Threading lightly, Part 1: **Synchronization is not the enemy**

When do we have to synchronize, and how expensive is it really?

Brian Goetz

July 17, 2001

Unlike many other programming languages, the Java Language Specification included explicit support for threading and concurrency. While having language support for concurrency makes it easier to specify and manage constraints on shared data and the timing of operations across threads, it doesn't make the complexities of concurrent programming any easier to understand. This three-part series aims to help programmers understand some of the major issues behind multithreaded programming in the Java language, and in particular to understand the impact of thread safety on Java program performance.

For most programming languages, the language specification is silent on the topic of threading and concurrency; these topics have historically been left for the platform or operating system to specify. In contrast, the Java Language Specification (JLS) explicitly includes a threading model and provides several language elements for developers to use for rendering their programs thread-safe.

Explicit support for threading can be both a blessing and a curse. While it makes it easier for us to write programs that take advantage of the power and convenience of threading, it also means that we have to pay attention to the thread-safety of the classes we write, because any given class is much more likely to be used in a multithreaded environment.

Many users first find themselves having to understand threading not because they are writing programs that create and manage threads, but because they are using a tool or framework that is itself multithreaded. Any developer who has used the Swing GUI framework, or has written a servlet or JSP page, has been exposed (knowingly or not) to the complexities of threading.

The Java architects wanted to create a language that would perform well on modern hardware, including multiprocessor systems. To achieve this goal, the job of managing coordination between threads was largely pushed back to the developer; programmers must specify where data will be shared between threads. The primary tool for managing coordination between threads in Java programs is the `synchronized` keyword. In the absence of synchronization, the JVM is free to take
a great deal of liberty in the timing and ordering of operations executing in different threads. Most of the time this is desirable, as it results in higher performance, but it places an additional burden on the programmer to identify when such optimizations would compromise program correctness.

**What does synchronized really mean?**

Most Java programmers think of a synchronized block or method entirely in terms of enforcing a mutex (mutual exclusion semaphore) or defining a critical section (a block of code which must run atomically). While the semantics of `synchronized` do include mutual exclusion and atomicity, the reality of what happens prior to monitor entry and after monitor exit is considerably more complicated.

**Don't miss the rest of this series**

Part 2, "Reducing contention" (September 2001)
Part 3, "Sometimes it's best not to share" (October 2001)

The semantics of `synchronized` do guarantee that only one thread has access to the protected section at one time, but they also include rules about the synchronizing thread's interaction with main memory. A good way to think about the Java Memory Model (JMM) is to assume that each thread is running on a separate processor, and while all processors access a common main memory space, each processor has its own cache that may not always be synchronized with main memory. In the absence of synchronization, it is allowable (according to the JMM) for two threads to see different values in the same memory location. When synchronizing on a monitor (lock), the JMM requires that this cache be invalidated immediately after the lock is acquired, and flushed (writing any modified memory locations back to main memory) before it is released. It's not hard to see why synchronization can have a significant effect on program performance; flushing the cache frequently can be expensive.

**Walking a fine line**

The consequences of failing to properly synchronize are severe: data corruption and race conditions, which can cause programs to crash, produce incorrect results, or behave unpredictably. Even worse, these conditions are likely to occur only rarely and sporadically (making the problem hard to detect and reproduce.) If the test environment differs substantially from the production environment, either in configuration or in load, these problems may not occur at all in the test environment, leading to the erroneous conclusion that our programs are correct, when in fact they simply have not failed yet.

**Race conditions defined**

A race condition is a situation in which two or more threads or processes are reading or writing some shared data, and the final result depends on the timing of how the threads are scheduled. Race conditions can lead to unpredictable results and subtle program bugs.

On the other hand, using synchronization inappropriately or excessively can lead to other problems, such as poor performance and deadlock. While poor performance is certainly a less severe problem than data corruption, it can still be a serious problem. Writing good multithreaded
programs requires walking a fine line, synchronizing enough to protect your data from corruption, but not so much as to risk deadlock or impair program performance unnecessarily.

**How expensive is synchronization?**

Because of the rules involving cache flushing and invalidation, a synchronized block in the Java language is generally more expensive than the critical section facilities offered by many platforms, which are usually implemented with an atomic "test and set bit" machine instruction. Even when a program contains only a single thread running on a single processor, a synchronized method call is still slower than an unsynchronized method call. If the synchronization actually requires contending for the lock, the performance penalty is substantially greater, as there will be several thread switches and system calls required.

