Многопоточность в Java

1. [Введение](#Введение)
2. [Создание потока 1-й способ (new Thread).](#Созданиепотока1йспособnewThread)
3. [Метод sleep()](#Методsleep)
4. [Использование интерфейса Runnable (2-й способ).](#ИспользованиеинтерфейсаRunnable2йс)
5. [Выводы](#Выводы)
6. [Ключевое слово volatile](#Ключевоесловоvolatile)
7. [Описание проблем при работе с многопоточными приложениями](#Описаниепроблемприработесмногопоточ)
8. [Пример использования volatile](#Примериспользованияvolatile)

Введение

В Java-машине, все потоки выполняются в главном потоке. Многопоточность представлена в виде виртуальных параллельных потоков. Процессор очень быстро переключается между потоками, чтобы выполнить отдельные участки кода.

Потоки выполняются в одном процессе.

Процесс – это один экземпляр выполняемой программы. Для каждого процесса выделяется виртуальное адресное пространство.

Создание потока 1-й способ (new Thread).

**public class** MyThread **extends** Thread {}

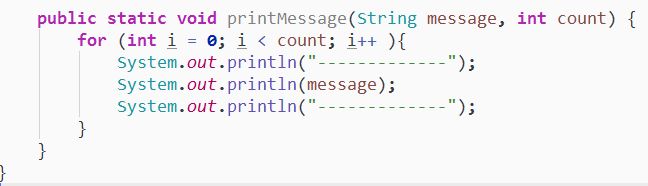
В данном классе, мы создадим *новый* поток, который будет выполняться в главном потоке.

Главный поток выполняется в точке входа в приложение (то есть в методе main()).

ОС сама берет на себя роль, распределять ресурсы процессора, между потоками. Если ОС «решит», что для вашего приложения не следует выделять для каждого потока одно ядро процессора (если у вас процессор многоядерный), тогда ваше приложение будет выполняться на одном ядре, просто ОС будет очень быстро переключаться между потоками, и каждому выделять процессорное время.



Внутри данного метода, мы запускаем код, который будет выполняться в отдельном потоке.



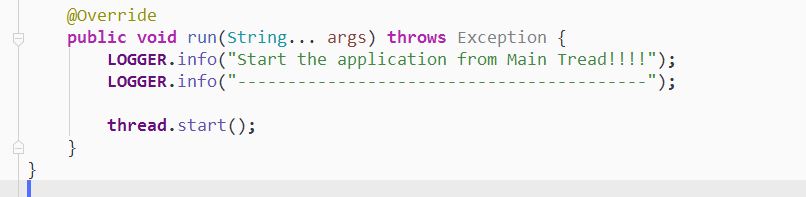


Так как используется Spring boot, поэтому код будем запускать из другого метода, но все равно, новый поток будет выполняться внутри главного потока.

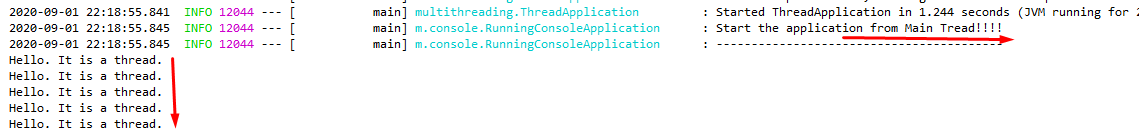


Важно. Созданный поток запускается через метод *start()*, который определен в базовом классе Thread.

метод *start()* – создает новый поток.

