РЕФЕРАТ

Расчетно–пояснительная записка 24 с., 0 рис., 0 табл., 6 ист, 1 прил.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Оперативная память является одним из основных ресурсов любой вычислительной системы, требующим чёткого управления. Под памятью (memory) здесь и в дальнейшем 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 операций в языке 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.

Существуют 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 сбор-

ку мусора.

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

1. Провести анализ предметной области 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. Провести сравнительный анализ существующих решений 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 компьютера. Эти устройства включают в себя регистры процессора, кэш, ОЗУ, MMU (Memory Management Unit, блоки управления памятью) и вторичную (дисковую 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 подробно будет рассмотрено управление памятью с точки зрения операционной системы и приложений.

1.2. Управление памятью с точки зрения операционной системы

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 абстракцию, то есть в модель, а затем управлять этой абстракцией — и есть задача операционной системы. Та часть операционной системы, которая управ-

ляет иерархией памяти (или ее частью), называется **менеджером**, или **диспетчером**, **памяти** [1]. Он предназначен для действенного управления памятью и 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]

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

СТРУКТУРА АДРЕСНОГО ПРОСТРАНСТВА ПРОЦЕССА (ОПРЕДЕЛЕНИЕ КУЧИ)

Для обеспечения выполнения ???????????? 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 пере-

мещения 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.

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 перестает нуждаться в дополнительной памяти, оно может возвратить ее системе. Выделение памяти случайной длины в случайные моменты времени из общего пула памяти приводит к фрагментации и, вследствие этого, к неэффективному ее использованию. Дефрагментация памяти тоже является функцией операционной системы.

Ещё одна важная задача операционной системы — защита памяти. Она состоит в том, чтобы не позволить выполняемому процессу записывать или читать данные из памяти, назначенной другому процессу.

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.
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: [4]

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. [5]

1.3.2. Управление памятью

Менеджер памяти приложения должен учитывать следующие ограничения. [2]

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

Управление памятью приложения объединяет две взаимосвязанные задачи: выделение памяти (allocation) и её переиспользование (recycling), когда она больше не требуется. За выделение памяти отвечает **аллокатор** [6]. 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 менеджеру памяти приложения, что заданная область памяти может быть освобождена и переиспользована.

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 его поведение менеджера памяти либо путем выделения больших блоков 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.

1.3.2.2. Автоматическое управление памятью

Описание и подводка к сборке мусора

1.3.3. Сборка мусора

Понятие и определение сборщика мусора.
Определение неиспользуемого объекта в памяти?

1.3.3.1. Подсчёт ссылок

1.3.3.2. Трассирующая сборка мусора

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.

Язык C++ разрабатывался с расчетом на использование ручного управления памятью. Некоторые языковые возможности, такие как адресная арифметика и приведение типов указателей, затрудняют реализацию сборщиков мусора для этого языка. Несмотря на это, существует ряд подходов к сборке мусора для языка C++.

Одни из самых неприятных ошибок, которые портят жизнь программисту, это, безусловно, ошибки, связанные с управлением памятью. В таких языках, как C и C++, в которых управление памятью целиком возложено на программиста, львиная доля времени, затрачиваемого на отладку программы, приходится на борьбу с подобными ошибками.

Давайте перечислим типичные ошибки при управлении памятью (некоторые из них особенно усугубляются в том случае, если в программе существуют несколько указателей на один и тот же блок памяти):

Преждевременное освобождение памяти (premature free). Эта беда слу-

чается, если мы пытаемся использовать объект, память для которого была уже освобождена. Указатели на такие объекты называются висящими (dangling pointers), а обращение по этим указателям дает непредсказуемый результат.

Двойное освобождение (double free). Иногда бывает важно не перестараться и не освободить ненужный объект дважды.

Утечки памяти (memory leaks). Когда мы постоянно выделяем новые блоки памяти, но забываем освобождать блоки, ставшие ненужными, память в конце концов заканчивается.

Фрагментация адресного пространства (external fragmentation). При интенсивном выделении и освобождении памяти может возникнуть ситуация, когда непрерывный блок памяти определенного размера не может быть выделен, хотя суммарный объем свободной памяти вполне достаточен. Это происходит, если используемые блоки памяти чередуются со свободными блоками и размер любого из свободных блоков меньше, чем нам нужно.

Проблема особенно критична в серверных приложениях, работающих в течение длительного времени.

В программах, работающих в среде .NET, все вышеперечисленные ошибки никогда не возникают, потому что эти программы используют реализованное в CLR автоматическое управление памятью, а именно – сборщик мусора.

Работа сборщика мусора заключается в освобождении памяти, занятой ненужными объектами. При этом сборщик мусора также умеет «двигать» объекты в памяти, тем самым устраняя фрагментацию адресного пространства.

Все эти чудеса, которые творит сборщик мусора, возможны исключительно благодаря тому, что во время выполнения программы известны типы всех используемых в ней объектов. Другими словами, данные, с которыми работает программа, находятся под полным контролем среды выполнения и называются, соответственно, управляемыми данными (managed data).

