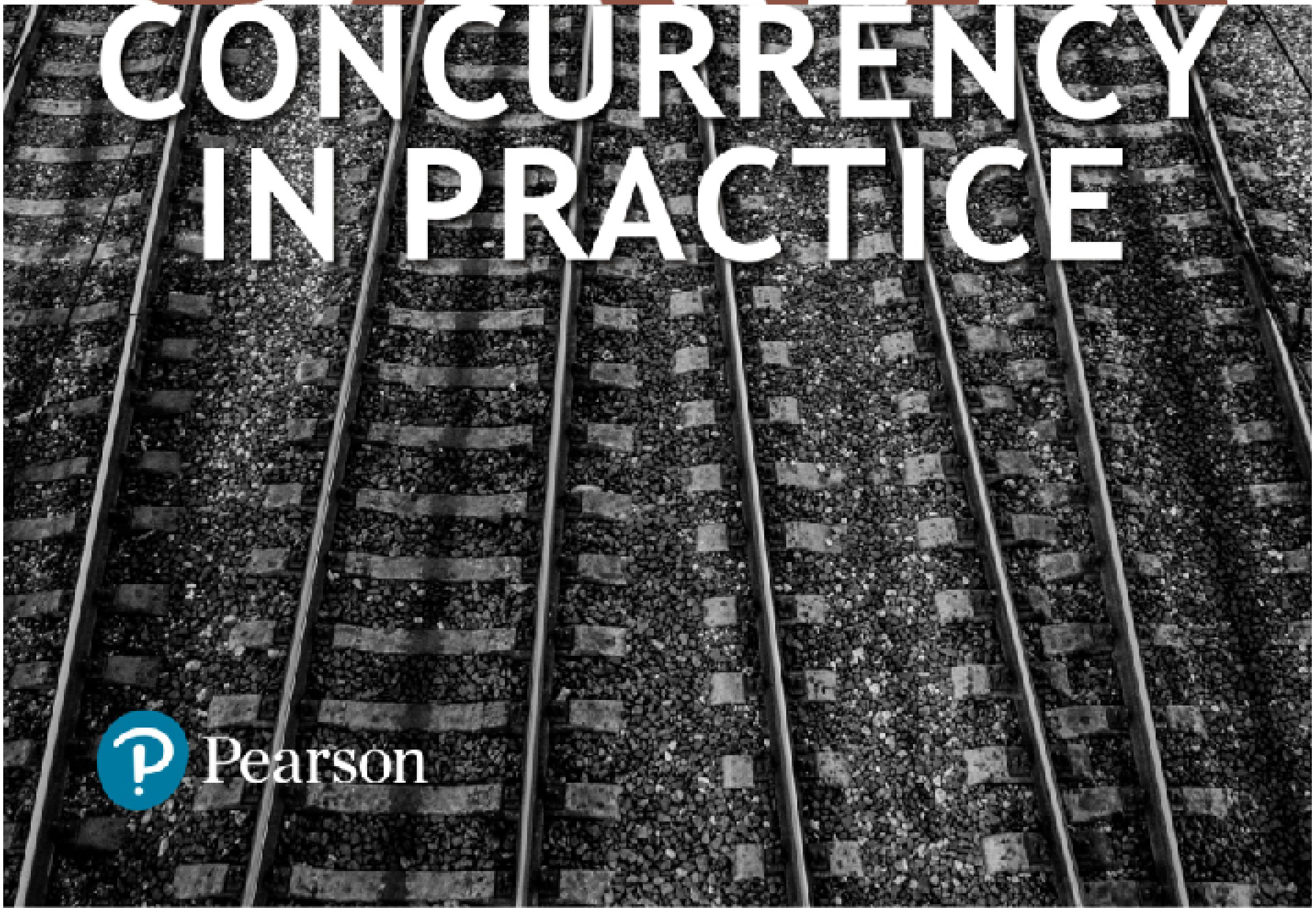
This online edition of Java Concurrency in Practice contains only three chapters

BRIAN GOETZ

WITH TIM PEIERLS, JOSHUA BLOCH
JOSEPH BOWBEER, DAVID HOLMES
AND DOUG LEA

JAVA CONCURRENCY IN PRACTICE



Pearson

Chapter 14

Atomic Variables and Nonblocking Synchronization

Many of the classes in `java.util.concurrent`, such as `Semaphore` and `ConcurrentLinkedQueue`, provide better performance and scalability than alternatives using `synchronized`. In this chapter, we take a look at the primary source of this performance boost: atomic variables and nonblocking synchronization.

Much of the recent research on concurrent algorithms has focused on *nonblocking algorithms*, which use low-level atomic machine instructions such as *compare-and-swap* instead of locks to ensure data integrity under concurrent access. Nonblocking algorithms are used extensively in operating systems and JVMs for thread and process scheduling, garbage collection, and to implement locks and other concurrent data structures.

Nonblocking algorithms are considerably more complicated to design and implement than lock-based alternatives, but they can offer significant scalability and liveness advantages. They coordinate at a finer level of granularity and can greatly reduce scheduling overhead because they don't block when multiple threads contend for the same data. Further, they are immune to deadlock and other liveness problems. In lock-based algorithms, other threads cannot make progress if a thread goes to sleep or spins while holding a lock, whereas nonblocking algorithms are impervious to individual thread failures. As of Java 5.0, it is possible to build efficient nonblocking algorithms in Java using the *atomic variable classes* such as `AtomicInteger` and `AtomicReference`.

Atomic variables can also be used as "better volatile variables" even if you are not developing nonblocking algorithms. Atomic variables offer the same memory semantics as volatile variables, but with additional support for atomic updates—making them ideal for counters, sequence generators, and statistics gathering while offering better scalability than lock-based alternatives.

14.1 Disadvantages of locking

Coordinating access to shared state using a consistent locking protocol ensures that whichever thread holds the lock guarding a set of variables has exclusive access to those variables, and that any changes made to those variables are visible to other threads that subsequently acquire the lock.

Modern JVMs can optimize uncontended lock acquisition and release fairly effectively, but if multiple threads request the lock at the same time the JVM enlists the help of the operating system. If it gets to this point, some unfortunate thread will be suspended and have to be resumed later.[1] When that thread is resumed, it may have to wait for other threads to finish their scheduling quanta before it is actually scheduled. Suspending and resuming a thread has a lot of overhead and generally entails a lengthy interruption. For lock-based classes with fine-grained operations (such as the synchronized collections classes, where most methods contain only a few operations), the ratio of scheduling overhead to useful work can be quite high *when the lock is frequently contended*.

Volatile variables are a lighter-weight synchronization mechanism than locking because they do not involve context switches or thread scheduling. However, volatile variables have some limitations compared to locking: while they provide similar visibility guarantees, they cannot be used to construct atomic compound actions. This means that volatile variables cannot be used when one variable depends on another, or when the new value of a variable depends on its old value. This limits when volatile variables are appropriate, since they cannot be used to reliably implement common tools such as counters or mutexes.[2]

For example, while the increment operation (`++i`) may *look* like an atomic operation, it is actually three distinct operations—fetch the current value of the variable, add one to it, and then write the updated value back. In order to not lose an update, the entire read-modify-write operation must be atomic. So far, the only way we've seen to do this is with locking, as in `Counter` on page 56.

`Counter` is thread-safe, and in the presence of little or no contention performs just fine. But under contention, performance suffers because of context-switch overhead and scheduling delays. When locks are held so briefly, being put to sleep is a harsh penalty for asking for the lock at the wrong time.

Locking has a few other disadvantages. When a thread is waiting for a lock, it cannot do anything else. If a thread holding a lock is delayed (due to a page fault, scheduling delay, or the like), then no thread that needs that lock can make progress. This can be a serious problem if the blocked thread is a high-priority thread but the thread holding the lock is a lower-priority thread—a performance hazard known as *priority inversion*. Even though the higher-priority thread should have precedence, it must wait until the lock is released, and this effectively downgrades its priority to that of the lower-priority thread. If a thread holding a lock is permanently blocked (due to an infinite loop, deadlock, livelock, or other liveness failure), any threads waiting for that lock can never make progress.

Even ignoring these hazards, locking is simply a heavyweight mechanism for fine-grained operations such as incrementing a counter. It would be nice to have a finer-grained technique for managing contention between threads—something

1. A smart JVM need not necessarily suspend a thread if it contends for a lock; it could use profiling data to decide adaptively between suspension and spin locking based on how long the lock has been held during previous acquisitions.

2. It is theoretically possible, though wholly impractical, to use the semantics of `volatile` to construct mutexes and other synchronizers; see (Raynal, 1986).

like volatile variables, but offering the possibility of atomic updates as well. Happily, modern processors offer us precisely such a mechanism.

14.2 Hardware support for concurrency

Exclusive locking is a *pessimistic* technique—it assumes the worst (if you don't lock your door, gremlins will come in and rearrange your stuff) and doesn't proceed until you can guarantee, by acquiring the appropriate locks, that other threads will not interfere.

For fine-grained operations, there is an alternate approach that is often more efficient—the *optimistic* approach, whereby you proceed with an update, hopeful that you can complete it without interference. This approach relies on *collision detection* to determine if there has been interference from other parties during the update, in which case the operation fails and can be retried (or not). The optimistic approach is like the old saying, "It is easier to obtain forgiveness than permission", where "easier" here means "more efficient".

