусора

Умные указатели можно использовать для реализации сборщика мусора. Одной из таких реализаций является библиотека неконсервативной сборки мусора, которая была написана в лаборатории JetBrains. Данная реализация использует умные указатели для того, чтобы строить корневое множество, а также для определения указателей внутри объектов. Пользователю для корректной работы с этим сборщиком мусора необходимо использовать эти умные указатели, а также предоставленные функции выделения памяти. Библиотека также позволяет работать одновременно и с объектами, управляемыми сборщиком мусора, и с управляемыми вручную пользователями. Также библиотека предоставляет интерфейс для перевода объектов из управляемых в ручные и обратно.

В отличие от некоторых других реализаций сборщика мусора на умных указателях, данная библиотека не накладывает ограничений на используемое множество языка. Также пользователю не нужно указывать расположение указателей внутри объекта — сборщик мусора находит их самостоятельно.

К сожалению, данный сборщик мусора имеет недостатки. Во-первых, он не поддерживает многопоточность и корректно работает

[5]http://www.boost.org
[6]http://www.qt.io

только для программ, использующих один поток. Во-вторых, данная реализация использует самый простой алгоритм сборки мусора — mark-and-sweep, который проигрывает более “продвинутым” алгоритмам. В-третьих, реализация имеет высокие накладные расходы на сканирование полей объекта при его создании для обнаружения указателей.

Данная работа посвящена развитию описанной выше библиотеки.

2. Реализация

В данном разделе описана работа по устранению недостатков библиотеки.

2.1. Работа с объектами

C++ — мультипарадигменный язык, который поддерживает объектно-ориентированное программирование. В данной парадигме одним из основных понятий является объект — сущность, которая, используя свои данные, может реагировать на посылаемые ей сообщения. В C++ посылка и приём сообщений реализованы с помощью методов — функций, которые определяют, какие действия может выполнять объект. Большинство методов имеют неявный аргумент *this* — указатель на объект, у которого вызывается данный метод.

Во время изучения возможностей библиотеки неконсервативной сборки мусора была обнаружена ситуация при работе с объектами, которая приводит к ошибкам в работе программы. Рассмотрим следующий пример, который лучше всего иллюстрируется кодом в листинге 1.

Листинг 1: Пример ошибки

```
1  class B {
2      gc_ptr<A> a;
3  }
4
5  class A {
6      void f() {
7          b->a = NULL;
8          // gc
9          g();
10     }
11
12     void g() {
13         ...
14     }
15 }
```

На объект A существует единственный управляемый указатель из объекта B. Если в процессе выполнения какого-нибудь метода A этот указатель перестанет указывать на A, и сразу после этого будет совершена сборка мусора, то сборщик мусора посчитает A недостижимым, и память из-под него будет освобождена. Однако приложение может обратиться к объекту с помощью указателя *this*, что и приводит к ошибке.

Для решения данной проблемы необходимо хранить все ссылки на *this*, которые находятся на стеке, но C++ не предоставляет возможности для этого. Для того, чтобы вызвать метод у объекта, на который указывает управляемый указатель, сначала нужно получить "сырой" указатель на этот объект. При получении этого "сырого" указателя, он сохраняется в специальную структуру данных — хэш-таблицу для разыменованных указателей. Это происходит не только при вызове метода у объекта, но и при обращении к одному из его полей, но далее мы увидим, что это будет также полезно.

При сборке мусора для обнаружения достижимых объектов используются не только управляемые указатели, но и стек приложения. Для этого обходится каждое слово стека. Если слово содержится в хэш-таблице разыменованных указателей, то этот указатель считается корнем, и все объекты, достижимые от него, являются живыми.

После того, как метод выполняется полностью, объект перестаёт быть доступным из указателя *this* этого метода. Этот момент сборщик мусора не может отследить, поэтому можно считать, что выполняется следующее приближение: после завершения сборки мусора, из хэш-таблицы разыменованных указателей удаляются все указатели, которые не были найдены на стеке.

Данная процедура является консервативной, то есть возможна ситуация при которой не весь мусор будет найден. Но такая ситуация является очень редкой: для её воспроизведения необходимо чтобы, во-первых, не осталось никаких указывающих на объект указателей и, во-вторых, при этом на стеке было расположено число равное его адресу. Поэтому сборщик мусора можно назвать почти точным.

