ВИРТУАЛЬНЫЕ ПОТОКИ – ЭВОЛЮЦИЯ ПАРАЛЛЕЛЬНОГО ПРОГРАММИРОВАНИЯ В ЯЗЫКЕ JAVA

VIRTUAL THREADS – EVOLUTION OF CONCURRENT PROGRAMMING IN JAVA LANGUAGE

УДК 004.4'2

Вялков Кирилл Андреевич, Аспирант, федеральное государственное автономное образовательное учреждение высшего образования «Национальный исследовательский университет ИТМО», Россия, г. Санкт-Петербург. E-mail: akvone@mail.ru

Vyalkov Kirill Andreevich, Postgraduate student, Saint Petersburg National Research University of Information Technologies, Mechanics and Optics Russia, Saint-Petersburg. E-mail: akvone@mail.ru

Аннотация

Исторически сложилось, что написание параллельного кода является намного более сложным процессом, чем написание обычного последовательного кода. Для написания параллельных приложений на языке Java с самых ранних версий использовались потоки. Но со временем довольно сложные программы стали еще сложнее из-за того, что для обеспечения максимальной производительности пришлось применять подходы, которые позволили оптимизировать повторное использование потоков при этом сильно снизив удобство написания и читаемость кода. Появившийся в 2018 году проект Loom поставил своей задачей дать возможность разработчикам Java перестать использовать данные подходы,

упростить разработку и при этом писать масштабируемый код. Для реализации этой задачи в Java Development Kit была добавлена концепция виртуальных потоков, более известных в научной литературе как файберы.

В данной работе будет описана проблема, с которой столкнулись разработчики Java, рассмотрена текущая система абстракций параллельного программирования платформы и нововведения проекта Loom. В финальной части работы будет приведена апробация работы ранних сборок Java Development Kit, а также рассмотрены текущие ограничения архитектуры.

Annotation

Historically, writing concurrent programs has been much more complex than writing the sequential ones. Java language has threads from the very beginning to build concurrent applications. But over time, fairly complex programs have become even more complex since to maximize performance it was necessary to use approaches that allowed to reuse the threads which in turn led to a decrease in code maintainability and readability. Launched in 2018, the Loom project has the goal to make Java developers stop using these approaches, simplify development, and still write scalable code at the same time. To accomplish this task, the Java Development Kit has added the concept of virtual threads, which is known in the scientific literature as fibers.

This paper describes the problem faced by the developers, the current system of concurrent abstractions of the Java platform, and the innovations of the Loom project. In the final part of the work, we test the early builds of the Java Development Kit and describe the current limitations of the architecture that should be considered.

Ключевые слова: поток, виртуальный поток, файбер, планировщик задач, Java, проект Loom, схема M:N.

Key words: Thread, virtual thread, fiber, task scheduler, Java, project Loom, M:N schema

Введение

Множество приложений, написанных на языке Java, являются параллельными: программы, такие как веб-сервера, могут параллельно обслуживать множество запросов при этом конкурируя за вычислительные ресурсы. Любая из таких программ использует потоки. Появившись в версии 1.0 в языке Java абстракция потоков стала удобной и широко используемой. При этом известно, что создание потока Java – это затратный процесс. Ведь потоки Java являются небольшой «оберткой» вокруг потоков операционной системы, а потоки операционной системы являются «тяжеловесными» по причине того, что они должны поддерживать любые языки и любые типы нагрузки. Более того, наиболее распространенные операционные системы Windows и Linux имеют некоторые ограничения в количестве одновременно используемых потоков. Обычно сервера могут создавать до нескольких десятков тысяч потоков, после чего стабильность работы системы резко снижается. Поэтому для повышения пропускной способности приложений, разработчики стали использовать различные подходы, оптимизирующие использование потоков. Например, пулы потоков – заранее созданный набор потоков, который используется в течении всего жизненного цикла приложения. Также стали популярны некоторые парадигмы, такие как асинхронное и реактивное программирование. Все это позволило получить масштабируемые программы, но при этом усложнило написание и поддержку кода.

