

The
Pragmatic
Programmers

Семь моделей конкуренции и параллелизма за семь недель

Раскрываем
тайны потоков

Пол Батчер



УДК 004.42
ББК 32.973
Б28

Б28 Пол Батчер

Семь моделей конкуренции и параллелизма за семь недель. /
Пер. с англ. Киселев А. Н. – М.: ДМК Пресс, 2015. – 360 с.: ил.

ISBN 978-5-97060-244-7

С появлением микропроцессоров, обладающих большим числом ядер, понимание конкуренции и параллелизма при разработке программного обеспечения стало еще более важным, чем прежде. В книге вы познакомитесь с преимуществами функционального программирования с точки зрения конкуренции, узнаете, как применять акторы для разработки распределенного программного обеспечения, и исследуете приемы параллельной обработки огромных объемов информации на нескольких процессорах. Эта книга поможет вам приобрести новые навыки в разработке программ, благодаря чему вы будете готовы решать сложные задачи в ближайшие несколько лет.

УДК 004.42
ББК 32.973

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-937785-65-9 (англ.)
ISBN 978-5-97060-244-7 (рус.)

© 2014 The Pragmatic Programmers, LLC.
© Оформление, перевод на русский язык
ДМК Пресс, 2015



ОГЛАВЛЕНИЕ

Положительные отзывы к книге «Семь моделей конкуренции и параллелизма за семь недель»	5
Предисловие	13
Благодарности	15
Вступление	17
О книге	17
Чем не является эта книга.....	18
Примеры кода	18
Примечание для пользователей IDE.....	19
Примечание для пользователей Windows.....	19
Ресурсы в Сети.....	19
Глава 1. Введение	21
Конкуренция или параллелизм?	21
Похожие, но разные.....	21
За рамками последовательного программирования.....	23
Параллельная архитектура	23
Параллелизм на уровне битов	24
Параллелизм на уровне инструкций	24
Параллелизм данных.....	24
Параллелизм на уровне задач	25
Конкуренция: за рамками множества ядер	26
Конкурентные программы для конкурентного мира	26
Распределенные программы для распределенного мира.....	27
Надежные программы для непредсказуемого мира.....	27
Простые программы в сложном мире	28
Семь моделей	28
Глава 2. Потоки выполнения и блокировки	31
Самое простое из того, что может работать.....	31
День 1: взаимоисключение и модели памяти.....	32
Создание потока	33
Наша первая блокировка.....	34
Загадочная память	37

Видимость памяти	38
Несколько блокировок	39
Опасности сторонних методов	43
В завершение первого дня	44
День 2: помимо встроенных блокировок	46
Прерываемое блокирование	47
Тайм-ауты	49
Блокирование методом перебора	51
Условные переменные	54
Атомарные переменные	57
В завершение второго дня	58
День 3: на плечах гигантов	60
Еще раз о создании потоков	61
Копирование при записи	62
Законченная программа	64
В завершение третьего дня	74
В завершение	75
Сильные стороны	75
Слабые стороны	76
Другие языки	78
Напоследок	79

Глава 3. Функциональное программирование..... 80

Если какие-то действия вредят вам, перестаньте выполнять их	80
День 1: программирование без изменяемого состояния	81
Опасности изменяемого состояния	81
Краткий экскурс в язык Clojure	84
Первая функциональная программа	86
Параллелизм без усилий	87
Функциональный подсчет слов	89
Лень – это благо	93
В завершение первого дня	94
День 2: функциональный параллелизм	95
По одной странице за раз	95
Разделение данных на пакеты для увеличения производительности	98
Редукторы (reducers)	99
Внутреннее устройство редукторов	100
Разделяй и властвуй	103
Поддержка функции fold	104
Подсчет слов с помощью fold	105
В завершение второго дня	107
День 3: функциональная конкуренция	108
Та же структура, разный порядок вычислений	108

