Why there was a need for a new Java Memory Model specification

Taivo Käsper University of Tartu
Supervised by Oleg Batrashev

Abstract

Java Memory Model (JMM) was first time that a programming language specification brought in a consistent semantics for concurrency across a variety of architectures. Nevertheless defining highly technical specification which would be both consistent and intuitive to programmers did not prove to be that easy. To change the specification a Java Specification Request 133 (JSR-133) was submitted and it took effect in 2004 with the release of Java version 1.5 on 30. September [1]. Till today there has been two versions of the memory model.

I. Introduction

Java Memory Model (JMM) describes how data like variables, fields, static fields, data structures are stored and retrieved to the physical memory of the computer. At one point most of the data will reach main memory but the compiler, runtime, processor or some level of cache will make this process more complex because a write to a variable does not mean that the new value will be immediately flushed to the physical memory of the machine.

These small optimizations are done to receive greater performance and they happen automatically without any aid from the programmer. The previous gets impossibly complex to do seamlessly when dealing with a multiprocessor system - that is where the memory model comes to focus. JMM is what describes what alternations may be done to the source code when compiling, producing native code from bytecode and optimizations that the hardware may perform on the native code.

Java was designed to run on multiplatform but that task proved to be harder than the designers first thought. The specification was not clear enough on some aspects and that lead to people developing different understanding of the JMM which led to several variations in the different implementations of the underlying compilers, too much confusion on the synchronization and thread safety and finally developers could not be sure that the written code is consistent. This regards the sections from Java specification [3] about volatile fields and final fields.

II. Out of order execution

Processor might not execute the commands in the same order as they appear in the source code. Reordering may be done by the compiler, runtime or the processor itself. It is a way to optimize the execution.
Let's consider the source code example from Listing 1

```
1 int a = 1;
2 int b = 2
3 int c = 3;
4 int sum = a + b;
```

Listing 1: Example code fragment

The code fragment shown in Listing 1 can be executed for example in the order shown in Listing 2

```
int a = 1;
int b = 2
int sum = a + b;
int c = 3;
```

Listing 2: Example reordering

Alternation in Listing 2 can be useful if processor has a and b in it's registers but there is not enough room for both sum and c but only one, so it reorders sum and c as shown in Listing 2 then calculates the sum and flushes the register to main memory. After that reads in variable c. Without reordering it might happen that processor reads a, b, c to register and then flushes it to the main memory then reads a and b, calculates the sum and then flushes sum to main memory. The main goal for reordering is to achieve the same results with less steps by the processor resulting in a faster program execution.

II.1 Happens-before relationship

Let A and B represent operations performed by a multithreaded process. If A happens-before B, then the memory effects of A effectively become visible to the thread performing B before B is performed [2]. This definition is illustrated with a code example in Listing 3.

```
int A;
int B;

void foo() {
    // This store to A in thread 1 ...
    A = 5;

    // ... effectively becomes visible
        before the following loads in
        thread 2.
    B = A * A;
}
```

Listing 3: Happens before example code fragment [2]

In example Listing 3 a Data Race is said to happen when two accesses from different threads conflict and they are not ordered by happens before which results in an inconsistent output.

III. Final fields

The new specification says about the final fields that they are initialized once and after that all threads see the initialized value and compilers are allowed to keep the value cached in a register and not reload it from memory in situations where a non-final field would have to be reloaded [3]. If there is a try to change the value of a final field using reflection then programmer has to make sure that the object is not visible to any thread before all the final modifications are done. That is necessary because the specification allows aggressive optimizations so final fields are not reloaded from main memory.

The properties of a final field lead programmers to a conclusion that the value of a final field will automatically be visible to all threads but the old JMM did not treat final fields differently than non-final ones [5]. This behavior created a lot of concurrency bugs because it was counterintuitive.