В 2018 году как часть платформы OpenJDK был впервые представлен проект Loom [1][2]. Проект Loom ставит своей задачей значительно упростить сложность написания параллельных программ и убрать компромисс между простотой написания-поддержки кода и производительности приложения [3]. Для выполнения этой задачи данный проект добавляет в платформу Java поддержку виртуальных потоков. Виртуальные потоки являются реализацией потоков пользовательского уровня, а наиболее близкой к ним абстракцией широко известной в научной литературе можно считать файберы.

С точки зрения разработчика виртуальные потоки – это обычные потоки, но при этом их создание и блокировка – это не затратный процесс. В то время как операционная система может поддерживать несколько десятков тысяч активных потоков, виртуальная машина Java может поддерживать миллионы виртуальных потоков. Все это позволит поменять подходы к программированию: больше не будет необходимости использовать пулы потоков или асинхронное API.

Теоретические основы

В научных работах под разными терминами параллельного программирования авторы подразумевают различные абстракции, что может приводить к разночтениям полученных результатов. Так понятие «файбер» имеет очень широкое применение. Для того чтобы предоставить некоторый базис, дадим определения важным терминам, которые будут использоваться в данной работе, а также приведем описание основных концепций.

В данной работе широко используются следующие термины [4]:

- Поток (или поток выполнения) – абстракция операционной системы, которая позволяет выполнять последовательный набор инструкций с возможностью приостановки и продолжения их выполнения. Несколько потоков, принадлежащих к одному процессу, могут иметь доступ к ресурсам данного процесса.
- Поток уровня ядра – поток выполнения, который полностью контролируется ядром операционной системы. Потоки ядра выполняются только в пространстве ядра и обслуживают такие задачи, как асинхронный ввод-вывод, управление памятью и обработка сигналов.
- Поток пользовательского уровня – поток выполнения, который разделяет ресурсы в рамках одного процесса. Каждый поток включает некоторое состояние (включает память стека, состояние регистров, набор локальных переменных, приоритет потока), а также может получать доступ к ресурсам,

совместно используемым другими потоками (глобальные переменные, файловые дескрипторы).

- Файбер – абстракция программы (например, виртуальной машины), которая позволяет выполнять последовательный набор инструкций с возможностью приостановки и продолжения их выполнения. В отличие от потока несколько файберов могут иметь доступ к контексту потока, в котором они выполняются, при этом операционная система не имеет доступа к информации о файбере. Файбер имеет некоторое состояние (стек и состояние регистров) для возможности повторного планирования.
- Задача – единица работы.

Независимо от уровня абстракции задачи, существует два способа планирования:

- Кооперативный (cooperative) или невытесняющий;
- Упреждающий (preemptive) или вытесняющий;

Невытесняющее планирование – это способ планирования, при котором активная задача выполняется до тех пор, пока она сама, по собственной инициативе, не отдаст управление планировщику для того, чтобы тот выбрал из очереди другую готовую к выполнению задачу.

Вытесняющее планирование – это способ, при котором решение о переключении процессора с выполнения одной задачи на выполнение другой задачи принимается планировщиком, а не самой выполняемой задачей.

Также существует три модели сопоставления потока пользовательского уровня на поток уровня ядра или соответствия файбера на поток пользовательского уровня [7][8]:

- N:1;
- 1:1;
- M:N;

При использовании модели N:1 несколько пользовательских потоков ставятся в соответствие одному потоку ядра. Основным преимуществом такого механизма можно считать низкую нагрузку на планировщик ядра, малые затраты на создание потока и эффективное управление ресурсами. Недостатками такого подхода являются неоптимальная производительность на многопроцессорных системах, а также возможность блокировки потока ядра, из-за чего программа не будет совершать никакого прогресса.

