Source: http://digitalhealthage.com/how-digital-technologies-can-support-mental-health-patients/

Handling Java Memory Consistency with happens-before relationship

Kasun Dharmadasa
Dec 31, 2018 · 5 min read

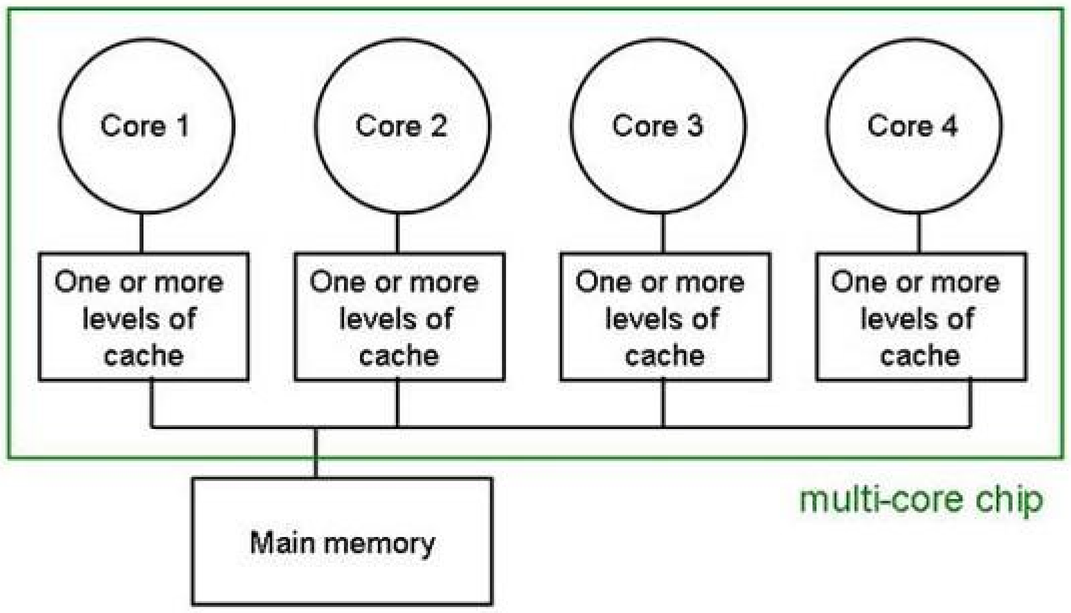
If you are developing multi-threaded applications in Java, you need to have an understanding of how the shared variables are handled inside the memory of Java. One important factor is the happens-before relationship. In order to understand the happens-before relationship in Java, you need to be familiar with the concept of visibility in concurrent programming.

Visibility

> *If an action in one thread is* visible *to another thread, then the result of that action can be observed by the second thread.*

To understand more on the above statement, let's have a look at the architecture of a modern shared-memory multiprocessor.

Almost all of the computers today have multiple cores inside their processors, inside their processors, each capable of handling multiple threads of execution. For each of this core, there exist several levels of caches.


Core 1
Core 2
Core 3
Core 4
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
multi-core chip

Source: https://wiki.sei.cmu.edu/confluence/display/java/Concurrency%2C+Visibility%2C+and+Memory

The visibility of writing operations on shared variables can cause problems during delays that occur when writing to the main memory due to caching in each core. This can result in another thread reading a stale value (not the last updated value) of the variable.

Let’s consider a situation where two threads perform a read an write operation to the same variable.

```java
public class StopThread {
        private static boolean stopRequested;
        public static void main(String[] args)
                throws InterruptedException {
            Thread backgroundThread = new Thread(new Runnable() {
                public void run() {
                    int i = 0;
                    while (!stopRequested)
                        i++;
                }
            });
            backgroundThread.start();
            TimeUnit.SECONDS.sleep(1);
            stopRequested = true;
        }
    }
```

memsharesample(bad).java hosted with ❤ by GitHub view raw

In the above example from the book “Effective Java by Joshua Bloch”, there is a background thread (backgroundThread) that will increment the value of “i” until the “stopRequested” boolean becomes true. After starting the thread by the main thread of the program, it sleeps for 1 second and makes the “stopRequested” to true.

What would be the outcome? Ideally, the program should run for 1 second and after the “stopRequested” has become true, the “backgroundThread” should end, terminating the whole program.

But if you run the above on a computer with multiple cores, you will observe that the program keeps on executing without getting terminated. The problem occurs with the write operation on the “stopRequested” variable. There is no guarantee that the change of the value in “stopRequested” variable (from the main thread) becoming **visible** to the “backgroundThread” that we created. As the write operation to the “stopRequested” variable to true from the main method is invisible to the “backgroundThread”, it will go into an infinite loop.