2 Описание существующих решений

2.1. C#

2.1.1. Управление памятью

2.1.2. Алгоритм работы сборщика мусора

https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/fundamentals

https://learn.microsoft.com/en-us/dotnet/csharp/advanced-topics/performance/

https://learn.microsoft.com/en-us/dotnet/standard/garbage-collection/

https://www.bytehide.com/blog/garbage-collection-dotnet

https://prographers.com/blog/in-depth-memory-management-in-c

https://habr.com/ru/articles/44208/

https://habr.com/ru/articles/590475/

https://metanit.com/sharp/tutorial/8.1.php

https://medium.com/c-programming/c-memory-management-part-1-c03741c24e4b

https://medium.com/c-programming/c-memory-management-part-2-finalizer-dispose-d3b3e43c08d1

https://medium.com/c-programming/c-memory-management-part-3-garbage-collection-18faf118cbf1

2.2. Python

2.2.1. Управление памятью

2.2.2. Алгоритм работы сборщика мусора

Рассматриваем только реализацию Cython.

https://docs.python.org/3/c-api/memory.html

https://devguide.python.org/internals/garbage-collector/

https://stackify.com/python-garbage-collection/

https://realpython.com/python-memory-management/https://www.honeybadger.io/blog/memory-management-in-python/

https://medium.com/analytics-vidhya/python-memory-management-8854f4952ba0

https://scoutapm.com/blog/python-garbage-collection

https://www.javatpoint.com/python-memory-management

https://towardsdatascience.com/memory-management-and-garbage-collection-in-python-c1cb51d1612c

https://rushter.com/blog/python-garbage-collector/

https://scoutapm.com/blog/python-garbage-collection

2.3. Java

2.3.1. Управление памятью

2.3.2. Алгоритм работы сборщика мусора

https://www.amazon.com/dp/0137142528

https://www.amazon.com/dp/1420082795

https://en.wikipedia.org/wiki/Garbage-first_collector

https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage

https://www.oracle.com/technetwork/java/javase/tech/memorymanagement-whitepaper-1-150020.pdf

https://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html

https://docs.oracle.com/en/java/javase/11/gctuning/garbage-collector-implementation.html

https://docs.oracle.com/javase/8/docs/technotes/guides/vm/gctuning/toc.html

https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html

https://www.oracle.com/technetwork/tutorials/tutorials-1876574.html

https://openjdk.org/jeps/0

https://cyberleninka.ru/article/n/java-garbage-collectors/viewer

https://www.javatpoint.com/memory-management-in-java

https://opensource.com/article/22/7/garbage-collection-java

https://www.baeldung.com/java-choosing-gc-algorithm

https://www.baeldung.com/jvm-garbage-collectors

https://developers.redhat.com/articles/2021/11/02/how-choose-best-java-garbage-collector

https://developers.redhat.com/articles/2021/08/20/stages-and-levels-java-garbage-collection

https://howtodoinjava.com/java/garbage-collection/revisiting-memory-management-and-garbage-collection-mechanisms-in-java/

https://howtodoinjava.com/java/garbage-collection/all-garbage-collection-algorithms/

https://assets.ctfassets.net/oxjq45e8ilak/2K7aShH1CgWcUMc4Yqyg0g/b2606c497
prepared-to-g1-gc-or-evolution-of-the-g1-gc.pdf

https://habr.com/ru/articles/269621/

https://java-ru-blog.blogspot.com/2020/01/garbage-first-g1-java-vm.html

https://java-online.ru/garbage-collection.xhtml

2.4. Golang

2.4.1. Управление памятью

2.4.2. Алгоритм работы сборщика мусора

Мои конспекты в Obsidian

Лекция

https://habr.com/ru/companies/oleg-bunin/articles/676332/

https://go101.org/article/memory-block.html

https://go.dev/ref/mem

https://go.dev/doc/gc-guide

https://github.com/golang/proposal/blob/master/design/48409-soft-memory-limit.md (ALSO FOR JAVA)

https://github.com/golang/go/issues/51317

https://backendinterview.ru/goLang/memory.html

https://medium.com/safetycultureengineering/an-overview-of-memory-management-in-go-9a72ec7c76a8 https://medium.com/@ali.can/memory-optimization-in-go-23a56544ccc0

2.5. Haskell

2.5.1. Управление памятью

2.5.2. Алгоритм работы сборщика мусора

https://ru.hexlet.io/blog/posts/haskell-yazyk-pozvolyayuschiy-glubzhe-ponyat-programmirovanie-kak-on-ustroen-i-pochemu-ego-vybirayut-razrabotchiki

$https://wiki.haskell.org/GHC/Memory\_Management$

http://simonmar.github.io/bib/papers/parallel-gc.pdf

https://simonmar.github.io/bib/papers/ExploringBarrierToEntry.pdf

https://www.researchgate.net/publication/221032889_Parallel_generational-copying_garbage_collection_with_a_block-structured_heap

https://stackoverflow.com/questions/36772017/reducing-garbage-collection-pause-time-in-a-haskell-program/36779227

https://well-typed.com/blog/aux/files/nonmoving-gc/design.pdf

https://well-typed.com/blog/2021/03/memory-return/

https://pusher.com/blog/latency-working-set-ghc-gc-pick-two

https://www.channable.com/tech/lessons-in-managing-haskell-memory

https://pusher.com/blog/making-efficient-use-of-memory-in-haskell/

https://ro-che.info/articles/2017-08-06-manage-allocated-memory-haskell

3 Классификация существующих решений

3.1. Критерии сравнения алгоритмов распределения памяти

GC паузы...

Реализован ли алгоритм поколений в GC

3.2. Сравнительный анализ алгоритмов распределения памяти

3.3. Вывод

ЗАКЛЮЧЕНИЕ

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. Common Language Runtime (CLR) overview [Электронный ресурс]. — Режим доступа, URL: https://learn.microsoft.com/en-us/dotnet/standard/clr (дата обращения: 12.10.2023).
5. 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.
6. Allocation techniques [Электронный ресурс]. — Режим доступа, URL: https://www.memorymanagement.org/mmref/alloc.html#mmref-alloc (дата обращения: 12.10.2023).

ПРИЛОЖЕНИЕ А