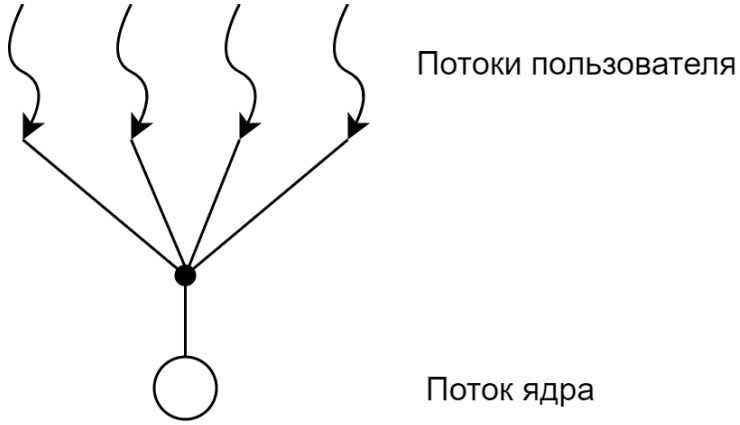


Рисунок 1 Модель N:1

При использовании модели 1:1 каждый поток пользовательского уровня ставится в соответствие одному потоку уровня ядра. Основным преимуществом данного механизма считается простота реализации планировщика пользовательских потоков, так как фактически она полностью зависит от реализации планировщика потока ядра. Основным недостатком можно считать неоптимальное использование ресурсов и большие затраты на создание потока.

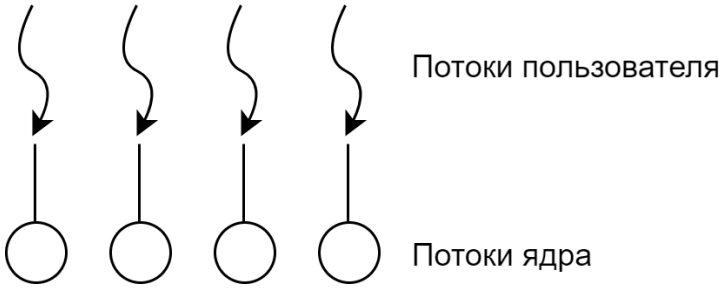


Рисунок 2 Модель 1:1

Модель M:N является комбинацией моделей N:1 и 1:1. Преимуществом такой модели является возможность получения оптимальной производительности. Основной проблемой является сложность реализации планировщика пользовательских потоков.

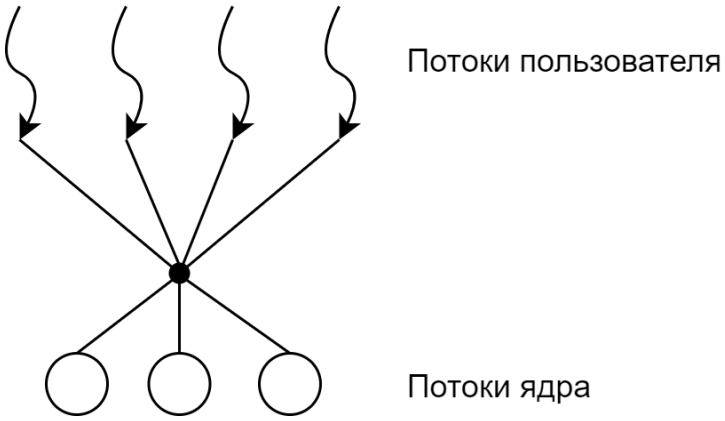


Рисунок 3 Модель M:N

Java потоки и нововведения проекта Loom

На данный момент потоки Java реализуют планирование по схеме 1:1 – один поток Java ставится в соответствие одному потоку ядра. При этом используется вытесняющий способ планирования. На рисунке ниже изображена данная схема работы.