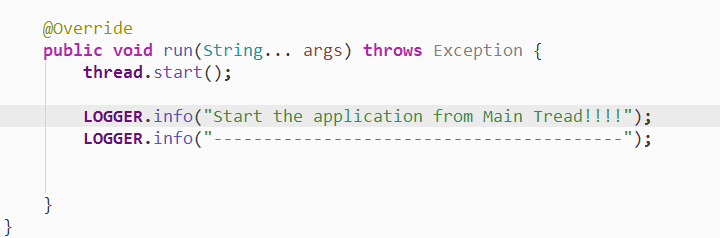


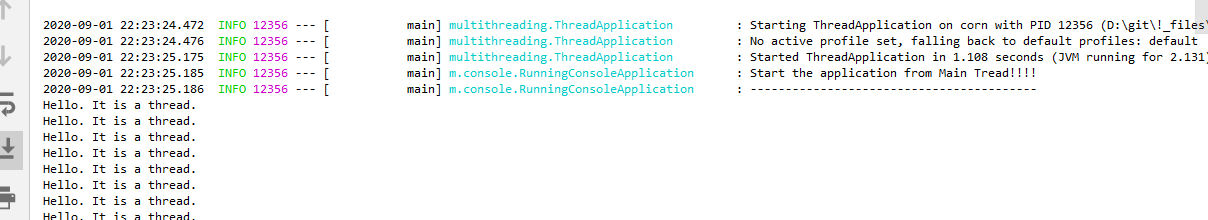
Сначала запустился главный поток и в нем был создан еще один поток, который и выполнил свой код.



Поменяем местами запуск кода.

Сначала попробуем запустить созданный поток, и только потом команды в главном потоке.



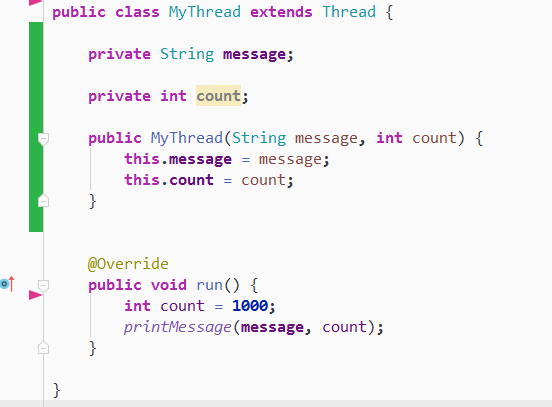


То есть потоки в Java не синхронизированы и они могут выполняться в любом порядке.

То есть главный поток создал новый поток и выполняет свои инструкции, а созданный поток выполняет свои и эти потоки не связаны между собой, они выполняются отдельно друг от друга, забирая процессорное время.

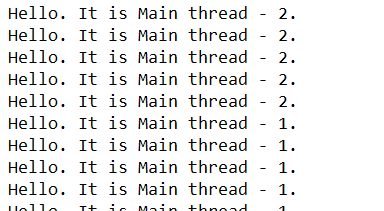
Создание 2-го потока

Для того, чтобы мы могли использовать два различных компонента одного и того же типа, нужно указать для Spring, что он должен создавать компонент, каждый раз, когда к нему обращаются в коде.



Будем переиспользовать класс.





 Как видим, потоки выполняются не по порядку, но вперемешку. Сначала может несколько раз выполниться первый поток, затем несколько раз второй и т.д.

То есть запуск одного потока, происходит через некоторое время относительно другого потока.

Но если бы потоки были синхронизированы, тогда каждый поток выполнялся по очереди:

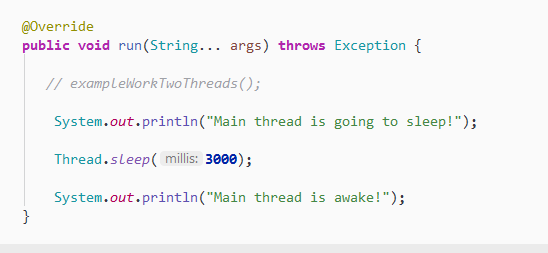
сначала 1-й поток, затем 2-й, затем 1-й и т.д.

В данном случае, потоки получают процессорное время – случайным образом.

Метод sleep()

Данный метод sleep() заставляет приостанавливаться указанный поток, на определенное время.

 Для начала, попробуем приостановить Главный поток (в методе *main()*).



Метод sleep() в качестве аргумента, принимает в миллисекундах, тип ***long***: время, на которое нужно приостановить поток.

Пробуем приостановить поток на 3 секунды.

