IBM

Java theory and practice: Going atomic

The new atomic classes are the hidden gems of java.util.concurrent

Level: Intermediate

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

23 Nov 2004

Until JDK 5.0, it was not possible to write wait-free, lock-free algorithms in the Java language without using native code. The addition of the atomic variable classes in java.util.concurrent changes that situation. Follow along with concurrency expert Brian Goetz as he explains how these new classes have enabled the development of highly scalable nonblocking algorithms in the Java language.

Fifteen years ago, multiprocessor systems were highly specialized systems costing hundreds of thousands of dollars (and most of them had two to four processors). Today, multiprocessor systems are cheap and plentiful, nearly every major microprocessor has built-in support for multiprocessing, and many support dozens or hundreds of processors.

To exploit the power of multiprocessor systems, applications are generally structured using multiple threads. But as anyone who's written a concurrent application can tell you, simply dividing up the work across multiple threads isn't enough to achieve good hardware utilization -- you must ensure that your threads spend most of their time actually doing work, rather than waiting for more work to do, or waiting for locks on shared data structures.

The problem: coordination between threads

Few tasks can be truly parallelized in such a way as to require *no* coordination between threads. Consider a thread pool, where the tasks being executed are generally independent of each other. If the thread pool feeds off a common work queue, then the process of removing elements from or adding elements to the work queue must be thread-safe, and that means coordinating access to the head, tail, or inter-node link pointers. And it is this coordination that causes all the trouble.

The standard approach: locking

The traditional way to coordinate access to shared fields in the Java language is to use synchronization, ensuring that all access to shared fields is done holding the appropriate lock. With synchronization, you are assured (assuming your class is properly written) that whichever thread holds the lock that protects a given set of variables will have exclusive access to those variables, and changes to those variables will become visible to other threads when they subsequently acquire the lock. The downside is that if the lock is heavily contended (threads frequently ask to acquire the lock when it is already held by another thread), throughput can suffer, as contended synchronization can be quite expensive. (Public Service Announcement: uncontended synchronization is now quite inexpensive on modern JVMs.)

Another problem with lock-based algorithms is that if a thread holding a lock is delayed (due to a page fault, scheduling delay, or other unexpected delay), then *no* thread requiring that lock may make progress.

Volatile variables can also be used to store shared variables at a lower cost than that of synchronization, but they have limitations. While writes to volatile variables are guaranteed to be immediately visible to other threads, there is no way to render a read-modify-write sequence of operations atomic, meaning, for example, that a volatile variable cannot be used to reliably implement a mutex (mutual exclusion lock) or a counter.

Implementing counters and mutexes with locking

Consider the development of a thread-safe counter class, which exposes `get()`, `increment()`, and `decrement()` operations. Listing 1 shows an example of how this class might be implemented using locks (synchronization). Note that all the methods, even `get()`, need to be synchronized for the class to be thread-safe, to ensure that no updates are lost, and that all threads see the most recent value of the counter.

Listing 1. A synchronized counter class

```
public class SynchronizedCounter {
    private int value;

    public synchronized int getValue() { return value; }
    public synchronized int increment() { return ++value; }
    public synchronized int decrement() { return --value; }
}
```

The `increment()` and `decrement()` operations are atomic read-modify-write operations -- to safely increment the counter, you must take the current value, add one to it, and write the new value out, all as a single operation that cannot be interrupted by another thread. Otherwise, if two threads tried to execute the increment simultaneously, an unlucky interleaving of operations would result in the counter being incremented only once, instead of twice. (Note that this operation cannot be achieved reliably by making the value instance variable volatile.)

The atomic read-modify-write combination shows up in many concurrent algorithms. The code in Listing 2 implements a simple mutex, and the `acquire()` method is also an atomic read-modify-write operation. To acquire the mutex, you have to ensure that no one else holds it (`curOwner == null`), and then record the fact that you own it (`curOwner = Thread.currentThread()`), all free from the possibility that another thread could come along in the middle and modify the `curOwner field`.

