Проблемы многопоточности

В теоретических трудах по многопоточности вы можете встретить описание трех задач, которые по словам авторов, покрывают все возможные задачи многопоточности — задача производитель-потребитель, задача читатели-писатели, задача обедающие философы. Аллегория красивая и по своему достаточно неплохая, но на мой взгляд, для неокрепшего молодого программиста, совершенно ничего не говорящая. Посему опишу проблемы со своей колокольни. Проблем всего две.

Проблема первая — доступ к одному ресурсу из нескольких потоков. Мы уже описывали проблему с одной лопатой. Можете расширить вариант — есть один бак с водой (с одним краником), 25 жаждущих пить рудокопов и 5 кружек на всех. Придется договариваться, иначе смертоубийство может начаться. Причем надо не только сохранить кружки в целостности — надо еще организовать все так, чтобы всем удалось попить. Это частично переходит на проблему номер два.
Проблема вторая — синхронизация взаимодействия. Как-то мне предложили задачу — написать простую программу, чтобы два потока играли в пинг-понг. Один пишет «Пинг», а второй — «Понг». Но они это должны делать по очереди. А теперь представим, что надо сделать такую же задачу, но на 4 потока — играем пара на пару.

Т.е. постановка проблем весьма несложная. Раз — надо организовать упорядоченный и безопасный доступ к разделяемому ресурсу. Два — надо выполнять потоки в какой-то очередности.

Дело за реализацией. И вот тут нас подстерегает много сложностей, про которые с предыханием и говорят (и может не зря). Начнем с разделяемого ресурса.

Совместный ресурс для нескольких потоков

Предлагаю сразу продемонстрировать проблему на несложном примере. Его задача — запустить 200 потоков класса **CounterThread**. Каждый поток получает ссылку на один единственный объект **Counter**. В процессе выполнения поток вызывает у этого объекта метод **increaseCounter** одну тысячу раз. Метод увеличивает переменную **counter** на 1. Запустив 200 потоков мы ожидаем их окончания (просто засыпаем на 1 секунду — этого вполне достаточно). И в конце печатаем результат. Посмотрите код — по-моему, там все достаточно прозрачно:

```
1  package edu.javacourse.counter;
2
3  public class CounterTester
4  {
5      public static void main(String[] args) throws InterruptedException {
6          Counter counter = new Counter();
7          for(int i=0; i<200; i++) {
8              CounterThread ct = new CounterThread(counter);
9              ct.start();
10         }
11         Thread.sleep(1000);
12
13         System.out.println("Counter:" + counter.getCounter());
14     }
15 }
16
17 class Counter
18 {
19     private long counter = 0L;
```

```
20
21      public void increaseCounter() {
22          counter++;
23      }
24
25      public long getCounter() {
26          return counter;
27      }
28  }
29
30  class CounterThread extends Thread
31  {
32      private Counter counter;
33
34      public CounterThread(Counter counter) {
35          this.counter = counter;
36      }
37
38      @Override
39      public void run() {
40          for(int i=0; i<1000; i++) {
41              counter.increaseCounter();
42          }
43      }
44  }
```

По логике мы должны получить следующий результат — 200 потоков по 1000 прибавлений = 200000. Но, о ужас, это совсем не так. У меня результаты бывают разные, но явно не 200000. В чем же проблема ? Проблема в том, что мы из 200 потоков одновременно пытаемся вызвать метод **increaseCounter**. На первый взгляд в нем ничего страшного не происходит — мы просто прибавляем к переменной **counter** единицу. Что же тут такого ужасного ?
Ужасно то, что безобидный на первый взгляд код прибавления единицы, на самом деле выполняется не за один шаг. Сначала мы считываем значение переменной в регистр, потом прибавляем к нему единицу, потом записываем результат обратно в переменную. Как видите, шагов больше, чем один (по секрету — их даже больше чем три, которые я описал). И вот теперь представим, что два потока (или даже больше) одновременно считали значение переменной — например там было значение 99. Теперь оба потока прибавляют к 99 по единице, получают оба 100 и оба записывают это значение в переменную. Что там получается ? Нетрудно видеть, что будет 100. А должно быть 101. Может быть даже хуже, если какой-то поток «умудрился» считать 98 и «застрял» в очереди потоков на исполнение. Мы тогда даже 100 не получим. Неувязочка 🙂

