Concurrent Programming in Java
© 1996-1999 Doug Lea

Synchronization and the Java Memory Model

This set of excerpts from section 2.2 includes the main discussions on how the Java Memory Model impacts concurrent programming.

For information about ongoing work on the memory model, see Bill Pugh's Java Memory Model pages.

Consider the tiny class, defined without any synchronization:

```
final class SetCheck {
  private int  a = 0;
  private long b = 0;

  void set() {
    a =  1;
    b = -1;
  }

  boolean check() {
    return ((b ==  0) ||
            (b == -1 && a == 1));
  }
}
```

In a purely sequential language, the method check could never return false. This holds even though compilers, run-time systems, and hardware might process this code in a way that you might not intuitively expect. For example, any of the following might apply to the execution of method set:

- The compiler may rearrange the order of the statements, so b may be assigned before a. If the method is inlined, the compiler may further rearrange the orders with respect to yet other statements.
- The processor may rearrange the execution order of machine instructions corresponding to the statements, or even execute them at the same time.
- The memory system (as governed by cache control units) may rearrange the order in which writes are committed to memory cells corresponding to the variables. These writes may overlap with other computations and memory actions.
- The compiler, processor, and/or memory system may interleave the machine-level effects of the two statements. For example on a 32-bit machine, the high-order word of b may be written first, followed by the write to a, followed by the write to the low-order word of b.

- The compiler, processor, and/or memory system may cause the memory cells representing the variables not to be updated until sometime after (if ever) a subsequent check is called, but instead to maintain the corresponding values (for example in CPU registers) in such a way that the code still has the intended effect.

In a sequential language, none of this can matter so long as program execution obeys as-if-serial semantics. Sequential programs cannot depend on the internal processing details of statements within simple code blocks, so they are free to be manipulated in all these ways. This provides essential flexibility for compilers and machines. Exploitation of such opportunities (via pipelined superscalar CPUs, multilevel caches, load/store balancing, interprocedural register allocation, and so on) is responsible for a significant amount of the massive improvements in execution speed seen in computing over the past decade. The as-if-serial property of these manipulations shields sequential programmers from needing to know if or how they take place. Programmers who never create their own threads are almost never impacted by these issues.

Things are different in concurrent programming. Here, it is entirely possible for check to be called in one thread while set is being executed in another, in which case the check might be "spying" on the optimized execution of set. And if any of the above manipulations occur, it is possible for check to return false. For example, as detailed below, check could read a value for the long b that is neither 0 nor -1, but instead a half-written in-between value. Also, out-of-order execution of the statements in set may cause check to read b as -1 but then read a as still 0.

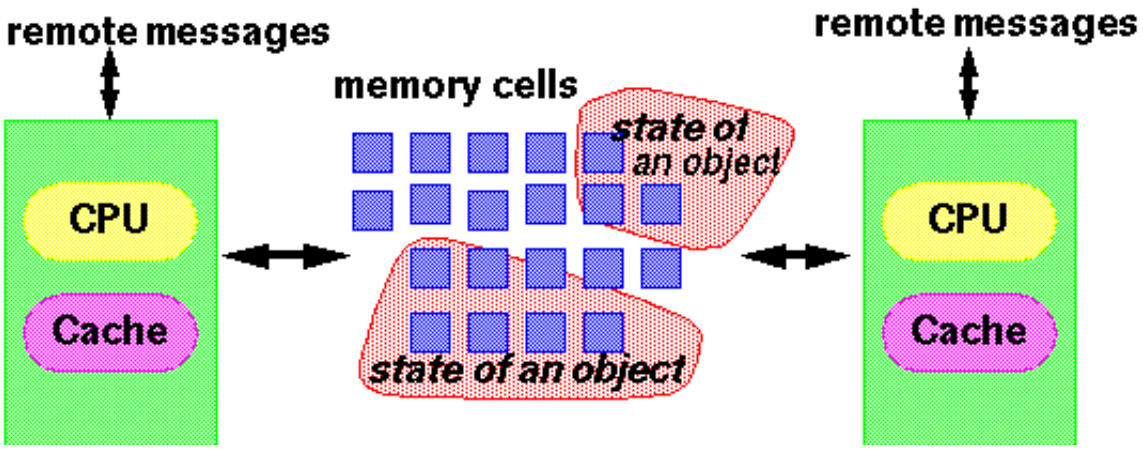
In other words, not only may concurrent executions be interleaved, but they may also be reordered and otherwise manipulated in an optimized form that bears little resemblance to their source code. As compiler and run-time technology matures and multiprocessors become more prevalent, such phenomena become more common. They can lead to surprising results for programmers with backgrounds in sequential programming (in other words, just about all programmers) who have never been exposed to the underlying execution properties of allegedly sequential code. This can be the source of subtle concurrent programming errors.