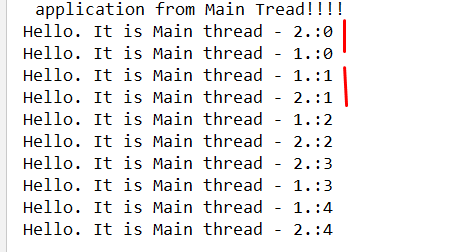


Использование метода sleep() для различных потоков.

**public class** OutputStream {  
  
 **public static void** printMessage(String message**, int** i) {  
 System**.***out***.**println(message + ":" + i)**;** }  
}

**public class** MyThread **extends** Thread {  
  
 **private** String **message;  
  
 private int count;  
  
 public** MyThread(String message**, int** count) {  
 **this.message** = message**;  
 this.count** = count**;** }  
  
  
 @Override  
 **public void** run() {  
  
 **for**(**int** i = **0;** i <= **count;** i++){  
  
 **try** {  
 Thread**.***sleep*(**3000**)**;** } **catch** (InterruptedException e) {  
 e**.**printStackTrace()**;** }  
  
 *printMessage*(**message,** i)**;** }  
 }  
  
}

@Component  
**public class** RunningConsoleApplication **implements** CommandLineRunner {  
  
 Logger **LOGGER** = LoggerFactory**.***getLogger*(RunningConsoleApplication**.class**)**;  
  
 public** RunningConsoleApplication() {  
 }  
  
 @Override  
 **public void** run(String... args) **throws** Exception {  
  
 exampleWorkTwoThreads()**;** }  
  
  
  
 **void** exampleWorkTwoThreads() **throws** InterruptedException {  
 String message\_1 = "Hello. It is Main thread - 1."**;** String message\_2 = "Hello. It is Main thread - 2."**;  
 int** count = **100;  
  
 LOGGER.**info("Start the application from Main Tread!!!!")**;** Thread thread\_1 = **new** MyThread(message\_1**,** count)**;** thread\_1**.**start()**;** Thread thread\_2 = **new** MyThread(message\_2**,** count)**;** thread\_2**.**start()**;** }  
  
}



Это позволило даже синхронизировать потоки.

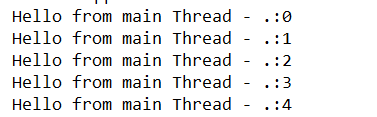
Использование интерфейса Runnable (2-й способ).

**public class** Runner **implements** Runnable {  
  
**private int count;  
  
private** String **message;  
  
public** Runner(**int** count**,** String message) {  
 **this.count** = count**;  
 this.message** = message**;**}

@Override  
 **public void** run() {  
 **for**(**int** i = **0;** i <= **count;** i++){  
 **try** {  
 Thread**.***sleep*(**3000**)**;** } **catch** (InterruptedException e) {  
 e**.**printStackTrace()**;** }  
  
 *printMessage*(**message,** i)**;** }  
 }  
}

запускаемый код.

@Override  
**public void** run(String... args) **throws** Exception {  
  
 *// exampleWorkTwoThreads();* withRunnable()**;**}  
  
  
**void** withRunnable(){  
  
 String message = "Hello from main Thread - ."**;  
 int** count = **100;** Thread thread = **new** Thread(**new** Runner(count**,** message))**;** thread**.**start()**;**}



Выводы

* Потоки в Java, по умолчанию – не синхронизированы.
* Пока не выполняться все дочерние потоки, главный поток в методе main() – не будет завершен.
* Для того, чтобы приостановить поток, используется метод sleep().
* Для создания потоков либо используется наследование от класса Thread(), либо можно реализовать интерфейс Runnable (что предпочтительнее).

Когда вы расширяете класс ***Thread***, каждый ваш поток **создает** уникальный объект и связывается с ним.

Но когда вам нужно будет наследоваться от какого-то класса, то вы не можете вторым базовым классом определить ***Thread***, так как множественное наследование запрещено.

Тогда предпочтительнее использовать интерфейс Runnable, что дает нам гибкость в управлении потоками.

 Когда вы реализуете Runnable, он разделяет один и тот же объект на несколько потоков.

***Thread*** - это абстракция над физическим потоком.

Runnable - это абстракция над выполняемой задачей.

