Java theory and practice: More flexible, scalable locking in JDK 5.0

New lock classes improve on synchronized -- but don't count synchronized out just yet

Brian Goetz (brian@quiotix.com)
Principal Consultant
Quiotix

26 October 2004

JDK 5.0 offers developers some powerful new choices for developing high-performance concurrent applications. For example, the `ReentrantLock` class in `java.util.concurrent.lock` is offered as a replacement for the `synchronized` facility of the Java language -- it has the same memory semantics, the same locking semantics, better performance under contention, and features not offered by `synchronized`. Does this mean that we should forget about `synchronized`, and instead use `ReentrantLock` exclusively? Concurrency expert Brian Goetz returns from his summer hiatus to supply the answer.

View more content in this series

Multithreading and concurrency are nothing new, but one of the innovations of the Java language design was that it was the first mainstream programming language to incorporate a cross-platform threading model and formal memory model directly into the language specification. The core class libraries include a `Thread` class for creating, starting, and manipulating threads, and the language includes constructs for communicating concurrency constraints across threads -- `synchronized` and `volatile`. While this simplifies the development of platform-independent concurrent classes, it by no means makes writing concurrent classes trivial -- just easier.

A quick review of synchronized

Declaring a block of code to be synchronized has two important consequences, generally referred to as *atomicity* and *visibility*. Atomicity means that only one thread at a time can execute code protected by a given monitor object (lock), allowing you to prevent multiple threads from colliding with each other when updating shared state. Visibility is more subtle; it deals with the vagaries of memory caching and compiler optimizations. Ordinarily, threads are free to cache values for variables in such a way that they are not necessarily immediately visible to other threads (whether it be in registers, in processor-specific caches, or through instruction reordering or other compiler

optimizations), but if the developer has used synchronization, as shown in the code below, the runtime will ensure that updates to variables made by one thread prior to exiting a `synchronized` block will become immediately visible to another thread when it enters a `synchronized` block protected by that same monitor (lock). A similar rule exists for `volatile` variables. (See Resources for more information on synchronization and the Java Memory Model.)

```
synchronized (lockObject) {
  // update object state
}
```

So synchronization takes care of everything needed to reliably update multiple shared variables without race conditions or corrupting data (provided the synchronization boundaries are in the right place), and ensures that other threads that properly synchronize will see the most up-to-date values of those variables. By defining a clear, cross-platform memory model (which was modified in JDK 5.0 to fix some errors in the initial definition), it becomes possible to create "Write Once, Run Anywhere" concurrent classes by following this simple rule:

> Whenever you will be writing a variable that may next be read by another thread, or reading a variable that may have last been written by another thread, you must synchronize.

Even better, in recent JVMs, the performance cost of uncontended synchronization (when no thread attempts to acquire a lock when another thread already holds it) is quite modest. (This was not always true; synchronization in early JVMs had not yet been optimized, giving rise to the then-true, but now mythical belief that synchronization, whether contended or not, has a big performance cost.)

Improving on synchronized

So synchronization sounds pretty good, right? Then why did the JSR 166 group spend so much time developing the `java.util.concurrent.lock` framework? The answer is simple -- synchronization is good, but not perfect. It has some functional limitations -- it is not possible to interrupt a thread that is waiting to acquire a lock, nor is it possible to poll for a lock or attempt to acquire a lock without being willing to wait forever for it. Synchronization also requires that locks be released in the same stack frame in which they were acquired, which most of the time is the right thing (and interacts nicely with exception handling), but a small number of cases exist where non-block-structured locking can be a big win.

The ReentrantLock class

The `Lock` framework in `java.util.concurrent.lock` is an abstraction for locking, allowing for lock implementations that are implemented as Java classes rather than as a language feature. It makes room for multiple implementations of `Lock`, which may have different scheduling algorithms, performance characteristics, or locking semantics. The `ReentrantLock` class, which implements `Lock`, has the same concurrency and memory semantics as `synchronized`, but also adds features like lock polling, timed lock waits, and interruptible lock waits. Additionally, it offers far better performance under heavy contention. (In other words, when many threads are attempting to

access a shared resource, the JVM will spend less time scheduling threads and more time executing them.)