Ссылочная прозрачность	109
Потоки данных	110
Механизм future	111
Механизм promise	112
Функциональная веб-служба	113
В завершение третьего дня	121
В завершение	122
Сильные стороны	124
Слабые стороны	124
Другие языки	124
Напоследок	125
Глава 4. Путь Clojure – разделение идентичности и состояния	126
Лучшее из двух миров	126
День 1: атомы и сохраняемые структуры данных	127
Атомы	127
Многопоточная веб-служба с изменяемым состоянием	129
Сохраняемые структуры данных	130
Идентичность или состояние	134
Повторения	134
Валидаторы	135
Функции-наблюдатели	135
Гибридная веб-служба	136
В завершение первого дня	140
День 2: агенты и программная транзакционная память	141
Агенты	141
Журнал в памяти	145
Программная транзакционная память	146
Изменяемое разделяемое состояние	151
В завершение второго дня	151
День 3: погружение в глубину	152
Решение задачи о философах на основе STM	153
Решение задачи о философах без применения STM	155
Атомы или STM?	157
Собственная реализация конкуренции	158
В завершение третьего дня	160
В завершение	161
Сильные стороны	161
Слабые стороны	162
Другие языки	162
Напоследок	162
Глава 5. Акторы	164
Не объекты, а скорее ориентированные на объекты	164

День 1: сообщения и почтовые ящики.....	166
Наш первый актер	166
Почтовые ящики и очереди.....	167
Прием сообщений	168
Связывание процессов.....	169
Актеры с сохранением состояния	170
Соккрытие сообщений за фасадом API.....	171
Двунаправленное взаимодействие.....	172
Именованное взаимодействие	174
Отступление – функции первого порядка.....	176
Параллельная версия map().....	176
В завершение первого дня	177
День 2: обработка ошибок и отказоустойчивость.....	178
Актер кэширования	179
Определение момента отказа.....	182
Слежение за работой процессов	185
Тайм-ауты.....	186
Ядро ошибки	187
И пусть падает!	189
В завершение второго дня.....	190
День 3: распределенные приложения.....	191
ОТР	191
Узлы	196
Распределенный счетчик слов.....	200
В завершение третьего дня	206
В завершение	207
Сильные стороны	208
Слабые стороны	209
Другие языки	209
Напоследок	210

Глава 6. Взаимодействие последовательных процессов 211

Взаимодействия – это все.....	211
День 1: каналы и блоки go.....	213
Каналы	213
Блоки go	217
Операции с каналами	222
В завершение первого дня	225
День 2: множество каналов и ввод/вывод.....	227
Обслуживание множества каналов	227
Асинхронный опрос	230
Асинхронный ввод/вывод	233
В завершение второго дня.....	240
День 3: модель CSP на стороне клиента	241

Конкуренция – это образ жизни	242
Привет, ClojureScript	242
Обработка событий	245
Усмирение функций обратного вызова	247
Отправляемся в путь, чтобы увидеть Мастера	248
В завершение третьего дня	251
В завершение	251
Сильные стороны	252
Слабые стороны	252
Другие языки	253
Напоследок	253
Глава 7. Параллелизм данных	254
В недрах вашего ноутбука спрятан суперкомпьютер	254
День 1: программирование GPGPU	255
Обработка данных и параллелизм данных	255
Наша первая программа для OpenCL	258
Профилирование	264
Множество возвращаемых значений	265
Обработка ошибок	266
В завершение первого дня	268
День 2: многомерность и рабочие группы	270
Многомерные массивы рабочих элементов	270
Получение информации об устройстве	273
Модель платформы	275
Модель памяти	276
Параллельная свертка	277
Свертка с одной рабочей группой	277
В завершение второго дня	282
День 3: OpenCL и OpenGL – храните данные в GPU	283
Водная рябь	284
LWJGL	284
Отображение сетки в OpenGL	285
Доступ к буферу OpenGL из ядра OpenCL	287
Имитация ряби	288
В завершение третьего дня	291
В завершение	293
Сильные стороны	293
Слабые стороны	294
Другие языки	294
Напоследок	294
Глава 8. Лямбда-архитектура	295
Параллелизм позволяет обрабатывать гигантские объемы данных	295