In almost all cases, there is an obvious, simple way to avoid contemplation of all the complexities arising in concurrent programs due to optimized execution mechanics: Use synchronization. For example, if both methods in class SetCheck are declared as synchronized, then you can be sure that no internal processing details can affect the intended outcome of this code.

But sometimes you cannot or do not want to use synchronization. Or perhaps you must reason about someone else's code that does not use it. In these cases you must rely on the minimal guarantees about resulting semantics spelled out by the Java Memory Model. This model allows the kinds of manipulations listed above, but bounds their potential effects on execution semantics and additionally points to some techniques programmers can use to control some aspects of these semantics (most of which are discussed in §2.4).

The Java Memory Model is part of The JavaTM Language Specification, described primarily in JLS chapter 17. Here, we discuss only the basic motivation, properties, and programming consequences of the model. The treatment here reflects a few clarifications and updates that are missing from the first edition of JLS.

The assumptions underlying the model can be viewed as an idealization of a standard SMP machine of the sort described in §1.2.4:



For purposes of the model, every thread can be thought of as running on a different CPU from any other thread. Even on multiprocessors, this is infrequent in practice, but the fact that this CPU-per-thread mapping is among the legal ways to implement threads accounts for some of the model's initially surprising properties. For example, because CPUs hold registers that cannot be directly accessed by other CPUs, the model must allow for cases in which one thread does not know about values being manipulated by another thread. However, the impact of the model is by no means restricted to multiprocessors. The actions of compilers and processors can lead to identical concerns even on single-CPU systems.

The model does not specifically address whether the kinds of execution tactics discussed above are performed by compilers, CPUs, cache controllers, or any other mechanism. It does not even discuss them in terms of classes, objects, and methods familiar to programmers. Instead, the model defines an abstract relation between threads and main memory. Every thread is defined to have a working memory (an abstraction of caches and registers) in which to store values. The model guarantees a few properties surrounding the interactions of instruction sequences corresponding to methods and memory cells corresponding to fields. Most rules are phrased in terms of when values must be transferred between the main memory and per-thread working memory. The rules address three intertwined issues:

Atomicity
: Which instructions must have indivisible effects. For purposes of the model, these rules need to be stated only for simple reads and writes of memory cells representing fields - instance and static variables, also including array elements, but not including local variables inside methods.

Visibility
: Under what conditions the effects of one thread are visible to another. The effects of interest here are writes to fields, as seen via reads of those fields.

Ordering
: Under what conditions the effects of operations can appear out of order to any given thread. The main ordering issues surround reads and writes associated with sequences of assignment statements.

When synchronization is used consistently, each of these properties has a simple characterization: All changes made in one synchronized method or block are atomic and visible with respect to other synchronized methods and blocks employing the same lock, and processing of synchronized methods or blocks within any given thread is in program-specified order. Even though processing of statements within blocks may be out of order, this cannot matter to other threads employing synchronization.