What do we mean by a *reentrant* lock? Simply that there is an acquisition count associated with the lock, and if a thread that holds the lock acquires it again, the acquisition count is incremented and the lock then needs to be released twice to truly release the lock. This parallels the semantics of `synchronized`; if a thread enters a synchronized block protected by a monitor that the thread already owns, the thread will be allowed to proceed, and the lock will not be released when the thread exits the second (or subsequent) `synchronized` block, but only will be released when it exits the first `synchronized` block it entered protected by that monitor.

In looking at the code example in Listing 1, one immediate difference between `Lock` and synchronization jumps out -- the lock must be released in a finally block. Otherwise, if the protected code threw an exception, the lock might never be released! This distinction may sound trivial, but, in fact, it is extremely important. Forgetting to release the lock in a finally block can create a time bomb in your program whose source you will have a hard time tracking down when it finally blows up on you. With synchronization, the JVM ensures that locks are automatically released.

Listing 1. Protecting a block of code with ReentrantLock.

```
Lock lock = new ReentrantLock();

lock.lock();
try {
  // update object state
}
finally {
  lock.unlock();
}
```

As a bonus, the implementation of `ReentrantLock` is far more scalable under contention than the current implementation of synchronized. (It is likely that there will be improvements to the contended performance of synchronized in a future version of the JVM.) This means that when many threads are all contending for the same lock, the total throughput is generally going to be better with `ReentrantLock` than with `synchronized`.

Comparing the scalability of ReentrantLock and synchronization

Tim Peierls has constructed a simple benchmark for measuring the relative scalability of `synchronized` versus `Lock`, using a simple linear congruence pseudorandom number generator (PRNG). This example is nice because the PRNG actually does some real work each time `nextRandom()` is called, so that this benchmark is actually measuring a reasonable, real-world application of `synchronized` and `Lock`, rather than timing contrived or do-nothing code (like many so-called benchmarks.)

In this benchmark, we have an interface for `PseudoRandom`, which has a single method, `nextRandom(int bound)`. This interface is very similar to the functionality of the `java.util.Random` class. Because PRNGs use the last number generated as an input when generating the next

random number, and the last number generated is maintained as an instance variable, it is important that the portion of the code that updates this state not be preempted by other threads, so we use some form of locking to ensure this. (The `java.util.Random` class does this, too.) We have constructed two implementations of `PseudoRandom`; one that uses synchronization, and one that uses `java.util.concurrent.ReentrantLock`. The driver program spawns a number of threads, each of which madly rolls the dice, then calculates how many rolls per second the different versions were able to execute. The results are summarized for different numbers of threads in Figure 1 and Figure 2. This benchmark is not perfect, and it was only run on two systems (a dual Xeon with hyperthreading running Linux, and a single-processor Windows system), but should be good enough to suggest that `ReentrantLock` has a scalability advantage over `synchronized`.