Использование Runnable состоит в том, что это позволяет логически отделить выполнение задачи от логики управления потоками.

Ключевое слово volatile

Данное слово, устанавливается около переменной (свойства объекта), когда эта переменная является общим ресурсом для нескольких потоков.

В компьютере есть два вида памяти :

* глобальная (обычная) и
* встроенная в процессор.

Встроенная в процессор делится на:

* регистры, затем
* кэш первого уровня (L1),
* кэш второго уровня (L2) и
* третьего уровня (L3).

Эти виды памяти отличаются по скорости работы.

Самая быстрая и самая маленькая память – это регистры, затем идет кэш процессора (L1, L2, L3) и, наконец, глобальная память (самая медленная).

 Скорость работы глобальной памяти и кэша процессора сильно отличаются, поэтому Java-машина позволяет каждой нити хранить самые часто используемые переменные в локальной памяти нити (в кэше процессора).

Есть одна маленькая проблемка.

Когда две нити работают с одной и той же переменной, каждая из них может сохранить ее копию в своем ***внутреннем локальном кэше***.

И тогда может получится такая ситуация, что одна нить переменную **меняет**, а **вторая** *не видит этого изменения*, т.к. по-прежнему работает со своей копией переменной.

 На этот случай разработчики Java предусмотрели специальное ключевое слово – volatile.

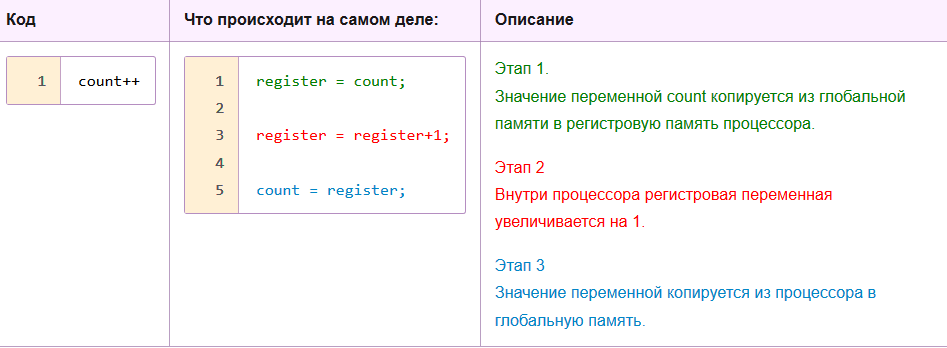
Если есть переменная, к которой обращаются из разных нитей, ее нужно пометить модификатором volatile, **чтобы Java-машина** не помещала ее в кэш.

Вот как это обычно выглядит:



 Модификатор volatile гарантирует только безопасное **чтение**/запись переменной, ***но*** *не ее изменение*.

Вот как происходит изменение переменной



Таким образом, значения **копируются** туда-сюда: из памяти в процессор и обратно.

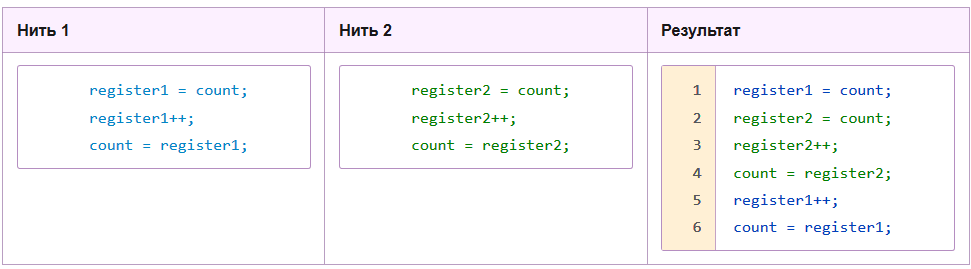
Так вот, модификатор volatile, гарантирует, что при обращении к переменной count она будет прочитана из памяти (этап 1).

А если какая-то нить захочет присвоить ей новое значение, то оно обязательно окажется в глобальной памяти (**этап 3**).

 Но Java-машина не гарантирует, что не будет переключения нитей между этапами 1 и 3.

Увеличение переменной на 1 – это фактически три операции.