When synchronization is not used or is used inconsistently, answers become more complex. The guarantees

made by the memory model are weaker than most programmers intuitively expect, and are also weaker than those typically provided on any given JVM implementation. This imposes additional obligations on programmers attempting to ensure the object consistency relations that lie at the heart of exclusion practices: Objects must maintain invariants as seen by all threads that rely on them, not just by the thread performing any given state modification.

The most important rules and properties specified by the model are discussed below.

Atomicity

Accesses and updates to the memory cells corresponding to fields of any type except long or double are guaranteed to be atomic. This includes fields serving as references to other objects. Additionally, atomicity extends to volatile long and double. (Even though non-volatile longs and doubles are not guaranteed atomic, they are of course allowed to be.)

Atomicity guarantees ensure that when a non-long/double field is used in an expression, you will obtain either its initial value or some value that was written by some thread, but not some jumble of bits resulting from two or more threads both trying to write values at the same time. However, as seen below, atomicity alone does not guarantee that you will get the value most recently written by any thread. For this reason, atomicity guarantees per se normally have little impact on concurrent program design.

Visibility

Changes to fields made by one thread are guaranteed to be visible to other threads only under the following conditions:

- A writing thread releases a synchronization lock and a reading thread subsequently acquires that same synchronization lock.

  In essence, releasing a lock forces a flush of all writes from working memory employed by the thread, and acquiring a lock forces a (re)load of the values of accessible fields. While lock actions provide exclusion only for the operations performed within a synchronized method or block, these memory effects are defined to cover all fields used by the thread performing the action.

  Note the double meaning of synchronized: it deals with locks that permit higher-level synchronization protocols, while at the same time dealing with the memory system (sometimes via low-level memory barrier machine instructions) to keep value representations in synch across threads. This reflects one way in which concurrent programming bears more similarity to distributed programming than to sequential programming. The latter sense of synchronized may be viewed as a mechanism by which a method running in one thread indicates that it is willing to send and/or receive changes to variables to and from methods running in other threads. From this point of view, using locks and passing messages might be seen merely as syntactic variants of each other.

- If a field is declared as volatile, any value written to it is flushed and made visible by the writer thread before the writer thread performs any further memory operation (i.e., for the purposes at hand it is flushed immediately). Reader threads must reload the values of volatile fields upon each access.

- The first time a thread accesses a field of an object, it sees either the initial value of the field or a value since written by some other thread.

  Among other consequences, it is bad practice to make available the reference to an incompletely constructed object (see §2.1.2). It can also be risky to start new threads inside a constructor, especially in a class that may be subclassed. Thread.start has the same memory effects as a lock release by the thread calling start, followed by a lock acquire by the started thread. If a Runnable superclass invokes new Thread(this).start() before subclass constructors execute, then the object might not be fully initialized when the run method executes. Similarly, if you create and start a new thread T and then create an object X used by thread T, you cannot be sure that the fields of X will be visible to T unless you employ synchronization surrounding all references to object X. Or, when applicable, you can create X before starting T.

- As a thread terminates, all written variables are flushed to main memory. For example, if one thread synchronizes on the termination of another thread using Thread.join, then it is guaranteed to see the effects made by that thread (see §4.3.2).

Note that visibility problems never arise when passing references to objects across methods in the same thread.

The memory model guarantees that, given the eventual occurrence of the above operations, a particular update to a particular field made by one thread will eventually be visible to another. But eventually can be an arbitrarily long time. Long stretches of code in threads that use no synchronization can be hopelessly out of synch with other threads with respect to values of fields. In particular, it is always wrong to write loops waiting for values written by other threads unless the fields are volatile or accessed via synchronization (see §3.2.6).

The model also allows inconsistent visibility in the absence of synchronization. For example, it is possible to obtain a fresh value for one field of an object, but a stale value for another. Similarly, it is possible to read a fresh, updated value of a reference variable, but a stale value of one of the fields of the object now being referenced.