Figure 1. Throughput for synchronization and Lock, single CPU

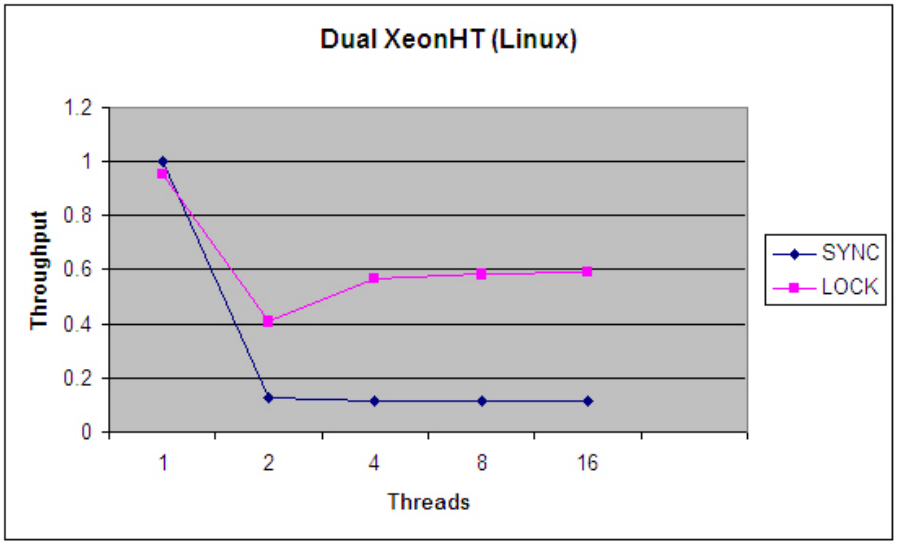
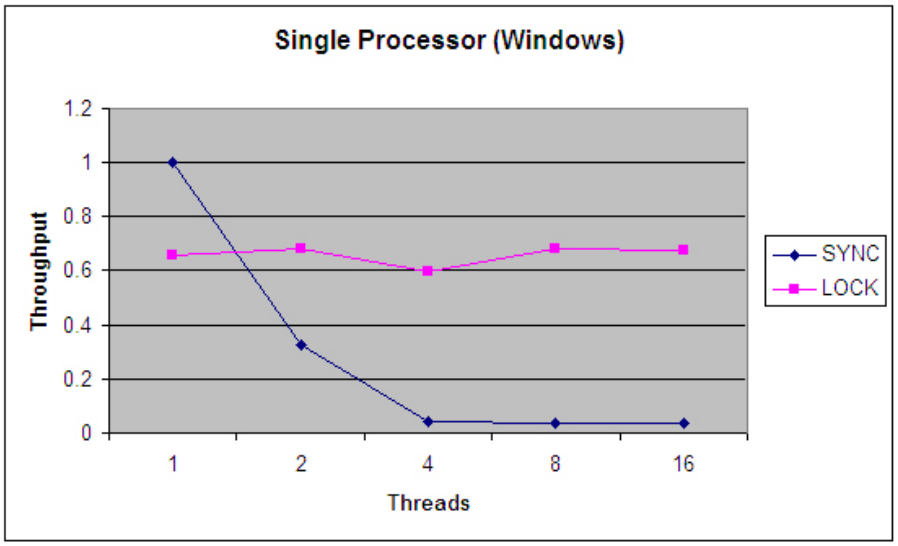


Figure 2. Throughput (normalized) for synchronization and Lock, four CPUs



The charts in Figure 1 and Figure 2 show the throughput in calls per second, normalized to the 1-thread `synchronized` case, of the various implementations. Each of the implementations converges

relatively quickly on a steady-state throughput, which generally entails that the processors are fully utilized and are spending some percentage of their time doing real work (computing random numbers) and some percentage of their time on scheduling overhead. You'll notice that the synchronized version does considerably worse in the face of any kind of contention, whereas the `Lock` version spends much less of its time on scheduling overhead, making room for much higher throughput and more effective CPU utilization.

Condition variables

The root `Object` class includes some special methods for communicating across threads -- `wait()`, `notify()`, and `notifyAll()`. These are advanced concurrency features, and many developers don't ever use them -- which is probably good, because they're quite subtle and easy to use incorrectly. Fortunately, with the addition of `java.util.concurrent` to JDK 5.0, even fewer cases exist where developers need to use these methods.

There is an interaction between notification and locking -- to `wait` or `notify` on an object, you must hold the lock for that object. Just as `Lock` is a generalization for synchronization, the `Lock` framework includes a generalization of `wait` and `notify` called `Condition`. A `Lock` object acts as a factory object for condition variables bound to that lock, and unlike with the standard `wait` and `notify` methods, there can be more than one condition variable associated with a given `Lock`. This simplifies the development of many concurrent algorithms. For example, the Javadoc for `Condition` shows an example of a bounded buffer implementation using two condition variables, "not full" and "not empty", which is more readable (and efficient) than the equivalent implementation with a single wait set per lock. The `Condition` methods that are analogous to `wait`, `notify`, and `notifyAll`, are named `await`, `signal`, and `signalAll`, because they cannot override the corresponding methods in `Object`.

That's not fair

If you peruse the Javadoc, you'll see that one of the arguments to the constructor of `ReentrantLock` is a boolean value that lets you choose whether you want a *fair* or an *unfair* lock. A fair lock is one where the threads acquire the lock in the same order they asked for it; an unfair lock may permit barging, where a thread can sometimes acquire a lock before another thread that asked for it first.

Why wouldn't we want to make all locks fair? After all, fairness is good and unfairness is bad, right? (It's not accidental that whenever kids want to appeal a decision, "that's not fair" almost certainly comes up. We think fairness is pretty important, and they know it.) In reality, the fairness guarantee for locks is a very strong one, and comes at a significant performance cost. The bookkeeping and synchronization required to ensure fairness mean that contended fair locks will have much lower throughput than unfair locks. As a default, you should set fair to `false` unless it is critical to the correctness of your algorithm that threads be serviced in exactly the order they queued up.

