СОДЕРЖАНИЕ

1 АННОТАЦИЯ

Сопрограммы (англ. coroutine) — программный модуль, особым образом организованный для обеспечения взаимодействия с другими модулями по принципу кооперативной многозадачности[1]. Выполнение сопрограммы может быть приостановлено в точках явного планирования и предано другому модулю. При этом будет сохранено полное состояние сопрограммы (включая стек, значения регистров и счётчик команд). Отличие сопрограмм от потоков операционной системы заключается в потреблении меньшего объема системных ресурсов – адресного пространства и памяти. Кроме того, переключение контекста сопрограммы требует меньших издержек, чем потока, поскольку осуществляется в пользовательском пространстве операционной системы средой исполнения языка.

Сопрограммы используются для реализации генераторов, итераторов, бесконечных списков, конечные автоматов внутри одной подпрограммы. Так же их можно использовать для упрощения написания асинхронного и неблокирующего кода. Сопрограммы будут полезны разработчикам многопоточных web-сервисов, поскольку позволяют уменьшить время отклика приложения из-за своих преимуществ, перечисленных выше. Они уже были реализованы во многих популярных языках программирования, таких как Go, Kotlin, C#, но на момент написания работы сопрограммы не реализованы в языке Java. Целю является создание работающего прототипа сопрограмм в языке программирования Java. Работа проводилась на базе HuaweiJDK, альтернативной реализации виртуальной машины Java. Для достижения поставленной цели необходимо было решить следующие задачи:

– Разработать тесты для сравнения производительности потоков и сопрограмм.
– Реализовать переключение сопрограмм.
– Поддержать трассировку ссылок объектов на стеках сопрограмм для сборки мусора.

– Сравнить производительность сопрограмм и потоков.

2 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

2.1 Что такое Сопрограммы?

Сопрограммы (англ. coroutine) — программный модуль, особым образом организованный для обеспечения взаимодействия с другими модулями по принципу кооперативной многозадачности [1]. Выполнение сопрограммы может быть приостановлено в точках явного планирования и предано другому модулю. Выполнение модуля может быть приостановлено в определённой точке и предано другой сопрограмме. При этом будет сохранено полное состояние сопрограммы (включая стек, значения регистров и счётчик команд).

Концепция сопрограмм не нова: впервые они появились в языках программирования Симула[3], Модула-2[2] в 1960-е — 1970-е годы и использовались как еще одно языковое средство для реализации итераторов, генераторов, бесконечных списков и так далее. В тот момент концепция не получила широкого распространения и в более поздних языках программирования, таких как Си, C++, Java, она не была применена. Если нужна альтернатива сопрограммаам, то использовались потоки. В случае языков Си и C++ можно применять системные функции для переключения контекста потока для реализации сопрограмм в виде библиотеки.

Сопрограммы по своему поведению, очень похожи на потоки операционной системы: они могут сохранять контекст[1] и при останавливать свое выполнения. Но есть ряд существенных отличий:

а) Переключение сопрограммы происходит в пользовательском пространстве операционной системы в отличие от потоков, которые управляются ядром ОС. Кроме того, сопрограммы являются объектом среды исполнения языка, что позволяет оптимизировать переключение контекста под конкретный язык и виртуальную машину. В сумме эти два фактора позволяют уменьшить накладные расходы на переключение потока

[1]Контекст - регистры и указатель на стек потока или сопрограммы.

выполнения. Как итог, системы, построенные на сопрограммаах, будут иметь лучшее время отклика, чем те, которые используют только потоки.

б) Сопрограммы как правило имеют меньший размер стека. Как говорилось в прошлом пункте, сопрограммы являются сущностями среды исполнения в отличии от потоков, а значит виртуальная машина имеет больший контроль над стеком первых. Потому появляется возможность выделять сравнительно небольшие стеки под сопрограммы. Если он переполняется, то создается новый стек большего размера и содержимое старого копируется в новый. Благодаря меньшему размеру стека, виртуальная машина способна создавать число сопрограмм, превосходящее по количеству потоков ОС на той же аппаратуре.

в) Известно, что при операциях ввода/вывода вызывающий поток блокируется операционной системой[7]. Это делается во избежание лишнего простоя процессора, который в момент работы вызова ввода/вывода может переключится на выполнение другой задачи, которую запланирует ядро ОС. Сопрограммаами же управляет среда исполнения языка, которая способна вытеснить сопрограмму, инициирующую операцию ввода вывода, избежав при этом блокировки потока. Если закрепить за потоком ядро процессора и запретить операционной системе вытеснять его, то возможно техника блокирование сопрограмм даст прирост производительности.