However, the rules do not require visibility failures across threads, they merely allow these failures to occur. This is one aspect of the fact that not using synchronization in multithreaded code doesn't guarantee safety violations, it just allows them. On most current JVM implementations and platforms, even those employing multiple processors, detectable visibility failures rarely occur. The use of common caches across threads sharing a CPU, the lack of aggressive compiler-based optimizations, and the presence of strong cache consistency hardware often cause values to act as if they propagate immediately among threads. This makes testing for freedom from visibility-based errors impractical, since such errors might occur extremely rarely, or only on platforms you do not have access to, or only on those that have not even been built yet. These same comments apply to multithreaded safety failures more generally. Concurrent programs that do not use synchronization fail for many reasons, including memory consistency problems.

Ordering

Ordering rules fall under two cases, within-thread and between-thread:

- From the point of view of the thread performing the actions in a method, instructions proceed in the normal as-if-serial manner that applies in sequential programming languages.

- From the point of view of other threads that might be "spying" on this thread by concurrently running unsynchronized methods, almost anything can happen. The only useful constraint is that the relative orderings of synchronized methods and blocks, as well as operations on volatile fields, are always preserved.

Again, these are only the minimal guaranteed properties. In any given program or platform, you may find stricter orderings. But you cannot rely on them, and you may find it difficult to test for code that would fail on JVM implementations that have different properties but still conform to the rules.

Note that the within-thread point of view is implicitly adopted in all other discussions of semantics in JLS. For example, arithmetic expression evaluation is performed in left-to-right order (JLS section 15.6) as viewed by the thread performing the operations, but not necessarily as viewed by other threads.

The within-thread as-if-serial property is helpful only when only one thread at a time is manipulating variables, due to synchronization, structural exclusion, or pure chance. When multiple threads are all running unsynchronized code that reads and writes common fields, then arbitrary interleavings, atomicity failures, race conditions, and visibility failures may result in execution patterns that make the notion of as-if-serial just about meaningless with respect to any given thread.

Even though JLS addresses some particular legal and illegal reorderings that can occur, interactions with these other issues reduce practical guarantees to saying that the results may reflect just about any possible interleaving of just about any possible reordering. So there is no point in trying to reason about the ordering properties of such code.

Volatile

In terms of atomicity, visibility, and ordering, declaring a field as volatile is nearly identical in effect to using a little fully synchronized class protecting only that field via get/set methods, as in:

```
final class VFloat {
  private float value;

  final synchronized void  set(float f) { value = f; }
  final synchronized float get()        { return value; }
}
```

Declaring a field as volatile differs only in that no locking is involved. In particular, composite read/write operations such as the "++" operation on volatile variables are not performed atomically.

Also, ordering and visibility effects surround only the single access or update to the volatile field itself. Declaring a reference field as volatile does not ensure visibility of non-volatile fields that are accessed via this reference. Similarly, declaring an array field as volatile does not ensure visibility of its elements. Volatility cannot be manually propagated for arrays because array elements themselves cannot be declared as volatile.

Because no locking is involved, declaring fields as volatile is likely to be cheaper than using synchronization, or at least no more expensive. However, if volatile fields are accessed frequently inside methods, their use is likely to lead to slower performance than would locking the entire methods.

Declaring fields as volatile can be useful when you do not need locking for any other reason, yet values must be accurately accessible across multiple threads. This may occur when:

- The field need not obey any invariants with respect to others.
- Writes to the field do not depend on its current value.
- No thread ever writes an illegal value with respect to intended semantics.
- The actions of readers do not depend on values of other non-volatile fields.

Using volatile fields can make sense when it is somehow known that only one thread can change a field, but many other threads are allowed to read it at any time. For example, a Thermometer class might declare its temperature field as volatile. As discussed in §3.4.2, a volatile can be useful as a completion flag. Additional examples are illustrated in §4.4, where the use of lightweight executable frameworks automates some aspects of synchronization, but volatile declarations are needed to ensure that result field values are visible across tasks.

Doug Lea
Last modified: Sat Jul 29 13:21:07 EDT 2000