What about synchronization? Are built-in monitor locks fair? The answer is, to the surprise of many, that they aren't and never were. And no one complained about thread starvation, because the JVM ensures that all threads will eventually be granted the lock they are waiting for.

A statistical fairness guarantee is generally sufficient for most cases and costs a lot less than a deterministic fairness guarantee. So the fact that `ReentrantLock`s are "unfair" by default is simply making explicit something that was true all along for synchronization. If you didn't worry about it in the case of synchronization, don't worry about it for `ReentrantLock`.

Figure 3 and Figure 4 contain the same data as Figure 1 and Figure 2, with an additional data set for a new version of our random number benchmark, which uses a fair lock instead of the default barging lock. As you can see, fairness isn't free. Pay for it if you need it, but don't make it your default.

Figure 3. Relative throughput for synchronization, barging Lock, and fair Lock, with four CPUs

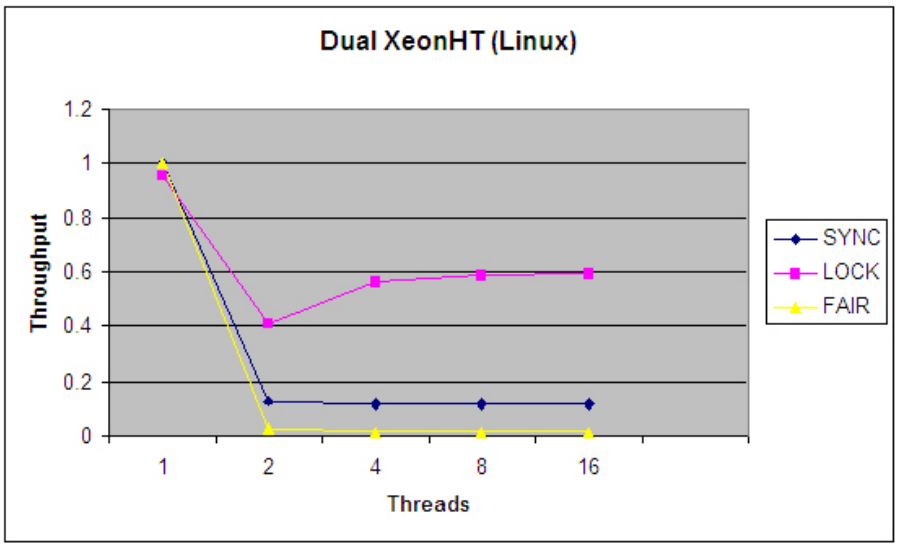
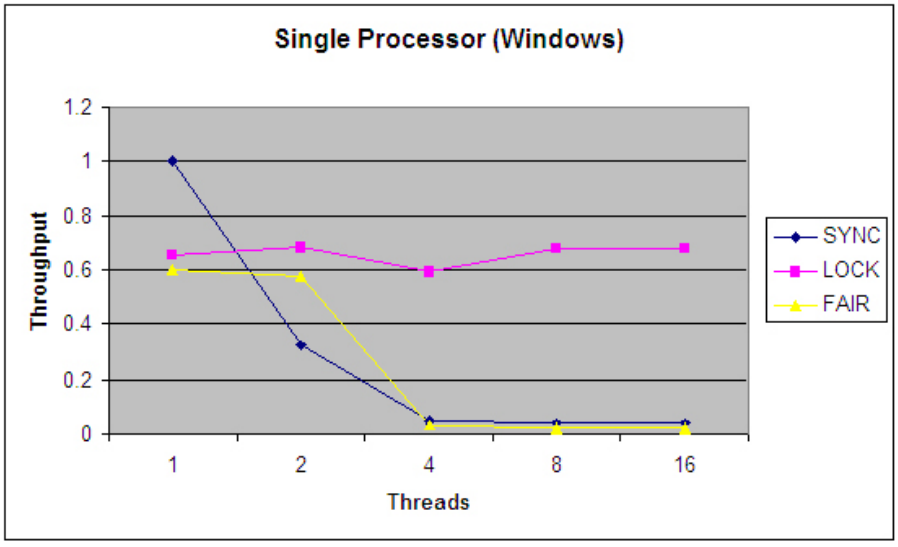


Figure 4. Relative throughput for synchronization, barging Lock, and fair Lock, with single CPU



Better in every way?