Listing 2. A synchronized mutex class

```
public class SynchronizedMutex {
    private Thread curOwner = null;

    public synchronized void acquire() throws InterruptedException {
        if (Thread.interrupted()) throw new InterruptedException();
        while (curOwner != null)
            wait();
        curOwner = Thread.currentThread();
    }

    public synchronized void release() {
        if (curOwner == Thread.currentThread()) {
            curOwner = null;
            notify();
        } else
            throw new IllegalStateException("not owner of mutex");
    }
}
```

The counter class in Listing 1 works reliably, and in the presence of little or no contention will perform fine. However, under heavy contention, performance will suffer dramatically, as the JVM spends more time dealing with scheduling threads and managing contention and queues of waiting threads and less time doing real work, like incrementing counters. You might recall the graphs from last month's column showing how throughput can drop dramatically once multiple threads contend for a built-in monitor lock using synchronization. While that column showed how the new `ReentrantLock` class is a more scalable replacement for synchronization, for some problems there is an even better approach.

Problems with locking

With locking, if one thread attempts to acquire a lock that is already held by another thread, the thread will block until the lock becomes available. This approach has some obvious drawbacks, including the fact that while a thread is blocked waiting for a lock, it cannot do anything else. This scenario could be a disaster if the blocked thread is a high-priority task (a hazard known as *priority inversion*).

Using locks has some other hazards as well, such as deadlock (which can happen when multiple locks are acquired in an inconsistent order). Even in the absence of hazards like this, locks are simply a relatively coarse-grained coordination mechanism, and as such, are fairly "heavyweight" for managing a simple operation such as

incrementing a counter or updating who owns a mutex. It would be nice if there were a finer-grained mechanism for reliably managing concurrent updates to individual variables; and on most modern processors, there is.

Hardware synchronization primitives

As stated earlier, most modern processors include support for multiprocessing. While this support, of course, includes the ability for multiple processors to share peripherals and main memory, it also generally includes enhancements to the instruction set to support the special requirements of multiprocessing. In particular, nearly every modern processor has instructions for updating shared variables in a way that can either detect or prevent concurrent access from other processors.

Compare and swap (CAS)

The first processors that supported concurrency provided atomic test-and-set operations, which generally operated on a single bit. The most common approach taken by current processors, including Intel and Sparc processors, is to implement a primitive called *compare-and-swap*, or CAS. (On Intel processors, compare-and-swap is implemented by the cmpxchg family of instructions. PowerPC processors have a pair of instructions called "load and reserve" and "store conditional" that accomplish the same goal; similar for MIPS, except the first is called "load linked.")

A CAS operation includes three operands -- a memory location (V), the expected old value (A), and a new value (B). The processor will atomically update the location to the new value if the value that is there matches the expected old value, otherwise it will do nothing. In either case, it returns the value that was at that location prior to the CAS instruction. (Some flavors of CAS will instead simply return whether or not the CAS succeeded, rather than fetching the current value.) CAS effectively says "I think location V should have the value A; if it does, put B in it, otherwise, don't change it but tell me what value is there now."

The natural way to use CAS for synchronization is to read a value A from an address V, perform a multistep computation to derive a new value B, and then use CAS to change the value of V from A to B. The CAS succeeds if the value at V has not been changed in the meantime.

Instructions like CAS allow an algorithm to execute a read-modify-write sequence without fear of another thread modifying the variable in the meantime, because if another thread did modify the variable, the CAS would detect it (and fail) and the algorithm could retry the operation. Listing 3 illustrates the behavior (but not performance characteristics) of the CAS operation, but the value of CAS is that it is implemented in hardware and is extremely lightweight (on most processors):

Listing 3. Code illustrating the behavior (but not performance) of compare-and-swap

```
public class SimulatedCAS {
     private int value;

     public synchronized int getValue() { return value; }

        public synchronized int compareAndSwap(int expectedValue, int newValue) {
         int oldValue = value;
         if (value == expectedValue)
             value = newValue;
         return oldValue;
     }
}
```

Implementing counters with CAS

Concurrent algorithms based on CAS are called *lock-free*, because threads do not ever have to wait for a lock (sometimes called a mutex or critical section, depending on the terminology of your threading platform). Either the CAS operation succeeds or it doesn't, but in either case, it completes in a predictable amount of time. If the CAS fails, the caller can retry the CAS operation or take other action as it sees fit. Listing 4 shows the counter

class rewritten to use CAS instead of locking:

Listing 4. Implementing a counter with compare-and-swap

```
public class CasCounter {
    private SimulatedCAS value;

    public int getValue() {
        return value.getValue();
    }

    public int increment() {
        int oldValue = value.getValue();
        while (value.compareAndSwap(oldValue, oldValue + 1) != oldValue)
            oldValue = value.getValue();
        return oldValue + 1;
    }
}
```

Lock-free and wait-free algorithms

An algorithm is said to be *wait-free* if every thread will continue to make progress in the face of arbitrary delay (or even failure) of other threads. By contrast, a *lock-free* algorithm requires only that *some* thread always make progress. (Another way of defining wait-free is that each thread is guaranteed to correctly compute its operations in a bounded number of its own steps, regardless of the actions, timing, interleaving, or speed of the other threads. This bound may be a function of the number of threads in the system; for example, if ten threads each execute the `CasCounter.increment()` operation once, in the worst case each thread will have to retry at most nine times before the increment is complete.)

Substantial research has gone into wait-free and lock-free algorithms (also called *nonblocking algorithms*) over the past 15 years, and nonblocking algorithms have been discovered for many common data structures. Nonblocking algorithms are used extensively at the operating system and JVM level for tasks such as thread and process scheduling. While they are more complicated to implement, they have a number of advantages over lock-based alternatives -- hazards like priority inversion and deadlock are avoided, contention is less expensive, and coordination occurs at a finer level of granularity, enabling a higher degree of parallelism.

Atomic variable classes

Until JDK 5.0, it was not possible to write wait-free, lock-free algorithms in the Java language without using native code. With the addition of the atomic variables classes in the `java.util.concurrent.atomic` package, that has changed. The atomic variable classes all expose a compare-and-set primitive (similar to compare-and-swap), which is implemented using the fastest native construct available on the platform (compare-and-swap, load linked/store conditional, or, in the worst case, spin locks). Nine flavors of atomic variables are provided in the `java.util.concurrent.atomic` package (`AtomicInteger`; `AtomicLong`; `AtomicReference`; `AtomicBoolean`; array forms of atomic integer; long; reference; and atomic marked reference and stamped reference classes, which atomically update a pair of values).

The atomic variable classes can be thought of as a generalization of `volatile` variables, extending the concept of volatile variables to support atomic conditional compare-and-set updates. Reads and writes of atomic variables have the same memory semantics as read and write access to volatile variables.

While the atomic variable classes might look superficially like the `SynchronizedCounter` example in Listing 1, the similarity is only superficial. Under the hood, operations on atomic variables get turned into the hardware primitives that the platform provides for concurrent access, such as compare-and-swap.

Finer grained means lighter weight

A common technique for tuning the scalability of a concurrent application that is experiencing contention is to reduce the granularity of the lock objects used, in the hopes that more lock acquisitions will go from contended to

uncontended. The conversion from locking to atomic variables achieves the same end -- by switching to a finer-grained coordination mechanism, fewer operations become contended, improving throughput.

Atomic variables in java.util.concurrent

Nearly all the classes in the `java.util.concurrent` package use atomic variables instead of synchronization, either directly or indirectly. Classes like `ConcurrentLinkedQueue` use atomic variables to directly implement wait-free algorithms, and classes like `ConcurrentHashMap` use `ReentrantLock` for locking where needed. `ReentrantLock`, in turn, uses atomic variables to maintain the queue of threads waiting for the lock.

These classes could not have been constructed without the JVM improvements in JDK 5.0, which exposed (to the class libraries, but not to user classes) an interface to access hardware-level synchronization primitives. The atomic variable classes, and other classes in java.util.concurrent, in turn expose these features to user classes.

The ABA problem
Because CAS basically asks "is the value of V still A" before changing V, it is possible for a CAS-based algorithm to be confused by the value changing from A to B and back to A between the time V was first read and the time the CAS on V is performed. In such a case, the CAS operation would succeed, but in some situations the result might not be what is desired. (Note that the counter and mutex examples from Listing 1 and Listing 2 are immune to this problem, but not all algorithms are.) This problem is called the *ABA problem*, and is generally dealt with by associating a tag, or version number, with each value to be CASed, and atomically updating both the value and the tag. The `AtomicStampedReference` class provides support for this approach.