Если две нити одновременно захотят исполнить count++, то они могут помешать друг другу.



 Как видно, когда второй поток изменил значение переменной и записал обратно в глобальную память, то второй поток, все еще работает со старым значением, и когда он сделал свои вычисления, тогда он запишет совсем другое значение в count, которая находится в глобальной памяти.

Здесь видим, что volatile не защищает от несогласованности данных, между несколькими потоками, когда они изменяют переменную.

Поэтому, для того, чтобы избежать такой проблемы, блок кода, который работает с переменной count, нужно помещать в блок synchronized или указывать данное слово перед методом, в котором будет производится работа с переменной count.

Описание проблем при работе с многопоточными приложениями

При создании многопоточных приложений мы можем столкнуться с двумя серьезными проблемами.

**Во-первых, в процессе работы многопоточного приложения** *разные потоки* **могут *кэшировать значения переменных*** (мы уже говорили об этом [в лекции «Применение volatile»](https://javarush.ru/quests/lectures/questmultithreading.level06.lecture04)).

Возможна ситуация, когда один поток **изменил** *значение* переменной, а второй поток не увидел этого изменения, потому что работал со своей, кэшированной копией переменной.

Естественно, последствия могут быть серьезными.

Представь, что это не просто какая-то «переменная», а, например, баланс твоей банковской карты, который вдруг начал *random* скакать туда-сюда.

**Во-вторых, в Java *операции чтения и записи* полей всех типов,** *кроме* **Long и Double, являются атомарными.**

*Что такое атомарность?*

Ну, например, если ты в одном потоке меняешь значение переменной **int**, а в другом потоке читаешь значение этой переменной, ты получишь либо ее старое значение, либо новое — то, которое получилось после изменения в потоке 1.

 Никаких «промежуточных вариантов» там появиться *не может*.

Однако **Long** и **Double** – не являются типами полей объектов, с которыми о ***операции чтения и записи* этих полей, - являются** атомарными.

* Почему? Из-за кроссплатформенности.

Java-приложение запускается **на абсолютно разных платформах**.

Например, на операционных системах Windows, разных вариантах Linux или MacOS, и везде это приложение будет стабильно работать.

**Long** и **Double** — самые «тяжеловесные» примитивы в Java: они весят по 64 бита.

И в некоторых 32-битных платформах просто *не реализована атомарность* чтения и записи 64-битных переменных.

**Такие переменные** читаются и записываются в две операции.

Сначала в переменную записываются **первые** 32 бита, потом еще 32.

Например, если на такой платформе мы попытаемся записать в переменную число

**9223372036854775809**

— оно будет занимать 64 бита.

В двоичной форме оно будет выглядеть так:

1000000000000000000000000000000000000000000000000000000000000001

Первый поток начнет запись этого числа в переменную, и сначала запишет первые 32 бита:

10000000000000000000000000000000 ,

а потом вторые 32 бита:

0000000000000000000000000000001

И **в этот промежуток** может вклиниться второй поток, и прочитать промежуточное значение переменной —

10000000000000000000000000000000,

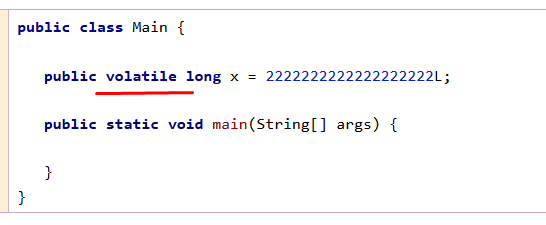
первые 32 бита, которые уже были записаны.

В десятичной системе это число равняется **2147483648** (то есть оно уже отличается от того числа с которым работаем).

То есть мы всего лишь хотели записать число *9223372036854775809* в переменную, но из-за того, что эта операция на некоторых платформах является *не атомарной*, у нас из ниоткуда возникло «левое», ненужное нам число **2147483648**, и неизвестно как оно повлияет на работу программы.

 *Второй поток* просто прочитал значение переменной до того, как оно окончательно записалось, то есть первые 32 бита он увидел, а *вторые* 32 бита — нет.

