УДК 004.05
DOI: 10.25559/SITITO.17.202103.603-612

Научная статья

Верификация кучи памяти объектов виртуальной машины

И. А. Петров
ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова», г. Москва, Российская Федерация
119991, Российская Федерация, г. Москва, ГСП-1, Ленинские горы, д. 1
petrov.igor.a@gmail.com

Аннотация

В настоящей статье рассмотрены алгоритмы верификации объектной кучи памяти (heap) виртуальной машины Java, а также разработан и реализован собственный алгоритм верификации объектной памяти на языке программирования C++, основанный на использовании собственных аллокаторов. Данный верификационный алгоритм обходит все объекты кучи, после чего применяет два этапа проверки ссылок на другие объекты в объектной памяти виртуальной машины специальными верификационными функциями. На первом этапе проверяется принадлежность ссылки к области памяти выделенной определённым аллокатором (специальной компонентой виртуальной машины, которая занимается выделением памяти под её различные структуры). На втором этапе, зная к какой области памяти, выделенной определённым аллокатором, принадлежит проверяемая ссылка, определяется непосредственно корректной данной ссылки. Данные этапы верификатора можно применять до и после работы сборщика мусора по опциям, подаваемым при запуске виртуальной машины. Такой алгоритм позволяет проверить правильность работы с виртуальной памятью как сборщика мусора, так и других компонент виртуальной машины, которые взаимодействуют с объектной кучей. Представленная работа является частью большого проекта компании Huawei Technologies Co. Ltd. по разработке виртуальной машины Java для мобильных устройств. Разработанный инструмент уже используется разработчиками компании для отлаживания ошибок времени выполнения и выявления недочётов работы с объектами в исходном коде программы. Для проверки правильности верификации реализован набор unit-тестов, покрывающий многочисленные ситуации работы с памятью виртуальной машины как правильные, так и заведомо ошибочные. Данный верификатор может совершенствоваться для увеличения скорости работы виртуальной машины и уменьшение воздействия на её работу.

Ключевые слова: верификация объектной кучи, управление памятью, виртуальные машины, Java, аллокаторы, сборщик мусора

Благодарности: автор выражает особенные благодарности доктору технических наук, профессору, заведующему лабораторией открытых информационных технологий факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова Владимиру Александровичу Сухомлину за методические рекомендации и ценные советы в подготовке статьи, а также senior-разработчикам Huawei Technologies Co., Ltd Дмитрию Николаевичу Трубенкову и Александру Николаевичу Емеленко за оказанную помощь и поддержку при проведении данного исследования.

Автор заявляет об отсутствии конфликта интересов.

Для цитирования: Петров, И. А. Верификация кучи памяти объектов виртуальной машины / И. А. Петров. – DOI 10.25559/SITITO.17.202103.603-612 // Современные информационные технологии и ИТ-образование. – 2021. – Т. 17, № 3. – С. 603-612.

Original article

Object Memory Verification in Virtual Machines

I. A. Petrov
Lomonosov Moscow State University, Moscow, Russian Federation
1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation
petrov.igor.a@gmail.com

Abstract

In this paper we discuss object memory heap verification algorithms of the Java virtual machine and developed and implemented a proprietary object memory verification algorithm. This verification algorithm iterates over all heap objects, after which it applies two stages of checking references to other objects in the object memory of the virtual machine with special verification functions. At the first stage, it is checked that the reference to references to the memory area allocated by some allocator (a special component of the virtual machine that allocates memory for its various structures). At the second stage, knowing to which memory area allocated by some allocator the checked reference belongs, it is determined directly by the correct given reference. These stages of the verifier can be applied before and after the garbage collector by the options given when starting the virtual machine. This algorithm allows us to check that the garbage collector and other virtual machine components that interact with the object heap are working correctly with virtual memory. The developed tool is already used by the company's developers to debug run-time errors and identify shortcomings in working with objects in the program source code. To check the correctness of verification, a set of unit tests was implemented that covers many situations of working with the memory of a virtual machine, both correct and deliberately erroneous. This verifier can be improved to increase the speed of the virtual machine and reduce the impact on its operation.

Keywords: Object memory verification, virtual machines, garbage collectors, memory management, Java, allocators

Acknowledgements: The author expresses special gratitude to Vladimir Sukhomlin, Dr. Sci. (Tech.), Professor, Head of the Open Information Technologies Lab of the Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University, for recommendations and advice in preparing the article, as well as senior developers of Huawei Technologies Co., Ltd to Dmitrii Trubenkov and Aleksandr Emelenko for their help and support during this study.

The author declares no conflict of interest.

For citation: Petrov I.A. Object Memory Verification in Virtual Machines. *Sovremennye informacionnye tehnologii i IT-obrazovanie = Modern Information Technologies and IT-Education.* 2021; 17(3):603-612. DOI: https://doi.org/10.25559/SITITO.17.202103.603-612

1. Введение

Виртуальная машина, будь то виртуальная машина Java, C# или JavaScript – это очень сложная архитектура, множество структур данных и взаимодействующих между собой разнообразных компонент. При разработке новых виртуальных машин возникает множество вопросов, проблем и, разумеется, методов их решения.

В статье подразумевается взаимодействие с виртуальной машиной Java, предназначенной для работы на мобильных устройствах[1]. Такое использование накладывает дополнительные ограничения на параметры виртуальной машины, такие как размер используемой памяти, количество потоков исполнения и некоторые другие.

