Тестирование конкурентных структур данных на соответствие моделям согласованности

Морковкин Василий

2020

Аннотация

Чтение данных с записывающего устройства, проведение сетевых запросов и одновременная обработка поступающих команд - одни из задач, стоящих перед программным обеспечением. Результатом оптимизации затрат процессорного времени на решение этих задач стало появление ряда техник *конкурентного программирования*. Однако необходимость использования ресурсов, не поддерживающих одновременный доступ, зависимость от системного планировщика задач, а также особенности устройства процессорных кешей могут приводить к неопределенным или нежелаемым исполнениям компьютерных программ. Целью конкурентного программирования является предоставление механизмов и методик, которые бы позволили писать корректный, обладающий определенным поведением код. Одним из важнейших механизмов являются *структуры данных*, пригодные для использования в конкурентной среде. Для описания их свойств и предоставляемых гарантий был разработан ряд *моделей согласованности*.

Проверка алгоритмов на соответствие этим моделям может быть осуществлена с помощью методов формальной верификации. Однако для проверки конкретных их реализаций на языках программирования доступно лишь тестирование с перебором некоторого подмножества всех сценариев исполнения. В этой работе будут рассмотрены существующие модели согласованности конкурентных структур данных, разработаны варианты их тестирования, и, наконец, реализованы в виде библиотеки на языке программирования Scala.

Оглавление

1 Введение **4**

1.1 Основные понятия 4

1.2 Цели работы 5

1.3 План работы 5

2 Конкурентное программирование **6**

2.1 Проблемы 6

2.1.1 Гонка на данных 6

2.1.2 Состояние гонки 7

2.1.3 Взаимная блокировка 8

2.2 Модели согласованности 10

2.2.1 Последовательная согласованность 10

2.2.2 Линеаризуемость 12

2.2.3 Ослабленная линеаризуемость 12

3 Тестирование моделей согласованности **13**

3.1 Алгоритмическая сложность тестирования 13

3.2 Алгоритмы тестирования 13

3.2.1 Последовательная согласованность 13

3.2.2 Линеаризуемость 13

3.2.3 Ослабленная линеаризуемость 13

3.3 Разработка требований к библиотеке 13

3.3.1 Интерфейс 13
3.3.2 Конфигурация 13

4 Реализация библиотеки **14**
4.1 Интерфейс 14
4.2 Генерация тестовых сценариев 14
4.3 Запуск тестовых сценариев 14
4.4 Валидация результатов исполнения 14
4.5 Генерация отчета тестирования 14

5 Выводы **15**

6 Литература **16**

Введение

1.1 Основные понятия

Введем основные понятия, необходимые в дальнейшем.

Операция — действие, состоящее из некоторого числа программных инструкций. Под этим словом будут подразумеваться отдельные взаимодействия со структурой данных (вставка, удаление, поиск и т.д.).

Процесс — последовательность операций, исполняемая любой единицей исполнения (системным процессом, системной нитью, нитью виртуальной машины, легковесной нитью и т.д.)

Структура данных — формат хранения и управления данным, предоставляющий интерфейс для доступа к ним и их изменения.

Конкурентные операции — две или более операций, временные интервалы исполнения которых пересекаются.

Конкурентные процессы — процессы, исполняющие конкурентные операции.

Конкурентная структура данных — структура данных, подчиняющаяся законам некоторой модели согласованности. Иными словами, структура данных, пригодная для использования конкурентными процессами.

Модель согласованности — набор гарантий о предсказуемости результатов чтения, записи и изменения данных, позволяющий рассуждать о поведении компьютерной программы.

1.2 Цели работы

- Изучить проблемы, возникающие при конкурентном программировании,
- исследовать наиболее распространенные модели согласованности,
- разработать алгоритмы тестирования моделей согласованности,
- реализовать библиотеку для тестирования конкретных реализаций структур данных на выполнение выбранных моделей согласованности.

1.3 План работы

В *Главе 2* будут рассмотрены основные проблемы, а также причины их возникновения, с которыми сталкиваются разработчики при конкурентном программировании, такие как *состояние гонки*, *гонка на данных* и *взаимная блокировка*. Затем будут выделены модели согласованности конкурентных структур данных, такие как *последовательная согласованность*, *линеаризуемость* и различные ослабленные варианты линеаризуемости.