г) В отличии от потоков, сопрограммы планируются средой исполнения языка. Это дает огромный плюс. Благодаря этому, разработчику приложения дается возможность оптимизировать планировщик сопрограмм под конкретную задачу.

Сопрограммы уже поддерживаются многими языками программирования, такими как C++ стандарта 20, C#, JavaScript, Go и многие другие. Эти языки используют различные подходы к реализации сопрограмм. В следущих разделах рассмотрим способы переключения в управляемых средах.

2.2 JavaScript и C#

В JavaScript для работы с асинхронным вводом выводом введен класс Promise. Он представляет собой обёртку для значения, неизвестного на момент создания объекта. Promise позволяет обрабатывать результаты асинхронных операций так, как если бы они были синхронными: вместо результата асинхронного метода возвращается обещание получить результат в будущем. Для удобной работы с Promise (с Task в случае C#), существует специальный синтаксис, который называется «async/await».

```
async function f() {
  return 100;
}
```

Ключевое слово async перед функцией означает, что функция всегда возвращает Promise.

```
let value = await promise;
```

Ключевое слово await заставляет ждать, пока Promise не исполнится, и возвращает результат операции. Это работает только в функциях, помеченных ключевым словом async.

Механизм async/await работает как потоки, но точки совместного планирования явно отмечены значком await. Это позволяет писать масштабируемый синхронный код и решает проблему контекста, вводя его новый вид, который представляет собой поток во всем, но несовместим с потоками операционной системы. Синхронный и асинхронный код обычно не могут быть смешаны в одином блоке кода, и в результате языки с поддержкой async/await требуют два разных API для приостановки выполнения async блока кода и текущего потока. В Kotlin существует таже самая проблема: один API предназначен для приостановки потока, а другой для остановки новой конструкции, которая похожа на поток, но не является им.

2.3 Язык Go

Сопрограммы в Go еще называют горутинами. Это функции, которые запускаются конкурентно с другими функциями. При запуске новой сопрограммы нужно перед вызываемой функцией вставить ключевое слово **go**.

```
package main
import ”fmt”

func foo() {
  fmt.Println(”Foo called.”)
}
func bar() {
  fmt.Println(”Bar called.”)
}
func main() {
  go foo()
  go bar()
}
```

Эта программа содержит 3 сопрограммы. Первая - это функция main, являющаяся неявной сопрограммой. Вторая и третья это foo и bar. Обычно при вызове функции, наша программа выполняет все ее операторы, а затем возвращает управление на следующую строку после вызова. С помощью сопрограммы управление немедленно переходит к следующей строке без необходимости дожидаться завершения функции. Среда исполнения языка не позволяет использовать потоки операционной системы напрямую. Разрешается создавать только сопрограммы.

Язык Go, начиная с версии 1.3 использует непрерывный стек корутин. В прологе вызываемой функции вставляется проверка, что текущего размера стека будет достаточно для исполнения кода. Если старый стек слишком мал, то выделяется память под новый стек и содержимое старого копируется в новый. Имеющиеся указатели на данные в с стеке изменяются на новые. Данная реализация позволяет иметь небольшие стеки порядка 4-8кБ, которые могут расти в неограниченных пределах. Но проверка выхода за границы размера в прологе дает накладные расходы, что является минусом.

Механизм переключение контекста сопрограммы в языке Go похож на переключения потоков[kernel.org???]: сохраняются в отдельный буфер необходимые регистры и информация, характерная только для данного потока/горутины, а затем в указатель стека записывается другой адрес. Но в отличии от потоков, при переключении сопрограммы Go выполняется меньше операций и сохраняются не все регистры[4].

2.4 Проект ”Loom”

В текущей версии языка Java - JDK16, поддержка сопрограмм отсутствует. Но с конца 2017 года ведется работа в этом направлении проектом ”Loom”. Он направлен на сокращение усилий по написанию, поддержке высокопроизводительных параллельных приложений, которые максимально используют доступное оборудование[5].

Сопрограммы Loom - это виртуальные потоки, создание и блокировка которых требует меньше накладных расходов[5]. Они управляются средой исполнения языка Java. В отличие от представленных в стандартной библиотеке потока ”java.lang.Thread”, виртуальные потоки не являются оболочками потоков ОС, а реализованы в JDK.