День 1: MapReduce	297
Практические аспекты	298
Основы Hadoop	299
Подсчет слов с помощью Hadoop	301
Опробование на Amazon EMR	305
Обработка XML	308
В завершение первого дня	310
День 2: пакетный уровень	313
Проблемы с традиционными системами данных	313
Вечные истины	315
Лучшие данные – исходные данные	315
Авторы правок в Википедии	317
Завершение картины	323
В завершение второго дня	325
День 3: уровень ускорения	327
Архитектура уровня ускорения	328
Подсчет правок с помощью Storm	335
В завершение третьего дня	341
В завершение	342
Сильные стороны	343
Слабые стороны	343
Альтернативы	343
Напоследок	343
Глава 9. В заключение	344
Куда мы идем?	344
Будущее за неизменяемостью	345
Будущее за распределенными вычислениями	346
Темы, оставшиеся за бортом	346
Fork/Join и захват задачи	347
Потоки данных	347
Реактивное программирование	347
Функциональное реактивное программирование	348
Grid-вычисления	348
Пространства коротежей	348
Выбор за вами	349
Библиография	350
Предметный указатель	352



ГЛАВА 2.

Потоки выполнения и блокировки

Механизм потоков выполнения и блокировок (threads-and-locks) можно сравнить с автомобилем Ford Model T¹. Этот автомобиль сможет доставить вас из точки А в точку Б, но он очень прост в конструкции, сложен в управлении, ненадежен и опасен в сравнении с современными автомобилями.

Несмотря на все сопутствующие проблемы, потоки выполнения и блокировки, тем не менее по-прежнему широко используются для создания конкурентных программ и к тому же лежат в основе многих других технологий, которые будут рассматриваться в этой книге. Даже если вы не планируете использовать их непосредственно, вам все же следует знать, как они работают.

Самое простое из того, что может работать

Потоки и блокировки – это не более чем формализация действий аппаратной части компьютера. В этом их главное достоинство и главный недостаток.

Благодаря их простоте потоки и блокировки поддерживаются практически всеми языками программирования в той или иной форме, и они накладывают не слишком много ограничений. Но они весьма сложны для начинающих программистов. Программы, написанные с применением потоков и блокировок, трудно читать и еще труднее поддерживать.

Далее мы будем знакомиться с программированием потоков и блокировок на примере языка Java, но принципы, излагаемые в этой гла-

¹ https://ru.wikipedia.org/wiki/Ford_Model_T – Прим. перев.

ве, можно распространить на любой другой язык, поддерживающий потоки выполнения. В первый день мы рассмотрим основы разработки многопоточного кода на Java, основные ловушки, с которыми вы можете столкнуться, а также некоторые правила, которые помогут вам избежать их. Во второй день мы двинемся дальше и займемся исследованием инструментов из пакета `java.util.concurrent`. Наконец, в третий день мы посмотрим на некоторые из конкурентных структур данных, поддерживаемых стандартной библиотекой, и попробуем задействовать их для решения практической задачи.

О наиболее эффективных приемах

В первую очередь мы познакомимся с низкоуровневыми потоками выполнения и примитивами блокировок в языке Java. В современном коде редко приходится использовать эти примитивы непосредственно, благодаря наличию высокоуровневых инструментов, о которых мы поговорим во второй и третий дни. Однако, чтобы понять, от чего зависят эти инструменты, необходимо знать и понимать, как действуют базовые механизмы. Именно поэтому мы начнем с основ. Но вы должны знать, что в программах практически нет необходимости напрямую использовать класс `Thread`.

День 1: взаимное исключение и модели памяти

Если прежде вам приходилось заниматься созданием конкурентных программ, вы наверняка знакомы с понятием *взаимного исключения* (*mutual exclusion*) – применения блокировок, чтобы гарантировать доступность данных только для одного потока выполнения в каждый конкретный момент времени. И вы также наверняка знакомы с проблемами, которые возникают при неправильном использовании блокировок, включая состояние гонки и взаимоблокировки (не волнуйтесь, если эти термины вам неизвестны – они описываются чуть ниже).

Это действительно очень большие проблемы, и мы потратим немало времени на их обсуждение, но, как оказывается, существует еще кое-что, не менее, если не более, важное, что следует знать для работы с разделяемой памятью, – Модель Памяти. И если вы думаете, что

состояние гонки и взаимоблокировки способны породить фантастически странное поведение программы, ваше мнение наверняка изменится, когда мы увидим, насколько странной и причудливой может быть разделяемая память.

Однако мы, кажется, пытаемся бежать впереди паровоза – давайте притормозим немного и для начала посмотрим, как создать поток.