Processors designed for multiprocessor operation provide special instructions for managing concurrent access to shared variables. Early processors had atomic *test-and-set*, *fetch-and-increment*, or *swap* instructions sufficient for implementing mutexes that could in turn be used to implement more sophisticated concurrent objects. Today, nearly every modern processor has some form of atomic read-modify-write instruction, such as *compare-and-swap* or *load-linked/store-conditional*. Operating systems and JVMs use these instructions to implement locks and concurrent data structures, but until Java 5.0 they had not been available directly to Java classes.

14.2.1 Compare and swap

The approach taken by most processor architectures, including IA32 and Sparc, is to implement a *compare-and-swap* (CAS) instruction. (Other processors, such as PowerPC, implement the same functionality with a pair of instructions: *load-linked* and *store-conditional*.) CAS has three operands—a memory location V on which to operate, the expected old value A, and the new value B. CAS atomically updates V to the new value B, but only if the value in V matches the expected old value A; otherwise it does nothing. In either case, it returns the value currently in V. (The variant called compare-and-set instead returns whether the operation succeeded.) CAS means "I think V should have the value A; if it does, put B there, otherwise don't change it but tell me I was wrong." CAS is an optimistic technique—it proceeds with the update in the hope of success, and can detect failure if another thread has updated the variable since it was last examined. `SimulatedCAS` in Listing 14.1 illustrates the semantics (but not the implementation or performance) of CAS.

When multiple threads attempt to update the same variable simultaneously using CAS, one wins and updates the variable's value, and the rest lose. But the losers are not punished by suspension, as they could be if they failed to acquire a lock; instead, they are told that they didn't win the race this time but

```
@ThreadSafe
public class SimulatedCAS {
    @GuardedBy("this") private int value;

    public synchronized int get() { return value; }

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

    public synchronized boolean compareAndSet(int expectedValue,
                                              int newValue) {
        return (expectedValue
                == compareAndSwap(expectedValue, newValue));
    }
}
```

Listing 14.1. Simulated CAS operation.

can try again. Because a thread that loses a CAS is not blocked, it can decide whether it wants to try again, take some other recovery action, or do nothing.[3] This flexibility eliminates many of the liveness hazards associated with locking (though in unusual cases can introduce the risk of *livelock*—see Section 8.3.3).

The typical pattern for using CAS is first to read the value *A* from *V*, derive the new value *B* from *A*, and then use CAS to atomically change *V* from *A* to *B* so long as no other thread has changed *V* to another value in the meantime. CAS addresses the problem of implementing atomic read-modify-write sequences without locking, because it can detect interference from other threads.

14.2.2 A nonblocking counter

`CasCounter` in Listing 14.2 implements a thread-safe counter using CAS. The increment operation follows the canonical form—fetch the old value, transform it to the new value (adding one), and use CAS to set the new value. If the CAS fails, the operation is immediately retried. Retrying repeatedly is usually a reasonable strategy, although in cases of extreme contention it might be desirable to wait or back off before retrying to avoid livelock.

3. Doing nothing may be a perfectly sensible response to a failed CAS; in some nonblocking algorithms, such as the linked queue algorithm in Section 14.4.2, a failed CAS means that someone else already did the work you were planning to do.

CasCounter does not block, though it may have to retry several[4] times if other threads are updating the counter at the same time. (In practice, if all you need is a counter or sequence generator, just use AtomicInteger or AtomicLong, which provide atomic increment and other arithmetic methods.)

```
@ThreadSafe
public class CasCounter {
    private SimulatedCAS value;

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

    public int increment() {
        int v;
        do {
            v = value.get();
        }
        while (v != value.compareAndSwap(v, v + 1));
        return v + 1;
    }
}
```

Listing 14.2. Nonblocking counter using CAS.

At first glance, the CAS-based counter looks as if it should perform worse than a lock-based counter; it has more operations and a more complicated control flow, and depends on the seemingly complicated CAS operation. But in reality, CAS-based counters significantly outperform lock-based counters if there is even a small amount of contention, and often even if there is no contention. The fast path for uncontended lock acquisition typically requires at least one CAS plus other lock-related housekeeping, so more work is going on in the best case for a lock-based counter than in the normal case for the CAS-based counter. Since the CAS succeeds most of the time (assuming low to moderate contention), the hardware will correctly predict the branch implicit in the while loop, minimizing the overhead of the more complicated control logic.

The language syntax for locking may be compact, but the work done by the JVM and OS to manage locks is not. Locking entails traversing a relatively complicated code path in the JVM and may entail OS-level locking, thread suspension, and context switches. In the best case, locking requires at least one CAS, so using locks moves the CAS out of sight but doesn't save any actual execution cost. On the other hand, executing a CAS from within the program involves no JVM code, system calls, or scheduling activity. What looks like a longer code path at the application level is in fact a much shorter code path when JVM and OS activity are

4. Theoretically, it could have to retry arbitrarily many times if other threads keep winning the CAS race; in practice, this sort of starvation rarely happens.

taken into account. The primary disadvantage of CAS is that it forces the caller to deal with contention (by retrying, backing off, or giving up), whereas locks deal with contention automatically by blocking until the lock is available.[5]

CAS performance varies widely across processors. On a single-CPU system, a CAS typically takes on the order of a handful of clock cycles, since no synchronization across processors is necessary. As of this writing, the cost of an uncontended CAS on multiple CPU systems ranges from about ten to about 150 cycles; CAS performance is a rapidly moving target and varies not only across architectures but even across versions of the same processor. Competitive forces will likely result in continued CAS performance improvement over the next several years. A good rule of thumb is that the cost of the "fast path" for *uncontended* lock acquisition and release on most processors is approximately twice the cost of a CAS.

14.2.3 CAS support in the JVM

So, how does Java code convince the processor to execute a CAS on its behalf? Prior to Java 5.0, there was no way to do this short of writing native code. In Java 5.0, low-level support was added to expose CAS operations on `int`, `long`, and object references, and the JVM compiles these into the most efficient means provided by the underlying hardware. On platforms supporting CAS, the runtime inlines them into the appropriate machine instruction(s); in the worst case, if a CAS-like instruction is not available the JVM uses a spin lock. This low-level JVM support is used by the atomic variable classes (`AtomicXxx` in `java.util.concurrent.atomic`) to provide an efficient CAS operation on numeric and reference types; these atomic variable classes are used, directly or indirectly, to implement most of the classes in `java.util.concurrent`.

14.3 Atomic variable classes

Atomic variables are finer-grained and lighter-weight than locks, and are critical for implementing high-performance concurrent code on multiprocessor systems. Atomic variables limit the scope of contention to a single variable; this is as fine-grained as you can get (assuming your algorithm can even be implemented using such fine granularity). The fast (uncontended) path for updating an atomic variable is no slower than the fast path for acquiring a lock, and usually faster; the slow path is definitely faster than the slow path for locks because it does not involve suspending and rescheduling threads. With algorithms based on atomic variables instead of locks, threads are more likely to be able to proceed without delay and have an easier time recovering if they do experience contention.

The atomic variable classes provide a generalization of volatile variables to support atomic conditional read-modify-write operations. `AtomicInteger` represents an `int` value, and provides `get` and `set` methods with the same memory

5. Actually, the biggest disadvantage of CAS is the difficulty of constructing the surrounding algorithms correctly.

semantics as reads and writes to a volatile `int`. It also provides an atomic `compareAndSet` method (which if successful has the memory effects of both reading and writing a volatile variable) and, for convenience, atomic add, increment, and decrement methods. `AtomicInteger` bears a superficial resemblance to an extended `Counter` class, but offers far greater scalability under contention because it can directly exploit underlying hardware support for concurrency.

There are twelve atomic variable classes, divided into four groups: scalars, field updaters, arrays, and compound variables. The most commonly used atomic variables are the scalars: `AtomicInteger`, `AtomicLong`, `AtomicBoolean`, and `AtomicReference`. All support CAS; the `Integer` and `Long` versions support arithmetic as well. (To simulate atomic variables of other primitive types, you can cast `short` or `byte` values to and from `int`, and use `floatToIntBits` or `doubleToLongBits` for floating-point numbers.)

The atomic array classes (available in `Integer`, `Long`, and `Reference` versions) are arrays whose elements can be updated atomically. The atomic array classes provide volatile access semantics to the elements of the array, a feature not available for ordinary arrays—a `volatile` array has `volatile` semantics only for the array reference, not for its elements. (The other types of atomic variables are discussed in Sections 14.4.3 and 14.4.4.)

While the atomic scalar classes extend `Number`, they do not extend the primitive wrapper classes such as `Integer` or `Long`. In fact, they cannot: the primitive wrapper classes are immutable whereas the atomic variable classes are mutable. The atomic variable classes also do not redefine `hashCode` or `equals`; each instance is distinct. Like most mutable objects, they are not good candidates for keys in hash-based collections.