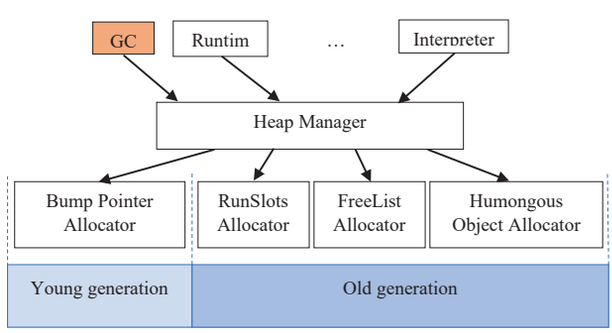
В программах на языке программирования Java программист не заботится о выделении и освобождении памяти под объекты. Эту задачу на себя берёт виртуальная машина. Важными частями виртуальной машины Java являются сборщики мусора (Garbage Collector или GC) и аллокаторы[2] [1-3]. Сборщик мусора взаимодействует со многими компонентами виртуальной машины, например: компилятор, runtime, интерпретатор и другие. Разработка GC является трудоёмкой задачей и во время его реализации возникает огромное множество ошибок, которые разработчикам виртуальной машины сложно находить. Сборщики мусора обычно разделяют на два вида: с подсчётом ссылок (reference counting) и построенные на основе достижимости объектов (tracing). В современных реализациях сборщики мусора, основанные на подсчёте ссылок, почти не используются. В нашем случае используется сборщик мусора, основанный на достижимости объектов [4]. Для сборщиков мусора на основе достижимости объектов существуют различные оптимизации, которые, возможно, придётся учитывать [5-8]. Общий алгоритм работы такого сборщика мусора можно описать следующим образом:

1. В начале собираются так называемые корневые объекты (GC Root): локальные, статические переменные, основной поток и некоторые другие;
2. Из корневых объектов итеративно происходит обход остальных объектов, пока все объекта, на которые есть ссылки, не будут пройдены. При обходе объекты помечаются специальным образом как «живые» (reachable);
3. Все объекты, не помеченные как «живые», считаются «мёртвыми» (non-reachable), и сборщик мусора может их удалить из кучи.

Важно понимать, что сборщик мусора работает вместе с работой основного приложения, что может влиять на его производительность. Существуют различные стратегии GC. Большинство из них основано на так называемой «слабой гипотезе о поколениях»: большинство объектов умирают молодыми [9]. То есть, чем раньше был создан объект, тем больше вероятность того, что в скором времени он будет удалён [10]. Эта гипотеза хорошо показывает себя на практике, поэтому кучу памяти для хранения объектов разбивают на две части: молодое поколение (young generation) и старое поколение (old или tenured generation).

При создании объектов они помещаются в молодое поколение. После заполнения молодого поколения начинается «малая сборка мусора», при которой удаляются мёртвые объекты из молодого поколения. Если же места в молодом поколении недостаточно (например, все объекты в молодом поколении остались живыми или новый выделяемый объект слишком большой), то живые объекты перемещаются в старое (old) поколение. При этом надо помнить, что сборщик мусора должен обновить ссылки на перемещённые объекты, ведь они теперь расположены в другой области памяти.

Объекты в обоих поколениях выделяются специальными компонентами управления виртуальной памятью, называемыми аллокаторами. Аллокаторы следят за выделением, распределением и удалением нативной памяти, имея внутри себя специальные структуры данных для работы с ней. Выделение объектов различными частями виртуальной машины происходит через специальную её часть, названную Heap Manager (менеджер кучи), который и управляет выделением и освобождением памяти[3] [1; 2; 11]. Как отмечалось ранее освобождением памяти, занимаемой мёртвыми объектами, управляет сборщик мусора (GC) (Рисунок 1).



Р и с. 1. Управление памятью объектов в виртуальной машине на примере сборщика мусора с поколениями

F i g. 1. Managing object memory in a virtual machine using a generational garbage collector as an example

Стоит отметить, что множество аллокаторов выбрано не случайным образом. Каждый аллокатор предназначен для выделения и освобождения объектов соответствующего размера в соответствующей области памяти.

До и после фазы работы сборщика мусора куча памяти виртуальной машины должна находиться в консистентном состоянии. Под консистентным состоянием кучи в данной статье будет пониматься, что любая ссылка из любого объекта должна

[1] Gosling J., Joy B., Steele G., Bracha G., Buckley A. The Java® Language Specification [Электронный ресурс] // Oracle. Java SE 8 Edition. Oracle America, Inc., 2015. 768 p. URL: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf (дата обращения: 06.08.2021).

[2] Memory Management in the Java HotSpot™ Virtual Machine [Электронный ресурс] // Oracle. Sun Microsystems, Inc., 2006. 21 p. URL: https://www.oracle.com/technetwork/java/javase/memorymanagement-whitepaper-150215.pdf (дата обращения: 06.08.2021).

[3] Jones R., Hosking A., Moss E. The Garbage Collection Handbook: The Art of Automatic Memory Management. 2nd Ed. Chapman & Hall/CRC, 2011. 511 p.

либо указывать на другой объект в куче, либо быть null.

Вследствие ошибок при разработке в исходном коде виртуальной машины, куча памяти может приходить в неконсистентное состояние. Типы ошибок могут разные, например:

1. До стадии сборки мусора какой-либо компонент виртуальной машины изменил ссылку на некоторый объект, но не уведомил об этом GC, однако сборщик мусора в отдельном потоке начал работу с этим объектом. Получается, что сборщик мусора может ошибочно удалить или переместить объект, неправильно обновить ссылки. При этом сам алгоритм сборки мусора может быть написан корректно, просто сборщик мусора не был уведомлён другой компонентой виртуальной машины об обновлении объекта;
2. Ошибка может крыться в самом сборщике мусора, и он неправильно обработает объекты, что приведёт к неконсистентности кучи памяти.

Такие ошибки разработчикам находить крайне сложно, поскольку ошибочное поведение может проявиться через несколько стадий, а может не проявиться во время тестирования вообще. Поэтому разработка инструмента, позволяющего автоматически определять консистентность кучи памяти, позволит повысить качество и скорость отладки, а следовательно, и качество всей разработки виртуальной машины.

Дополнительно стоит отметить, что поскольку целевыми платформами данной виртуальной машины являются мобильные устройства, отладка на них сама по себе является сложной задачей. А наличие инструмента, который можно включить по специальной опции не зависимо от целевой платформы, ускорит поиск ошибок как на мобильном устройстве, так и при отладке исходного кода на серверных компьютерах.