Во время работы программы может произойти такая ситуация, что

сборка мусора не вызывается длительное время, но количество разыменованных указателей продолжает увеличиваться. Это приводит к тому, что следующий запуск сборщика мусора будет работать очень долгое время. Чтобы этого избежать, при достижении хэш-таблицы разыменованных указателей некоторого размера происходит его очистка: обходится стек исполнения, и из таблицы удаляются все объекты, которые не были найдены на стеке.

2.2. Многопоточность

Большинство промышленных приложений являются многопоточными. Поэтому поддержка многопоточности в сборщике мусора является естественным требованием. Исходная библиотека сборки мусора это требование не выполняла — она корректно работала только с выполняющимися в одном потоке программами. В данном разделе подробно описывается реализация поддержки многопоточности.

2.2.1. Работа с потоками

Для построения корневого множества библиотека использует структуры данных, которые хранят управляемые указатели и разыменованные указатели. Эти структуры являются глобальными, и их использование в многопоточном приложении может привести к ошибкам, так как несколько потоков могут одновременно обратиться к ним. Для решения этой проблемы было решено создать для каждого потока свой экземпляр этих структур. Для реализации была использована возможность *C++11* указывать глобальные переменные как локальные для потока (*threadlocal*).

В C++ не существует стандартной реализации поддержки многопоточности. Однако самой распростаненной из них является POSIX Threads(*pthreads*)[9]. Поэтому было решено добавить поддержку многопоточности в сборщике мусора на основе этой библиотеки.

Для сборки мусора в многопоточном окружении необходимо знать все потоки данного процесса. Однако в *pthreads* отсутствует стандарт-

ный способ для получения этой информации. Для решения этой проблемы была реализована обёртка над *pthreads*, которая заменяет функции создания потоков и работы с ними на следующие:

- *thread_ create* — функция создания потока, в неё передаётся точка входа — указатель на функцию, с которого начнёт исполнение новый поток;
- *thread_ join* — функция присоединения к потоку, она получает на вход идентификатор потока, завершения которого будет ждать текущий;
- *thread_ exit* — функция завершения исполнения текущего потока;
- *thread_ cancel* — функция завершения потока по его идентификатору.

Также переопределяются функции для работы с мьютексами и условными переменными.

Для корректной работы сборщика мусора клиентское приложение обязано использовать эти функции.

Для каждого из потоков создаётся дескриптор — структура, которая содержит в себе целочисленный идентификатор, ссылки на необходимые для сборки мусора структуры этого потока, точку входа и флаги состояния. Все дескрипторы работающих потоков хранятся в виде односвязного списка.

Во время этапа маркировки в алгоритме сборки мусора для построения корневого множества обходится стек вызовов, поэтому необходимо определять его вершину для каждого потока. Для этого при создании потока в качестве точки входа в него устанавливается вспомогательная функция *start_ routine*. Эта функция выполняется первой в новом потоке, поэтому адрес её кадра стека можно использовать как вершину стека для потока. Она также используется для корректной инициализации локальных для потока структур данных, так как ей доступны указатели на них. Поэтому в начале своей работы она записывает в

дескриптор вершину стека и указатели на необходимые для сборщика мусора структуры. После этого она добавляет дескриптор в список дескрипторов и запускает настоящую точку входа. После завершения основной фукнции, *start_ routine* убирает дескриптор из списка и завершает свою работу. Также дескрипторы убираются из списка в функциях *thread_ exit* и *thread_ cancel*.

Одновременное изменение списка дескрипторов небезопасно и может привести к ошибкам. Поэтому для того, чтобы получить доступ к этому списку, поток должен сначала захватить блокировку на мьютексе *gc_ mutex*, что гарантирует изменение списка не более, чем одним потоком в один момент времени.

2.2.2. Безопасные точки

Сборка мусора не может запускаться из произвольной точки программы. Например, если сборка мусора начнёт свою работу посредине создания объекта, то, так как на объект ещё нет ссылок, то сборщик мусора посчитает его недостижимым и освободит память из-под него. Но после этого программа продолжит создавать этот объект и будет обращаться к освобождённой памяти, что повлечёт ошибки.

Для того чтобы избежать ошибок, в программе необходимо отметить участки кода, в которых может безопасно выполниться сборка мусора. Такие участки называются безопасными точками. Для корректной работы приложения сборка мусора должна запускаться только, когда все потоки находятся в безопасных точках.

В каждую безопасную точку вставляется код, который гарантирует, что сборка мусора будет запускаться только внутри них. Дескриптор потока содержит флаг, который указывает, что поток находится в безопасной точке.