В *Главе 3* будет представлен анализ задачи тестирования конкурентных структур данных, а также предложен ряд конкретных алгоритмов тестирования. Также в главе пойдет рассуждение о хорошем дизайне для библиотеки с описанным функционалом тестирования.

Глава 4 включит в себя заметки о реализации библиотеки на языке программирования *Scala*. Выбор языка обусловлен его встроенными метапрограммированием, возможностью написания на нем удобных предметно-ориентированных языков, возможностью использования мощной стандартной библиотеки *java.util.concurrent*, а также практической нуждой в подобной библиотеке. Все примеры кода в работе также будут приведены на языке *Scala* версии *2.13*.

Конкурентное программирование

2.1 Проблемы

В данном разделе мы рассмотрим ряд проблем, возникающих при помещении программ в конкурентную среду. Первая из них — гонка на данных.

2.1.1 Гонка на данных

Гонкой на данных называют [1] обращения к одному и тому же участку памяти, совершаемые из конкурентных процессов при условии, что хотя бы одно из обращений является обращением на запись, и обращения происходят не в рамках операций синхронизации. Под операциями синхронизации подразумевают, например, вызовы методов мониторов и барьеров памяти.

Рассмотрим пример, содержащий гонку на данных:

```
1  var flag: Boolean = false
2  def raiseFlag = { flag = true }
3
4  ForkJoinPool.commonPool.execute(DataRaceExample.raiseFlag _)
5  while (!flag) {}
```

Листинг 2.1: Пример гонки на данных

Результатом исполнения приведенного кода может стать зависание программы в бесконечном цикле. Причиной этого является копирование значений переменных, с которыми работает процесс, в кеш вычислительного процессора, и последующего чтения из него в то время, когда значение переменной уже было изменено. Данное копирование происходит с целью увеличения эффективности, так как доступ к процессорному кешу значительно быстрее [2] доступа к оперативной памяти.

Описанная проблема решается путем синхронизации обращений к переменных через примитивы синхронизации, встроенные в язык программирования (например, мониторов в Java). Для обнаружения гонок на данных были разработаны специальные инструменты [3, 4].

2.1.2 Состояние гонки

Другая проблема — *состояние гонки*. Несмотря на похожие названия, описанная ранее проблема не имеет прямого отношения к текущей. Гонка на данных возможна без состояния гонки и наоборот [5]

Состоянием гонки называют [6] ситуацию, в которой несколько процессов производят чтение и запись в разделяемую память, и при этом результат их операций зависит от порядка исполнения.

Рассмотрим пример кода, содержащий состояние гонки:

```
1  def transfer(amount: Int, from: AtomicInteger, to: AtomicInteger) = {
2    val fromBalance = from.get()
3    val toBalance   = to.get()
4    from.set(fromBalance - amount)
5    to.set(toBalance + amount)
6  }
7
8  transfer(10, 100, 100)
```

Листинг 2.2: Пример состояния гонки

Результат исполнения приведенного кода двумя процессами зависит от относительного порядка исполнения отдельных операций в этих процессах:

Таблица 2.1: Исполнение 1

Процесс 1	Процесс 2
from.get: 100	
to.get: 100	
from.set: 90	
	from.get: 90
	to.get: 100
	from.set: 80
to.set: 110	
	to.set: 110
Итог: from = 80, to = 110	

Таблица 2.2: Исполнение 2

Процесс 1	Процесс 2
from.get: 100	
to.get: 100	
	from.get: 100
	to.get: 100
	from.set: 90
from.set: 90	
to.set: 110	
	to.set: 120
Итог: from = 90, to = 120	

Видно, что в первом исполнении суммарное число денег на банковских счетах **from** и **to** уменьшилось, а во втором увеличилось, в то время как при исполнении на одном процессе оно не меняется. Данный пример показывает, что состояние гонки может появляться и в хорошо синхронизированном коде, в котором отсутствуют гонки на данных. Для рассуждения о программах, которые не подвержены таким проблемам, нам понадобится ввести понятия моделей согласованности, о которых речь пойдет далее.

2.1.3 Взаимная блокировка

Ещё одной частой причиной некорректного поведения конкурентных программ является состояние *взаимной блокировки*, при котором каждый процесс находится в ожидании совершения какого-то действия (отправки сообщения, отпускания блокировки) другим процессом. Система, находящаяся в таком состоянии, не делает прогресса.