14.3.1 Atomics as "better volatiles"

In Section 3.4.2, we used a `volatile` reference to an immutable object to update multiple state variables atomically. That example relied on check-then-act, but in that particular case the race was harmless because we did not care if we occasionally lost an update. In most other situations, such a check-then-act would not be harmless and could compromise data integrity. For example, `NumberRange` on page 67 could not be implemented safely with a `volatile` reference to an immutable holder object for the upper and lower bounds, nor with using atomic integers to store the bounds. Because an invariant constrains the two numbers and they cannot be updated simultaneously while preserving the invariant, a number range class using `volatile` references or multiple atomic integers will have unsafe check-then-act sequences.

ext Page

We can combine the technique from `OneValueCache` with atomic references to close the race condition by *atomically* updating the reference to an immutable object holding the lower and upper bounds. `CasNumberRange` in Listing 14.3 uses an `AtomicReference` to an `IntPair` to hold the state; by using `compareAndSet` it can update the upper or lower bound without the race conditions of `NumberRange`.

```
public class CasNumberRange {
    @Immutable
    private static class IntPair {
        final int lower; // Invariant: lower <= upper
        final int upper;
        ...
    }
    private final AtomicReference<IntPair> values =
        new AtomicReference<IntPair>(new IntPair(0, 0));

    public int getLower() { return values.get().lower; }
    public int getUpper() { return values.get().upper; }

    public void setLower(int i) {
        while (true) {
            IntPair oldv = values.get();
            if (i > oldv.upper)
                throw new IllegalArgumentException(
                    "Can't set lower to " + i + " > upper");
            IntPair newv = new IntPair(i, oldv.upper);
            if (values.compareAndSet(oldv, newv))
                return;
        }
    }
    // similarly for setUpper
}
```

Listing 14.3. Preserving multivariable invariants using CAS.

14.3.2 Performance comparison: locks versus atomic variables

To demonstrate the differences in scalability between locks and atomic variables, we constructed a benchmark comparing several implementations of a pseudo-random number generator (PRNG). In a PRNG, the next "random" number is a deterministic function of the previous number, so a PRNG must remember the previous number as part of its state.

Listings 14.4 and 14.5 show two implementations of a thread-safe PRNG, one using `ReentrantLock` and the other using `AtomicInteger`. The test driver invokes each repeatedly; each iteration generates a random number (which fetches and modifies the shared `seed` state) and also performs a number of "busy-work" iterations that operate strictly on thread-local data. This simulates typical operations that include some portion of operating on shared state and some portion of operating on thread-local state.

ext Page

Figures 14.1 and 14.2 show throughput with low and moderate levels of simulated work in each iteration. With a low level of thread-local computation, the lock or atomic variable experiences heavy contention; with more thread-local compu-

```java
@ThreadSafe
public class ReentrantLockPseudoRandom extends PseudoRandom {
    private final Lock lock = new ReentrantLock(false);
    private int seed;

    ReentrantLockPseudoRandom(int seed) {
        this.seed = seed;
    }

    public int nextInt(int n) {
        lock.lock();
        try {
            int s = seed;
            seed = calculateNext(s);
            int remainder = s % n;
            return remainder > 0 ? remainder : remainder + n;
        } finally {
            lock.unlock();
        }
    }
}
```

Listing 14.4. Random number generator using `ReentrantLock`.

```java
@ThreadSafe
public class AtomicPseudoRandom extends PseudoRandom {
    private AtomicInteger seed;

    AtomicPseudoRandom(int seed) {
        this.seed = new AtomicInteger(seed);
    }

    public int nextInt(int n) {
        while (true) {
            int s = seed.get();
            int nextSeed = calculateNext(s);
            if (seed.compareAndSet(s, nextSeed)) {
                int remainder = s % n;
                return remainder > 0 ? remainder : remainder + n;
            }
        }
    }
}
```

Listing 14.5. Random number generator using `AtomicInteger`.

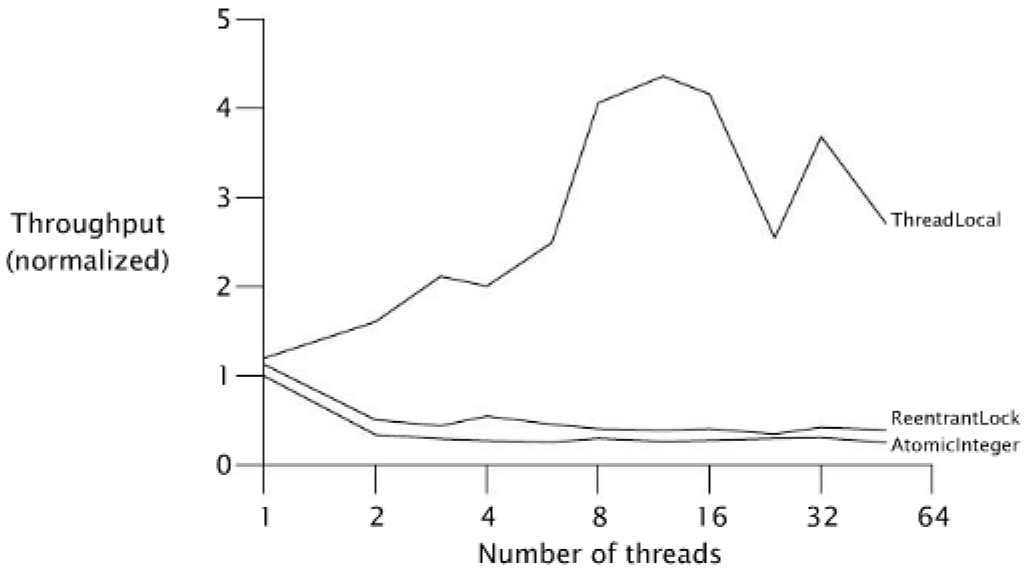


FIGURE 14.1. `Lock` and `AtomicInteger` performance under high contention.

tation, the lock or atomic variable experiences less contention since it is accessed less often by each thread.

As these graphs show, at high contention levels locking tends to outperform atomic variables, but at more realistic contention levels atomic variables outperform locks.[6] This is because a lock reacts to contention by suspending threads, reducing CPU usage and synchronization traffic on the shared memory bus. (This is similar to how blocking producers in a producer-consumer design reduces the load on consumers and thereby lets them catch up.) On the other hand, with atomic variables, contention management is pushed back to the calling class. Like most CAS-based algorithms, `AtomicPseudoRandom` reacts to contention by trying again immediately, which is usually the right approach but in a high-contention environment just creates more contention.

Before we condemn `AtomicPseudoRandom` as poorly written or atomic variables as a poor choice compared to locks, we should realize that the level of contention in Figure 14.1 is unrealistically high: no real program does nothing but contend for a lock or atomic variable. In practice, atomics tend to scale better than locks because atomics deal more effectively with typical contention levels.

The performance reversal between locks and atomics at differing levels of contention illustrates the strengths and weaknesses of each. With low to moderate contention, atomics offer better scalability; with high contention, locks offer better contention avoidance. (CAS-based algorithms also outperform lock-based ones on single-CPU systems, since a CAS always succeeds on a single-CPU system

6. The same holds true in other domains: traffic lights provide better throughput for high traffic but rotaries provide better throughput for low traffic; the contention scheme used by ethernet networks performs better at low traffic levels, but the token-passing scheme used by token ring networks does better with heavy traffic.

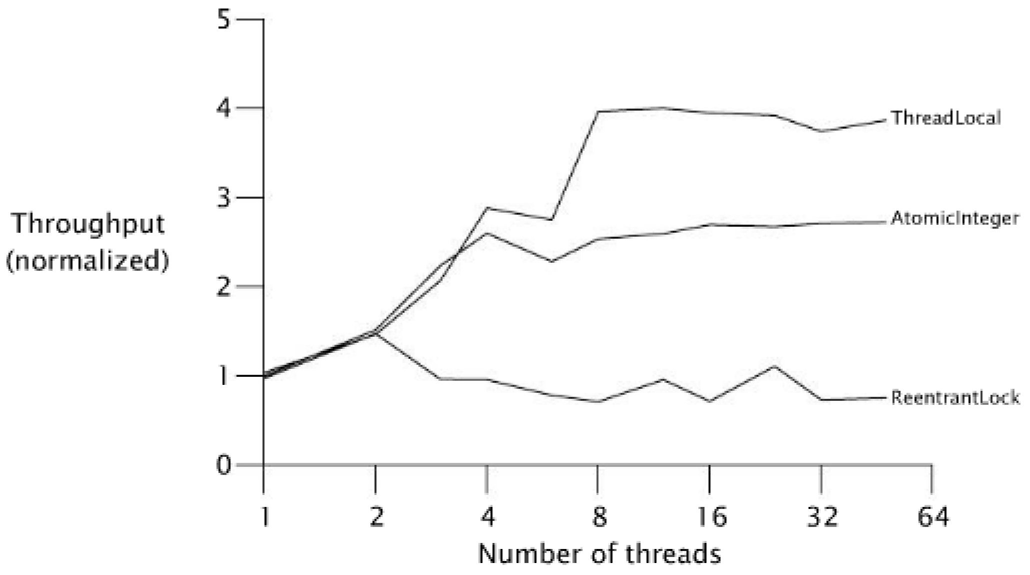


FIGURE 14.2. `Lock` and `AtomicInteger` performance under moderate contention.

except in the unlikely case that a thread is preempted in the middle of the read-modify-write operation.)

Figures 14.1 and 14.2 include a third curve; an implementation of `PseudoRandom` that uses a `ThreadLocal` for the PRNG state. This implementation approach changes the behavior of the class—each thread sees its own private sequence of pseudorandom numbers, instead of all threads sharing one sequence—but illustrates that it is often cheaper to not share state at all if it can be avoided. We can improve scalability by dealing more effectively with contention, but true scalability is achieved only by eliminating contention entirely.