В данной реализации сборщика мусора точка является безопасной, когда на каждый достижимый объект в куче существует управляемый указатель. То есть безопасная точка не может располагаться посредине создания нового объекта, а также между взятием сырого указателя на объект и добавлением его в хэш-таблицу разыменованых объектов.

Для того чтобы пользователю не приходилось вручную расставлять код безопасной точки, в данной реализации безопасные точки заранее расставлены внутри заранее некоторых функций библиотеки, например, внутри функции выделения памяти.

Код безопасной точки приведён в следующем разделе.

2.2.3. Stop-the-world

Если запустить сборку мусора параллельно работе программы, то это может привести к различным ошибкам, так как куча может изменится между стадией маркировки и удаления недостижимых объектов. Для решения этой проблемы существует несколько основных подходов к работе сборщика мусора в многопоточной программе.

- Stop-the-world сборщики мусора, во время работы которых все потоки приложения приостанавливаются.

- Инкрементальные сборщики мусора, в которых сборка мусора выполняется параллельно работе программы. Они используют сложные алгоритмы для синхронизации сборщика мусора и приложения, но увеличивают накладные расходы на сборку мусора.

В данной работе реализован stop-the-world сборщик мусора.

Существует несколько способов выбора потока, в котором будет выполняться сборка мусора. В данной работе используется следующая стратегия: каждый поток может заявить о том, что он будет собирать мусор в этом цикле сборки мусора. Если таких потоков несколько в один момент времени, то сборщиком будет первый заявивший. Для реализации данного поведения используется глобальная переменная *gc_thread*, в которую поток записывает свой дескриптор, если он хочет быть сборщиком мусора. Для того чтобы в эту переменную одновременно не записали несколько потоков, для обращения к ней поток должен захватить мьютекс *gc_mutex*.

Как отмечалось ранее, сборка мусора возможно только, когда все потоки находятся в безопасных точках. Чтобы это выполнялось, каж-

дый поток обходит список дескрипторов и для каждого из них проверяет, установлен ли флаг безопасной точки. Если это не верно, то этот поток приостанавливается, ожидая, пока кто-нибудь не сообщит о том, что он находится в безопасной точке. Данное поведение реализовано с помощью условной переменной (*condition variable*) *safepoint_reached*.

Внутри безопасной точки поток проверяет, записан ли в переменную *gc_thread* дескриптор какого-нибудь потока. Если это так, это значит, что сборщик мусора ожидает, пока какой-то поток окажется в безопасной точке. Текущий поток отправляет сигнал потоку, ожидающему условную переменную *safepoint_reached*, и приостанавливается до завершения сборки мусора.

Итоговый код безопасной точки описан далее в алгоритме 1.

Algorithm 1 Безопасная точка

```
1: lock(gc_mutex)
2: current_thread ← дескриптор текущего потока
3: current_thread.safepoint ← true
4: if gc_thread ≠ NULL then
5:     notify(safepoint_reached)
6:     wait(gc_finished)
7: end if
8: current_thread.safepoint ← false
9: unlock(gc_mutex)
```

Кроме этого, поток, который ждёт завершения работы другого потока или освобождения мьютекса, также находится в безопасной точке. Поэтому во всех функциях, которые блокируют поток (например, в функции *thread_join*) также выставляется флаг безопасной точки для текущего потока.

После того, как все потоки достигли безопасной точки и остановились, поток со сборщиком мусора выполняет сборку мусора. После этого он сбрасывает переменную *gc_thread* и сообщает всем потокам о завершении с помощью условной переменной *gc_finished*.

Полная реализация *stop-the-world* описана в алгоритме 2.

Algorithm 2 Stop-the-world

```
1: lock(gc_mutex)
2: current_thread ←
3: current_thread.safepoint ← true
4: if gc_thread = NULL then
5:     gc_thread ← current_thread
6:     for all thread ← threads_list do
7:         if thread.safepoint = false then
8:             wait(safepoint_reached)
9:         end if
10:    end for
11:    gc()
12:    notify(gc_finished)
13:    gc_thread ← NULL
14: else
15:    notify(safepoint_reached)
16:    wait(gc_finished)
17: end if
18: current_thread.safepoint ← false
19: unlock(gc_mutex)
```

2.3. Сжатие

В этом разделе подробно описывается реализация используемого в данной библиотеке сжимающего алгоритма сборки мусора.