Создание потока

Базовым элементом конкуренции в Java является *поток выполнения* (thread), который, как можно заключить из названия, состоит из единственного *потока управления*. Потоки могут взаимодействовать друг с другом посредством разделяемой (или совместно используемой) памяти.

Ни одна книга по программированию не обходится без примера программы «Hello, World!», поэтому не будем отступать от традиций и рассмотрим многопоточную версию такой программы:

```
ThreadsLocks/HelloWorld/src/main/java/com/paulbutcher/HelloWorld.java2
```

```
public class HelloWorld {  
  
    public static void main(String[] args) throws InterruptedException  
    {  
        Thread myThread = new Thread() {  
            public void run() {  
                System.out.println("Hello from new thread");  
            }  
        };  
  
        myThread.start();  
        Thread.yield();  
        System.out.println("Hello from main thread");  
        myThread.join();  
    }  
}
```

Этот код создает экземпляр класса `Thread` и затем запускает его. С этого момента метод `run()` нового потока будет выполняться конкурентно с остальным кодом в методе `main()`. Наконец, метод `join()` ждет завершения работы потока (что произойдет с завершением метода `run()`).

Если запустить эту программу, она выведет следующее:

² <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/HelloWorld/src/main/java/com/paulbutcher/HelloWorld.java>

```
Hello from main thread
Hello from new thread
```

Или:

```
Hello from new thread
Hello from main thread
```

В какой именно последовательности появятся сообщения, зависит от того, какой из потоков успеет первым вызвать функцию `println()` (когда я тестировал эту программу, у меня получилось, что обе ситуации примерно равновероятны). Такая зависимость от времени является одной из самых больших сложностей при программировании многопоточных программ – только потому, что вы наблюдали какое-то определенное поведение, запустив код, не означает, что оно будет проявляться постоянно.



Вопрос Джо: зачем нужен метод `Thread.yield()`?

В нашей многопоточной версии «Hello, World!» имеется следующая строка:

```
Thread.yield();
```

Согласно документации, метод `yield()`:

подсказывает планировщику, что текущий поток добровольно желает отдать остаток выделенного ему кванта процессорного времени.

Без этого вызова, из-за необходимости выполнения процедур инициализации нового потока, главный поток почти наверняка добрался бы до вызова `println()` первым (впрочем, это не гарантируется – как мы увидим далее, если в конкурентной программе что-то *может* произойти, рано или поздно это произойдет, причем в самый неподходящий момент).

Попробуйте закомментировать вызов этого метода и посмотрите, что из этого получится. Посмотрите также, что получится, если заменить вызов `Thread.yield()` вызовом `Thread.sleep(1)`?

Наша первая блокировка

Когда к разделяемой памяти обращаются сразу несколько потоков выполнения, они могут, говоря иносказательно, «оттопать ноги» друг другу. Избежать этой неприятности можно с помощью *взаимоисключающей блокировки*, удерживать которую может только какой-то один поток.

Давайте создадим пару потоков, взаимодействующих друг с другом:

```
ThreadsLocks/Counting/src/main/java/com/paulbutcher/Counting.java3
```

```
public class Counting {
    public static void main(String[] args) throws InterruptedException {
        class Counter {
            private int count = 0;
            public void increment() { ++count; }
            public int getCount() { return count; }
        }
        final Counter counter = new Counter();
        class CountingThread extends Thread {
            public void run() {
                for(int x = 0; x < 10000; ++x)
                    counter.increment();
            }
        }

        CountingThread t1 = new CountingThread();
        CountingThread t2 = new CountingThread();
        t1.start(); t2.start();
        t1.join(); t2.join();
        System.out.println(counter.getCount());
    }
}
```

Здесь мы имеем очень простой класс `Counter` и два потока, каждый из которых вызывает метод `increment()` класса 10 000 раз. Очень просто и ... совершенно неправильно.

Попробуйте несколько раз запустить этот пример, и каждый раз вы будете получать разные результаты. Например, запустив пример три раза, я получил числа 13850, 11867 и 12616. Причина такого поведения — *состояние гонки* (race condition, поведение, зависящее от относительной продолжительности выполнения операций), возникающее из-за использования двумя потоками члена `count` класса `Counter`.