Решение проблемы

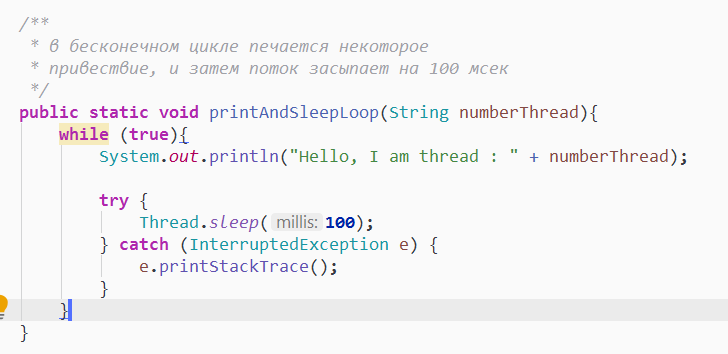


Мы объявляем в нашей программе какую-то переменную, со словом volatile…

… это означает, что:

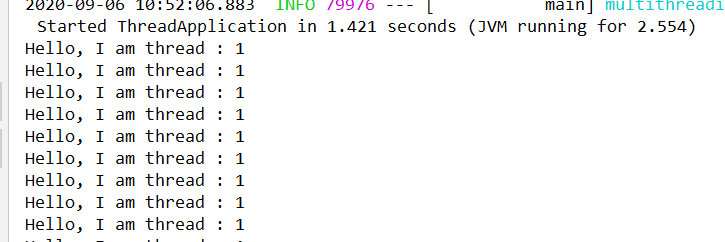
1. Она всегда будет атомарно читаться и записываться. Даже если это 64-битные **Long** и **Double**.
2. Java-машина *не будет помещать ее в кэш*. Так что ситуация, когда 10 потоков работают со своими локальными копиями исключена.

Пример использования volatile









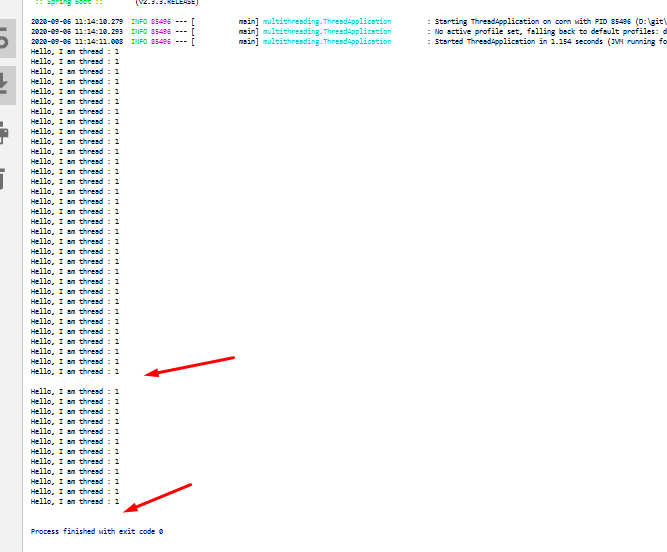
Теперь нужно остановить данный цикл из главного потока.

Модифицируем класс, который создает новый поток.

**public class** MyThread **extends** Thread {  
  
 **private** String **numberThread;  
  
 private boolean isRunning** = **true;  
  
 public** MyThread(String numberThread) {  
 **this.numberThread** = numberThread**;** }  
  
 @Override  
 **public void** run() {  
  
 printAndSleepLoop(**numberThread**)**;** }  
  
  
 **public void** setRunningAndShutdown(**boolean** running) {  
 **isRunning** = running**;** }  
  
  
 */\*\*  
 \* в бесконечном цикле печается некоторое  
 \* привествие, и затем поток засыпает на 100 мсек  
 \*/* **public void** printAndSleepLoop(String numberThread){  
 **while** (**isRunning**){  
 System**.***out***.**println("Hello, I am thread : " + numberThread)**;  
  
 try** {  
 Thread**.***sleep*(**100**)**;** } **catch** (InterruptedException e) {  
 e**.**printStackTrace()**;** }  
 }  
 }  
}

Теперь пробуем остановить поток.