2. Формальная постановка задачи верификации объектной памяти

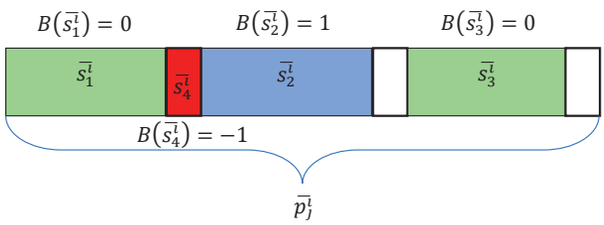
Управлением памятью виртуальной машины занимаются аллокаторы. Каждому *аллокатору предоставляются через взаимодействие с операционной системой определённые пулы памяти (некоторые непрерывные пространства памяти). В этих пулах памяти аллокаторы организуют свои структуры и предоставляют необходимую память для нужд виртуальной машины.*

Задача верификации кучи памяти состоит в следующем: определить являются ли все ссылки из объектов (ссылки по своей сути представляют адреса памяти) объектами в куче *(heap)* виртуальной машины.

То есть на вход работы алгоритма подаётся целочисленное неотрицательное число (потенциальная ссылка на другой объект). Необходимо каким-то образом понять является ли это число адресом памяти, который выделил один из аллокаторов.

Для начала введём понятие отрезка памяти. *Отрезком памяти* $\bar{s}$ *длины* n будем называть упорядоченное по возрастанию множество натуральных чисел вида $\bar{s} = \{m, m+1, \dots, m+n-1\}$, где $m, n \in \mathbb{N}$. При этом число m называется началом отрезка, а число $m+n-1$ концом отрезка. *Индексом* $\bar{s}[i]$ *отрезка памяти* $\bar{s}$, называется число $m+i$. $|\bar{s}| \stackrel{\text{def}}{=} n$.

Пусть имеется множество аллокаторов $A = \{A_1, \dots, A_k\}$, где каждому аллокатору A_i соответствует множество взаимно непересекающихся пулов памяти $P_i = \{\bar{p}_1^i, \dots, \bar{p}_{h(i)}^i\}$ $(P = \{P_1, \dots, P_k\})$. Каждый пул памяти представляется отрезком памяти $\bar{p}_j^i$. Каждый аллокатор использует пулы для своих внутренних структур. Пусть множество памяти, выделенной под данные внутренние структуры аллокатора A_i с множеством пулов P_i, определяется множеством взаимно непересекающихся отрезков памяти $S_i = \{\bar{s}_1^i, \dots, \bar{s}_{h(i)}^i\}$, где $S_i = \{\bar{s}_1^i, \dots, \bar{s}_{h(i)}^i\}$. При этом каждая структура $\bar{s}_j^i$ может находиться в одном из трёх состояний: свободна (0), занята (1) или «специальная структура» (-1).



Р и с. 2. Пример пула памяти аллокатора

F i g. 2. An example of an allocator memory pool

Свободный структура (0) – это свободное пространство памяти, которое используется аллокатором для выделения памяти под объекты (они же занятые структуры (1)) или под свои специальные структуры (-1) для организации работы с виртуальной памятью.

$$S \stackrel{\text{def}}{=} \bigcup_i \bigcup_{j=1}^{h(i)} \bar{s}_j^i,$$

S – множество всех отрезков памяти, которые используют аллокаторы. Пусть задана функция $B: S \to \{0, \pm 1\}$, определяющая свободен отрезок, занят или используется для специальной структуры.

$$S_i^b \stackrel{\text{def}}{=} \bigcup_{j, B(\bar{s}_j^i)=b} \bar{s}_j^i, b \in \{0, \pm 1\}, \qquad S^b = \bigcup_i \bigcup_{j, B(\bar{s}_j^i)=b} \bar{s}_j^i$$

Пусть также имеется множество $R = \{r_1, \dots, r_t\}$ ссылок на объекты. Они получаются из значений полей, которые уже выделены аллокаторами и лежат по некоторым адресам $\bar{s}_i^j[l]$, где $0 < l < |\bar{s}_i^j|$. Получается, что консистентность кучи памяти будет означать следующее:

$\forall r \in R\ \exists i, j \in \mathbb{N}$: $\bar{s}_j^i[0] = r$ && $B(\bar{s}_j^i) = 1$

Таким образом, необходимо реализовать алгоритм, который для заранее известного множества аллокаторов A и заданного распределения структур S_i определяет консистентность кучи объектов виртуальной машины:

$$Verifier(A, P, S) = \begin{cases} True\ if\ \forall r \in R\ \exists i, j \in \mathbb{N}: \bar{s}_j^i[0] = r\ and\ B(\bar{s}_j^i) = 1 \\ False, \quad otherwise \end{cases}$$

3. Алгоритм верификации объектной памяти

В данной главе описан алгоритм разработанного верификатора для проверки консистентности кучи памяти объектов виртуальной машины. Шаги данного алгоритма в общем виде выглядят следующим образом:

1. Все этапы происходят до начала и после конца работы GC, когда все потоки остановлены, чтобы не влиять на работу верификатора;
2. Для каждого аллокатора $A_i \in A$ проходим по всем его занятым структурам $\overline{s_j^i} \in S_i^1$;
3. Для каждой такой структуры выбираем ссылки на объекты, которые не имеют значение null, получая таким образом список проверяемых ссылок $R_i \subseteq R$;
4. Для каждой проверяемой ссылки $r \in R$ устанавливаем для всех аллокаторов из A (для каждого аллокатора своя проверка) является ли она в нём корректной.

Самыми нетривиальными частями работы алгоритма являются функции проверка для каждого аллокатора, поскольку для этого нужно учитывать их внутреннюю структуру и добавлять своё специфичное поведение в их работу. К счастью, разработкой аллокаторов, как и всей виртуальной машины занимается одна команда разработчиков, поэтому у нас есть возможность добавлять новое специальное поведение во внутренние структуры виртуальной машины.