14.4 Nonblocking algorithms

Lock-based algorithms are at risk for a number of liveness failures. If a thread holding a lock is delayed due to blocking I/O, page fault, or other delay, it is possible that no thread will make progress. An algorithm is called *nonblocking* if failure or suspension of any thread cannot cause failure or suspension of another thread; an algorithm is called *lock-free* if, at each step, *some* thread can make progress. Algorithms that use CAS exclusively for coordination between threads can, if constructed correctly, be both nonblocking and lock-free. An uncontended CAS always succeeds, and if multiple threads contend for a CAS, one always wins and therefore makes progress. Nonblocking algorithms are also immune to deadlock or priority inversion (though they can exhibit starvation or livelock because they can involve repeated retries). We've seen one nonblocking algorithm so far: `CasCounter`. Good nonblocking algorithms are known for many common data structures, including stacks, queues, priority queues, and hash tables—though designing new ones is a task best left to experts.

14.4.1 A nonblocking stack

Nonblocking algorithms are considerably more complicated than their lock-based equivalents. The key to creating nonblocking algorithms is figuring out how to limit the scope of atomic changes to a *single* variable while maintaining data consistency. In linked collection classes such as queues, you can sometimes get away with expressing state transformations as changes to individual links and using an `AtomicReference` to represent each link that must be updated atomically.

Stacks are the simplest linked data structure: each element refers to only one other element and each element is referred to by only one object reference. `ConcurrentStack` in Listing 14.6 shows how to construct a stack using atomic references. The stack is a linked list of `Node` elements, rooted at `top`, each of which contains a value and a link to the next element. The `push` method prepares a new link node whose `next` field refers to the current top of the stack, and then uses CAS to try to install it on the top of the stack. If the same node is still on the top of the stack as when we started, the CAS succeeds; if the top node has changed (because another thread has added or removed elements since we started), the CAS fails and `push` updates the new node based on the current stack state and tries again. In either case, the stack is still in a consistent state after the CAS.

`CasCounter` and `ConcurrentStack` illustrate characteristics of all nonblocking algorithms: some work is done speculatively and may have to be redone. In `ConcurrentStack`, when we construct the `Node` representing the new element, we are hoping that the value of the `next` reference will still be correct by the time it is installed on the stack, but are prepared to retry in the event of contention.

Nonblocking algorithms like `ConcurrentStack` derive their thread safety from the fact that, like locking, `compareAndSet` provides both atomicity and visibility guarantees. When a thread changes the state of the stack, it does so with a `compareAndSet`, which has the memory effects of a volatile write. When a thread examines the stack, it does so by calling `get` on the same `AtomicReference`, which has the memory effects of a volatile read. So any changes made by one thread are safely published to any other thread that examines the state of the list. And the list is modified with a `compareAndSet` that atomically either updates the `top` reference or fails if it detects interference from another thread.

14.4.2 A nonblocking linked list

The two nonblocking algorithms we've seen so far, the counter and the stack, illustrate the basic pattern of using CAS to update a value speculatively, retrying if the update fails. The trick to building nonblocking algorithms is to limit the scope of atomic changes to a single variable. With counters this is trivial, and with a stack it is straightforward enough, but for more complicated data structures such as queues, hash tables, or trees, it can get a lot trickier.

A linked queue is more complicated than a stack because it must support fast access to both the head and the tail. To do this, it maintains separate head and tail pointers. Two pointers refer to the node at the tail: the `next` pointer of the current

```
@ThreadSafe
public class ConcurrentStack <E> {
    AtomicReference<Node<E>> top = new AtomicReference<Node<E>>();

    public void push(E item) {
        Node<E> newHead = new Node<E>(item);
        Node<E> oldHead;
        do {
            oldHead = top.get();
            newHead.next = oldHead;
        } while (!top.compareAndSet(oldHead, newHead));
    }

    public E pop() {
        Node<E> oldHead;
        Node<E> newHead;
        do {
            oldHead = top.get();
            if (oldHead == null)
                return null;
            newHead = oldHead.next;
        } while (!top.compareAndSet(oldHead, newHead));
        return oldHead.item;
    }

    private static class Node <E> {
        public final E item;
        public Node<E> next;

        public Node(E item) {
            this.item = item;
        }
    }
}
```

ext Page

Listing 14.6. Nonblocking stack using Treiber's algorithm (Treiber, 1986).

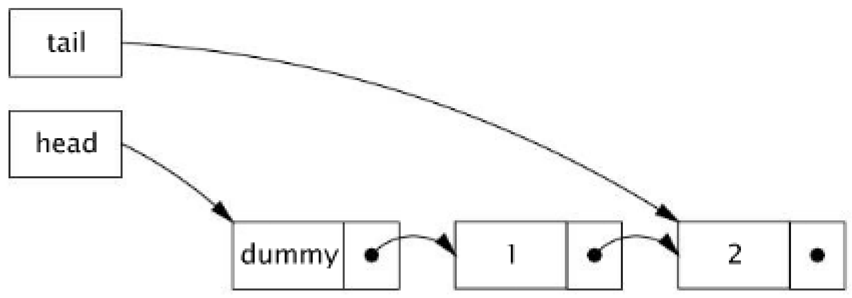


Figure 14.3. Queue with two elements in quiescent state.

last element, and the tail pointer. To insert a new element successfully, both of these pointers must be updated—atomically. At first glance, this cannot be done with atomic variables; separate CAS operations are required to update the two pointers, and if the first succeeds but the second one fails the queue is left in an inconsistent state. And, even if both operations succeed, another thread could try to access the queue between the first and the second. Building a nonblocking algorithm for a linked queue requires a plan for both these situations.

We need several tricks to develop this plan. The first is to ensure that the data structure is always in a consistent state, even in the middle of an multi-step update. That way, if thread *A* is in the middle of a update when thread *B* arrives on the scene, *B* can tell that an operation has been partially completed and knows not to try immediately to apply its own update. Then *B* can wait (by repeatedly examining the queue state) until *A* finishes, so that the two don't get in each other's way.

While this trick by itself would suffice to let threads "take turns" accessing the data structure without corrupting it, if one thread failed in the middle of an update, no thread would be able to access the queue at all. To make the algorithm nonblocking, we must ensure that the failure of a thread does not prevent other threads from making progress. Thus, the second trick is to make sure that if *B* arrives to find the data structure in the middle of an update by *A*, enough information is already embodied in the data structure for *B* to *finish the update for A*. If *B* "helps" *A* by finishing *A*'s operation, *B* can proceed with its own operation without waiting for *A*. When *A* gets around to finishing its operation, it will find that *B* already did the job for it.

`LinkedQueue` in Listing 14.7 shows the insertion portion of the Michael-Scott nonblocking linked-queue algorithm (Michael and Scott, 1996), which is used by `ConcurrentLinkedQueue`. As in many queue algorithms, an empty queue consists of a "sentinel" or "dummy" node, and the head and tail pointers are initialized to refer to the sentinel. The tail pointer always refers to the sentinel (if the queue is empty), the last element in the queue, or (in the case that an operation is in mid-update) the second-to-last element. Figure 14.3 illustrates a queue with two elements in the normal, or *quiescent*, state.

Inserting a new element involves updating two pointers. The first links the new node to the end of the list by updating the `next` pointer of the current last element; the second swings the tail pointer around to point to the new last ele-

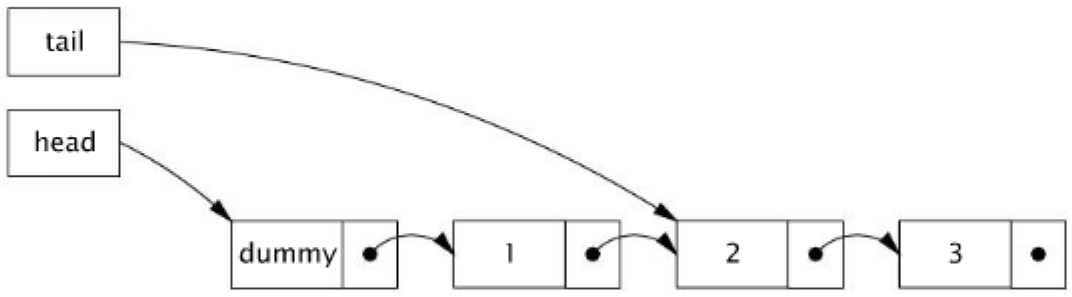


Figure 14.4. Queue in intermediate state during insertion.

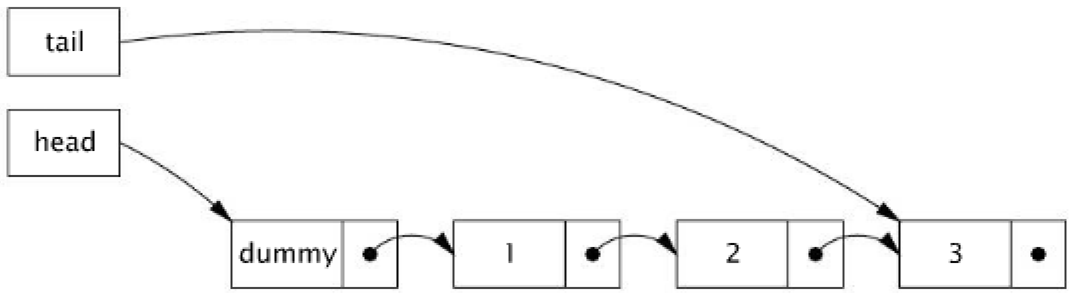


Figure 14.5. Queue again in quiescent state after insertion is complete.

ment. Between these two operations, the queue is in the *intermediate* state, shown in Figure 14.4. After the second update, the queue is again in the quiescent state, shown in Figure 14.5.