As the main thread and our “backgroundThread” is running on two different cores inside the processor, the “stopRequested” will be loaded into the cache of the core that executes the “backgroundThread”. The main thread will keep the updated value of the “stopRequested” value in a cache of a different core. Since now the “stopRequested” value resides in two different caches, the updated value may not be visible to the “backgroundThread”.

To avoid these type of memory inconsistencies, Java has introduced the happens-before relationship.

. . .

Happens-before relationship

Java defines a happens-before relationship as follows.

> *Two actions can be ordered by a* happens-before *relationship. If one action* happens-before *another, then the first is visible to and ordered before the second.*

According to this, if there is a happens-before relationship between a write and read operation, the results of a write by one thread are guaranteed to be visible to a read by another thread. Therefore, we will be able to maintain the memory consistency if we are able to have the happens-before relationship between our actions.

Synchronizing

“Synchronized” keyword is widely used to achieve mutual exclusion among threads. This means that using synchronized keyword we have the ability to limit the access of a particular block of code or a method to only one thread. A single lock is passed around the threads that wish to access a particular synchronized block or a method. Each thread will have wait until the other thread finishes executing the synchronized block/method and release the lock.

However, synchronizing has another important use. It can also be used to achieve a happens-before relationship between code blocks or methods. If there are two synchronized blocks/methods having the same lock, there is a happens-before relationship between the actions inside the synchronization blocks/methods. This is due to the fact that an unlock on an object lock (exiting synchronized block/method) happens before every subsequent acquiring on the same object lock.

Let’s change our initial code to include synchronized methods for read and write operations of the “stopRequested” variable

```
public class StopThread {
        private static boolean stopRequested;
        private static synchronized void requestStop() {
            stopRequested = true;
        }
        private static synchronized boolean stopRequested() {
            return stopRequested;
        }
        public static void main(String[] args)
                throws InterruptedException {
            Thread backgroundThread = new Thread(new Runnable() {
                public void run() {
                    int i = 0;
                    while (!stopRequested())
                        i++;
                }
            });
            backgroundThread.start();
            TimeUnit.SECONDS.sleep(1);
            requestStop();
        }
    }
```

memsharesample(sync).java hosted with ❤ by GitHub view raw

Since now both read and write operations to the “stopRequested” variable is synchronized, a happens-before relationship has been established between the read/write operations of the “stopRequested” variable enabling the visibility to all the threads. It is important to note that **both** of the read and write operations need to be synchronized in order to achieve a happens-before relationship.

For situations similar to our example, using the synchronized keyword just to have visibility might not be the optimum solution. Synchronization has a performance impact due to the threads getting blocked while acquiring a lock. Hence, the synchronized keyword is more suitable when mutual exclusiveness (where only one thread is allowed to access a given resource at a time) is required.

For situations which require only visibility, Java has introduced a simple new keyword called “volatile”.

Volatile Fields

> *A write to a volatile field happens-before every subsequent read of that same field.*

We can use the “volatile” keyword to make “stopRequested” variable a volatile field creating a happens-before relationship with the write and read operations for “stopRequested”.

```java
public class StopThread {
        private static volatile boolean stopRequested;
        public static void main(String[] args)
                throws InterruptedException {
            Thread backgroundThread = new Thread(new Runnable() {
                public void run() {
                    int i = 0;
                    while (!stopRequested)
                        i++;
                }
            });
            backgroundThread.start();
            TimeUnit.SECONDS.sleep(1);
            stopRequested = true;
        }
    }
```

memsharesample(vol).java hosted with ❤ by GitHub view raw

However, it is important to keep note that the volatile keyword is not a replacement for synchronized blocks/methods. It will be useful only when achieving visibility for shared variables using a happens-before relationship. We will still have to use the synchronization when we need to achieve mutual exclusiveness between threads.

Consider the below example of a serial number generator

```
private static int nextSerialNumber = 0;

public static int generateSerialNumber() {
    return nextSerialNumber++;
}
```

The above code is not thread safe due to lack of atomicity (not happening all read-modify-write operations at once) in the increment operator (++). Various threads can end up in different states (read or write) when executing the following line.

```
return nextSerialNumber++;
```

However, making the "*nextSerialNumber*" volatile will not achieve mutual exclusiveness during the increment operation (as volatile keyword can only be used to achieve visibility). A proper fix would be to make the generateSerialNumber() method synchronized.

Apart from the synchronization and volatility, Java defines several sets of rules for a happens-before relationship. You can find them in detail from the Oracle Docs.

References

[1] Effective java (2nd edition) by joshua bloch

[2] https://docs.oracle.com/javase/specs/jls/se8/html/jls-17.html#jls-17.4.5

[3] https://wiki.sei.cmu.edu/confluence/display/java/Concurrency%2C+Visibility%2C+and+Memory