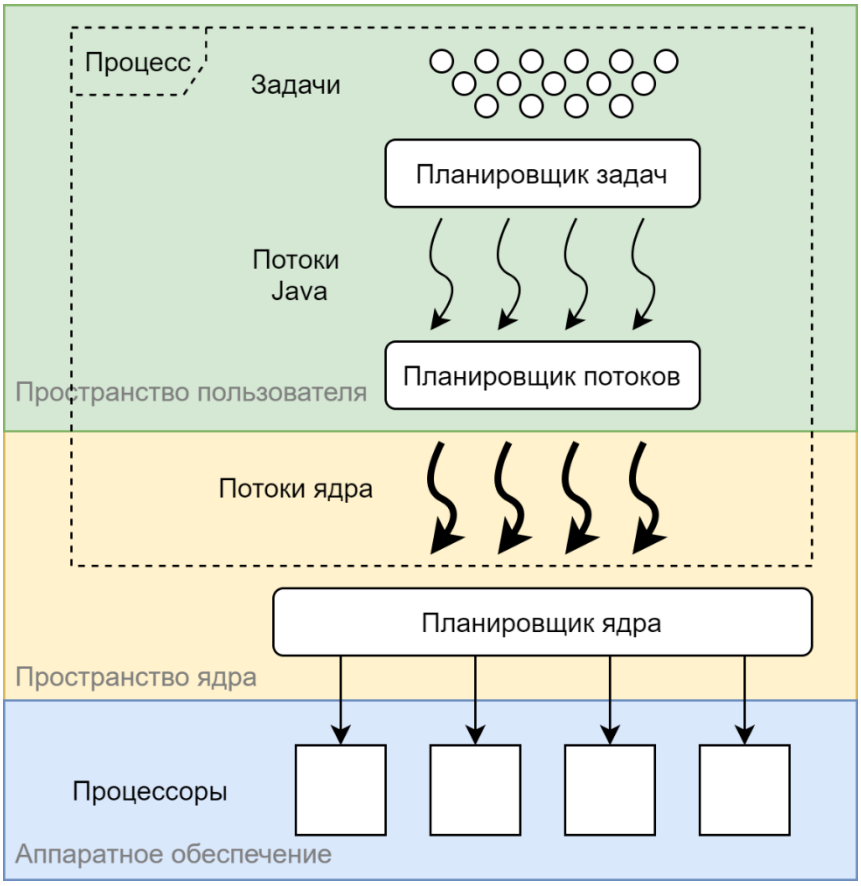


Рисунок 4 Текущая схема работы потоков Java

Дополнительно к обычным потокам, логика которых находится в классе Thread, проект Loom добавляет виртуальные потоки, логика которых содержится в классе VirtualThread. При этом обычные потоки для явного разделения получили дополнительное название – платформенные потоки [5]. Виртуальные потоки выполняются на платформенных потоках с помощью планировщика виртуальных потоков. Таким образом виртуальные потоки реализуют планирование по схеме M:N.