Fortunately, continuous improvements in the JVM have both improved overall Java program performance and reduced the relative cost of synchronization with each release, and future improvements are anticipated. Further, the performance costs of synchronization are often overstated. One well-known source has cited that a synchronized method call is as much as 50 times slower than an unsynchronized method call. While this statement may be true, it is also quite misleading and has led many developers to avoid synchronizing even in cases where it is needed.

It makes little sense to cast the performance penalty of synchronization strictly in percentage terms, because an uncontended synchronization imposes a fixed performance penalty on a block or method. The percentage performance penalty implied by this fixed delay depends on how much work is being done in the synchronized block. A synchronized call to an empty method may be 20 times slower than an unsynchronized call to an empty method, but how often do we call empty methods? When we measure the synchronization penalty for more representative small methods, the percentage numbers fall quite rapidly to something much more tolerable.

Table 1 puts some of these numbers in perspective. It compares the cost of a synchronized method call to an equivalent unsynchronized one in several different cases and on several different platforms and JVMs. In each case, I ran a simple program that measured the run time of a loop calling a method 10,000,000 times, calling both a synchronized and an unsynchronized version, and compared the results. The data in the table is the ratio of run time of the synchronized version to the unsynchronized version; it shows the performance penalty of synchronization. In each run, it calls one of the simple methods shown in Listing 1.

Table 1 shows only the relative performance of a synchronized method call versus an unsynchronized one; in order to gauge the performance penalty in absolute terms, you must also factor in the speed improvements of the JVM, which is not shown in this data. In most of the tests, the overall JVM performance improved substantially with each JVM version, and it is quite likely that the performance of the 1.4 Java virtual machine will be yet a further improvement when it is released.

**Table 1. Performance penalty of uncontended synchronization**
Listing 1. Example methods

```java
public static void staticEmpty() { }

public void empty() { }

public Object fetch() { return field; }

public Object singleton() {
    if (singletonField == null)
        singletonField = new Object();
    return singletonField;
}

public Object hashmapGet() {
    return hashMap.get("this");
}

public Object create() {
    return new Object();
}
```

These small benchmarks also illustrate the challenge of interpreting performance results in the presence of dynamic compilers. The dramatic differences in the numbers for the 1.3 JDK with and without the JIT require some explanation. For the very simple methods (empty and fetch), the nature of the benchmark test (it does nothing but execute a tight loop that does almost no work) enables the JIT to dynamically compile the entire loop, squeezing the run time to almost nothing. Whether the JIT would be able to do so in a real-world program depends on a lot of factors, and so the non-JIT timing numbers are probably more useful for making a fair comparison. In any case, for the more substantial methods (create and hashmapGet), the JIT was unable to make the huge
improvement to the unsynchronized case as it was with the simpler methods. Also, there is no telling whether the JVM was able to optimize away significant portions of the test. Similarly, the differences between the comparable IBM and Sun JDKs reflect the fact that the IBM Java SDK more aggressively optimized the unsynchronized loops, not that the synchronized version was more expensive; this was evident in the raw timing numbers (not presented here.)

The conclusion we can draw from these numbers is that while there is still a performance penalty for uncontended synchronization, it falls to a "reasonable" level for many non-trivial methods; that penalty is somewhere between 10 percent and 200 percent (of a relatively small number) in most cases. As a result, while it is still inadvisable to synchronize every method (this also increases the likelihood of deadlock), we need not be so fearful of synchronization. The simple tests used here suggest that an uncontended synchronization is cheaper than the cost of an object creation or a HashMap lookup.

When early books and articles suggested that there was a huge cost to uncontended synchronization, many programmers went to great lengths to avoid synchronizing at all costs. This fear led to many problematic techniques, such as the double-checked locking idiom (DCL). DCL is widely recommended in a number of books and articles on Java programming, and it seems a quite clever way to avoid synchronizing unnecessarily, but in fact it does not work and should be avoided. The reasons it doesn't work are quite complicated and beyond the scope of this article (see Related topics for follow-up links).

Nolo contendre

Assuming that synchronization is used appropriately, the real performance impact of synchronization is felt when threads actually contend for a lock. The cost difference between an uncontended synchronization and a contended one is huge; a simple test program suggested that a contended synchronization is 50 times slower than an uncontended one. This fact combined with the observations drawn above suggests that a contended synchronization is comparable in cost to at least 50 object creations.

In tuning an application's use of synchronization, then, we should try hard to reduce the amount of actual contention, rather than simply try to avoid using synchronization at all. Part 2 of this series will focus on techniques for reducing contention, including reducing lock granularity, reducing the size of synchronized blocks, and reducing the amount of data that is shared across threads.