Lets take a look at the example code for final fields in Listing 4.

```
class FinalFieldExample {
  final int x;
  int y;

  static FinalFieldExample f;

  public FinalFieldExample() {
```

```
8       x = 3;
9       y = 4;
10    }
11
12    /**
13     * Called from thread 1
14     */
15    static void writer() {
16      f = new FinalFieldExample();
17    }
18
19    /**
20     * Called from thread 2
21     */
22    static void reader() {
23      if (f != null) {
24        int i = f.x;
25        int j = f.y;
26      }
27    }
28  }
```

Listing 4: Final fields example [3]

Code example Listing 4 is guaranteed to see value 3 on line 24 when new memory model is in use but with old memory model there might be the default value of integer (0).

IV. Volatile fields

Volatile is used to indicate that a variable's value will be modified by different threads and therefor all reads and writes will go straight to main memory and access to the variable acts as though it was enclosed in a synchronized block [6]. The main differences between volatile and synchronized are that a primitive variable may be declared volatile but you can not synchronize on a primitive, an access never blocks and the variable may hold null if declared as an object [6].

Sequential consistency is a very strong guarantee that is made about visibility and ordering in an execution of a program. Within a sequentially consistent execution, there is a total order over all individual actions (such as reads and writes) which is consistent with the order of the program, and each individual action is atomic and is immediately visible to every thread [4]. This means that whenever I write to a volatile field everything that is visible (regular fields) to any other thread is also copied to the main memory.

Regarding volatile fields with the new memory model let's take a look at the example from Listing 5.

```
1   class VolatileExample {
2     int count = 0;
3     volatile boolean bool = false;
4
5     /**
6      * Called from thread 1
7      */
8     public void write() {
9       count = 5;
10      bool = true;
11    }
12
13    /**
14     * Called from thread 2
15     */
16    public void read() {
17      if (bool == true) {
18        int countVal = count;
19      }
20    }
21  }
```

Listing 5: Volatile field example

The example code Listing 5 sees value 5 in row 18 when under new memory model but under the old one it might see 0 (integer default value) or 5. This can happen if rows 9 and 10 are reordered. Therefor the write to a volatile field happens before variable count gets it's value 5. Value 0 is put to main memory and value 5 stays in cache of thread 1.

Under the old JMM, the reads and writes to volatile variables were allowed to be reordered with respect to the read and write operations on normal variables which causes such kind

of inconsistencies. Under new JMM that optimization is not allowed [4].

V. Demo

For the purposes of demo a repository [7] was created to show the different results under the old JMM and new JMM.

For the old JMM a virtual environment is automatically created with Java 1.4.2 installed using Vagrant and VirtualBox and basic Bash provisioning. It is important to note that concurrency issues usually show at heavy load and certain circumstances and therefor the author saw the problem with volatile fields only once and the problem with final fields never.

For the new JMM a stress testing framework Jcstress [8] was used which requires Java version 1.8. The framework creates heavy load and runs the tests multiple times gathering all the results into and HTML files.

The author thinks that demos can definitely be improved to reveal the differences more easily and more often.

VI. Summary

Java memory model was the first try on any programming language to specify how multithreaded applications could keep their sequential consistency across processor architectures and operating systems. Main problems that were faced were that trying to specify something technical in an intuitive way so that nothing important gets lost is hard. Therefor using compiler implementations from different vendors ended up acting differently although the main idea of Java was to be cross platform. In 2004 the whole Java community decided to submit change requests and from these ideas JSR-133 was put together.

References

[1] Java 5.0 Press release http://web.archive.org/web/20080207083457/http://www.sun.com/smi/Press/sunflash/2004-09/sunflash.20040930.1.xml

[2] The happens-before relation http://preshing.com/20130702/the-happens-before-relation

[3] Java specification for the new memory model http://docs.oracle.com/javase/specs/jls/se5.0/html/memory.html#17.4

[4] Java specification for the new memory model regarding volatiles http://docs.oracle.com/javase/specs/jls/se5.0/html/memory.html#17.4.3

[5] Java specification for the old memory model http://www.oracle.com/technetwork/java/javasebusiness/downloads/java-archive-downloads-142docs-2045554.html

[6] The volatile keyword in Java http://www.javamex.com/tutorials/synchronization_volatile.shtml

[7] Demo BitBucket repository https://bitbucket.org/taivokasper/distributed-systems-seminas-jmm

[8] Jcstress testing framework http://openjdk.java.net/projects/code-tools/jcstress