СОДЕРЖАНИЕ

1 АННОТАЦИЯ

Сопрограммы (или корутины) — программный модуль, особым образом организованный для обеспечения взаимодействия с другими модулями по принципу кооперативной многозадачности [1]. Выполнение корутины может быть приостановлено в определённой точке и предано другой сопрограмме. При этом будет сохранено полное состояние сопрограммы (включая стек, значения регистров и счётчик команд). Преимущество корутин перед потоками 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.

Сопрограммы используются для реализации генераторов, итераторов, бесконечных списков, конечные автоматов внутри одной подпрограммы. Но с недавнего времени разработчики поняли, что сопрограммы можно использовать для написания асинхронного и неблокирующего кода. Корутины очень полезны разработчикам многопоточных веб-сервисов , поскольку они позволяют уменьшить время отклика приложения из-за своих преимуществ, перечисленных выше. Потому сопрограммы были реализованы во многих популярных языках программирования, таких как Go, Kotlin, C#. К сожалению, сейчас не существует нативного решения для языка Java. Целю работы является создание работающего прототипа корутин в языке Java. Работа проводилась на базе виртуальной машины Excelsior Research Virtual Machine. Для достижения поставленной цели необходимо было решить следующие задачи:

– Разработать тесты для сравнения эффективности различных реализаций корутин.
– Реализовать поддержку сопрограмм в Excelsior RVM.
– Сравнить производительность с другими языками, используя ранее разработанные метрики.

2 ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Сопрограмма (с английского coroutine) — программная компонента, особым образом организованная для обеспечения взаимодействия с другими компонентами по принципу кооперативной многозадачности. 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. К сожалению, 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.

Начиная с 2010-го года все стало стремительно меняться. Программы все это время росли и появилась проблема масштабируемости таких приложений. Одним из способов масштабирования программ это параллелизм: если хотим обработать кусок данных, который может быть достаточно крупным, то мы можем распределить его обработку на несколько потоков. Но есть другой, более сложный и распространенный вид масштабирования, который касается одновременной обработки относительно независимых задач, требуемых от приложения - конкуренция(с англ. concurrency). То, что они должны обслуживаться одновременно, - это не выбор реализации, а требование.

Рассмотрим на примере веб-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 кли-

енту. Каждый запрос не взаимодействует с другими одновременными HTTP-запросами, но они конкурирует с ними за процессорное время и ресурсы ввода-вывода. Каждое параллельное приложение имеет некоторые единицы параллелизма, естественные для его области, причем исполнение некоторой работы выполняется независимо от другой в то же время. Для веб-сервера это может быть HTTP-запрос; для базы данных это может быть транзакция.

Проблема заключается в том, что поток, программная единица параллелизма, не может соответствовать масштабу естественных единиц параллелизма приложения - пользовательского сеанса, HTTP-запроса или транзакция в базу данных. Сервер может обрабатывать до миллиона одновременных открытых сокетов, но операционная система не может эффективно обрабатывать более нескольких тысяч активных (не бездействующих) потоков. Она были придуманы для виртуализации скудных вычислительных ресурсов с целью совместного использования, но сами стали дефицитными ресурсами. Поскольку создание новых потоков дорогостоящая операция, их объединяют в пул для дальнейшего переиспользования.

К сожалению, пул предлагает слишком грубый механизм разделения потоков. Часто в пуле их просто недостаточно для представления всех независимых задач, выполняемых одномоментно. Заимствование потока ОС из пула на все время выполнения задачи удерживает поток, даже когда он ожидает какого-либо внешнего события, например, ответ от базы данных, сервера, или любого другого действия, которое может его заблокировать. Потоки ОС слишком важны, чтобы за них можно было держаться, когда задача просто ждет. Чтобы совместно использовать потоки более точно и эффективно, стоило бы возвращать поток в пул каждый раз, когда задача должна ждать некоторого результата. Это означает, что задача больше не привязана к одному потоку для всего своего выполнения. Это также означает, что мы должны не допускать блокировки потока, потому что такой поток станет недоступен для любой другой работы.

Сопрограммы способны решить проблемы, перечисленные выше. В среде исполнения языка есть возможность реализовать передачу управления дру-

гой сопрограмме вместо того, чтобы блокировать ее при ожидании результата ввода/ вывода. При реализации корутин в среде исполнения языка появляется возможность контролировать их выполнение, определяя состояние не как ресурс операционной системы, а как объект, известный виртуальной машине, и находящийся под прямым контролем среды исполнения. Объекты надежно и эффективно моделируют всевозможные конечные автоматы и структуры данных, поэтому они также хорошо подходят для выполнения модели. Виртуальная машина знает, как ее код использует стек и память, поэтому может более компактно представлять состояние выполнения. Кроме того, прямой контроль над выполнением сопрограмм также позволяет переопределять планировщики на те, которые лучше подходят для задачи. Фактически, программисту дается возможность использовать настраиваемые планировщики. В то время как операционная система может поддерживать всего лишь несколько тысяч активных потоков, корутин может быть созданно несколько миллионов. Таким образом, каждая единица параллелизма в приложении может быть представлена собственной корутиной, а это значительно упрощает программирование параллельных приложений. Таким образом, понимание кода средой выполнения языка позволяет снизить стоимость потоков.

Некоторые языки программирования борятся со сложностями асинхронного кода.

2.1 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.

2.2 Язык Go

Сопрограммы в Go - 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. Отмечу, что 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.

2.3 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.

В основе лежит концепция виртуальных потоков: это потоки, создание и блокировка которых обходится дешево. 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.

Проект ”Loom” использует необычный подход к переключению сопрограмм. Когда корутина начинает свою работу, она использует стек потока, на котором она запущена. При переключении корутины 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.

3 ЦЕЛИ И ЗАДАЧИ

Целью данной работы является реализация прототипа сопрограмм в языке программирования Java. Для достижения поставленной цели необходимо решить следующие задачи:

– Разработать метрики для сравнения эффективности различных реализаций корутин.
– Реализовать поддержку сопрограмм в Excelsior RVM с учетом особенностей языка Java.
– Сравнить производительность с другими языками, используя ранее разработанные метрики.

Для того, что бы базовая версия сопрограмм заработала, необходимо реализовать на базе Jet RVM:

– Механизм переключения контекста потока.
– Трассирование стеков сопрограмм для сборки мусора.

4 ОПИСАНИЕ РЕШЕНИЯ

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

Таблица 4.1 – Число переключений корутин

Число сопрограмм, шт.	Число переключений, шт/мин		
	Go	Java потоки	Java(Loom)
100	8213552	6550504	4010587
1000	8213552	2143438	3897875
5000	3958044	1440942	3868023
10000	2736352	1343995	3418569
100000	2562853	888052	3889537
1000000	2135474	1089360	3227888

Таблица 4.2 – Число запросов к эхо серверу

Соединений, шт.	Число запросов, мл. шт/мин		
	Go	Java потоки	Java(Loom)
100	11,5	9,7	7,1
1000	7,8	9,5	5,7
2000	7,6	9,3	5,2
3000	7,4	-	4,6
4000	7,2	-	4,3
5000	7,1	-	4,6

Реализация сопрограмм в Excelsior RVM отличается от подхода в проекте ”Loom”. Вместо копирования стека сопрограммы при каждом переключении используется механизм, похожий на переключения потоков операционной системы.

5 ЗАКЛЮЧЕНИЕ

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