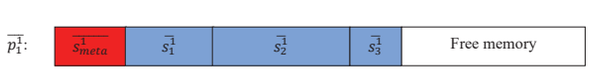
Чтобы описать действующий верификатор кучи необходимо описать как работают аллокаторы, реализованные в виртуальной машине, и как встраивать в них новое поведение. Далее описаны основные принципы работы таких аллокаторов.

3.1 Аллокаторы в виртуальной машине

3.1.1 Bump Pointer аллокатор

Bump Pointer аллокатор используется как аллокатор с быстрым выделением памяти под объект и быстрым «мгновенным» освобождением памяти всех объектов [12-14]. Продемонстрируем его внутреннюю структуру.

Как отмечалось ранее, данный аллокатор используется для объектов в молодом поколении. Для молодого поколения используется один пул памяти, поэтому $P_{bump} = P_1 = \{\overline{p_1^1}\}$. В начале пула лежит некоторая метаинформация о нём (например, размер занятой в пуле под объекты памяти). Далее при запросах на выделение объекта, память под них выделяется последовательно (Рисунок 3).



Р и с. 3. Структура Bump Pointer аллокатора
F i g. 3. Bump Pointer Allocator Structure

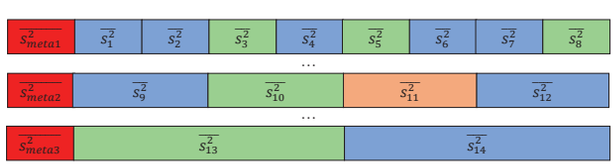
Удаление объектов из молодого поколение происходит сразу на стадии большой сборки мусора, таким образом всё занятое под объекты пространство очищается сразу. На самом деле для оптимизации многопоточных выделений объектов в конце пула для Bump Pointer аллокатора, организуется специальная структура, называемая TLAB, но её обработка в верификаторе идентична обработке в основном пространстве памяти, поэтому здесь данная структура будет опущена.

3.1.2 Run of Slots аллокатор

Run of Slots аллокатор используется для выделения и освобождения объектов «регулярного» размера [15; 16]. Это небольшие объекты, которые часто встречаются в мобильных приложениях на Java. Данный аллокатор использует несколько связанных в двухсвязный список пулов памяти для объектов разного размера. Каждый пул имеет структуру с метаинформацией, а остальная часть пула разбивается на части одинакового размера (обычно выбирается разбиение по степеням 2-ки). Такой пул и называется «Run-of-Slots». Таким образом, есть пулы с разбиением по 8-байтным частям (в них размещаются объекты размером не более 8-байт), 16-байтным частям (в них размещаются объекты размером не более 16-байт) и так далее (Рисунок 4). Каждая часть либо свободна, либо занята, за исключением самый первой с метаинформацией («специальная структура»).

Для определения занятости блока для каждого Run-of-Slots существует специальная структура – Bitmap, в которой каждому свободному блоку сопоставляется бит 0, а каждому занятому блоку бит 1 [17; 18]. Определим её следующим образом:

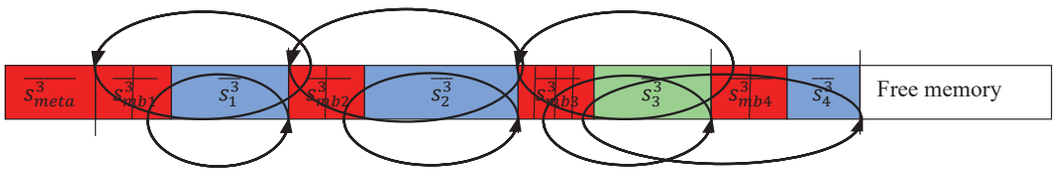
$$Bitmap_2(m) = \begin{cases} 0, m \in \overline{s_j^2} \in S_2^0 \\ 1, otherwise \end{cases}$$



Р и с. 4. Структура Run of Slots аллокатора
F i g. 4. Run of Slots Allocator Structure

3.1.3 Free List аллокатор

Free List аллокатор используется для выделения и удаления объектов большого размера [19-22]. Такими объектами могут являться массивы, строки или классы с большим количеством полей. В данном аллокаторе пулы разбиваются по принципу размеров объектов, для которых они предназначены при помощи специальной структуры, называемой Segregated List [23]. Подробно на ней не будем останавливаться, так как на алгоритм верификации это не влияет. Каждый пул в данном аллокаторе представляет из себя список специального вида (Рисунок 5). Каждый элемент списка представляет из себя блок из двух структур: метаинформация блока и память (свободная или занятая) под объект. Свободные блоки в метаинформации содержат указатель на предыдущий и следующий блок, а также указатели на предыдущий и следующий свободные блоки памяти. Занятый блок в метаинформации содержит указатель на предыдущий блок и размер данного блока. Также все блоки с метаинформацией хранят значение о занятости соответствующего объектного блока.



Р и с. 5. Пример пула в Free List аллокаторе

F i g. 5. An example of a pool in a Free List allocator

Пулы памяти в Free List образуют двусвязный список при помощи указателей на следующий и предыдущий пул в метаинформации в пулах.

3.1.4 Humongous Object аллокатор

Humongous Object аллокатор используется для создания очень больших объектов. Примером такого объекта может служить очень большой массив элементов (например, больше 5 мегабайт). Для объекта такого большого размера выделяется целый пул памяти (Рисунок 6).

Р и с. 6. Пул памяти в Humongous Object аллокаторе

F i g. 6. Memory pool in the Humongous Object allocator

Для оптимизации работы с памятью, при удалении объекта освобождаемый пул не сразу возвращается в список свободных пулов, а резервируется аллокатором для дальнейшего использования. Для определения занятости пула используется Bitmap [24]:

$$Bitmap_4(m) = \begin{cases} 1, m \in \bar{s}_j^4 \in S_4^1 \\ 0, otherwise \end{cases}$$

3.2 Верификационные функции аллокаторов

Теперь, зная как устроены аллокаторы в виртуальной машине, можно попытаться разработать индивидуально для каждого аллокатора функцию, которая для заданной ссылки $r \in R$ определит, является ли она корректной.