Если для вас это оказалось сюрпризом, давайте посмотрим, какой код получается в результате компиляции инструкции `++count`. Ниже приводится байт-код, сгенерированный компилятором:

```
getfield #2
iconst_1
iadd
putfield #2
```

³ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/Counting/src/main/java/com/paulbutcher/Counting.java>

Даже для тех, кто незнаком с байт-кодом JVM, совершенно очевидно, что здесь происходит: инструкция `getfield #2` извлекает значение `count`, инструкция `iconst_1` и следующая за ней инструкция `iadd` прибавляют 1 к этому значению, и инструкция `putfield #2` записывает результат обратно в `count`. Этот шаблон часто называют *прочитать–изменить–записать*.

Теперь представьте, что метод `increment()` был вызван двумя потоками одновременно. Поток 1 выполнит инструкцию `getfield #2` и получит значение 42. Прежде чем он сможет сделать что-то еще, поток 2 также выполнит инструкцию `getfield #2` и получит то же значение 42. Далее события начинают развиваться не по нашему сценарию, потому что оба потока увеличат на 1 одно и то же число 42, и оба запишут один и тот же результат, 43, обратно в член `count`. В конечном итоге получается, что были выполнены две операции увеличения на 1, а значение увеличилось только на 1.

Решить эту проблему можно путем *синхронизации* доступа к члену `count`. Одно из решений состоит в том, чтобы использовать *встроенную блокировку*, имеющуюся в каждом объекте Java (иногда ее называют *мьютекс*, *монитор* или *критическая секция*), объявив метод `increment()` синхронизированным:

```
ThreadsLocks/CountingFixed/src/main/java/com/paulbutcher/Counting.java4
```

```
class Counter {  
    private int count = 0;  
    > public synchronized void increment() { ++count; }  
    public int getCount() { return count; }  
}
```

Теперь метод `increment()` будет приобретать блокировку объекта `Counter` при вызове и освобождать ее в момент возврата управления. То есть в каждый конкретный момент времени тело метода будет выполняться только в каком-то одном потоке. Любые другие потоки, вызвавшие метод, будут заблокированы до момента освобождения блокировки (далее в этой главе мы увидим, что для таких простых случаев, когда в конкуренцию вовлечена единственная переменная, отличной альтернативой блокировке объекта может служить пакет `java.util.concurrent.atomic`).

Как и следовало ожидать, новая версия каждый раз будет возвращать верный результат, 20000.

⁴ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/CountingFixed/src/main/java/com/paulbutcher/Counting.java>

Однако не все так прекрасно – наш новый код все еще содержит одну малозаметную ошибку, причину которой мы обсудим далее.

Загадочная память

Давайте добавим немного загадочности. Как вы думаете, что выведет следующий код?

```
ThreadsLocks/Puzzle/src/main/java/com/paulbutcher/Puzzle.java5
```

```
1 public class Puzzle {
-   static boolean answerReady = false;
-   static int answer = 0;
-   static Thread t1 = new Thread() {
5     public void run() {
-       answer = 42;
-       answerReady = true;
-     }
-   };
10  static Thread t2 = new Thread() {
-     public void run() {
-       if (answerReady)
-         System.out.println("The meaning of life is: " + answer);
-       else
15      System.out.println("I don't know the answer");
-     }
-   };
-
-   public static void main(String[] args) throws InterruptedException {
20     t1.start(); t2.start();
-     t1.join(); t2.join();
-   }
- }
```

Если вы сразу подумали о состоянии гонки, то вы совершенно правы. Вы можете увидеть текст «The meaning of life is: 42» (число жизни: 42) или «I don't know the answer» (я не знаю ответ), в зависимости от того, в каком порядке будут выполняться потоки. Но это еще не все! Вы можете увидеть и такой вариант:

```
The meaning of life is: 0
```

Но как?! Как `answer` может иметь нулевое значение, когда `answerReady` уже имеет значение `true`, если такое возможно, только если поменять местами строки 6 и 7.

⁵ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/Puzzle/src/main/java/com/paulbutcher/Puzzle.java>

Не торопитесь с выводами. Как оказывается, такое действительно возможно, например по следующим причинам:

- в ходе статической оптимизации кода компилятор может изменять порядок выполнения операций;
- в ходе динамической оптимизации виртуальная машина JVM также может изменять порядок выполнения операций;
- микропроцессор, выполняющий код, тоже может выполнять операции в другом порядке.