The key observation that enables both of the required tricks is that if the queue is in the quiescent state, the `next` field of the link node pointed to by `tail` is null, and if it is in the intermediate state, `tail.next` is non-null. So any thread can immediately tell the state of the queue by examining `tail.next`. Further, if the queue is in the intermediate state, it can be restored to the quiescent state by advancing the tail pointer forward one node, finishing the operation for whichever thread is in the middle of inserting an element.[7]

`LinkedQueue.put` first checks to see if the queue is in the intermediate state before attempting to insert a new element (step *A*). If it is, then some other thread is already in the process of inserting an element (between its steps *C* and *D*). Rather than wait for that thread to finish, the current thread helps it by finishing the operation for it, advancing the tail pointer (step *B*). It then repeats this check in case another thread has started inserting a new element, advancing the tail pointer until it finds the queue in the quiescent state so it can begin its own insertion.

ext Page

The CAS at step *C*, which links the new node at the tail of the queue, could fail if two threads try to insert an element at the same time. In that case, no harm is done: no changes have been made, and the current thread can just reload the tail pointer and try again. Once *C* succeeds, the insertion is considered to have

7. For a full account of the correctness of this algorithm, see (Michael and Scott, 1996) or (Herlihy and Shavit, 2006).

```
@ThreadSafe
public class LinkedQueue <E> {
    private static class Node <E> {
        final E item;
        final AtomicReference<Node<E>> next;

        public Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<Node<E>>(next);
        }
    }

    private final Node<E> dummy = new Node<E>(null, null);
    private final AtomicReference<Node<E>> head
            = new AtomicReference<Node<E>>(dummy);
    private final AtomicReference<Node<E>> tail
            = new AtomicReference<Node<E>>(dummy);

    public boolean put(E item) {
        Node<E> newNode = new Node<E>(item, null);
        while (true) {
            Node<E> curTail = tail.get();
            Node<E> tailNext = curTail.next.get();
            if (curTail == tail.get()) {
                if (tailNext != null) {                       🅐
                    // Queue in intermediate state, advance tail
                    tail.compareAndSet(curTail, tailNext);    🅑
                } else {
                    // In quiescent state, try inserting new node
                    if (curTail.next.compareAndSet(null, newNode)) { 🅒
                        // Insertion succeeded, try advancing tail
                        tail.compareAndSet(curTail, newNode);        🅓
                        return true;
                    }
                }
            }
        }
    }
}
```

Listing 14.7. Insertion in the Michael-Scott nonblocking queue algorithm (Michael and Scott, 1996).

taken effect; the second CAS (step *D*) is considered "cleanup", since it can be performed either by the inserting thread or by any other thread. If *D* fails, the inserting thread returns anyway rather than retrying the CAS, because no retry is needed—another thread has already finished the job in its step *B*! This works because before any thread tries to link a new node into the queue, it first checks to see if the queue needs cleaning up by checking if `tail.next` is non-null. If it is, it advances the tail pointer first (perhaps multiple times) until the queue is in the quiescent state.

14.4.3 Atomic field updaters

Listing 14.7 illustrates the algorithm used by `ConcurrentLinkedQueue`, but the actual implementation is a bit different. Instead of representing each `Node` with an atomic reference, `ConcurrentLinkedQueue` uses an ordinary volatile reference and updates it through the reflection-based `AtomicReferenceFieldUpdater`, as shown in Listing 14.8.

```
private class Node<E> {
    private final E item;
    private volatile Node<E> next;

    public Node(E item) {
        this.item = item;
    }
}

private static AtomicReferenceFieldUpdater<Node, Node> nextUpdater
        = AtomicReferenceFieldUpdater.newUpdater(
                Node.class, Node.class, "next");
```

Listing 14.8. Using atomic field updaters in `ConcurrentLinkedQueue`.

The atomic field updater classes (available in `Integer`, `Long`, and `Reference` versions) represent a reflection-based "view" of an existing volatile field so that CAS can be used on existing volatile fields. The updater classes have no constructors; to create one, you call the `newUpdater` factory method, specifying the class and field name. The field updater classes are not tied to a specific instance; one can be used to update the target field for any instance of the target class. The atomicity guarantees for the updater classes are weaker than for the regular atomic classes because you cannot guarantee that the underlying fields will not be modified directly—the `compareAndSet` and arithmetic methods guarantee atomicity only with respect to other threads using the atomic field updater methods.

In `ConcurrentLinkedQueue`, updates to the `next` field of a `Node` are applied using the `compareAndSet` method of `nextUpdater`. This somewhat circuitous approach is used entirely for performance reasons. For frequently allocated, short-lived objects like queue link nodes, eliminating the creation of an `AtomicReference` for each `Node` is significant enough to reduce the cost of insertion operations.

However, in nearly all situations, ordinary atomic variables perform just fine—in only a few cases will the atomic field updaters be needed. (The atomic field updaters are also useful when you want to perform atomic updates while preserving the serialized form of an existing class.)

14.4.4 The ABA problem

The *ABA problem* is an anomaly that can arise from the naive use of compare-and-swap in algorithms where nodes can be recycled (primarily in environments without garbage collection). A CAS effectively asks "Is the value of V still A?", and proceeds with the update if so. In most situations, including the examples presented in this chapter, this is entirely sufficient. However, sometimes we really want to ask "Has the value of V changed since I last observed it to be A?" For some algorithms, changing V from A to B and then back to A still counts as a change that requires us to retry some algorithmic step.

This *ABA problem* can arise in algorithms that do their own memory management for link node objects. In this case, that the head of a list still refers to a previously observed node is not enough to imply that the contents of the list have not changed. If you cannot avoid the ABA problem by letting the garbage collector manage link nodes for you, there is still a relatively simple solution: instead of updating the value of a reference, update a *pair* of values, a reference and a version number. Even if the value changes from A to B and back to A, the version numbers will be different. `AtomicStampedReference` (and its cousin `AtomicMarkableReference`) provide atomic conditional update on a pair of variables. `AtomicStampedReference` updates an object reference-integer pair, allowing "versioned" references that are immune[8] to the ABA problem. Similarly, `AtomicMarkableReference` updates an object reference-boolean pair that is used by some algorithms to let a node remain in a list while being marked as deleted.[9]

Summary

Nonblocking algorithms maintain thread safety by using low-level concurrency primitives such as compare-and-swap instead of locks. These low-level primitives are exposed through the atomic variable classes, which can also be used as "better volatile variables" providing atomic update operations for integers and object references.

Nonblocking algorithms are difficult to design and implement, but can offer better scalability under typical conditions and greater resistance to liveness failures. Many of the advances in concurrent performance from one JVM version to the next come from the use of nonblocking algorithms, both within the JVM and in the platform libraries.

8. In practice, anyway; theoretically the counter could wrap.

9. Many processors provide a double-wide CAS (CAS2 or CASX) operation that can operate on a pointer-integer pair, which would make this operation reasonably efficient. As of Java 6, `AtomicStampedReference` does not use double-wide CAS even on platforms that support it. (Double-wide CAS differs from DCAS, which operates on two unrelated memory locations; as of this writing, no current processor implements DCAS.)

Chapter 15

The Java Memory Model

Throughout this book, we've mostly avoided the low-level details of the Java Memory Model (JMM) and instead focused on higher-level design issues such as safe publication, specification of, and adherence to synchronization policies. These derive their safety from the JMM, and you may find it easier to use these mechanisms effectively when you understand *why* they work. This chapter pulls back the curtain to reveal the low-level requirements and guarantees of the Java Memory Model and the reasoning behind some of the higher-level design rules offered in this book.

15.1 What is a memory model, and why would I want one?

Suppose one thread assigns a value to `aVariable`:

```
aVariable = 3;
```

A memory model addresses the question "Under what conditions does a thread that reads `aVariable` see the value 3?" This may sound like a dumb question, but in the absence of synchronization, there are a number of reasons a thread might not immediately—or ever—see the results of an operation in another thread. Compilers may generate instructions in a different order than the "obvious" one suggested by the source code, or store variables in registers instead of in memory; processors may execute instructions in parallel or out of order; caches may vary the order in which writes to variables are committed to main memory; and values stored in processor-local caches may not be visible to other processors. These factors can prevent a thread from seeing the most up-to-date value for a variable and can cause memory actions in other threads to appear to happen out of order—if you don't use adequate synchronization.

In a single-threaded environment, all these tricks played on our program by the environment are hidden from us and have no effect other than to speed up execution. The Java Language Specification requires the JVM to maintain *within-thread as-if-serial semantics*: as long as the program has the same result as if it were executed in program order in a strictly sequential environment, all these games are permissible. And that's a good thing, too, because these rearrangements are responsible for much of the improvement in computing performance in recent years. Certainly higher clock rates have contributed to improved performance, but so has increased parallelism—pipelined superscalar execution units, dynamic instruction scheduling, speculative execution, and sophisticated multilevel memory caches. As processors have become more sophisticated, so too have compilers, rearranging instructions to facilitate optimal execution and using sophisticated global register-allocation algorithms. And as processor manufacturers transition to multicore processors, largely because clock rates are getting harder to increase economically, hardware parallelism will only increase.

