Memory Model

Introduction

The laws of Physics put a cap on how fast processors can be. Increasing the clock rates for processors has become harder and harder over the years. Consequently, parallelism, concurrency, and code optimizations are the other avenues that have been explored to make programs run more efficiently and faster. In this context optimizations by the runtime or the compiler can lead to concurrency bugs if the developer fails to use synchronization constructs to signal the platform that access to data is shared. The memory model in conjunction with the synchronization constructs play a vital role in establishing what optimizations are legal.

Formally, *a memory model describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program.*

In layman's terms, you can think of a memory model as the a set of rules which determine when writes by one thread are visible to another thread. Consequently the model permits the compiler, the processor or the runtime to reorder memory operations or program statements for optimization and performance reasons, as long as, write visibilities guaranteed by the model aren't violated. However, this freedom can wreak havoc in a multithreaded program when the memory model is not well-understood with unexpected program outcomes. Understanding the memory model allows one to reason and draw conclusions about a program's behavior and enumerate all of its possible outcomes.

Consider the below code snippet executed in our **main** thread. Assume the application also spawns a couple of other threads, that'll execute the method `runMethodForOtherThreads()`

```
1.  public class BadExample {
2.
3.      int myVariable = 0;
4.      boolean neverQuit = true;
5.
6.      public void runMethodForMainThread() {
7.
8.          // Change the variable value to Lucky 7
9.          myVariable = 7;
10.     }
11.
12.     public void runMethodForOtherThreads() {
13.
14.         while (neverQuit) {
15.             System.out.println("myVariable : " + myVariable);
16.         }
17.     }
18. }
```

Now you would expect that the other threads would see the `myVariable` value change to **7** as soon as the **main** thread executes the assignment on **line 9**. This assumption is false in modern architectures. Other threads may see the change in the value of the variable `myVariable` immediately, with a delay or never at all. Below are some of the reasons that can cause this to happen

- Use of sophisticated multi-level memory caches or processor caches that can get out of sync with the main memory, i.e. lack of *cache coherency*
- Reordering of statements by the compiler which may differ from the source code ordering
- Other optimizations that the hardware, runtime or the compiler may apply.

Cache Coherence

Wikipedia defines cache coherence in a multiprocess system with local caches as the *uniformity of shared resource data that ends up being stored in local caches*. Consider two processors in a system that share the main memory but also maintain individual local caches. Both processors cache a particular memory block from the main memory. One of the processors can make a change to its cached copy of the memory block, and the change may not be immediately flushed to the main memory. This causes two different views of the same main memory block in the two local caches. Even if the changes are immediately propagated to the main memory, the other processor may not choose to refresh its cache any time soon. The goal of cache coherence is to mitigate such situations and keep a consistent view of shared data across all memory sub-systems.

Going back to our above example program. One likely scenario can be that the variable is updated with the new value in the processor's cache but not in the main memory. When another thread running on another core requests the variable `myVariable`'s value from the memory, it still sees the stale value of **0**. This is a manifestation of violating ***cache coherence***. Different processor architectures have different policies as to when an individual processor's cache is reconciled with the main memory.

JSR-133

Java's memory model (JMM) was developed for the first time in 1995 and was considered broken by most. It was revised and updated in 2004 via Java Specification Request 133 (JSR-133). We won't delve too deep into the technicals details of JSR-133 but enough to avoid surprises/bugs when writing concurrent code in Java.

In summary, the memory model describes possible behaviors/outcomes of a program. The memory model determines the conditions under which a thread observes the latest value of a shared variable which is being written to by multiple threads. An implementation is free to produce any code it likes, as long as all the resulting executions of a program produce a result that conforms to the rules of the memory model. This extends a great deal of flexibility to the platform to undertake a myriad of code transformations, including the reordering of actions and removal of unnecessary synchronization.

within-thread as-if-serial

The Java language specification (JLS) mandates the Java Virtual Machine (JVM) to maintain ***within-thread as-if-serial*** semantics. What this means is that, as long as the result of the program is exactly the same as when the program statements were to be executed sequentially then the JVM is free to undertake any optimizations it may deem necessary.

Let's see what within-thread as-if-serial means in the context of a single-threaded program:

```
int x = 5;
int y = 10;

y*=y;
x*=x;

System.out.println(x);
System.out.println(y);
```

The order in which the six statements in the above program occur is called the *program order*. The program simply computes the square of the two variables `x` and `y` and prints them. A JVM implementation can choose to re-order the above statements as follows and still conform to *within-thread as-if-serial* semantics:

```
int x = 5;
x*=x;

int y = 10;
y*=y;

System.out.println(x);
System.out.println(y);
```

The above re-ordering is a valid one. Note that the program output prints the variable `x` first and then the variable `y` and as long as that print order is maintained, the JVM is free to move the other statements around or introduce other optimizations. When a single thread executes these optimized re-ordered statements in isolation the result is the same as if the thread executed the statements serially in program order - hence the name *within-thread as-if-serial*.

So far so good, but the moment these variables `x` and `y` become shared with other threads, the re-orderings and optimizations create different possibilities of when and what values of these variables are observed by other threads. We'll see in the next lesson how these optimizations can give surprising results in multithreaded scenarios.

← Back lesson
Miscellaneous Topics

Completed Next →
Reordering Effects