Доступ к разделяемому ресурсу — это одна из самых больших проблем многопоточности. Потому что она весьма коварна. Можно сделать все очень надежно, но тогда производительность упадет. А как только даешь «слабину» (сознательно, для производительности), обязательно возникнет ситуация, что «слабина» вылезет во всей своей красе.

Волшебное слово — synchronized

Что можно сделать для того, чтобы избавиться от ситуации, в которую мы попали с нашими замечательными потоками. Давайте для начала немного порассуждаем. Когда мы приходим в магазин, то для оплаты мы подходим к кассе. Кассир одновременно обслуживает только одного человека. Мы все выстраиваемся к ней в очередь. По сути касса становится эксклюзивным ресурсом, которым может воспользоваться одновременно только один покупатель. В многопоточности предлагается точно такой же способ — вы можете определить некоторый ресурс как эксклюзивно предоставляемый одновременно только одному потоку. Такой ресурс называется «монитором». Это самый обычный объект, который поток должен «захватить». Все потоки, которые хотят получить доступ к этому монитору (объекту) выстраиваются в очередь. Причем для этого не надо писать специальный код — достаточно просто попробовать «захватить» монитор. Но как же обозначить это ? Давайте разбираться.
Предлагаю запустить наш пример, но с одним дополнительным словом в описании метода **increaseCounter** — это слово **synchronized**.

```
1  package edu.javacourse.counter;
2
3  public class CounterTester
4  {
5      public static void main(String[] args) throws InterruptedException {
6          Counter counter = new Counter();
7          for(int i=0; i<200; i++) {
8              CounterThread ct = new CounterThread(counter);
9              ct.start();
10         }
11         Thread.sleep(1000);
12
13         System.out.println("Counter:" + counter.getCounter());
14     }
15 }
16
17 class Counter
18 {
19     private long counter = 0L;
20
21     public synchronized void increaseCounter() {
22         counter++;
23     }
24
25     public long getCounter() {
26         return counter;
27     }
28 }
29
30 class CounterThread extends Thread
31 {
32     private Counter counter;
33
34     public CounterThread(Counter counter) {
35         this.counter = counter;
```

```
36      }
37
38      @Override
39      public void run() {
40          for(int i=0; i<1000; i++) {
41              counter.increaseCounter();
42          }
43      }
44  }
```

И … о чудо. Все заработало. Мы получаем ожидаемый результат — 200000. Что же делает это волшебное слово — **synchronized** ?
Слово **synchronized** говорит о том, что прежде чем поток сможет вызвать этот метод у нашего объекта, он должен «захватить» наш объект и потом выполнить нужный метод. Еще раз и внимательно (иногда предлагается несколько иной подход, который на мой взгляд, крайне опасен и ошибочен — чуть позже опишу) — сначала поток «захватывает» (лочит — от слова lock — замок, блокировать) объект-монитор (в нашем случае это объект класса **Counter**) и только после этого поток сможет выполнить метод **increaseCounter**. Эксклюзивно, в полном одиночестве без конкурентов.
Существует иная трактовка **synchronized**, которая может ввести в заблуждение — она звучит как-то так: в synchronized метод не может зайти несколько потоков одновременно. Это НЕВЕРНО. Потому как тогда получается, что если у класса несколько методов **synchronized**, то одновременно можно выполнять два разных метода одного объекта, помеченные как **synchronized**. Это НЕВЕРНО. Если у класса 2, 3 и более методов **synchronized**, то при выполнении хотя бы одного, блокируется весь объект. Это значит, что все методы, обозначенные как **synchronized** недоступны для других потоков. Если метод не обзозначен так. то не проблема — выполняйте на здоровье.
И еще раз — сначала «захватили», потом выполнили метод, потом «отпустили». Теперь объект свободен и кто первый успел из потоков его захватить — тот и прав.
В случае если метод объявлен как **static**, то объектом-монитором становится класс целиком и доступ к нему блокируется на уровне всех объектов этого класса.