2.3.1. Устройство кучи

Для того чтобы реализовать сжимающую сборку мусора, необходима реализации кучи, которая поддерживала бы перенос объектов и осуществляла его. В исходной библиотеке использовалась куча Дага Ли[7] — один из самых известных менеджеров динамической памяти, долгое время являющийся стандартным в некоторых сборках стандартной библиотеки языка C и операционной системы Linux. К сожалению, эта куча не поддерживает перенос данных. К тому же данная куча имеет запутанную структуру исходного кода, а также сильно оптимизирована. Поэтому в неё нелегко добавить поддержку копирования. Поэтому

[7] http://g.oswego.edu/dl/html/malloc.html

в данной работе была написана своя реализация кучи, которая поддерживает копирование.

В данной реализации вся память кучи поделена на страницы одинакового размера. По умолчанию страницы имеют размер, равный размеру страницы памяти в операционной системе. Страницы не обязаны находиться в одном непрерывном куске памяти, и поэтому хранятся в односвязном списке *pages*. Память под страницы запрашивается у операционной системы с помощью функции *mmap*. Из-за того, что *mmap* является достаточно долгой операцией, и запросить один блок большого размера быстрее, чем запросить блок маленького размера, за один запрос у операционной системы получается сразу несколько страниц. Неиспользуемые страницы хранятся в односвязном списке *free_ list*. Если необходимо получить новую страницу, куча сначала берет свободные страницы из *free_ list*, а если он пуст, то запрашивает следующий блок у операционной системы. С целью улучшения производительности с каждым новым запросом к операционной системе количество запрашиваемой памяти увеличивается в два раза. Данный процесс описан в алгоритме 3.

Algorithm 3 Запрос свободной страницы

```
1:  if empty(free_list) then
2:      new_memory ← mmap(PAGE_SIZE * per_request)
3:      curr ← new_memory
4:      while curr < new_memory + PAGE_SIZE * per_request do
5:          push(free_list, curr)
6:          curr ← curr + PAGE_SIZE
7:      end while
8:      per_request ← per_request * 2
9:  end if
10: return pop(free_list)
```

Каждая страница состоит из заголовка и непрерывного куска памяти, который целиком делится на блоки. Каждый блок содержит свой размер и полезные данные. Блоки делятся на два типа — свободные и занятые. В занятых блоках хранится информация, которая в данный момент используется приложением. В каждой странице все блоки,

кроме самого последнего, обязательно являются занятыми, а последний обязательно является свободным. Поэтому, при выделении памяти внутри страницы, занятый блок создаётся на месте последнего свободного, а оставшаяся память становится новым свободным блоком. Для ускорения этого процесса заголовок страницы содержит указатель на свой свободный блок.

В данной куче для ускорения работы выделение памяти происходит только в последней странице. Если в последней странице не хватает места, то запрашивается новая страница и блок выделяется в ней.

Для блоков памяти, размер которых больше, чем размер страницы, куча напрямую запрашивает память у операционной системы. Для таких объектов в куче создаётся отдельный односвязный список. Алгоритм 4 описывает процесс выделения памяти.

Algorithm 4 Выделение памяти

```
1:  if size > BIG_BLOCK_THRESHOLD then
2:      block ← mmap(size + header_size)
3:      push(big_blocks, block)
4:      return block.data
5:  end if
6:  if pages = NULL then
7:      pages ← request_new_page()
8:      last_page ← pages
9:  end if
10: if not last_page.free_block.size ≥ size then
11:     page ← request_new_page()
12:     last_page.next ← page
13:     last_page ← page
14: end if
15: block ← last_page.free_block
16: last_page.free_block ← block.data + header_size
17: last_page.free_block.size ← block.size − size − header_size
18: block.size ← size
19: return block.data
```

Так как данная куча используется только вместе со сборщиком мусора, то в ней нет необходимости освобождать память вне процесса сборки мусора.

2.3.2. Закрепление объектов

Некоторые объекты невозможно перенести, избежав при этом ошибок в работе программы. Например, при вызове метода какого-нибудь объекта на стек записывается указатель *this*, точное положение которого невозможно узнать без поддержки компилятора.

Поэтому в предложенной реализации существует возможность указать куче, что некоторый блок памяти является закреплённым — его нельзя переносить. Для этого в заголовке каждого блока создаётся специальный флаг закреплённости. Для уменьшения размера заголовка, флаг закреплённости располагается в младшем бите размера блока. Чтобы это не создавало ошибок, куча при выделении памяти для блока округляет его размер вверх до восьми. В таком случае, младшие три бита размера всегда равны нулю, и их можно использовать для расположения различных флагов.