Но вы можете заметить, что срабатывание произошло не сразу.

 Почему?

Причина в том, что мы из главного потока, меняли значение поля

**private boolean isRunning** = **true;** ,

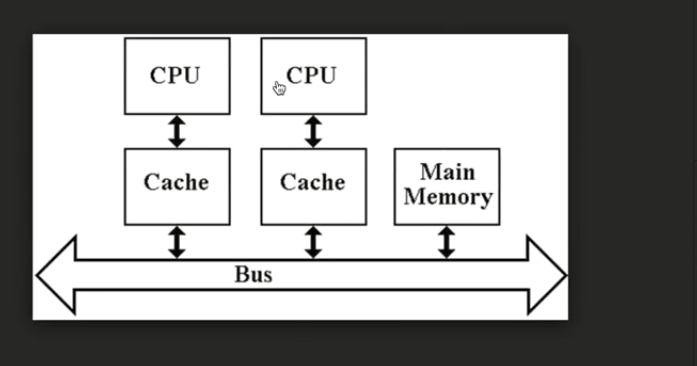
объекта MyThread.

Но дочерний поток, также работает с этим полем.

Таким образом, когда главный поток изменил это поле, то его значение пока было в кэше процессора и дочерний поток снова успел захватить поле **isRunning** и оно было записано старым значением, которое еще хранит дочерний поток.

Но второй раз, главный поток смог переписать значение поля **isRunning**, в основной памяти и дочерний поток останавливается.

 То есть мы получили проблему из-за плохой когерентности кэшей.



Здесь мы видим 2 ядра в процессоре и у каждого ядра свой кэш. Данные из кэшей процессоров, будут сливаться в главную память.

Исходные значения, ядра читают из главной памяти и помещают в свой кэш.

В программе выше, мы видели что у нас есть главный поток и дочерний поток. Каждый поток выполнялся на своем ядре.

Каждый поток кэшировал значения поля **isRunning**, в кэш своего ядра и работал со в данными из своего кэша.

Таким образом, когда главный поток изменил значение переменной **isRunning**, но значение не было видно из другого кэша, так как из первого кэша, измененное значение не попало в главную память.

Поэтому дочерний поток продолжал работать со значением поля **isRunning**, из своего кэша.

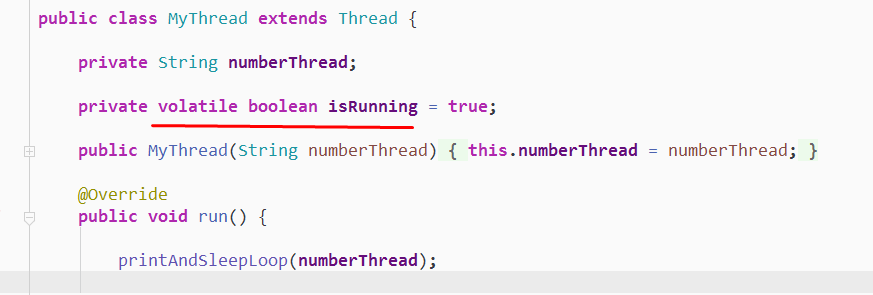
Затем возможно для ядра, в котором работает дочерний поток, была команда, проверить главную память, так как там поменялось значение поля **isRunning**.

Тогда ядро, в котором работает дочерний поток, запросило данные из главной памяти, прочитало измененное значение из переменной **isRunning** , и только после этого, дочерний поток прекратил свою работу.

Когерентность – это совпадение кэшей… свойство кэшей, означающее целостность данных, хранящихся в локальных кэшах для разделяемого ресурса

Как исправить эту проблему?

Для того, чтобы избежать данной проблемы, мы должны сообщить Java-машине, что переменную **isRunning**, запрещено помещать в локальные кэши, а можно работать с данной переменной только из главной памяти.



**volatile** – указывает на то, что данная переменная может быть изменена и ее не нужно кэшировать.

Таким образом, данное слово, гарантирует правильную когерентность кэшей ядер процессора, при работе с такой переменной.