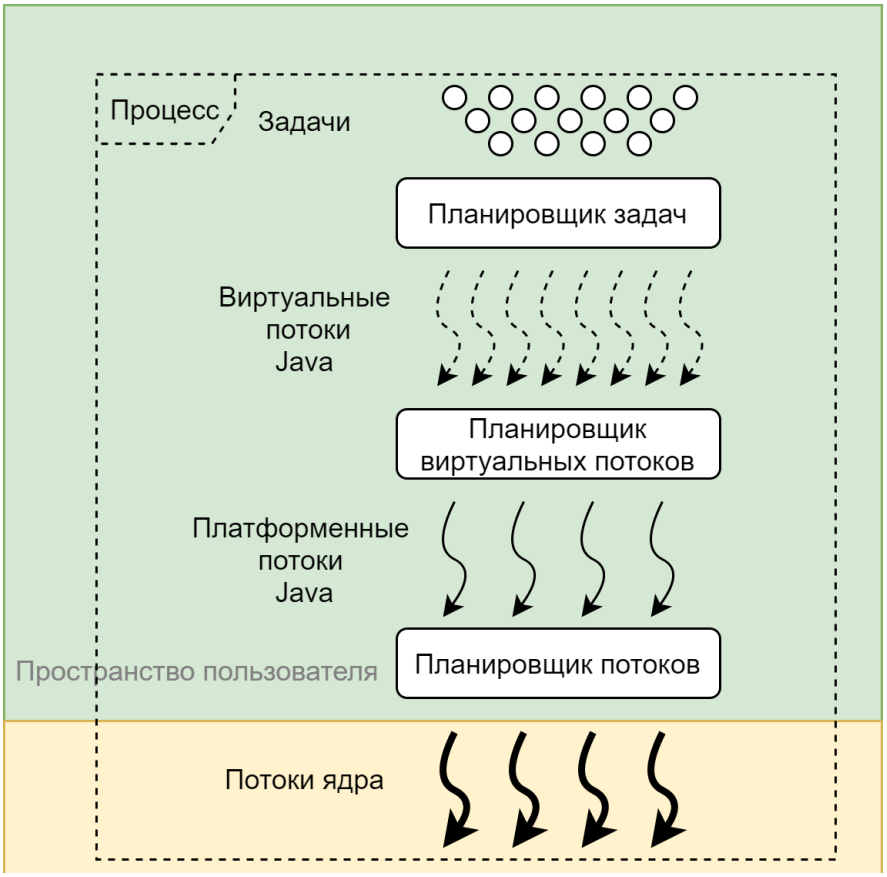


Рисунок 5 Новая схема работы потоков Java

Планирование виртуальных потоков на платформенные потоки происходит в вытесняющем режиме: при этом вытеснение происходит в моменты блокировки на операциях ввода-вывода или синхронизации. На данный момент стандартным планировщиком виртуальных потоков является ForkJoinPool, но имеется возможность указать любой другой планировщик. При этом каждый виртуальный поток присваивается одному конкретному планировщику, после чего его невозможно изменить. Планировщики виртуальных потоков могут использовать различные алгоритмы, при этом самому планировщику необходимо реализовать стандартный интерфейс java.util.concurrent.Executor. Независимо от выбранного планировщика виртуальные потоки обеспечивают гарантии модели памяти Java (Java Memory Model).

Использование различных планировщиков может быть полезно в том случае, если приложение состоит из множества модулей и компонентов, каждый из

которых имеет свою специфику работы с блокирующими операциями. К примеру, на выбор могут влиять среднее время ожидания ответа операций ввода-вывода или частота использования мониторов для параллельных запросов.

Проверка работы

Для проведения анализа возможностей текущего состояния проекта Loom использовалась конфигурация Intel i7-10700U, 16 GB RAM. При тестировании использовалась сборка OpenJDK 64-Bit Server VM AdoptOpenJDK-16.0.1+9 для обычных потоков и ранняя сборка OpenJDK 64-Bit Server VM 16-loom+9-316 для виртуальных потоков.

Далее в работе будут использоваться несколько утилитных методов: «inputOutputOperation», «join» и «spinUntilAllThreadsBecomeWaiting». Метод «inputOutputOperation» имитирует длительную блокирующую операцию, такую как запись файла, чтение из базы данных или HTTP запрос.

```
public static void runInputOutputOperation() {
  runInputOutputOperation(5000);
}

public static void runInputOutputOperation(int millis) {
  try {
    Thread.sleep(millis);
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}
```

Рисунок 6 Метод runInputOutputOperation

Метод «join» ожидает завершения всех потоков:

```
public static void join(List<Thread> threads)
    throws InterruptedException {
  for (Thread thread : threads) {
    thread.join();
  }
}
```

Рисунок 7 Метод join

Метод «spinUntilAllThreadsBecomeWaiting» ожидает перехода всех потоков в состояние «WAITING». Данный метод используется для отслеживания

выполнения метода «inputOutputOperation»: если виртуальный поток находится в состоянии «WAITING», это имитирует ожидание системы ввода-вывода.

```
public static void spinUntilAllThreadsBecomeWaiting(List<Thread>
threads)
    throws InterruptedException {
  while (!threads.stream()
          .allMatch(it -> it.getState() == WAITING)) {
    System.out.println("Не все потоки в состоянии WAITING");
    Thread.sleep(100);
  }
}
```

Рисунок 8 Метод spinUntilAllThreadsBecomeWaiting

Напишем простейшую программу для проверки одновременной работы большого количества потоков. Для примера возьмем один миллион потоков, единственная задача которых – выполнить метод «inputOutputOperation»:

```
var threads = new ArrayList<Thread>();
for (int i = 0; i < 1_000_000; i++) {
  Thread plainThread = new Thread(() -> runInputOutputOperation());
  plainThread.start();
  threads.add(plainThread);
}
join(threads);
```

Рисунок 9 Программа, работающая с миллионом платформенных потоков

Запустив данную программу, система быстро становится неотзывчивой, а ее выполнение занимает более 10 минут. При этом, согласно мониторингу, количество одновременно работающих платформенных потоков не превышает 30 тысяч.