Loom использует другой подход к переключению сопрограмм, отличающийся от языка Go. Когда сопрограмма начинает свою работу, она использует стек потока, на котором она запущена. При переключении сопрограммы происходит копирование части стека, используемой сопрограммой, в отдельный буфер памяти. Если нужно передать управление обратно в сопрограмму, то происходит опустошение буфера и копирование его содержимого на вершину стека потока[5]. Такая техника выбрана в целях совместимости со всеми существующими сборщиками мусора, которые есть в OpenJDK. Некоторые из них не могут поддерживать объекты на куче, которые могут хранить ссылки по смещениям в памяти, меняющиеся на протяжении всего времени существования объекта на куче.

2.5 Применение сопрограмм

Сопрограммы имеют ряд практических применений. При появлении они использовались как средство для создания генераторов, итераторов, бесконечных списков и так далее.

С недавнего времени, с помощью сопрограмм стали реализовывать структурированный параллелизм: управление разделяется на параллельные задачи, они снова объединяются. Если основная задача разбивается на несколько параллельных подзадач, которые должны выполняться порожденными сопрограммы, завершающиеся до основной задачи. Вызывающий метод не должен заботиться о том, разбивает ли метод работу на подзадачи, которые выполняются миллионом сопрограмм. Когда метод завершается, все сопрограммы, запланированные методом, должны быть завершены.

Сопрограммы подходят для работы со множеством относительно независимых задач.

Рассмотрим на примере http-сервера. Каждый из запросов, которые он обслуживает, в значительной степени независим от других. Для каждого из них выполняется синтаксический анализ пакета, делается запрос к базе данных и/или к другому серверу, формируется ответ, который отправляется в клиенту. Каждый запрос не взаимодействует с другими одновременными HTTP-запросами, но они конкурирует с ними за процессорное время и ресурсы ввода-вывода. Другими словами, http-сервер имеет некоторые единицы параллелизма домена приложения, специфичные для его области, причем исполнение некоторой задачи выполняется независимо от другой в то же время. В данном случае единицей параллелизма является http-запрос. Для базы данных это может быть транзакция.

Проблема заключается в том, что поток, не может соответствовать масштабу единиц параллелизма домена приложения. Потенциально сервер может обрабатывать до миллиона одновременных открытых сокетов, но операционная система не может эффективно обрабатывать более нескольких тысяч не бездействующих потоков.

Сопрограммы позволяют избежать такую проблему. В силу своих преимуществ, их может быть созданно несколько миллионов, в то время как операционная система может поддерживать всего лишь несколько тысяч активных потоков. В добавок они уменьшают время отклика сервера из–за большей скорости переключения.

Кроме того, сопрограммы позволяют легко организовать работу с асинхронным кодом. Неблокирующие операции, в том числе и асинхронный ввод–вывод, строится на обратных обработчиках. Когда такого кода очень много, его становится очень трудно отлаживать и исправлять. Сопрограммы помогают избежать такой проблемы, представляя список обработчиков как последовательный кусок кода.

3 ЦЕЛИ И ЗАДАЧИ

Целью данной работы является реализация прототипа сопрограмм в языке программирования Java. Разработка проводилась на базе HuaweiJDK: альтернативной реализации виртуальной машины Java, которая поддерживает компиляцию перед исполнением. Создание сопрограмм в управляемой среде имеет ряд требований.

а) Управляемая среда должна уметь переключать контекст выполнения. Без этого механизма невозможно представить реализацию сопрограмм.

б) Виртуальная машина должна уметь проводить сборку мусора объектов, чьи ссылки лежат на стеках и в сохраненных регистрах сопрограмм.

в) Сопрограммы должны корректно работать в критических секциях, то есть в случае Java кода внутри блоков, помеченных ключевым словом ”synchronized”.

г) Среда исполнения должна уметь вытеснять сопрограмму, которая инициировала блокирующую операцию, и запускать новую. Примером блокирующей операции может послужить сетевой ввод-вывод или блокировка мьютекса. Как говорилось раннее, это поможет избежать блокирование потока.

д) Необходимо уметь корректно обрабатывать исключение, брошенное из сопрограммы. Причем поведение броска исключения может быть разным: когда исключение развернуло весь стек вызовов, оно может быть переброшено в несущий поток либо привести к приостановки сопрограммы и печати ее стека вызовов.