It appears that `ReentrantLock` is better in every way than `synchronized` -- it can do everything `synchronized` does, has the same memory and concurrency semantics, has features that `synchronized` does not, and has better performance under load. So should we just forget about `synchronized`, relegating it to the scrap heap of good ideas that have been subsequently improved upon? Or even rewrite our existing `synchronized` code in terms of `ReentrantLock`? In fact, several introductory books on Java programming take this approach in their chapters on multithreading, casting their examples entirely in terms of `Lock` while mentioning synchronization only in passing. I think this is taking a good thing too far.

Don't count synchronization out yet

While `ReentrantLock` is a very impressive implementation and has some significant advantages over synchronization, I believe the rush to consider synchronization a deprecated feature to be a serious mistake. *The locking classes in `java.util.concurrent.lock` are advanced tools for advanced users and situations*. In general, you should probably stick with synchronization unless you have a specific need for one of the advanced features of `Lock`, or if you have demonstrated evidence (not just a mere suspicion) that synchronization in this particular situation is a scalability bottleneck.

Why do I advocate such conservatism in adopting a clearly "better" implementation? Synchronization still has a few advantages over the locking classes in `java.util.concurrent.lock`. For one, it is impossible to forget to release a lock when using synchronization; the JVM does that for you when you exit the `synchronized` block. It is easy to forget to use a `finally` block to release the lock, to the great detriment of your program. Your program will pass its tests and lock up in the field, and it will be very difficult to figure out why (which itself is a good reason not to let junior developers use `Lock` at all).

Another reason is because when the JVM manages lock acquisition and release using synchronization, the JVM is able to include locking information when generating thread dumps. These can be invaluable for debugging, as they can identify the source of deadlocks or other unexpected behaviors. The `Lock` classes are just ordinary classes, and the JVM does not (yet) have any knowledge of which `Lock` objects are owned by specific threads. Furthermore, synchronization is familiar to nearly every Java developer and works on all versions of the JVM. Until JDK 5.0 becomes the standard, which will probably be at least two years from now, using the `Lock` classes will mean taking advantage of features not present on every JVM and not familiar to every developer.

When to select ReentrantLock over synchronized

So, when should we use `ReentrantLock`? The answer is pretty simple -- use it when you actually need something it provides that synchronized doesn't, like timed lock waits, interruptible lock waits, non-block-structured locks, multiple condition variables, or lock polling. `ReentrantLock` also has scalability benefits, and you should use it if you actually have a situation that exhibits high contention, but remember that the vast majority of synchronized blocks hardly ever exhibit

any contention, let alone high contention. I would advise developing with synchronization until synchronization has proven to be inadequate, rather than simply assuming "the performance will be better" if you use `ReentrantLock`. Remember, these are advanced tools for advanced users. (And truly advanced users tend to prefer the simplest tools they can find until they're convinced the simple tools are inadequate.) As always, make it right first, and then worry about whether or not you have to make it faster.

Summary

The `Lock` framework is a compatible replacement for synchronization, which offers many features not provided by `synchronized`, as well as implementations offering better performance under contention. However, the existence of these obvious benefits are not a good enough reason to always prefer `ReentrantLock` to `synchronized`. Instead, make the decision on the basis of whether you *need* the power of `ReentrantLock`. In the vast majority of cases, you will not -- synchronization works just fine, works on all JVMs, is understood by a wider range of developers, and is less error-prone. Save `Lock` for when you really need it. In those cases, you'll be glad you have it.

Resources

- Developers often confuse the cost of uncontended synchronization with contended synchronization. "Synchronization is not the enemy" (*developerWorks*, July 2001) offers some rough benchmarks for estimating the cost of uncontended synchronization, and "Reducing contention" (*developerWorks*, September 2001) offers some guidelines for reducing the impact of lock contention in applications.
- The Javadoc for Lock, ReentrantLock, and Condition offers further information on the application and behavior of the new locking classes.
- Doug Lea's *Concurrent Programming in Java, Second Edition* is a masterful book on the subtle issues surrounding multithreaded programming in Java programming.
- Find hundreds more Java technology resources on the *developerWorks* Java technology zone.
- Browse for books on these and other technical topics.

About the author

Brian Goetz

Brian Goetz has been a professional software developer for over 17 years. He is a Principal Consultant at Quiotix, a software development and consulting firm located in Los Altos, California, and he serves on several JCP Expert Groups. See Brian's published and upcoming articles in popular industry publications.