In a multithreaded environment, the illusion of sequentiality cannot be maintained without significant performance cost. Since most of the time threads within a concurrent application are each "doing their own thing", excessive inter-thread coordination would only slow down the application to no real benefit. It is only when multiple threads share data that it is necessary to coordinate their activities, and the JVM relies on the program to identify when this is happening by using synchronization.

The JMM specifies the minimal guarantees the JVM must make about when writes to variables become visible to other threads. It was designed to balance the need for predictability and ease of program development with the realities of implementing high-performance JVMs on a wide range of popular processor architectures. Some aspects of the JMM may be disturbing at first if you are not familiar with the tricks used by modern processors and compilers to squeeze extra performance out of your program.

15.1.1 Platform memory models

In a shared-memory multiprocessor architecture, each processor has its own cache that is periodically reconciled with main memory. Processor architectures provide varying degrees of *cache coherence*; some provide minimal guarantees that allow different processors to see different values for the same memory location at virtually any time. The operating system, compiler, and runtime (and sometimes, the program, too) must make up the difference between what the hardware provides and what thread safety requires.

Ensuring that every processor knows what every other processor is doing at all times is expensive. Most of the time this information is not needed, so processors relax their memory-coherency guarantees to improve performance. An architecture's *memory model* tells programs what guarantees they can expect from the memory system, and specifies the special instructions required (called *memory barriers* or *fences*) to get the additional memory coordination guarantees required when sharing data. In order to shield the Java developer from the differences between memory models across architectures, Java provides its own memory model, and the JVM deals with the differences between the JMM and the underlying platform's memory model by inserting memory barriers at the appropriate places.

One convenient mental model for program execution is to imagine that there is a single order in which the operations happen in a program, regardless of what processor they execute on, and that each read of a variable will see the last write in the execution order to that variable by any processor. This happy, if unrealistic, model is called *sequential consistency*. Software developers often mistakenly assume sequential consistency, but no modern multiprocessor offers sequential consistency and the JMM does not either. The classic sequential computing model, the von Neumann model, is only a vague approximation of how modern multiprocessors behave.

The bottom line is that modern shared-memory multiprocessors (and compilers) can do some surprising things when data is shared across threads, unless you've told them not to through the use of memory barriers. Fortunately, Java programs need not specify the placement of memory barriers; they need only identify when shared state is being accessed, through the proper use of synchronization.

15.1.2 Reordering

In describing race conditions and atomicity failures in Chapter 2, we used interaction diagrams depicting "unlucky timing" where the scheduler interleaved operations so as to cause incorrect results in insufficiently synchronized programs. To make matters worse, the JMM can permit actions to appear to execute in different orders from the perspective of different threads, making reasoning about ordering in the absence of synchronization even more complicated. The various reasons why operations might be delayed or appear to execute out of order can all be grouped into the general category of *reordering*.

`PossibleReordering` in Listing 15.1 illustrates how difficult it is to reason about the behavior of even the simplest concurrent programs unless they are correctly synchronized. It is fairly easy to imagine how `PossibleReordering` could print $(1,0)$, or $(0,1)$, or $(1,1)$: thread A could run to completion before B starts, B could run to completion before A starts, or their actions could be interleaved. But, strangely, `PossibleReordering` can also print $(0,0)$! The actions in each thread have no dataflow dependence on each other, and accordingly can be executed out of order. (Even if they are executed in order, the timing by which caches are flushed to main memory can make it appear, from the perspective of B, that the assignments in A occurred in the opposite order.) Figure 15.1 shows a possible interleaving with reordering that results in printing $(0,0)$.

`PossibleReordering` is a trivial program, and it is still surprisingly tricky to enumerate its possible results. Reordering at the memory level can make programs behave unexpectedly. It is prohibitively difficult to reason about ordering in the absence of synchronization; it is much easier to ensure that your program uses synchronization appropriately. Synchronization inhibits the compiler, runtime, and hardware from reordering memory operations in ways that would violate the visibility guarantees provided by the JMM.[1]

15.1.3 The Java Memory Model in 500 words or less

The Java Memory Model is specified in terms of *actions*, which include reads and writes to variables, locks and unlocks of monitors, and starting and joining with

1. On most popular processor architectures, the memory model is strong enough that the performance cost of a volatile read is in line with that of a nonvolatile read.

```
public class PossibleReordering {
    static int x = 0, y = 0;
    static int a = 0, b = 0;

    public static void main(String[] args)
            throws InterruptedException {
        Thread one = new Thread(new Runnable() {
            public void run() {
                a = 1;
                x = b;
            }
        });
        Thread other = new Thread(new Runnable() {
            public void run() {
                b = 1;
                y = a;
            }
        });
        one.start(); other.start();
        one.join();  other.join();
        System.out.println("( "+ x + "," + y + ")");
    }
}
```

Listing 15.1. Insufficiently synchronized program that can have surprising results. *Don't do this.*

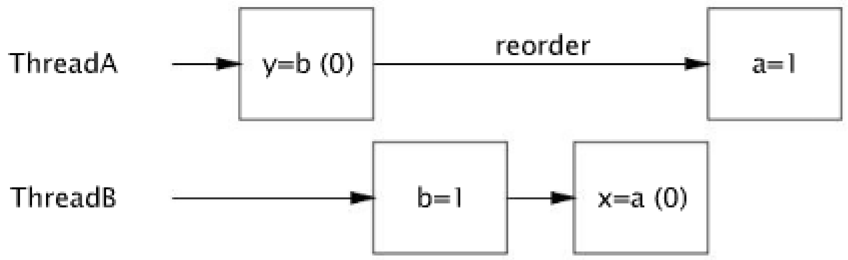


Figure 15.1. Interleaving showing reordering in `PossibleReordering`.

threads. The JMM defines a partial ordering[2] called *happens-before* on all actions within the program. To guarantee that the thread executing action *B* can see the results of action *A* (whether or not *A* and *B* occur in different threads), there must be a *happens-before* relationship between *A* and *B*. In the absence of a *happens-before* ordering between two operations, the JVM is free to reorder them as it pleases.

2. A partial ordering $\prec$ is a relation on a set that is antisymmetric, reflexive, and transitive, but for any two elements x and y, it need not be the case that $x \prec y$ or $y \prec x$. We use partial orderings every day to express preferences; we may prefer sushi to cheeseburgers and Mozart to Mahler, but we don't necessarily have a clear preference between cheeseburgers and Mozart.

A *data race* occurs when a variable is read by more than one thread, and written by at least one thread, but the reads and writes are not ordered by *happens-before*. A *correctly synchronized program* is one with no data races; correctly synchronized programs exhibit sequential consistency, meaning that all actions within the program appear to happen in a fixed, global order.

The rules for *happens-before* are:

Program order rule. Each action in a thread *happens-before* every action in that thread that comes later in the program order.

Monitor lock rule. An unlock on a monitor lock *happens-before* every subsequent lock on that same monitor lock.[3]

Volatile variable rule. A write to a volatile field *happens-before* every subsequent read of that same field.[4]

Thread start rule. A call to `Thread.start` on a thread *happens-before* every action in the started thread.

Thread termination rule. Any action in a thread *happens-before* any other thread detects that thread has terminated, either by successfully return from `Thread.join` or by `Thread.isAlive` returning `false`.

Interruption rule. A thread calling `interrupt` on another thread *happens-before* the interrupted thread detects the interrupt (either by having `InterruptedException` thrown, or invoking `isInterrupted` or `interrupted`).

Finalizer rule. The end of a constructor for an object *happens-before* the start of the finalizer for that object.

Transitivity. If *A happens-before B*, and *B happens-before C*, then *A happens-before C*.

Even though actions are only partially ordered, synchronization actions—lock acquisition and release, and reads and writes of `volatile` variables—are totally ordered. This makes it sensible to describe *happens-before* in terms of "subsequent" lock acquisitions and reads of `volatile` variables.

Figure 15.2 illustrates the *happens-before* relation when two threads synchronize using a common lock. All the actions within thread *A* are ordered by the program

3. Locks and unlocks on explicit `Lock` objects have the same memory semantics as intrinsic locks.
4. Reads and writes of atomic variables have the same memory semantics as volatile variables.