При обсуждении статьи мне указали на некорректность, которую я сознательно допустил (для простоты), но наверно есть смысл о ней упомянуть. Речь идет о методе **getCounter**. Строго говоря, он тоже должен быть обозначен как **synchronized**, потому что в момент изменения нашей переменной какой-то другой поток захочет ее прочитать. И чтобы не было проблем, доступ к этой переменной надо делать синхронизированным во всех метода. Хотя что касается **getCounter**, то здесь можно использовать еще более интересную особенность — атомарность операций. О ней можно прочитать в статье Atomic access. Основная мысль — чтение и запись некоторых элементарных типов и ссылок производится за один шаг и в принципе безопасна. Если бы поле **counter** было например **int**, то читать можно было бы и не в синхронном методе. Для типа **long** и **double** мы должны объявить переменную **counter** как **volatile**. Почему это может быть любопытно — надо учесть, что **int** состоит из 4 байт и можно представить ситуацию, что число будет записано не за один шаг. Но это исключительно теоретически — JVM нам гарантирует, что чтение и запись элементарного типа int делает за один шаг и ни один поток не сможет вклиниться в эту операцию и что-то испортить.

Есть и другой способ использования слова **synchronized** — не в описании метода, а внутри кода. Давайте еще раз изменим наш пример в части метода **increaseCounter**.

```
1  package edu.javacourse.counter;
```

```
2
3  public class CounterTester
4  {
5      public static void main(String[] args) throws InterruptedException {
6          Counter counter = new Counter();
7          for(int i=0; i<200; i++) {
8              CounterThread ct = new CounterThread(counter);
9              ct.start();
10         }
11         Thread.sleep(1000);
12
13         System.out.println("Counter:" + counter.getCounter());
14     }
15 }
16
17 class Counter
18 {
19     private long counter = 0L;
20
21     public void increaseCounter() {
22         synchronized(this) {
23             counter++;
24         }
25     }
26
27     public long getCounter() {
28         return counter;
29     }
30 }
31
32 class CounterThread extends Thread
33 {
34     private Counter counter;
35
36     public CounterThread(Counter counter) {
37         this.counter = counter;
38     }
39
40     @Override
41     public void run() {
42         for(int i=0; i<1000; i++) {
43             counter.increaseCounter();
44         }
45     }
46 }
```

Отдельное использование **synchronized** делает по сути тоже самое — сначала блокирует/лочит переданный ему в скобках объект и потом начинает выполнять код, который находится внутри **synchronized**. Здесь надо учесть, что блокировка будет происходит не на входе в метод, а при входе в блок **synchronized**. Кстати за счет этого можно сделать синхронизацию внутри одного объекта по нескольким. Например, два метода блокируют один объект, а два другие — другой. Тогда вы сможете вызывать методы из разных групп одновременно. Я такое использовал в своей практике.
В принципе идея **synchronized** на этом исчерпывается. Теперь важным становится правильное его использование. Потому что с одной стороны может возникнуть желание все методы делать **synchronized**, но это будет влиять на производительность — думаю, что это очевидно. С другой стороны — могут возникнуть сложности при неаккуратном обращении с несинхронизированными объектами. Так что будьте бдительны.