Для каждого аллокатора опишем верификационные функции, которые возвращают значение true если заданная ссылка выделена данным аллокатором и является действующим объектом, и значение false иначе. Каждая верификационная функция состоит из двух частей: сначала происходит некоторая простая проверка на принадлежность заданной ссылки к области памяти, отведённой заданному аллокатору, затем в случае успеха, происходит более точная проверка, которая окончательно устанавливает корректность входной ссылки.

3.2.1 Верификационная функция Bump Pointer аллокатора

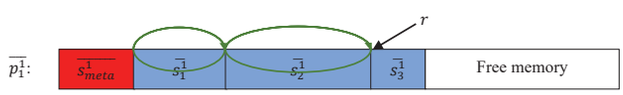
Итак, в Bump Pointer аллокаторе у нас есть единственный пул памяти $\bar{p}_1^1$, в начале которого имеется метаинформация об этом пуле, а затем последовательно друг за другом располагаются объекты. Для начала проверяется, что входная ссылка лежит в пределах объектного пространства пула, то есть:

$$FirstCheck = \begin{cases} true\ if\ \bar{s}_{first}^1[0] \le r \le \bar{s}_{last}^1[|\bar{s}_{last}^1| - 1] \\ false, otherwise \end{cases}$$

Поскольку в начале пула есть информация о размере пространства, занятого объектами, такая проверка является возможной.

Теперь необходимо проверить, что заданная ссылка действительно является адресом некоторого объекта в памяти. Для этого последовательно проверяется, что входная ссылка совпадает с началом какого-либо объекта из объектного пространства ():

0. Задаётся переменная $\bar{t} := \bar{s}_{first}^1$;
1. Если $\bar{t}[0] = r$, то заданная ссылка действительно является объектом в кучи, верификационная функция возвращает true;
2. Если $\bar{t}[0] \neq r$, то $\bar{t} := \bar{t} + Size(\bar{t})$;
3. Если новое значение $\bar{t}$ не лежит в пространстве объектов (то есть оказалось, что $\bar{t}[0] > \bar{s}_{last}^1[|\bar{s}_{last}^1| - 1]$), то входная ссылка не лежит в пространстве памяти, отведённому этому аллокатору, и верификационная функция вернёт значение false;
4. Иначе переходим к шагу 1.



Р и с. 7. Пример нескольких проходов алгоритма верификации в Bump Pointer аллокаторе

F i g. 7. An example of several passes of the verification algorithm in the Bump Pointer allocator

Данный алгоритм позволяет полностью убедиться в корректности проверяемой ссылки r для Bump Pointer аллокатора. Если ссылка указывает не в начало какого-либо отрезка, либо за пределы используемой в текущий момент памяти, то алгоритм вернёт ожидаемой значение false.

3.2.2 Верификационная функция Run of Slots аллокатора

В Run of Slots аллокаторе имеется несколько пулов. На первом этапе необходимо проверить, указывает ли входная ссылка внутрь какого-либо пула. Поскольку все пулы связаны между собой в двусвязный список и размеры пула известны, то простой итерацией по пулам можно выполнить проверку на принадлежность ссылки к памяти, отведённой аллокатору:

$$FisrtCheck = \begin{cases} true\ if\ \exists\ j = \overline{1, n(2)}, \bar{p}_j^2 \in P_2 \colon\ \bar{p}_j^2[0] < r < \bar{p}_j^2[|\bar{p}_j^2| - 1] \\ false, otherwise \end{cases}$$

Зная, что r находится внутри некоторого пула $\bar{p}_j^2 \in P_2$, при помощи $Bitmap_2$ можно определить занят ли слот памяти или свободен. Если занят, то остаётся только проверить, что r совпадает с началом этого слота. В общем виде вторая проверка

выглядит следующим образом:
$\sqsupset r \in \bar{s}_h^2$ – конкретный слот находится путём выравнивания r *по слоту.*

$$SecondCheck = \begin{cases} true\ if\ Bitmap_2(r) = 1\ and\ r = \bar{s}_h^2[0] \\ false, otherwise \end{cases}$$

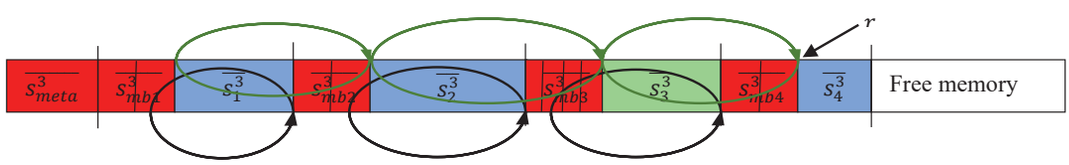
3.2.3 Верификационная функция Free List аллокатора

Как и в других аллокаторах, в Free List аллокаторе производится сначала простая проверка на наличие входной ссылки в какой-либо пул. В метаинформации пула хранится значение с размером данного пула, а значит, пройдя по всем пулам (они связаны двухсвязным списком), можно определить, указывает ли ссылка внутрь какого-либо пула, принадлежащего пространству памяти Free List аллокатора.

$$FisrtCheck = \begin{cases} true\ if\ \exists\ j = \overline{1}, n(3), \bar{p}_j^3 \in P_3 \colon\ \bar{p}_j^3[0] < r < \bar{p}_j^3[|\bar{p}_j^3| - 1] \\ false, otherwise \end{cases}$$

На момент начала второй проверки мы знаем пул, в котором потенциально хранится объект, на который указывает входная ссылка r. В этом пуле и будет проходить итеративный процесс поиска нужного объекта (Рисунок 8):

0. Задаётся переменная $\bar{t} := \bar{s}_{first}^3$;
1. Если текущий участок памяти занят, то есть $B(\bar{t}) = 1$, то происходит проверка того, что входная ссылка указывает на начало соответствующего объектного блока: $\bar{t}[0] = r$;
2. Если проверка прошла успешно, то верификационная функция возвращает значение true;
3. Если же проверяемая ссылка не указывает на начло текущего объекта, то, имея указатель на следующий блок, пытаемся перейти, собственно, к следующему блоку;
4. Если указатель на следующий блок пуск (равен nullptr), то это значит, что блоки в данном пуле закончились и для заданной ссылки не существует соответствующего объекта в данном пуле, верификационная функция возвращает значение false;
5. Если следующий блок существует, то обновляем, значение переменной $\bar{t} := NextBlock(\bar{t})$ и переходим к шагу 1.