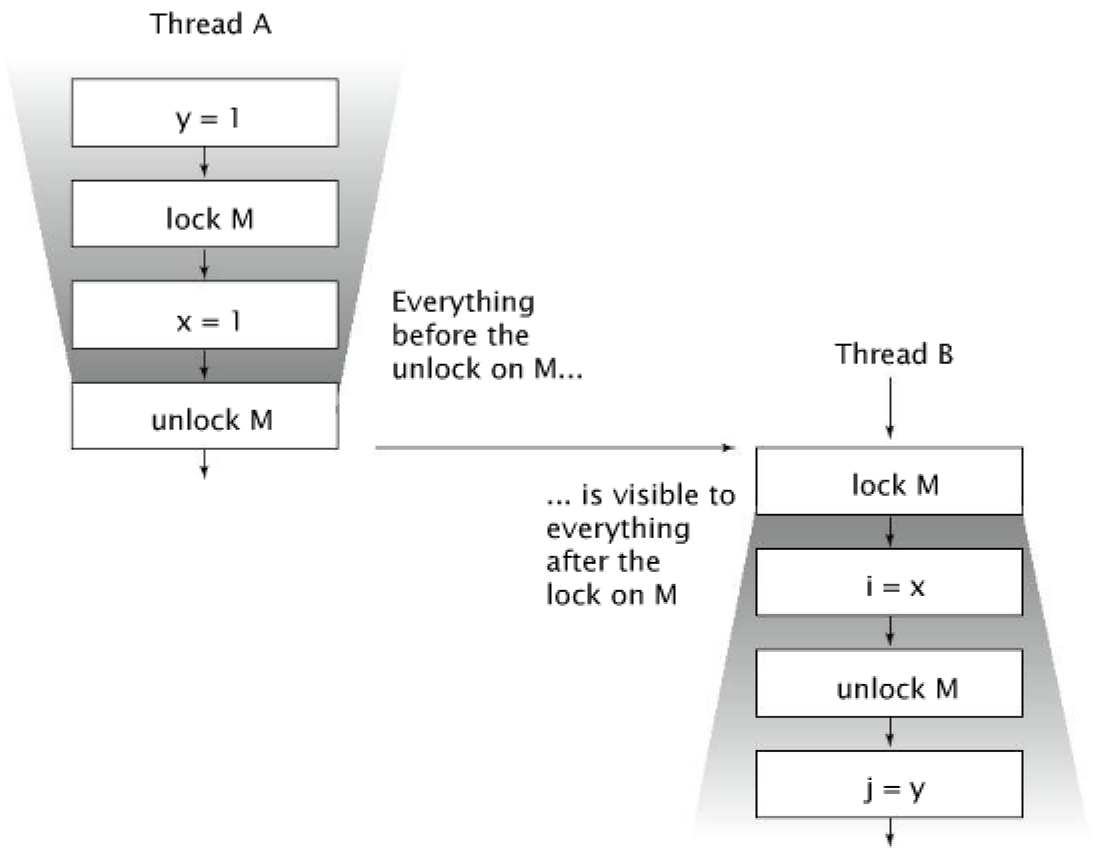

Thread A
y = 1
lock M
x = 1
unlock M
Everything before the unlock on M...
Thread B
lock M
... is visible to everything after the lock on M
i = x
unlock M
j = y

FIGURE 15.2. Illustration of *happens-before* in the Java Memory Model.

order rule, as are the actions within thread B. Because A releases lock M and B subsequently acquires M, all the actions in A before releasing the lock are therefore ordered before the actions in B after acquiring the lock. When two threads synchronize on *different* locks, we can't say anything about the ordering of actions between them—there is no *happens-before* relation between the actions in the two threads.

15.1.4 Piggybacking on synchronization

Because of the strength of the *happens-before* ordering, you can sometimes piggyback on the visibility properties of an existing synchronization. This entails combining the program order rule for *happens-before* with one of the other ordering rules (usually the monitor lock or volatile variable rule) to order accesses to a variable not otherwise guarded by a lock. This technique is very sensitive to the order in which statements occur and is therefore quite fragile; it is an advanced technique that should be reserved for squeezing the last drop of performance out of the most performance-critical classes like `ReentrantLock`.

The implementation of the protected `AbstractQueuedSynchronizer` methods in `FutureTask` illustrates piggybacking. AQS maintains an integer of synchronizer state that `FutureTask` uses to store the task state: running, completed, or cancelled. But FutureTask also maintains additional variables, such as the result of the computation. When one thread calls set to save the result and another thread calls get to retrieve it, the two had better be ordered by *happens-before*. This could be done by making the reference to the result volatile, but it is possible to exploit existing synchronization to achieve the same result at lower cost.

FutureTask is carefully crafted to ensure that a successful call to tryReleaseShared always *happens-before* a subsequent call to tryAcquireShared; tryReleaseShared always writes to a volatile variable that is read by tryAcquireShared. Listing 15.2 shows the innerSet and innerGet methods that are called when the result is saved or retrieved; since innerSet writes result before calling releaseShared (which calls tryReleaseShared) and innerGet reads result after calling acquireShared (which calls tryAcquireShared), the program order rule combines with the volatile variable rule to ensure that the write of result in innerGet *happens-before* the read of result in innerGet.

```
// Inner class of FutureTask
private final class Sync extends AbstractQueuedSynchronizer {
    private static final int RUNNING = 1, RAN = 2, CANCELLED = 4;
    private V result;
    private Exception exception;

    void innerSet(V v) {
        while (true) {
            int s = getState();
            if (ranOrCancelled(s))
                return;
            if (compareAndSetState(s, RAN))
                break;
        }
        result = v;
        releaseShared(0);
        done();
    }

    V innerGet() throws InterruptedException, ExecutionException {
        acquireSharedInterruptibly(0);
        if (getState() == CANCELLED)
            throw new CancellationException();
        if (exception != null)
            throw new ExecutionException(exception);
        return result;
    }
}
```

Listing 15.2. Inner class of FutureTask illustrating synchronization piggybacking.

We call this technique "piggybacking" because it uses an existing *happens-before* ordering that was created for some other reason to ensure the visibility of object *X*, rather than creating a *happens-before* ordering specifically for publishing *X*.

Piggybacking of the sort employed by `FutureTask` is quite fragile and should not be undertaken casually. However, in some cases piggybacking is perfectly reasonable, such as when a class commits to a *happens-before* ordering between methods as part of its specification. For example, safe publication using a `BlockingQueue` is a form of piggybacking. One thread putting an object on a queue and another thread subsequently retrieving it constitutes safe publication because there is guaranteed to be sufficient internal synchronization in a `BlockingQueue` implementation to ensure that the enqueue *happens-before* the dequeue.

Other *happens-before* orderings guaranteed by the class library include:

- Placing an item in a thread-safe collection *happens-before* another thread retrieves that item from the collection;
- Counting down on a `CountDownLatch` *happens-before* a thread returns from `await` on that latch;
- Releasing a permit to a `Semaphore` *happens-before* acquiring a permit from that same `Semaphore`;
- Actions taken by the task represented by a `Future` *happens-before* another thread successfully returns from `Future.get`;
- Submitting a `Runnable` or `Callable` to an `Executor` *happens-before* the task begins execution; and
- A thread arriving at a `CyclicBarrier` or `Exchanger` *happens-before* the other threads are released from that same barrier or exchange point. If `CyclicBarrier` uses a barrier action, arriving at the barrier *happens-before* the barrier action, which in turn *happens-before* threads are released from the barrier.

15.2 Publication

Chapter 3 explored how an object could be safely or improperly published. The safe publication techniques described there derive their safety from guarantees provided by the JMM; the risks of improper publication are consequences of the absence of a *happens-before* ordering between publishing a shared object and accessing it from another thread.

15.2.1 Unsafe publication

The possibility of reordering in the absence of a *happens-before* relationship explains why publishing an object without adequate synchronization can allow another thread to see a *partially constructed object* (see Section 3.5). Initializing a new object involves writing to variables—the new object's fields. Similarly, publishing a reference involves writing to another variable—the reference to the new object.

If you do not ensure that publishing the shared reference *happens-before* another thread loads that shared reference, then the write of the reference to the new object can be reordered (from the perspective of the thread consuming the object) with the writes to its fields. In that case, another thread could see an up-to-date value for the object reference but *out-of-date values for some or all of that object's state*—a partially constructed object.

Unsafe publication can happen as a result of an incorrect lazy initialization, as shown in Figure 15.3. At first glance, the only problem here seems to be the race condition described in Section 2.2.2. Under certain circumstances, such as when all instances of the `Resource` are identical, you might be willing to overlook these (along with the inefficiency of possibly creating the `Resource` more than once). Unfortunately, even if these defects are overlooked, `UnsafeLazyInitialization` is still not safe, because another thread could observe a reference to a partially constructed `Resource`.

```
@NotThreadSafe
public class UnsafeLazyInitialization {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null)
            resource = new Resource(); // unsafe publication
        return resource;
    }
}
```

Listing 15.3. Unsafe lazy initialization. *Don't do this.*

Suppose thread *A* is the first to invoke `getInstance`. It sees that `resource` is `null`, instantiates a new `Resource`, and sets `resource` to reference it. When thread *B* later calls `getInstance`, it might see that `resource` already has a non-null value and just use the already constructed `Resource`. This might look harmless at first, but *there is no happens-before ordering between the writing of* `resource` *in A and the reading of* `resource` *in B*. A data race has been used to publish the object, and therefore *B* is not guaranteed to see the correct state of the `Resource`.

The `Resource` constructor changes the fields of the freshly allocated `Resource` from their default values (written by the `Object` constructor) to their initial values. Since neither thread used synchronization, *B* could possibly see *A*'s actions in a different order than *A* performed them. So even though *A* initialized the `Resource` before setting `resource` to reference it, *B* could see the write to `resource` as occurring *before* the writes to the fields of the `Resource`. *B* could thus see a partially constructed `Resource` that may well be in an invalid state—and whose state may unexpectedly change later.

With the exception of immutable objects, it is not safe to use an object that has been initialized by another thread unless the publication *happens-before* the consuming thread uses it.