When do I need to synchronize?

To make your programs thread-safe, you must first identify what data will be shared across threads. If you are writing data that may be read later by another thread, or reading data that may have been written by another thread, then that data is shared, and you must synchronize when accessing it. Some programmers are surprised to learn that these rules also apply in situations where you are simply checking if a shared reference is non-null.

Many people find these definitions surprisingly stringent. It is a commonly held belief that you do not need to acquire a lock to simply read an object's fields, especially since the JLS
guarantees that 32-bit reads will be atomic. Unfortunately, this intuition is incorrect. Unless the fields in question are declared `volatile`, the JMM does not require the underlying platform to provide cache coherency or sequential consistency across processors, so it is possible, on some platforms, to read stale data in the absence of synchronization. See Related topics for more detail.

After you've identified what data is being shared, you must then also identify how you are going to protect that data. In simple cases, you can protect data fields by simply declaring them `volatile`; in other cases, you must acquire a lock before reading or writing the shared data, and it is a good practice to explicitly identify what lock is being used to protect a given field or object, and document that with your code.

It is also worth noting that simply synchronizing accessor methods (or declaring the underlying fields `volatile`) may not be sufficient to protect a shared field. Consider this example:

```java
private int foo;
public synchronized int getFoo() { return foo; }
public synchronized void setFoo(int f) { foo = f; }
```

If a caller wants to increment the `foo` property, the following code to do so is not thread-safe:

```java
setFoo(getFoo() + 1);
```

If two threads attempt to increment `foo` at the same time, the result might be that the value of `foo` gets increased by one or by two, depending on timing. Callers will need to synchronize on a lock to prevent this race condition; it is a good practice for your class JavaDoc to specify what lock to synchronize on, so that callers of your class don't have to guess.

The above situation is a good example of how we must pay attention to data integrity at multiple levels of granularity; synchronizing the accessor functions ensures that callers have access to a consistent and recent version of the property value, but if we want future values of the property to be consistent with current values, or multiple properties to be consistent with each other, we must also synchronize composite operations, possibly on a coarser-grained lock.

**If in doubt, consider a synchronized wrapper**

Sometimes, when writing a class, we don't know if it is going to be used in a shared context or not. We want our classes to be thread-safe, but we also don't want to burden a class that will always be used in a single-threaded environment with the overhead of synchronization, and we may not know what the appropriate locking granularity will be when the class is used. Fortunately, we can often have it both ways by providing a synchronized wrapper. The Collections classes are a good example of this technique; they are unsynchronized, but for each interface defined in the framework, there is a synchronized wrapper (for example, `Collections.synchronizedMap()`) that wraps each method with a synchronized version.
Conclusion

While the JLS gives us tools with which we can make our programs thread-safe, thread-safety does not come free. Using synchronization entails a performance penalty, and the failure to use it correctly exposes us to the risks of data corruption, inconsistent results, or deadlocks. Fortunately, JVMs have improved substantially in the past several years, reducing the performance penalties associated with using synchronization properly. By carefully analyzing how data will be shared across threads, and synchronizing operations on shared data appropriately, you can render your programs thread-safe without incurring excessive performance overhead.
Related topics

- *Java Performance and Scalability, Volume 1: Server-Side Programming Techniques* by Dov Bulka (Addison-Wesley, 2000) provides a wealth of tips and tricks to help you increase the performance of your apps.
- Brian Goetz' recent article "Double-checked locking: Clever, but broken" (*JavaWorld*, February 2001) explores the JMM in detail and describes the surprising consequences of failing to synchronize in certain situations.
- Recognized multithreading authority Allen Holub reveals why most of the tricks to reduce synchronization overhead don't work in his article "Warning: Threading in a multiprocessor world" (*JavaWorld*, February 2001).
- In his article "Writing multithreaded Java applications" (*developerWorks*, February 2001) Alex Roetter introduces the Java Thread API, outlines issues involved in multithreading, and offers solutions to common problems.
- "Synchronization and the Java Memory Model" is an excerpt from Doug Lea's book that focuses on the actual meaning of synchronized.
- Bill Pugh's *Java Memory Model page* provides a great starting point for your study of the JMM.
- The "Double Checked Locking is Broken" Declaration describes why the DCL won't work when implemented in the Java language.
- The performance modeling and analysis team at IBM Thomas J. Watson Research Center is researching several projects in the areas of performance and performance management.