

Тестирование конкурентных структур данных на соответствие моделям согласованности

Морковкин Василий

2020

Мотивация

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 нежелательным 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	8
2.2.1	Последовательная согласованность	8
2.2.2	Линеаризуемость	8
2.2.3	Ослабленная линеаризуемость	8
3	Тестирование моделей согласованности	9
3.1	Алгоритмическая сложность тестирования	9
3.2	Алгоритмы тестирования	9
3.2.1	Последовательная согласованность	9
3.2.2	Линеаризуемость	9
3.2.3	Ослабленная линеаризуемость	9
3.3	Разработка требований к библиотеке	9

3.3.1	Интерфейс	9
3.3.2	Конфигурация	9
4	Реализация библиотеки	10
4.1	Интерфейс	10
4.2	Генерация тестовых сценариев	10
4.3	Запуск тестовых сценариев	10
4.4	Валидация результатов исполнения	10
4.5	Генерация отчета тестирования	10
5	Выводы	11
6	Литература	12

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*.

Конкурентное программирование

2.1 Проблемы

В данной секции 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 переменным через примитивы синхронизации, встроенные в язык программирования (например, мониторы 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] называют 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 потоке 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

2.2 Модели согласованности

2.2.1 Последовательная согласованность

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] Jeff 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] Steve Carr et al. Race conditions: A case study. 2001.