Потокобезопасные классы — thread safe

Для Java написано достаточно большое количество классов, которые используются для работы с данными — одни коллекции чего стоят. Когда вы будете смотреть документацию по таким классам, то теперь вы будете понимать, почему выделяются потокобезопасные и потоконебезопасные. Например, коллекция **ArrayList** — потоконебезопасная. а класс **Vector** — потокобеопасный. Но зачем так сделано ? Основное объяснение — производительность. Когда у вас обращение к одному объекту требует предварительной блокировке и только потом можно вызвать метод — это падение производительности. Т.к. коллекции могут содержать иногда миллионы записей, то крошечная задержка при каждом обращении к коллекции выливается в большие затраты в целом. Вам придется выбирать — если вы используете объект только внутри одного потока, то есть смысл не делать его потокобезопасным. В общем это достаточно нетривиальная задача, которая требует опыта и вдумчивого отношения.
Различные аспекты и хитрости использования и построения многопоточных приложений описаны в некоторых специализированных статьях и книгах. Я могу посоветовать «Java Concurrency in Practice» Brian Goetz, «Concurrent Programming in Java» Doug Lea. Можно найти еще что-нибудь.

И теперь нас ждет следующая статья: Работаем с XML.

23 comments to *Многопоточность и синхронизация*

- 
  Август 1, 2015 at 10:26
  u-235 says:

  Метод getCounter можно просто объявить в документации как потоконебезопасный. А если в классе Counter реализовать метод setCounter, то делать эти два метода синхронизированными не имеет никакого смысла.

  Reply

  - 

Август 1, 2015 at 14:39
admin says:

Ну здесь же была задача показать, что много потоков портят данные. И уже после этого показать, как это можно решить. У меня такое ощущение, что пояснительный текст никто не читает, а смотрят исключительно код и делают выводы только по нему. Да, код в какой-то степени спорен, но он показывает именно ту идею, которую я пытался показать.

Reply

- 
  Август 1, 2015 at 16:56
  u-235 says:

  Целью моего высказывания была не критика, а дальнейшее обсуждение проблем многопоточности.

  Reply

  - 
    Август 2, 2015 at 11:28
    admin says:

    Дальнейшее обсуждение конечно же возомжно. Я же в дальнейших комментариях об этом написал. Посмотрим. Мне хочется дать широкий взгляд на важный набор технологий, но насколько надо будет углубляться, точно не знаю — будем работать.

    Reply

- 
  Октябрь 7, 2016 at 10:55
  Derrt says:

  Прикольно, а у меня первый пример дал ровно 200000. =)))

  Reply

Октябрь 7, 2016 at 12:44
admin says:

Такое бывает иногда — я не разбирался в причинах.

Reply

Ноябрь 23, 2016 at 18:41
Владимир says:

Уважаемый admin, в классе CounterTester в строке 11: Thread.sleep(1000); Вопрос: какой из 200 запущенных раннее в цикле потоков ставит на паузу вызов этого метода (вопрос возник в связи с тем что он находится вне тела цикла после инициализации потоков).
(Закоментил его — Counter «не успевал» сосчитать до 200000.)

Reply

Ноябрь 24, 2016 at 09:46
admin says:

Вызов Thread.sleep() останавливает текущий поток. Т.е. те 200, что запущены, честно отработали. А поток внутри метода main ждет.

Reply

Ноябрь 24, 2016 at 18:13
Владимир says:

Уважаемый Admin, правильно ли я понимаю, что есть поток внутри метода main, в котором (в потоке) запускается код (цикл), в свою очередь запускающий 200 потоков, которые отрабатывают; при этом мы ставим на паузу поток метода main, чтобы он дождался

отработки этих 200 потоков перед тем как сам (метод main) завершит свое действие.
Или же поток метода main запускается, параллельно с ним запускаются наши 200; Thread.sleep(1000) дает возможность им отработать, затем поток метода main возобновляется, выкидывает в консоль сообщение и далее закрывается.
Извините за вопрос, в начальном курсе Java вы все это «разжевываете»?

Reply

Ноябрь 24, 2016 at 18:19
admin says:

Да, метод main выполняется в отдельном потоке. Да, он запускает другие потоки и сам засыпает для демонстрации, что другие потоки завершились.
Потоки в принципе «равны». И поток метода main и все потоки, которые он запустил.
Да, в начальном курсе мы это все разжевываем и делаем много примеров.