2.3.3. Алгоритм

Реализованный в библиотеке алгоритм сборки мусора состоит из нескольких этапов, самыми важными из которых являются пометка и сжатие. Далее будут подробно рассмотрены все этапы работы алгоритма.

1. Поток, инициирующий сборку мусора, приостанавливает приложение, как описано в разделе 2.2.3.

2. Сборщик мусора обходит корневое множество и запускает поиск достижимых из них объектов. Достижимые объекты помечаются флагом достижимости в соответствующем блоке в куче.

3. Сборщик мусора обходит стеки вызовов всех потоков и ищет в них указатели, содержащиеся в хэш-таблице разыменованных объектов. Если такие указатели были найдены, то объекты, на которые они указывают, помечаются флагом закреплённости. Благодаря этому все объекты, которые нельзя переносить, не будут перене-

сены. Также от таких объектов выполняется поиск достижимых объектов, как в предыдущем этапе.

4. Когда все живые объекты помечены, сборщик мусора приступает к освобождению памяти. Для начала он обходит все блоки с большими объектами, находит те, у которых не выставлен ни флаг достижимости, ни флаг закреплённости и освобождает память из-под них с помощью функции *munmap*.

5. Далее сборщик мусора обходит все страницы в *pages* и ищет в них закреплённые блоки. Если на какой-то странице содержится хотя бы один закреплённый блок, то эта страница считается пережившей сборку мусора и переносится в список выживших страниц *alive_pages*.

6. После этого алгоритм производит сжатие. Для этого он обходит все блоки в оставшихся в *pages* страницах. Если у блока выставлен флаг достижимости, то в *alive_pages* находится наименьший свободный блок, размер которого превосходит размер копируемого блока. Если такой блок не находится, то в *alive_pages* добавляется новая страница. После этого информация из старого блока копируется в новый, а в старый блок записывается указатель на новый блок. Для ускорения процесса поиска наименьшего блока сборщик мусора строит список свободных блоков каждого размера. Более подробно этот этап описан в алгоритме 5.

7. После этого сборщик мусора исправляет указатели на перенесённые объекты. Для этого сборщик обходит корневое множество, и если объект, на который указывает корень, был перенесён, то изменяет его на указатель на новый блок. Такую же операцию сборщик мусора проделывает со всеми указателями внутри объектов в *alive_pages*.

8. Далее сборщик мусора переносит все страницы из *pages* во *free_list*, заменяет *pages* на *alive_pages* и устанавливает новый

last_page.

9. Сборщик мусора обходит хэш-таблицу разыменованных указателей и удаляет оттуда те, которые не были найдены на этапе 3.

10. Последним этапом сборщик мусора сообщает всем потокам о завершении работы алгоритма.

Algorithm 5 Сжатие

```
1:  for i ← 1..PAGE_SIZE do
2:      clear(free[i])
3:  end for
4:  for page ← alive_pages do
5:      push(free[page.free_block.size], page)
6:  end for
7:  for page ← pages do
8:      for block ← page do
9:          if block.marked then
10:             size ← block.size
11:             page_to_copy ← NULL
12:             for i ← size..PAGE_SIZE do
13:                 if not empty(free[i]) then
14:                     page_to_copy ← pop(free[i])
15:                     break
16:                 end if
17:             end for
18:             if page_to_copy = NULL then
19:                 page_to_copy ← request_new_page()
20:                 push(alive_pages, page_to_copy)
21:             end if
22:             copy(block.data, page_to_copy.free_block.data)
23:             block.data = page_to_copy.free_block.data
24:             page_to_copy.free_block ← block.data + header_size
25:             new_size ← block.size − size − header_size
26:             page_to_copy.free_block.size ← new_size
27:             push(free[new_size], page_to_copy)
28:         end if
29:     end for
30: end for
```

3. Тестирование и апробация

Данная реализация сборщика мусора была функционально протестирована на тестах, написанных вручную. Для проверки отсутствия регрессионных ошибок использовались существующие тесты, для проверки нового функционала были написаны новые тесты.

Также для данной реализации было произведено тестирование производительности. Сравнение проводилось с производительностью работы аналогичных тестов без использования автоматического управления памятью и с подсчётом ссылок.