Для того, чтобы продемонстрировать одно из основных преимуществ виртуальных потоков, модифицируем пару строчек кода, заменив обычные потоки на виртуальные:

```
var threads = new ArrayList<Thread>();
for (int i = 0; i < 1_000_000; i++) {
  Thread virtualThread = Thread.builder()
      .virtual() // теперь все потоки - виртуальные
      .task(() -> runInputOutputOperation())
      .start();
  threads.add(virtualThread);
}

spinUntilAllThreadsBecomeWaiting(threads);
join(threads);
```

Рисунок 10 Программа, работающая с миллионом виртуальных потоков

Запустив данную программу, все виртуальные потоки переходят в состояние WAITING в среднем через 2,2 секунды. Выполнение программы происходит в среднем за 46 секунд

Также отметим, что любые программы, которые корректно отрабатывают при использовании платформенных потоков также корректно отработают и при использовании виртуальных потоков. Для примера рассмотрим код, который выполняет параллельный инкремент атомарного счетчика:

```
var atomicInteger = new AtomicInteger();
var threads = new ArrayList<Thread>();
for (int i = 0; i < 100_000; i++) {
  Thread plainThread = Thread.builder()
      .virtual()
      .task(atomicInteger::incrementAndGet)
      .start();
  threads.add(plainThread);
}
join(threads);
System.out.println("Финальное значение: " + atomicInteger.get());
```

Рисунок 11 Программа, проверяющая корректность многопоточной программы

Независимо от типа используемых потоков, в конце выполнения данной программы мы увидим корректный результат – 100000.

Текущие ограничения

Работа над проектом Loom все еще ведется и на данный момент нужно учитывать некоторые особенности при написании нового кода или использовании существующего.

Одна из основных проблем – это то, что в некоторых случаях вызов метода может «закрепить» виртуальный поток к платформенному потоку. Виртуальный поток считается закрепленным к платформенному, если он привязан к нему и не может быть отвязан. Если виртуальный поток блокируется в то время, когда он закреплен, то он блокирует и платформенный поток. В таком случае платформенный поток становится недоступным для других виртуальных потоков [5]. Данное ограничение появляется в двух случаях:

- При использовании нативного кода (JNI);
- При использовании монитора (секции “synchronized”);

Виртуальный поток открепляется от платформенного сразу после завершения нативного кода, либо при освобождении монитора (выхода из секции “synchronized”).

Для демонстрации закрепления виртуального потока при использовании нативных методов рассмотрим следующий код:

```
public static void main(String[] args) throws InterruptedException {
  var executorService = Executors.newSingleThreadExecutor();
  for (int i = 0; i < 100; i++) {
    Thread.builder().virtual(executorService).task(() -> {
      computeInNative(); // Виртуальный поток будет "закреплен" здесь
    }).start();
  }
  executorService.shutdown();
  executorService.awaitTermination(1, TimeUnit.MINUTES);
}

native static void computeInNative();
```

Рисунок 12 Закрепление виртуального потока при использовании нативных методов

В данной программе создается 100 виртуальных потоков, каждый из которых выполняет нативный метод «computeInNative». Все виртуальные потоки используют единственный платформенный поток.

Несмотря на выполнение задач в отдельных потоках фактически выполнение программы становится последовательным.