е) И наконец, HuaweiJDK с модулем сопрограмм должен проходить набор тестов JCK. [2]

Для того, чтобы минимальный прототип сопрограмм заработал в управляемой среде исполнения, достаточно реализовать первые два пункта. Как было показано в обзоре предметной области, существует несколько способов

[2] Java Compatibility Kit (JCK) - набор тестов на совместимость (что он еще там проверяет???)

переключения контекста сопрограмм. Для того, чтобы выбрать оптимальный вариант, необходимо разработать набор тестов на производительность сопрограмм в управляемых средах Go, Java/Loom, поскольку на текущий момент не существует подходящих тестов, которые бы измеряли скорость переключения контекста и размер физической памяти, потребляемой сопрограммаами или потоками. Полезно так же сравнить производительность сопрограмм и потоков, чтобы доказать утверждение, данное ранее. В дальнейшем, эти же тесты можно будет использовать для сравнение производительности сопрограмм в HuaweiJDK от OpenJDK и Go.

В итоге, полный список задач, поставленных для достижения поставленной цели выглядит следущим образом:

а) Разработать тесты для сравнения производительности потоков и различных реализаций сопрограмм в управляемых средах.
б) Реализовать переключение сопрограмм.
в) Поддержать трассировку ссылок объектов на стеках сопрограмм для сборки мусора.
г) Сравнить производительность сопрограмм и потоков.

4 ОПИСАНИЕ РЕШЕНИЯ

4.1 Разработка тестов

Пред началом реализации сопрограмм в Excelsior RVM были разработаны тесты производительности для языков Go и ранней версией проекта ”Loom”.

Todo

4.2 Переключение сопрограмм в HuaweiJDK

В таблице 4.1 показаны результаты измерений скорости переключения сопрограмм Go и OpenJDK/Loom[3]. Видно, что сопрограммы из языка Go выигрывают OpenJDK в скорости переключения.

Таблица 4.1 – Число переключений сопрограмм

Шт.	*Переключений, тыс./сек.*	
	OpenJDK/”Loom”	*Go*
100	*1 900 ± 20*	*18 187 ± 219*
1 000	*1 775 ± 20*	*17 934 ± 332*
5 000	*1 703 ± 30*	*12 892 ± 339*
10 000	*1 924 ± 235*	*8 307 ± 80*
20 000	*1 863 ± 217*	*7 045 ± 72*
30 000	*1 772 ± 182*	*6 391 ± 94*
40 000	*1 606 ± 194*	*5 790 ± 67*
50 000	*1 503 ± 157*	*5 292 ± 122*

Поэтому, для реализации в HuaweiJDK был выбран подход языка Go. Из-за простоты использования, в раннем прототипе для переключения контекста

[3]все измерения проводились на операционной системе Ubuntu, kernel 4.15, Intel Core i7-8700, 4.6 ГГц, 32 Гб ОЗУ

использовались функции из библиотеки glibc getcontext и swapcontext. Результат измерения раннего прототипа приведен в таблице 4.2.

Таблица 4.2 – Число переключений сопрограмм

Шт.	*Переключений, тыс./сек.*
	HuaweiJDK
100	*1 956 ± 38*
1 000	*1 829 ± 12*
5 000	*1 578 ± 39*
10 000	*1 316 ± 20*
20 000	*1226 ± 8*
30 000	*1068 ± 7*
40 000	*928 ± 7*
50 000	*881 ± 5*

Анализ исходного кода функции swapcontext показал, что переключение можно ускорить. Она используется для переключения потоков в операционных системах на ядре Linux[ref]. Функция getcontext предоставляет информацию о пользовательском контексте, описывающую состояние потока перед активацией обработчика сигнала, в том числе и предшествующую маску сигналов и сохраненные значения регистров, в частности, программный счетчик и указатель стека[7]. Функция swapcontext предварительно делает системный вызов для сохранения текущей маски сигналов потока, в чем нет необходимости при переключении контекста сопрограмм. Это один из факторов, побудивший реализовать аналоги функций getcontext, swapcontext внутри HuaweiJDK, которые бы учитывали особенности виртуальной машины. На таблице 4.4 представлено сравнение результатов скоростей переключения.