Р и с. 8. Пример нескольких проходов алгоритма верификации в Free List аллокаторе

F i g. 8. An example of several passes of the verification algorithm in the Free List allocator

Представленный алгоритм всегда правильно определяет наличие объекта, в случае если проверяемая ссылка r является корректной, или некорректность входной ссылки, так как проходит по всем блокам внутри пула.

3.2.4 Верификационная функция Humongous Object аллокатора

Осталось разобраться в принципе работы верификационной функции в Humongous Object аллокаторе. Сначала проверятся, что входная ссылка r лежит внутри некоторого пула. Это проверятся при помощи описанной выше функции $Bitmap_4$:

$$FirstCheck = \begin{cases} true\ if\ Bitmap_4(r) = 1 \\ false, otherwise \end{cases}$$

Теперь есть уверенность, что лежит в занятом пуле. Поскольку весь пул состоит из одного объекта, а размер блока с метаинформацией всегда имеет фиксированное значение, то остаётся сравнить значение входной ссылки со значением начала занятого отрезка памяти под объект в этом пуле. В общем виде это выглядит следующим образом:

$$\sqsupset r \in \bar{s}_h^4 \in S_4^1$$

$$SecondCheck = \begin{cases} true\ if\ r = \bar{s}_h^4[0] \\ false, otherwise \end{cases}$$

3.3 Вывод

Все выше представленные верификационные функции полностью решают поставленную задачу. Встроенный внутрь аллокаторов функционал и использование их метаинформации позволило создать алгоритм, который определяет корректность множества ссылок:

1. Пусть дано множество ссылок R, которые нужно проверить;
2. Для каждого $r \in R$ последовательно проверяем корректность ссылки при помощи верификационных функций каждого аллокатора $A_i \in A$;
3. Если при очередной проверке на каком-нибудь аллокаторе, результат оказался равен true, то переходим к проверке следующей ссылки из множества R;
4. Если для всех аллокаторов хотя бы одна ссылка вернула значения false, то сообщается о неконсистентности кучи памяти виртуальной машины.

3.4. Реализация верификатора

Реализация верификатора представляет из себя встроенные во внутренние аллокаторы виртуальной машины верификационные функции, написанные на языке программирования C++.

Детали реализации можно посмотреть в открытом репозитории разрабатываемой виртуальной машины: https://gitee.com/openharmony/ark_runtime_core/blob/master/runtime/mem/heap_verifier.cpp

Реализация каждой верификационной функции включает в себя два этапа:

1. Простая проверка на наличие проверяемой ссылки в области памяти, отведённой аллокатору, чтобы не применять более сложные проверки для участков памяти, в ко-

торых заведомо нет целевого объекта;
2. Более сложная проверка для участка памяти, в котором потенциально находится целевой объект.

Для простой проверки в каждый аллокатор добавлена функция ContainObject, имеющая следующую сигнатуру:

```
bool ContainObject(const void * potentialReference) {
…
}
```

Данная функция возвращает булевское значение, где true означает, что корректность ссылки необходимо проверять в данном аллокаторе, а false означает, что в данном аллокаторе точно не содержится объекта, соответствующего проверяемой ссылке.

Для более сложной проверки используется встроенная в аллокаторы функция IsLive со следующей сигнатурой:

```
bool IsLive(const void * potentialReference) {
…
}
```

Данная функция также возвращает булевское значение, где true означает, что данный аллокатор содержит в себе объект, на который указывает потенциальная ссылка, а false означает, что в данном аллокаторе потенциальная ссылка не указывает ни на один объект, что говорит о неконсистентности объектной кучи памяти.

4. Тестирование верификатора

В данной главе представлены детали тестирования верификационных функций для каждого аллокатора. Для проверки качества верификатора использовался фреймворк от Google, называемый googletest. Данный фреймворк позволяет удобно писать unit-тесты для тестирования отдельных компонент.

Для тестирования верификационных функций был создан набор unit-тестов, которые должны проверят работу всего верификатора.

Для того, чтобы убедиться, что верификатор работает корректно, необходимо проверить несколько сценариев. Общий подход состоит в следующем: каждый из заданных аллокаторов выделяет несколько объектов. В unit-тестах мы запоминаем по каким адресам аллокаторы расположили заданные объекты. Далее для каждого такого адреса применим специальную функцию, которой на вход подаётся корректный адрес объекта и его размер, а на выход заведомо некорректный адрес, формируя его как адрес в середину этого объекта, тем самым получая некорректный адрес объекта.

После создания объектов и формирования заведомо некорректных адресов происходит проверка, что для адресов, которые получены из самих аллокаторов верификационные функции вернули значение true, а для некорректных адресов значение false. Результаты представлены в Таблице 1.

Таблица 1. **Результаты для корректных и некорректных адресов**
Table 1. **Results for correct and incorrect addresses**

Аллокатор	Корректные адреса	Некорректные адреса
Bump Pointer	true	false
Run of Slots	true	false
Free List	true	false
Humongous	true	false

Как можно заменить верификатор полностью правильно обработал как корректные ссылки, так и некорректные ссылки, что демонстрирует его работоспособность для ссылок, которые потенциально находятся внутри пула.

Для тестирования также необходимо проверить ссылки, которые не указывают в области памяти, отведённые аллокаторами в принципе. Для такого тестирования выбраны ссылки, которые указывают в самые начало и конец, доступной виртуальной машине памяти, где объекты точно не могут располагаться.