Achieving higher throughput with atomic variables

Last month, I looked at how the `ReentrantLock` class offered a scalability benefit over synchronization, and constructed a simple, high-contention example benchmark that simulated rolling dice with a pseudorandom number generator. I showed you implementations that did their coordination with synchronization, `ReentrantLock`, and fair `ReentrantLock`, and presented the results. This month, I'll add another implementation to that benchmark, one that uses `AtomicLong` to update the PRNG state.

Listing 5 shows the PRNG implementation using synchronization, and the alternate implementation using CAS. Note that the CAS must be executed in a loop, because it may fail one or more times before succeeding, which is always the case with code that uses CAS.

Listing 5. Implementing a thread-safe PRNG with synchronization and atomic variables

```
public class PseudoRandomUsingSynch implements PseudoRandom {
    private int seed;

    public PseudoRandomUsingSynch(int s) { seed = s; }

    public synchronized int nextInt(int n) {
        int s = seed;
        seed = Util.calculateNext(seed);
        return s % n;
    }
}

public class PseudoRandomUsingAtomic implements PseudoRandom {
    private final AtomicInteger seed;

    public PseudoRandomUsingAtomic(int s) {
        seed = new AtomicInteger(s);
    }

    public int nextInt(int n) {
        for (;;) {
            int s = seed.get();
            int nexts = Util.calculateNext(s);
            if (seed.compareAndSet(s, nexts))
                return s % n;
        }
    }
```

```
}
```

The graphs in Figures 1 and 2 below are similar to those shown last month, with the addition of another line for the atomic-based approach. The graphs show throughput (in rolls per second) for random number generation using various numbers of threads on an 8-way Ultrasparc3 box and a single-processor Pentium 4 box. The numbers of threads in the tests are deceptive; these threads exhibit far more contention than is typical, so they show the break-even between `ReentrantLock` and atomic variables at a far lower number of threads than would be the case in a more realistic program. You'll see that atomic variables offer an additional improvement over `ReentrantLock`, which already had a big advantage over synchronization. (Because so little work is done in each unit of work, the graphs below probably understate the scalability benefits of atomic variables compared to ReentrantLock.)

Figure 1. Benchmark throughput for synchronization, ReentrantLock, fair Lock, and AtomicLong on an 8-way Ultrasparc3