Взаимная блокировка определяется одновременным выполнением следую-

щих условий [7]:

- Процессы требуют эксклюзивного владения ресурсами.
- Процессы, владеющие ресурсами, также ждут возможности завладеть некоторыми дополнительными ресурсами.
- Блокировка ресурса может быть отпущена только тем процессом, который им владеет в данный момент.
- В системе существует замкнутая цепь процессов, таких, что каждый процесс из цепи владеет одним или несколькими ресурсами, которые также необходимы следующему процессу в цепи.

Для наглядности рассмотрим пример:

```
1  val lock1 = new Object
2  val lock2 = new Object
3  Future {
4    lock1.synchronized {
5      lock2.synchronized()
6    }
7  }
8  Future {
9    lock2.synchronized {
10     lock1.synchronized()
11   }
12 }
```

Листинг 2.3: Пример взаимной блокировки

В зависимости от системного планировщика приведенная программа может как завершаться, так и повисать навечно в случае возникновения состояния взаимной блокировки, когда процесс первого блока *Future* не успевает захватить блокировку на объект *lock2* прежде процесса второго блока *Future*.

2.2 Модели согласованности

Все рассмотренные нами проблемы обладают общим свойством – они проявляются не всегда, а только при стечении определенных обстоятельств, не подконтрольных программисту, таких как состояние процессорных кешей и результат работы системного планировщика. Такое непостоянство, а также большое мест в коде, которые могут вызывать проблемы, делает сложным воспроизведение и исправление ошибок.

Поэтому была разработан формализм *моделей согласованности*, который вводит свойства корректности всех конкурентных исполнений некоторого кода, а также дает возможность рассуждать о композиции этих свойств при объединении корректных по отдельности участков кода.

В этом разделе мы рассмотрим несколько широко применяемых моделей согласованности, таких как *последовательная согласованность* и *линеаризумость*.

2.2.1 Последовательная согласованность

Говорят [8], что некоторый участок кода является *последовательно согласованным*, если результат любого его конкурентного исполнения такой же, как если бы все операции всех процессов исполнялись в некотором последовательном порядке, в котором сохранялся бы относительный порядок операций каждого отдельного процесса. Это свойство также не допускает возникновения состояний взаимной блокировки

Рассмотрим это понятие на примере. Пусть есть структура данных, которая поддерживает семантику “первым пришел – первым ушел” при исполнениях на одном процессе. Пусть она имеет следующий интерфейс:

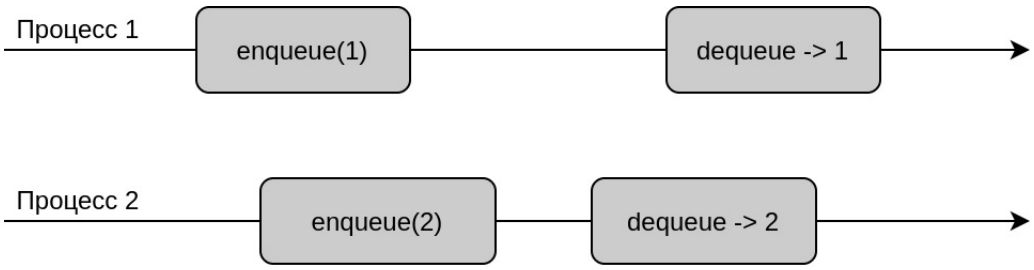
```
1  trait Queue {
2    def enqueue(e: Int): Unit
3    def dequeue: Int
```

```
4   }
```

Листинг 2.4: Интерфейс очереди

Если ее реализация утверждает, что она последовательно согласованна, то она не должна допускать подобных исполнений:

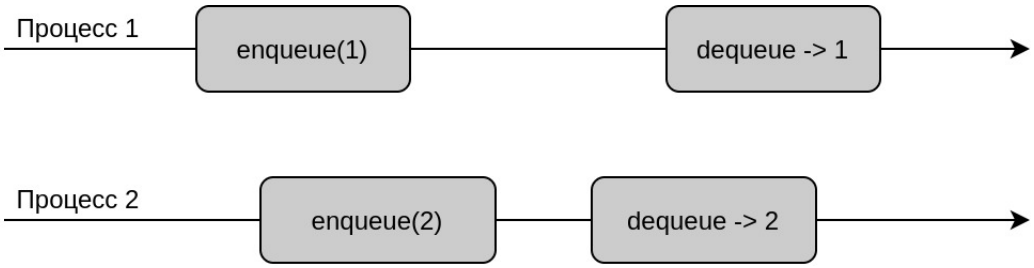
Рис. 2.1: Последовательно не согласованное исполнение



В данном случае нарушается требование существования исполнения на одном процессе, в котором операция *dequeue* вернула бы единицу дважды.

