Java Memory Model – Quick overview and things to notice

javacodegeeks.com Manoj Khangaonkar February 3rd, 2011 view original

In computing, a Memory model describes how threads interact through memory, or more generally specify what assumptions the compiler is allowed to make when generating code for segmented memory or paged memory platforms. It essentially describes, given a program and an execution trace of that program, whether the execution trace is a legal execution of the program.
The Java memory model describes how threads in the Java programming language interact through memory. Together with the description of single-threaded execution of code, the memory model provides the semantics of the Java programming language. The original Java memory model, developed in 1995, was widely perceived as broken, preventing many runtime optimizations and not providing strong enough guarantees for code safety. It was updated through the Java Community Process, as Java Specification Request 133 (JSR-133), which took effect in 2004, for Tiger (Java 5.0).

You can find some very useful information in the "Threads and Locks" chapter of the Java Language Specification and on this Java Memory model discussion page.

Let us now see some insights on the topic provided by our latest JCG partner, Manoj from the "The Khangaonkar Report".

(NOTE: The original post has been slightly edited to improve readability)

The Java memory model describes the rules that define how variables written to memory are seen, when such variables are written and read by multiple threads.

When a thread reads a variable, it is not necessarily getting the latest value from memory. The processor might return a cached value. Additionally, even though the programmer authored code where a variable is

first written and later read, the compiler might reorder the statements as long as it does not change the program semantics. It is quite common for processors and compilers to do this for performance optimization. As a result, a thread might not see the values it expects to see. This can result in hard to fix bugs in concurrent programs.

The Java programming language provides the “synchronized”, “volatile” and “final” keywords to help write safe multithreaded code. However earlier versions of Java had several issues because the memory model was underspecified. JSR 133 (Java Memory Model and Thread Specification Revision) fixed some of the flaws in the earlier memory model.

Most programmers are familiar with the fact that entering a synchronized block means obtaining a lock on a monitor that ensures that no other thread can enter the synchronized block. Less familiar but equally important are the facts that

(1) Acquiring a lock and entering a synchronized block forces the thread to refresh data from memory.
(2) Upon exiting the synchronized block, data written is flushed to memory.

This ensures that values written by a thread in a synchronized block are visible to other threads in synchronized blocks.

Ever heard of “happens before” in the context of Java? JSR 133 introduced the term “happens before” and provided some guarantees about the ordering of actions within a program. These guarantees are:

(1) Every action in a thread happens before every other action that comes after it in the thread.
(2) An unlock on a monitor happens before a subsequent lock on the same monitor
(3) A volatile write on a variable happens before a subsequent volatile read on the same variable
(4) A call to Thread.start() happens before any other statement in that thread
(5) All actions in thread happen before any other thread returns from a join() on that thread

The term “action” is defined in section 17.4.2 of the Java language specification as statements that can be detected or influenced by other threads. Normal read/write, volatile read/write, lock/unlock are some actions.

Rules 1 ,4 and 5 guarantee that within a single thread, all actions will execute in the order in which they

appear in the authored program. Rules 2 and 4 guarantee that between multiple threads working on shared data, the relative ordering of synchronized blocks and the order of read/writes on volatile variables is preserved.

Rules 2 and 4 makes volatile very similar to a synchronized block. Prior to JSR 133, volatile still meant that a write to volatile variable is written directly to memory and a read is read from memory. But a compiler could reorder volatile read/writes with non volatile read/writes causing incorrect results. Not possible after JSR 133.

One additional notable point. This is related to final members that are initialized in the constructor of a class. As long as the constructor completes execution properly, the final members are visible to other threads without synchronization. If you however share the reference to the object from within the constructor, then all bets are off.

The proposed specification describes the semantics of threads, locks, volatile variables and data races. This includes what has been referred to as the Java memory model.