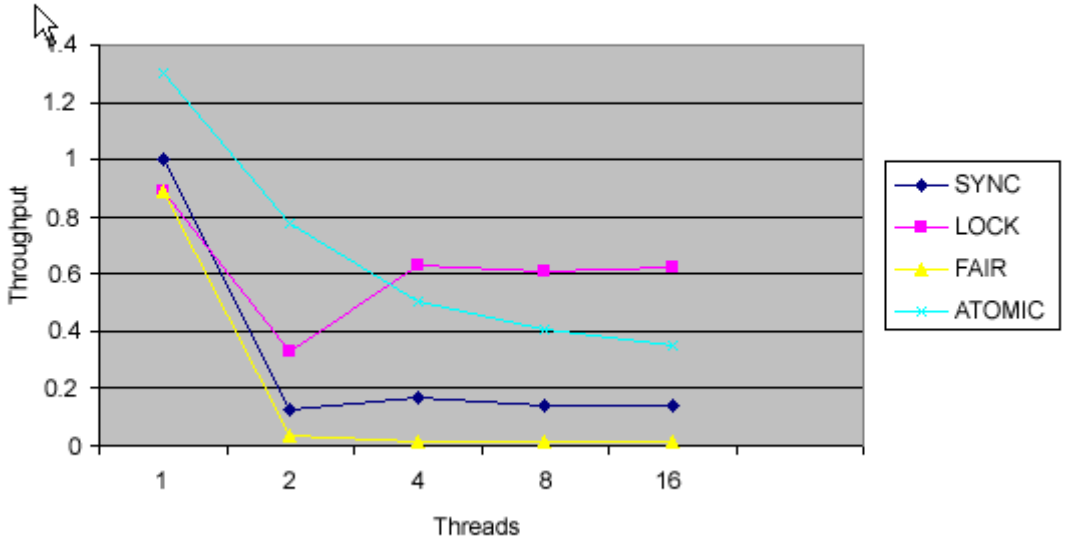
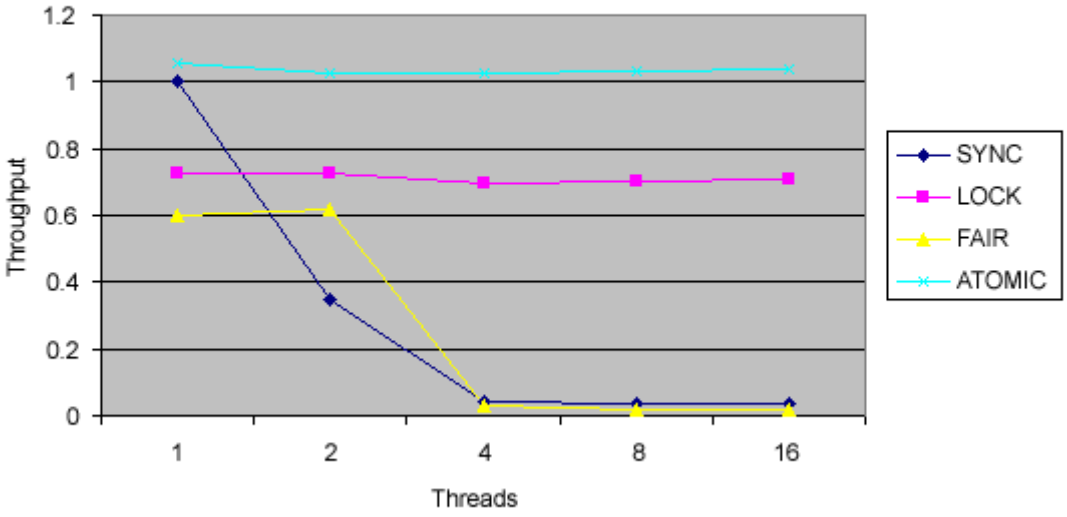


Figure 2. Benchmark throughput for synchronization, ReentrantLock, fair Lock, and AtomicLong on a single-processor Pentium 4



Most users are unlikely to develop nonblocking algorithms using atomic variables on their own -- they are more

likely to use the versions provided in `java.util.concurrent`, such as `ConcurrentLinkedQueue`. But in case you're wondering where the performance boost of these classes comes from, compared to their analogues in prior JDKs, it's the use of the fine-grained, hardware-level concurrency primitives exposed through the atomic variable classes.

Developers may find use for atomic variables directly as a higher-performance replacement for shared counters, sequence number generators, and other independent shared variables that otherwise would have to be protected by synchronization.

Summary

JDK 5.0 is a huge step forward in the development of high-performance, concurrent classes. By exposing new low-level coordination primitives internally, and providing a set of public atomic variable classes, it now becomes practical, for the first time, to develop wait-free, lock-free algorithms in the Java language. The classes in `java.util.concurrent` are in turn built on these low-level atomic variable facilities, giving them their significant scalability advantage over previous classes that performed similar functions. While you may never use atomic variables directly in your classes, there are good reasons to cheer for their existence.

Resources

- Participate in the discussion forum.
- Read the complete *Java theory and practice* series by Brian Goetz.
- The package documentation for the java.util.concurrent.atomic package is a good place to start for understanding the atomic variable classes.
- Web sites like Wikipedia include definitions for lock-free and wait-free synchronization.
- The C2 Wiki also offers definitions for wait-free and lock-free synchronization.
- "Concurrent programming without locks" by Keir Fraser and Tim Harris summarizes the alternatives to locking, including compare-and-swap, for building concurrent algorithms.
- The WARPing Group (Wait-free techniques for real-time processing) site summarizes research in wait-free algorithms.
- "More flexible, scalable locking in JDK 5.0" (developerWorks, October 2004), examined the scalability advantage of `ReentrantLock` and introduced the random-number generation benchmark used in this column.
- Doug Lea's *Concurrent Programming in Java, Second Edition* (Addison-Wesley Professional 1999) is a masterful book on the subtle issues surrounding multithreaded programming in Java applications.
- 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 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.