Результаты тестирования представлены ниже в таблицах. В таблице 1 находятся результаты исполнения широко используемого теста производительности Бёма[8]. В данном тесте строятся двоичные деревья различной глубины с разной продолжительностью жизни, а также различным способом построения. В таблице в колонке “raw” указаны результаты для ручного управления памятью, в колонке “shared” — подсчета ссылок, и наконец в “mostly” — результаты данной библиотеки. Каждая из этих колонок в свою очередь делится на три области. Колонка “top” подказывает общее время построения деревьев сверху-вниз, “bottom” показывает время построения снизу-вверх, а колонка “total” обозначает общее время работы приложения. Каждой строке соответствует максимальный размер дерева, его значение указано в столбце “размер”. Результаты приводятся как среднее время из 20 запусков теста, все значения указаны в миллисекундах.

Из данной таблицы видно, что обе реализации автоматического управления памятью примерно на порядок хуже реализации с ручным управлением памяти. Данная библиотека, хотя и проигрывает в скорости для небольших деревьев подсчёту ссылок, для деревьев большой глубины показывает лучшую производительность.

В таблице 2 и представлен результат теста, который демонстрирует производительность работы с коллекциями. В нем создаётся список размером 100000 элементов, над которым потом производятся различ-

[8]http://hboehm.info/gc/gc_bench.html

	raw			shared			mostly		
размер	top	bottom	total	top	bottom	total	top	bottom	total
8	0	0	13	1	0	86	1	0	177
10	0	0	13	6	6	98	4	3	148
12	2	2	15	35	32	141	22	61	216
14	11	11	35	167	153	403	117	183	433
16	57	54	127	822	777	1730	677	443	1325
18	278	275	608	3848	3593	7813	3846	2407	6727

Таблица 1: Тест Бёма

	raw	shared	mostly
Создание	3	7	23
Инвертирование	3	3	3
Сортировка	34	37	38
Всего	44	51	66

Таблица 2: Тест с односвязным списком

ные действия. Результаты также являются средними из 20 запусков, значения указаны в миллисекундах.

Из результатов видно, что хотя в данном тесте затрачивается много времени на создание объектов, производительность библиотеки сравнима с производительностью аналогов.

Также совершенно естественным является то, что для любой реализации можно придумать тест, на котором она будет показывать плохую производительность. Например, для исследуемой библиотеки таким является тест аналогичный предыдущему, но с использованием вектора из стандартной библиотеки языка. Результаты данного теста представлены в таблице 3.

В этом тесте библиотека тратит много времени на построение корневого множества, так как вектор по-умолчанию размещает объекты в

	raw	shared	mostly
Создание	5	30	2993
Сортировка	45	486	770
Всего	53	525	3764

Таблица 3: Тест с вектором

своей собственной области памяти. Предположительно данная проблема может быть решена с помощью введения специального менеджера памяти для коллекций стандартной библиотеки шаблонов.

Заключение

Результатом данной работы является:

- исправление ошибок в работе с объектами;
- добавление поддержки многопоточности;
- реализация сжимающего алгоритма сборки мусора;
- тестирование и апробация полученой реализации.

Исходный код библиотеки распологается по адресу: http://github.com/evagl/mostly-precise-gc.

Список литературы

[1] Compacting garbage collection with ambiguous roots : Rep. : WRL-RR-88-2 / Digital (Palo Alto, CA US ; Cambridge, MA US). Systems research center ; Executor: Joel F. Bartlett : 1988.

[2] Bartlett Joel F. Mostly-Copying Garbage Collection Picks Up Generations and C++ // Technical Report TN-12, DEC Western Research Laboratory. — 1989.

[3] Berezun Daniil, Boulytchev Dmitri. Precise Garbage Collection for C++ with a Non-cooperative Compiler // Proceedings of the 10th Central and Eastern European Software Engineering Conference in Russia. — CEE-SECR '14. — 2014.

[4] Boehm Hans-Juergen. Space Efficient Conservative Garbage Collection. — 1993.

[5] Boehm Hans-J., Demers Alan J., Shenker Scott. Mostly Parallel Garbage Collection // SIGPLAN Not. — 1991. — . — Vol. 26, no. 6.

[6] Boehm Hans-Juergen, Weiser Mark. Garbage Collection in an Uncooperative Environment // Software: Practice and Experience. Vol. 18, no. 9. — 1988.

[7] Fenichel Robert R., Yochelson Jerome C. A LISP Garbage-collector for Virtual-memory Computer Systems // Commun. ACM. — 1969. — . — Vol. 12, no. 11.

[8] Mccarthy John. Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I. — 1960.

[9] Standard for Information Technology — Portable Operating System Interface (POSIX) - System Application Program Interface (API) Amendment 2: Threads Extension (C Language) // IEEE Std. 1003.1c-1995. — 1995.