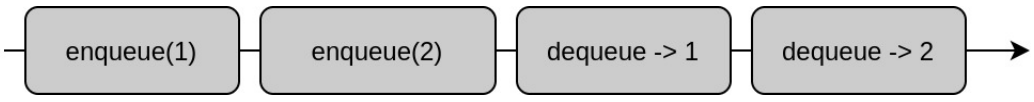
Однако подобные исполнения:

Рис. 2.2: Последовательно согласованное исполнение



допускаются, так как в данном случае все операции можно упорядочить в линейную историю, которая подчиняется спецификации объекта и сохраняет относительный порядок операций каждого из процессов:

Рис. 2.3: Объясняющее последовательное исполнение

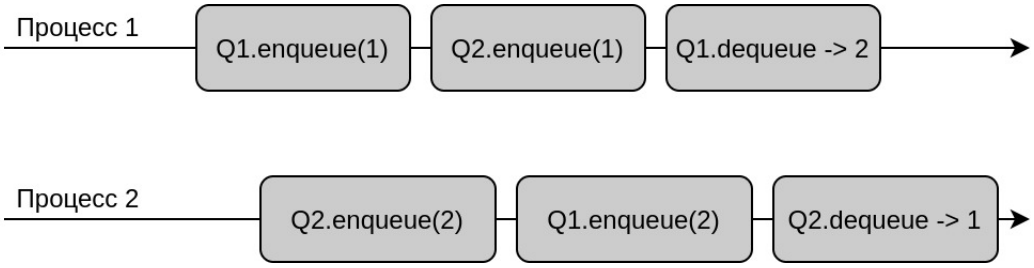


Корректность конкурентного исполнения, приведенного на рисунке 2.2, может показаться контр-интуитивной, поскольку операция *dequeue -> 2* заверши-

лась раньше во времени, чем *dequeue -> 1* в отличие от объясняющего последовательного исполнения на рисунке 2.3.

Также стоит отметить, что результат композиции последовательно согласованных объектов может не являться последовательно согласованным. Рассмотрим некоторую конкурентную историю для двух очередей *Q1* и *Q2*:

Рис. 2.4: Композиция последовательно согласованных объектов



Для каждой из очередей по отдельности можно составить объясняющие последовательные истории, однако для всего исполнения в данном случае этого сделать нельзя.

Более строгой моделью согласованности, которая учитывает временной порядок операций, а также обладает свойством композиции, является *линеаризуемость*.

2.2.2 Линеаризуемость

2.2.3 Ослабленная линеаризуемость

Тестирование моделей согласованности

3.1 Алгоритмическая сложность тестирования

3.2 Алгоритмы тестирования

3.2.1 Последовательная согласованность

3.2.2 Линеаризуемость

3.2.3 Ослабленная линеаризуемость

3.3 Разработка требований к библиотеке

3.3.1 Интерфейс

3.3.2 Конфигурация

Реализация библиотеки

4.1 Интерфейс

4.2 Генерация тестовых сценариев

4.3 Запуск тестовых сценариев

4.4 Валидация результатов исполнения

4.5 Генерация отчета тестирования

Выводы

Литература

[1] J. Erikson et al. Effective data-race detection for the kernel. *Microsoft Research*, 2010.

[2] J. Dean. Latency numbers every programmer should know.

[3] K. Serebryany. Threadsanitizer – data race detection in practice. 2009.

[4] Jep: Java thread sanitizer.

[5] J. Regehr. Race condition vs. data race, 2011.

[6] S. Carr et al. Race conditions: A case study. 2001.

[7] E. Coffman et al. System deadlocks. 1971.

[8] L. Lamport. How to make a correct multiprocess program execute correctly on a multiprocessor. 1979.