Reply

Ноябрь 25, 2016 at 19:19
Владимир says:

Благодарю за терпимость.

Ноябрь 23, 2016 at 19:43
Владимир says:

В предпоследнем абзаце предпоследнего параграфа: «Кстати за счет этого можно сделать синхронизацию внутри одного объекта по нескольким. Например, два метода блокируют один объект, а два другие — другой. Тогда вы сможете вызывать методы из разных групп одновременно.»
Ув.Админ. Ничего не понятно из сказанного, можно то же но другими словами.

Reply

Ноябрь 24, 2016 at 09:48
admin says:

Другими словами вряд ли станет понятнее — проще код написать. Я подумаю.

Reply

Январь 20, 2017 at 15:23
JohnDoe says:

Такая задача: есть МФУ, оно может печать или сканировать одновременно. Есть два принтера и два сканера. Написать программу так, чтобы в один момент времени МФУ был доступен только одному принтеру и одному сканеру, но не двум принтерам \ сканерам одновременно. Т.е. в один момент времени может печататься страница, и параллельно сканироваться, но нельзя одновременно печатать \ сканировать две страницы. Работа принтера \ сканера выводится в консоль: «принт..», «скан..». Как-то так, кривовато написано, кто понял — поясните детальнее.

Reply

Январь 8, 2017 at 13:58
Максим says:

Первый пример работает без synchronized если основной поток main будет ждать завершения порожденных им потоков до строки вывода значения переменной counter с помощью вызова join() на порожденных потоках в цикле main. Как ни странно работает не только с типом int, но и long переменной counter.

Reply

Июль 24, 2018 at 19:08

Иван says:

Не знаю как у вас, а у меня код из первой картинки (самая верхняя без synchronized) всегда выдает ответ 200 000.

Reply

- 
  Июль 28, 2018 at 04:23
  admin says:

  Зависит нередко из под какой IDE — например под IDEA достаточно часто и правда выдает точный ответ.

  Reply

Сентябрь 6, 2018 at 14:57
andy says:

Вы указываете, что:
«Основная мысль — чтение и запись некоторых элементарных типов и ссылок производится за один шаг и в принципе безопасна. Если бы поле counter было например int, то читать можно было бы и не в синхронном методе. Для типа long и double мы должны объявить переменную counter как volatile.»
Однако, у вас поле counter типа long, но вы так и не добавили для этого поля модификатор volatile. — где логика?

Reply

- 
  Сентябрь 6, 2018 at 15:46
  admin says:

  Так метод синхронизированный — зачем делать volatile ?

  Reply

Сентябрь 6, 2018 at 19:52
andy says:

Речь идет о методе getCounter? — он не синхронизированный. Выше указано, что «он тоже должен быть обозначен как synchronized» и в этом контексте дальше речь шла про volatile.

Reply

Сентябрь 9, 2018 at 23:52
admin says:

Ну да — он тоже должен быть синхронизирован. Это указано — я поэтому в коде и не написал. Специально. Для внимательных.

Reply

Сентябрь 7, 2018 at 14:38
andy says:

Синхронизован только increaseCounter(), а как насчет getCounter()? Вы же сами говорите «он тоже должен быть обозначен как synchronized» и дальше речь идет про атомарность операций, к которой и относился мой коммент. Т.о. тут, как я понимаю, нужно делать или synchronized getCounter() или volatile long counter.

Reply

Сентябрь 9, 2018 at 23:54
admin says:

Да. И именно отсутствие synchronized должно был пробудить желание попробовать и исправить. Или написать такое замечание. Вы написали — значит разобрались. И это хорошо.

Reply

Leave a reply

Comment

You may use these HTML tags and attributes: `<a href="" title=""> <abbr title=""> <acronym title=""> <b> <blockquote cite=""> <cite> <code class="" title="" data-url=""> <del datetime=""> <em> <i> <q cite=""> <s> <strike> <strong> <pre class="" title="" data-url=""> <span class="" title="" data-url="">`

Имя *

E-mail *

Сайт

7 + два =

Add comment