Таблица 4.3 – Сравнение числа переключений

Шт.	*Число переключений, тыс./сек.*	
	getcontext/setcontext	*Функции из HuaweiJDK*
100	*1 956 ± 38*	*12 980 ± 540*
1 000	*1 829 ± 12*	*11 420 ± 694*
5 000	*1 578 ± 39*	*5 875 ± 183*
10 000	*1 316 ± 20*	*4 459 ± 162*
20 000	*1226 ± 8*	*3 604 ± 93*
30 000	*1068 ± 7*	*3 031 ± 94*
40 000	*928 ± 7*	*2 653 ± 87*
50 000	*881 ± 5*	*2 315 ± 60*

Видно, что функции из glibc проигрывают в несколько раз при любом количестве сопрограмм.

4.3 Сборка мусора

Следущим шагом работы стала сборка мусора объектов, расположенных в зоне видимости функций, вызванных в сопрограмме. Виртуальная машина Java хранит список начал и вершин всех потоков, созданных в процессе работы. Это необходимо для того, чтобы при сборке мусора стало возможным нахождение всего корневого множества живых объектов.

В случае сопрограмм необходимо повторить данную логику: требуется хранить все адреса начал и вершин в некотором буфере. Но в отличии от потоков, нужно еще сохранять регистры приостановленных сопрограмм для корректной сборки.

4.4 Потребление памяти

После реализации базового прототипа, наступил этап измерения потребления физической памяти.

Таблица 4.4 – Измерение потребления физической памяти

Шт.	*Резидентная память*		
	HuaweiJDK	*OpenJDK/”Loom”*	*Go*
100	*18 Мб*	*130 Мб*	*3,040 Мб*
1000	*22 Мб*	*161 Мб*	*3,105 Мб*
5000	*32 Мб*	*187 Мб*	*3,156 Мб*
10000	*37 Мб*	*193 Мб*	*3,308 Мб*
20000	*45 Мб*	*196 Мб*	*3,320 Мб*
30000	*49 Мб*	*197 Мб*	*3,350 Мб*
40000	*51 Мб*	*200 Мб*	*3,390 Мб*
50000	*57 Мб*	*202 Мб*	*3,407 Мб*

Таблица 4.5 – Сравнение потребления памяти сопрограмм и потоков.

Шт.	*Размер физической памяти*	
	Сопрограммы	*Потоки*
100	*18 Мб*	*34 Мб*
1000	*22 Мб*	*35 Мб*
5000	*32 Мб*	*37 Мб*
10000	*37 Мб*	*40 Мб*
20000	*45 Мб*	*49 Мб*
30000	*49 Мб*	*56 Мб*
40000	*51 Мб*	*63 Мб*
50000	*57 Мб*	*72 Мб*

5 РЕЗУЛЬТАТЫ

В результате данной работы был разработан набор тестов для сравнения производительности потоков и сопрограмм в управляемых средах языка Go и OpenJDK/Loom. Тесты позволяют измерить скорость переключения и потребление физической памяти [укажи что RES???!!?]. Так же был реализован базовый прототип сопрограмм в HuaweiJDK, который способен переключать контекст и поддерживает сборку мусора объектов, ссылки которых находятся на стеках сопрограмм. Далее, учитывая особенности виртуальной машины Java, была оптимизирована скорость переключения более чем в 3 раза.

С помощью ранее разработанных тестов были получены результаты измерения производительности сопрограмм в HuaweiJDK, и проведено сравнение результатов с OpenJDK/Loom и языком Go. Созданные в рамках этой работы сопрограммы в HuaweiJDK обходят производительность OpenJDK в 3–8 раз в зависимости от их числа.

В дальнейшем планируется:

а) Поддержка synchronized блоков языка Java.

б) Реализация вытеснения сопрограмм, инициирующих блокирующие операции.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Сопрограммы [Электронный ресурс] https://en.wikipedia.org/wiki/Coroutine

2. Модула-2 [Электронный ресурс] https://en.wikipedia.org/wiki/Modula-2

3. Симула [Электронный ресурс] https://en.wikipedia.org/wiki/Simula

4. Переключение контекста в языке Go. [Электронный ресурс] https://github.com/golang/go/blob/master/src/runtime/asm_amd64.s

5. Проект Loom [Электронный ресурс] https://wiki.openjdk.java.net/display/loom/Main

6. Исходный код библиотеки Glibc 2.33 [Электронный ресурс] https://www.gnu.org/software/libc/

7. Майкл Керриск. Linux API. Исчерпывающее руководство. — СПб.: Питер, 2018.