Таблица 2. **Результаты тестирования верификатора для начала и конца памяти VM**
Table 2. **Verifier test results for the beginning and end of VM memory**

Аллокатор	Начало памяти VM	Конец памяти VM
Bump Pointer	false	false
Run of Slots	false	false
Free List	false	false
Humongous	false	false

И снова тестирование дало верные результаты.

Также в общий набор тестов добавлены тесты с чуть более хитрым сценарием. У нас имеются объекты, которые выделили аллокаторы. Проведём тестирование, в котором аллокаторы сначала удалят эти объекты, а потом адреса удалённых объектов подадим на вход алгоритму работы верификатора. Очевидно, что адреса на удалённые объекты не могут быть корректными.

Таблица 3. **Тестирование с удалёнными объектами**
Table 3. **Testing with deleted objects**

Аллокатор	Удалённые объекты
Bump Pointer	false
Run of Slots	false
Free List	false
Humongous	false

В очередной раз верификатор показал правильные результаты работы.

Тестирование верификатора показало, что верификатор способен определять как корректные, так и некорректные ситуации при работе с объектной кучей виртуальной машины Java.

Верификатор продемонстрировал правильность своей работы в сценариях с объектами внутри областей памяти аллокаторов и вне этих областей, что говорит о качестве его проверок.

5. Заключение

В рамках данной статьи разработан и реализован алгоритм верификации объектной кучи виртуальной машины Java на основе использования внутренних аллокаторов.

Для этого сформулирована формальная постановка задачи. Разработан алгоритм, позволяющий определять консистентность кучи памяти объектов, который основан на использовании внутренних разработанных аллокаторов. Созданный алгоритм представляет из себя набор функций внутри отдельно взятого аллокатора, что позволяет учитывать специфику работы аллокаторов и применять необходимые оптимизации для ускорения работы верификатора, что может являться критичным при работе на мобильных устройствах.

Разработанный алгоритм не зависит от целевой платформы, на которой работает виртуальная машина, что делает возможным отладку программ как на серверных компьютерах, так и на мобильных устройствах.

Для проверки работоспособности верификатора разработан и реализован набор тестов, покрывающий множество ситуаций работы с объектной кучей. Тестирование продемонстрировало, что разработанный верификатор действительно находит ошибочные ситуации, а также верно определяет консистентность кучи памяти в случаях, когда объектная куча корректна. Реализованный верификатор уже используется разработчиками в Huawei Technologies Co. Ltd. для поиска ошибок в исходном коде.

Дальнейшие направления работы представляют расширение возможностей верификатора (например, для аллокатора, основанного на регионах [25]) и оптимизацию скорости его работы. Для верификации объектной кучи памяти можно применять проверки не только в перед и после запуска сборщика мусора, но и его отдельных стадий. В последующих работах будет рассмотрена возможность таких проверок.

References

[1] Lee S.-M., Kim D.-G., Shin D.-R. General purpose hardware abstraction layer for multiple virtual machines in mobile devices. *Proceedings of the 11th International Conference on Advanced Communication Technology (ICACT '09)*. IEEE Press, Phoenix Park; 2009. p. 362-364. (In Eng.)

[2] Sivapriyan R., Sakshi N., Mahesh K. Intelligent Garbage Collection Application for Smart City. *Proceedings of the International Conference on Advancements in Electrical, Electronics, Communication, Computing and Automation (ICAECA).* IEEE Press, Coimbatore, India; 2021. p. 1-6. (In Eng.) DOI: https://doi.org/10.1109/ICAECA52838.2021.9675572

[3] Rodriguez-Rivera G. Conservative garbage collection for general memory allocators. *Proceedings of the 2nd international symposium on Memory management (ISMM '00).* Association for Computing Machinery, New York, NY, USA; 2000. p. 71-79. (In Eng.) DOI: https://doi.org/10.1145/362422.362464

[4] Agesen O., Detlefs D., Moss E. Garbage Collection and Local Variable Type-Precision and Liveness in Java Virtual Machines. *ACM SIGPLAN Notices*. 1998; 33(5):269-279. (In Eng.) DOI: https://doi.org/10.1145/277652.277738

[5] Wimmer C., Mössenböck H. Automatic Feedback-Directed Object Inlining in the Java HotSpot™ Virtual Machine. *Proceedings of the 3rd International Conference on Virtual Execution Environments (VEE '07).* Association for Computing Machinery, New York, NY, USA; 2007. p. 12-21. (In Eng.) DOI: https://doi.org/10.1145/1254810.1254813

[6] Shaham R., Kolodner E.K., Sagiv M. On effectiveness of GC in Java. *Proceedings of the 2nd international symposium on Memory management (ISMM '00).* Association for Computing Machinery, New York, NY, USA; 2000. p. 12-17. (In Eng.) DOI: https://doi.org/10.1145/362422.362430

[7] Boehm H.-J. Reducing garbage collector cache misses, On effectiveness of GC in Java. *Proceedings of the 2nd international symposium on Memory management (ISMM '00).* Association for Computing Machinery, New York, NY, USA; 2000. p. 59-64. (In Eng.) DOI: https://doi.org/10.1145/362422.362438

[8] Bacon D.F., Fink S.J., Grove D. An efficient parallel heap compaction algorithm. *Proceedings of the 16th European Conference on Object-Oriented Programming (ECOOP '02).* Springer-Verlag, Berlin, Heidelberg; 2002. p. 111-132. (In Eng.) DOI: https://doi.org/10.5555/646159.680023

[9] Jones R.E., Ryder C. A Study of Java Object Demographics. *Proceedings of the 7th international symposium on Memory management (ISMM '08).* Association for Computing Machinery, New York, NY, USA; 2008. p. 121-130. (In Eng.) DOI: https://doi.org/10.1145/1375634.1375652