Для демонстрации закрепления виртуального потока при использовании мониторов рассмотрим следующий код:

```
static final Object monitor = new Object();

public static void main(String[] args) throws
InterruptedException {
  var executorService = Executors.newSingleThreadExecutor();
  // часть 1
  Thread.builder().virtual(executorService).task(() -> {
    synchronized (monitor) { // Виртуальный поток будет
"закреплен" здесь
      inputOutputOperation();
    }
  }).start();
  // часть 2
  for (int i = 0; i < 100; i++) {
    Thread.builder()
        .virtual(executorService)
        .task(() -> inputOutputOperation())
        .start();
  }

  executorService.shutdown();
  executorService.awaitTermination(1, TimeUnit.MINUTES);
}
```

Рисунок 13 Закрепление виртуального потока при использовании мониторов

В первой части программы создается виртуальный поток, который выполняет метод «inputOutputOperation» под монитором. Во второй части программы создается 100 виртуальных потоков, которые выполняют тот же метод, но без использования монитора. Все виртуальные потоки используют единственный платформенный поток. Также для простоты будем считать, что виртуальный поток из первой части будет запланирован для выполнения первым.

При запуске этого кода после попадания в секцию «synchronized» виртуальный поток из первой части будет закреплен, и его выполнение остановит выполнение 100 виртуальных потоков из второй части. После завершения виртуального потока из первой части, 100 виртуальных потоков из второй части будут выполнены параллельно.

Еще одно важное ограничение – это использование механизма ThreadLocal в существующем коде. ThreadLocal позволяет использовать объекты внутри одного потока без явной передачи их в качестве аргументов методов. Данный подход может использоваться для кэширования «тяжелых» объектов (например, так делают для класса SimpleDateFormat); или для хранения контекста операции (например, для хранения состояния транзакции базы данных). Основная проблема связана с тем, что чаще всего использование ThreadLocal в существующих библиотеках подразумевает относительно небольшое количество потоков, к примеру до нескольких десятков тысяч. Это означает, что использование тех же библиотек внутри, скажем, миллиона виртуальных потоков может привести к проблемам с нехваткой памяти. Основное решение данной проблемы – это модификация существующего кода.

Выводы

Рассмотрен проект Loom, который вносит новую абстракцию параллельного программирования в язык Java – виртуальные потоки.

В данной работе разобрана проблема переиспользования потоков Java, а также перечислены текущие подходы к ее решению подходы. Рассмотрена текущая система абстракций параллельного программирования платформы и нововведения проекта Loom. В финальной части работы приведена апробация работы ранних сборок Java Development Kit, а также рассмотрены текущие ограничения архитектуры.

Литература

1. The loom-dev Archives http://mail.openjdk.java.net/pipermail/loom-dev
2. Project Loom Wiki https://wiki.openjdk.java.net/display/loom/Main
3. Project Loom: Fibers and Continuations for the Java Virtual Machine http://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html
4. Fibers are not (P)Threads: The Case for Loose Coupling of Asynchronous Programming Models and MPI Through Continuations https://dl.acm.org/doi/fullHtml/10.1145/3416315.3416320
5. State of Loom https://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part1.html
6. State of Loom: Part 2 https://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part2.html
7. Runtime Performance Analysis of the M-to-N Scheduling Model http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.4682&rep=rep1&type=pdf
8. Thread models Semantics: Solaris and Linux M:N to 1:1 thread model https://www.researchgate.net/publication/270217184_Thread_models_Semantics_Solaris_and_Linux_MN_to_11_thread_model

Literature

1. The loom-dev Archives http://mail.openjdk.java.net/pipermail/loom-dev
2. Project Loom Wiki https://wiki.openjdk.java.net/display/loom/Main
3. Project Loom: Fibers and Continuations for the Java Virtual Machine http://cr.openjdk.java.net/~rpressler/loom/Loom-Proposal.html
4. Fibers are not (P) Threads: The Case for Loose Coupling of Asynchronous Programming Models and MPI Through Continuations https://dl.acm.org/doi/fullHtml/10.1145/3416315.3416320
5. State of Loom https://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part1.html
6. State of Loom: Part 2 https://cr.openjdk.java.net/~rpressler/loom/loom/sol1_part2.html

7. Runtime Performance Analysis of the M-to-N Scheduling Model http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.50.4682&rep=rep1&type=pdf
8. Thread models Semantics: Solaris and Linux M: N to 1: 1 thread model https://www.researchgate.net/publication/270217184_Thread_models_Semantics_Solaris_and_Linux_MN_to_11_thread_model