Хуже того, иногда такие эффекты вообще невидимы для других потоков. Представьте, что мы переписали метод `run()` потока `t2`, как показано ниже:

```
public void run() {  
    while (!answerReady)  
        Thread.sleep(100);  
    System.out.println("The meaning of life is: " + answer);  
}
```

В этом случае наша программа может никогда не завершиться, потому что `answerReady` может никогда не получить значение `true`.

Если первой вашей реакцией было заставить компилятор, JVM и микропроцессор убрать свои «грязные руки» от вашего кода, вас можно понять. Но, к сожалению, это невозможно – значительная доля прироста производительности, который мы наблюдаем последние годы, как раз и обусловлена подобными оптимизациями. В частности, от них зависит быстродействие компьютеров с разделяемой памятью. Соответственно, нам остается только бороться со следствиями.

Очевидно, что такое положение вещей не может оставаться бесконтрольным – нам нужно что-то, что могло бы сообщить, на что мы можем положиться, а на что – нет. Для этого нам требуется познакомиться с моделью памяти в Java.

Видимость памяти

Модель памяти в Java определяет, когда изменения в памяти, выполненные одним потоком, должны быть *видимы* остальным потокам.⁶ Результатом является полное отсутствие каких-либо гарантий, если только оба потока – читающий и пишущий – не будут синхронизированы.

⁶ <http://docs.oracle.com/javase/specs/jls/se7/html/jls-17.html#jls-17.4>

Мы уже видели один пример синхронизации, основанный на получении внутренней блокировки объекта. К числу других примеров синхронизации можно отнести запуск потока, ожидание завершения потока вызовом метода `join()` и применение множества классов из пакета `java.util.concurrent`.

Обратите особое внимание, что инструменты синхронизации должны использовать *оба* потока. Для решения проблемы недостаточно, если синхронизацию будет выполнять только поток, вносящий изменения. Это и есть причина малозаметной ошибки в коде. Чтобы избавиться от нее, одной только синхронизации метода `increment()` недостаточно – необходимо также синхронизировать метод `getCount()`. Если этого не сделать, поток, вызывающий `getCount()`, может увидеть устаревшее значение (так получилось, что в нашем примере метод `getCount()` не проявляет побочных эффектов, и только потому, что вызывается после вызова метода `join()`, но это – бомба замедленного действия, ждущая, когда кто-нибудь решит воспользоваться классом `Counter`).

Мы обсудили состояние гонки и видимость памяти – две основные проблемы, которые могут быть причиной неожиданного поведения многопоточной программы. Теперь перейдем к третьей проблеме: взаимоблокировке.

Несколько блокировок

Вас можно понять, если после прочтения предыдущих разделов вы решите, что единственный способ обезопасить себя в многопоточном мире – сделать все методы синхронизированными. К сожалению, все не так просто.

Во-первых, это было бы жутко неэффективно. Если синхронизировать каждый метод, большую часть своего времени потоки проводили бы в заблокированном состоянии, сводя на нет все преимущества конкурентного программирования. Но это одна из наименьших проблем – как только в программе появляется более одной блокировки (напомню, что в Java каждый объект имеет свою блокировку), открывается возможность взаимоблокировки потоков.

Продемонстрируем ситуацию взаимоблокировки на простом примере, часто используемом в академических статьях, посвященных проблемам реализации конкуренции – задаче «обедающих философов». Представьте пять философов, сидящих за круглым столом, и пять (не десять) палочек для еды, как показано ниже:

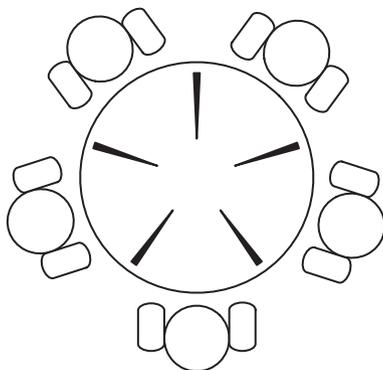


Рис. 3. Задача об обедающих философах

Каждый философ может или размышлять, или есть. Чтобы приступить к еде, он должен взять палочки с обеих сторон от себя, после чего он может поглощать пищу в течение некоторого времени (да, кстати, наши философы – мужчины; женщины ведут себя более разумно). Закончив, он кладет палочки на место.