[10] Barrett D.A., Zorn B.G. Using lifetime predictors to improve memory allocation performance. *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation (PLDI '93).* Association for Computing Machinery, New York, NY, USA, 1993. p. 187-196. (In Eng.) DOI: https://doi.org/10.1145/155090.155108

[11] Singh T. The Hotspot Java Virtual Machine: Memory and Architecture. *International Journal of Allied Practice, Research and Review*. 2014. p. 60-64. (In Eng.) DOI: https://doi.org/10.13140/2.1.2442.7206

[12] Lattner C., Adve V. Automatic Pool Allocation: Improving Performance by Controlling Data Structure Layout in the Heap. *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (PLDI '05).* Association for Computing Machinery, New York, NY, USA; 2005. p. 129-142. (In Eng.) DOI: https://doi.org/10.1145/1065010.1065027

[13] Novark G., Berger E.D. DieHarder: securing the heap. *Proceedings of the 17th ACM conference on Computer and communications security (CCS '10).* Association for Computing Machinery, New York, NY, USA; 2010. p. 573-584. (In Eng.) DOI: https://doi.org/10.1145/1866307.1866371

[14] Silvestro S., Liu H., Crosser C., Lin Zh., Liu T. FreeGuard: A Faster Secure Heap Allocator. *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS '17).* Association for Computing Machinery, New York, NY, USA; 2017. p. 2389-2403. (In Eng.) DOI: https://doi.org/10.1145/3133956.3133957

[15] Soares A.M.M., de Sousa R.T.Jr. A Technique for Extraction and Analysis of Application Heap Objects within Android Runtime (ART). *Proceedings of the 3rd International Conference on Information Systems Security and Privacy (ICISSP).* Porto, Portugal; 2017. p. 147-156. (In Eng.) DOI: https://doi.org/10.5220/0006204101470156

[16] Bhatia R., Saltaformaggio B., Yang S.J., Ali-Gombe A., Zhang X., Xu D., Richard III G.G. Tipped Off by Your Memory Allocator: Device-Wide User Activity Sequencing from Android Memory Images. *Proceedings on Network and Distributed Systems Security (NDSS) Symposium 2018*. San Diego, CA, USA; 2018. 15 p. (In Eng.) DOI: http://dx.doi.org/10.14722/ndss.2018.23324

[17] Ogasawara T. An algorithm with constant execution time for dynamic storage allocation. *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*. IEEE Press, Tokyo, Japan; 1995. p. 21-25. (In Eng.) DOI: https://doi.org/10.1109/RTCSA.1995.528746

[18] Wilson P.R., Johnstone M.S., Neely M., Boles D. Dynamic Storage Allocation: A Survey and Critical Review. In: Ed. by H. G. Baler. *Memory Management. IWMM 1995. Lecture Notes in Computer Science*. 1995; 986:1-116. Springer, Berlin, Heidelberg. (In Eng.) DOI: https://doi.org/10.1007/3-540-60368-9_19

[19] Masmano M., Ripoll I., Crespo A., Real J. TLSF: a new dynamic memory allocator for real-time systems. *Proceedings of the 16th Euromicro Conference on Real-Time Systems (ECRTS 2004).* IEEE Press, Catania, Italy; 2004. p. 79-88. (In Eng.) DOI: https://doi.org/10.1109/EMRTS.2004.1311009

[20] Johnstone M.S., Wilson P.R. The memory fragmentation problem: solved? *ACM SIGPLAN Notices*. 1999; 34(3):26-36. (In Eng.) DOI: https://doi.org/10.1145/301589.286864

[21] Bacon D.F., Cheng P., Rajan V.T. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for Java. *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems (LCTES '03).* Association for Computing Machinery, New York, NY, USA; 2003. p. 81-92. (In Eng.) DOI: https://doi.org/10.1145/780732.780744

[22] Grunwald D., Zorn B. CustoMalloc: efficient synthesized memory allocators. *Software: Practice and Experience*. 1993; 23(8):851-869. (In Eng.) DOI: https://doi.org/10.1002/spe.4380230804

[23] Masmano M., Ripoll I., Real J., Crespo A., Wellings A.J. Implementation of a constant-time dynamic storage allocator. *Software: Practice and Experience.* 2008; 38(10):995-1026. (In Eng.) DOI: https://doi.org/10.1002/spe.858

[24] Martínez-Bazan N., Ángel Águila-Lorente M., Muntés-Mulero V., Dominguez-Sal D., Gómez-Villamor S., Larriba-Pey J.-L. Efficient graph management based on bitmap indices. *Proceedings of the 16th International Database Engineering & Applications Sysmposium (IDEAS '12).* Association for Computing Machinery, New York, NY, USA; 2012. p. 110-119. (In Eng.) DOI: https://doi.org/10.1145/2351476.2351489

[25] Qian F., Hendren L. An adaptive, region-based allocator for java. *Proceedings of the 3rd international symposium on Memory management (ISMM '02).* Association for Computing Machinery, New York, NY, USA; 2002. p. 127-138. (In Eng.) DOI: https://doi.org/10.1145/512429.512446

Поступила 06.08.2021; одобрена после рецензирования 10.09.2021; принята к публикации 16.09.2021.
Submitted 06.08.2021; approved after reviewing 10.09.2021; accepted for publication 16.09.2021.

Об авторе:

Петров Игорь Андреевич, магистрант факультета вычислительной математики и кибернетики, ФГБОУ ВО «Московский государственный университет имени М. В. Ломоносова» (119991, Российская Федерация, г. Москва, ГСП-1, Ленинские горы, д. 1), **ORCID: https://orcid.org/0000-0003-1751-3139,** petrov.igor.a@gmail.com

Автор прочитал и одобрил окончательный вариант рукописи.

About the author:

Igor A. Petrov, Master student of the Faculty of Computational Mathematics and Cybernetics, Lomonosov Moscow State University (1 Leninskie gory, Moscow 119991, GSP-1, Russian Federation), **ORCID: https://orcid.org/0000-0003-1751-3139,** petrov.igor.a@gmail.com

The author has read and approved the final manuscript.