15.2.2 Safe publication

The safe-publication idioms described in Chapter 3 ensure that the published object is visible to other threads because they ensure the publication *happens-before* the consuming thread loads a reference to the published object. If thread *A* places *X* on a `BlockingQueue` (and no thread subsequently modifies it) and thread *B* retrieves it from the queue, *B* is guaranteed to see *X* as *A* left it. This is because the `BlockingQueue` implementations have sufficient internal synchronization to ensure that the `put` *happens-before* the `take`. Similarly, using a shared variable guarded by a lock or a shared volatile variable ensures that reads and writes of that variable are ordered by *happens-before*.

This *happens-before* guarantee is actually a stronger promise of visibility and ordering than made by safe publication. When *X* is safely published from *A* to *B*, the safe publication guarantees visibility of the state of *X*, but not of the state of other variables *A* may have touched. But if *A* putting *X* on a queue *happens-before* *B* fetches *X* from that queue, not only does *B* see *X* in the state that *A* left it (assuming that *X* has not been subsequently modified by *A* or anyone else), but *B* sees *everything* *A* did before the handoff (again, subject to the same caveat).[5]

Why did we focus so heavily on `@GuardedBy` and safe publication, when the JMM already provides us with the more powerful *happens-before*? Thinking in terms of handing off object ownership and publication fits better into most program designs than thinking in terms of visibility of individual memory writes. The *happens-before* ordering operates at the level of individual memory accesses—it is a sort of "concurrency assembly language". Safe publication operates at a level closer to that of your program's design.

15.2.3 Safe initialization idioms

It sometimes makes sense to defer initialization of objects that are expensive to initialize until they are actually needed, but we have seen how the misuse of lazy initialization can lead to trouble. `UnsafeLazyInitialization` can be fixed by making the `getResource` method `synchronized`, as shown in Listing 15.4. Because the code path through `getInstance` is fairly short (a test and a predicted branch), if `getInstance` is not called frequently by many threads, there is little enough contention for the `SafeLazyInitialization` lock that this approach offers adequate performance.

The treatment of static fields with initializers (or fields whose value is initialized in a static initialization block [JPL 2.2.1 and 2.5.3]) is somewhat special and

5. The JMM guarantees that *B* sees a value at least as up-to-date as the value that *A* wrote; subsequent writes may or may not be visible.

```
@ThreadSafe
public class SafeLazyInitialization {
    private static Resource resource;

    public synchronized static Resource getInstance() {
        if (resource == null)
            resource = new Resource();
        return resource;
    }
}
```

Listing 15.4. Thread-safe lazy initialization.

offers additional thread-safety guarantees. Static initializers are run by the JVM at class initialization time, after class loading but before the class is used by any thread. Because the JVM acquires a lock during initialization [JLS 12.4.2] and this lock is acquired by each thread at least once to ensure that the class has been loaded, memory writes made during static initialization are automatically visible to all threads. Thus statically initialized objects require no explicit synchronization either during construction or when being referenced. However, this applies only to the *as-constructed* state—if the object is mutable, synchronization is still required by both readers and writers to make subsequent modifications visible and to avoid data corruption.

```
@ThreadSafe
public class EagerInitialization {
    private static Resource resource = new Resource();

    public static Resource getResource() { return resource; }
}
```

Listing 15.5. Eager initialization.

Using eager initialization, shown in Listing 15.5, eliminates the synchronization cost incurred on each call to `getInstance` in `SafeLazyInitialization`. This technique can be combined with the JVM's lazy class loading to create a lazy initialization technique that does not require synchronization on the common code path. The *lazy initialization holder class* idiom [EJ Item 48] in Listing 15.6 uses a class whose only purpose is to initialize the `Resource`. The JVM defers initializing the `ResourceHolder` class until it is actually used [JLS 12.4.1], and because the `Resource` is initialized with a static initializer, no additional synchronization is needed. The first call to `getResource` by any thread causes `ResourceHolder` to be loaded and initialized, at which time the initialization of the `Resource` happens through the static initializer.

```
@ThreadSafe
public class ResourceFactory {
    private static class ResourceHolder {
        public static Resource resource = new Resource();
    }

    public static Resource getResource() {
        return ResourceHolder.resource;
    }
}
```

Listing 15.6. Lazy initialization holder class idiom.

15.2.4 Double-checked locking

No book on concurrency would be complete without a discussion of the infamous double-checked locking (DCL) antipattern, shown in Listing 15.7. In very early JVMs, synchronization, even uncontended synchronization, had a significant performance cost. As a result, many clever (or at least clever-looking) tricks were invented to reduce the impact of synchronization—some good, some bad, and some ugly. DCL falls into the "ugly" category.

Again, because the performance of early JVMs left something to be desired, lazy initialization was often used to avoid potentially unnecessary expensive operations or reduce application startup time. A properly written lazy initialization method requires synchronization. But at the time, synchronization was slow and, more importantly, not completely understood: the exclusion aspects were well enough understood, but the visibility aspects were not.

DCL purported to offer the best of both worlds—lazy initialization without paying the synchronization penalty on the common code path. The way it worked was first to check whether initialization was needed without synchronizing, and if the `resource` reference was not `null`, use it. Otherwise, synchronize and check again if the `Resource` is initialized, ensuring that only one thread actually initializes the shared `Resource`. The common code path—fetching a reference to an already constructed `Resource`—doesn't use synchronization. And that's where the problem is: as described in Section 15.2.1, it is possible for a thread to see a partially constructed `Resource`.

The real problem with DCL is the assumption that the worst thing that can happen when reading a shared object reference without synchronization is to erroneously see a stale value (in this case, `null`); in that case the DCL idiom compensates for this risk by trying again with the lock held. But the worst case is actually considerably worse—it is possible to see a current value of the reference but stale values for the object's state, meaning that the object could be seen to be in an invalid or incorrect state.

Subsequent changes in the JMM (Java 5.0 and later) have enabled DCL to work *if* `resource` is made `volatile`, and the performance impact of this is small since volatile reads are usually only slightly more expensive than nonvolatile reads.

```
@NotThreadSafe
public class DoubleCheckedLocking {
    private static Resource resource;

    public static Resource getInstance() {
        if (resource == null) {
            synchronized (DoubleCheckedLocking.class) {
                if (resource == null)
                    resource = new Resource();
            }
        }
        return resource;
    }
}
```

LISTING 15.7. Double-checked-locking antipattern. *Don't do this.*

However, this is an idiom whose utility has largely passed—the forces that motivated it (slow uncontended synchronization, slow JVM startup) are no longer in play, making it less effective as an optimization. The lazy initialization holder idiom offers the same benefits and is easier to understand.

15.3 Initialization safety

The guarantee of *initialization safety* allows properly constructed *immutable* objects to be safely shared across threads without synchronization, regardless of how they are published—even if published using a data race. (This means that `UnsafeLazyInitialization` is actually safe *if* `Resource` is immutable.)

Without initialization safety, supposedly immutable objects like `String` can appear to change their value if synchronization is not used by both the publishing and consuming threads. The security architecture relies on the immutability of `String`; the lack of initialization safety could create security vulnerabilities that allow malicious code to bypass security checks.

Initialization safety guarantees that for *properly constructed* objects, all threads will see the correct values of final fields that were set by the constructor, regardless of how the object is published. Further, any variables that can be *reached* through a final field of a properly constructed object (such as the elements of a final array or the contents of a `HashMap` referenced by a final field) are also guaranteed to be visible to other threads.[6]

6. This applies only to objects that are reachable *only* through `final` fields of the object under construction.

For objects with final fields, initialization safety prohibits reordering any part of construction with the initial load of a reference to that object. All writes to final fields made by the constructor, as well as to any variables reachable through those fields, become "frozen" when the constructor completes, and any thread that obtains a reference to that object is guaranteed to see a value that is at least as up to date as the frozen value. Writes that initialize variables reachable through final fields are not reordered with operations following the post-construction freeze.

Initialization safety means that `SafeStates` in Listing 15.8 could be safely published even through unsafe lazy initialization or stashing a reference to a `SafeStates` in a public static field with no synchronization, even though it uses no synchronization and relies on the non-thread-safe `HashSet`.

```
@ThreadSafe
public class SafeStates {
    private final Map<String, String> states;

    public SafeStates() {
        states = new HashMap<String, String>();
        states.put("alaska", "AK");
        states.put("alabama", "AL");
        ...
        states.put("wyoming", "WY");
    }

    public String getAbbreviation(String s) {
        return states.get(s);
    }
}
```

Listing 15.8. Initialization safety for immutable objects.

However, a number of small changes to `SafeStates` would take away its thread safety. If `states` were not final, or if any method other than the constructor modified its contents, initialization safety would not be strong enough to safely access `SafeStates` without synchronization. If `SafeStates` had other nonfinal fields, other threads might still see incorrect values of those fields. And allowing the object to escape during construction invalidates the initialization-safety guarantee.

Initialization safety makes visibility guarantees only for the values that are reachable through final fields as of the time the constructor finishes. For values reachable through nonfinal fields, or values that may change after construction, you must use synchronization to ensure visibility.

Summary

The Java Memory Model specifies when the actions of one thread on memory are guaranteed to be visible to another. The specifics involve ensuring that operations are ordered by a partial ordering called *happens-before,* which is specified at the level of individual memory and synchronization operations. In the absence of sufficient synchronization, some very strange things can happen when threads access shared data. However, the higher-level rules offered in Chapters 2 and 3, such as `@GuardedBy` and safe publication, can be used to ensure thread safety without resorting to the low-level details of *happens-before*.