Ниже показано, как можно было бы реализовать наших философов:

ThreadsLocks/DiningPhilosophers/src/main/java/com/paulbutcher/
Philosopher.java⁷

```

1 class Philosopher extends Thread {
-   private Chopstick left, right;
-   private Random random;
-
5   public Philosopher(Chopstick left, Chopstick right) {
-       this.left = left; this.right = right;
-       random = new Random();
-   }
-
10  public void run() {
-       try {
-           while(true) {
-               Thread.sleep(random.nextInt(1000)); // Размышлять
-               synchronized(left) { // Взять палочку слева
15              synchronized(right) { // Взять палочку справа
-                   Thread.sleep(random.nextInt(1000)); // Есть
-               }
-           }
-       }
-   }

```

⁷ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/DiningPhilosophers/src/main/java/com/paulbutcher/Philosopher.java>

```
-      }  
20    } catch (InterruptedException e) {}  
-    }  
- }
```

Строки 14 и 15 демонстрируют альтернативный способ приобретения внутренней блокировки объекта: `synchronized(object)`.

На моем компьютере эта программа благополучно выполнялась часами (я наблюдал за ее поведением в течение недели). А затем внезапно останавливалась.

Немного поразмыслив, легко можно понять причину — если все пять философов одновременно решат отвлечься от размышлений и приступить к еде, все они возьмут палочки слева и затем замрут — у каждого в левой руке будет одна палочка, и каждый будет ждать, пока освободится палочка, взятая философом справа. Возникает взаимоблокировка.

Опасность взаимоблокировки возникает всякий раз, когда поток пытается захватить более одной блокировки. К счастью, есть простое правило, гарантирующее избавление от взаимоблокировки: всегда приобретайте блокировки в глобальном фиксированном порядке.

Вот как это можно реализовать:

```
ThreadsLocks/DiningPhilosophersFixed/src/main/java/com/paulbutcher/  
Philosopher.java8
```

```
class Philosopher extends Thread {  
> private Chopstick first, second;  
private Random random;  
  
public Philosopher(Chopstick left, Chopstick right) {  
> if (left.getId() < right.getId()) {  
>     first = left; second = right;  
> } else {  
>     first = right; second = left;  
> }  
    random = new Random();  
}  
public void run() {  
    try {  
        while (true) {  
>             Thread.sleep(random.nextInt(1000)); // Размышлять  
>             synchronized (first) { // Взять первую палочку  
>                 synchronized (second) { // Взять вторую палочку  
>                     Thread.sleep(random.nextInt(1000)); // Есть
```

⁸ <http://media.pragprog.com/titles/pb7con/code/ThreadsLocks/DiningPhilosophersFixed/src/main/java/com/paulbutcher/Philosopher.java>

```

    }
  }
} catch (InterruptedException e) {}
}
}

```

Вместо того чтобы брать палочки слева, а затем справа, мы теперь берем первую и вторую палочки, используя член `id` класса `Chopstick`, чтобы гарантировать правильный порядок (нас не интересуют фактические идентификационные номера палочек – они нужны лишь для уникальности и упорядочения). Теперь можно не сомневаться, что взаимоблокировка больше не возникнет.



Вопрос Джо: можно ли использовать хэш-код объекта для упорядочения приобретения блокировок?

Вам часто будет встречаться совет использовать хэш-коды объектов для упорядочения приобретения блокировок, как показано ниже:

```

if (System.identityHashCode(left)
    < System.identityHashCode(right)) {
    first = left; second = right;
} else {
    first = right; second = left;
}

```

Преимущество такого подхода состоит в том, что он может применяться при работе с любыми объектами и позволяет избежать необходимости заводить дополнительное поле, упорядочивающее объекты. Но хэш-коды не гарантируют уникальности (весьма маловероятно, что два разных объекта будут иметь одинаковые хэш-коды, но такое может произойти). Поэтому, между нами говоря, я не стал бы использовать такое решение, разве только в случае, когда не остается ничего другого.

Как видите, совсем не сложно придерживаться правила глобального фиксированного порядка, когда приобретение блокировок выполняется в одном месте. Но в больших программах, где понятие глобальности слишком обширно, реализовать нечто подобное становится практически невозможно.